Владимир Дронов

Windows 8: разработка Metro-приложений для мобильных устройств

Санкт-Петербург
«БХВ-Петербург»
2012

УДК 681.3.06
ББК 32.973.26-018.2
Д75

Дронов В. А.

Д75 Windows 8: разработка Metro-приложений для мобильных устройств. — СПб.: БХВ-Петербург, 2012. — 528 с.: ил. — (Профессиональное программирование)

ISBN 978-5-9775-0832-2

Книга посвящена разработке Metro-приложений — нового класса приложений, работающих под управлением платформы Metro, входящей в состав Windows 8. Описана разработка приложений, предназначенных для устройств с сенсорными экранами, в частности планшетных компьютеров. Рассказано о разработке приложений на языках HTML, CSS и JavaScript, широко применяемых в Web-программировании. Рассмотрены элементы управления и их использование, разметка интерфейса приложений, вывод графики и мультимедиа, работа с файлами, удаленными интернет-сервисами, флэш-дисками, встроенными фото- и видеокамерами. Показаны способы реализации обмена данными между приложениями, вывода информации на плитки меню **Пуск**, создания локализованных и платных приложений. Описан процесс публикации готовых приложений в магазине Windows Store.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Елена Васильева*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Марины Дамбиевой*

Подписано в печать 30.06.12.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 42,57.
Тираж 1200 экз. Заказ №
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9 линия, 12/28

ISBN 978-5-9775-0832-2

Оглавление

Введение

Индустрия информационных технологий пребывает в тревожном ожидании. Ведущие игроки рынка нервничают, рядовые пользователи роются в Интернете в поисках любой информации, не брезгуя и слухами, именитые аналитики не знают, что и сказать. Настоящее затишье перед бурей.

И только одна компания знает все, но предпочитает молчать, выдавая сведения о своем грядущем детище строго гомеопатическими дозами.

Что это за компания?

Microsoft.

А что это за продукт, которого так все ждут и даже заранее побаиваются?

Windows 8.

А если говорить точнее, входящая в ее состав платформа Metro.

Планшетный бум

Планшеты как разновидность персональных компьютеров существуют уже довольно давно, не менее пятнадцати лет. (Автор, проживающий в глубокой провинции, своими глазами видел планшет в руках у торгового экспедитора еще в 2004 году.) Но вплоть до 2010 года они имели исключительно нишевое применение, используясь, в основном, как носимые терминалы в крупных компаниях. Рядовые пользователи планшетами практически не пользовались.

Все изменилось в 2010 году. Именно тогда компания Apple выпустила свой планшет iPad, предназначенный именно для потребительского рынка. Успех был поистине оглушительным, а звучное слово "ай-пад" стало нарицательным названием любого планшета, и совсем не обязательно выпущенного Apple.

Планшетный бум имел несколько предпосылок. Во-первых, что очевидно, развитие микроэлектроники, позволившее поместить в тонкий компактный корпус весьма серьезную "начинку". Во-вторых, широкое распространение беспроводных сетей, позволявших подключаться к Интернету практически везде. В-третьих, всеобщая "мобилизация" пользователей, которые больше не хотели сидеть сиднем за гро-

моздкими настольными компьютерами. В-четвертых, планшеты от Apple оказались настолько модными и стильными, что все, буквально все захотели ими обладать.

Неудивительно, что другие компании пожелали отхватить кусок нарождающегося рынка. Компания Google спешно адаптировала свою "телефонную" систему Android для работы на планшетах; как говорят, получилось не очень удачно. Компания Research In Motion, производитель некогда популярных телефонов Blackberry, поступила так же, но добилась еще меньших успехов.

Многие ждали, что же предпримет Microsoft...

Windows + планшет = Metro

Ответным ходом компании-гиганта стал анонс новой версии операционной системы Windows, носящей номер 8, в числе важнейших ключевых особенностей которой была заявлена успешная работа, в том числе и на планшетных компьютерах.

Переработка традиционно "настольной" ОС для функционирования на планшетах оказалась весьма нетривиальной задачей. Потребовалось изменить очень многое...

- Прежде всего, переработке подвергся пользовательский интерфейс. Он был максимально адаптирован для устройств с сенсорными экранами (к которым и относятся планшеты).
- Далее, были заметно уменьшены аппаратные требования. Планшеты — устройства компактные, предназначенные для использования на ходу, и мощные процессоры, объемистая оперативная память и жесткие диски в них просто не помещаются.
- Наконец, была полностью заменена сама платформа, на которой работают пользовательские приложения.

Вот о последнем изменении в Windows следует поговорить подробнее.

Windows-приложения, предназначенные для планшетов, работают под управлением принципиально новой программной платформы, функционирующей параллельно программным интерфейсам, унаследованным от предыдущих версий Windows. Эта платформа носит название Metro, а приложения, написанные по ней, — Metro-приложений.

Основные возможности платформы Metro таковы:

- наличие всех средств для создания развитых прикладных приложений, работающих с текстом, графикой, мультимедийными данными, файлами, интернет-сервисами и встроенными в компьютер устройствами и датчиками;
- обеспечение высокой производительности приложений и исключительного уровня безопасности;
- разработка приложений с применением широкого спектра современных компьютерных технологий: HTML+CSS+JavaScript, XAML и DirectX;
- создание приложений, не требующих сложного обслуживания. Так, установить Metro-приложение на свой компьютер может даже неопытный пользователь.

Можно сказать, что Windows 8 вообще и платформа Metro в частности позаимствовали все достоинства конкурирующих систем — Apple iOS и Google Android, — не "прихватывая" заодно и их ключевые недостатки. Так, Windows 8 значительно более открыта, чем iOS, и позволяет применять более широкий набор средств разработки приложений, чем Android.

Эта книга посвящена разработке Metro-приложений с применением "связки" HTML+CSS+JavaScript. Мы рассмотрим основные возможности платформы Metro, которые пригодятся большинству разработчиков, изучим языки HTML, CSS и JavaScript и познакомимся со средствами разработки, предназначенными именно для создания Metro-приложений. Ну, и создадим несколько вполне функциональных приложений, которые могут стать отправной точкой для тех, кто захочет связать свою судьбу с программированием под платформу Metro.

Чем будем пользоваться

При написании этой книги автор использовал следующие программные продукты:

- операционную систему Windows 8. Точнее, ее предварительную редакцию, носящую наименование Windows 8 Customer Preview. Окончательная редакция этой системы на момент написания книги еще не была доступна;
- средство разработки Metro-приложений Microsoft Visual Studio 11 Express for Windows 8, точнее, ее бета-версию, т. к. окончательная редакция, опять же, на момент написания книги еще не существовала.

Оба этих продукта успешно функционировали в виртуальном окружении. (Автор пользовался диспетчером виртуальных машин Oracle VirtualBox 4.1.8, однако рекомендует пользоваться более новой его версией.)

Типографские соглашения

В книге будут часто приводиться форматы написания различных конструкций, применяемых в языках HTML, CSS и JavaScript. В них применяются особые типографские соглашения, которые мы сейчас изучим.

> ***Внимание!***
>
> Все эти типографские соглашения применяются автором только в форматах написания языковых конструкций HTML, CSS и JavaScript. В реальном программном коде они не имеют смысла.

В угловые скобки (<>) заключаются наименования различных значений, которые дополнительно выделяются курсивом. В реальный код, разумеется, должны быть подставлены реальные значения. Например:

```
WinJS.Utilities.addClass(<элемент>, <стилевой класс>)
```

Здесь вместо подстроки *<элемент>* должен быть подставлен реальный элемент интерфейса, а вместо подстроки *<стилевой класс>* — реальный стилевой класс.

В квадратные скобки ([]) заключаются необязательные фрагменты кода. Например:

```
<файл>.renameAsync(<новое имя>[, <что делать в случае конфликта имен>])
```

Здесь второй параметр метода renameAsync — *<что делать в случае конфликта имен>* — может присутствовать, а может и отсутствовать.

Символом вертикальной черты (|) разделяются фрагменты кода, из которых в данном месте должен присутствовать только один.

```
display: none|inline|block
```

Здесь в качестве значения атрибута стиля display должна присутствовать только одна из доступных строк: none, inline или block.

Слишком длинные, не помещающиеся на одной строке языковые конструкции автор разрывает на несколько строк и в местах разрывов ставит знаки ↳. Например:

```
file.deleteAsync(
↳Windows.Storage.StorageDeleteOption.tryMoveToRecycleBin).
↳then(function() { });
```

Приведенный код разбит на три строки, но должен быть набран в одну. Знаки ↳ при этом должны быть удалены.

> ***ЕЩЕ РАЗ ВНИМАНИЕ!***
>
> Все приведенные ранее типографские соглашения имеют смысл только в форматах написания конструкций языков HTML, CSS и JavaScript. В коде примеров используется только знак ↳.

Благодарности

Автор приносит благодарности своим родителям, знакомым и коллегам по работе.

- Белову Алексею Васильевичу, начальнику отдела ОИТ Волжского гуманитарного института (г. Волжский Волгоградской обл.), где работает автор, — за понимание и поддержку.
- Всем работникам отдела ОИТ — за понимание и поддержку.
- Родителям — за терпение, понимание и поддержку.
- Архангельскому Дмитрию Борисовичу — за дружеское участие.
- Шапошникову Игорю Владимировичу — за содействие.
- Рыбакову Евгению Евгеньевичу, заместителю главного редактора издательства "БХВ-Петербург", — за неоднократные побуждения к работе, без которых автор давно бы обленился.
- Издательству "БХВ-Петербург" — за издание моих книг.
- Разработчикам Windows 8 и платформы Metro — за замечательные программные продукты.
- Всем своим читателям и почитателям — за прекрасные отзывы о моих книгах.
- Всем, кого я забыл здесь перечислить, — за все хорошее.

ЧАСТЬ I

Основы Metro-программирования

ГЛАВА 1

Что такое платформа Metro

В последнее время компьютерный рынок претерпевает коренные изменения. Продажи традиционных ПК падают, и на их место приходят другие устройства — более мобильные и ориентированные на неподготовленного пользователя. Это планшеты, или, говоря иначе, мобильные устройства с большим сенсорным экраном.

Планшеты оккупируют полки компьютерных магазинов. Планшеты не сходят со страниц компьютерной прессы. Планшеты — это модно. Планшеты — это стильно. Планшеты — это фетиш нового времени.

Платформа Metro как средство завоевания рынка планшетов

Существуют три операционные системы, под управлением которых работают планшеты. Первая — Apple iOS, которую можно назвать старожилом рынка. Вторая — Android, появившаяся позже. Третья — Microsoft Windows 8, точнее, входящая в ее состав платформа Metro; напористый новичок, пока не столь популярный, как первые две системы, но уверенно сокращающий отставание.

Metro — программная платформа, предназначенная для создания и выполнения мобильных приложений. Она предназначена именно для планшетов и поддерживает все ключевые особенности этого класса устройств. Вероятно, это первая разработка Microsoft подобного рода, и совсем не похоже, чтобы первый блин вышел комом.

Армия владельцев Windows-планшетов велика, и потребности ее неизмеримы. Говоря проще, им требуется масса разнообразных приложений, работающих под управлением платформы Metro. И если рынок традиционных Windows-приложений уже давно перенасыщен, то рынок приложений под платформу Metro *(Metro-приложений)* пока еще очень и очень далек от насыщения.

Здесь раздолье для независимых разработчиков. Так что начинайте создавать Metro-приложения прямо сейчас! Вы потесните конкурентов, завоюете симпатии

пользователей и обзаведетесь массой поклонников. Вы успеете раньше всех, займете свою рыночную нишу и станете пионерами, легендами индустрии. Наконец, вы сможете заработать, продавая свои приложения, — и Microsoft поддержит вас в этом.

А начать собственно разработку Metro-приложений вам поможет эта книга.

Ключевые особенности планшетов и их поддержка платформой Metro

Но чем так хороша платформа Metro? Вообще, что она поддерживает?

Основные возможности Metro мы будем рассматривать вместе с ключевыми особенностями планшетов как класса мобильных устройств. Они неотделимы друг от друга.

Сенсорный экран

Первая ключевая особенность планшетов — разумеется, сенсорный экран. Более того, это обязательная составная часть любого подобного устройства; планшетов без сенсорного экрана просто не существует.

Понятно, что работа с планшетом выполняется путем манипуляций с сенсорным экраном. Пользователь тыкает пальцем в кнопочки, листает списки, перемещает регуляторы, прокручивает текст и выполняет все остальные действия, поддерживаемые приложением (их еще называют *жестами*). А если возникает необходимость набрать текст, он делает это с помощью экранной клавиатуры, опять же, пальцами.

На традиционных ПК для этой цели применяются не менее традиционные клавиатура и мышь. В чем-то они удобнее, в чем-то — нет, но самое главное — они имеют принципиальные отличия от сенсорного экрана.

С одной стороны, мышью можно указать в конкретное место экрана с максимальной точностью, буквально ткнуть ей в единичный пиксел. Это позволяет разработчикам приложений создавать очень компактные элементы интерфейса, скажем, служебные кнопки небольшого размера, узкие заголовки окон, крошечные регуляторы, находящиеся в строке статуса, и т. п. Пользователь в любом случае не промахнется.

В случае сенсорного экрана так сделать не получится. Человеческий палец имеет слишком большие размеры. Так что, если пользователь попытается нажать какую-либо маленькую кнопку, он либо не попадет по ней, либо заодно нажмет сразу все соседние кнопки. Понятно, что ничего хорошего из этого не получится.

С другой стороны, с помощью сенсорного экрана можно реализовать поддержку жестов, выполняемых сразу несколькими пальцами. Например, пользователь может приложить к экрану два пальца и раздвинуть их, чтобы увеличить размер изображения (кстати, этот жест реально применяется в планшетах Apple).

Мышью же в каждый конкретный момент можно указать только в одну точку экрана. Поэтому о "многопальцевых" жестах придется забыть.

Из этого следуют два важных вывода.

- Интерфейс мобильных приложений, предназначенных для планшетов, придется делать с учетом того, что с ними будут работать пальцами. Это значит, что элементы интерфейса должны иметь достаточно большие размеры, и промежутки между ними следует увеличить, чтобы пользователь случайно не нажал сразу две кнопки.
- В мобильных приложениях можно предусмотреть поддержку жестов, в том числе и "многопальцевых". Если, конечно, приложение от этого выиграет...

Платформа Metro изначально поддерживает и "пальцевый" ввод, и жесты. В большинстве случаев разработчикам даже не придется реализовывать это специально — за них все сделает сама эта платформа.

Однако Metro поддерживает и традиционные устройства ввода — клавиатуру и мышь. Так что пользователь сможет подключить к своему планшету USB-клавиатуру и набирать текст со всеми удобствами.

Ограниченные системные ресурсы

Аппаратные платформы, на основе которых создаются планшеты, не столь мощны, как платформы традиционных ПК. Центральные процессоры, применяемые в планшетах, имеют существенно меньшее быстродействие, и объем установленной в них оперативной памяти также невелик.

Далее, все без исключения планшеты питаются от аккумулятора. Этот аккумулятор имеет вполне конечную емкость и неприятную особенность разряжаться в самый "интересный" момент. Если же запитать планшет от сети, то он перестанет быть мобильным устройством и, следовательно, лишится своего главного преимущества по сравнению с традиционным ПК — мобильности.

Иначе говоря, системные ресурсы планшетов ограничены.

Вспомним, как работает пользователь традиционного ПК. Он запускает сразу несколько различных приложений — Web-обозреватель, клиенты электронной почты и системы мгновенных сообщений, текстовый процессор — и переключается между ними. Конечно, каждое запущенное приложение загружает центральный процессор и отнимает часть оперативной памяти, но, поскольку процессор имеет достаточное быстродействие и объем оперативной памяти весьма велик, производительность если и падает, то совсем неощутимо. А, раз компьютер питается от сети, пользователь не рискует внезапно оказаться наедине с сообщением "Ваш аккумулятор разряжен".

С планшетами так не разгуляешься... Как уже говорилось, центральный процессор планшета относительно слаб, и памяти в нем также мало. Следовательно, запустив несколько приложений, пользователь столкнется с существенным падением производительности. При этом, раз нагрузка на процессор и память возросла, аккумулятор будет разряжаться интенсивнее и, следовательно, истощится быстрее.

Выход из этого положения таков:

- все неактивные приложения принудительно приостанавливаются. При этом они продолжают отнимать оперативную память, но, по крайней мере, не будут загружать центральный процессор;
- что касается задач, которые должны выполняться даже будучи неактивными (например, фоновая загрузка файлов), то здесь возможны два сценария:
  - сама система может очертить круг задач, выполняемых в неактивном состоянии, и они будут выполняться, даже если запустившее их приложение неактивно и, следовательно, приостановлено;
  - приложение, которое должно выполняться в неактивном состоянии, может "попросить" систему не приостанавливать его. Естественно, пользователь должен быть в курсе;
- приложения, с которыми пользователь давно не работал, могут принудительно завершаться и выгружаться из памяти, чтобы освободить системные ресурсы.

Платформа Metro поступает именно так. Она приостанавливает неактивные приложения, выгружает из памяти те, что давно не использовались, и может выполнять определенные задачи даже в неактивном состоянии. В общем, делает все, чтобы продлить время "жизни" аккумулятора.

Ограниченный объем долговременной памяти

Традиционные ПК имеют вместительные жесткие диски. Настолько вместительные, что даже после установки нескольких десятков приложений, в том числе пары современных игр, и записи тысяч фотографий, сотен музыкальных альбомов и десятков фильмов на них все еще остается немало места.

В планшетах жестких дисков нет. В них применяется другой тип долговременной памяти — flash-память. Хоть она лучше подходит для мобильных устройств (более устойчива к механическим воздействиям и имеет меньшее энергопотребление, чем жесткие диски), ее удельная стоимость существенно выше.

Так что много долговременной памяти в планшеты не ставят.

Из чего следует, что количество приложений, которые могут быть установлены на планшет, крайне ограничено. (Ведь пользователь может занять память не только приложениями, но и теми же фотографиями, музыкой и фильмами.) И вполне может наступить момент, когда очередное жизненное необходимое приложение пользователь установить уже не сможет.

Есть два способа решить эту проблему: попросить пользователей умерить аппетиты (что маловероятно) или попытаться сделать приложения как можно более компактными (а это уже вполне реализуемо).

Компактными их можно сделать довольно просто — переложить максимум типичных задач на операционную систему. К таким задачам можно отнести, например, обработку новостей RSS, получение изображения с фото- или видеокамеры, загруз-

ку файлов, "общение" с магазином приложений (о нем — чуть позже) и т. п. В результате разработчику, чтобы реализовать, скажем, получение изображения со встроенной фотокамеры, достаточно будет написать всего одну команду, которая займет в коде готового приложения очень мало места.

Платформа Metro может похвастаться всеми перечисленными ранее "умениями". И уже в *главе 2*, создавая наше первое приложение для данной платформы, мы в этом убедимся.

Иная целевая группа пользователей

Традиционные ПК, что бы ни говорили, предназначены для более-менее подготовленного пользователя, который способен самостоятельно найти нужное ему приложение, установить его и удалить, если оно ему больше не нужно. А самое главное — пользователь должен быть в состоянии решать проблемы, которые могут быть вызваны конфликтами этого приложения с другими, уже установленными на компьютере, и с программами, входящими в состав операционной системы. И хорошо, если пользователь будет способен убирать постоянно появляющийся "мусор", в частности временные файлы, в изобилии оставляемые некорректно работающими приложениями.

Планшеты, как уже говорилось, рассчитаны на пользователей, не знакомых с компьютерами. Их можно рассматривать как бытовую технику, которая начинает нормально работать сразу после извлечения из коробки и, в идеале, без чтения инструкции.

Стало быть, операционные системы для планшетов должны удовлетворять следующим требованиям.

- Пользователь должен быстро и без проблем найти нужное ему приложение и при этом быть уверенным, что оно полностью функционально, не содержит ошибок и вредоносного кода.
- Обычно все приложения, предназначенные для планшетов, публикуются на особом Web-сайте — *магазине приложений*. Все приложения, предлагаемые к публикации в таком магазине, обязательно проходят проверку, по крайней мере, на функциональность и отсутствие вредоносного кода. При этом пользователь может установить на планшет только приложения, загруженные из магазина; установить приложение, полученное из другого источника, как правило, невозможно.
- Установка приложения должна быть максимально простой и не требовать вмешательства пользователя. Наилучший вариант — когда от пользователя требуется всего лишь нажать кнопку **Установить.**
- Устанавливаемые приложения (по крайней мере, прикладные) не должны вносить изменений в саму операционную систему.
- Приложения (по крайней мере, прикладные) не должны конфликтовать ни друг с другом, ни с операционной системой. Или, говоря на жаргоне профессиональ-

ных программистов, приложения должны быть максимально изолированы друг от друга.

- Обновление приложения должно выполняться максимально просто и прозрачно для пользователя. От пользователя должно требоваться лишь согласие на обновление.
- Удаление ненужного приложения должно быть максимально простым и не требовать никакого вмешательства пользователя, кроме нажатия кнопки **Удалить.**
- Удаление приложения должно сопровождаться удалением всех сохраненных им служебных данных: настроек, временных файлов и пр. Разумеется, созданные в приложении документы удаляться не должны!
- Если уж приложения "мусорят", и с этим ничего не поделаешь, система сама должна время от времени "подметать" за ними.

Платформа Metro все это делает. Она поддерживает магазин приложений, обеспечивает простую установку и надежное удаление приложений, надежно изолирует приложения друг от друга и убирает за ними "мусор". Так что пользователь-"чайник" будет доволен!

Другие особенности платформы Metro

А теперь рассмотрим прочие особенности платформы Metro, о которых нам следует знать.

- Начать следует с того, что Metro — не полноценная операционная система. Она представляет собой один из компонентов Windows 8, работающий совместно с другими ее компонентами и, вместе с тем, достаточно независимый от них.
- Вместе с тем, Metro не является очередной надстройкой над традиционными интерфейсами программирования Windows (к таким надстройкам относится, в частности, популярная платформа .NET). Ее можно рассматривать как совершенно независимый интерфейс программирования, работающий на том же "уровне", что и традиционный. Благодаря этому достигается высокое быстродействие мобильных приложений и скромные требования к системным ресурсам.
- Metro входит в состав только Windows 8. Выпуск отдельной редакции этой платформы для более старых версий Windows не планируется.
- Наряду с процессорной архитектурой Intel *x*86, на которой основаны процессоры традиционных ПК, Windows 8 впоследствии получит поддержку архитектуры ARM. (Вообще, это первая в истории версия Windows, которая будет ее поддерживать.)
- Для распространения Metro-приложений служит магазин приложений *Windows Store*. Этот магазин позволяет публиковать платные, условно-бесплатные и полностью бесплатные приложения (в том числе и с открытым исходным кодом), производить покупки платных приложений, выполнять обновления и даже распространять бета-версии приложений среди независимых тестеров.

- Приложения для платформы Metro могут выполняться не только на планшетах, но и на любых компьютерах, на которых установлена Windows 8, в том числе и на традиционных ПК.
- На планшете (и вообще, любом компьютере) с Windows 8 могут одновременно исполняться как приложения, написанные для платформы Metro, так и традиционные Windows-приложения. При этом последние будут выполняться как обычно, без приостановок и принудительного удаления из памяти. Так что у владельцев Windows-планшетов появляется возможность работать с Microsoft Office и играть в Call of Duty (разумеется, подключив клавиатуру и мышь).

Как видим, Metro по-своему уникальна и сильно выделяется среди других планшетных платформ. Это своего рода мост между вселенной традиционных ПК и миром планшетов, призванный всемерно облегчить переход между ними.

Достоинства и недостатки платформы Metro

Теперь перечислим достоинства платформы Metro и не забудем о ее недостатках. Начнем с достоинств.

- Значительно бóльшая открытость в сравнении с Apple iOS. Windows 8 может быть установлена на любой планшет от любого производителя.
- Возможность выполнения традиционных Windows-приложений. (Правда, это относится не к самой Metro, а к Windows 8, частью которой она является.)
- Возможность публикации в магазине Windows Store бесплатных приложений с открытым исходным кодом, которая может привлечь многих разработчиков и еще большее количество потребителей. (Для сравнения: Apple запрещает публикацию таких приложений в своем магазине.)
- Простота разработки Metro-приложений. Даже начинающий программист может создать вполне функциональное приложение буквально за пять минут.
- Дружелюбность к разработчикам. Microsoft предлагает для создания Metro-приложений на выбор целых три технологии, причем все они уже давно присутствуют на рынке и прекрасно обкатаны. (Эти технологии мы рассмотрим чуть позже.)
- Дружелюбность к производителям оборудования. Чтобы обеспечить поддержку платформой Metro какого-либо устройства, например фотокамеры, производителю достаточно выпустить для него драйвер, который подойдет к любому компьютеру, на котором установлена Windows 8, в том числе и любому планшету.
- В конце концов, это Windows — самая популярная на данный момент операционная система.

Существенных недостатков же автор у платформы Metro не нашел, сколь ни искал. Это говорит о том, что Microsoft проделала большую работу и создала весьма впечатляющую платформу.

Как создаются Metro-приложения

Как уже говорилось, для создания Metro-приложения Microsoft предлагает на выбор три технологии. Давайте их рассмотрим.

Первая технология известна всем, кто создавал Web-страницы. Она включает в себя язык разметки *HTML* (HyperText Markup Language, язык гипертекстовой разметки), каскадные таблицы стилей *CSS* (Cascading Style Sheet) и язык программирования *JavaScript*. Язык HTML используется для описания интерфейса Metro-приложений, таблицы стилей CSS — для его оформления, а язык JavaScript — для создания программной логики.

Достоинства первой технологии таковы:

- Простота освоения, в том числе и такими начинающими разработчиками, как мы. Языки HTML, CSS и JavaScript исключительно просты и, вместе с тем, позволяют сделать довольно многое.
- Широкая распространенность. HTML, CSS и JavaScript знают очень и очень многие; в конце концов, это традиционные интернет-технологии.
- Metro-приложения, созданные с применением этой технологии, будут успешно выполняться на процессорах, основанных на любой архитектуре, что поддерживается Windows 8, — Intel *x*86 и ARM.

Теперь перечислим недостатки первой технологии.

- Невысокое быстродействие созданных с ее помощью Metro-приложений.
- Несколько ограниченный набор возможностей по созданию интерфейса приложений.
- Любое, даже самое простое Metro-приложение, созданное с применением этой технологии, будет состоять из множества файлов. В некоторых случаях это может быть критично.

С применением HTML, CSS и JavaScript рекомендуется создавать только самые простые Metro-приложения, которым не требуется развитый интерфейс, а быстродействие не является критичным. Кроме того, эта технология — идеальный выбор для начинающих Metro-разработчиков.

Вторая технология знакома всем .NET-программистам. Это язык XAML (eXtensible Application Markup Language, расширяемый язык разметки приложений) и языки C#, C++ .NET и Visual Basic .NET. На языке XAML описывается интерфейс и оформление приложения, а языки C#, C++ .NET и Visual Basic .NET служат для создания его логики. Можно сказать, что эта технология является подмножеством платформы .NET.

Достоинств у нее побольше...

- Более высокое быстродействие готовых приложений.
- Бóльшие возможности в плане создания интерфейса приложений и его оформления.

- Подавляющее большинство Metro-приложений, созданных с помощью этой технологии, будет состоять всего из одного файла.
- Metro-приложения, созданные с помощью этой технологии, также будут успешно выполняться на любых процессорах — и Intel *x*86, и ARM.

Недостатки:

- не самое высокое быстродействие готовых приложений. Так, применять эту технологию для создания сложных трехмерных игр не рекомендуется;
- освоить эту технологию значительно сложнее, чем первую, особенно начинающим разработчикам.

С помощью данной технологии уже можно создавать более сложные приложения, имеющие более развитый интерфейс и достаточно критичные в плане быстродействия. Также это наилучший выбор для переноса созданных ранее .NET-приложений на платформу Metro.

Третья технология — выбор для самых квалифицированных разработчиков. Это язык C++ и технология DirectX. Интерфейс, оформление и логика приложения описываются на языке C++, а DirectX применяется для вывода на экран как самого интерфейса приложения, так и результатов его работы.

Достоинства этой технологии весьма существенны:

- максимально возможное быстродействие готовых приложений;
- богатейшие возможности по созданию интерфейса приложений;
- вполне возможно создать приложение, состоящее всего из одного файла (хотя это, скорее всего, будет очень простое приложение).

Недостатки, увы, тоже...

- Интерфейс и все его элементы придется создавать заново для каждого из приложений. Готовых элементов интерфейса данная технология не предоставляет.
- Необходимо создавать две редакции каждого приложения — для каждой из процессорных архитектур, поддерживаемых Windows 8 (Intel *x*86 и ARM). При этом редакция, предназначенная для процессоров Intel *x*86, не будет работать на процессорах ARM, и наоборот.
- Освоить эту технологию очень сложно. Как уже говорилось, она рассчитана на самых квалифицированных программистов.

Данная технология рекомендуется для создания сложных трехмерных игр или графических приложений. Применять ее для разработки приложений другого назначения нецелесообразно.

Поскольку мы только начинаем свой путь в Metro-программирование, давайте выберем первую из рассмотренных технологий — "связку" HTML, CSS и JavaScript. С одной стороны, она очень проста для изучения, а с другой, позволяет создавать весьма впечатляющие приложения.

И на этом закончим с теоретической частью, чтобы поскорее приступить к практике.

Что дальше?

В этой главе мы познакомились с платформой Metro, являющейся частью грядущей Windows 8 и предназначенной для создания и выполнения "планшетных" приложений. Мы выяснили, насколько полно она поддерживает ключевые особенности планшетов, перечислили ее преимущества и безуспешно попытались найти хотя бы один недостаток. Напоследок мы узнали, какие технологии применяются для создания приложений, предназначенных для этой платформы. В общем, чистая "беллетристика"...

В следующей главе мы начнем знакомиться со средствами разработки Metro-приложений, создадим наше первое, пока еще совсем простое Metro-приложение, а позднее усовершенствуем его. По ходу дела мы узнаем, что собой представляют и как работают Metro-приложения. Так что скучать нам не придется!

ГЛАВА 2

Знакомство с Visual Studio. Первое Metro-приложение

Предыдущая глава была чисто ознакомительной. Мы познакомились с платформой Metro, узнали, как она поддерживает все ключевые особенности планшетов, и поговорили о самой этой платформе.

А еще мы познакомились с тремя технологиями, которые могут применяться для создания Metro-приложений, и выбрали для себя первую технологию — "связку" языков HTML, CSS и JavaScript, как самую простую в изучении.

Что ж, инструменты выбраны. Начнем работу!

Средства разработки Metro-приложений

На данный момент Microsoft для создания Metro-приложений предлагает два средства разработки:

- *Microsoft Visual Studio 11 Express for Windows 8* — очередную версию своего флагманского пакета Visual Studio. Это мощный программный пакет, предназначенный, в основном, для программистов;
- *Microsoft Expression Blend 5* — очередную версию пакета для разработки интерфейсов приложений, который предназначен, в основном, для специалистов, занимающихся дизайном интерфейсов; к сожалению, для программистов этот пакет не очень удобен.

Для разработки первых Metro-приложений мы выберем первое средство разработки — Microsoft Visual Studio 11 Express for Windows 8. (В дальнейшем для краткости будем называть его *Visual Studio*.) Особо сложных интерфейсов мы создавать не будем, а специфические "программистские" инструменты этого программного пакета нам очень и очень помогут.

Знакомство с Visual Studio

Перед тем как приступать к написанию нашего первого Metro-приложения в Visual Studio, давайте хотя бы поверхностно рассмотрим интерфейс этого пакета.

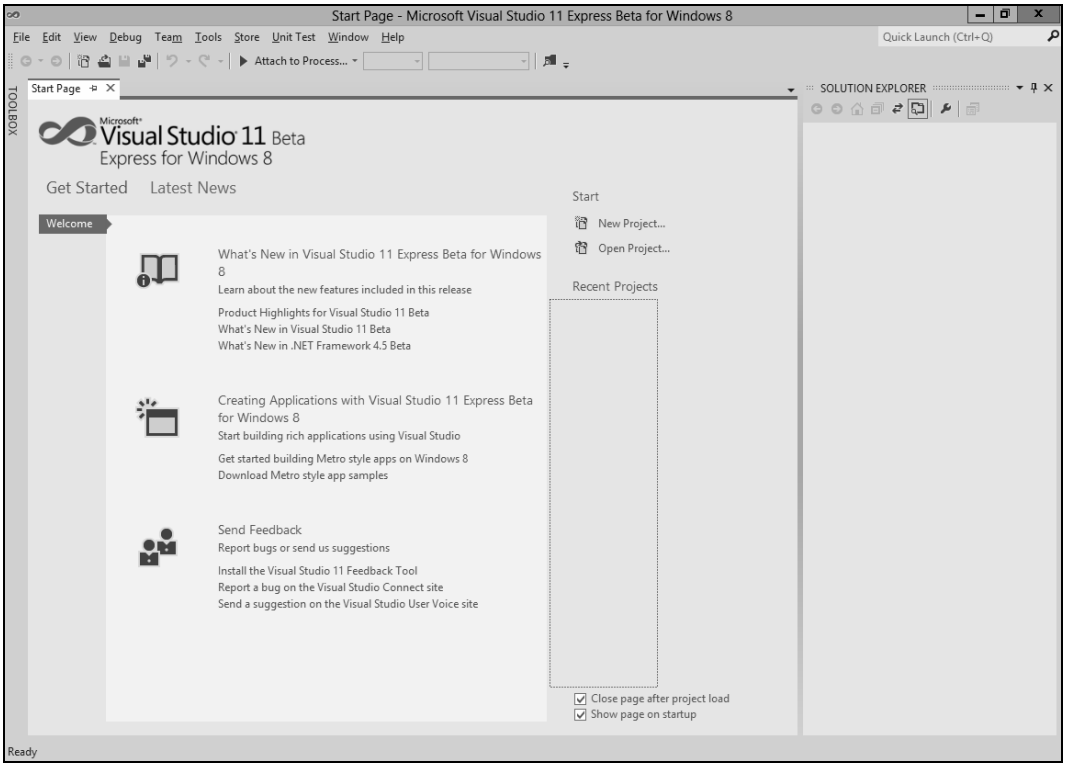


Рис. 2.1. Главное окно Visual Studio сразу после запуска пакета

Выйдем в меню **Пуск** (Start) Windows 8 и щелкнем на плитке[1] **Microsoft Visual Studio 11 Express for Windows 8**. Windows тотчас переключится на традиционный рабочий стол, и через некоторое время на экране появится главное окно Visual Studio (рис. 2.1).

> ***Внимание!***
>
> Сразу после первого запуска Visual Studio выведет окно-предупреждение, предлагающее установить лицензию разработчика Metro-приложений. Необходимо ответить положительно, нажав кнопку **I Agree**, т. к. без этой лицензии мы не сможем разрабатывать приложения такого типа. Впоследствии нам придется положительно ответить на появившееся на экране предупреждение системы UAC и выполнить вход на сервер лицензий под своим учетным именем, зарегистрированным в службе Microsoft Account.
>
> Установка лицензии разработчика Metro-приложений будет выполнена всего один раз. В дальнейшем выполнять это действие нам больше не придется.

Главное окно и его элементы

Вдоль верхнего края главного окна тянется *главное меню*, из которого доступны все команды, что поддерживаются пакетом. Чуть ниже находится *панель инструмен-*

[1] Также эти значки называют *тайлами* от англ. *tile*. — *Ред.*

тов — длинная и узкая полоса с кнопками; такие панели инструментов позволяют получить доступ к наиболее часто используемым командам. Подробно рассматривать все это мы не будем — и главное меню, и панели инструментов неоднократно встречались нам в других программах.

Лучше обратим внимание на правую часть главного окна. Там мы видим этакое "окошко", находящееся прямо в главном окне. Сходство с полноценным окном усиливается из-за наличия заголовка с названием и кнопками закрытия и сворачивания и даже небольшой панели инструментов. Что это такое?

Это одна из *панелей* Visual Studio. Такие панели обычно содержат всевозможные списки, например список файлов, из которых состоит разрабатываемое приложение. Эти списки, разумеется, важны, чтобы постоянно держать их перед глазами, но не настолько жизненно необходимы, чтобы отводить под них клиентскую область главного окна (о ней — чуть позже).

Панели можно перемещать с места на место, буксируя их мышью за заголовок. При этом если переместить панель к левому или правому краю главного окна, она автоматически пристыкуется к нему. Так, на рис. 2.1 изображена панель, пристыкованная к правому краю главного окна.

Пристыкованные панели могут накладываться друг на друга. В этом случае в нижней их части появится панель вкладок, перечисляющая накладывающиеся панели; чтобы переключиться на нужную панель, достаточно будет щелкнуть на вкладке, где написано название этой панели.

Если же переместить панель в какое-либо место экрана, достаточно удаленное от краев главного окна, панель станет плавающей. Плавающая панель отображается в собственном окне, независимом от главного окна пакета.

Наконец, ненужную панель можно вообще закрыть и тем самым убрать ее с экрана. Для этого следует щелкнуть на кнопке закрытия, расположенной в правой части ее заголовка и имеющей вид крестика.

А чтобы снова вывести на экран закрытую панель, придется воспользоваться главным меню. Далее в этой книге мы рассмотрим пункты главного меню, предназначенные для вывода всех нужных нам панелей.

В нижней части главного окна находится *строка статуса* — узкая серая полоса, на которой выводится различная служебная информация. Она нам также знакома по другим программам.

Остальная часть главного окна, не занятая главным меню, панелями инструментов, обычными панелями и строкой статуса, называется *клиентской областью*. В этой области будут выводиться окна документов, в которых вводится содержимое открытых файлов, и эти окна мы рассмотрим позже.

А сейчас в клиентской области отображается так называемая *стартовая страница* Visual Studio. Она содержит, в основном, краткую справочную информацию по данному пакету. Как только мы откроем хоть один файл, стартовая страница пропадет.

Создание нового приложения

Но хватит любоваться на главное окно Visual Studio! Все равно ничего особо интересного там пока нет.

Давайте лучше создадим новое Metro-приложение, пока еще "пустое", не содержащее ни интерфейса, ни оформления, ни логики. (Здесь, вообще-то, следовало бы написать "создадим новый проект приложения", но, поскольку мы еще не знаем, что такое проект, пусть все остается как есть.)

Пусть это будет приложение видеопроигрывателя. В конце концов, просмотр фильмов — одно из самых любимых занятий владельцев планшетов...

Проще всего создать новое приложение, щелкнув по гиперссылке **New Project**, расположенной в правом верхнем углу стартовой страницы. Также можно выбрать пункт **New Project** меню **File** или нажать комбинацию клавиш <Ctrl>+<Shift>+<N>.

После любого из этих действий на экране появится большое диалоговое окно **New Project** (рис. 2.2), где указываются основные параметры создаваемого приложения.

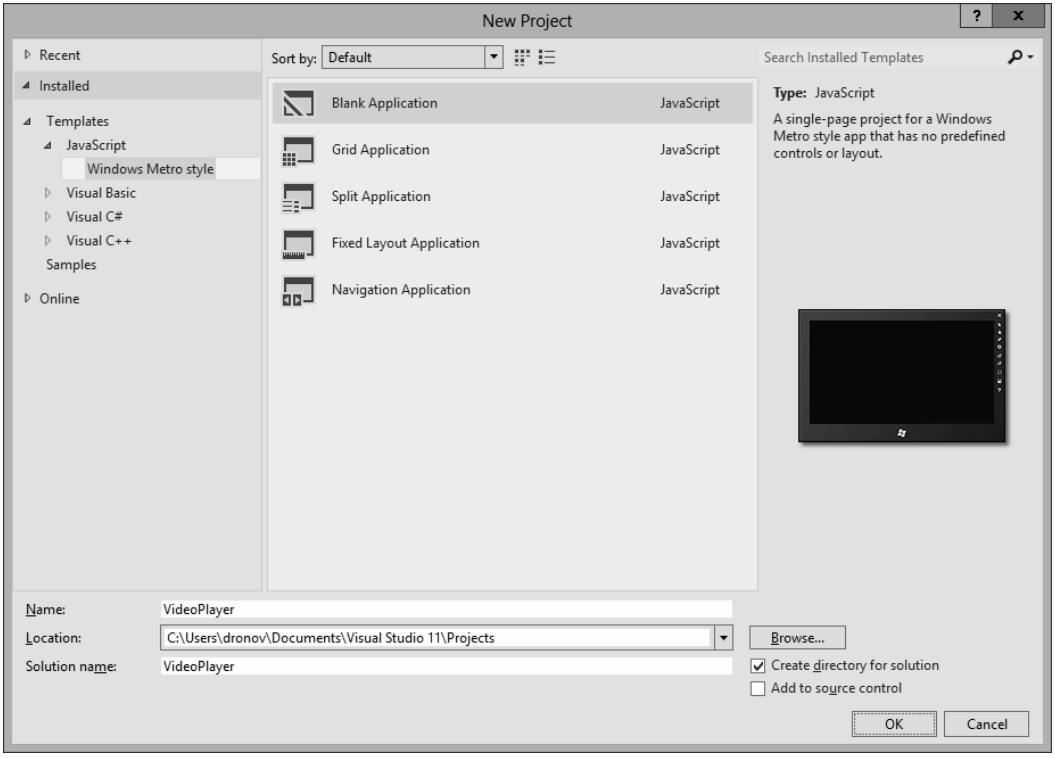


Рис. 2.2. Диалоговое окно **New Project**

Первое, что нам следует здесь указать, — технологию, с помощью которой мы собираемся создавать приложение. Как мы решили в *главе 1*, будем использовать "связку" HTML, CSS и JavaScript.

Посмотрим на левую часть окна **New Project**. Там находится большой иерархический список с тремя категориями. Переключимся на категорию **Installed**, содержащую перечень уже установленных в составе Visual Studio шаблонов (своего рода заготовок) приложений. Последовательно развернем "ветви" **Templates** (Шаблоны) и **JavaScript** (сокращенное наименование нужной нам технологии) и выберем единственный находящийся в последней "ветви" пункт **Windows Metro style** (Windows-приложение для платформы Metro).

Далее укажем тип создаваемого приложения. Мы хотим создать "пустое" приложение, не содержащее ни интерфейса, ни оформления, ни логики. Поэтому выберем в среднем списке пункт **Blank Application** ("Пустое" приложение).

Последнее наше действие — указание имени создаваемого приложения. Оно задается в поле ввода **Name**, расположенном в нижней части окна. Давайте дадим нашему приложению незатейливое имя `VideoPlayer`.

Вот, в принципе, и все. Нажмем кнопку **OK**, чтобы запустить процесс создания нового приложения. (Кнопка **Cancel** позволит нам отказаться от этого.)

Новое приложение будет создаваться довольно долго, и нам придется какое-то время подождать. А как только в клиентской области главного окна появится окно документа, в котором будет представлен код приложения, а панель **SOLUTION EXPLORER** перечислит все содержимое созданного нами проекта, мы можем приступать к работе.

Стоп! А что такое проект и окно документа? Давайте выясним.

Проект и решение

С понятиями проекта и решения мы будем постоянно сталкиваться в дальнейшем. Так что сейчас самое время дать им четкое определение.

Проект

Любое, даже самое простое Metro-приложение состоит из нескольких файлов разных типов. Сюда входят, прежде всего, файл, описывающий интерфейс приложения, файлы, содержащие описание его оформления, файлы с программной логикой, файлы графических изображений, что используются в приложении, и различные служебные файлы, хранящие разнообразную дополнительную информацию о приложении.

Поэтому возникает потребность объединить все файлы, составляющие приложение, в некую сущность. Это нужно, прежде всего, Visual Studio, чтобы "знать", из каких файлов формировать как отладочную версию приложения, так и пакет, содержащий его окончательную версию.

Такая сущность, объединяющая все файлы, что формируют приложение, и называется *проектом*.

Для организации файлов, входящих в проект, часто используются папки. Так, файлы с графическими изображениями помещаются в одну папку, файлы с оформле-

нием — в другую, служебные файлы — в третью и т. д. Причем такой способ организации файлов поддерживается самим Visual Studio; как мы узнаем чуть позже, при создании нового проекта он уже распределит входящие в него файлы по папкам.

Помимо файлов, проект хранит также параметры самого приложения. К таким параметрам относятся имя и изображение, отображаемые на плитке меню **Start**, список прав, которыми должно обладать приложение (например, право на загрузку файлов из Интернета или право на доступ к подключенной фотокамере), и другие сведения, подробнее о которых мы поговорим потом. Параметры приложения хранятся в отдельном файле, который, в свою очередь, также входит в состав проекта.

Теперь выясним, как организуется проект и его содержимое на диске.

Прежде всего, все содержимое проекта — файлы и папки — помещается в отдельной папке *(папке проекта)*. Эта папка создается при создании самого проекта и имеет то же имя, что мы указали для приложения в поле ввода **Name** окна **New Project** (см. рис. 2.2). Например, папка с содержимым только что созданного нами проекта будет иметь имя VideoPlayer.

В этой же папке находится и файл, хранящий список файлов, что входят в проект *(состав проекта)*, и некоторые служебные параметры проекта *(файл проекта)*. Этот файл имеет то же имя, что мы указали для создаваемого приложения, и расширение wwaproj. Так, файл созданного нами проекта получит имя VideoPlayer.jsproj.

Но зачем дополнительно хранить состав проекта в специальном файле? Ведь, по идее, все, что хранится в папке проекта, и является его содержимым. Все дело в том, что в процессе работы над приложением Visual Studio сохраняет в папке проекта различные служебные данные, которые нужны только ему и не требуются для нормальной работы приложения. Понятно, что эти данные не следует включать в отладочную и, тем более, окончательную версию приложения — они лишь будут занимать там место.

Если бы содержимое проекта дополнительно не хранилось в файле проекта, Visual Studio просто сформировал бы приложение на основе всех файлов, что хранятся в папке проекта, включая и ненужные служебные данные. А так он сможет "узнать", какие файлы действительно необходимы для успешной работы приложения, и включить в его состав только то, что нужно.

Решение

Часто приходится разрабатывать приложения, которые работают совместно. В случае платформы Metro это могут быть, например, приложения, обменивающиеся данными (подробнее об этом будет рассказано в *главе 19*).

Предположим, что мы разрабатываем Metro-приложение, которое предоставляет какие-либо данные другим приложениям. В этом случае нам понадобится проверить, как работает обмен данными. Самый простой способ сделать это — создать еще одно тестовое Metro-приложение, которое будет принимать эти данные и выводить их на экран. Так, кстати, и делают.

Но тут возникают две проблемы. Во-первых, проекты обоих этих приложений удобнее держать открытыми, чтобы в случае возникновения ошибки быстро ее исправить. Во-вторых, в любом случае в процессе отладки нам придется запускать оба этих приложения одновременно, и будет лучше, если это сделает сам Visual Studio.

Однако все средства разработки позволяют держать открытым только один проект; если же открыть другой проект, открытый ранее будет автоматически закрыт. Можно, конечно, запустить сразу две копии среды разработки, но работать с ними все равно будет неудобно.

Напрашивается следующий выход — разумно объединить оба проекта в некую сущность более высокого порядка. Или, если пользоваться терминологией Visual Studio, — *решение* (solution).

Решение может включать в себя сколько угодно проектов. Как только мы укажем Visual Studio начать процесс отладки, на основе всех этих проектов будут сформированы и подготовлены к запуску приложения. Но запущено будет только одно из них — то, чей проект мы пометили как *запускаемый*.

Для каждого нового проекта, создаваемого в Visual Studio, автоматически формируется новое решение, которое включает в себя вновь созданный проект. Имя этого решения задается в поле ввода **Solution name** диалогового окна **New Project** (см. рис. 2.2) и по умолчанию совпадает с именем создаваемого проекта. (Например, для нашего проекта `VideoPlayer` будет создано решение `VideoPlayer`.) Так что специально создавать решение нам не придется.

Как и в случае проекта, для каждого решения создается *папка решения*, в которой находятся все входящие в него проекты *(состав решения)*. Имя этой папки совпадает с указанным нами именем решения. Так, в нашем случае эта папка будет иметь имя VideoPlayer.

Дополнительно состав решения хранится в особом *файле решения*, который имеет имя, также совпадающее с именем решения, и расширение sln и хранится в папке решения. Например, состав нашего решения будет храниться в файле VideoPlayer.sln.

Панель *SOLUTION EXPLORER*

Но как нам узнать, что входит в состав проекта или решения? Нам ведь с ними еще работать и работать...

Для этого служит панель **SOLUTION EXPLORER** (рис. 2.3). По умолчанию она пристыкована к правому краю главного окна Visual Studio. Если же ее почему-то там нет, вывести ее можно выбором пункта **Solution Explorer** меню **View**.

Кстати, это необычный пункт меню. Как только мы его выберем, левее его названия появится галочка. Она обозначает, что функция, за которую "отвечает" пункт, активизирована — панель **SOLUTION EXPLORER** выведена на экран. Если же теперь выбрать данный пункт еще раз, галочка пропадет; это значит, что функция стала неактивной — упомянутая ранее панель закрыта. Такие пункты меню носят

название *пунктов-выключателей*; они будут встречаться нам в дальнейшем довольно часто.

Но вернемся к панели **SOLUTION EXPLORER**. В ней отображается иерархический список, представляющий все содержимое открытого решения. Этот список похож на тот, что выводится в левой панели Проводника и работает точно так же.

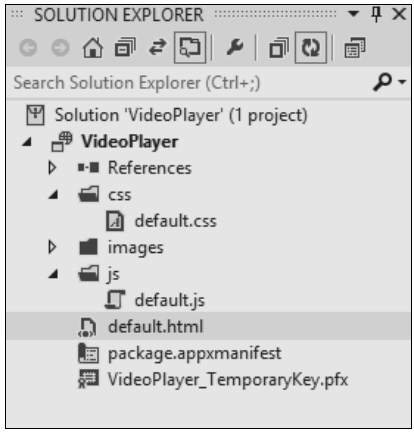


Рис. 2.3. Панель **SOLUTION EXPLORER**

Прежде всего, обратим внимание на пункт с названием **Solution '*<название решения>*' (*<количество проектов в решении>* project[s])**; в нашем случае он будет иметь название **Solution 'VideoPlayer' (1 project)**. Это открытое в Visual Studio решение, и находится оно на нулевом уровне вложенности (т. е. в самом верху иерархии пунктов этого списка).

В него вложены пункты, чьи названия совпадают с именами входящих в решение проектов. В нашем случае такой пункт будет всего один — **VideoPlayer**.

А из пунктов, представляющих отдельные проекты, "растут" целые "ветви". Они представляют папки и файлы, составляющие данный проект. Так, в нашем случае "ветвь" **VideoPlayer** будет содержать еще четыре "ветви" более низкого уровня вложенности и три пункта. Что означает, что в наш проект входят три файла и три папки, в свою очередь, содержащие другие файлы.

На этом пока закончим. Ведь мы еще не рассмотрели клиентскую область, в которой сейчас выводится что-то очень интересное...

Окна документов

Это *окна документов* (рис. 2.4). Они служат для вывода содержимого открытых в Visual Studio файлов.

Окна документов всегда выводятся в клиентской области главного окна. Вынести их за ее пределы невозможно.

В каждом окне документа выводится содержимое только одного файла. Это позволяет, с одной стороны, сосредоточиться на работе с нужным файлом, а с другой, при необходимости легко переключиться на другой файл.

Окна документов выводятся развернутыми; при этом они занимают всю клиентскую область. Если открыть сразу несколько окон документов, они будут накладываться друг на друга (совсем как панели, пристыкованные к одному краю главного окна).

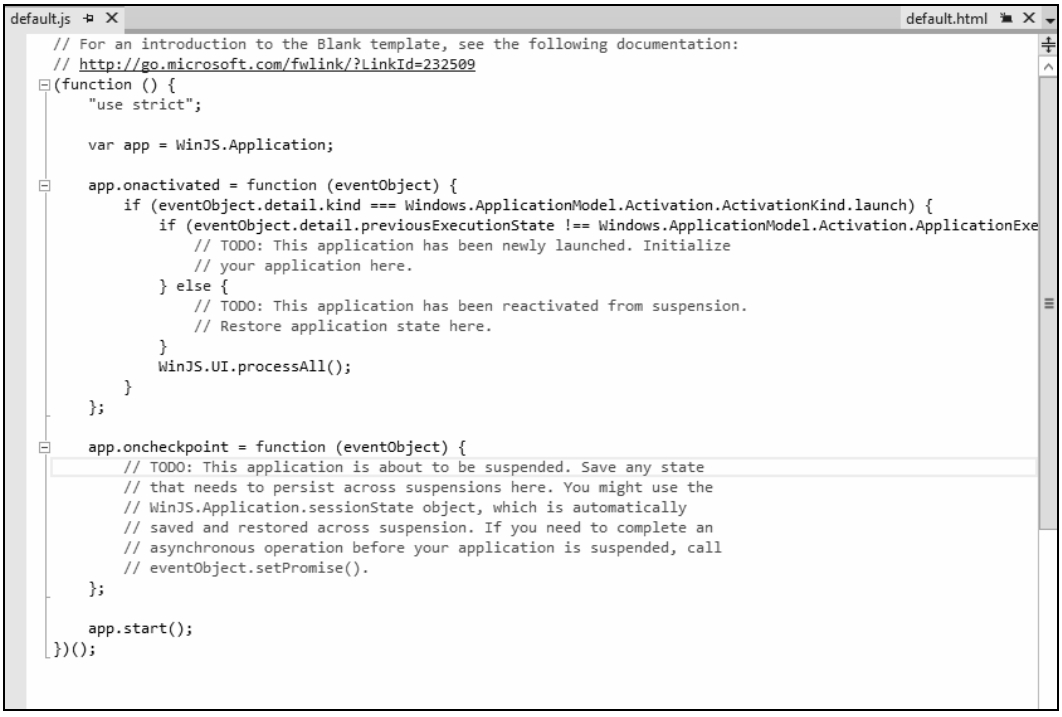


Рис. 2.4. Два окна документов, открытых в клиентской области (представлены двумя вкладками)

В этом случае нам поможет *панель вкладок* (рис. 2.5), находящаяся в верхней части клиентской области, ниже панели инструментов. С помощью панели вкладок можно переключиться на окно документа, в котором открыт нужный нам файл (сделать данное окно *активным*).

Рис. 2.5. Панель вкладок, позволяющая переключаться между двумя открытыми окнами документов

Содержимое типичного проекта Metro-приложения

А теперь давайте вернемся к панели **SOLUTION EXPLORER**, в которой, как мы недавно узнали, выводится содержимое открытого решения. И посмотрим, что же входит в созданный нами проект, который можно рассматривать как проект типичного Metro-приложения.

Файлы и папки, входящие в проект, мы перечислим в порядке их значимости для нас. То есть файлы и папки, с которыми мы будем иметь дело чаще всего, мы расположим в начале, а те, что, скорее всего, никогда не будут открыты, — в конце.

- default.html — основной файл, описывающий интерфейс Metro-приложения. Именно этот файл загружается, обрабатывается и выводится на экран при запуске приложения.

  НА ЗАМЕТКУ

  Знатоки Web-дизайна могут заявить, что файл default.html хранит не описание интерфейса Metro-приложения, а Web-страницу. Да, это так. Metro-приложение, написанное на языках HTML, CSS и JavaScript, фактически представляет собой Web-страницу.

- images — папка, хранящая файлы с графическими изображениями, которые используются в приложении. Изначально хранит несколько файлов с изображениями, используемыми, в частности, на стартовом экране и плитке меню **Пуск** (Start); все эти файлы помещает туда сам Visual Studio.

  В дальнейшем все графические файлы, применяемые в приложении, рекомендуется помещать в папку images. Для их организации можно использовать вложенные папки.

- js — папка, хранящая все файлы с программной логикой Metro-приложения. Изначально там присутствует единственный файл default.js, созданный самим Visual Studio.

  Логику простых Metro-приложений вполне можно уместить в файле default.js. Но если разрабатывается достаточно сложное Metro-приложение, его логику для удобства можно разбить на части и сохранить каждую из них в отдельном файле.

- css — папка, хранящая все файлы с описанием оформления Metro-приложения. Изначально там находится единственный файл default.css, созданный Visual Studio.

  Опять же, оформление простого Metro-приложения можно поместить в файл default.css. В случае же сложного оформления лучше разбить его на части и сохранить каждую часть в отдельном файле.

- package.appxmanifest — файл, хранящий параметры проекта.

- <*имя проекта*>_TemporaryKey.pfx — файл цифровой подписи.

- References — папка, хранящая файлы — ссылки на дополнительные модули. Описание работы с такими модулями выходит за рамки данной книги.

Конечно, разработчик может добавить в проект другие файлы. Это могут быть файлы фрагментов (подробнее о них разговор пойдет в *главе 13*), текстовые файлы, файлы данных XML и JSON, звуковые и видеофайлы, наконец, файлы с оформлением и программной логикой приложения и графические файлы. Также разработчик может создать в проекте сколько угодно папок.

Сразу же после создания проекта файлы default.html и default.js будут открыты. Так что мы сразу же сможем приступить к работе над интерфейсом и программной логикой приложения. Чем сейчас и займемся.

Первое Metro-приложение, часть первая

Наше первое Metro-приложение будет представлять собой простейший видеопроигрыватель. Пусть он пока что будет воспроизводить один-единственный видеофайл, который мы сейчас включим в состав проекта.

Прежде всего, давайте найдем подходящий видеофайл. Он не должен быть очень большим и, вместе с тем, обязан воспроизводиться, по крайней мере, в течение пары минут, чтобы мы смогли оценить, нормально ли работает наше приложение. Автор в качестве примера использовал трейлер фильма "Крик 4".

> ***Внимание!***
> Видеофайл должен иметь формат, поддерживаемый Windows 8, и хранящиеся в нем звук и видео также должны быть закодированы в поддерживаемых форматах. Автор рекомендует следующие комбинации формата файла, формата кодирования звука и формата кодирования видео: AVI / MP3 / MPEG IV ASP и MPEG IV File / AAC / MPEG IV AVC.

Добавление файла в проект

Итак, видеофайл у нас есть. Дадим ему имя, содержащее только буквы латинского алфавита, цифры и символы подчеркивания, — так нам впоследствии будет проще указать на него ссылку. Лучше всего вообще ему дать самое простое имя, например movie.

Теперь нам нужно добавить этот файл в проект. Выполняется это в два этапа.

На первом этапе нам следует поместить добавляемый файл в папку проекта. Сделать это можно в стандартном Проводнике или любой другой программе управления файлами.

Поместим наш фильм прямо в папку проекта. Вообще, по-хорошему, для него следует создать вложенную папку, но мы этого делать не будем. Все равно вскоре, совершенствуя наше приложение, мы его удалим.

На втором этапе мы собственно добавим файл в проект. Переключимся на Visual Studio и найдем в иерархическом списке панели **SOLUTION EXPLORER** пункт, представляющий проект `VideoPlayer`. Щелкнем на этом пункте правой кнопкой мыши и выберем в подменю **Add** появившегося на экране контекстного меню пункт **Existing Item**. (Также можно выбрать пункт **Add Existing Item** меню **Project** или нажать комбинацию клавиш <Shift>+<Alt>+<A>.) На экране появится диалоговое окно **Add Existing Item**, похожее на стандартное диалоговое окно открытия файла; выберем в нем добавляемый файл и нажмем кнопку открытия.

После этого наш видеофайл будет добавлен в состав проекта и появится в списке панели **SOLUTION EXPLORER**.

Открытие файла

Как мы уже знаем, интерфейс приложения описывается в файле default.html. Переключимся на него, щелкнув на соответствующей ему вкладке, что находится на панели вкладок (см. рис. 2.5).

Если же такой вкладки там нет (а, скорее всего, так и будет), нам следует открыть данный файл. Проще всего сделать это, воспользовавшись уже знакомой нам панелью **SOLUTION EXPLORER**. Найдем в ней пункт, представляющий нужный нам файл, и дважды щелкнем на нем. Кстати, двойным щелчком на папке мы можем открыть ее содержимое.

Visual Studio выведет содержимое открытого нами файла в новом окне документа и тотчас сделает его активным.

Создание интерфейса

К сожалению, Visual Studio на данный момент не имеет средств для визуального создания интерфейса Metro-приложений. Другими словами, мы не сможем нарисовать Metro-приложение мышью. Нам придется вводить код, описывающий интерфейс приложения, вручную.

Посмотрим на содержимое открытого нами файла default.html. Мы увидим что-то подобное этому:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>VideoPlayer</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.0.6/css/ui-dark.css" rel="stylesheet">
    <script src="//Microsoft.WinJS.0.6/js/base.js"></script>
    <script src="//Microsoft.WinJS.0.6/js/ui.js"></script>

    <!-- VideoPlayer references -->
    <link href="/css/default.css" rel="stylesheet">
    <script src="/js/default.js"></script>
</head>
<body>
    <p>Content goes here</p>
</body>
</html>
```

Весь этот код сгенерировал Visual Studio сразу при создании данного файла. Он определяет ссылки на файлы, хранящие описание оформления и программную логику, и различную служебную информацию.

Код, описывающий интерфейс, помещается между словами `<body>` и `</body>` (в предыдущем листинге выделены полужирным шрифтом). Уже присутствующий там код (выделен курсивом) можно удалить — это просто заготовка, созданная Visual Studio нам в помощь.

Для нашего приложения код интерфейса будет таким:

```
<video src="/movie.mp4" autoplay controls></video>
```

Он, образно говоря, указывает платформе Metro следующее:

1. Вывести на экран особый элемент интерфейса, предназначенный для воспроизведения видеофайла, — видеопроигрыватель.
2. Открыть в нем видеофайл movie.mp4, хранящийся в папке проекта. (Так данный файл называется у автора; у вас его имя может быть другим. Единственное: перед именем файла обязательно следует поставить символ слеша — /.)
3. Запустить воспроизведение файла сразу же после его открытия.
4. Вывести на экран элементы управления воспроизведением, поддерживаемые встроенным видеопроигрывателем. Они включают кнопку приостановки и возобновления воспроизведения, регулятор, показывающий текущую позицию воспроизведения, регулятор громкости и кнопку включения-отключения звука.

Теперь мы убедились, что платформа Metro выполняет многие задачи по воспроизведению видео самостоятельно. Нам для этого достаточно написать всего одну строчку кода.

Сохранение файлов

Теперь следует сохранить исправленный файл. Это необходимо проделывать всякий раз перед запуском приложения.

Сохранить содержимое файла, открытого в активном окне документа, проще простого — достаточно нажать комбинацию клавиш <Ctrl>+<S>. Также можно нажать расположенную на панели инструментов кнопку **Save *<имя файла, открытого в активном окне документа>*** () или выбрать в меню **File** одноименный пункт.

В дальнейшем, разрабатывая более сложные приложения, мы будем активно работать сразу с несколькими файлами. Чтобы сохранить содержимое сразу всех исправленных файлов, следует нажать комбинацию клавиш <Ctrl>+<Shift>+<S>. Если же мы не хотим отрывать руки от мыши и тянуться к клавиатуре, то можем нажать расположенную на панели инструментов кнопку **Save All** () или выбрать пункт **Save All** меню **File.**

Запуск Metro-приложения

Что ж, настала пора запустить наше первое Metro-приложение и посмотреть, работает ли оно.

Проще всего запустить приложение, нажав клавишу <F5>. Еще можно выбрать пункт **Start Debugging** меню **Debug** или нажать расположенную на панели инструментов кнопку 

.

Как выполняются Metro-приложения

Теперь немного отвлечемся от Visual Studio и рассмотрим только что запущенное Metro-приложение. На его примере мы выясним, по каким принципам работают все приложения подобного рода.

Изначально на экране появится *заставка* — графическое изображение, выводимое на экран в процессе запуска Metro-приложения. На рис. 2.6 можно видеть заставку по умолчанию, которую Visual Studio самостоятельно добавляет в каждое вновь созданное приложение; она представляет собой схематичное изображение часов на синем (на рис. 2.6 — сером) фоне.

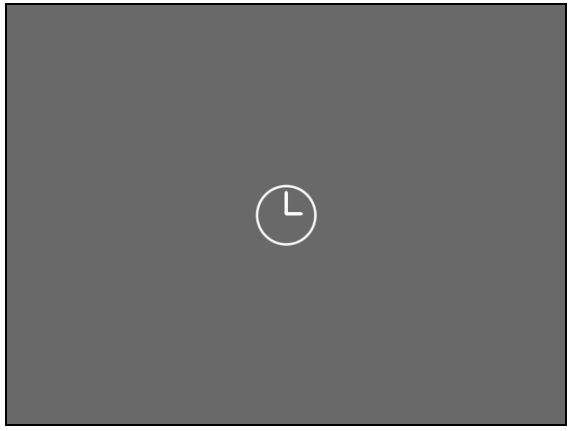


Рис. 2.6. Заставка Metro-приложения по умолчанию

Как только приложение будет успешно запущено и инициализировано, заставка пропадет, и ее сменит интерфейс, описанный в файле default.html. Интерфейс нашего приложения можно видеть на рис. 2.7.

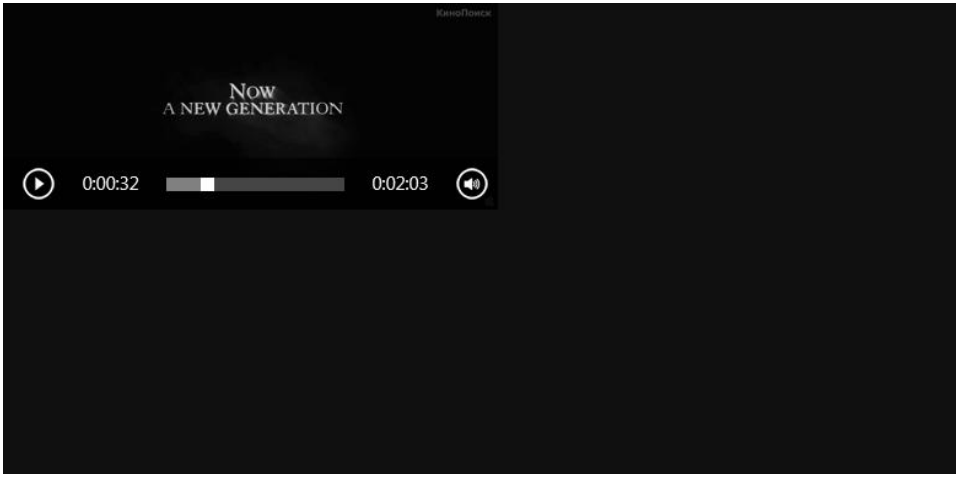


Рис. 2.7. Интерфейс нашего первого Metro-приложения

Здесь мы видим видеопроигрыватель, в котором воспроизводится фильм. Если мы наведем на этот элемент интерфейса курсор мыши, в его нижней части появятся элементы управления, с помощью которых мы можем управлять воспроизведением (на рис. 2.7 они как раз видны).

Вдоволь наигравшись со своим первым приложением, поговорим о принципах, согласно которым выполняются Metro-приложения. Их всего два.

Первый принцип — любое Metro-приложение заполняет весь экран компьютера. При этом никаких системных элементов интерфейса, наподобие заголовка окна или Панели задач, на экране присутствовать не будет.

Из этого следует важный вывод. Интерфейс Metro-приложения должен проектироваться с таким расчетом, чтобы максимально заполнять весь экран. Никакого пустого пространства на нем оставаться не должно. Так что наше первое приложение, с его маленьким видеопроигрывателем, сиротливо жмущимся в левом верхнем углу экрана, — очень плохой стиль программирования.

Второй принцип — Metro-приложения не должны предусматривать никаких команд (кнопок или пунктов меню) для их завершения. Вообще, Metro-приложения не должны завершаться самим пользователем — в этом их отличие от традиционных "настольных" приложений.

Вспомним, что говорилось в *главе 1*. Платформа Metro автоматически приостанавливает все неактивные приложения, а при существенной нехватке системных ресурсов она завершит приложения, с которыми пользователь давно не работал. То есть Metro берет управление приложениями в свои руки; это предусмотрено, опять же, с целью сделать работу с планшетом проще для неквалифицированного пользователя.

> ***НА ЗАМЕТКУ***
>
> Вообще-то, Microsoft разрешает включать в интерфейс приложения команду на его завершение. Но делать это рекомендуется только для целей отладки, чтобы в случае возникновения ошибки быстро завершить приложение.

Поставим воспроизводящийся фильм на паузу, чтобы он не "тормозил" компьютер. И переключимся на Visual Studio.

Первое Metro-приложение, часть вторая

Продолжим работу над нашим первым Metro-приложением. Для начала давайте увеличим размер видеопроигрывателя, чтобы сократить раздражающее пустое пространство вокруг него.

Активизируем окно документа, в котором открыт файл default.html. И найдем фрагмент кода, создающий на экране видеопроигрыватель. Вот он:

```
<video src="/movie.mp4" autoplay controls></video>
```

Немного исправим этот код, чтобы он выглядел вот так:

```
<video src="/movie.mp4" autoplay controls width="640" height="480">
↳</video>
```

Здесь мы указали платформе Metro размеры видеопроигрывателя — 640×480 пикселов. (В принципе, их можно сделать и больше, но пока сойдут и такие.)

Сохраним исправленный файл и снова запустим приложение.

Перезапуск Metro-приложения

Стоп! Но ведь наше приложение уже запущено! Выходит, сначала нам придется его завершить, а только после этого запустить снова — уже с внесенными нами исправлениями. Или, говоря другими словами, выполнить *перезапуск* приложения. Как это сделать?

Проще всего — нажав комбинацию клавиш <Ctrl>+<Shift>+<F5>. Также можно нажать кнопку **Restart** (), расположенную на панели инструментов, или выбрать пункт **Restart** меню **Debug**.

Сразу после этого Visual Studio сам завершит запущенное ранее приложение и запустит его исправленную версию.

Посмотрим, что у нас получилось, — рис. 2.8. Да, так уже гораздо лучше!

Полюбовавшись плодами трудов, снова переключимся на Visual Studio.

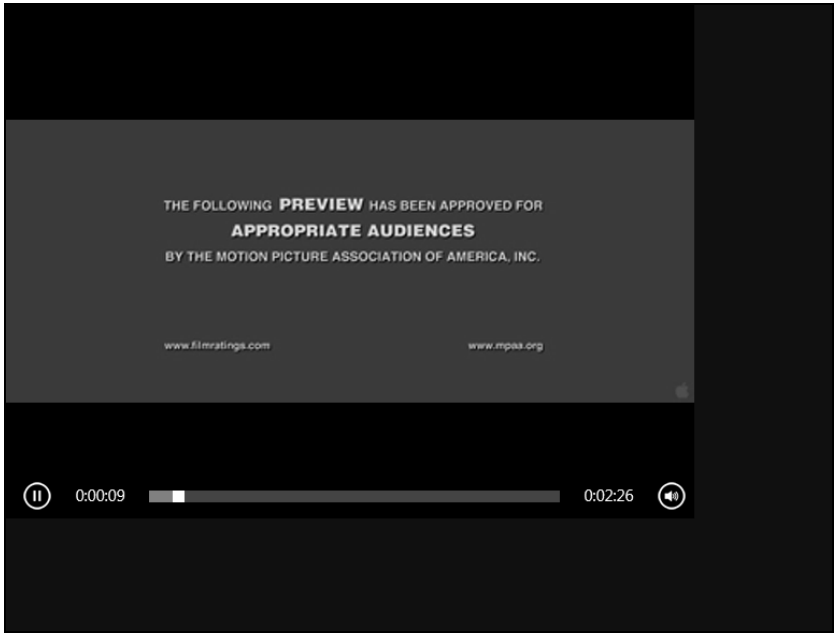


Рис. 2.8. Исправленное Metro-приложение с увеличенным видеопроигрывателем

Завершение Metro-приложения

Да, возможность перезапустить приложение очень полезна — тут не о чем спорить. Но как совсем завершить работу запущенного приложения?

Самый простой способ — нажать комбинацию клавиш <Shift>+<F5>. (Уж простите автора — он привык к клавиатуре...) Поклонники мыши могут нажать кнопку **Stop Debugging** () в панели инструментов или выбрать пункт **Stop Debugging** меню **Debug**.

Закрытие решения

Теперь предположим, что мы решили сделать перерыв и собираемся пока закрыть открытое в Visual Studio решение. Как это сделать?

Выбором пункта **Close Solution** меню **File**.

Если мы перед этим забыли выполнить сохранение какого-либо из исправленных файлов, входящих в решение, Visual Studio напомнит нам об этом. На экране появится диалоговое окно сохранения файлов, показанное на рис. 2.9.

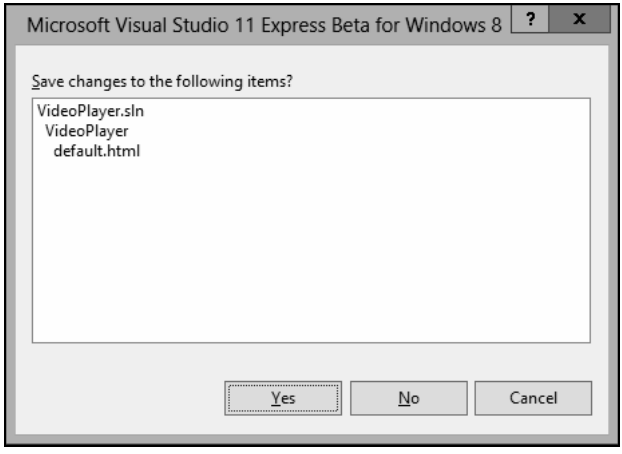


Рис. 2.9. Диалоговое окно сохранения файлов

Здесь мы видим список, занимающий бóльшую часть окна и перечисляющий все несохраненные файлы. Чтобы сохранить их, следует нажать кнопку **Yes**, а чтобы отказаться от этого — кнопку **No**. Нажатие кнопки **Cancel** отменяет закрытие решения.

Первое Metro-приложение, часть третья

Наше первое Metro-приложение совсем примитивно. Оно выполняет всего одну задачу — воспроизведения заранее заданного видеофайла. Реальная польза от него невелика.

Давайте доработаем это приложение таким образом, чтобы в нем можно было воспроизводить любой выбранный пользователем файл. Для этого нам придется добавить в приложение логику. И заодно, чтобы немного попрактиковаться, создадим его оформление, хотя бы самое простое.

Открытие решения

Сначала откроем решение `VideoPlayer`, которое мы ранее закрыли "на перерыв".

Проще всего сделать этого, открыв подменю **Recent Projects and Solutions** меню **File**. В этом подменю присутствуют пункты, представляющие созданные или от-

крывавшиеся ранее хотя бы раз проекты и решения; названия этих пунктов будут представлять собой пути к файлам проектов и решений. Нам останется только выбрать нужный пункт.

Есть и другой способ открыть решение, которое ранее открывалось хотя бы раз. Найдем на стартовой странице графу **Recent Projects**, в ней отыщем гиперссылку, представляющую нужное решение, и щелкнем на ней. (На рис. 2.1 эта графа пуста, т. к. мы до этого еще не открывали ни одного решения.)

Но что делать, если решение, которое мы собираемся открыть, еще ни разу не открывалось до этого? Выбрать пункт **Open Project** меню **File** или нажать комбинацию клавиш <Ctrl>+<Shift>+<O>. На экране появится стандартное диалоговое окно открытия файла Windows. Нам останется войти в папку нужного нам решения и выбрать его файл; в нашем случае это будут папка VideoPlayer и файл VideoPlayer.sln.

Как только решение будет открыто, откроем файл default.html. Найдем уже знакомый нам код, создающий видеопроигрыватель.

```
<video src="/movie.mp4" autoplay controls width="640" height="480">
↳</video>
```

Удалим из него ссылку на воспроизводимый видеофайл и добавим в него кое-какие служебные данные:

```
<video id="vidMain" autoplay controls width="640" height="480"></video>
```

Теперь нам нужно добавить код, создающий кнопку для открытия видеофайла. Поместим этот код ниже (добавленный код выделен полужирным шрифтом):

```
<video id="vidMain" autoplay controls width="640" height="480"></video>
<div class="button-cont">
    <input type="button" id="btnOpen" value="Открыть" />
</div>
```

Тем самым мы указали платформе Metro следующее:

1. Вставить ниже видеопроигрывателя блок — особый элемент интерфейса, о котором речь пойдет в *главе 9*.
2. Поместить в этом блоке кнопку **Открыть**.

Создание логики

На очереди — создание логики нашего приложения. Логика описывает поведение приложения в ответ на действия пользователя.

Как мы знаем, обычно логика описывается в файле default.js. Переключимся на окно документа, в котором открыт этот файл, и посмотрим на изначальный код, созданный самим Visual Studio (приводится в сокращении):

```
// For an introduction to the Blank template, see the following
↳documentation:
```

```
// http://go.microsoft.com/fwlink/?LinkId=232509
(function () {
    "use strict";

    var app = WinJS.Application;

    app.onactivated = function (eventObject) {
        . . .
    };

    app.oncheckpoint = function (eventObject) {
        . . .
    };

    app.start();
})();
```

Этот код запускает Metro-приложение на выполнение и производит некоторые важные предустановки. Поэтому удалять его ни в коем случае не следует.

Вставим сюда код, создающий логику нашего приложения. То, что у нас должно получиться в результате, показано далее (добавленный нами код выделен полужирным шрифтом).

```
// For an introduction to the Blank template, see the following
↳documentation:
(function () {
    "use strict";

    var app = WinJS.Application;

    var btnOpen, vidMain;

    app.onactivated = function (eventObject) {
        . . .
    };

    app.oncheckpoint = function (eventObject) {
        . . .
    };

    function btnOpenClick() {
        var oFP = new Windows.Storage.Pickers.FileOpenPicker();
        oFP.viewMode = Windows.Storage.Pickers.PickerViewMode.list;
        oFP.suggestedStartLocation =
        ↳Windows.Storage.Pickers.PickerLocationId.videosLibrary;
        oFP.fileTypeFilter.replaceAll([".avi", ".mp4", ".wmv"]);
```

```
        oFP.pickSingleFileAsync().then(function (oFile) {
            if (oFile) {
                vidMain.src = URL.createObjectURL(oFile);
            }
        });
    }

    document.addEventListener("DOMContentLoaded", function () {
        btnOpen = document.getElementById("btnOpen");
        vidMain = document.getElementById("vidMain");
        btnOpen.addEventListener("click", btnOpenClick);
    });

    app.start();
})();
```

Этот код описывает действия, которые наше приложение выполнит при нажатии созданной ранее кнопки **Открыть**.

1. Вывод на экран стандартного диалогового окна открытия файла платформы Metro. При этом:
   - данное окно представит файлы в виде списка, каждая позиция которого включит имя файла и его миниатюру;
   - данное окно изначально выведет содержимое системной библиотеки видеофайлов;
   - данное окно отобразит файлы с расширениями avi, mp4 и wmv.
2. После выбора пользователем файла — открытие его в видеопроигрывателе.

Сразу сохраним все исправленные файлы. Вообще, сохранять результаты работы следует как можно чаще — мало ли что...

Создание оформления

Оформление приложения описывает внешний вид его элементов интерфейса: их цвет, размеры, выравнивание, используемый для вывода текста шрифт и пр. Обычно оформление хранится в файле default.css, который находится в папке css.

Мы сразу увидим, что файл default.css уже имеет некое содержимое, помещенное в него самим Visual Studio:

```
body {
}

@media screen and (-ms-view-state: fullscreen-landscape) {
}

@media screen and (-ms-view-state: filled) {
}
```

```
@media screen and (-ms-view-state: snapped) {
}

@media screen and (-ms-view-state: fullscreen-portrait) {
}
```

Это содержимое ничего полезного не выполняет и является лишь заготовкой для создания оформления.

Давайте вставим в начало этого файла такой фрагмент кода:

```
.button-cont
{
    text-align: right;
    width: 640px;
}
```

Ранее, внося исправления в интерфейс нашего приложения, мы вставили ниже видеопроигрывателя блок, в который поместили кнопку **Открыть**. Так вот, приведенный ранее код задает оформление именно для данного блока.

- Все содержимое блока (т. е. кнопка) должно быть выровнено по его правому краю.
- Ширина блока должна составлять 640 пикселов — как у видеопроигрывателя.

В результате кнопка **Открыть** будет находиться у правого нижнего угла видеопроигрывателя, чуть ниже его.

Закрытие файла

Более файл default.css нам не понадобится, поэтому давайте сразу его закроем, чтобы освободить системные ресурсы компьютера для более полезных задач.

Проще всего закрыть файл, закрыв окно документа, в котором он открыт. Для этого достаточно сделать это окно активным и нажать кнопку закрытия, что находится в правой части вкладки данного окна, расположенной в панели вкладок, и имеет вид крестика (см. рис. 2.5).

Другой способ закрыть файл, открытый в активном окне документа, — выбрать пункт **Close** меню **File**.

Если мы еще не сохранили закрываемый файл, Visual Studio выведет на экран уже знакомое нам диалоговое окно сохранения файлов (см. рис. 2.9). Чтобы сохранить закрываемый файл, следует нажать кнопку **Yes** данного окна, а кнопка **No** позволит нам от этого отказаться. Что касается кнопки **Cancel**, то она отменяет закрытие файла.

Удаление файла из проекта

Поскольку в новой версии нашего Metro-приложения видеофайл для воспроизведения будет выбирать сам пользователь, файл movie.mp4, все еще входящий в состав проекта, нам больше не нужен. Так что давайте его удалим.

Сделать это можно в панели **SOLUTION EXPLORER**. Найдем в ее списке ненужный нам файл, щелкнем на нем правой кнопкой мыши и выберем в появившемся на экране контекстном меню пункт **Exclude From Project**. Данный файл сразу же будет исключен из проекта.

Отметим, что исключенный таким образом из проекта файл, тем не менее, останется на диске. Если он нам действительно больше не нужен, нам придется удалить его вручную.

Однако Visual Studio позволяет нам исключить файл из состава проекта с одновременным удалением его с диска. Для этого достаточно щелкнуть на ненужном файле правой кнопкой мыши и выбрать в появившемся контекстном меню пункт **Delete**. Также можно выделить файл простым щелчком мыши и нажать клавишу <Del>.

На экране появится окно-предупреждение, сообщающее, что данный файл сейчас будет удален как из состава проекта, так и с диска. Нажмем кнопку **OK**, чтобы удалить файл, или кнопку **Cancel**, чтобы отказаться от его удаления.

> ***Внимание!***
> Visual Studio поместит удаленный из проекта файл в системную Корзину, так что мы сможем в случае необходимости его восстановить.

Осталось только проверить исправленное приложение в работе. Сохраним все измененные файлы и запустим приложение на выполнение. Когда его интерфейс появится на экране (рис. 2.10), нажмем кнопку **Открыть**, выберем какой-либо видеофайл и посмотрим, как он будет воспроизводиться.

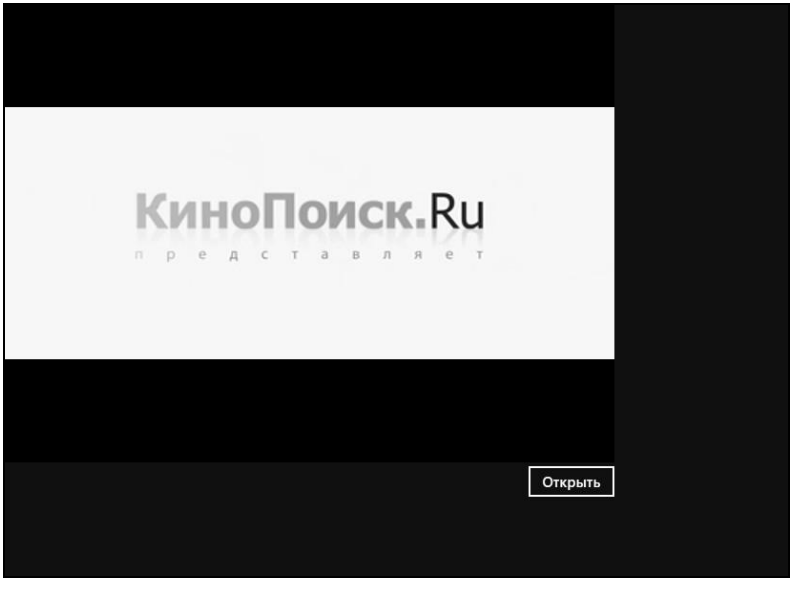


Рис. 2.10. Окончательная версия нашего первого Metro-приложения

Выявление и исправление ошибок

Если мы все делали правильно и не допускали ошибок в коде приложения, все у нас должно работать нормально. Но ведь так происходит далеко не всегда...

К счастью, грубые ошибки в коде логики выявляются сразу, прямо в процессе работы приложения. В этом случае произойдет автоматическое переключение на Visual Studio, и мы увидим на экране диалоговое окно ошибки (рис. 2.11).

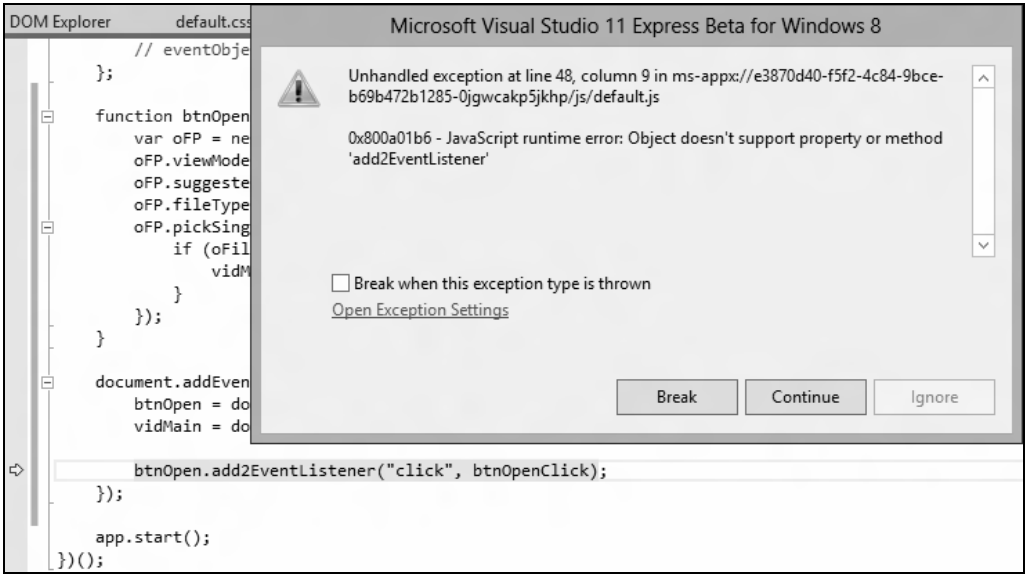


Рис. 2.11. Диалоговое окно ошибки и ошибочная команда, помеченная Visual Studio

В этом окне мы увидим большое текстовое поле с кратким описанием ошибки и три кнопки:

- **Break** — прерывает выполнение приложения;
- **Continue** и **Ignore** — в нашем случае продолжают выполнение приложения. Однако следует иметь в виду, что после нажатия этих кнопок приложение может работать не так, как было запланировано.

Если возникшая ошибка действительно вызвана неверной командой в коде логики (как и показано на рис. 2.11), следует нажать кнопку **Break**, исправить команду и перезапустить приложение.

Но как найти строку кода, содержащую неверную команду? Очень просто. Visual Studio сам выделит ее в окне документа желтым фоном и дополнительно пометит желтой стрелкой (на рис. 2.11 фон и стрелка — серые).

К сожалению, таким образом можно найти только ошибки в логике приложения. Ошибки в описании его интерфейса и оформления выявляются только, как говорится, глазами. Это еще один повод почаще запускать разрабатываемое приложение и смотреть, как оно выглядит и работает.

Что дальше?

В этой главе мы познакомились со средой разработки Visual Studio, которая, в числе прочего, позволяет создавать и Metro-приложения. Мы создали наше первое Metro-приложение, после чего доработали его с целью сделать более универсальным. Мы узнали, как создаются интерфейс, оформление и логика, как запускать, перезапускать и останавливать Metro-приложения, как в среде Visual Studio открыть, сохранить и закрыть файл и как там выявляются и исправляются ошибки. Наконец, мы увидели Metro-приложение воочию и рассмотрели некоторые особенности работы этого класса приложений.

Следующая глава будет посвящена языкам HTML и CSS, на которых описываются, соответственно, интерфейс и оформление Metro-приложения. Ограниченный объем книги не позволяет привести полный курс этих языков, но мы, по крайней мере, познакомимся с их основами.

ГЛАВА 3

Интерфейс и оформление Metro-приложения

В предыдущей главе мы создали свое первое Metro-приложение — видеопроигрыватель. Заодно мы познакомились с Visual Studio — средой разработки, позволяющей создавать Metro-приложения, и изучили ее основные особенности.

Эта глава посвящена языкам HTML и CSS, на которых описываются, соответственно, интерфейс и оформление Metro-приложений. Полное описание всех возможностей этих языков мы рассматривать не будем (им впору посвящать отдельную книгу), а изучим только азы.

Интерфейс приложения. Язык HTML

Как мы узнали еще в *главе 1*, интерфейс Metro-приложения в рамках выбранной нами технологии описывается на языке HTML. Настала пора познакомиться с этим языком.

Теги

Язык HTML определяет набор особых команд, каждая из которых соответствует определенной разновидности элементов интерфейса Metro-приложения: видеопроигрывателям, кнопкам, блокам, абзацам обычного текста и пр. Другими словами, если нам потребуется создать, скажем, кнопку, мы вставим в код, описывающий интерфейс, соответствующую ей команду. Эти команды называются *тегами* HTML.

Для примера давайте рассмотрим тег, создающий видеопроигрыватель, благо он нам уже знаком по *главе 2*:

```
<video></video>
```

Как видим, тег представляет собой слово *(имя тега)*, заключенное в символы "меньше" и "больше" (< и >). Собственно, эти символы и являются признаком тега.

Имя тега определяет, какой элемент интерфейса создает данный тег. Так, имя тега `video` говорит платформе Metro, чтобы она поместила на этом месте видеопроигрыватель. Имя тега `div` указывает Metro создать блок, имя тега `p` — абзац текста, а имя тега `textarea` — область редактирования.

Теперь давайте рассмотрим тег `<p>`, создающий абзац текста.

```
<p>Это абзац.</p>
```

Видно, что он фактически состоит из двух тегов. Первый тег — `<p>` — является *открывающим* и обозначает начало абзаца. А второй тег — `</p>` — является *закрывающим* и помечает его конец. Отметим, что закрывающий тег отличается от открывающего наличием символа слеша (`/`) между символом `<` и именем тега. Текст, который находится между этими тегами (*содержимое* тега), собственно и станет абзацем.

Подобные теги носят название *парных*. Они служат для создания сложных элементов интерфейса, имеющих какое-либо внутреннее содержимое: абзацев, заголовков, блоков и пр. В случае абзацев и заголовков содержимым парного тега является текст.

Нетрудно заметить, что тег `<video>` также является парным. Однако содержимого он практически никогда не имеет, т. к. в платформе Metro это содержимое никогда не выводится на экран и вообще не имеет смысла.

Рассмотрим другой парный тег. Он уже знаком нам по *главе 2*.

```
<div>
    <input />
</div>
```

Да, парный тег `<div>` формирует блок. В качестве его содержимого указываются теги, создающие элементы интерфейса, которые должны находиться в этом блоке (в нашем случае — тег `<input>`, создающий элемент управления). Кстати, это хороший пример того, что содержимым тега может быть не только обычный текст.

Теперь подробнее рассмотрим тег `<input>`.

```
<input />
```

Это *одинарный* тег. От парного он отличается, прежде всего, тем, что принципиально не может иметь содержимого. Признаком одинарного тега является символ слеша, который ставится между именем тега и символом `>` и отделяется от имени тега пробелом.

Вот еще один одинарный тег:

```
<img />
```

Он выводит графическое изображение.

Одинарные теги служат для формирования самых простых элементов интерфейса, не имеющих внутреннего содержимого: кнопок, полей ввода, флажков, переключателей, графических изображений и др.

А вот небольшой фрагмент HTML-кода:

```
<video></video>
<div>
    <input />
</div>
```

Он формирует на экране видеопроигрыватель (тег `<video>`) и блок (тег `<div>`), в который помещает элемент управления (тег `<input>`).

Тегов, поддерживаемых языком HTML, очень много. Далее в этой книге мы рассмотрим те из них, что обычно применяются для создания интерфейса Metro-приложений.

Атрибуты тегов

Вы думаете, тег `<input>` на предыдущем примере сформирует кнопку? Как бы не так! Он создаст поле ввода.

Дело в том, что данный тег "отвечает" за формирование сразу целой группы различных элементов управления, в которую входят, помимо кнопок, и поля ввода, и флажки, и переключатели. И по умолчанию, если специально не указать в этом теге тип элемента управления, который он должен сформировать, будет создано поле ввода.

Чтобы изменить поведение по умолчанию какого-либо тега, нам потребуется задать для него дополнительные параметры. Для тега `<input>` таким параметром станет тип создаваемого с его помощью элемента интерфейса.

Параметры тегов задаются посредством особых языковых конструкций HTML, называемых *атрибутами тега*. Каждый такой атрибут задает одну из характеристик тега и тем самым изменяет один из аспектов его поведения. Если же атрибут тега не задан, тег будет вести себя по умолчанию.

Давайте рассмотрим такой пример кода:

```
<input type="button" />
```

Это тег `<input>`, содержащий атрибут `type`, который задает один из параметров этого тега, а именно тип создаваемого им элемента интерфейса. Строка `button`, указанная нам для этого параметра, говорит Metro, что данный тег должен сформировать кнопку.

Рассмотрим этот атрибут тега подробнее. Прежде всего, мы видим *имя* атрибута тега (`type`), которое однозначно его идентифицирует. Далее стоит знак равенства (`=`), а за ним — *значение* данного атрибута (и, соответственно, представляемого им параметра тега). Отметим, что значение это обязательно берется в двойные кавычки.

Атрибуты тега записываются между именем тега и символами `>` или `/>`. Отделяться от имени тега и друг от друга они должны, по крайней мере, одним пробелом или символом разрыва строки.

В теге может быть указано сколько угодно атрибутов. Например:

```
<input type="button" value="Открыть" />
```

Здесь мы указали в теге `<input>` еще один атрибут — `value`, который в случае кнопки задает для нее надпись.

Как уже говорилось, атрибуты тега друг от друга и от имени тега можно отделять не только пробелами, но и символами разрыва строки. Приведенный ранее пример кода мы можем записать так:

```
<input type="button"
  value="Открыть" />
```

Вот еще два примера тегов с атрибутами:

```
<video src="/movie.mp4"></video>
<img src="/images/picture.jpg" />
```

Здесь с помощью атрибута тега `src` мы указали для видеопроигрывателя (тег `<video>`) ссылку на файл, который должен быть в нем открыт. После чего с помощью того же атрибута задали для тега графического изображения (`<img>`) ссылку на файл, который хранит выводимое на экран изображение.

Ранее мы рассматривали только атрибуты тега, для которых всегда указывается значение (их так и называют — *атрибуты со значением*). Но HTML поддерживает целый ряд атрибутов, для которых значение никогда не указывается *(атрибуты без значения)*. Своим присутствием в теге они задействуют какую-то возможность, предусматриваемую этим тегом; по умолчанию (т. е. если атрибут в теге не указан) эта возможность не задействуется.

Вот классический пример атрибута тега без значения:

```
<video controls></video>
```

Атрибут тега без значения `controls` своим присутствием указывает видеопроигрывателю вывести элементы для управления воспроизведением фильма. Если же его не указывать, эти элементы управления выведены не будут.

Язык HTML поддерживает много самых разных атрибутов тегов. Какие-то из них поддерживаются всеми тегами, другие же специфичны для определенной группы или вообще для одного-единственного тега. В дальнейшем мы познакомимся со всеми полезными для нас атрибутами тегов.

Порядок вывода элементов интерфейса. Блочные и встроенные элементы

Описание интерфейса Metro-приложения представляет собой последовательность тегов HTML, формирующих различные элементы этого самого интерфейса. Но в каком порядке они выводятся на экран?

Для примера давайте рассмотрим такой фрагмент кода:

```
<video src="/movie.mp4" controls></video>
<div>
```

```
    <input type="button" value="Открыть" />
    <input type="button" value="Пуск" />
    <input type="button" value="Стоп" />
</div>
```

Здесь мы видим видеопроигрыватель и блок с тремя кнопками. Так в каком порядке они будут расположены друг относительно друга?

Начать следует с того, что все элементы интерфейса и создающие их теги делятся на две группы.

- *Блочные* — выстраиваются друг относительно друга по вертикали сверху вниз в том порядке, в котором они определены в HTML-коде.

  Блочные элементы могут содержать внутри себя (т. е. в качестве содержимого) как другие блочные, так и встроенные элементы. (О встроенных элементах интерфейса — чуть позже.)

  Обычно блочными являются крупные интерфейсные элементы: абзацы, заголовки, таблицы, блоки и пр. Также блочным элементом является видеопроигрыватель.

- *Встроенные* — выстраиваются друг относительно друга по горизонтали слева направо в том порядке, в котором они определены в HTML-коде. Если встроенный элемент следует за блочным, оба этих элемента также выстроятся по горизонтали.

  Встроенные элементы могут содержать внутри себя только другие встроенные элементы более низкого уровня. В свою очередь, встроенные элементы обычно помещаются внутри блочных элементов.

  Встроенными являются мелкие интерфейсные элементы: отдельные элементы управления (кнопки, поля ввода, флажки, списки), фрагменты полужирного и курсивного текста и др. Также встроенными элементами являются графические изображения.

Теперь применим только что сказанное к приведенному ранее фрагменту кода.

- На самом верху поместится видеопроигрыватель (блочный элемент).
- Ниже его расположится блок (блочный элемент) с кнопками.
- Кнопки (встроенные элементы) внутри блока выстроятся по горизонтали слева направо в том порядке, в котором они определены в коде.

Давайте теперь мысленно удалим блок, но оставим находящиеся в нем кнопки:

```
<video src="/movie.mp4" controls></video>
<input type="button" value="Открыть" />
<input type="button" value="Пуск" />
<input type="button" value="Стоп" />
```

В таком случае кнопки разместятся правее видеопроигрывателя. Мы можем открыть созданное ранее Metro-приложение и проверить, так ли это. (Согласитесь, что получится совсем некрасиво.)

Существует, однако, способ выстроить элементы интерфейса в ином порядке — создание так называемой разметки. Мы обязательно этим займемся, но много позже — в *главе 9*.

Вложенность тегов

А теперь рассмотрим еще один важный вопрос. Он касается правильности написания HTML-кода.

Мы уже знаем, что теги могут вкладываться друг в друга. В качестве примера *вложенности* рассмотрим такой пример кода:

```
<body>
    <div>
        <input type="button" value="Открыть" />
        <input type="button" value="Пуск" />
        <input type="button" value="Стоп" />
    </div>
</body>
```

Парный тег `<body>` является служебным. Подробнее мы рассмотрим его потом.

Так вот, последовательность закрывающих тегов должны быть обратной последовательности тегов открывающих. Или, если говорить проще, теги должны вкладываться друг в друга, не оставляя "хвостов" снаружи.

Посмотрим на ранее приведенный код. Последовательность закрывающих тегов — `</div>` и `</body>` — обратна последовательности открывающих тегов `<body>` и `<div>`. Тег `<div>` полностью вложен в тег `<body>`.

Если же мы нарушим это правило, поставив закрывающие теги `</div>` и `</body>` в неверной последовательности (переставленные теги выделены полужирным шрифтом):

```
<body>
    <div>
        <input type="button" value="Открыть" />
        <input type="button" value="Пуск" />
        <input type="button" value="Стоп" />
    </body>
</div>
```

интерфейс приложения, скорее всего, будет отображаться некорректно.

Осталось запомнить еще несколько терминов, которыми мы будем активно пользоваться в дальнейшем.

- Тег, в который вложен данный тег, называется его *родителем*, или *родительским тегом*.
- Теги, вложенные в данный тег, называются его *потомками*, или *дочерними тегами*.

- Теги одного уровня вложенности (т. е. непосредственно вложенные в один и тот же тег) называются *соседями*.

Так, в приведенном ранее фрагменте кода для тега тег <body> является родителем, а теги <input> — дочерними. При этом теги <input> являются соседями.

Служебные теги. Структура HTML-файла

Напоследок поговорим о служебных тегах, определяющих вспомогательную информацию и структурирующих описание интерфейса для Metro-приложения.

Откроем наше первое Metro-приложение в Visual Studio и переключимся на файл default.html, где описывается интерфейс. Рассмотрим подробнее его содержимое (опустив собственно описание интерфейса, в целом, нам уже знакомое и понятное).

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>VideoPlayer</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.0.6/css/ui-dark.css" rel="stylesheet">
    <script src="//Microsoft.WinJS.0.6/js/base.js"></script>
    <script src="//Microsoft.WinJS.0.6/js/ui.js"></script>

    <!-- VideoPlayer references -->
    <link href="/css/default.css" rel="stylesheet">
    <script src="/js/default.js"></script>
</head>
<body>
    . . .
</body>
</html>
```

Здесь мы видим *служебные теги*. Как уже говорилось, они задают вспомогательные данные, которые используются самой платформой Metro и никогда не выводятся на экран, хотя и необходимы для успешной работы приложения.

Кратко рассмотрим эти теги.

Одинарный тег <!DOCTYPE html> указывает, что интерфейс Metro-приложения описывается на пятой версии языка HTML — *HTML5*. Обратим внимание, как записывается этот тег — без символа слеша перед замыкающим >.

Парный тег <html> объединяет весь HTML-код, описывающий интерфейс приложения, включая и служебные теги.

Парный тег <head>, вложенный в тег <html>, объединяет, во-первых, служебные теги, а во-вторых, теги — ссылки на файлы, хранящие оформление и программную

логику Metro-приложения. Можно сказать, что данный тег служит вместилищем для всех вспомогательных данных.

Одинарный тег `<meta>`, вложенный в тег `<head>`, задает кодировку, в которой сохранен данный файл. Это кодировка UTF-8 (значение атрибута тега `charset`).

> ***НА ЗАМЕТКУ***
>
> Описания интерфейсов всех Metro-приложений записываются на языке HTML5 и хранятся в кодировке UTF-8.

Парный тег `<title>`, также вложенный в тег `<head>`, задает название Metro-приложения. Это название, в общем-то, не нужно, однако стандарт языка HTML требует обязательно его указывать.

Парный тег `<body>`, вложенный в тег `<html>` (сосед тега `<head>`), объединяет весь код, собственно описывающий интерфейс приложения. Именно в него мы поместили наш первый HTML-код, когда создавали Metro-приложение в *главе 2*.

Остальные теги, вложенные в тег `<head>` и ссылающиеся на другие файлы, мы обсудим потом.

Комментарии HTML

Последнее, что мы рассмотрим, — это комментарии языка HTML.

Комментарий — это фрагмент содержимого HTML-файла, который не является частью кода, описывающего интерфейс приложения, а содержит какие-либо примечания разработчика. Эти примечания не обрабатываются платформой Metro, поэтому могут содержать что угодно.

Комментарии HTML заключаются в парный тег особого вида. Открывающий тег комментария записывается как `<!--`, а закрывающий — как `-->`.

```
<!-- WinJS references -->
```

Этот комментарий взят из реального HTML-кода, создаваемого в файле default.html самим Visual Studio. Он находится в теге `<head>` и помечает теги — ссылки на файлы, которые включаются в состав любого проекта без изменений.

```
<!-- Это блок с кнопками. -->
```

А такой комментарий мог бы создать сам разработчик.

Комментарии могут быть любого объема. Их следует делать как можно более лаконичными, ведь они увеличивают размер HTML-файла и, следовательно, самого Metro-приложения.

У комментариев есть и другое назначение. С их помощью разработчики в процессе отладки приложения могут временно скрыть часть HTML-кода его интерфейса, чтобы посмотреть, что получится в результате.

```
<body>
<!--    <div> -->
        <input type="button" value="Открыть" />
```

```
        <input type="button" value="Пуск" />
        <input type="button" value="Стоп" />
<!--    </div> -->
</body>
```

Здесь мы скрыли с помощью комментариев (закомментировали) теги, создающие блок.

Вот, пожалуй, и все о языке HTML. Остальные его теги и атрибуты тегов мы изучим потом, в процессе знакомства с книгой.

Оформление приложения. Каскадные таблицы стилей CSS

Оформление Metro-приложения определяет внешний вид элементов его интерфейса. Оно задает, в частности, такие параметры, как шрифт и цвет, которым будет выведен текст абзаца, его выравнивание, цвет фона самого приложения, параметры рамки, которая будет выведена вокруг видеопроигрывателя, и т. п.

Описание оформления для интерфейса хранится в файлах с расширением css. Для его создания применяется язык CSS. Он заметно отличается от уже знакомого нам HTML, но также очень прост.

Стили и атрибуты стилей

Оформление представляет собой комбинацию отдельных правил, каждое из которых описывает внешний вид либо одного-единственного элемента интерфейса, либо целой их группы, сформированной по какому-либо признаку (например, созданные с помощью одного и того же тега). Такие правила называются *стилями* CSS.

Давайте рассмотрим пример такого стиля.

```
.button-cont
{
    text-align: right;
    width: 640px;
}
```

Узнаете? Это описание оформления для нашего первого Metro-приложения, созданного в *главе 2*. Оно представляет собой один-единственный стиль, который действует на блок с кнопкой **Открыть**. Как мы помним, она задает для блока ширину, равную 640 пикселам, и выравнивание его содержимого по правому краю.

Итак, первый элемент описания, или *определения*, любого стиля — его *селектор*. Он записывается в самом начале определения и обозначает группу элементов интерфейса, на которую он будет действовать. В нашем случае селектором стиля является строка `.button-cont` (точка в начале для данной разновидности стилей обязательна).

После селектора через пробел или символ разрыва строки (его применяют чаще всего) задаются атрибуты стиля и их значения. Каждый *атрибут стиля* (не путать с атрибутом тега!) представляет определенный параметр оформления: название шрифта, цвет текста, выравнивание и др. А *значение* атрибута устанавливает конкретную величину данного параметра, например белый цвет текста или выравнивание по правому краю.

Список атрибутов стиля и их значений, присутствующий в определении стиля, обязательно берется в фигурные скобки. Между скобками и атрибутами стиля должен быть, по крайней мере, один пробел или символ разрыва строки.

Атрибут стиля и его значение отделяются друг от друга символом двоеточия и пробелом. Отметим, что значение атрибута стиля указывается без кавычек. В конце каждой пары "атрибут стиля — значение" должен стоять символ точки с запятой; на деле их дополнительно разделяют символами разрыва строки — так определение стиля лучше читается.

В нашем стиле присутствуют два атрибута:

- `text-align` — выравнивание содержимого данного элемента интерфейса (в нашем случае — выравнивание кнопки в блоке). Значение `right` этого атрибута стиля задает выравнивание по правому краю;
- `width` — ширина элемента интерфейса. Его значение указывается в любых единицах измерения, поддерживаемых CSS; в нашем случае это пикселы, о чем говорят символы `px`, поставленные после собственно значения ширины.

Подробнее о различных атрибутах стилей мы поговорим в последующих главах этой книги.

Рассмотрим еще один стиль:

```
progress
{
    width: 280px;
    height: 10px;
}
```

Он относится к другой разновидности, нежели стиль, рассмотренный нами в начале этого раздела. (Атрибут стиля `width` задает ширину элемента интерфейса; символы `px`, поставленные после собственно значения ширины, указывают, что она задана в пикселах. Атрибут стиля `height` задает высоту элемента интерфейса.)

В определении стиля можно указать сразу несколько селекторов, перечислив их через запятую:

```
.button-cont, other-cont { . . . }
```

В этом случае стиль будет применен к двум группам элементов интерфейса, описываемым обоими селекторами — и `.button-cont`, и `.other-cont`. Фактически мы таким образом создадим сразу два стиля.

Разновидности стилей. Привязка стилей

Все стили, поддерживаемые CSS, делятся на четыре разновидности. Разновидность, к которой относится стиль, определяется способом записи его селектора.

К первой из этих разновидностей относится самый первый из рассмотренных нами стилей:

```
.button-cont
{
    text-align: right;
    width: 640px;
}
```

Его селектор начинается с точки. Это значит, что стиль относится к так называемым *стилевым классам*.

Стилевые классы имеют следующие особенности:

- мы сами указываем, на какие элементы интерфейса будет действовать такой стиль. Или, другими словами, мы сами выполняем его *привязку* к элементам интерфейса;
- стилевой класс может быть привязан к любому элементу интерфейса и произвольному их количеству.

Чтобы привязать стилевой класс к элементу интерфейса, мы укажем его селектор в качестве значения атрибута `class` тега, который создает элемент. При этом селектор стилевого класса указывается в атрибуте тега `class` уже без точки в начале.

Вот пример того, как это делается:

```
<div class="button-cont">
    <input type="button" value="Открыть" />
</div>
```

Это фрагмент описания интерфейса нашего первого Metro-приложения. Здесь мы привязали стилевой класс `.button-cont` к блоку с кнопкой.

К одному и тому же элементу интерфейса можно привязать сразу несколько стилевых классов, перечислив их через пробелы:

```
<div class="button-cont special-cont"> . . . </div>
```

Теперь рассмотрим стиль, относящийся ко второй разновидности. Он нам также знаком.

```
progress
{
    width: 280px;
    height: 10px;
}
```

Селектор такого стиля не имеет в своем начале точки и фактически представляет собой имя тега. (Тег `<progress>` создает элемент управления — индикатор прогресса.) Это *стиль переопределения тега*.

Стили переопределения тега автоматически привязываются ко всем элементам интерфейса, что созданы с применением тега, чье имя совпадает с селектором стиля. Стало быть, приведенный ранее стиль будет привязан ко всем индикаторам прогресса (тегам <progress>), которые в результате получат ширину, равную 280 пикселам, и высоту в 10 пикселов.

Третья разновидность стилей нам пока не знакома. Вот пример стиля, относящегося к ней:

```
#button-cont
{
    text-align: right;
    width: 640px;
}
```

От стилевых классов и стилей переопределения тега он отличается наличием в начале своего селектора символа "решетки" (#). Такие стили называются *именованными*.

Особенности именованных стилей таковы:

- как и в случае стилевых классов, мы сами должны выполнять привязку именованных стилей к элементам интерфейса;
- именованный стиль может быть привязан только к одному элементу интерфейса.

Для привязки именованного стиля к элементу интерфейса применяется другой атрибут тега — id. Селектор привязываемого стиля указывается в качестве значения этого атрибута — уже без символа "решетки". Например:

```
<div id="button-cont">
    <input type="button" value="Открыть" />
</div>
```

> ***Внимание!***
>
> Атрибут тега id используется и для других целей. Подробнее об этом мы поговорим в *главе 5*.

Стили, относящиеся к четвертой разновидности, используют в качестве селекторов сложные конструкции, включающие в себя имена тегов, наименования стилевых классов и именованных стилей и специальные команды языка CSS, задающие особые условия. Поэтому их называют *комбинированными стилями*.

Рассмотрим несколько примеров селекторов таких стилей.

```
div input { . . . }
```

Будет применен к любому элементу управления, созданному тегом <input>, который вложен в блок (тег), причем не обязательно непосредственно.

```
<div>
    <input type="button" value="Открыть" />
    <p><input type="check" /></p>
</div>
```

Здесь мы видим два тега <input>, первый из которых создает кнопку, а второй — флажок (поскольку для его атрибута type задано значение check). Первый тег вложен непосредственно в блок, а второй помещен в абзац (тег <p>), который, в свою очередь, вложен в блок. Стиль с указанным ранее селектором будет применен к обоим тегам <input>, поскольку они, так или иначе, вложены в блок.

```
div > input { . . . }
```

Будет применен к любому элементу управления, созданному тегом <input>, который непосредственно вложен в блок. (Специальная команда > обозначает требование непосредственной вложенности.) Так, в приведенном ранее примере HTML-кода этот стиль будет применен только к первому тегу <input>, поскольку второй не вложен в блок непосредственно.

```
div > * { . . . }
```

Будет применен к любому тегу, непосредственно вложенному в блок. (Специальная команда * обозначает любой тег.)

```
.button-cont input { . . . }
```

Будет применен к любому элементу управления, созданному тегом <input>, который вложен в элемент интерфейса с привязанным к нему стилевым классом .button-cont.

```
div.button-cont input { . . . }
```

Будет применен к любому элементу управления, созданному тегом <input>, который вложен в блок с привязанным к нему стилевым классом .button-cont.

```
input[type=button] { . . . }
```

Будет применен только к тегу <input>, для атрибута type которого задано значение button, т. е. только к кнопкам.

```
input[type=button]:disabled { . . . }
```

Будет применен только к отключенным кнопкам. (Специальная команда :disabled обозначает отключение элемента управления.)

```
div input[type=button]:disabled { . . . }
```

Будет применен только к отключенным кнопкам, вложенным в блок.

Осталось только очертить круг применения всех четырех разновидностей стилей.

- Стилевые классы применяются, если требуется задать оформление нескольких "избранных" элементов интерфейса.
- Стили переопределения тега применяются при указании оформления всех элементов интерфейса, создаваемых определенным тегом.
- Именованные стили позволяют задать оформление для элементов, присутствующих в интерфейсе приложения в единственном экземпляре.
- Комбинированные стили применяются в различных сложных случаях, например, если требуется задать оформление на основе состояния элементов интерфейса (например, на основании того, отключена ли кнопка).

Таблицы стилей и их привязка

Стили, описывающие оформление для интерфейса Metro-приложения, организуются в виде *таблиц стилей*. Каждая такая таблица стилей может содержать сколько угодно стилей, в том числе и один.

В Metro-программировании рекомендуется применять только так называемые *внешние таблицы стилей*. Они сохраняются в отдельных файлах, имеющих расширение css. Да-да, это те самые, уже знакомые нам файлы, где хранится оформление Metro-приложения!

В самом простом случае приложение имеет в своем составе две таблицы стилей. Первая — задающая базовое оформление, точнее, одну из стандартных тем: "темную" (хранится в файле ui-dark.css) или "светлую" (хранится в файле ui-light.css). По умолчанию к приложению применяется "темная" тема оформления.

Файлы обеих перечисленных ранее таблиц стилей являются частью программного ядра платформы Metro. Они доступны только для чтения, так что править мы их не можем.

Вторая таблица стилей, напротив, создается самим разработчиком данного конкретного приложения и задает для этого приложения уникальное оформление. Она хранится в файле default.css в папке css; этот файл создается Visual Studio и изначально практически пуст (содержит только некоторые заготовки, не задающие никакого реального оформления).

Разумеется, мы можем создать дополнительные таблицы стилей. Это может понадобиться в случае, если мы создаем приложение с достаточно сложным оформлением.

А теперь — важная деталь. Дело в том, что само включение в состав проекта CSS-файлов с таблицами стилей еще не служит для платформы Metro "руководством к действию". Нам потребуется указать в HTML-коде описания интерфейса особые ссылки на файлы, хранящие эти таблицы стилей, — выполнить их *привязку*. Только после этого все стили, определенные в таблице, будут применены к интерфейсу приложения.

Привязка таблицы стилей выполняется помещением в его HTML-код особого тега. Это одинарный служебный тег `<link>`; он вставляется в парный тег `<head>`, рассмотренный нами ранее, и записывается в таком формате:

```
<link href="<ссылка на файл таблицы стилей>" rel="stylesheet" />
```

Сама ссылка на файл с таблицей стилей указывается в качестве атрибута тега `href`. Значение `stylesheet` атрибута тега `rel` сообщает платформе Metro, что выполняется привязка именно таблицы стилей.

Например:

```
<link href="//Microsoft.WinJS.0.6/css/ui-dark.css" rel="stylesheet">
```

Этот тег выполняет привязку таблицы стилей ui-dark.css, задающей "темную" тему базового оформления и входящую в состав программного ядра Metro.

ВНИМАНИЕ!

В ссылках на файлы подобного рода вместо привычного нам символа обратного слеша (\) следует указывать символ прямого слеша (/). В противном случае платформа Metro не сможет найти и загрузить файл.

НА ЗАМЕТКУ

Копии таблиц стилей, описывающих базовое оформление, можно отыскать в папке C:\Program Files\Microsoft SDKs\Windows\v8.0\ExtensionSDKs\Microsoft.WinJS\<*номер версии платформы Metro*>\DesignTime\Debug\Neutral\Microsoft.WinJS<*номер версии платформы Metro*>\css.

```
<link href="/css/default.css" rel="stylesheet" />
```

А этот тег выполняет привязку "рабочей" таблицы стилей default.css, что хранится в папке css.

НА ЗАМЕТКУ

Существуют также *внутренние таблицы стилей*, которые записываются прямо в HTML-коде описания интерфейса. Однако такие таблицы стилей не рекомендуется использовать в Metro-программировании.

Язык HTML также предоставляет возможность записать стиль прямо в теге, к которому он должен применяться *(встроенный стиль)*. Такие стили часто применяются при программном задании параметров оформления (подробнее об этом речь пойдет в *главе 9*).

Объединение стилей. Правила каскадности

А теперь давайте представим себе такую ситуацию. Мы создали в приложении несколько индикаторов прогресса (тег `<progress>`). И при очередном запуске поняли, что ширина некоторых из них слишком велика, чтобы "втиснуться" в разработанный нами интерфейс. Поэтому мы уменьшили ширину таких индикаторов, привязав к ним следующий стилевой класс:

```
.slim-progress
{
    width: 100px;
    background-color: red;
}
```

Заодно мы задали для них красный цвет фона, чтобы сделать заметнее для пользователя. Атрибут стиля `background-color` задает цвет фона, а его значение `red` — красный цвет.

Но если мы откроем таблицу стилей ui-dark.css (где ее можно найти, говорилось ранее), то найдем в ней вот такой стиль переопределения тега `<progress>`:

```
progress
{
    width: 180px;
    height: 6px;
}
```

Выходит, к нашим индикаторам прогресса будут привязаны сразу два стиля, задающие для них разные значения ширины! Как же платформа Metro выкрутится из столь щекотливой ситуации?

С честью! Она просто будет следовать *правилам каскадности*, предназначенными для разрешения подобного рода "конфликтов" различных стилей.

Всего таких правил три. Перечислим их в порядке от важнейшего к менее важному.

Первое, важнейшее, правило гласит: более конкретные стили имеют приоритет перед менее конкретными. Следовательно, параметры оформления, заданные более конкретными стилями, переопределят таковые, указанные в менее конкретных стилях.

Посмотрим на стиль переопределения тега `<progress>`. Он будет применен ко всем индикаторам прогресса без исключения. Напротив, стилевой класс `.slim-progress` будет привязан, во-первых, только к "избранным" индикаторам прогресса, а во-вторых, лично нами. Стало быть, стилевой класс `.slim-progress` в терминологии CSS является более конкретным, чем стиль переопределения тега `<progress>`.

Следовательно, платформа Metro возьмет стиль переопределения тега `<progress>` и наложит на него стилевой класс `.slim-progress` таким образом, чтобы атрибуты, присутствующие в последнем стиле (как более конкретном), заменили те же атрибуты, что записаны в первом стиле (как менее конкретном). Результирующий стиль будет выглядеть так:

```
{
    width: 100px;
    height: 6px;
    background-color: red;
}
```

Его-то платформа Metro и применит к нашим индикаторам прогресса.

А теперь предположим, что мы решили задать для одного из индикаторов зеленый фон. Для этого мы дополнительно привяжем к нему вот такой именованный стиль:

```
#exec-progress
{
    background-color: green;
}
```

Значение `green` задает зеленый цвет фона.

Именованный стиль является более конкретным, чем даже стилевой класс, поскольку может быть привязан только к одному элементу интерфейса во всем приложении. И к данному индикатору прогресса будет применен вот такой результирующий стиль:

```
{
    width: 100px;
    height: 6px;
    background-color: green;
}
```

Что касается комбинированных стилей, то степень конкретности стиля такого рода зависит от количества использованных в его селекторе наименований стилевых классов и именованных стилей и специальных команд языка CSS. В общем, чем больше таких элементов встречается в селекторе комбинированного стиля и чем "весомее" они (так, наименование именованного стиля имеет больший вес, чем наименование стилевого класса), тем более конкретным является данный стиль.

Второе правило касается стилей, относящихся к одной разновидности, и гласит: стили, определенные позже, имеют приоритет над ранее определенными стилями. То есть атрибуты, присутствующие в стиле, чье определение находится в CSS-коде позже, перекроют те же атрибуты, что находятся в стиле, определенном раньше.

Так, если мы запишем подряд два стиля с одним и тем же селектором:

```
progress
{
    width: 180px;
    height: 6px;
}
progress
{
    width: 100px;
}
```

второй стиль будет иметь приоритет. И к индикаторам прогресса будет применен такой результирующий стиль:

```
{
    width: 100px;
    height: 6px;
}
```

Третье правило также касается стилей, относящихся к одной разновидности, но уже определенных в разных таблицах стилей. Оно предписывает платформе Metro отдавать приоритет тем стилям, что определены в таблице стилей, привязанной позднее, перед теми, что хранятся в ранее привязанной таблице стилей.

Это значит, что стили из таблицы стилей default.css в любом случае будут иметь приоритет перед аналогичными им стилями, определенными в таблице стилей ui-dark.css (или ui-light.css). Как мы уже знаем, тег `<link>`, привязывающий вторую таблицу стилей, встречается в HTML-коде позже, чем тег, что привязывает первую таблицу стилей.

Правила каскадности — замечательное подспорье для Metro-программиста. Ему не придется скрупулезно описывать в стиле все параметры оформления для какого-либо элемента интерфейса. Достаточно указать только те атрибуты стиля, что задают уникальные для этого элемента параметры, — остальные будут "унаследованы" от стилей, задающих базовое оформление. Пример этому приведен в начале данного раздела.

Комментарии CSS

Язык CSS также поддерживает создание комментариев. Строки, составляющие комментарий, помещаются между последовательностей символов `/*` и `*/`.

```
/*
Этот стиль будет привязан к кнопкам,
флажкам и полям ввода
*/
```

Комментарии CSS можно использовать и для скрытия фрагментов CSS-кода в целях отладки и эксперимента.

Что дальше?

В этой главе мы изучили основы языков HTML и CSS, на которых описываются, соответственно, интерфейс и оформление Metro-приложения. Впоследствии мы неоднократно будем возвращаться к этим языкам и знакомиться с их тегами и атрибутами, создающими различные элементы интерфейса и описывающими параметры их оформления.

Следующая глава будет посвящена основам языка JavaScript, на котором записывается программная логика Metro-приложения. Этот язык заметно сложнее, чем HTML и CSS. Но ведь мы не боимся сложностей!

ГЛАВА 4

Логика Metro-приложения: основные понятия

В предыдущей главе мы изучали языки HTML и CSS. На языке HTML создается интерфейс Metro-приложения, а на языке CSS описывается его оформление.

Эта глава посвящена языку JavaScript, на котором пишется программная логика Metro-приложения. Программная логика определяет поведение Metro-приложения в ответ на действие пользователя и обеспечивает всю его работу. Если интерфейс и оформление можно назвать лицом приложения, то логика — если так можно выразиться, его душа.

Принцип, на основе которого создается логика, в корне отличается от принципов формирования интерфейса и оформления. Языки HTML и CSS описывают, что мы хотим получить на экране, а как платформа Metro это сделает — не наше дело. Язык JavaScript, напротив, описывает, что должна сделать платформа Metro, чтобы вывести нужный нам результат.

Поздравим себя — в этой главе мы займемся настоящим программированием! (Написание HTML- и CSS-кода к таковому не относится.)

Введение в язык JavaScript

Изучение языка JavaScript удобнее начать с примера. Давайте рассмотрим вот такой фрагмент JavaScript-кода:

```
x = 4;
y = 5;
z = x * y;
```

Больше похоже на набор каких-то формул. Но это не формулы, а *выражения* языка JavaScript; каждое выражение представляет собой описание одного законченного действия, выполняемого приложением.

Разберем этот код по выражениям. Вот первое из них:

```
x = 4;
```

Здесь мы видим число 4. В JavaScript такие числа, а также строки и прочие величины, значения которых никогда не изменяются, называются *константами*. В самом деле, значение числа 4 всегда равно четырем!

Еще мы видим здесь латинскую букву `x`. Это *переменная*, которую можно описать как участок памяти компьютера, имеющий уникальное имя и предназначенный для хранения какой-либо величины — константы или результата вычисления. Наша переменная имеет имя `x`.

Осталось выяснить, что делает символ равенства (`=`), поставленный между переменной и константой. Это *оператор* — команда, выполняющая определенные действия над данными. А если точнее, то символом `=` обозначается *оператор присваивания*. Он помещает значение *(операнд)*, расположенное справа, в переменную, расположенную слева, в нашем случае — значение 4 в переменную `x`. Или, как говорят программисты, присваивает значение 4 переменной `x`.

Каждое выражение JavaScript должно завершаться символом точки с запятой (`;`), обозначающим его конец; отсутствие этого знака вызовет ошибку в работе приложения.

Рассмотрим следующее выражение:

```
y = 5;
```

Оно аналогично первому и присваивает переменной `y` константу 5.

Третье выражение:

```
z = x * y;
```

Здесь мы видим все тот же оператор присваивания, присваивающий что-то переменной `z`. Но что? Результат вычисления произведения значений, хранящихся в переменных `x` и `y`. Вычисление произведения выполняет оператор умножения, который в JavaScript обозначается символом звездочки (`*`). Это *арифметический оператор*.

В результате выполнения приведенного ранее кода в переменной `z` окажется произведение значений 4 и 5 — 20.

Вот еще один пример математического выражения, на этот раз более сложного:

```
y = y1 * y2 + x1 * x2;
```

Оно вычисляется в следующем порядке:

1. Значение переменной `y1` умножается на значение переменной `y2`.
2. Перемножаются значения переменных `x1` и `x2`.
3. Полученные на шагах 1 и 2 произведения складываются (оператор сложения обозначается привычным нам знаком `+`).
4. Полученная сумма присваивается переменной `y`.

Но почему на шаге 2 выполняется умножение `x1` на `x2`, а не сложение произведения `y1` и `y2` с `x1`. Дело в том, что каждый оператор имеет *приоритет* — своего рода но-

мер в очереди их выполнения. Оператор умножения имеет более высокий приоритет, чем оператор сложения, поэтому умножение всегда выполняется перед сложением.

А вот еще одно выражение:

```
x = x + 3;
```

Оно абсолютно правильно с точки зрения JavaScript, хоть и выглядит нелепым. В нем сначала выполняется сложение значения переменной `x` и числа 3, после чего результат сложения снова присваивается переменной `x`.

Типы данных JavaScript

Данные, которые обрабатывает приложение, делятся на несколько разных типов. Их и называют *типами данных*.

- *Строковые* данные (или *строки*) — это последовательности букв, цифр, пробелов, знаков препинания и прочих символов, заключенные в одинарные или двойные кавычки.

  Примеры строк:

  ```
  "JavaScript"
  "1234567"
  'Строковые данные — это последовательности символов.'
  ```

  Строки могут иметь любую длину (определяемую количеством составляющих их символов), ограниченную лишь объемом свободной памяти компьютера. Теоретически существует предел в 2 Гбайт, но вряд ли в нашей практике встретятся столь длинные строки.

  > ***Внимание!***
  >
  > Если нам потребуется указать в строке JavaScript символ обратного слеша (\), то мы должны его продублировать:
  > `s = "c:\\test.txt";`

- *Числовые* данные (или *числа*) — это обычные числа, над которыми можно производить арифметические действия, извлекать из них квадратный корень и вычислять тригонометрические функции. Числа могут быть как целыми, так и дробными; в последнем случае целая и дробная части разделяются точкой (не запятой!).

  Примеры чисел:

  ```
  13756
  454.7873
  0.5635
  ```

  Дробные числа могут быть записаны в экспоненциальной форме:

  *<мантисса>*E*<порядок>*

Вот примеры заданных таким образом чисел (в скобках дано традиционное математическое представление):

`1E-5` (10^{-5})

`8.546E23` ($8{,}546 \cdot 10^{23}$)

Также имеется возможность записи целых чисел в шестнадцатеричном виде. Шестнадцатеричные числа записываются с символами `0x`, помещенными в начало (например, `0x35F`). Отметим, что в JavaScript так можно записывать только целые числа.

- *Логическая* величина может принимать только два значения: true и false — "истина" и "ложь", — обозначаемые ключевыми словами `true` и `false` соответственно. (*Ключевое слово* — это слово, имеющее в языке программирования особое значение.) Логические величины используются, как правило, в условных выражениях (о них речь пойдет позже).
- *null* — особый тип, который может принимать единственное значение `null`. Он применяется в особых случаях, когда требуется указать отсутствие в переменной любого значения.
- *NaN* — особый тип, который может принимать единственное значение `NaN` и обозначает значение, не являющееся числом (например, математическую бесконечность).
- *undefined* — особый тип, опять же, принимающий единственное значение `undefined`. Обозначает, что данная переменная еще не была объявлена (об объявлении переменных будет рассказано чуть позже).

> ***Внимание!***
>
> `undefined` — это не то же самое, что `null`!

Еще два типа данных, поддерживаемые JavaScript и не описанные здесь, мы рассмотрим позже.

Переменные

В начале этой главы мы кое-что узнали о переменных. Сейчас настало время обсудить их детальнее.

Именование переменных

Как мы уже знаем, каждая переменная должна иметь имя, которое однозначно ее идентифицирует. Об именах переменных стоит поговорить подробнее.

Прежде всего, имя переменной должно быть уникальным. Это понятно — ведь в противном случае платформа Metro не сможет однозначно идентифицировать переменную.

В имени переменной могут присутствовать только латинские буквы, цифры и символы подчеркивания (_), причем первый символ имени должен быть либо буквой,

либо символом подчеркивания. Например, `pageAddress`, `_link`, `userName` — правильные имена переменных, а `678vasya` и `Имя пользователя` — неправильные.

Язык JavaScript чувствителен к регистру символов, которыми набраны имена переменных. Это значит, что `pageaddress` и `pageAddress` — разные переменные.

Совпадение имени переменной с ключевым словом языка JavaScript не допускается.

Объявление переменных

Перед использованием переменной обязательно следует выполнить ее *объявление*. Для этого служит *оператор объявления переменной* `var`, после которого указывают имя переменной:

```
var x;
```

Теперь объявленной переменной можно присвоить какое-либо значение:

```
x = 1234;
```

и использовать в коде:

```
y = x * 2 + 10;
```

> ***Внимание!***
> Если обратиться к еще не объявленной переменной, она вернет значение `undefined`.

Значение переменной можно присвоить прямо при ее объявлении:

```
var x = 1234;
```

Можно объявить сразу несколько переменных, разделив их имена запятыми:

```
var x, y, textColor = "black";
```

Пока закончим с переменными. (Впоследствии, при рассмотрении функций, мы к ним еще вернемся.) И займемся операторами JavaScript.

Операторы

Операторов язык JavaScript поддерживает очень много — на все случаи жизни. Их можно разделить на несколько групп.

Арифметические операторы

Арифметические операторы служат для выполнения арифметических действий над числами. Все арифметические операторы, поддерживаемые JavaScript, перечислены в табл. 4.1.

Арифметические операторы делятся на две группы: *унарные* и *бинарные*. Унарные операторы выполняются над одним операндом; к ним относятся операторы смены знака, инкремента и декремента. Унарный оператор извлекает из переменной зна-

Таблица 4.1. *Арифметические операторы*

Оператор	Описание
-	Смена знака числа
+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Взятие остатка от деления
++	*Инкремент* (увеличение на единицу)
--	*Декремент* (уменьшение на единицу)

чение, изменяет его и снова помещает в ту же переменную. Приведем пример выражения с унарным оператором:

```
++r;
```

При выполнении этого выражения в переменной `r` окажется ее значение, увеличенное на единицу. А если записать вот так:

```
s = ++r;
```

то значение `r`, увеличенное на единицу, будет помещено еще и в переменную `s`.

Операторы инкремента и декремента можно ставить как перед операндом, так и после него. Если оператор инкремента стоит перед операндом, то значение операнда сначала увеличивается на единицу, а уже потом используется в дальнейших вычислениях. Если же оператор инкремента стоит после операнда, то его значение сначала вычисляется, а уже потом увеличивается на единицу. Точно так же ведет себя оператор декремента.

Бинарные операторы всегда имеют два операнда и помещают результат в третью переменную. Вот примеры выражений с бинарными операторами:

```
l = r * 3.14;
f = e / 2;
x = x + t / 3;
```

Оператор объединения строк

Оператор объединения строк `+` позволяет соединить две строки в одну. Например, сценарий:

```
s1 = "Java";
s2 = "Script";
s = s1 + s2;
```

поместит в переменную `s` строку `"JavaScript"`.

Операторы присваивания

Оператор присваивания = нам уже знаком. Его еще называют оператором *простого присваивания*, поскольку он просто присваивает переменной новое значение:

```
a = 2;
b = c = 3;
```

Второе выражение в приведенном примере выполняет присваивание значения 3 сразу двум переменным — `b` и `c`.

JavaScript также поддерживает *операторы сложного присваивания*, позволяющие выполнять присваивание одновременно с другой операцией:

```
a = a + b;
a += b;
```

Два этих выражения эквивалентны по результату. Просто во втором указан оператор сложного присваивания +=.

Все операторы сложного присваивания, поддерживаемые JavaScript, и их эквиваленты приведены в табл. 4.2.

Таблица 4.2. *Операторы сложного присваивания*

Оператор	Эквивалентное выражение
`a += b;`	`a = a + b;`
`a -= b;`	`a = a – b;`
`a *= b;`	`a = a * b;`
`a /= b;`	`a = a / b;`
`a %= b;`	`a = a % b;`
`a <<= b;`	`a = a << b;`
`a >>= b;`	`a = a >> b;`
`a >>>= b;`	`a = a >>> b;`
`a &= b;`	`a = a & b;`
`a ^= b;`	`a = a ^ b;`
`a |= b;`	`a = a | b;`

Операторы сравнения

Операторы сравнения сравнивают два операнда согласно определенному условию и выдают (или, как говорят программисты, возвращают) логическое значение. Если условие сравнения выполняется, возвращается значение `true`, если не выполняется — `false`.

Все поддерживаемые JavaScript операторы сравнения приведены в табл. 4.3.

Таблица 4.3. *Операторы сравнения*

Оператор	Описание	Оператор	Описание
<	Меньше	>=	Больше или равно
>	Больше	!=	Не равно
==	Равно	===	Строго равно
<=	Меньше или равно	!==	Строго не равно

```
a1 = 2 < 3;
a2 = -4 > 0;
a3 = r < t;
```

Переменная `a1` получит значение `true` (2 меньше 3), переменная `a2` — значение `false` (число –4 по определению не может быть больше нуля), а значение переменной `a3` будет зависеть от значений переменных `r` и `t`.

Можно сравнивать не только числа, но и строки:

```
a = "JavaScript" != "Java";
```

Переменная `a` получит значение `true`, т. к. строки `"JavaScript"` и `"Java"` не равны.

На двух последних операторах из табл. 4.3 — "строго равно" и "строго не равно" — нужно остановиться подробнее. Это операторы так называемого *строгого сравнения*. Обычные операторы "равно" и "не равно", если встречают операнды разных типов, пытаются преобразовать их к одному типу (о преобразованиях типов см. далее в этой главе). Операторы строгого равенства и строгого неравенства такого преобразования не делают, а в случае несовпадения типов операндов всегда возвращают `false`.

```
a1 = 2 == "2";
a2 = 2 === "2";
```

Переменная `a1` получит значение `true`, т. к. в процессе вычисления строка `"2"` будет преобразована в число 2, и условие сравнения выполнится. Но переменная `a2` получит значение `false`, т. к. сравниваемые операнды принадлежат разным типам.

Логические операторы

Логические операторы выполняют действия над логическими значениями. Все они приведены в табл. 4.4. А в табл. 4.5 и 4.6 показаны результаты выполнения этих операторов.

Таблица 4.4. *Логические операторы*

Оператор	Описание
!	НЕ (логическая инверсия)
&&	И (логическое умножение)
||	ИЛИ (логическое сложение)

Таблица 4.5. *Результаты выполнения операторов И и ИЛИ*

Операнд 1	Операнд 2	&& (И)	|| (ИЛИ)
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Таблица 4.6. *Результаты выполнения оператора НЕ*

Операнд	! (НЕ)
true	false
false	true

Основная область применения логических операторов — выражения сравнения (о них см. далее в этой главе). Приведем примеры таких выражений:

```
a = (b > 0) && (c + 1 != d);
flag = !(status = 0);
```

Оператор получения типа *typeof*

Оператор получения типа `typeof` возвращает строку, описывающую тип данных операнда. Операнд, тип которого нужно узнать, помещается после этого оператора и берется в круглые скобки.

```
s = typeof("str");
```

В результате выполнения этого выражения в переменной `s` окажется строка `"string"`, обозначающая строковый тип.

Все значения, которые может вернуть оператор `typeof`, перечислены в табл. 4.7.

Таблица 4.7. *Значения, возвращаемые оператором* `typeof`

Тип данных	Возвращаемая строка
Строковый	`"string"`
Числовой	`"number"`
Логический	`"boolean"`
Объектный (см. далее)	`"object"`
Функциональный (см. далее)	`"function"`
null	`"null"`
undefined	`"undefined"`

Совместимость и преобразование типов данных

Настала пора рассмотреть еще два важных вопроса: *совместимость* типов данных и *преобразование* одного типа к другому.

Что получится, если сложить два числовых значения? Правильно — еще одно числовое значение. А если сложить число и строку? Трудно сказать... Тут JavaScript сталкивается с проблемой несовместимости типов данных и пытается сделать эти типы совместимыми, преобразуя один из них к другому. Сначала он пытается преобразовать строку в число и, если это удается, выполняет сложение. В случае неудачи число будет преобразовано в строку, и две полученные строки будут объединены. Например, в результате выполнения такого кода:

```
var a, b, c, d, e, f;
a = 11;
b = "12";
c = a + b;
d = "JavaScript";
e = 2;
f = d + e;
```

значение переменной `b` при сложении с переменной `a` будет преобразовано в числовой тип; таким образом, переменная `c` будет содержать значение 23. Но так как значение переменной `d` нельзя преобразовать в число, значение `e` будет преобразовано в строку, и результат — значение `f` — станет равным `"JavaScript2"`.

Логические величины преобразуются либо в числовые, либо в строковые, в зависимости от конкретного случая. Значение `true` будет преобразовано в число 1 или строку `"1"`, а значение `false` — в 0 или `"0"`. И наоборот, число 1 будет преобразовано в значение `true`, а число 0 — в значение `false`. Также в `false` будут преобразованы значения `null` и `undefined`.

Видно, что JavaScript изо всех сил пытается правильно выполнить даже некорректно написанные выражения. Иногда это получается, но чаще все работает не так, как планировалось, и, в конце концов, выполнение приложения прерывается в связи с обнаружением ошибки совсем в другом его месте, на абсолютно верном операторе. Поэтому лучше не допускать подобных казусов.

Приоритет операторов

Последний вопрос, который мы здесь разберем, — приоритет операторов. Как мы помним, приоритет влияет на порядок, в котором выполняются операторы в выражении.

Пусть имеется следующее выражение:

```
a = b + c – 10;
```

В этом случае сначала к значению переменной `b` будет прибавлено значение `c`, а потом из суммы будет вычтено 10. Операторы этого выражения имеют одинаковый приоритет и поэтому выполняются строго слева направо.

Теперь рассмотрим такое выражение:

```
a = b + c * 10;
```

Здесь сначала будет выполнено умножение значения `c` на 10, а уже потом к полученному произведению будет прибавлено значение `b`. Дело в том, что оператор умножения имеет больший приоритет, чем оператор сложения.

Принцип выполнения всех операторов ясен: сначала выполняются операторы с более высоким приоритетом, а уже потом — операторы с более низким. Операторы с одинаковым приоритетом выполняются в порядке их следования — слева направо.

Самый низкий приоритет у операторов присваивания. Вот почему сначала вычисляется само выражение, а потом его результат присваивается переменной.

В табл. 4.8 перечислены все изученные нами операторы в порядке убывания их приоритетов.

Таблица 4.8. *Приоритет операторов (в порядке убывания)*

Операторы	**Описание**
`++`, `--`, `-`, `!`, `typeof`	Инкремент, декремент, смена знака, логическое НЕ, получение типа
`*`, `/`, `%`	Умножение, деление, взятие остатка
`+`, `-`	Сложение, объединение строк, вычитание
`<`, `>`, `<=`, `>=`	Операторы сравнения
`==`, `!=`, `===`, `!==`	
`&&`	Логическое И
`||`	Логическое ИЛИ
`?`	Условный оператор (см. далее)
`=`, *<оператор>*`=`	Присваивание, простое и сложное

Но что делать, если нам нужно нарушить обычный порядок выполнения операторов? Воспользуемся круглыми скобками. При такой записи заключенные в скобки операторы выполняются первыми:

```
a = (b + c) * 10;
```

Здесь сначала будет выполнено сложение значений переменных `b` и `c`, а потом получившаяся сумма будет умножена на 10.

Операторы, заключенные в скобки, также подчиняются приоритету. Поэтому часто используются многократно вложенные скобки:

```
a = ((b + c) * 10 — d) / 2 + 9;
```

Здесь операторы будут выполнены в такой последовательности:

1. Сложение `b` и `c`.
2. Умножение полученной суммы на 10.

3. Вычитание `d` из произведения.
4. Деление разности на 2.
5. Прибавление 9 к частному.

Если удалить скобки:

```
a = b + c * 10 – d / 2 + 9;
```

то порядок выполнения операторов будет таким:

1. Умножение `c` на 10.
2. Деление `d` на 2.
3. Сложение `b` и произведения `c` и 10.
4. Вычитание из полученной суммы частного от деления `d` на 2.
5. Прибавление 9 к полученной разности.

Получается совсем другой результат, не так ли?

Сложные выражения JavaScript

Сложные выражения получили свое название благодаря тому, что все они составлены из нескольких простых выражений. Сложные выражения служат для особых целей — в основном, для управления процессом выполнения содержащихся в них простых выражений.

Блоки

JavaScript позволяет нам объединить несколько выражений в одно. Такое выражение называется *блочным выражением* или просто *блоком* (не путать с блоками — элементами интерфейса!). Составляющие его выражения заключают в фигурные скобки, например:

```
{
  b = "12";
  c = a – b;
}
```

Как правило, блоки не существуют сами по себе. Чаще всего они входят в состав других сложных выражений.

Условные выражения

Условное выражение позволяет нам выполнить одно из двух входящих в него выражений в зависимости от истинности или ложности какого-либо *условия*. В качестве условия используется значение логической переменной или результат вычисления логического выражения.

Вот формат, в котором записывается условное выражение:

```
if (<условие>)
  <блок "то">
else
  <блок "иначе">
```

Для написания условных выражений предусмотрены особые ключевые слова `if` и `else`. Отметим, что *условие* всегда записывается в круглых скобках.

Существует также другая, "вырожденная" разновидность условного выражения, содержащая только одно выражение, которое выполняется при выполнении условия и пропускается, если условие не выполнено:

```
if (<условие>)
  <блок "то">
```

Если *условие* имеет значение `true`, то выполняется *блок "то"*. Если же *условие* имеет значение `false`, то выполняется *блок "иначе"* (если он присутствует в условном выражении). А если *блок "иначе"* отсутствует, выполняется следующее выражение.

> ***Внимание!***
> Значения `null` или `undefined` преобразуются в `false`. Не забываем об этом.

Рассмотрим несколько примеров.

```
if (x == 1) {
  a = "Единица";
  b = 1;
} else {
  a = "Не единица";
  b = 22222;
}
```

Здесь мы сравниваем значение переменной `x` с единицей и в зависимости от результатов сравнения присваиваем переменным `f` и `h` разные значения.

Условие может быть довольно сложным:

```
if ((x == 1) && (y > 10)) {
  f = 3;
} else {
  f = 33;
}
```

Здесь мы использовали сложное условие, возвращающее значение `true` в случае, если значение переменной `x` равно 1 *и* значение переменной `y` больше 10.

Условный оператор ?

Если условное выражение совсем простое, мы можем записать его немного по-другому — воспользовавшись *условным оператором* `?`:

```
<условие> ? <выражение "то"> : <выражение "иначе">;
```

Достоинство этого оператора в том, что он может быть частью выражения. Например:

```
f = (x == 1 && y > 10) ? 3 : 33;
```

Фактически мы записали условное выражение из предыдущего примера, но в виде простого выражения. Компактность кода налицо. Недостаток же оператора `?` в том, что с его помощью можно записывать только самые простые условные выражения.

Приоритет условного оператора один из самых низких. Приоритет ниже него имеют только операторы присваивания.

Выражения выбора

Выражение выбора — это фактически несколько условных выражений, объединенных в одном. Формат его записи таков:

```
switch (<исходное выражение>) {
  case <значение 1> :
    <блок 1>
    break;
  case <значение 2> :
    <блок 2>
    break;
  <... другие секции case>
  [default :
    <блок, исполняемый для остальных значений>]
}
```

В выражениях выбора присутствуют ключевые слова `switch`, `case` и `default`.

Результат вычисления *исходного выражения* последовательно сравнивается со *значением 1*, *значением 2* и т. д. и, если такое сравнение прошло успешно, выполняется соответствующий блок кода (*блок 1*, *блок 2* и т. д.). Если же ни одно сравнение не увенчалось успехом, выполняется блок кода, находящийся в секции `default` (если, конечно, она присутствует).

Вот пример выражения выбора:

```
switch (a) {
  case 1 :
    out = "Единица";
    break;
  case 2 :
    out = "Двойка";
    break;
  case 3 :
    out = "Тройка";
    break;
  default :
    out = "Другое число";
}
```

Если переменная `a` содержит значение 1, переменная `out` получит значение `"Единица"`, если 2 — значение `"Двойка"`, а если 3 — значение `"Тройка"`. Если же переменная `a` содержит какое-то другое значение, переменная `out` получит значение `"Другое число"`.

Может случиться так, что нам понадобится выполнить один и тот же блок кода сразу для нескольких значений. В этом случае мы используем такой формат:

```
switch (<исходное выражение>) {
  . . .
  case <значение 1> :
  case <значение 2> :
  <... другие значения>
    <блок>
    break;
  . . .
```

В этом случае *блок* будет выполнен, если исходное выражение совпадает со *значением 1*, *значением 2* и всеми последующими.

```
switch (a) {
  case 1 :
  case 2 :
    out = "Единица или двойка";
    break;
  case 3 :
    out = "Тройка";
    break;
}
```

Здесь, если переменная `a` хранит значение 1 или 2, переменная `out` получит значение `"Единица или двойка"`.

Циклы

Циклы — это особые выражения, позволяющие выполнить один и тот же блок кода несколько раз. JavaScript предлагает программистам несколько разновидностей циклов, которые мы сейчас рассмотрим.

Цикл со счетчиком

Цикл со счетчиком удобен, если нам нужно выполнить какой-то код строго определенное число раз. Вероятно, это наиболее распространенный вид цикла.

Цикл со счетчиком записывается в таком формате:

```
for (<выражение инициализации>; <условие>; <приращение>)
  <тело цикла>
```

Здесь используется ключевое слово `for`. Поэтому такие циклы часто называют "циклами for".

Выражение инициализации выполняется самым первым и всего один раз. Оно присваивает особой переменной, называемой *счетчиком цикла*, некое начальное значение (обычно 1). Счетчик цикла подсчитывает, сколько раз было выполнено *тело цикла* — собственно код, который нужно выполнить определенное количество раз.

Следующий шаг — проверка *условия*. Оно определяет момент, когда выполнение цикла прервется и начнет выполняться следующий за ним код. Как правило, *условие* сравнивает значение счетчика цикла с его граничным значением. Если *условие* возвращает `true`, выполняется *тело цикла*, в противном случае цикл завершается и начинается выполнение кода, следующего за циклом.

После прохода *тела цикла* выполняется выражение *приращения*, изменяющее значение счетчика. Это выражение обычно инкрементирует счетчик (увеличивает его значение на единицу). Далее снова проверяется *условие*, выполняется *тело цикла*, *приращение* и т. д., пока *условие* не станет равно `false`.

Пример цикла со счетчиком:

```
for (i = 1; i < 11; i++) {
  a += 3;
  b = i * 2 + 1;
}
```

Этот цикл будет выполнен 10 раз. Мы присваиваем счетчику `i` начальное значение 1 и инкрементируем его после каждого выполнения тела цикла. Цикл перестанет выполняться, когда значение счетчика увеличится до 11, и условие цикла станет ложным.

Счетчик цикла можно использовать в одном из выражений тела цикла, как это сделали мы. В нашем случае счетчик `i` будет содержать последовательно возрастающие значения от 1 до 10, которые используются в вычислениях.

Приведем еще два примера цикла со счетчиком:

```
for (i = 10; i > 0; i--) {
  a += 3;
  b = i * 2 + 1;
}
```

Здесь значение счетчика декрементируется. Начальное его значение равно 10. Цикл выполнится 10 раз и завершится, когда счетчик `i` будет содержать 0; при этом значения последнего будут последовательно уменьшаться от 10 до 1.

```
for (i = 2; i < 21; i += 2){
  b = i * 2 + 1;
}
```

А в этом примере начальное значение счетчика равно 2, а конечное — 21, но цикл выполнится, опять же, 10 раз. А все потому, что значение счетчика увеличивается на 2 и последовательно принимает значения 2, 4, 6, ..., 20.

Цикл с постусловием

Цикл с постусловием, как и цикл со счетчиком, выполняется до тех пор, пока остается истинным условие цикла. Причем условие проверяется не до, а после выполнения тела цикла, отчего цикл с постусловием и получил свое название. Такой цикл выполнится хотя бы один раз, даже если его условие с самого начала ложно.

Формат цикла с постусловием:

```
do
  <тело цикла>
while (<условие>);
```

Для задания цикла с постусловием предусмотрены ключевые слова `do` и `while`, поэтому такие циклы часто называют "циклами do-while".

Вот пример цикла с постусловием:

```
do {
  a = a * i + 2;
  ++i;
} while (a < 100);
```

А вот еще один пример:

```
var a = 0, i = 1;
do {
  a = a * i + 2;
  ++i;
} while (i < 20);
```

Хотя здесь удобнее был бы уже знакомый нам и специально предназначенный для таких случаев цикл со счетчиком.

Цикл с предусловием

Цикл с предусловием отличается от цикла с постусловием тем, что условие проверяется перед выполнением тела цикла. Так что, если оно (условие) изначально ложно, цикл не выполнится ни разу:

```
while (<условие>)
  <тело цикла>
```

Для создания цикла с предусловием предусмотрено ключевое слово `while`. Поэтому такие циклы называют еще "циклами while".

Пример цикла с предусловием:

```
while (a < 100) {
  a = a * i + 2;
  ++i;
}
```

Прерывание и перезапуск цикла

Иногда бывает нужно прервать выполнение цикла. Для этого JavaScript предоставляет Metro-программистам операторы `break` и `continue`.

Оператор прерывания `break` позволяет *прервать* выполнение цикла и перейти к следующему за ним выражению:

```
while (a < 100) {
  a = a * i + 2;
  if (a > 50) break;
  ++i;
}
```

В этом примере мы прерываем выполнение цикла, если значение переменной `a` превысит 50.

Оператор перезапуска `continue` позволяет *перезапустить* цикл, т. е. оставить невыполненными все последующие выражения, входящие в тело цикла, и запустить выполнение цикла с самого его начала: проверка условия, выполнение приращения и тела и т. д.

Пример:

```
while (a < 100) {
  i = ++i;
  if (i > 9 && i < 11) continue;
  a = a * i + 2;
}
```

Здесь мы пропускаем выражение вычисления `a` для всех значений `i` от 10 до 20.

Функции

Функция — это изолированный фрагмент логики нашего приложения, оформленный особым образом. Обычно в виде функций оформляется код, выполняемый сразу в нескольких местах логики приложения; вместо того, чтобы помещать этот код в нужные места целиком, мы можем просто вставить туда ссылку на него (выполнить *вызов* функции). Также функции используются в специальных случаях, о которых мы поговорим в *главе 5*.

Объявление функций

Прежде чем функция будет использована где-то в коде логики, ее нужно объявить. Функцию объявляют с помощью ключевого слова `function`:

```
function <имя функции>([<список параметров, разделенных запятыми>])
  <тело функции>
```

Уникальное *имя функции* однозначно идентифицирует функцию и позволяет впоследствии сослаться на нее. Для имен функций действуют те же правила, что и для имен переменных.

Функция может обрабатывать какие-то данные, которые передаются ей в качестве *параметров*. Функция может принимать сколько угодно параметров или не принимать их вовсе.

`Список параметров` функции представляет собой перечень переменных, в которые будут помещены параметры и которые будут доступны внутри тела функции. Мы можем придумать для этих переменных любые имена — все равно "вне" функции они существовать не будут. Это так называемые *формальные параметры* функции.

`Список параметров` функции помещают в круглые скобки и ставят после ее имени; сами параметры отделяют друг от друга запятыми. Если функция не требует параметров, следует указать пустые скобки — это обязательно.

Тело функции — это собственно фрагмент логики, ради которого эта функция и была создана. Он оформляется в виде блока.

В пределах *`тела функции`* над полученными ею параметрами (если они есть) и другими данными выполняются некоторые действия и, возможно, вырабатывается результат. *Оператор возврата* `return` возвращает результат из функции в выражение, из которого она была вызвана:

```
return <переменная или выражение>;
```

Здесь *`переменная`* должна содержать возвращаемое значение, а *`выражение`* должно его вычислять.

Пример объявления функции:

```
function divide(a, b) {
  var c;
  c = a / b;
  return c;
}
```

Данная функция принимает два параметра — `a` и `b`, — после чего делит `a` на `b` и возвращает частное от этого деления.

Эту функцию можно записать компактнее:

```
function divide(a, b) {
  return a / b;
}
```

Или даже так, в одну строку:

```
function divide(a, b) { return a / b; }
```

JavaScript позволяет нам создавать так называемые *необязательные параметры* функций — параметры, которые при вызове можно не указывать; в таком случае они примут некоторое значение по умолчанию. Вот пример функции с необязательным параметром `b`, имеющим значение по умолчанию 2:

```
function divide(a, b) {
  if (typeof(b) == "undefined") {
    var b = 2;
  }
```

```
  return a / b;
}
```

Понятно, что мы должны как-то выяснить, был ли при вызове функции указан параметр `b`. Для этого мы используем оператор получения типа `typeof`. Если параметр `b` не был указан, данный оператор вернет строку `"undefined"`; тогда мы создадим переменную с именем `b`, как и у необязательного параметра, и присвоим ей число 2 — значение этого параметра по умолчанию, — которое и будет использовано в теле функции. Если возвращенное оператором значение иное, значит, параметр `b` был указан при вызове, и мы используем значение, которое было передано с ним.

Осталось сказать, что все функции JavaScript относятся к так называемому *функциональному* типу данных.

Локальные переменные

В описанном ранее примере мы создали в теле функции переменную, заменяющую опущенный при вызове функции параметр. Эта переменная будет доступна только внутри тела функции, в которой была объявлена. После завершения выполнения функции эта переменная будет уничтожена. Переменные подобного рода называются *локальными*.

Разумеется, любая функция может обращаться к любой переменной, объявленной вне ее тела *(глобальной переменной)*. Однако при этом нужно помнить об одной вещи. Если существуют две переменные с одинаковыми именами: одна — глобальная, другая — локальная, то при обращении по этому имени будет получен доступ *к локальной переменной*. Одноименная глобальная переменная будет "замаскирована" своей локальной "тезкой".

Вызов функций

После объявления функции можно выполнить ее вызов из любого места кода приложения. Формат вызова функции таков:

<имя функции>([*<список фактических параметров, разделенных запятыми>*])

Здесь указывается `имя` нужной функции и в круглых скобках перечисляются *фактические* параметры — реальные значения, которые следует передать в функцию для обработки. Функция вернет результат, который можно присвоить переменной или использовать в выражении.

> ***Внимание!***
>
> При вызове функции подставляйте именно фактические параметры, а не формальные, указанные в объявлении функции.

Вот пример вызова объявленной нами ранее функции `divide()`:

```
d = divide(3, 2);
```

Здесь мы подставили в выражение вызова функции фактические параметры — константы 3 и 2.

А здесь мы выполняем вызов функции с переменными в качестве фактических параметров:

```
s = 4 * divide(x, r) + y;
```

Если функция имеет необязательные параметры и нас удовлетворяют их значения по умолчанию, мы можем при вызове не указывать эти параметры, все или некоторые из них. Например, функцию `divide()` со вторым необязательным параметром мы можем вызвать так:

```
s = divide(4);
```

Тогда в переменной `s` окажется число 2 — результат деления 4 (значение первого параметра) на 2 (значение второго, необязательного, параметра по умолчанию).

Если функция не возвращает результата, то ее вызывают так:

```
initVars(1, 2, 3, 6);
```

Более того, так можно вызвать и функцию, возвращающую результат, который в этом случае будет отброшен. Такой способ вызова может быть полезен, если результат, возвращаемый функцией, не нужен для работы приложения.

Если функция не принимает параметров, при ее вызове все равно нужно указать пустые скобки, иначе возникнет ошибка:

```
s = computeValue();
```

Функции могут вызывать друг друга. Вот пример:

```
function cmp(c, d, e) {
  var f;
  f = divide(c, d) + e;
  return f;
}
```

Здесь мы использовали в функции `cmp()` вызов объявленной ранее функции `divide()`.

Функция как значение. Анонимные функции

Язык JavaScript позволяет оперировать функцией как обычным значением. В частности, мы можем присвоить функцию переменной или передать ее в качестве параметра другой функции. Например:

```
someFunc = cmp;
otherFunc(2, 87, cmp);
```

Здесь мы сначала присвоили переменной `someFunc` объявленную ранее функцию `cmp`, а потом указали ее в качестве третьего параметра функции `otherFunc()`. Заметим, что в этом случае указывается только имя функции без скобок и параметров.

Впоследствии мы можем вызывать данную функцию, обратившись к переменной, которой она была присвоена:

```
c = someFunc(1, 2, 3);
```

А можно сделать и так:

```
someFunc = function(c, d, e) {
  var f;
  f = divide(c, d) + e;
  return f;
}
```

Здесь мы объявили функцию и сразу же присвоили ее переменной. Причем имени объявляемой функции мы не указали — в данном случае оно не нужно.

```
otherFunc(2, 87, function(c, d, e) {
  var f;
  f = divide(c, d) + e;
  return f;
});
```

А здесь мы объявили функцию и сразу же передали ее функции `otherFunc()` в качестве третьего параметра. И здесь имя объявленной функции мы не указали из-за его ненадобности.

Такие функции, не имеющие имени, называются *анонимными*. Впоследствии мы будем часто их использовать.

Встроенные функции

Существует ряд полезных функций, которые уже объявлены в самой платформе Metro. Они называются *встроенными функциями*.

К таким функциям относятся, в частности, `parseInt()` и `parseFloat()`. Обе принимают в качестве единственного параметра строку, содержащую число, и обе возвращают это число уже в числовом формате. Только первая функция возвращает это число в целочисленной форме, а вторая — в форме числа с плавающей точкой. Если же строку не удается преобразовать в число (например, она содержит только буквы), возвращается значение `NaN`.

```
var s1 = "1234";
var s2 = "1234.56";
var i = parseInt(s1);
var f = parseFloat(s2);
```

После выполнения этого кода в переменной `i` окажется число 1234, а в переменной `f` — число 1234,56.

Массивы

Массив — это пронумерованный набор переменных *(элементов)*, фактически хранящийся в одной переменной. Доступ к отдельному элементу массива выполняется по его порядковому номеру, называемому *индексом*. А общее число элементов массива называется его *размером*.

Внимание!

Нумерация элементов массива начинается с нуля.

Чтобы создать массив, нужно просто присвоить любой переменной список его элементов, разделенных запятыми и заключенных в квадратные скобки:

```
var someArray = [1, 2, 3, 4];
```

Здесь мы создали массив, содержащий четыре элемента, и присвоили его переменной `someArray`. После этого мы можем получить доступ к любому из элементов массива по индексу нужного элемента, указав его после имени переменной массива в квадратных скобках:

```
a = someArray[2];
```

В данном примере мы получили доступ к третьему элементу массива. (Нумерация элементов массива начинается *с нуля* — помните об этом!)

JavaScript также позволяет нам создать массив, вообще не содержащий элементов (пустой массив). Для этого достаточно присвоить любой переменной "пустые" квадратные скобки:

```
var someArray = [];
```

Разумеется, впоследствии мы можем и даже должны наполнить этот массив элементами:

```
someArray[0] = 1;
someArray[1] = 2;
someArray[2] = 3;
```

Определять сразу все элементы массива необязательно:

```
someArray2 = [1, 2, , 4];
```

Здесь мы пропустили третий элемент массива, и он остался неопределенным (т. е. будет содержать значение `undefined`).

При необходимости мы легко сможем добавить к массиву еще один элемент, просто присвоив ему требуемое значение:

```
someArray[4] = 9;
```

При этом будет создан новый, пятый по счету, элемент массива с индексом 4 и значением 9.

Можно даже сделать так:

```
someArray[7] = 9;
```

В этом случае будут созданы четыре новых элемента, и восьмой элемент получит значение 9. Пятый, шестой и седьмой останутся неопределенными (`undefined`).

Мы можем присвоить любому элементу массива другой массив (или, как говорят опытные программисты, создать *вложенный массив*):

```
someArray[2] = ["n1", "n2", "n3"];
someArray[3] = [];
```

```
someArray[3][0] = "n4";
someArray[3][1] = "n5";
```

После этого можно получить доступ к любому элементу вложенного массива, указав последовательно оба индекса: сначала — индекс во "внешнем" массиве, потом — индекс во вложенном:

```
str = someArray[2][1];
```

Переменная `str` получит в качестве значения строку, содержащуюся во втором элементе вложенного массива, — `"n2"`.

Ранее говорилось, что доступ к элементам массива выполняется по числовому индексу. Но JavaScript позволяет создавать и массивы, элементы которых имеют строковые индексы (*ассоциативные массивы*, или *хэши*).

Пример:

```
var hash;
hash["trailer"] = "/movie_trailer.avi";
hash["movie"] = "/movie.mp4";
. . .
movieFile = hash["movie"];
```

Массивы идеально подходят в тех случаях, когда нужно хранить в одном месте упорядоченный набор данных. Ведь массив фактически представляет собой одну переменную.

Ссылки

Осталось рассмотреть еще один момент, связанный с организацией доступа к данным. Это так называемые *ссылки* — своего рода указатели на массивы и экземпляры объектов (о них будет рассказано далее), хранящиеся в соответствующих им переменных.

Когда мы создаем массив, платформа Metro выделяет под него область памяти и помещает в нее значения элементов этого массива. Но в переменную, которой мы присвоили вновь созданный массив, помещается не сама эта область памяти, а ссылка на нее. Если теперь обратиться к какому-либо элементу этого массива, платформа Metro извлечет из переменной ссылку, по ней найдет нужную область памяти, вычислит местонахождение нужного элемента и вернет его значение.

Далее, если мы присвоим значение переменной массива другой переменной, будет выполнено присваивание именно ссылки. В результате получатся две переменные, ссылающиеся на одну область памяти, хранящую сам этот массив.

Рассмотрим такой пример:

```
var myArray = ["trailer", "poster", "movie"];
var newArray = myArray;
```

Здесь создается массив `myArray` с тремя элементами, который далее присваивается переменной `newArray` (при этом данная переменная получает ссылку на массив).

Если потом мы присвоим новое значение первому элементу массива `myArray`:

```
myArray[0] = "credits";
```

и обратимся к нему через переменную `newArray`:

```
s = newArray[0];
```

то в переменной `s` окажется строка `"credits"` — новое значение первого элемента этого массива. Фактически переменные `myArray` и `newArray` указывают на один и тот же массив.

> ***ВНИМАНИЕ!***
>
> В дальнейшем для простоты мы будем считать, что в переменной хранится не ссылка на массив (экземпляр объекта), а сам массив (экземпляр объекта). Если нужно будет специально указать, что переменная хранит ссылку, мы так и сделаем.

Переменная, хранящая ссылку на массив (объект), содержит данные *объектного типа*. Это последний тип данных, поддерживаемый JavaScript, который мы здесь рассмотрим.

> ***НА ЗАМЕТКУ***
>
> Ранее мы узнали, что можем присвоить функцию переменной. Так вот, фактически переменная, которой была присвоена функция, также хранит ссылку на нее.

Объекты и экземпляры объектов

Итак, мы познакомились с типами данных, переменными, константами, операторами, простыми и сложными выражениями, функциями и массивами. Но это была, так сказать, присказка, а сказка будет впереди. Настала пора узнать о самых сложных структурах данных JavaScript — объектах и их экземплярах.

Понятия объекта и экземпляра объекта

Все типы данных — строки, числа, логические величины, — с которыми мы до этого имели дело, были весьма просты и позволяли хранить всего одно значение. Их так и называют — *простые типы данных*.

Однако JavaScript предоставляет нам и *сложные типы данных*. Сущность, относящаяся к такому типу, может хранить сразу несколько значений. Один из примеров сложного типа данных — уже знакомые нам массивы.

Другой пример сложного типа данных — объекты. *Объект* — это сложная сущность, способная хранить сразу несколько значений разных типов. Для этого объект определяет набор своего рода внутренних переменных, называемых *свойствами*; такое свойство может хранить одну сущность, относящуюся к простому или сложному типу данных. Так, одно свойство объекта может хранить строки, другое — числа, третье — массивы.

А еще объект может содержать набор внутренних функций, называемых *методами*. Методы могут обрабатывать данные, хранящиеся в свойствах или полученные "извне", менять значения свойств и возвращать результат, как обычные функции.

Объект — это всего лишь тип данных. Сущность же, хранящая реальные данные и созданная на основе этого объекта, называется его *экземпляром*. Точно так же, как строка `"JavaScript"` — экземпляр строкового типа данных, хранящий реальную строку.

Экземпляры объектов имеют такую же природу, как и массивы. Сам экземпляр объекта находится где-то в памяти компьютера, а в переменной хранится ссылка на него. При присваивании ее значения другой переменной выполняется присваивание именно ссылки. Не забываем об этом.

Каждый объект должен иметь уникальное имя, по которому к нему можно обратиться, чтобы создать его экземпляр. К именам объектов предъявляют те же требования, что и к именам переменных и функций.

Объекты — невероятно мощное средство объединить данные (свойства) и средства их обработки (методы) воедино. Так, объект, представляющий видеопроигрыватель, объединяет параметры этого видеопроигрывателя, хранящиеся в различных свойствах, и инструменты для манипуляции им, предоставляемые соответствующими методами. Нам не придется "раскидывать" параметры видеопроигрывателя по десяткам переменных и пользоваться для работы с ним массой отдельных функций — все это сведено в один объект.

Экземпляры одного объекта — отдельные сущности, не влияющие друг на друга. Мы можем работать с одним экземпляром объекта, а другие экземпляры того же самого объекта останутся неизменными. Так, мы можем изменить параметры одного видеопроигрывателя, присвоив новые значения свойствам соответствующего экземпляра объекта, не затрагивая другие видеопроигрыватели в этом же приложении.

Получение экземпляра объекта

Прежде чем начать работу с экземпляром объекта, нам потребуется его получить. Как это сделать?

Платформа Metro сама предоставляет нам большое количество готовых экземпляров объекта. Они создаются самой платформой и хранятся в созданных ей же глобальных переменных.

Так, переменная `Math` хранит экземпляр одноименного объекта, предоставляющего множество методов для выполнения математических и тригонометрических вычислений над числами. И переменная, и экземпляр объекта создаются для нас самой платформой Metro.

```
var a = Math.sqrt(2);
```

Это выражение поместит в переменную `a` квадратный корень из 2. Метод `sqrt` объекта `Math` как раз вычисляет квадратный корень из числа, переданного ему в качестве единственного параметра.

Метод `sin` объекта `Math` вычисляет синус угла, заданного в радианах и переданного данному методу единственным параметром:

```
var b = Math.sin(0.1);
```

Как видим, мы просто используем переменную `Math`, созданную языком JavaScript, чтобы получить доступ к экземпляру объекта `Math` и его методам.

А переменная `document` хранит экземпляр объекта `HTMLDocument`, представляющий весь интерфейс нашего Metro-приложения. Все это также создано платформой Metro.

```
btnOpen = document.getElementById("btnOpen");
```

Здесь мы вызвали метод `getElementById` данного объекта. (Подробнее о нем и вообще об объекте `HTMLDocument` мы поговорим в *главе 5*.)

Помимо этого, платформа Metro создает экземпляры объектов, представляющие все элементы интерфейса нашего приложения, — по одному экземпляру объекта на каждый элемент. Как их получить, мы узнаем также в *главе 5*.

Кстати, упомянутый ранее метод `getElementById` в качестве результата возвращает экземпляр объекта. Это еще один способ получить нужный нам экземпляр.

Однако экземпляры многих объектов придется создавать нам самим. В этом нам поможет *оператор создания экземпляра* `new`:

```
new <имя объекта>([<список параметров, разделенных запятыми>])
```

`Список параметров` может как присутствовать, так и отсутствовать. Обычно он содержит значения, которые присваиваются свойствам экземпляра объекта при его создании.

Оператор `new` возвращает созданный экземпляр объекта. Его можно присвоить какой-либо переменной или свойству или передать в качестве параметра функции или методу.

В качестве примера рассмотрим создание экземпляра объекта `Date`, который предназначен для хранения значений даты и времени.

```
var dNow = new Date();
```

После выполнения этого выражения в переменной `dNow` окажется экземпляр объекта `Date`, хранящий сегодняшнюю дату и текущее время.

```
var dNewYear = new Date(2012, 0, 1);
```

Данное выражение поместит в переменную `dNewYear` экземпляр объекта `Date`, хранящий дату 1 января 2012 года и текущее время. Номер года мы указали первым параметром, номер месяца (нумерация месяцев в языке JavaScript начинается с нуля — запомним это сразу) — вторым, а число — третьим.

```
var dNewYear = new Date(2012, 0, 10, 12, 30, 0);
```

А это выражение поместит в переменную `dNewYear` экземпляр объекта `Date`, хранящий дату 10.01.2012 и время 12:30:00. Количество часов здесь указано четвертым параметром, количество минут — пятым, а количество секунд — шестым.

Вот еще один пример создания экземпляра объекта, взятый из кода нашего первого Metro-приложения:

```
var oFP = new Windows.Storage.Pickers.FileOpenPicker();
```

Здесь мы помещаем в переменную oFP экземпляр объекта с длинным именем Windows.Storage.Pickers.FileOpenPicker. Этот объект представляет стандартный диалог открытия файла.

Работа с экземпляром объекта

Получив тем или иным способом экземпляр объекта, мы можем с ним работать: вызывать его методы и получать доступ к его свойствам.

Ранее мы уже познакомились с методами некоторых объектов и выяснили, как они вызываются. Для этого к переменной, хранящей экземпляр объекта, добавляется справа точка, а после нее записывается вызов метода. Метод — это "собственная" функция объекта, и вызывается она так же (см. раздел, посвященный вызовам функций).

```
btnOpen = document.getElementById("btnOpen");
```

Здесь мы вызвали метод getElementById объекта HTMLDocument, экземпляр которого хранится в переменной document, передав данному методу параметр — строку "btnOpen". Возвращенный этим методом результат будет присвоен переменной btnOpen.

```
vidMain.play();
```

А здесь мы вызвали не принимающий метод play у экземпляра объекта, хранящегося в переменной vidMain.

Если результат, возвращаемый каким-либо методом (назовем его методом 1), представляет собой экземпляр объекта, мы можем сразу вызвать метод у него (это будет метод 2). Для этого мы добавим справа от вызова метода 1 точку, после которой поставим вызов метода 2.

```
oFP.pickSingleFileAsync().then( . . . );
```

Здесь мы сначала вызвали метод pickSingleFileAsync у экземпляра объекта, хранящегося в переменной oFP (метод 1). У полученного в результате вызова метода 1 результата — экземпляра объекта — мы вызвали метод then (метод 2).

Такие цепочки последовательных вызовов методов, когда стоящий справа метод вызывается у экземпляра, возвращенного методом, стоящим слева, встречаются в JavaScript-коде очень часто.

Доступ к свойствам объекта выполняется аналогично. К переменной, хранящей экземпляр объекта, добавляется справа точка, а после нее записывается имя свойства.

Пример:

```
var bIsPaused = vidMain.paused;
```

Здесь мы обратились к свойству paused экземпляра объекта, что хранится в переменной vidMain.

```
vidMain.autoplay = false;
```

Здесь мы присвоили новое значение свойству `autoplay` экземпляра объекта, что хранится в переменной `vidMain`.

```
oFP.fileTypeFilter.replaceAll([".avi", ".mp4", ".wmv"]);
```

А здесь мы сначала обратились к свойству `fileTypeFilter` экземпляра объекта, хранящегося в переменной `oFP`, после чего вызвали у значения этого свойства — также экземпляра объекта — метод `replaceAll`, передав ему в качестве параметра массив из трех строк.

Простые типы как объекты

Интересно, что все простые типы языка JavaScript — строковый, числовой, логический — на самом деле являются объектами. Стало быть, значения, принадлежащие к этим типам, являются экземплярами соответствующих объектов.

Возьмем, к примеру, строки. Все они являются экземплярами объекта `String`. Этот объект поддерживает различные свойства и методы, которые мы можем использовать.

Так, свойство `length` объекта `String` хранит длину строки в символах:

```
var s = "JavaScript";
var l = s.length;
```

Этот код поместит в переменную `l` длину строки `"JavaScript"`, равную 10.

Мы можем записать его и так:

```
var l = "JavaScript".length;
```

Метод `substr` объекта `String` возвращает фрагмент строки заданной длины, начинающийся с указанного символа:

```
substr(<номер первого символа>[, <длина фрагмента>]);
```

Первым параметром передается номер первого символа, включаемого в возвращаемый фрагмент строки.

> ***Внимание!***
> В JavaScript символы в строках нумеруются, начиная с нуля.

Второй, необязательный, параметр задает длину возвращаемого фрагмента в символах. Если он опущен, возвращаемый фрагмент будет содержать все оставшиеся символы строки.

После выполнения этого кода

```
var s1 = s.substr(4);
var s2 = s.substr(4, 2);
```

в переменной `s1` окажется строка `"Script"`, а в переменной `s2` — строка `"Sc"`.

Все числа представляют собой экземпляры объекта `Number`, а все логические величины — экземпляры объекта `Boolean`. Правда, никаких особо полезных для нас свойств и методов они не поддерживают.

Более сложные типы также представляют собой экземпляры особых объектов. Так, массивы являются экземплярами объекта `Array`, а функции — экземплярами объекта `Function`.

Все эти объекты поддерживают метод `toString`. Он не принимает параметров и в качестве результата возвращает строковое представление экземпляра данного объекта.

```
var n = 1234;
var s = n.toString();
```

После выполнения этого кода в переменной `s` окажется строка `"1234"` — строковое представление числа 1234.

Объект *Object* и использование его экземпляров

А еще платформа Metro поддерживает весьма специфический объект с незатейливым именем `Object`. И, хотя полезных для нас свойств и методов он не поддерживает, его экземпляры применяются в Metro-программировании довольно часто.

Экземпляры объекта `Object` используются для хранения сложных структур данных, включающих произвольный набор свойств и методов. Один экземпляр этого объекта может иметь один набор свойств и методов (в дополнение к унаследованным от самого объекта `Object`), а другой экземпляр — совсем другой.

> ***НА ЗАМЕТКУ***
>
> Надо сказать, что язык JavaScript поддерживает возможность задавать собственные свойства и методы у экземпляров всех его объектов. Это вторая из интересных особенностей данного языка.

Экземпляры объекта `Object` создают с помощью особых выражений, называемых *инициализаторами*. Такой инициализатор чем-то похож на определение стиля *(см. главу 3)*:

```
{
  <имя свойства 1>: <значение свойства 1>,
  <имя свойства 2>: <значение свойства 2>,
  . . .
  <имя метода 1>: <функция, реализующая метод 1>,
  <имя метода 2>: <функция, реализующая метод 2>,
  . . .
}
```

После выполнения инициализатора JavaScript вернет нам готовый экземпляр объекта `Object`, который мы можем присвоить какой-либо переменной или свойству либо использовать в качестве параметра функции или метода.

```
var oVideo = { item: "trailer",
               file: "/movie_trailer.avi",
               desc: "Трейлер фильма",
               playVideo: function () { . . . }
             };
```

Здесь мы получили экземпляр объекта `Object` со свойствами `item`, `file` и `desc` и методом `playVideo` и сохранили получившийся экземпляр в переменной `oVideo`.

Обратим внимание на два момента. Во-первых, функцию, реализующую метод `playVideo`, мы объявили прямо в инициализаторе. Во-вторых, создание метода в данном случае — это присваивание функции, которая реализует этот метод, свойству, имя которого станет именем метода. Следовательно, здесь тот же самый случай, что и с присваиванием функции переменной *(см. разд. "Функция как значение. Анонимные функции" ранее в этой главе)*.

Правила написания выражений

В этой главе мы изучили множество выражений JavaScript. Но так и не узнали, по каким правилам они пишутся. Настала пора восполнить этот пробел в наших знаниях.

- Между операндами, операторами, вызовами функций и методов и ключевыми словами допускается сколько угодно пробелов и разрывов строк.
- Запрещаются разрывы строк внутри константы, ключевого слова, имени переменной, свойства, функции, метода или объекта. В противном случае мы получим сообщение об ошибке.
- Признаком конца выражения служит символ точки с запятой (`;`).

В качестве примера давайте возьмем одно из выражений, приведенных в начале главы:

```
y = y1 * y2 + x1 * x2;
```

Мы можем записать его так, разнеся на две строки:

```
y = y1 * y2 +
x1 * x2;
```

Или даже так:

```
y =
y1 *
y2 +
x1 *
x2;
```

И в любом случае оно будет выполнено, т. к. при написании выражения мы не нарушили ни одного из перечисленных правил.

Но если мы нарушим их, выполнив перенос строк внутри имени переменной `y2`:

```
y = y1 * y
2 + x1 * x2;
```

получим сообщение об ошибке.

JavaScript весьма либерально относится к тому, как мы пишем его выражения. Благодаря этому мы можем форматировать JavaScript-код с помощью разрывов строк и пробелов для удобства его чтения.

Комментарии JavaScript

Из *главы 3* мы знаем о существовании комментариев — особых фрагментов кода HTML и CSS, которые не обрабатываются платформой Metro и служат для того, чтобы разработчик смог оставить какие-либо заметки для себя или своих коллег. Было бы странно, если бы JavaScript не предоставлял аналогичной возможности.

Комментарии JavaScript бывают двух видов.

Комментарий, состоящий из одной строки, создают с помощью последовательности символов `//`, которую помещают в самом начале строки комментария:

```
//Создаем экземпляр объекта Date
var dNow = new Date();
```

Комментарий, состоящий из произвольного числа строк, создают с помощью последовательностей символов `/*` и `*/`. Между ними помещают строки, которые станут комментарием:

```
/*
Создаем экземпляр объекта Date.
Он будет хранить значение текущих
даты и времени
*/
var dNow = new Date();
```

Что дальше?

В этой главе мы знакомились с языком JavaScript. Мы изучили принципы написания его выражений, поддерживаемые им типы данных и операторы, разобрались с функциями, массивами, объектами и их экземплярами.

Следующая глава также будет посвящена JavaScript, но уже в более прикладном аспекте. Мы рассмотрим характерные особенности платформы Metro, работу с элементами интерфейса Metro-приложения и принципы, согласно которым создается программная логика. А в качестве примера рассмотрим реальный, работающий код, написанный нами для нашего первого Metro-приложения.

ГЛАВА 5

Логика Metro-приложения: приемы программирования

В предыдущей главе мы познакомились с языком программирования JavaScript, на котором пишется программная логика Metro-приложений. Знакомство это было сугубо теоретическим и касалось самого этого языка и его возможностей.

В этой главе мы будем рассматривать принципы, на основе которых собственно создается программная логика. Мы узнаем, где хранится код логики, познакомимся с объектной моделью документа, событиями и их обработкой, структурой кода логики и специфическими инструментами платформы Metro: объектами-обязательствами, пространствами имен и перечислениями. Вооружившись всеми этими знаниями, мы поймем, как работают Metro-приложения и как заставить их работать так, как нам нужно.

Где хранится код логики Metro-приложения. Файлы логики

Из *главы 2* мы знаем, что код программной логики хранится в файлах с расширением js. Их часто называют *файлами логики*.

Прежде всего, это файл default.js, хранящийся в папке js. В нем содержится уникальная логика создаваемого приложения, та, что пишем мы сами. Изначально этот файл включает некий начальный код, который можно рассматривать как заготовку для создания логики.

Далее, два файла — base.js и ui.js, что входят в состав самой платформы Metro. Они хранят код, необходимый для успешной работы Metro-приложения *(базовую логику)*.

Также мы можем создать дополнительные файлы логики; это может пригодиться в случае разработки очень сложного приложения, логику которого целесообразно разделить на несколько частей. Все эти дополнительные файлы также рекомендуется поместить в папку js или вложенные в нее папки.

Как и в случае с таблицами стилей *(см. главу 3)*, включение в состав проекта файлов логики не является для платформы Metro указанием обязательно "принять их

к исполнению". Нам потребуется выполнить *привязку* хранящейся в них логики к описанию интерфейса приложения.

Для этого служит парный тег `<script>`, который записывается в следующем формате:

```
<script src="<ссылка на файл логики>"></script>
```

Сама ссылка на привязываемый файл логики указывается в качестве значения атрибута `src` данного тега. Содержимое же у тега `<script>` в таком случае никогда не указывается.

```
<script src="//Microsoft.WinJS.0.6/js/base.js"></script>
<script src="//Microsoft.WinJS.0.6/js/ui.js"></script>
```

Эти два тега привязывают к описанию интерфейса файлы base.js и ui.js, хранящие базовую логику.

> ***НА ЗАМЕТКУ***
>
> Копии файлов базовой логики находятся в папке
> C:\Program Files\Microsoft SDKs\Windows\v8.0\ExtensionSDKs\Microsoft.WinJS\
> *<номер версии платформы Metro>*\DesignTime\Debug\
> Neutral\Microsoft.WinJS*<номер версии платформы Metro>*\js.

```
<script src="/js/default.js"></script>
```

А этот тег привязывает файл default.js, что хранит создаваемую нами часть логики.

Как только платформа Metro встретит в HTML-коде тег `<script>`, она загрузит файл, на который ссылается данный тег, и сразу же выполнит код, хранящийся в этом файле. И только после этого она продолжит обработку HTML-кода, следующего за этим тегом.

> ***НА ЗАМЕТКУ***
>
> Язык HTML позволяет указать логику приложения прямо в файле с описанием интерфейса. В таком случае код логики помещается внутрь тега `<script>` (атрибут `src` в этом теге, соответственно, не указывается). Но использовать такой подход в Metro-программировании не рекомендуется.

Объектная модель документа

В процессе работы Metro-приложения его логика будет активно взаимодействовать с элементами интерфейса. Она станет получать значения, введенные в поля ввода, реагировать на нажатия кнопок, выводить результаты обработки данных и даже создавать какую-то часть интерфейса программно.

Выходит, что платформа Metro должна предоставить нам какие-то инструменты, которые позволят логике взаимодействовать с интерфейсом и при этом будут достаточно простыми в использовании. Но что это за инструменты?

Экземпляры особых объектов. Каждый такой экземпляр будет представлять один элемент интерфейса. А собственно взаимодействие будет осуществляться с по-

мощью свойств и методов, поддерживаемых объектами, на основе которых они созданы. Что может быть проще!

Для примера давайте рассмотрим интерфейс нашего приложения. Оно содержит видеопроигрыватель и блок, в который вложена кнопка. Соответственно, платформа Metro создаст для нашего приложения такие экземпляры объектов:

- экземпляр объекта `HTMLVideoElement`, представляющий видеопроигрыватель;
- экземпляр объекта `HTMLDivElement`, представляющий блок;
- экземпляр объекта `HTMLInputElement`, представляющий кнопку.

Как видим, для каждого типа элементов интерфейса Metro "припасла" свой особый объект, полностью поддерживающий все возможности этого элемента. Подробнее об этих объектах мы поговорим потом, когда начнем рассматривать различные элементы интерфейса, поддерживаемые Metro, более подробно.

Кроме этого, Metro создаст для нас еще два важных экземпляра объекта.

- Экземпляр объекта `HTMLDocument`. Этот объект представляет описание всего интерфейса приложения вместе со служебными тегами и фактически представляет тег `<html>` (подробности *см. в главе 3*). Созданный экземпляр данного объекта будет помещен в глобальную переменную `document`.

  Объект `HTMLDocument` предоставляет доступ к важным инструментам, которыми мы обязательно будем пользоваться. Так что сразу же возьмем его на заметку.

- Экземпляр объекта `HTMLBodyElement`. Этот объект представляет описание всего интерфейса приложения, но уже без служебных тегов, и фактически представляет тег `<body>`.

  Объект `HTMLDocument`, помимо всего прочего, поддерживает свойство `body`. В этом свойстве будет храниться экземпляр объекта `HTMLBodyElement` после его создания.

> ***НА ЗАМЕТКУ***
>
> На самом деле платформа Metro создаст еще множество экземпляров объекта. В их состав войдут экземпляры, представляющие тег `<head>` и все хранящиеся в нем служебные теги. Однако вряд ли они нам пригодятся, по крайней мере на первых порах.

Теперь предположим, что мы решили добавить в наше приложение еще и абзац с текстом (тег `<p>`). В этом случае состав экземпляров объектов пополнится еще двумя:

- экземпляром объекта `HTMLParagraphElement`, который представляет собственно абзац;
- экземпляром объекта `Text`, представляющим текст — содержимое абзаца.

Текстовое содержимое тегов также представляется отдельными экземплярами объектов.

Элементы интерфейса Metro-приложения вложены друг в друга — в этом мы убедились еще в *главе 2*. Поэтому все представляющие их экземпляры объектов будут

сведены в особую иерархическую структуру, представляющую эту вложенность. В нашем случае схематично отобразить эту структуру можно так (вложенность элементов обозначена отступами):

```
Описание интерфейса со служебными тегами (тег <html>)
  Описание интерфейса без служебных тегов (тег <body>)
    Видеопроигрыватель
    Блок
      Кнопка
    Абзац
      Текстовое содержимое абзаца
```

Исследуя эту структуру, мы можем проследить, кто является родителем данного элемента интерфейса, кто его потомки и соседи. Вдобавок, сама платформа Metro активно использует данную структуру для собственных нужд.

Осталось сказать, что такая иерархическая структура из экземпляров объектов, представляющих элементы интерфейса, называется *объектной моделью документа*, или *DOM* (Document Object Model).

Как получить доступ к элементу интерфейса из кода логики

Проблема в том, что получить доступ к нужному нам элементу интерфейса, просто обратившись к переменной, где хранится представляющий его экземпляр объекта, мы не сможем. Платформа Metro не создает таких переменных.

Нам придется использовать другие инструменты для доступа к элементам интерфейса. Сейчас мы их рассмотрим.

Получение доступа к элементу по его имени

Самый простой способ добраться до элемента интерфейса — дать ему уникальное *имя* и обратиться к нему по этому имени. Так поступают чаще всего.

Сначала нам следует указать само имя элемента. Оно указывается в качестве значения в атрибуте тега `id`, а этот атрибут ставится в теге, создающем нужный элемент интерфейса.

```
<video id="vidMain"></video>
```

Здесь мы указали для видеопроигрывателя имя `vidMain`.

```
<input type="button" id="btnOpen" />
```

А здесь для кнопки указано имя `btnOpen`.

К именам элементов интерфейсов предъявляются следующие требования. Во-первых, они должны быть уникальными. Во-вторых, они должны содержать только буквы латинского алфавита, цифры и символы подчеркивания, причем начинаться должны с буквы или символа подчеркивания.

Из *главы 3* мы знаем, что атрибут тега `id` служит также для привязки к элементу интерфейса именованного стиля. Так что, как видим, этот атрибут тега позволяет нам "убить двух зайцев": и привязать именованный стиль, и задать имя.

После этого мы сможем собственно получить доступ к этому элементу интерфейса. Для этого служит метод `getElementById` объекта `HTMLDocument`. В качестве единственного параметра он принимает строку с именем элемента, заданного с помощью атрибута тега `id`. А в качестве результата он возвращает экземпляр объекта, представляющий элемент с данными именем, или `null`, если элемент с таким именем отсутствует.

```
vidMain = document.getElementById("vidMain");
```

Получаем экземпляр объекта, представляющий видеопроигрыватель (элемент с именем `vidMain`), и присваиваем его переменной `vidMain`. Обратим внимание, что мы вызвали метод `getElementById` у экземпляра объекта `HTMLDocument`, что хранится в переменной `document`.

```
btnOpen = document.getElementById("btnOpen");
```

Еще нам пригодится экземпляр объекта, представляющий кнопку **Открыть** (она имеет имя `btnOpen`). Этот экземпляр мы сохраним в переменной `btnOpen`.

Получение доступа к элементам по имени создающего их тега

Иногда бывает необходимо получить доступ ко всем элементам интерфейса, созданным с помощью одного тега. Для такого случая платформа Metro предоставляет нам метод `getElementsByTagName`.

Этот метод поддерживается всеми объектами, представляющими элементы интерфейса, а также объектом `HTMLBodyElement`, который представляет все описание интерфейса без служебных тегов. В качестве единственного параметра он принимает строку с именем нужного тега без символов < и >.

Что касается возвращаемого этим методом результата, то он представляет собой экземпляр особого объекта `HTMLElementCollection`. Этот объект следует рассмотреть подробнее.

В *главе 4* мы узнали о существовании массивов — наборов пронумерованных значений, хранящихся в одной переменной. Так вот, объект `HTMLElementCollection` можно рассматривать как массив, но более специализированный и с несколько расширенной функциональностью. Объекты подобного рода называются *коллекциями*.

Экземпляр объекта `HTMLElementCollection`, возвращенный методом `getElementsByTagName`, — это коллекция экземпляров объектов, представляющих элементы интерфейса, что были созданы тегом с указанным нами именем. То есть в результате его вызова мы получим все элементы, созданные с применением указанного тега.

```
colDivs = document.body.getElementsByTagName("div");
```

Здесь мы вызвали метод `getElementsByTagName` у экземпляра объекта `HTMLBodyElement`. Как мы уже знаем, он хранится в свойстве `body` объекта `HTMLDocument`, единственный экземпляр которого находится в переменной `document`. Как результат в переменной `colDivs` окажется коллекция (экземпляр объекта `HTMLElementCollection`), хранящая все блоки (тег `<div>`).

Но как нам получить доступ к отдельному экземпляру объекта, хранящемуся в коллекции (ее *элементу*)? Точно так же, как и к элементу обычного массива, — по его индексу.

```
objDiv = colDivs[0];
```

Здесь мы сохраняем в переменной `objDiv` первый (с нулевым индексом) элемент полученной ранее коллекции — наш единственный блок с кнопками.

Объект `HTMLElementCollection`, как и массив, поддерживает свойство `length`, возвращающее количество элементов в коллекции в виде числа.

Если в интерфейсе приложения нет элементов, созданных с применением какого-то тега, метод `getElementsByTagName` вернет нам пустую коллекцию, не содержащую элементов.

```
colProgresses = document.body.getElementsByTagName("progress");
if (colProgresses.length == 0) {
  s = "В приложении нет индикаторов прогресса";
}
```

Ранее говорилось, что метод `getElementsByTagName` поддерживается также всеми объектами, представляющими элементы интерфейса. Это позволит нам выполнить поиск элементов не во всем интерфейсе приложения, а в каком-либо его элементе, например в блоке.

```
colInputs = objDiv.getElementsByTagName("input");
```

Здесь мы получаем коллекцию всех элементов интерфейса, созданных тегом `<input>` и находящихся в полученном ранее блоке.

Получение доступа к элементам по наименованию привязанного к ним стилевого класса

Еще существует возможность получить все элементы интерфейса, к которым был привязан определенный стилевой класс. Для этого применяется метод `getElementsByClassName`.

Он поддерживается всеми объектами, представляющими элементы интерфейса, а также объектом `HTMLBodyElement`. В качестве единственного параметра он принимает строку с наименованием стилевого класса без символа точки в начале. В качестве результата будет возвращена коллекция — экземпляр объекта `HTMLElementCollection`, содержащий все найденные элементы.

```
colBC = document.body.getElementsByClassName("button-cont");
```

Здесь мы получаем коллекцию всех элементов, к которым был привязан стилевой класс `.button-cont`.

Получение доступа к элементам по сложному критерию

Наконец, высший пилотаж — получение всех элементов интерфейса, удовлетворяющих какому-либо сложному критерию. В качестве таких критериев используются составные селекторы, схожие с теми, что применяются в комбинированных стилях *(см. главу 3)*.

Для такого случая платформа Metro предоставляет нам два метода. Они поддерживаются всеми объектами, представляющими элементы интерфейса, и объектом `HTMLBodyElement`. В качестве единственного параметра они принимают строку с селектором.

Метод `querySelector` возвращает экземпляр объекта, представляющий первый из найденных элементов, что соответствует заданному селектору. Если ни одного подходящего элемента найдено не было, возвращается `null`.

```
objButton = document.body.querySelector("input[type=button]");
```

Здесь мы ищем первый элемент, созданный с помощью тега `<input>`, с атрибутом `type`, чье значение равно `button`, т. е. кнопку. Причем поиск мы выполняем во всем интерфейсе приложения.

```
objButton = document.body.querySelector("div > input[type=button]");
```

Здесь мы также ищем кнопку, но уже непосредственно вложенную в блок.

```
objButton = objDiv.querySelector("input[type=button]");
```

А здесь мы ищем кнопку в блоке, полученном ранее.

Метод `querySelectorAll` возвращает коллекцию, содержащую все найденные элементы.

```
colButtons = document.body.querySelectorAll("input[type=button]");
```

Получаем все присутствующие в приложении кнопки.

События и их обработка

Следующий вопрос, который мы разберем, очень важен. Он касается принципов, по которым работают Metro-приложения.

События

Любое Metro-приложение бóльшую часть времени занимается тем, что ожидает действий пользователя — простаивает. В самом деле, быстродействие компьютера очень велико, и большинство операций он выполняет молниеносно; люди же, к сожалению, не столь резвы...

Понятно, что во время простоя Metro-приложения никакой код из его программной логики не выполняется (иначе это состояние не называлось бы простоем). Напротив, как только пользователь выполнил какое-то действие, например нажал на кнопку, платформе Metro следует выполнить соответствующий этому действию фрагмент логики. Что тоже понятно.

Непонятно только, как сопоставить действие пользователя с фрагментом логики, который следует исполнить в ответ на это действие. Существуют ли какие-то инструменты, позволяющие сделать это?

Существуют. И сейчас мы их рассмотрим.

Первый инструмент — *события*. Так называются особые условия, возникающие при выполнении пользователем различных действий над интерфейсом приложения. Например, когда пользователь нажимает на кнопку, возникает событие, называемое "нажатие". А когда пользователь перемещает палец по сенсорному экрану на видеопроигрывателе, возникает событие "перемещение пальца".

Следует сразу уяснить, что событие возникает именно в том элементе интерфейса, в котором пользователь выполнил действие. Так, в предыдущем примере событие "нажатие" возникнет в кнопке, на которой нажал пользователь, а событие "перемещение пальца" — в видеопроигрывателе, на котором перемещался палец. Такой подход позволяет локализовать событие и облегчить его обработку (о которой речь пойдет далее).

События в Metro-приложении возникают не только в результате действий пользователя. Многие события порождаются самой платформой Metro в процессе работы приложения. Так, событие "построение DOM" возникает, когда Metro закончит обработку интерфейса приложения и построит в памяти компьютера представляющую его структуру DOM. События "активация" и "деактивация" возникают, соответственно, при активации и деактивации приложения. А событие "изменение координат" возникает, когда встроенный GPS-сенсор сообщает о том, что географические координаты устройства изменились.

Каждое событие имеет уникальное *имя* в терминологии Metro. Например, событие "нажатие" носит имя `click`, событие "перемещение пальца" — имя `MSGestureChange`, а событие "построение DOM" — имя `DOMContentLoaded`.

Платформа Metro имеет еще одну интересную особенность. Как уже говорилось ранее, событие возникает в том элементе интерфейса, действие с которым было выполнено пользователем. Впоследствии это же событие возникает в родителе этого элемента, потом — в родителе его родителя и т. д. Наконец событие возникает в теге `<body>` (описании интерфейса без служебных тегов) и там прекращает свое существование.

Например, в нашем случае при нажатии на кнопке **Открыть** событие "нажатие" возникнет сначала в этой кнопке, потом — в блоке, где содержится эта кнопка, а напоследок — в теге `<body>`. Событие будто следует вверх по иерархии элементов интерфейса.

Этот процесс называется *всплытием* события. Во многих случаях оно позволяет разработчикам упростить обработку событий...

Обработчики событий

Об обработке событий самое время сейчас поговорить.

Второй инструмент для сопоставления действия пользователя фрагменту логики, который должен реагировать на это действие, — *обработчик событий*. Так называется функция, тело которой представляет собой фрагмент логики, что сопоставляется с действием пользователя.

```
function btnOpenClick() {
    . . .
}
```

Здесь мы объявили функцию `btnOpenClick()`. Она содержит фрагмент логики, который будет сопоставлен с событием `click` ("нажатие"), возникающим в кнопке **Открыть** (`btnOpen`). То есть она будет выполняться при нажатии на этой кнопке.

Чтобы обычная функция стала обработчиком события, нам следует выполнить ее *привязку*. Привязка эта выполняется, во-первых, к элементу интерфейса, в котором будет происходить событие, а во-вторых, к самому событию. В нашем случае нам потребуется привязать функцию `btnOpenClick` к кнопке **Открыть** (`btnOpen`) и событию `click` ("нажатие").

Для привязки обработчика события служит метод `addEventListener`. Он поддерживается всеми объектами, представляющими элементы интерфейса. Формат вызова этого метода таков:

<элемент интерфейса>.addEventListener(*<событие>*, *<обработчик>*)

Прежде всего, этот метод вызывается у экземпляра объекта, представляющего элемент интерфейса, к которому мы хотим привязать обработчик. Имя события, к которому он привязывается, указывается в виде строки первым параметром. А вторым параметром задается функция, которая станет обработчиком. Результата этот метод не возвращает.

```
btnOpen.addEventListener("click", btnOpenClick);
```

Здесь мы привязали объявленную ранее функцию к кнопке `btnOpen` (она была получена ранее и хранится в переменной `btnOpen`) и событию `click`.

Обработчик события можно оформить и в виде анонимной функции:

```
btnOpen.addEventListener("click", function () { . . . });
```

Так даже удобнее, если мы впоследствии не собираемся "отвязывать" обработчик от события (как это сделать, будет рассказано далее).

```
colButtons = document.body.querySelectorAll("input[type=button]");
for (var i = 0; i > colButtons.length; i++) {
    colButtons[i].addEventListener("click", btnClick);
}
```

А здесь мы вызовом рассмотренного ранее метода `querySelectorAll` получаем сразу все кнопки приложения и привязываем к ним обработчик `btnClick`.

Платформа Metro допускает и другой способ привязки обработчика к событию. Вот формат, в котором записывается выражение, выполняющее привязку:

```
<элемент интерфейса>.on<свойство, представляющее событие> = <обработчик>;
```

Здесь функция-обработчик просто присваивается свойству, представляющему событие. Имя этого свойства совпадает с именем события, набранным строчными (маленькими) буквами, а в его начале обязательно ставятся символы `on`. Например:

```
btnOpen.onclick = btnOpenClick;
```

Этот способ проще, чем описанный ранее, однако не совсем укладывается в парадигму HTML и JavaScript. Поэтому его использование не рекомендовано (хотя на самом деле мало кто следует этой "нерекомендации").

На всякий случай узнаем, как отменить привязку обработчика к элементу интерфейса и событию ("отвязать" его). Для этого применяется метод `removeEventListener`, также поддерживаемый всеми объектами, что представляют элементы интерфейса. Формат его вызова такой же, что и у метода `addEventListener`.

```
btnOpen.removeEventListener("click", btnOpenClick);
```

Удаляем привязку функции-обработчика `btnOpenClick` к кнопке `btnOpen` и событию `click`.

Вряд ли мы будем часто выполнять "отвязывание" обработчика от события (а возможно, вообще никогда). Но знать о такой возможности не помешает.

События, поддерживаемые всеми элементами интерфейса

Теперь давайте познакомимся с несколькими событиями, которые поддерживаются всеми объектами — элементами интерфейса. Все эти события возникают при манипуляциях пальцами на сенсорном экране.

- `MSGestureTap` — возникает при кратковременном касании пальцами сенсорного экрана на элементе интерфейса или щелчке мышью.
- `click` — то же самое, что и `MSGestureTap`.
- `MSGestureDoubleTap` — возникает при двойном кратковременном касании пальцами сенсорного экрана на элементе интерфейса или двойном щелчке мышью.
- `MSGestureStart` — возникает сразу после касания пальцами сенсорного экрана на элементе интерфейса или при нажатии кнопки мыши.
- `MSGestureHold` — возникает при удерживании пальцев на элементе интерфейса в течение определенного времени.
- `MSGestureChange` — возникает при перемещении пальцев по сенсорному экрану над элементом интерфейса, а также при перемещении курсора мыши при нажатой кнопке. Это событие также возникает при выполнении "многопальцевых" жестов.

- `MSGestureEnd` — возникает при снятии пальцев с сенсорного экрана на элементе интерфейса или отпускании кнопки мыши.

Как видим, в случае отсутствия сенсорного экрана платформа Metro имитирует его функциональность с помощью мыши. Так что мы можем без проблем разрабатывать планшетные Metro-приложения на традиционных ПК, не оснащенных сенсорными экранами, пользуясь для их отладки только мышью.

> ***НА ЗАМЕТКУ***
>
> Надо сказать, что платформа Metro поддерживает и специфические "мышиные" события. Но, поскольку для планшетов с их сенсорными экранами такие события неактуальны, мы не будем их рассматривать.

Еще платформа Metro поддерживает события клавиатуры, события, связанные с процессом загрузки дополнительных файлов, события, порождаемые специализированными устройствами планшета, и события, генерируемые самой платформой Metro. Мы рассмотрим их позже.

Получение сведений о событии

Любой обработчик события может получить в качестве единственного параметра экземпляр особого объекта. Этот объект предназначен для хранения сведений о событии.

Чтобы получить этот параметр, нам потребуется указать его в объявлении функции — обработчика события.

```
function btnOpenClick(evt) {
    . . .
}
```

Здесь мы дали этому параметру имя `evt`.

Существует довольно много объектов, каждый из которых соответствует определенной группе событий: события сенсорного экрана, события клавиатуры и др. Однако все они порождены от объекта `Event`, который хранит базовые сведения о событии.

Объект `Event` поддерживает следующие интересные для нас свойства:

- `currentTarget` — возвращает элемент интерфейса, в котором в данный момент выполняется обработка события, т. е. элемент, к которому привязан данный обработчик события;
- `target` — возвращает элемент интерфейса, в котором возникло событие. Это не обязательно тот же самый элемент, в котором выполняется обработка события, т. к. событие могло всплыть (о всплытии событий говорилось ранее).

Кроме того, объект `Event` поддерживает метод `stopPropagation`, который прерывает всплытие события. Этот метод не принимает параметров и не возвращает результата.

```
function btnOpenClick(evt) {
    . . .
    var objTarget = evt.target;
    . . .
    evt.stopPropagation();
    . . .
}
```

Объект `MSGestureEvent` представляет сведения о событиях сенсорного экрана, в том числе и "многопальцевых" жестах. Он порожден на основе объекта `Event` и, следовательно, поддерживает все его свойства и методы.

Помимо этого, объект `MSGestureEvent` поддерживает следующие свойства, актуальные для "однопальцевых" жестов:

- `clientX` — возвращает горизонтальную координату точки касания относительно левого верхнего угла клиентской области приложения в виде числа в пикселах;
- `clientY` — возвращает вертикальную координату точки касания относительно левого верхнего угла клиентской области приложения в виде числа в пикселах;
- `offsetX` — возвращает горизонтальную координату точки касания относительно левого верхнего угла элемента, в котором возникло событие, в виде числа в пикселах;
- `offsetY` — возвращает вертикальную координату точки касания относительно левого верхнего угла элемента, в котором возникло событие, в виде числа в пикселах.

А эти свойства данного объекта будут полезны в случае "многопальцевых" жестов:

- `expansion` — возвращает диаметр воображаемой окружности, находящейся между пальцами в случае выполнения "многопальцевых" жестов, в виде числа;
- `rotation` — возвращает величину угла, на который был выполнен поворот, в виде числа в градусах. Величина поворота считается по часовой стрелке. Жест поворота выполняется наложением на сенсорный экран двух пальцев и последующим вращением их вокруг центра области, расположенной между ними;
- `scale` — возвращает величину масштаба, заданную жестом масштабирования, в виде числа. Жест масштабирования выполняется наложением на сенсорный экран двух пальцев и последующим их сдвиганием или раздвиганием;
- `translationX` — возвращает величину сдвига пальца от точки касания по горизонтали в виде числа в пикселах;
- `translationY` — возвращает величину сдвига пальца от точки касания по вертикали в виде числа в пикселах;
- `velocityAngular` — возвращает величину угловой скорости при выполнении жеста вращения в виде числа;
- `velocityExpansion` — возвращает величину скорости, с которой изменяется диаметр воображаемой окружности, находящейся между пальцами в случае выполнения "многопальцевых" жестов, в виде числа;

- `velocityX` — возвращает величину скорости перемещения пальца по горизонтали в виде числа;
- `velocityY` — возвращает величину скорости перемещения пальца по вертикали в виде числа.

Существуют и другие объекты, представляющие сведения о событиях других типов. Все они, так или иначе, порождены от объекта `Event`. Эти объекты мы рассмотрим потом.

Структура файла default.js

А теперь давайте вернемся к файлу default.js — "душе" любого Metro-приложения. Еще в *главе 2* мы убедились, что он имеет довольно сложную структуру.

Служебные строки и выражения

В качестве примера возьмем файл default.js из нашего первого приложения. Откроем его и мысленно уберем весь незначащий код и код, созданный нами самими. Останется вот что:

```
(function () {
    'use strict';

    var app = WinJS.Application;

    app.onactivated = function (eventObject) {
        . . .
    };

    app.oncheckpoint = function (eventObject) {
        . . .
    };

    app.start();
})();
```

Видно сразу, что весь код, реализующий логику, помещается между строками `(function () {` и `})();`. (Что делают эти строки, мы узнаем потом.)

Сразу за строкой `(function () {` следует непонятное выражение `'use strict';`. Оно указывает платформе Metro задействовать так называемый строгий режим выполнения, при котором объявления переменных обязательны.

Далее мы видим выражение, объявляющее переменную `app` и присваивающее ей экземпляр объекта `WinJS.Application`. Данный объект представляет само Metro-приложение; единственный его экземпляр создается самой платформой Metro и доступен через переменную `WinJS.Application`.

Следующие два выражения привязывают обработчики к событиям `activated` и `checkpoint` только что полученного экземпляра объекта `WinJS.Application`. (Подробнее об этих событиях разговор пойдет в *главе 23*.) Эти обработчики, созданные самим Visual Studio, не выполняют никакой полезной работы, а являются лишь заготовкой.

В самом конце, перед строкой `})();`, мы видим вызов метода `start` все того же экземпляра объекта `WinJS.Application`. Этот метод запускает работу Metro-приложения.

Все описанные здесь строки и выражения являются служебными и обеспечивают само функционирование Metro-приложения. Удалять ни в коем случае не следует; в противном случае приложение не будет работать как надо.

Куда помещается собственно код логики

Код, описывающий логику, помещается в любое место между выражением `'use strict';` и вызовом метода `start` экземпляра объекта `WinJS.Application`. Нужно только помнить, что объявление переменной или функции должно предшествовать ее использованию.

В качестве примера также рассмотрим наше первое приложение. Создавая его логику, мы объявили переменные `btnOpen` и `vidMain` и функцию `btnOpenClick()`. Переменные будут хранить экземпляры объектов, представляющие кнопку **Открыть** и видеопроигрыватель, а функция станет обработчиком события `click` кнопки `btnOpen`. Вот код, выполняющий все эти объявления (выделен полужирным шрифтом):

```
(function () {
    'use strict';
    . . .
    var app = WinJS.Application;

    var btnOpen, vidMain;
    . . .
    function btnOpenClick() {
        . . .
    }

    app.start();
})();
```

Как правило, объявления всех глобальных переменных обычно ставятся в самом начале кода, сразу после выражения `var app = WinJS.Application;`, а объявления функций помещаются перед выражением `app.start();`. (Мы, собственно, так и сделали.) Такое расположение частей кода считается хорошим стилем программирования.

Инициализация Metro-приложения

Самое первое действие, которое нам потребуется сделать в процессе написания логики, — получить все элементы интерфейса, с которыми мы будем работать все время, пока функционирует приложение, и привязать к их событиям обработчики. Профессиональные программисты называют это действие *инициализацией* приложения.

И когда мы начнем выполнять инициализацию, то сразу натолкнемся на очередной "подводный камень".

Предположим, мы вставили код, выполняющий инициализацию, в самом конце — перед вызовом метода `start` экземпляра объекта `WinJS.Application` (вставленный код выделен полужирным шрифтом):

```
    . . .
    btnOpen = document.getElementById("btnOpen");
    vidMain = document.getElementById("vidMain");
    btnOpen.addEventListener("click", btnOpenClick);

    app.start();
})();
```

В этом случае приложение работать не будет — мы получим сообщение об ошибке. И вот почему...

Вспомним, как платформа Metro обрабатывает файлы с описанием интерфейса и файлы логики. Как только она находит тег `<script>`, она сразу же загружает файл логики, на который он ссылается, и выполняет хранящийся в нем код; обработка остального HTML-кода на это время приостанавливается.

Стало быть, выражения, что мы вставили в код ранее, будут выполнены еще до того, как будет обработан остальной HTML-код, формирующий, в том числе, и элементы интерфейса. Экземпляры объектов, представляющие эти элементы, созданы не будут, и, как следствие, мы не сможем получить к ним доступ.

Есть ли способ выполнить этот код только после того, как весь HTML-код приложения будет загружен и обработан и соответствующая структура DOM будет сформирована? Есть.

Ранее упоминалось, что объект `HTMLDocument`, представляющий все описание интерфейса, поддерживает событие `DOMContentLoaded`. Это событие возникает сразу по окончании формирования DOM. То, что нам нужно!

Создаем обработчик события `DOMContentLoaded` и помещаем в его тело код, выполняющий инициализацию (выделен полужирным шрифтом).

```
    . . .
    document.addEventListener("DOMContentLoaded", function () {
        btnOpen = document.getElementById("btnOpen");
        vidMain = document.getElementById("vidMain");
```

```
        btnOpen.addEventListener("click", btnOpenClick);
    });

    app.start();
})();
```

Вот теперь все заработает как надо!

Объект-обязательство

Настала пора рассмотреть функцию `btnOpenClick()` — обработчик события `click` кнопки `btnOpen`. В ее теле есть один нюанс, о котором нам нужно знать.

Чтобы дать пользователю возможность воспроизвести в нашем первом Metro-приложении произвольный видеофайл, мы использовали стандартный диалог открытия файла, предоставляемый платформой Metro. Для этого мы создали экземпляр объекта `Windows.Storage.Pickers.FileOpenPicker`, представляющего этот диалог:

```
var oFP = new Windows.Storage.Pickers.FileOpenPicker();
```

Задав параметры диалога с помощью различных свойств данного объекта, мы вызвали его метод `pickSingleFileAsync`, чтобы вывести диалог на экран.

```
oFP.pickSingleFileAsync();
```

Как только диалог будет выведен на экран, платформа Metro начнет выполнять код, следующий за вызовом упомянутого ранее метода. То есть, пока пользователь будет гадать, какой фильм ему выбрать для просмотра, приложение будет продолжать работу.

Но вот пользователь все-таки выбрал файл. Какой именно? Как это выяснить? Очень просто.

Метод `pickSingleFileAsync` возвращает в качестве результата экземпляр объекта `WinJS.Promise`. Этот объект хранит сведения о действии, выполнение которого в данный момент продолжается и завершится в будущем, и называется *обязательством*.

```
objPromise = oFP.pickSingleFileAsync();
```

Здесь мы получаем в переменной `objPromise` экземпляр объекта `Promise` — обязательство в будущем предоставить сведения о файле, выбранном пользователем.

Объект `WinJS.Promise` поддерживает метод `then`. Формат этого метода таков:

```
<обязательство>.then(<функция завершения>[, <функция ошибки>
↳[, <функция прогресса>]])
```

`Функция завершения` вызывается, когда выполнение действия, обозначаемого обязательством, завершится. В качестве единственного параметра она должна принимать данные, являющиеся результатом выполнения данного действия. В нашем случае это экземпляр объекта `Windows.Storage.StorageFile`, представляющий выбранный файл (мы рассмотрим данный объект в *главе 16*).

Функция ошибки вызывается в случае возникновения ошибки в процессе выполнения операции. В качестве единственного параметра она должна принимать экземпляр объекта `Error`, описывающий ошибку.

Функция прогресса периодически вызывается в процессе выполнения действия. В качестве единственного параметра она должна принимать экземпляр объекта, хранящий данные о прогрессе выполнения данного действия; эти объекты мы рассмотрим потом.

Функция завершения — единственный обязательный параметр метода `then`. *Функции ошибки* и *прогресса* указывать необязательно.

```
objPromise.then(function (oFile) { . . . });
```

Здесь мы вызываем метод `then` у полученного ранее обязательства и задаем в качестве его единственного параметра функцию завершения, которая откроет выбранный файл в видеопроигрывателе.

В реальности пишут вот так:

```
oFP.pickSingleFileAsync().then(function (oFile) { . . . });
```

Так заметно короче, и нет нужды объявлять лишнюю переменную.

Пространства имен

Платформа Metro поддерживает огромное количество объектов, представляющих различные программные инструменты. Их настолько много, что нетрудно запутаться.

Понятие пространства имен

Поэтому возникает нужда объединить объекты, представляющие родственные инструменты, в некую сущность более высокого порядка. Такая сущность называется *пространством имен*.

Возьмем, к примеру, уже знакомый нам объект-обязательство `Promise`. Он находится в пространстве имен `WinJS`, объединяющем различные вспомогательные инструменты.

Пространства имен могут вкладываться друг в друга. Так, объект `FileOpenPicker` находится в пространстве имен `Pickers`, вложенное в пространство имен `Storage`, которое, в свою очередь, вложено в пространство имен `Windows`.

Каждое пространство имен имеет *имя*, уникальное в пространстве имен, в которое оно вложено. К этим именам предъявляются те же требования, что и к именам переменных, функций и объектов.

Чтобы получить доступ к объекту, находящемуся в пространстве имен, следует указать его полное имя. *Полное имя объекта* включает, прежде всего, имена всех пространств имен, в которые он последовательно вложен, перечисленных в порядке

от "внешних" к "внутренним" и разделенные точками. После них, также через точку, указывается собственно имя объекта.

```
WinJS.Promise
```

Это полное имя объекта `Promise`.

```
Windows.Storage.Pickers.FileOpenPicker
```

А это полное имя объекта `FileOpenPicker`.

В дальнейшем мы будем использовать только полные имена объектов.

На деле пространства имен сами представляют собой экземпляры особых объектов. Находящиеся в них объекты и вложенные пространства имен представляют собой свойства этих объектов. (Да, объекты тоже можно присваивать переменным и свойствам.) Например, пространство имен `WinJS` представляется одноименным экземпляром объекта, а находящийся в нем объект `Promise` — одноименным свойством данного экземпляра.

Если же объект не вложен ни в одно из пространств имен, то говорят, что он находится внутри *глобального пространства имен*. К таким объектам принадлежат рассмотренные нами в *главе 4* объекты `String`, `Number`, `Object` и пр., а также все объекты, представляющие элементы интерфейса и сам интерфейс.

Анонимное пространство имен

Ранее мы выяснили, что весь код логики помещается между строками `(function () {` и `})();`. Но так и не узнали, зачем они нужны.

Эти строки определяют анонимную функцию, тело которой будет выполнено сразу же при обработке файла default.js. В результате все объявленные нами переменные и функции окажутся вложенными в особое пространство имен, защищенное и от прочих пространств имен, поддерживаемых платформой Metro, и от глобального пространства имен. Это пространство имен не имеет своего имени, отчего и называется *анонимным*.

Это сделано для того, чтобы не допустить "засорения" переменными и функциями глобального пространства имен. К тому же, помещать код приложения в отдельное пространство имен (хотя бы анонимное) считается хорошим стилем JavaScript-программирования. Так что давайте и впредь поступать подобным образом.

Перечисления

Еще платформа Metro предоставляет нам новый тип данных. Поскольку мы в дальнейшем будем часто сталкиваться со значениями этого типа, рассмотрим его прямо сейчас.

Это *перечисление* — строго определенный набор значений, которые являются строками либо числами. Каждое из этих значений *(элементов перечисления)* имеет строго определенное имя, по которому к нему можно обратиться.

Доступ к элементу перечисления выполняется так же, как к свойству или методу объекта. Сначала записывается имя перечисления, потом ставится точка, а за ней — имя элемента.

Для примера давайте рассмотрим вот такое выражение из кода нашего первого Metro-приложения:

```
oFP.viewMode = Windows.Storage.Pickers.PickerViewMode.list;
```

Здесь мы присвоили свойству `viewMode` экземпляра объекта `oFP` значение элемента `list` перечисления `Windows.Storage.Pickers.PickerViewMode`. (Это перечисление представляет режимы отображения файлов, поддерживаемые стандартным диалогом открытия файла платформы Metro.)

Перечисления фактически представляют собой единственные экземпляры объектов, а их элементы являются свойствами этих объектов. Так, в нашем случае имеется экземпляр объекта `Windows.Storage.Pickers.PickerViewMode`, имеющий, в числе прочего, свойство `list`.

Что дальше?

В этой главе мы познакомились с основными приемами программирования, применяемыми при разработке Metro-приложений. Мы узнали, что такое DOM, выяснили, как получить доступ к экземпляру объекта, представляющему элемент интерфейса, разобрались с событиями и их обработкой, выяснили, как пишется код логики, и напоследок поговорили о специфических конструкциях языка JavaScript, поддерживаемых платформой Metro.

Следующая глава будет еще более прикладной. Мы начнем изучение простых элементов управления, которые поддерживаются самим языком HTML. Хоть они и называются простыми, но предлагают нам довольно много возможностей. Так что не расслабляемся!

ЧАСТЬ II

Создание интерфейса Metro-приложений

ГЛАВА 6

Элементы управления HTML

В предыдущей главе мы изучали приемы программирования, используемые при создании Metro-приложений. Мы узнали, как пишется программная логика, что такое DOM, что такое события и как они обрабатываются, и познакомились с некоторыми программными средствами платформы Metro, которые будем активно использовать в дальнейшем.

Однако все это была присказка. Сказка начнется сейчас, с рассказа об элементах управления, что поддерживаются платформой Metro.

Начать следует с того, что все элементы управления, поддерживаемые Metro, делятся на две группы.

- Поддерживаемые языком HTML 5, или *элементы управления HTML*. Каждый такой элемент управления создается с помощью особого тега и не требует для нормальной работы никакого дополнительного кода.
- Реализуемые платформой Metro, или *элементы управления Metro*. Представляют собой комбинации тегов HTML, стилей CSS и кода JavaScript. Необходимо включение дополнительного кода, чтобы они могли нормально работать.

В этой главе мы рассмотрим только элементы управления HTML. Элементам управления Metro будет посвящена *глава 7*.

Основные элементы управления HTML

Разговор об элементах управления HTML мы начнем с так называемых основных элементов, которые предназначены собственно для получения данных от пользователя. К ним относятся всевозможные кнопки, поля ввода, флажки, переключатели, списки, области редактирования и пр.

Все основные элементы управления HTML являются встроенными элементами интерфейса. (О встроенных элементах рассказывалось в *главе 3*.)

Кнопки

Кнопки — вероятно, самые популярные элементы управления. Интерфейс практически любого приложения включает в себя, по крайней мере, одну кнопку.

Язык HTML поддерживает создание кнопок двух видов. Рассмотрим их.

Простая кнопка

Простая кнопка HTML в качестве надписи может содержать только обычный неформатированный текст. Такая кнопка создается с помощью одинарного тега `<input>`, для атрибута `type` которого задано значение `button`.

```
<input type="button" id="btnOpen" value="Открыть" />
```

Этот тег создает простую кнопку с именем `btnOpen` и надписью **Открыть**.

Тег `<input>` служит для создания самых разных элементов управления: кнопок, полей ввода, флажков и переключателей. Тип создаваемого элемента управления указывает атрибут тега `type`. Так, значение `button` этого атрибута тега указывает платформе Metro вывести на экран простую кнопку.

Для указания надписи служит атрибут тега `value`. Текст надписи задается в качестве его значения.

Атрибут тега без значения `disabled` делает кнопку недоступной для нажатия (запрещенной).

```
<input type="button" id="btnStart" value="Пуск" disabled />
```

Здесь мы создаем кнопку **Пуск** и делаем ее изначально недоступной.

Кнопка представляется объектом `HTMLInputElement`. Этот объект поддерживает несколько полезных для нас свойств и уже знакомое нам по *главе 5* событие `click`.

Свойство `value` хранит строку с текстом надписи и соответствует одноименному атрибуту тега `<input>`.

```
btnOpen = document.getElementById("btnOpen");
btnOpen.value = "Закрыть";
```

Здесь мы меняем надпись созданной ранее кнопки на **Закрыть**.

Свойство `disabled` хранит логическое значение, указывающее на доступность кнопки: значение `false` указывает, что кнопка доступна, а значение `true` — что кнопка недоступна. Понятно, что это свойство соответствует атрибуту тега `disabled`.

```
btnStart = document.getElementById("btnStart");
btnStart.disabled = false;
```

Здесь мы делаем созданную ранее кнопку **Пуск** доступной для нажатия.

Событие `click` возникает при нажатии кнопки.

```
btnOpen.addEventListener("click", btnOpenClick);
```

Привязываем к событию `click` кнопки **Открыть** функцию-обработчик `btnOpenClick`.

НА ЗАМЕТКУ

Вообще-то, для отслеживания нажатия кнопки можно обрабатывать событие `MSGestureTap` *(см. главу 5)*. Просто имя события `click` проще запомнить...

Сложная кнопка

Сложная кнопка HTML, в отличие от простой, может иметь в качестве надписи любой элемент интерфейса — текст, графическое изображение, таблицу и даже другой элемент управления — или их совокупность. Обычно сложные кнопки применяются для создания *графических кнопок* — кнопок, в качестве надписи использующих графические изображения.

Сложная кнопка создается с помощью парного тега `<button>`. В качестве содержимого этого тега указывается HTML-код, формирующий надпись кнопки.

```
<button id="btnOpen">Открыть</button>
```

Создаем сложную кнопку с текстовой надписью **Открыть**.

```
<button id="btnOpen"><img src="/images/buttons/open.png" /></button>
```

Создаем графическую кнопку. Графическое изображение, которое станет надписью для нее, берем из файла open.png, хранящегося в папке images\buttons.

```
<button id="btnOpen">
  <img src="/images/buttons/open.png" />
  Открыть
</button>
```

А эта сложная кнопка получит в качестве надписи комбинацию изображения и текста.

Платформа Metro предоставляет нам простой способ создать графическую кнопку **Назад** (Back) с изображением стрелки, направленной вправо. Для этого следует создать "пустую" (не имеющую содержимого) сложную кнопку HTML и привязать к ней стилевой класс `.win-backbutton`.

```
<button id="btnBack" class="win-backbutton"></button>
```

Сложная кнопка представляется объектом `HTMLButtonElement`. Он поддерживает знакомые нам свойство `disabled` и событие `click`. Так что, как видим, сложная кнопка практически аналогична простой.

Поле ввода

Второй по популярности элемент управления — поле ввода. В самом деле, многие приложения требуют ввести какие-то данные вручную, и поле ввода в таком случае незаменимо.

Работа с полями ввода

Поле ввода HTML создается с помощью того же тега `<input>`. В качестве значения атрибута `type` этого тега указывается строка, задающая тип создаваемого поля вво-

да. Все эти строки и соответствующие им типы полей ввода, поддерживаемые HTML, перечислены далее.

- `text` — обычное поле ввода, в которое можно ввести любые данные.

> ***На заметку***
>
> Обычное поле ввода также будет создано, если в теге `<input>` вообще не указывать атрибут `type`. Или, говоря другими словами, `text` является для атрибута тега `type` значением по умолчанию.

- `password` — поле ввода пароля. Все вводимые в такое поле символы представляются точками. Обычно используется для указания паролей и других конфиденциальных данных.
- `number` — поле для ввода чисел.
- `url` — поле для ввода интернет-адресов.
- `email` — поле для ввода адресов электронной почты.
- `tel` — поле для ввода телефонного номера.

```
Имя: <input type="text" id="txtName" /><br />
Пароль: <input type="password" id="txtPassword" />
```

Создаем обычное поле ввода и поле ввода пароля. (Тег `<br>` выполняет разрыв строки в том месте, в котором установлен. Подробнее о нем мы поговорим в *главе 8*.)

Атрибут тега `value` позволяет указать изначальное значение, которое будет подставлено в поле ввода сразу после его создания.

```
Год рождения: <input type="number" id="txtYear" value="1970" />
```

Здесь мы создаем поле для ввода числа и задаем для него в качестве изначального значения строку 1970.

Атрибут тега без значения `readonly` делает поле ввода доступным только для чтения. Пользователь не сможет ничего занести в поле ввода; у него будет возможность только прочитать находящееся там значение, выделить его и скопировать в буфер обмена.

```
<input type="text" id="txtOutput" readonly />
```

Создаем поле ввода, доступное только для чтения. Впоследствии мы можем использовать его для вывода результатов вычисления.

Помимо этого, поле ввода может быть сделано недоступным, для чего достаточно указать в создающем его теге атрибут `disabled`.

Поле ввода представляется объектом `HTMLInputElement`. Этот объект поддерживает свойства `value`, `readOnly` и `disabled`.

Свойство `value` мы сможем использовать, чтобы получить занесенное в поле ввода значение.

```
txtName = document.getElementById("txtName");
name = txtName.value;
```

Отметим, что значение свойства `value` в любом случае будет представлять собой строку, даже если это свойство принадлежит полю ввода числовой величины. Так что нам придется в случае необходимости преобразовать это значение в число.

Свойство `readOnly` хранит логическое значение, разрешающее или запрещающее пользователю вводить в данное поле значение. Если свойство имеет значение `false`, пользователь может вводить в поле значение, если значение `true` — не может (поле ввода доступно только для чтения). Это свойство соответствует атрибуту тега `readonly`.

Проверка корректности введенных данных

Мы уже знаем, что HTML поддерживает специальные поля ввода, предназначенные для занесения в них чисел, интернет-адресов и адресов электронной почты. Причем при вводе значений в такие поля платформа Metro сама проверяет их на корректность. (Как выяснить, корректное ли значение введено в то или иное поле, мы узнаем потом.)

Но мы можем задавать для полей ввода дополнительные требования к заносимым в них данным. Выполняется это с помощью особых атрибутов тега `<input>`.

Так, атрибут тега `maxlength` задает предельную длину значения, которое может ввести пользователь, в символах. Если введено значение предельной длины, ввод остальных символов просто блокируется.

```
Имя: <input type="text" id="txtName" size="20" maxlength="30" />
```

Пусть максимальная длина имени будет равна 30 символам.

Если атрибут тега `maxlength` не указан, длина вводимого пользователем значения не ограничена.

Атрибут тега без значения `required` позволяет указать, что в данное поле ввода пользователь обязательно должен ввести значение. Этот атрибут тега можно указать только для обычного поля ввода, поля ввода интернет-адреса и поля ввода адреса электронной почты.

```
Имя: <input type="text" id="txtName" required />
```

Теперь пользователь не отвертится от ввода своего имени!

Атрибуты тега `min`, `max` и `step` можно указать только для поля ввода числа.

- `min` — задает минимальное число, которое может ввести пользователь.
- `max` — задает максимальное число, которое может ввести пользователь.
- `step` — задает "шаг" вводимых значений, или, другими словами, минимальную величину, на которую должны отличаться вводимые в это поле значения.

```
Год рождения:
<input type="number" id="txtYear" min="1900" max="2100" step="1" />
```

Пользователь сможет ввести в это поле только числа от 1900 до 2100, отличающиеся друг от друга минимум на 1.

К сожалению, платформа Metro не проверяет корректность телефонного номера, вводимого в специально предназначенное для этого поле ввода. (Дело в том, что различных форматов телефонных номеров в мире существует очень много, и нет никакой возможности реализовать автоматическую проверку их всех.) В этом случае нам пригодятся два атрибута тега, которые мы сейчас рассмотрим.

Атрибут тега `pattern` позволяет указать формат, которому должно соответствовать введенное в поле значение, или *маску ввода*. Этот атрибут можно указать для обычного поля ввода и поля ввода телефонного номера.

> ***Внимание!***
>
> Маска в атрибуте тега `pattern` указывается в виде регулярного выражения JavaScript. Синтаксис регулярных выражений описан в статье MSDN, расположенной по интернет-адресу **http://msdn.microsoft.com/en-us/library/1400241x(VS.94).aspx**.

```
<input type="tel" id="txtPhone"
pattern="\(\d{3,4}\) \d{2,3}-\d{2}-\d{2}" />
```

Задаем для поля ввода телефонного номера маску `([x]xxx) [x]xx-xx-xx`, где `x` — цифра.

Атрибут тега `title` задает описание формата, в котором должны вводиться данные; это описание впоследствии будет добавлено к стандартному сообщению об ошибке ввода (о нем — чуть позже).

```
<input type="tel" id="txtPhone" pattern="\(\d{3,4}\) \d{2,3}-\d{2}-\d{2}"
title="([x]xxx) [x]xx-xx-xx" />
```

Атрибут тега `placeholder` позволяет указать для поля ввода пример вводимого в него значения. Этот пример будет изначально подставлен в поле ввода и набран более тусклым шрифтом.

```
<input type="tel" id="txtPhone" pattern="\(\d{3,4}\) \d{2,3}-\d{2}-\d{2}"
title="([x]xxx) [x]xx-xx-xx" placeholder="([x]xxx) [x]xx-xx-xx" />
```

Теперь пользователь сразу увидит, в каком формате он должен ввести значение.

Остался всего один вопрос: как узнать, корректно ли были введены данные, и, если нет, какую ошибку допустил пользователь при вводе?

Объект `HTMLInputElement`, представляющий поле ввода, поддерживает метод `checkValidity`. Он не принимает параметров и возвращает `true`, если введенное в поле значение корректно, и `false` в противном случае.

```
txtPhone = document.getElementById("txtPhone");
if (txtPhone.checkValidity()) {
  //Введенное значение корректно
} else {
  //Введенное значение некорректно
}
```

Однако таким образом мы сможем только выяснить, что в поле было введено ошибочное значение, но никак не определить допущенную при его вводе ошибку. Для

этого мы воспользуемся свойствами validationMessage и validity объекта HTMLInputElement.

Свойство validationMessage возвращает текстовое описание допущенной пользователем ошибки, сгенерированное самой платформой Metro, в виде строки.

```
txtPhone = document.getElementById("txtPhone");
if (!(txtPhone.checkValidity())) {
  s = txtPhone.validationMessage;
}
```

Здесь мы проверяем, была ли допущена ошибка ввода, и, если была, помещаем в переменную s описание этой ошибки. Обратим внимание на условие в выражении проверки — мы берем результат, возвращенный методом checkValidity, и инвертируем его (выполняем над ним операцию логического НЕ).

А свойство validity возвращает экземпляр объекта ValidityState, хранящий более развернутые сведения об ошибке ввода. Рассмотрим все полезные для нас свойства этого объекта:

- patternMismatch — возвращает true, если введенное значение не совпадает с маской ввода (атрибут тега pattern), и false в противном случае;
- rangeOverflow — возвращает true, если введенное число больше, чем указанный максимум (атрибут тега max), и false в противном случае;
- rangeUnderflow — возвращает true, если введенное число меньше, чем указанный минимум (атрибут тега min), и false в противном случае;
- stepMismatch — возвращает true, если введенное число не укладывается в заданный "шаг" (атрибут тега step), и false в противном случае;
- tooLong — возвращает true, если длина введенного значения превышает заданную максимальную (атрибут тега maxlength), и false в противном случае;
- typeMismatch — возвращает true, если в специализированное поле ввода занесено некорректное значение (например, в поле ввода интернет-адреса задан некорректный интернет-адрес), и false в противном случае;
- valueMissing — возвращает true, если в обязательное поле ввода (атрибут тега reqiured) не было занесено никакого значения, и false в противном случае;
- valid — возвращает true, если введенное значение полностью корректно, и false в противном случае.

```
txtPhone = document.getElementById("txtPhone");
if (!(txtPhone.checkValidity())) {
  if (txtPhone.validity.patternMismatch)
    //Введенный телефонный номер не соответствует шаблону
  }
}
```

События клавиатуры

Помимо всего описанного ранее, поля ввода поддерживают набор событий, возникающих при вводе данных с клавиатуры. Эти события перечислены далее:

- `keydown` — возникает при нажатии любой клавиши;
- `keypress` — возникает при нажатии алфавитно-цифровой клавиши;
- `keyup` — возникает при отпускании ранее нажатой клавиши.

Функциям-обработчикам этих событий передается в качестве единственного параметра экземпляр объекта `KeyboardEvent`, представляющий сведения о событии. Из всех поддерживаемых им свойств нам будут наиболее полезны следующие:

- `altKey` — возвращает `true`, если пользователь удерживает нажатой клавишу <Alt>, и `false` в противном случае;
- `ctrlKey` — возвращает `true`, если пользователь удерживает нажатой клавишу <Ctrl>, и `false` в противном случае;
- `key` — возвращает строку, содержащую:
  - либо строку с символом, соответствующим нажатой алфавитно-цифровой клавише ("a", "1", "." и т. п.);
  - либо строку с наименованием нажатой специальной клавиши ("Enter", "Tab" и т. п.; наименования различных специальных клавиш можно найти в статье MSDN с интернет-адресом **http://msdn.microsoft.com/en-us/library/windows/apps/gg305568.aspx**);
  - либо строку вида `U<`*`шестнадцатеричный код нажатой клавиши в кодировке Unicode`*`>`;
- `location` — возвращает число, обозначающее местоположение нажатой клавиши:
  - 0 — клавиша находится в основном блоке клавиш и присутствует там в единственном экземпляре (к таким относятся, в частности, алфавитно-цифровые клавиши);
  - 1 — клавиша находится слева (например, левый <Shift>);
  - 2 — клавиша находится справа (например, правый <Shift>);
  - 3 — клавиша находится в дополнительном блоке ("серые" цифровые клавиши);
  - 4 — клавиша, физическая или виртуальная, находится на мобильном устройстве;
  - 5 — клавиша находится на геймпаде или джойстике;
- `metaKey` — возвращает `true`, если пользователь удерживает нажатой клавишу <Meta>, и `false` в противном случае.
- `repeat` — возвращает `true`, если клавиша нажата и удерживается, и `false` в противном случае;
- `shiftKey` — возвращает `true`, если пользователь удерживает нажатой клавишу <Shift>, и `false` в противном случае.

Еще объект KeyboardEvent поддерживает метод preventDefault. Он отменяет действие по умолчанию для события; в случае событий клавиатуры таким действием будет занесение в поле ввода символа, соответствующего нажатой клавише. Параметров этот метод не принимает и результата не возвращает.

```
txtName = document.getElementById("txtName");
txtName.addEventListener("keypress", function (evt) {
  if (evt.key == "0") {
    evt.preventDefault();
  }
});
```

Здесь мы привязываем к событию keypress поля ввода txtName обработчик. Он проверяет, нажата ли клавиша <0>, и, если это так, отменяет действие по умолчанию для данного события. В результате пользователь не сможет ввести в поле ввода символ нуля.

Флажок

Флажок создается с помощью одинарного тега <input>, для атрибута type которого задано значение checkbox.

```
<input type="checkbox" id="chkOption" />
```

По умолчанию флажок формируется сброшенным. Однако если указать в теге <input> атрибут без значения checked, флажок будет изначально установлен.

```
<input type="checkbox" id="chkOption" checked />
```

Помимо этого, флажок поддерживает атрибут тега disabled.

Флажок также представляется объектом HTMLInputElement. Наряду со свойством disabled и событием click, которое можно использовать для отслеживания переключения флажка, он поддерживает свойство checked. Это свойство хранит значение true, если флажок установлен, и значение false, если он сброшен.

```
chkOption = document.getElementById("chkOption");
if (chkOption.checked) {
  //Флажок установлен. Что-то делаем
}
```

Переключатель

Переключатель формируется одинарным тегом <input>, для атрибута type которого задано значение radio.

```
<input type="radio" id="radOption1" />
```

От одного переключателя толку никакого. Поэтому их объединяют в группы. Для этого в каждом переключателе, что входит в группу, указывается атрибут name, в качестве значения которого ставится уникальное *имя группы*.

```
<input type="radio" id="radOption1" name="grp1" />
<input type="radio" id="radOption2" name="grp1" />
<input type="radio" id="radOption3" name="grp1" />
```

Здесь мы создали группу `grp1`, содержащую три переключателя: `radOption1`, `radOption2` и `radOption3`.

Атрибут стиля `checked` позволит сделать один из переключателей группы изначально установленным.

```
<input type="radio" id="radOption2" name="grp1" checked />
```

Переключатель, как и все элементы управления, создаваемые тегом `<input>`, представляются объектом `HTMLInputElement`. Он поддерживает свойства `checked` и `disabled` и событие `click`.

```
radOption1 = document.getElementById("radOption1");
radOption2 = document.getElementById("radOption2");
radOption3 = document.getElementById("radOption3");
if (radOption1.checked) {
  //Первый переключатель установлен
}
if (radOption2.checked) {
  //Второй переключатель установлен
}
if (radOption3.checked) {
  //Третий переключатель установлен
}
```

Область редактирования

Область редактирования создается с помощью другого тега. Это парный тег `<textarea>`, в качестве содержимого которого может указываться изначальное значение, занесенное в область редактирования.

```
<textarea id="txtData"></textarea>
```

Создаем пустую область редактирования.

```
<textarea id="txtData">Введите сюда текст.</textarea>
```

А эта область редактирования будет иметь изначальное значение.

Тег `<textarea>` также поддерживает атрибуты `disabled`, `placeholder`, `readonly` и `required`.

Область редактирования представляется объектом `HTMLTextareaElement`. Из поддерживаемых им свойств нам наиболее интересно свойство `value`, хранящее введенное в область редактирования значение.

```
txtData = document.getElementById("txtData");
s = txtData.value;
```

Помимо этого, объект `HTMLTextareaElement` поддерживает свойства `disabled`, `readOnly`, `validationMessage` и `validity`, метод `checkValidity` и события клавиатуры.

Список

Список также создается с помощью другого тега — <select>. Это парный тег, внутри которого помещаются теги, создающие пункты списка.

```
<select id="lstMode"></select>
```

Атрибут тега size позволяет задать высоту списка в пунктах (позициях списка). Если задать высоту, равную 1, будет сформирован раскрывающийся список.

```
<select id="lstMode" size="10"></select>
```

Создаем список высотой в 10 пунктов.

```
<select id="lstMode" size="1"></select>
```

Создаем раскрывающийся список.

По умолчанию в списке можно выбрать только один пункт. Но если включить в тег <select> атрибут без значения multiple, появится возможность выбора сразу нескольких пунктов. Отметим, что это актуально только для обычных списков; в раскрывающемся списке выбрать сразу несколько пунктов невозможно.

```
<select id="lstMode" size="10" multiple></select>
```

Если в тег <select> включить атрибут без значения required, пользователь должен будет выбрать в списке, по крайней мере, один пункт.

Еще тег <select> поддерживает атрибут disabled.

Для формирования пункта списка применяется парный тег <option>. Внутри этого тега помещается надпись пункта. А сами теги <option> помещаются внутрь тега <select>.

```
<select id="lstMode" size="1">
  <option>Горизонтальный</option>
  <option>Вертикальный</option>
</select>
```

Создаем раскрывающийся список с двумя пунктами.

Атрибут тега value позволяет задать *значение пункта* — строку, которая однозначно идентифицирует пункт. Впрочем, делать это необязательно.

```
<select id="lstMode" size="1">
  <option value="horz">Горизонтальный</option>
  <option value="vert">Вертикальный</option>
</select>
```

Атрибут тега без значения selected позволяет указать, что данный пункт списка должен быть изначально выбран.

```
<select id="lstMode" size="1">
  <option value="horz" selected>Горизонтальный</option>
  <option value="vert">Вертикальный</option>
</select>
```

Существует также возможность объединения пунктов списка в *группы* по какому-либо признаку. Это может быть полезно в больших списках с множеством пунктов, которые могут быть разделены на несколько категорий. Каждая группа имеет текстовый заголовок, указывающий на признак, по которому пункты объединены в данную группу.

Группа пунктов создается с помощью парного тега `<optgroup>`. Обязательный атрибут `label` этого тега задает заголовок группы.

```
<select id="lstMode" size="4" multiple>
  <optgroup label="Типичные режимы">
    <option>Горизонтальный</option>
    <option>Вертикальный</option>
  </optgroup>
  <optgroup label="Нетипичные режимы">
    <option>Диагональный</option>
    <option>Наклонный</option>
  </optgroup>
</select>
```

Список представляется объектом `HTMLSelectElement`. Он поддерживает свойства `disabled`, `validationMessage` и `validity`, метод `checkValidity` и событие `click`, возникающее при выборе пункта списка.

Кроме этого, данный объект поддерживает два очень полезных свойства. Свойство `selectedIndex` хранит числовой номер выбранного в данный момент пункта. Нумерация пунктов списка начинается с нуля; так, первый пункт носит номер 0, второй — 1 и т. д. Если ни один из пунктов списка не выбран, это свойство хранит значение –1.

Свойство `selectedIndex` полезно только в том случае, если в списке можно выбрать лишь один пункт. В противном случае нам придется использовать другие инструменты, чтобы выяснить, какие пункты были выбраны; эти инструменты мы рассмотрим чуть позже.

Второе свойство — `options`. Оно хранит массив экземпляров объекта `HTMLOptionElement`, представляющих пункты списка.

Объект `HTMLOptionElement` поддерживает ряд полезных свойств. Прежде всего, это свойство `selected`. Оно хранит значение `true`, если данный пункт выбран, и `false` в противном случае.

```
var lstMode = document.getElementById("lstMode");
var colItems = lstMode.options;
var selectedItems = [];
for (var i = 0; i < colItems.length; i++) {
  if (colItems[i].selected) {
    selectedItem.push(i);
  }
}
```

Здесь мы получаем массив пунктов созданного ранее списка `lstMode` и, используя цикл со счетчиком, просматриваем его в поисках пунктов, которые были выбраны пользователем. Номера выбранных пунктов мы помещаем в новый массив `selectedItems`.

Свойство `length` массива (точнее, представляющего его объекта `Array`) возвращает его размер. А метод `push` того же объекта добавляет в массив новый элемент, значение которого передано в параметре данного метода, и дает ему числовой индекс, на единицу больший, чем индекс предыдущего элемента. (Это, кстати, еще один способ добавить в массив новый элемент, не ломая голову, какой индекс ему присвоить.)

Другое полезное свойство объекта `HTMLOptionElement` — `value`. Оно хранит значение пункта и соответствует атрибуту тега `value`.

Регулятор

Регулятор создается при помощи давно знакомого нам тега `<input>`, для атрибута `type` которого указано значение `range`.

```
<input type="range" id="sldVolume" />
```

Здесь нам очень пригодятся следующие атрибуты тега:

- `max` — задает максимальное значение, которое можно будет установить с помощью регулятора;
- `min` — задает минимальное значение, которое можно будет установить с помощью регулятора;
- `step` — задает значение "шага" регулятора;
- `value` — задает или возвращает значение, установленное пользователем с помощью регулятора.

Значения всех этих атрибутов тега должны представлять собой целые числа.

```
<input type="range" id="sldVolume" min="0" max="100" step="10"
       value="50" />
```

По умолчанию создается горизонтальный регулятор. Если нам потребуется вертикальный регулятор, мы привяжем к создающему его тегу `<input>` стилевой класс `.win-vertical`.

```
<input type="range" id="sldVolume" class="win-vertical" />
```

Объект `HTMLInputElement`, представляющий регулятор, поддерживает, помимо свойств `disabled` и `value`, еще и событие `change`. Это событие возникает при перемещении пользователем движка регулятора.

```
sldVolume = document.getElementById("sldVolume");
vidMain = document.getElementById("vidMain");
sldVolume.addEventListener("change", function() {
  vidMain.volume = sldVolume.value / 100;
});
```

Здесь мы используем обработчик события `change` регулятора `sldVolume`, чтобы присвоить заданное в нем значение (предварительно разделив его на 100) свойству `volume` видеопроигрывателя `vidMain`. Таким образом, мы создали регулятор громкости.

Вспомогательные элементы управления HTML

Вспомогательные элементы управления применяются для вывода данных и для визуального оформления основных элементов. Их совсем немного.

Вспомогательные элементы управления также являются встроенными элементами.

Индикатор прогресса

Индикатор прогресса создается с помощью одинарного тега `<progress>`.

```
<progress id="prgPlaying" />
```

Платформа Metro позволяет создавать индикаторы прогресса трех разновидностей. Они различаются как внешним видом (рис. 6.1), так и назначением.

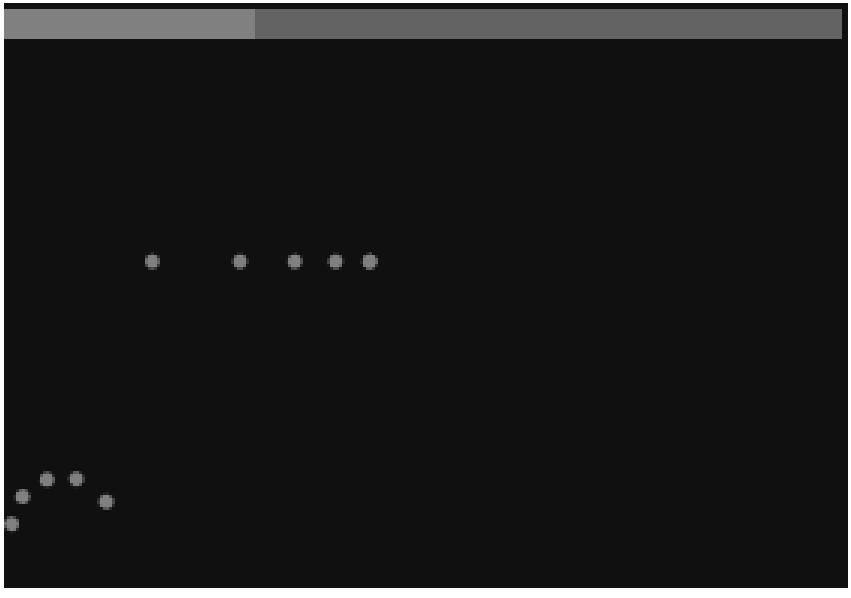


Рис. 6.1. Сверху вниз: индикатор прогресса с определенным состоянием, индикатор прогресса с неопределенным состоянием, кольцевой индикатор прогресса

Прежде всего, это, конечно, *индикатор прогресса с определенным состоянием* (на рис. 6.1 — наверху). Он имеет вид горизонтальной полосы, которая заполняется в направлении слева направо, и служит для показа прогресса операций, продолжительность которых точно известна.

Чтобы создать такой индикатор прогресса, нам потребуется задать для тега `<progress>` два следующих атрибута:

- `max` — максимальное значение, которое может показать индикатор. Понятно, что оно равно продолжительности операции, прогресс которой показывается индикатором;
- `value` — текущее значение индикатора. Обозначает текущее положение хода операции.

Значения обоих этих атрибутов тега задаются в виде целых чисел.

Предположим, что мы хотим отображать с помощью индикатора прогресса ход воспроизведения фильма. (Плохая идея, но в качестве примера сойдет.) Тогда в качестве максимального значения индикатора мы зададим общую продолжительность фильма, а в качестве текущего значения будем задавать позицию его воспроизведения в данный момент времени.

```
<progress id="prgPlaying" max="100000" value="0" />
```

Индикатор прогресса представляется объектом `HTMLProgressElement`. Он поддерживает свойства `max` и `value`, соответствующие одноименным атрибутам тега `<progress>`.

Свойство `max` позволит нам указать максимальное значение индикатора, если изначально оно нам не было известно.

```
prgPlaying = document.getElementById("prgPlaying");
vidMain = document.getElementById("vidMain");
prgPlaying.max = vidMain.duration;
```

Свойство `duration` видеопроигрывателя возвращает продолжительность фильма в миллисекундах.

А чтобы задать новое значение индикатора прогресса, мы присвоим его свойству `value`.

```
prgPlaying.value = vidMain.currentTime;
```

Свойство `currentTime` видеопроигрывателя хранит текущую позицию воспроизведения фильма в миллисекундах.

Кстати, объект `HTMLProgressElement` также поддерживает свойство `position`. Оно возвращает число с плавающей точкой от 0 до 1, показывающее процент выполнения операции.

Индикатор прогресса с неопределенным состоянием (на рис. 6.1 — в центре), напротив, показывает прогресс операций, продолжительность которых вычислить принципиально невозможно. Он выглядит как тонкая горизонтальная линия, по которой бегут маленькие шарики.

Создать такой индикатор прогресса проще всего. Для этого достаточно не указывать в теге `<progress>` атрибутов `max` и `value`.

```
<progress id="prgConnecting" />
```

Наконец, самая эффектная разновидность — *кольцевой индикатор прогресса*, или *кольцо прогресса* (на рис. 6.1 — внизу). Он также показывает прогресс операций, продолжительность которых вычислить принципиально невозможно. Но выглядит при этом как окружность, по которой бегают шарики.

Чтобы создать такой индикатор прогресса, достаточно привязать к создающему его тегу стилевой класс `.win-ring`.

```
<progress id="prgConnecting" class="win-ring" />
```

Мы можем создать кольцевой индикатор прогресса трех доступных размеров:

- малого — создается по умолчанию;
- среднего — для его создания следует дополнительно привязать к тегу `<progress>` стилевой класс `.win-medium`;
- большого — для его создания нужно дополнительно привязать к тегу `<progress>` стилевой класс `.win-large`.

```
<progress id="prgConnecting" class="win-ring win-large" />
```

Создаем индикатор прогресса большого размера.

Надпись

Каждый элемент управления должен иметь текстовую надпись. В противном случае пользователь просто не поймет, какие данные в него следует вводить.

Проще всего дело обстоит с кнопкой — она уже включает надпись. А остальные элементы управления?

Конечно, можно просто поместить текст, составляющий надпись, возле тега, создающего элемент управления.

```
Имя: <input type="text" id="txtName" />
Пароль: <input type="password" id="txtPassword" />
<input type="checkbox" id="chkOption" checked /> Сохранить имя и пароль
```

Но лучше всего использовать для создания надписи специальный тег. Это парный тег `<label>`. Текст надписи помещается внутри данного тега. А в качестве значения обязательного атрибута `for` указывается имя элемента управления, для которого создается надпись.

```
<label for="txtName">Имя:</label>
<input type="text" id="txtName" />

<label for="txtPassword">Пароль:</label>
<input type="password" id="txtPassword" />

<input type="checkbox" id="chkSave" checked />
<label for="chkSave">Сохранить имя и пароль</label>
```

Группа

Группа служит для визуального объединения элементов управления, выполняющих общую функцию. Например, в группу можно объединить поля ввода имени и пароля из предыдущего примера.

Группа имеет вид тонкой рамки, в которую заключаются элементы управления. Помимо этого, группа имеет *заголовок*, обычно располагающийся в ее верхней части.

Группа создается парным тегом `<fieldset>`. Внутри его помещаются теги, создающие элементы управления, что входят в группу.

Для создания заголовка группы служит парный тег `<legend>`. В качестве содержимого данного тега указывается текст заголовка. Сам же тег `<legend>` ставится внутри тега `<fieldset>`, в том месте его содержимого, где должен присутствовать заголовок. (Поскольку обычно он находится в верхней части группы, то создающий его тег `<legend>` нужно поместить в начало содержимого тега `<fieldset>`.)

```
<fieldset>
  <legend>Введите имя и пароль</legend>
  <label for="txtName">Имя:</label>
  <input type="text" id="txtName" />
  <label for="txtPassword">Пароль:</label>
  <input type="password" id="txtPassword" />
</fieldset>
```

Простые всплывающие подсказки

А еще платформа Metro предлагает разработчикам очень простой способ создать у элементов интерфейса всплывающие подсказки. Такие подсказки появляются, если поместить на элемент интерфейса палец и некоторое время его удерживать.

Простую всплывающую подсказку, содержащую только неформатированный текст, можно создать с помощью атрибута тега `title`. Этот атрибут поддерживается всеми тегами, формирующими элементы интерфейса.

Атрибут `title` помещается в тег, создающий элемент интерфейса. В качестве его значения указывается текст всплывающей подсказки.

```
<input type="button" id="btnOpen" value="Открыть"
title="Открыть видеофайл для воспроизведения" />
```

Единственное исключение — если тег создает поле ввода и включает атрибут `pattern`, то значение атрибута тега `title` будет добавлено к тексту сообщения об ошибке ввода. (Подробнее об этом говорилось в *разд. "Проверка корректности введенных данных" ранее в этой главе*.)

Пример приложения: арифметический калькулятор

Напоследок закрепим полученные знания. Создадим Metro-приложение, использующее элементы управления HTML, — простейший калькулятор, выполняющий четыре действия арифметики.

Создадим в Visual Studio новый проект и назовем его `SimpleCalc`. (Как сделать новый проект, описывалось в *главе 2*.) Откроем файл default.html, где описывается интерфейс приложения, и поместим внутри тега `<body>` вот такой код:

```
<input type="text" id="txtOutput" value="0" readonly />
<input type="button" id="btnC" value="C" />
<input type="button" id="btnCE" value="CE" />
<input type="button" id="btn1" value="1" />
<input type="button" id="btn2" value="2" />
<input type="button" id="btn3" value="3" />
<input type="button" id="btn4" value="4" />
<input type="button" id="btn5" value="5" />
<input type="button" id="btn6" value="6" />
<input type="button" id="btn7" value="7" />
<input type="button" id="btn8" value="8" />
<input type="button" id="btn9" value="9" />
<input type="button" id="btn0" value="0" />
<input type="button" id="btnDot" value="." />
<input type="button" id="btnPlus" value="+" />
<input type="button" id="btnMinus" value="-" />
<input type="button" id="btnMult" value="*" />
<input type="button" id="btnDiv" value="/" />
<input type="button" id="btnEqual" value="=" />
```

Здесь все нам уже знакомо. Мы создаем поле ввода `txtOutput`, которое используем в качестве индикатора, и делаем его доступным только для чтения. Также мы создаем набор кнопок для ввода цифр, указания выполняемых действий и очистки поля ввода, кнопок, характерных для обычного калькулятора.

Теперь откроем файл логики default.js и впишем в него код, объявляющий необходимые переменные. Куда именно вписать этот код, говорилось в *главе 5*.

```
var arg1 = "", arg2 = "", op = "", txtOutput;
```

Переменная `arg1` будет хранить тот аргумент операции, что в данный момент набирает пользователь (набираемый аргумент), переменная `arg2` — второй аргумент, а переменная `op` — обозначение самой операции. Все эти значения мы будем хранить в строковом формате и всем трем переменным изначально присвоим пустые строки. Что касается переменной `txtOutput`, то она сохранит экземпляр объекта, представляющий поле ввода `txtOutput`.

При запуске приложения нам потребуется выполнить некоторые предварительные действия: получить доступ к элементам интерфейса, которыми будем часто пользо-

ваться в дальнейшем, и привязать к элементам обработчики событий. (Подробнее об этом рассказывалось в *главе 5*.) Для этого мы создадим такой код:

```
document.addEventListener("DOMContentLoaded", function () {
  txtOutput = document.getElementById("txtOutput");
  var colButtons = document.body.querySelectorAll("input[type=button]");
  for (var i = 0; i < colButtons.length; i++) {
    colButtons[i].addEventListener("click", btnClick);
  }
});
```

Это уже знакомый нам обработчик события `DOMContentLoaded`. В нем мы, прежде всего, получаем доступ к полю ввода `txtOutput`, к которому будем потом постоянно обращаться. Далее с помощью метода `querySelectorAll` ищем все элементы интерфейса, созданные с помощью тега `<input>`, для атрибута `type` которого задано значение `button`, т. е. все кнопки. Напоследок в цикле перебираем все кнопки — элементы полученной от метода `querySelectorAll` коллекции — и к каждой привязываем функцию `btnClick()` в качестве обработчика события `click`.

Да-да, мы привяжем один обработчик события сразу ко всем кнопкам. Конечно, можно написать для каждой кнопки свою функцию-обработчик, но это не очень удобно.

И, наконец, объявим саму функцию `btnClick()`.

```
function btnClick(evt) {
    var oTarget = evt.target;
    switch (oTarget.id) {
        case "btn0":
        case "btn1":
        case "btn2":
        case "btn3":
        case "btn4":
        case "btn5":
        case "btn6":
        case "btn7":
        case "btn8":
        case "btn9":
        case "btnDot":
            arg1 += oTarget.value;
            txtOutput.value = arg1;
            break;
        case "btnPlus":
        case "btnMinus":
        case "btnMult":
        case "btnDiv":
            arg2 = arg1;
            arg1 = "";
            op = oTarget.value;
            break;
```

```
        case "btnEqual":
            if (op != "") {
                var a1 = parseFloat(arg1);
                var a2 = parseFloat(arg2);
                var a3;
                switch (op) {
                    case "+":
                        a3 = a2 + a1;
                        break;
                    case "-":
                        a3 = a2 – a1;
                        break;
                    case "*":
                        a3 = a2 * a1;
                        break;
                    case "/":
                        a3 = a2 / a1;
                        break;
                }
                arg1 = a3.toString();
                txtOutput.value = arg1;
                op = "";
            }
            break;
        case "btnC":
            arg1 = "";
            arg2 = "";
            txtOutput.value = "0";
            break;
        case "btnCE":
            arg1 = "";
            txtOutput.value = "0";
            break;
    }
}
```

Эта функция весьма сложна, так что давайте рассмотрим ее работу по частям.

Поскольку мы привязали одну и ту же функцию-обработчик сразу ко всем кнопкам нашего приложения, то в теле этой функции должны определить, какую именно кнопку нажал пользователь. Сделать это очень просто.

Из *главы 5* мы знаем, что функция-обработчик получает в качестве единственного параметра (у нас этот параметр носит имя `evt`) экземпляр объекта, который хранит сведения о возникшем событии. В том числе и элемент интерфейса, в котором это событие изначально возникло, — в свойстве `target` данного объекта. В нашем случае этим элементом интерфейса будет кнопка, нажатая пользователем.

Каждый объект, представляющий элемент интерфейса, поддерживает свойство `id`, хранящее имя этого элемента, которое мы задали в атрибуте тега `id`. Из той же *гла-*

вы 5 мы знаем, что имена элементов интерфейса должны быть уникальными. Следовательно, обратившись к этому свойству и получив его значение, мы можем однозначно сказать, какая кнопка была нажата.

Для определения нажатой кнопки и выполнения соответствующего ее фрагмента кода можно использовать набор условных выражений. Но удобнее применить для этого выражение выбора. (Об условных выражениях и выражениях выбора *см. в главе 4*.)

Первым делом мы объявляем локальную переменную `oTarget`, в которую помещаем значение свойства `target` экземпляра объекта, хранящего сведения о событии, — нажатую пользователем кнопку. Это позволит нам несколько упростить код.

Далее находится условное выражение, в котором имя нажатой кнопки последовательно сравнивается с именами всех кнопок, составляющих интерфейс нашего приложения.

- Если пользователь нажал кнопку-цифру или кнопку-точку, мы получаем надпись этой кнопки (свойство `value`), т. е. нажатую цифру или точку, добавляем ее к значению набираемого аргумента (это значение хранится в переменной `arg1` в виде строки) и выводим набираемый аргумент в поле ввода `txtOutput`:

```
arg1 += oTarget.value;
txtOutput.value = arg1;
```

- Если пользователь нажал кнопку со знаком арифметического действия, мы присваиваем значение набираемого аргумента (переменная `arg1`) второму аргументу (переменная `arg2`), очищаем набираемый аргумент (присваиваем переменной `arg1` пустую строку), получаем надпись нажатой кнопки — обозначение операции — и сохраняем ее в переменной `op`:

```
arg2 = arg1;
arg1 = "";
op = oTarget.value;
```

- Ситуация, когда пользователь нажимает кнопку со знаком равенства, запуская тем самым вычисление, — самая сложная. Сначала мы проверяем, хранит ли переменная `op` обозначение операции (любое значение, отличное от пустой строки). Это нужно, чтобы исключить ситуации, когда пользователь нажимает кнопку со знаком равенства, не указав саму операцию.

```
if (op != "") {
```

  Далее, значения аргументов в переменных `arg1` и `arg2` мы храним в строковом виде, и перед вычислением результата нам придется преобразовать их в числа. Если мы этого не сделаем, при выполнении сложения платформа Metro не сложит представляемые этими аргументами числа, а объединит их в одну строку.

  Нам поможет функция `parseFloat()`, о которой уже упоминалось в *главе 4*. Вспомним: она принимает в качестве аргумента строку, содержащую число с плавающей точкой, и возвращает это число в числовом виде в качестве результата. То, что нам нужно!

```
    var a1 = parseFloat(arg1);
    var a2 = parseFloat(arg2);
    var a3;
```

Здесь мы преобразуем строковые значения обоих аргументов в числовой вид и помещаем их во вновь объявленные переменные `a1` и `a2`. Также мы объявляем переменную `a3`, куда потом поместим результат вычисления.

Обозначение выбранной пользователем операции мы храним в переменной `op`. Чтобы выяснить, что это за операция, и выполнить соответствующий ей фрагмент кода, мы применим выражение выбора:

```
    switch (op) {
        case "+":
            a3 = a2 + a1;
            break;
        case "-":
            a3 = a2 – a1;
            break;
        case "*":
            a3 = a2 * a1;
            break;
        case "/":
            a3 = a2 / a1;
            break;
    }
```

Результат выполнения действия теперь хранится в переменной `a3`. Преобразуем его в строковый вид с помощью метода `toString`, который также упоминался в *главе 4*. (Этот метод возвращает в качестве результата строковое представление экземпляра данного объекта, в нашем случае — строку, содержащую число.) Сразу присвоим полученную строку набираемому аргументу (переменной `arg1`):

```
    arg1 = a3.toString();
```

Напоследок выведем результат в поле ввода `txtOutput` и очистим переменную, где хранится обозначение операции (`op`):

```
    txtOutput.value = arg1;
    op = "";
}
```

- Если пользователь нажал кнопку полного сброса (**C**), мы очищаем значения обоих аргументов — и набираемого, и второго, после чего выводим в поле ввода `txtOutput` число 0:

```
arg1 = "";
arg2 = "";
txtOutput.value = "0";
```

- Наконец, если пользователь нажал кнопку сброса набираемого аргумента (**CE**), мы очищаем только значение набираемого аргумента и также выводим в поле ввода `txtOutput` число 0:

```
arg1 = "";
txtOutput.value = "0";
```

Сохраним все файлы, запустим приложение и проверим его в действии. Конечно, пока что его интерфейс совсем не впечатляет (рис. 6.2), но со временем мы это исправим.

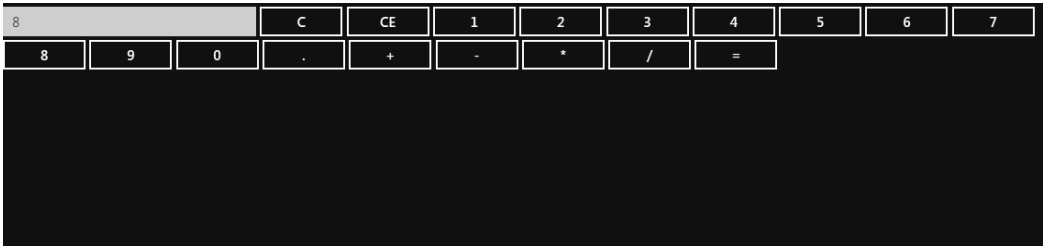


Рис. 6.2. Интерфейс приложения `SimpleCalc` — простейшего калькулятора

Как видим, мы создали вполне функциональное и полезное Metro-приложение, написав совсем немного кода. В этом заслуга языков HTML и JavaScript, предоставляющих нам весьма мощные и при этом простые в использовании средства, позволяющие создавать элементы управления и манипулировать их параметрами. Удачно, что для создания приложений мы выбрали именно эти языки!

Что дальше?

В этой главе мы знакомились с элементами управления, поддерживаемыми самим языком HTML: кнопками, полями ввода, флажками, переключателями, областями редактирования, списками, регуляторами, индикаторами прогресса, надписями и группами. Пожалуй, во многих случаях Metro-программирования можно обойтись только ими.

А в следующей главе мы познакомимся с элементами управления, чья работа обеспечивается уже платформой Metro. Такие элементы управления представляют собой сложную комбинацию тегов HTML, стилей CSS и кода JavaScript, и работать с ними заметно сложнее. Но результат того стоит!

ГЛАВА 7

Элементы управления Metro

Предыдущая глава была посвящена элементам управления, поддерживаемым самим языком HTML. Эти элементы управления создаются с помощью одного-единственного специализированного тега и для успешной работы не требуют никакого дополнительного кода.

Конечно, для многих Metro-приложений вполне хватит элементов управления HTML. Но случаются ситуации, когда их оказывается недостаточно. Например, если мы собираемся создавать приложение, манипулирующее значениями даты, нам понадобится элемент управления для ввода этих значений. Но язык HTML не позволяет создать такой элемент.

Введение в элементы управления Metro

Решение этой проблемы было найдено довольно давно. Заключается оно в том, что элементы управления, не поддерживаемые HTML, имитируются с помощью элементов, которые им поддерживаются. При этом они оформляются особым образом, чтобы выглядеть как части единого элемента управления, и особым же образом функционируют; для всего этого используются специальные стили CSS и специальный JavaScript-код.

Например, элемент для ввода даты в простейшем случае имитируется с помощью трех списков, в которых выбираются, соответственно, число, месяц и год. К этим элементам привязываются стили CSS, задающие для них соответствующее оформление, и обработчики событий, которые обеспечивают их совместное функционирование.

Этот же подход используется и в платформе Metro для формирования элементов управления, известных как элементы управления Metro. Именно таким элементам и будет посвящена данная глава.

Здесь мы рассмотрим создание самых простых элементов управления Metro: элементов для ввода значений даты, времени и рейтинга и выключателя, а также панели вывода. Более сложные и специализированные элементы Metro мы рассмотрим в последующих главах этой книги.

Основные принципы работы с элементами управления Metro

Все элементы управления Metro создаются и обрабатываются согласно сходным принципам. Давайте сразу же рассмотрим эти принципы, чтобы потом к ним не возвращаться.

Создание элементов управления Metro

Все элементы управления Metro создаются в три этапа. Рассмотрим их.

На первом этапе в то место, где должен присутствовать элемент Metro, вставляется "заготовка", на основе которого он будет создан, или *элемент-основа*. Этой "заготовкой" служит обычный блок — тег `<div>`.

Кстати, поскольку все элементы управления Metro создаются на основе блока, то все они являются блочными элементами. В этом одно из ключевых отличий элементов Metro от элементов HTML.

На втором этапе в тег `<div>` помещается атрибут `data-win-control`. В качестве значения данного атрибута тега указывается имя объекта, который представляет нужный нам элемент управления.

На третьем, необязательном, этапе задаются параметры создаваемого элемента управления, фактически — значения свойств экземпляра объекта, что его представляет. Они указываются в качестве значения атрибута тега `data-win-options`.

Параметры элемента задаются в виде пар *<имя свойства>*: *<значение>*; строковые *значения* при этом берутся не в двойные, а в одинарные кавычки. Весь набор пар, задающих свойства элемента, берется в фигурные скобки; сами пары отделяются друг от друга запятыми.

Вот пример кода, создающего элемент управления Metro, который предназначен для ввода даты:

```
<div id="divDate" data-win-control="WinJS.UI.DatePicker"
data-win-options="{minYear: 1900, maxYear: 2300}"></div>
```

Здесь мы дополнительно задаем для этого элемента минимальное и максимальное значения года (свойства `minYear` и `maxYear` соответственно).

Инициализация элементов управления Metro

Что ж, основы для элементов управления Metro мы создали. Теперь нам нужно выполнить их инициализацию, в процессе которой Metro собственно сформирует эти элементы и подготовит их к работе.

Откроем файл default.js, отыщем обработчик события `DOMContentLoaded` или создадим его, если еще не сделали этого. И вставим в самом его начале вот такое выражение (выделено полужирным шрифтом):

```
document.addEventListener("DOMContentLoaded", function () {
    WinJS.UI.processAll();
    . . .
});
```

Мы вызвали метод `processAll` объекта `WinJS.UI` без параметров.

Что касается объекта `WinJS.UI`, то он содержит набор служебных методов, которыми мы будем часто пользоваться в дальнейшем. Единственный экземпляр этого объекта создается самой платформой Metro и хранится в переменной `WinJS.UI`.

Работа с элементами управления Metro

Когда мы начнем работать с элементами управления Metro в коде JavaScript, то столкнемся с еще одной проблемой. Если мы попытаемся получить к нему доступ, как делали это ранее

```
divDate = document.getElementById("divDate");
```

то получим не сам элемент управления, а его элемент-основу — блок `divDate`. Что, в принципе, вполне понятно...

Получить доступ к самому элементу управления Metro можно с помощью свойства `winControl`. Это свойство поддерживается всеми объектами, представляющими элементы интерфейса.

```
ctrDate = divDate.winControl;
```

Получаем элемент для ввода даты, сформированный на основе блока `divDate`, и помещаем его в переменную `ctrDate`.

```
ctrDate = document.getElementById("divDate").winControl;
```

Выполняем те же действия в одном выражении — так проще и быстрее.

```
d = ctrDate.current;
```

Сохраняем в переменной `d` дату, установленную пользователем в элементе выбора даты. Эта дата хранится в свойстве `current` данного элемента.

Элементы управления Metro, помимо свойств и методов, поддерживают и события. Обработчики этих событий могут быть установлены как с помощью привычного нам метода `addEventListener`, так и присвоением функции-обработчика свойству, представляющему это событие.

```
ctrDate.addEventListener("change", ctrDateChange);
```

Привязываем функцию `ctrDateChange()` в качестве обработчика события `change` элемента для выбора даты `ctrDate`.

Основные элементы управления Metro

По аналогии с элементами управления HTML из *главы 6*, элементы Metro мы также разделим на основные и вспомогательные. Основные элементы, которые мы сейчас рассмотрим, предназначены для ввода данных.

Элемент для ввода даты

Элемент Metro для ввода значения даты представляет собой комбинацию из трех раскрывающихся списков, в которых выбираются месяц, число и год (рис. 7.1).

Рис. 7.1. Элемент для ввода даты

Этот элемент представляется объектом `WinJS.UI.DatePicker`. Именно его имя указывается в качестве значения для атрибута `data-win-control` тега `<div>`, что создаст для него элемент-основу.

Пример кода, создающего элемент для ввода даты, мы приводить не будем. Ранее, изучая работу с элементами управления Metro, мы уже рассмотрели несколько примеров.

Так что давайте сразу перейдем к свойствам объекта `WinJS.UI.DatePicker`, которые могут быть нам полезны.

- `current` — хранит значение даты, заданное в этом элементе управления, в виде экземпляра объекта `Date` (об этом объекте рассказывалось в *главе 4*). Причем имеет смысл только дата, хранящаяся в этом экземпляре объекта; время может быть любым. Значение по умолчанию — текущая дата.

  Для указания значения этого свойства в HTML-коде следует использовать строку формата *<месяц>/<число>/<год>*.

- `maxYear` — хранит максимальное значение года, которое может выбрать пользователь, в виде числа. Значение по умолчанию — текущий год + 100.

- `minYear` — хранит минимальное значение года, которое может выбрать пользователь, в виде числа. Значение по умолчанию — текущий год – 100.

Также этот элемент поддерживает знакомое нам свойство `disabled`.

```
<div id="divDate" data-win-control="WinJS.UI.DatePicker"
data-win-options="{current: '01/01/2012', minYear: 2000, maxYear: 2100}">
</div>
```

Задаем в качестве текущей даты 1 января 2012 года, 2000 год в качестве минимального и 2100-й — в качестве максимального.

Событие `change` возникает, когда пользователь выбирает в элементе управления новую дату.

Элемент для ввода времени

Элемент для ввода значения времени также представляет собой комбинацию из трех раскрывающихся списков (рис. 7.2). В них выбираются, соответственно, часы, минуты и признак того, задано ли время до полудня (AM) или после (PM).

Рис. 7.2. Элемент для ввода времени

Элемент для ввода времени представляется объектом `WinJS.UI.TimePicker`.

```
<div id="divTime" data-win-control="WinJS.UI.TimePicker"></div>
```

Полезных для нас свойств объект `WinJS.UI.TimePicker` поддерживает всего два.

- `current` — хранит значение времени, заданное в этом элементе управления, в виде экземпляра объекта `Date`. Имеет смысл только время, хранящееся в этом экземпляре объекта; дата может быть любой.

  ```
  var ctrTime = document.getElementById("divTime").winControl;
  ctrTime.current = new Date(2012, 1, 1, 8, 30);
  ```

  Устанавливаем для созданного нами элемента изначальное время, равное 8:30.

  Для задания изначального значения времени в HTML-коде мы используем строку в формате `<часы>:<минуты>:<секунды>`. Значение секунд может быть любым — все равно в элементе ввода времени оно никак не отображается.

  ```
  <div id="divTime" data-win-control="WinJS.UI.TimePicker"
  data-win-options="{current: '08:30:00'}"></div>
  ```

- `minuteIncrement` — хранит значение шага между значениями минут, в виде числа. Например, если задать для этого свойства значение 10, то пользователь сможет выбрать только значения минут 10, 20, 30 и т. д. Значение по умолчанию — 1.

Еще элемент ввода времени поддерживает свойство `disabled`.

Событие `change` возникает в данном элементе, когда пользователь выбирает новое значение времени.

Переключатель

Переключатель Metro (рис. 7.3) чем-то похож на флажок. Он позволяет указать значение параметра, который может принимать только два состояния: включено и выключено. Однако, в отличие от флажка, переключатель служит для немедленного изменения значения этого параметра, например для немедленной активизации или деактивизации какой-то функции приложения.

Рис. 7.3. Переключатели: отключенный *(сверху)* и включенный *(снизу)*

Переключатель представляется объектом `WinJS.UI.ToggleSwitch`.

```
<div id="divCam" data-win-control="WinJS.UI.ToggleSwitch"></div>
```

Вот свойства данного объекта, которые могут оказаться для нас полезными.

- `checked` — хранит значение `true`, если переключатель включен, а `false`, если отключен. Значение по умолчанию — `false` (т. е. переключатель изначально отключен).
- `labelOff` — хранит текст, который появляется на отключенном переключателе, в виде строки. Значение по умолчанию — пустая строка (т. е. надписи нет).
- `labelOn` — хранит текст, который появляется на включенном переключателе, в виде строки. Значение по умолчанию — пустая строка.
- `title` — хранит надпись для переключателя в виде строки; эта надпись выводится левее переключателя. Значение по умолчанию — пустая строка (т. е. надпись отсутствует; вместо нее отображаются надписи "Выкл."/"Вкл." ("Off"/"On") в зависимости от состояния переключателя).

Разумеется, поддерживается и свойство `disabled`.

Событие `change` возникает, когда пользователь переводит переключатель в другое состояние: включает отключенный переключатель или выключает включенный.

```
<div id="divCam" data-win-control="WinJS.UI.ToggleSwitch"
data-win-options="{labelOff: 'Нет', labelOn: 'Да',
↳title: 'Активизировать встроенную камеру?', change: divCamChange}">
</div>
```

Здесь мы, помимо всего прочего, задаем обработчик события `change`, который в зависимости от нового состояния переключателя активизирует или деактивизирует встроенную камеру. И обратим внимание, как мы задали строковые значения — заключив их в одинарные кавычки (если заключить их в двойные кавычки, при запуске приложения возникнет ошибка).

Элемент для ввода рейтинга

Элемент для ввода рейтинга (рис. 7.4) может быть актуален для многих приложений: мультимедийных проигрывателей, утилит для чтения новостей и статей из Интернета и др. Он представляет собой шкалу, состоящую из звездочек и отображающую средний рейтинг данного материала; пользователь может щелкнуть на этой шкале и установить тем самым новое значение рейтинга.

Рис. 7.4. Элемент для ввода рейтинга

Элемент для ввода рейтинга представляется объектом `WinJS.UI.Rating`.

```
<div id="divRat" data-win-control="WinJS.UI.Rating"></div>
```

Далее перечислены полезные свойства объекта `WinJS.UI.Rating`.

- `averageRating` — хранит значение среднего рейтинга в виде числа с плавающей точкой от 1 до величины максимального рейтинга (о нем — чуть позже).

- `enableClear` — хранит значение `true`, если у пользователя есть возможность очистить рейтинг, и `false`, если пользователь такой возможности не имеет. Значение по умолчанию — `true`.
- `maxRating` — хранит максимальное значение рейтинга, которое может установить пользователь, в виде ненулевого целого числа (фактически — количество звездочек в шкале этого элемента). Значение по умолчанию — 5.
- `tooltipStrings` — хранит массив строк, каждая из которых представляет всплывающую подсказку для одного из делений (звездочек) шкалы данного элемента. Элементов в массиве должно быть столько же, сколько звездочек присутствует в шкале, или, другими словами, их количество должно совпадать с максимальным значением рейтинга (свойство `maxRating`). Также в этот массив может быть добавлен еще один элемент — для всплывающей подсказки, которая будет выводиться на экран, если рейтинг очищен. Значение по умолчанию для этого свойства — `null` (т. е. массив подсказок отсутствует; в этом случае будут выводиться подсказки, содержащие числовые значения рейтинга для каждой из звездочек в шкале).
- `userRating` — хранит значение рейтинга, установленное пользователем, в виде целого числа от 0 до максимального значения рейтинга. Значение по умолчанию — `null` (т. е. пользовательский рейтинг вообще не задан).

Еще объект `WinJS.UI.Rating` поддерживает свойство `readOnly`.

```
var ctrRat = document.getElementById("divRat").winControl;
ctrRat.averageRating = 2.7;
ctrRat.maxRating = 3;
ctrRat.tooltipStrings = ["Кисло", "Средне", "Круто", "Пока неизвестно"];
```

Здесь мы устанавливаем параметры для созданного ранее элемента ввода рейтинга в JavaScript-коде. Мы задаем средний рейтинг, равный 2,7, и максимальный рейтинг, равный 3. Также мы создаем массив всплывающих подсказок из четырех элементов; первые три будут отображаться, соответственно, для первого, второго и третьего деления (звездочки) шкалы, а четвертое — в случае если пользователь очистит рейтинг.

Что касается событий, то элемент для ввода рейтинга поддерживает целых три:

- `change` — возникает после того, как пользователь установит и зафиксирует новое значение рейтинга;
- `previewchange` — возникает, если пользователь установит новое значение рейтинга, но не зафиксирует его;
- `cancel` — возникает, если пользователь не стал фиксировать новое значение рейтинга.

```
<div id="divRat" data-win-control="WinJS.UI.Rating"
data-win-options="{change: ctrRatChange}"></div>
. . .
```

```
function ctrRatChange() {
  . . .
  newRating = ctrRat.userRating;
  . . .
}
```

Здесь мы задаем для нашего элемента ввода рейтинга обработчик события `change` и в его теле получаем новое значение рейтинга, введенное посетителем.

Панель вывода

Что касается вспомогательных элементов управления Metro, то наиболее полезным из них для нас будет так называемая панель вывода.

Панель вывода (в терминологии Metro — viewbox) устанавливает размеры своего единственного потомка таким образом, чтобы этот потомок занимал все пространство внутри данного элемента. При этом панель вывода сама следит за тем, чтобы соотношение сторон потомка не исказилось. Настоящая находка для разработчиков, собирающихся создавать приложения для просмотра графики и видео!

Панель вывода представляется объектом `WinJS.UI.ViewBox`.

```
<div id="divVB" data-win-control="WinJS.UI.ViewBox">
  <video id="vidMain" autoplay controls></video>
</div>
. . .
#divVB {
  width: 100%;
  height: 100%;
}
```

Здесь мы создали панель просмотра, поместили в нее видеопроигрыватель и привязали к ней стиль, задающий такие размеры, чтобы она заняла все свободное пространство на экране (значение `100%` для атрибутов стиля `width` и `height`).

Пример 1: усовершенствованный видеопроигрыватель

Теперь, чтобы попрактиковаться в создании элементов управления Metro, давайте усовершенствуем видеопроигрыватель, созданный еще в *главе 2*. Ведь не дело, когда фильм, вместо того чтобы заполнить все свободное пространство, сиротливо жмется где-то в углу...

Откроем решение `VideoPlayer` и сразу же переключимся на файл default.html. Удалим из тега `<body>` весь код, что мы создали ранее, и введем туда вот что:

```
<div id="divVideo" data-win-control="WinJS.UI.ViewBox">
  <video id="vidMain" autoplay controls></video>
</div>
```

```
<div id="divButtons">
  <input type="button" id="btnOpen" value="Открыть" />
</div>
```

Здесь мы поместили видеопроигрыватель в блок `divVideo` и сформировали на его основе панель вывода. Помимо этого, мы дали имя `divButtons` второму блоку, в котором находится пока что единственная кнопка нашего приложения. Впоследствии мы привяжем к обоим блокам именованные стили.

После этого откроем файл с описанием оформления default.css, удалим оттуда созданный ранее стилевой класс `.button-cont` и создадим вместо него два именованных стиля. Рассмотрим их по очереди.

Именованный стиль `#divButtons` будет привязан к блоку с кнопкой:

```
#divButtons {
  text-align: right;
  width: 100%;
  height: 32px;
}
```

Здесь мы установили выравнивание для содержимого этого блока по правому краю, растянули блок по ширине на все свободное пространство (значение `100%` для атрибута стиля `width`) и задали для него высоту, равную 32 пикселам (значение `32px` для атрибута стиля `height`). 32 пикселов хватит, чтобы полностью вместить кнопку.

Именованный стиль `#divVideo` будет привязан к блоку с видеопроигрывателем:

```
#divVideo {
  width: 100%;
  height: calc(100% — 32px);
}
```

Этот блок мы также растянули по ширине на все свободное пространство, а в качестве значения его высоты задали полную высоту свободного пространства минус 32 пиксела (это высота блока с кнопками `divButtons`). Функция CSS с именем `calc` позволяет нам выполнить вычисления со значениями размеров.

Осталось переключиться на файл логики default.js, найти обработчик события `DOMContentLoaded` и вставить в самое его начало код, запускающий формирование элементов управления Metro (выделен полужирным шрифтом):

```
document.addEventListener("DOMContentLoaded", function () {
  WinJS.UI.processAll();
  btnOpen = document.getElementById("btnOpen");
  . . .
}
```

Сохраним все исправленные файлы и запустим приложение на выполнение. Выберем какой-либо видеофайл и посмотрим, что получилось (рис. 7.5). Что ж, стало заметно лучше, не так ли?

Рис. 7.5. Интерфейс видеопроигрывателя после усовершенствования

Пример 2: калькулятор значений даты

Второй пример, что мы создадим, будет представлять собой калькулятор значений даты. Пользователь выберет в нем значение даты, укажет, сколько дней к нему следует прибавить, после чего приложение добавит это количество к выбранной дате и выдаст результат также в виде значения даты.

Создадим в Visual Studio новый проект с именем `DateCalc`. Переключимся на файл default.html и введем внутрь тега `<body>` следующий код:

```
<div id="divDate" data-win-control="WinJS.UI.DatePicker"></div>
<div>
  <label for="txtDays">Прибавить, дней</label>
  <input type="number" id="txtDays" value="0" />
</div>
<div>
  <input type="button" id="btnCalculate" value="=" />
</div>
<div id="divResult" data-win-control="WinJS.UI.DatePicker"></div>
```

Здесь мы сформировали:

- элемент для ввода даты `divDate`, в который пользователь будет заносить значение даты:

- поле ввода `txtDays`, куда пользователь занесет количество дней, с надписью;
- кнопку `btnCalculate`, запускающую процесс вычисления результата;
- элемент для ввода даты `divResult`, в котором будет выводиться результат вычисления.

Поскольку поле ввода и кнопка являются встроенными элементами, мы вложили их в блоки. В результате все элементы управления нашего нового приложения будут выстроены на экране друг за другом по направлению сверху вниз.

Закончив описывать интерфейс, займемся оформлением. Давайте уменьшим ширину поля ввода `txtDays`, поскольку изначально она слишком велика (целых 256 пикселов). Откроем файл default.css и создадим именованный стиль `#txtDays`. Понятно, что он будет привязан к одноименному полю ввода.

```
#txtDays { width: 64px; }
```

64 пикселов вполне хватит, чтобы вместить двузначное значение.

"На закуску" осталось самое главное — логика. Переключимся на файл default.js и введем код, объявляющий все необходимые нам переменные.

```
var ctrDate, txtDays, btnCalculate, ctrResult;
```

Напишем код, привязывающий к событию `DOMContentLoaded` обработчик.

```
document.addEventListener("DOMContentLoaded", function () {
  WinJS.UI.processAll();

  txtDays = document.getElementById("txtDays");
  ctrDate = document.getElementById("divDate").winControl;
  ctrResult = document.getElementById("divResult").winControl;
  btnCalculate = document.getElementById("btnCalculate");

  btnCalculate.addEventListener("click", btnCalculateClick);
});
```

Здесь мы сначала получаем доступ к элементам интерфейса, с которыми будем "общаться" в дальнейшем. После этого мы получаем доступ к элементам управления Metro с помощью вызовов метода `getControl` (за подробностями обращайтесь в начало этой главы). Напоследок мы привязываем обработчик события `click` кнопки `btnCalculate`, который собственно выполнит вычисление.

На очереди — функция `btnCalculateClick()`, обработчик события `click` кнопки `btnCalculate`, который и будет выполнять вычисление нового значения даты. Вот код, объявляющий эту функцию:

```
function btnCalculateClick() {
  var d = ctrDate.current;
  var nDays = parseInt(txtDays.value);
  var nD = d.getDate();
  d.setDate(nD + nDays);
  ctrResult.current = d;
}
```

Здесь мы выполняем следующие действия:

1. Получаем дату, указанную пользователем в элементе `divDate` (соответствующий ему элемент управления Metro хранится в переменной `ctrDate`).
2. Преобразуем количество дней, введенное в поле ввода `txtDays`, в целое число с помощью функции `parseInt()` (она упоминалась в *главе 4*).
3. Получаем из введенной пользователем даты число, т. е. номер дня месяца. Для этого мы используем метод `getDate` объекта `Date`; он не принимает параметров и возвращает в качестве результата число в целочисленном виде.
4. Складываем полученное число с количеством дней и устанавливаем полученную сумму в качестве нового числа для даты. Сделать это позволит метод `setDate` объекта `Date`; в качестве единственного параметра он принимает новое значение в виде целочисленной величины и не возвращает результата. Причем если значение нового числа превышает количество дней в текущем месяце, "лишние" дни будут переведены в следующий месяц; в этом смысле объект `Date` весьма "разумен".
5. Выводим новое значение даты в элемент, предназначенный для вывода даты (`ctrResult`).

   По идее, этот элемент следует сделать доступным только для чтения, если уж он используется исключительно для вывода результата. Но, к сожалению, платформа Metro не позволяет этого сделать. Можно, конечно, сделать его недоступным для ввода, присвоив его свойству `disabled` значение `true`, но получится совсем некрасиво. Так что давайте уже не будем трогать этот элемент.

Все! Сохраним все файлы, запустим приложение на выполнение и попробуем его в действии. Выглядит оно, конечно, пока неважно (рис. 7.6), но ведь мы еще не занимались разметкой...

Рис. 7.6. Интерфейс калькулятора для вычисления значений даты

На этом мы пока закончим рассмотрение элементов управления Metro. В последующих главах мы познакомимся с более сложными и специализированными элементами и обязательно опробуем их в реальных приложениях.

Что дальше?

Эта глава была посвящена элементам управления Metro, точнее, самым простым из них. Мы выяснили, как создаются и используются в работе элементы для ввода даты, времени и рейтинга, переключатель, панель вывода и сложная всплывающая подсказка. И по ходу дела усовершенствовали свое первое приложение и создали новое, уже третье по счету.

В следующей главе мы рассмотрим средства HTML по выводу на экран обычного текста и его форматирования. Нам это очень пригодится, ведь результат работы многих приложений представляет собой именно текст.

ГЛАВА 8

Вывод и форматирование текста

В предыдущей главе мы обсуждали элементы управления Metro, их создание и использование в приложениях. Теперь мы знаем, как получить от пользователя данные для обработки.

Настала пора узнать, как вывести результат обработки этих данных. А в очень и очень многих случаях этот результат выводится в виде текста.

Конечно, его можно вывести в какой-либо элемент управления, например поле ввода, область редактирования или элемент для ввода даты. Но если результирующие данные достаточно объемисты, и существует необходимость их отформатировать (разбить на абзацы, добавить заголовки, выделить как-то часть текста), элемент управления однозначно не подойдет. В этом случае выход один — выводить данные напрямую.

Язык HTML предоставляет богатые возможности по выводу текста, а язык CSS поддерживает не менее функциональные инструменты для его оформления. В этой главе мы познакомимся с базовыми инструментами обоих этих языков, которых нам хватит в большинстве случаев.

Структурирование текста

И первым, чем мы займемся, станут средства для структурирования текста — разбиения его на отдельные относительно независимые фрагменты.

Абзацы

Прежде всего, любой текст для удобства чтения разбивают на отдельные абзацы. Каждый такой абзац содержит несколько предложений, выражающих одну мысль или связанных по смыслу.

Еще в *главе 3* мы узнали, что для создания абзаца используется парный тег `<p>`. Содержимое этого тега становится текстом абзаца.

```
<p>Платформа Metro входит в состав Windows 8.</p>
<p>Платформа Metro служит для разработки и выполнения приложений,
изначально предназначенных для планшетов.</p>
```

Абзац является блочным элементом; это значит, что абзацы выводятся последовательно в направлении сверху вниз, как и в обычной "бумажной" книге. Каждый абзац отделяется небольшим отступом от предыдущего и последующего элементов интерфейса. Если абзац по ширине не помещается в родительский элемент полностью, его текст будет разбит на несколько коротких строк.

Заголовки

Большой текст для удобства чтения и поиска в нем нужного фрагмента обычно делят на более крупные части: параграфы, главы, разделы. Каждая такая часть предваряется заголовком.

Заголовок имеет определенный *уровень*, указывающий, насколько крупную часть текста он открывает. Язык HTML поддерживает шесть уровней заголовка, обозначаемых числами от 1 до 6.

- Заголовок первого уровня (1) открывает самую крупную часть текста. Как правило, это название книги, статьи или самого Metro-приложения. Платформа Metro выводит заголовок первого уровня самым большим шрифтом.
- Заголовок второго уровня (2) открывает более мелкую часть текста, обычно большой раздел книги, статьи или фрагмент интерфейса приложения. Платформа Metro выводит заголовок второго уровня меньшим шрифтом, чем заголовок первого уровня.
- Заголовок третьего уровня (3) открывает еще более мелкую часть текста; обычно главу в разделе. Metro выводит такой заголовок еще меньшим шрифтом.
- Заголовки четвертого, пятого и шестого уровней (4–6) открывают отдельные параграфы, крупные, более мелкие и самые мелкие соответственно. Платформа Metro выводит заголовки четвертого и пятого уровня еще меньшим шрифтом, а заголовок шестого уровня — тем же шрифтом, что и обычные абзацы, только полужирным.

Наиболее часто используются заголовки первого, второго и третьего уровней. Меньшие уровни заголовков применяются только в очень больших и сложно структурированных текстах.

Для создания заголовка служит парный тег <h*n*>, где *n* — уровень заголовка. Содержимое этого тега станет текстом заголовка.

```
<h1>Metro: разработка приложений для мобильных устройств</h1>
<h2>Разработка интерфейса</h2>
<h3>Создание элементов управления</h3>
<h4>Создание элементов управления HTML</h4>
<h5>Основные элементы управления HTML</h5>
<h6>Кнопки</h6>
```

Здесь мы написали структуру заголовков для гипотетической книги по Metro-программированию. Заголовок первого уровня относится ко всей книге, заголовок второго уровня — к большому разделу и т. д.

Заголовок также относится к блочным элементам. При его выводе на экран платформа Metro следует тем же правилам, что и в случае абзаца.

Списки

Списки используются для того, чтобы представить пользователю перечень каких-либо позиций, пронумерованных или непронумерованных, — пунктов списка. Список с пронумерованными пунктами так и называется — *нумерованным*, а с непронумерованными — *маркированным*. В маркированных списках пункты помечаются особым значком — *маркером*, который ставится левее пункта списка.

Маркированные списки обычно служат для простого перечисления каких-либо позиций, порядок следования которых не важен. Если же необходимо обратить внимание читателя на то, что позиции должны следовать друг за другом именно в том порядке, в котором они перечислены, нужно применить нумерованный список.

Любой список создается в два этапа. Сначала пишут строки, которые станут пунктами списка, и каждую из этих строк помещают внутрь парного тега `<li>`. Затем все эти пункты помещают внутрь парного тега `<ul>` (если создается маркированный список) или `<ol>` (в случае создания нумерованного списка) — эти теги формируют сам список.

```
<ul>
  <li>Элементы управления HTML.</li>
  <li>Элементы управления Metro.</li>
</ul>
```

Этот маркированный список из двух пунктов перечисляет все элементы управления, поддерживаемые Metro.

```
<ol>
  <li>Создаем элемент-основу.</li>
  <li>Указываем имя объекта.</li>
  <li>Задаем параметры.</li>
</ol>
```

А этот нумерованный список перечисляет все шаги, необходимые для создания элемента управления Metro и рассмотренные нами в *главе 7*.

Списки можно помещать друг в друга, создавая *вложенные списки*. Делается это следующим образом. Сначала во "внешнем" списке (в который должен быть помещен вложенный) отыскивают пункт, после которого должен находиться вложенный список. Затем HTML-код, создающий вложенный список, помещают в разрыв между текстом этого пункта и его закрывающим тегом `</li>`. Если же вложенный список должен помещаться в начале "внешнего" списка, его следует вставить между открывающим тегом `<li>` первого пункта "внешнего" списка и его текстом. Что, впрочем, логично.

```
<ul>
  <li>Элементы управления HTML:
    <ul>
      <li>Кнопка.</li>
      <li>Поле ввода.</li>
      <li>Флажок.</li>
      . . .
    </ul>
  </li>
  <li>Элементы управления Metro
    <ul>
      <li>Элемент для ввода даты.</li>
      <li>Элемент для ввода времени.</li>
      <li>Элемент для ввода рейтинга.</li>
      . . .
    </ul>
  </li>
</ul>
```

Мы также можем вкладывать нумерованный список внутрь маркированного и наоборот. Глубина вложения списков не ограничена.

Еще Metro позволяет создать так называемый *список определений*, представляющий собой перечень терминов и их разъяснений. Такой список создают с помощью парного тега `<dl>`. Внутри него помещают пары "термин — разъяснение", причем термины заключают в парный тег `<dt>`, а разъяснения — в парный тег `<dd>`.

```
<dl>
  <dt>Тег p</dt>
  <dd>Создает абзац</dd>
  <dt>Тег hn</dt>
  <dd>Создает заголовок уровня n</dd>
</dl>
```

Осталось сказать, что списки и их пункты относятся к блочным элементам, и при их выводе на экран платформа Metro руководствуется теми же правилами, что и при выводе абзацев и заголовков.

Содержимое списка выводится с отступом слева. Отступы между пунктами списка делаются меньшими, чем в случае абзацев или заголовков.

Цитаты

Язык HTML позволяет нам выделить особым образом большие цитаты, помещенные в текст. Код, создающий такие цитаты, помещается внутрь парного тега `<blockquote>`.

```
<blockquote>
  <p>Windows 8 — кодовое имя находящейся в разработке операционной
  системы (ОС), принадлежащей семейству ОС Microsoft Windows и
  разрабатываемой транснациональной корпорацией Microsoft.</p>
```

```
  <p>(Цитата взята из русской Википедии.)</p>
</blockquote>
```

Содержимое большой цитаты может быть любым: абзацы, заголовки, списки и пр. Размер цитаты не ограничен.

Большая цитата также относится к блочным элементам. Платформа Metro выводит цитаты с отступом слева.

Адреса

Любое приложение должно содержать сведения о его разработчиках: их имена и контактные данные. Это нужно для того, чтобы пользователь в случае возникновения проблем смог связаться с ними.

Сведения о разработчиках оформляются в виде так называемого *адреса*. Для создания адреса применяется парный тег `<address>`.

```
<address>Все права защищены. Тяп-Ляп Хай-Тек Пресайжн Индастри</address>
```

Адрес ведет себя так же, как абзац, но его содержимое выводится курсивом.

Разрывы строк

Часто возникает необходимость перенести какую-то часть текста на другую строку, не создавая при этом новый абзац. Обычно так поступают из соображений компактности: как мы уже знаем, абзац выводится с заметными отступами сверху и снизу и, стало быть, занимает довольно много места на экране, размер которого ограничен.

В таком случае нам может помочь *разрыв строк*. Он указывает платформе Metro просто перенести следующую за ним часть текста на другую строку, оставляя ее при этом в составе того же элемента интерфейса (абзаца, заголовка, пункта списка или адреса).

Разрыв строк создается с помощью одинарного тега `<br>`. Он ставится в том месте текста, в котором должен быть выполнен перенос на другую строку.

```
<address>Все права защищены.<br />Тяп-Ляп Хай-Тек Пресайжн
Индастри</address>
```

В этом случае фрагмент "Тяп-Ляп Хай-Тек Пресайжн Индастри" будет перенесен на следующую строку, тем не менее, оставаясь в составе адреса.

В отличие от рассмотренных нами ранее элементов интерфейса, разрыв строк является встроенным элементом. Это значит, что он может присутствовать только в блочных элементах, например, абзацах или, как у нас, адресах.

Выделение фрагментов текста

А еще язык HTML поддерживает теги, позволяющие нам выделить фрагмент текста и дать ему особую значимость. С помощью этих тегов мы можем, например, выделить аббревиатуру или указать, что какой-то фрагмент является очень важным.

Некоторые из этих тегов, самые полезные для нас, перечислены в табл. 8.1. Все они парные и представляют собой встроенные элементы.

Таблица 8.1. *Некоторые теги HTML, предназначенные для выделения фрагментов текста*

Тег	Назначение	Выводится на экран
`<abbr>`	Аббревиатура	Подчеркнутым
`<acronym>`	Аббревиатура. Фактически то же самое, что и тег `<abbr>`	Подчеркнутым
`<cite>`	Небольшая цитата	Курсивом
`<dfn>`	Новый термин	Курсивом
`<em>`	Менее важный текст	Курсивом
`<q>`	Небольшая цитата. Фактически то же самое, что и тег `<cite>`	Обычным шрифтом
`<strong>`	Очень важный текст	Полужирным шрифтом

```
<p>Платформа <dfn>Metro</dfn> входит в состав Windows 8.</p>
```

Здесь мы пометили название платформы Metro как новый термин.

```
<blockquote>
  <p><strong>Windows 8</strong> — кодовое имя находящейся в разработке
  операционной системы (ОС), принадлежащей семейству ОС Microsoft
  Windows и разрабатываемой транснациональной корпорацией Microsoft.</p>
  <p>(<em>Цитата взята из русской Википедии.</em>)</p>
</blockquote>
```

А здесь мы пометили название новой операционной системы от Microsoft как очень важную часть текста, а ссылку на русскую Википедию — как менее важную.

Оформление текста

На очереди — оформление уже набранного текста. Разберемся, какие инструменты для этого предоставляет язык CSS.

Задание параметров шрифта

Прежде всего, мы можем задать параметры шрифта, которым будет выведен текст: имя, размер, начертание и пр.

Атрибут стиля `font-family` задает имя шрифта, которым будет выведен текст:

`font-family:` *`<имя шрифта>`*

Имя шрифта задается в виде его названия, например, Segoe UI или Arial. Если имя шрифта содержит пробелы, его нужно взять в кавычки.

```
p { font-family: Arial; }
h1 ( font-family: "Segoe UI Light"; }
```

Атрибут стиля `font-size` определяет размер шрифта:

```
font-size: <размер>
```

Размер задается в виде числа, после которого, без всяких пробелов, ставится обозначение *единицы измерения CSS*. Так, пикселы обозначаются символами `px`, пункты — символами `pt`, а миллиметры — символами `mm`.

```
h1 { font-size: 42pt; }
```

Атрибут стиля `color` задает цвет текста:

```
color: <цвет>
```

Само значение цвета можно задать в трех форматах, перечисленных далее.

- С помощью CSS-функции `rgb()`. Она имеет такой формат вызова:

  ```
  rgb(<доля красного цвета>, <доля зеленого цвета>,
  ↳<доля синего цвета>)
  ```

  Доли всех цветов указываются в виде целых чисел от 0 до 255.

- С помощью CSS-функции `rgba()`. Формат ее вызова таков:

  ```
  rgba(<доля красного цвета>, <доля зеленого цвета>,
  ↳<доля синего цвета>, <степень прозрачности>)
  ```

  Доли всех цветов также указываются в виде целых чисел от 0 до 255, а *степень прозрачности* — в виде числа с плавающей точкой от 0 до 1.

- По имени нужного цвета. Так, черный цвет имеет имя `black`, белый — `white`, красный — `red`, зеленый — `green`, синий — `blue`, а белый — `white`.

> ***Внимание!***
>
> Полный список имен и соответствующих им цветов можно посмотреть на Web-странице **http://msdn.microsoft.com/en-us/library/windows/apps/hh453226.aspx**.

```
body { color: rgb(255, 255, 255); }
```

Задаем для всего текста в приложении (теге `<body>`) белый цвет.

```
h1 { color: blue; }
```

Заголовки первого уровня пусть выводятся синим цветом.

```
.win-itemTitle { color: rgba(127, 127, 127, 0.8); }
```

А содержимое тегов с привязанным стилевым классом `.win-itemTitle` — серым цветом со степенью прозрачности в 0,8 (80%).

Атрибут стиля `font-weight` устанавливает степень "полужирности" шрифта:

```
font-weight: normal|bold|100|200|300|400|500|600|700|800|900
```

Здесь доступны семь абсолютных значений от `100` до `900`, представляющих различную "полужирность" шрифта, от минимальной до максимальной; при этом обыч-

ный шрифт будет иметь "полужирность" `400` (или `normal`), а полужирный — `700` (или `bold`). Значение по умолчанию — `400` (`normal`).

```
acronym { font-weight: bold; }
```

Атрибут `font-style` задает начертание шрифта:

```
font-style: normal|italic|oblique
```

Доступны три значения, представляющие обычный шрифт (`normal`), курсив (`italic`) и особое декоративное начертание, похожее на курсив (`oblique`).

```
.special-text { font-style: italic; }
```

Атрибут стиля `text-decoration` задает "украшение" (подчеркивание или зачеркивание), которое будет применено к тексту:

```
text-decoration: none|underline|overline|line-through
```

Здесь доступны четыре значения:

- `none` убирает все "украшения", заданные для шрифта родительского элемента;
- `underline` подчеркивает текст;
- `overline` "надчеркивает" текст, т. е. проводит линию над строками;
- `line-through` зачеркивает текст.

Атрибут стиля `font-variant` позволяет указать, как будут выглядеть строчные буквы шрифта:

```
font-variant: normal|small-caps
```

Значение `small-caps` задает такое поведение шрифта, когда его строчные буквы выглядят точно так же, как прописные, просто имеют меньший размер. Значение `normal` задает для шрифта обычные прописные буквы.

Атрибут стиля `text-transform` позволяет изменить регистр символов текста:

```
text-transform: none|capitalize|uppercase|lowercase
```

Мы можем преобразовать текст к верхнему (значение `uppercase` этого атрибута стиля) или нижнему (`lowercase`) регистру, преобразовать к верхнему регистру первую букву каждого слова (`capitalize`) или оставить в изначальном виде (`none`).

Атрибут стиля `line-height` задает высоту строки текста:

```
line-height: <высота>
```

После значения высоты следует указать единицу измерения, например пикселы (`px`).

```
h2 { line-height: 20px; }
```

Если же единицу измерения не указывать, то высота строки станет равной произведению указанного нами значения на значение высоты строки по умолчанию.

```
h1 { line-height: 2; }
```

Устанавливаем удвоенную высоту строки текста для заголовка первого уровня.

Атрибут стиля `letter-spacing` позволяет задать дополнительное расстояние между символами текста:

```
letter-spacing: <расстояние>
```

Отметим, что это именно дополнительное расстояние; оно будет добавлено к изначальному, установленному самой платформой Metro.

```
h1 { letter-spacing: 5px; }
```

Теперь текст в заголовке первого уровня будет выглядеть разреженным.

Атрибут стиля `word-spacing` задает дополнительное расстояние между отдельными словами текста:

```
word-spacing: <расстояние>
```

Это также дополнительное расстояние, добавляемое к изначальному.

```
h2 { word-spacing: 5mm; }
```

И напоследок рассмотрим атрибут стиля `font`, позволяющий задать одновременно сразу несколько параметров шрифта:

```
font: [<начертание>] [<вид строчных букв>] [<"жирность">]
[<размер>[/<высота строки текста>]] <имя шрифта>
```

Как видим, обязательным является только *имя шрифта* — остальные параметры могут отсутствовать.

```
body { font: 11pt/15pt "Segoe UI Semilight"; }
```

Задаем для всего текста, что есть в приложении, шрифт Segoe UI Semilight, размер шрифта в 11 пунктов и высоту строки в 15 пунктов.

```
.title { font: 42pt/48pt "Segoe UI Light"; }
```

А для стилевого класса `.title` задаем шрифт Segoe UI Light, размер 42 пункта и высоту строки 48 пунктов. Получится весьма крупная надпись.

Параметры фона

Еще CSS позволяет нам задать цвет фона для всего приложения или отдельного элемента его интерфейса.

Задать его позволит атрибут стиля `background-color`.

```
background-color: transparent|<цвет>
```

Значение цвета можно указать в любом из трех форматов, рассмотренных нами ранее. Значение `transparent` задает для элемента интерфейса "прозрачный" фон.

```
body { background-color: rgb(17, 17, 17); }
```

Задаем для всего приложения темный, почти черный фон.

Контейнеры. Встроенные контейнеры

Очень часто возникает необходимость выделить определенным образом не весь текст, скажем, абзаца, а только некоторый его фрагмент. Причем фрагмент, не помещенный ни в какой другой тег.

Сделать это можно с помощью специального инструмента HTML, называемого контейнером. *Контейнер* — это элемент, служащий для выделения фрагмента содержимого интерфейса с целью либо задать для него оформление, либо управлять им программно, из логики. Никакого форматирования для своего содержимого он не предусматривает и никакого особого значения ему не дает.

В данном случае нам понадобится *встроенный контейнер*, который ведет себя как встроенный элемент. Он формируется парным тегом `<span>`, в который заключается нужный фрагмент текста.

```
<p>Платформа Metro служит для разработки и выполнения приложений,
<span>изначально предназначенных для планшетов</span>.</p>
```

Здесь мы заключили во встроенный контейнер часть текста абзаца.

Теперь нам останется создать стиль и привязать его к только что созданному контейнеру.

```
.underlined { text-decoration: underline; }
<p>Платформа Metro служит для разработки и выполнения приложений,
<span class="underlined">изначально предназначенных для
планшетов</span>.</p>
```

После чего часть абзаца, являющаяся содержимым контейнера, будет выводиться подчеркнутой.

В *главе 9* мы изучим блочные контейнеры, которые можно использовать для той же цели. Хотя область применения таких контейнеров много шире; в частности, они активно применяются при создании разметки.

Параметры абзацев и списков

Также часто возникает необходимость задать параметры вывода текста в абзацах, заголовках, списках и пр. К ним относятся выравнивание текста, отступ "красной строки" и форма маркера или нумерации в пунктах списков.

> ***Внимание!***
> Параметры абзацев и списков могут быть заданы только для блочных элементов.

Атрибут стиля `text-align` задает выравнивание текста:

```
text-align: left|right|center|justify
```

Здесь доступны значения `left` (выравнивание по левому краю; обычное поведение платформы Metro при выводе текста), `right` (по правому краю), `center` (по центру) и `justify` (полное выравнивание).

```
.centered { text-align: center; }
p { text-align: center; }
```

Атрибут стиля `text-indent` задает величину отступа для "красной строки":

```
text-indent: <отступ "красной строки">
```

По умолчанию отступ "красной строки" равен нулю.

```
p { text-indent: 5mm; }
```

Теперь все абзацы будут иметь "красную строку" с отступом в 5 мм.

Атрибут стиля `list-style-type` задает вид маркеров или нумерации у пунктов списка:

```
list-style-type: disc|circle|square|decimal|decimal-leading-zero|
lower-roman|upper-roman|lower-greek|lower-alpha|lower-latin|
upper-alpha|upper-latin|armenian|georgian|none
```

Как видим, доступных значений у этого атрибута стиля очень много. Они обозначают как различные виды маркеров, так и разные способы нумерации:

- `disc` — маркер в виде кружка с заливкой (обычное поведение для маркированных списков);
- `circle` — маркер в виде кружка без заливки;
- `square` — маркер в виде квадратика;
- `decimal` — нумерация арабскими цифрами (обычное поведение для нумерованных списков);
- `decimal-leading-zero` — нумерация арабскими цифрами от 01 до 99 с начальным нулем;
- `lower-roman` — нумерация маленькими римскими цифрами;
- `upper-roman` — нумерация большими римскими цифрами;
- `lower-greek` — нумерация маленькими греческими буквами;
- `lower-alpha` и `lower-latin` — нумерация маленькими латинскими буквами;
- `upper-alpha` и `upper-latin` — нумерация большими латинскими буквами;
- `armenian` — нумерация традиционными армянскими цифрами;
- `georgian` — нумерация традиционными грузинскими цифрами;
- `none` — маркер и нумерация отсутствуют (обычное поведение для не-списков).

Атрибут стиля `list-style-type` можно задавать как для самих списков, так и для отдельных пунктов списков. В последнем случае создается список, в котором пункты имеют разные маркеры или нумерацию. Иногда это может пригодиться.

```
ul { list-style-type: square; }
```

Устанавливаем для всех маркированных списков маркер в виде квадратика.

```
.squared { list-style-type: square; }
. . .
<ul>
  <li>Кнопка.</li>
```

```
  <li class="squared">Поле ввода.</li>
  <li>Флажок.</li>
</ul>
```

А здесь мы задаем маркер в виде квадратика только для второго пункта списка (все остальные пункты будут иметь маркер по умолчанию — кружок с заливкой).

Вставка недопустимых символов. Литералы

Предположим, что мы решили написать Metro-приложение — руководство по программированию под платформу Metro. Мы набрали и отформатировали текст всех разделов и глав этого руководства и разбили его на отдельные фрагменты (о фрагментах мы будем говорить в *главе 13*). И столкнулись с необходимостью вставить в текст примеры HTML-кода.

Проблема в том, что теги HTML включают в свой состав символы < и >. Если мы просто наберем эти символы в тексте, платформа Metro посчитает их частью тегов и попытается как-то обработать. Результат этой обработки будет удручающим; скорее всего, данные символы и текст между ними вообще не будут выведены на экран, да и остальной интерфейс будет выведен с искажениями.

Символы < и >, как и многие другие, имеют в HTML-коде особое значение и являются недопустимыми в обычном тексте (их так и называют — *недопустимые символы*). Поэтому их заменяют особыми последовательностями символов — *литералами*. Встретив литерал, платформа Metro "поймет", что здесь должен присутствовать соответствующий недопустимый символ, и выведет его на экран.

Литералы HTML начинаются с символа `&` и заканчиваются символом `;` (точка с запятой). Между ними помещается определенная последовательность букв. Так, символ < определяется литералом `<`, а символ > — литералом `>`.

```
<dl>
  <dt>Тег &lt;p&gt;</dt>
  <dd>Создает абзац</dd>
  <dt>Тег &lt;hn&gt;</dt>
  <dd>Создает заголовок уровня n</dd>
</dl>
```

Мы взяли список определений, описывающий два тега HTML, и дополнили написание этих тегов символами < и >, применив соответствующие литералы.

Литералов HTML поддерживает довольно много. Самые часто применяемые из них перечислены в табл. 8.2.

Таблица 8.2. *Некоторые литералы языка HTML*

Недопустимый символ	Литерал HTML
— (длинное тире)	`&mdash;`
– (короткое тире)	`&ndash;`
"	`&quot;`

Таблица 8.2 *(окончание)*

Недопустимый символ	Литерал HTML
&	&amp;
<	&lt;
>	&gt;
©	&copy;
®	&reg;
Левая двойная кавычка	&ldquo;
Левая угловая кавычка	&laquo;
Левый апостроф	&lsquo;
Многоточие	&hellip;
Неразрывный пробел	
Правая двойная кавычка	&rdquo;
Правая угловая кавычка	&raquo;
Правый апостроф	&rsquo;
Символ евро	&euro;

Среди перечисленных в табл. 8.2 литералов и обозначаемых ими недопустимых символов особенно выделяется один. Это *неразрывный пробел*, обозначаемый литералом ` `. По этому пробелу платформа Metro никогда не будет выполнять перенос строк.

Неразрывный пробел необходим, если в каком-то месте предложения перенос строк никогда не должен выполняться. Так, правила правописания русского языка не допускают перенос строк перед длинным тире. Поэтому крайне рекомендуется отделять длинное тире от предыдущего слова неразрывным пробелом:

```
<p>Неразрывный пробел &mdash; очень важный литерал.</p>
```

Кстати, если уж на то пошло, мы можем в сведениях об авторских правах вставить символ ©. Вот так:

```
<address>Все права защищены.<br />&copy; Тяп-Ляп Хай-Тек Пресайжн
Индастри</address>
```

HTML также позволяет вставить в текст любой символ, поддерживаемый кодировкой Unicode, просто указанием его кода. Для этого предусмотрен литерал вида `&#`*`<десятичный код символа>`*`;`.

Но как узнать код нужного символа? Очень просто. В этом нам поможет утилита Таблица символов[1] (Character Map), поставляемая в составе Windows. Давайте запустим ее и посмотрим на ее окно (рис. 8.1).

[1] На момент написания книги русской редакции Windows 8 еще не было. Но поскольку книга адресована русскоязычным читателям, в книге программы, к которым привыкли пользователи предыдущих версий этой ОС, называются по-русски.

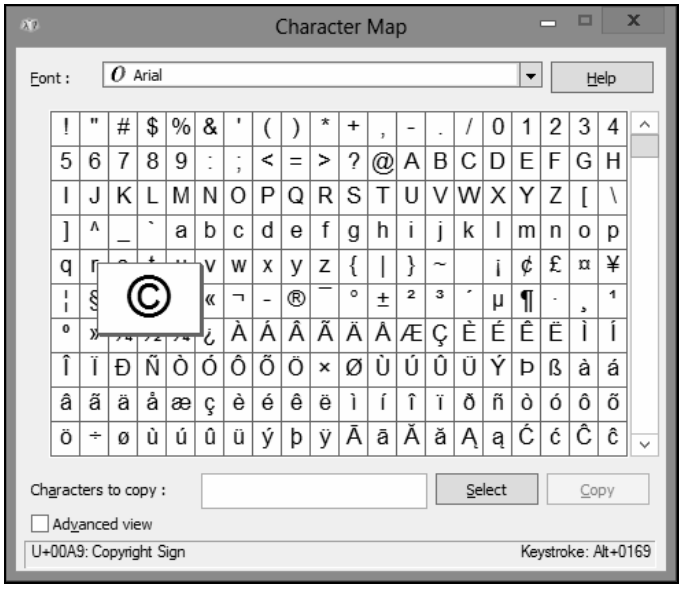


Рис. 8.1. Окно утилиты Таблица символов (выбран символ ©)

В большом списке символов, занимающем почти все окно этой утилиты, выберем нужный нам символ. После этого посмотрим на строку статуса, расположенную вдоль нижнего края окна. В правой ее части находится надпись вида **Клавиша: Alt+<*десятичный код символа*>** (Keystroke: Alt+<*десятичный код символа*>). Этот-то код нам и нужен!

> ***Внимание!***
>
> Надпись **Клавиша: Alt+<*десятичный код символа*>** появляется в строке статуса окна Таблицы символов только при выборе символов, которые нельзя ввести непосредственно с клавиатуры.

Так, мы можем вставить в сведения об авторских правах символ ©, использовав литерал `&#0169;`, где `0169` — десятичный код данного символа (см. рис. 8.1):

```
<address>Все права защищены.<br />&#0169; Тяп-Ляп Хай-Тек Пресайжн
Индастри</address>
```

Создание таблиц

Таблицы — лучший способ вывести структурированные данные, заняв при этом минимум места на экране. Так что давайте познакомимся с ними.

Формирование таблиц

Таблицы HTML создаются в четыре этапа.

На первом этапе в HTML-коде с помощью парного тега `<table>` формируют саму таблицу.

```
<table>
</table>
```

На втором этапе формируют строки таблицы. Для этого служит парный тег <tr>; каждый такой тег создает отдельную строку. Теги <tr> помещают внутрь тега <table>.

```
<table>
  <tr>
  </tr>
  <tr>
  </tr>
  . . .
</table>
```

На третьем этапе создают ячейки таблицы, для чего используют парные теги <td> и <th>. Тег <td> создает обычную ячейку, а тег <th> — *ячейку заголовка*, в которой будет помещаться "шапка" соответствующего столбца таблицы. Теги <td> и <th> помещают в теги <tr>, создающие строки таблицы, в которых должны находиться эти ячейки.

```
<table>
  <tr>
    <th></th>
    <th></th>
    <th></th>
  </tr>
  <tr>
    <td></td>
    <td></td>
    <td></td>
  </tr>
  . . .
</table>
```

На четвертом, последнем, этапе указывают содержимое ячеек, которое помещают в соответствующие теги <td> и <th>.

```
<table>
  <tr>
    <th>Тег</th>
    <th>Назначение</th>
    <th>Разновидность</th>
  </tr>
  <tr>
    <td>p</td>
    <td>Абзац</td>
    <td>Парный</td>
  </tr>
```

```
  <tr>
    <td>hn</td>
    <td>Заголовок уровня n</td>
    <td>Парный</td>
  </tr>
  <tr>
    <td>br</td>
    <td>Разрыв строк</td>
    <td>Одинарный</td>
  </tr>
  . . .
</table>
```

Если нам нужно поместить в ячейку таблицы простой текст, мы просто вставим его в соответствующий тег `<td>` или `<th>` (как показано в приведенном ранее примере). Никакие дополнительные элементы интерфейса, например абзацы или блоки, в этом случае не нужны.

Если же нам потребуется как-то оформить содержимое ячеек, мы применим изученные в этой главе теги. Например, мы можем придать именам тегов особую важность, воспользовавшись тегом `<strong>`.

```
. . .
<tr>
  <td><strong>p</strong></td>
  <td>Абзац</td>
  <td>Парный</td>
</tr>
. . .
```

Вообще, мы можем поместить в ячейку таблицы какое угодно содержимое: абзацы, заголовки, списки, блоки, графические изображения и комбинацию всего этого. Мы можем даже вставить туда другую таблицу.

```
. . .
<tr>
  <td><strong>p</strong></td>
  <td>
    <p>Абзац.</p>
    <h6>Пример:</h6>
    <p class="sample">&lt;p&gt;Абзац&lt;/p&gt;</p>
  </td>
  <td>Парный</td>
</tr>
. . .
```

Здесь мы поместили во вторую ячейку строки два абзаца и заголовок.

Таблица представляет собой блочный элемент интерфейса. Текст, содержащийся в обычных ячейках таблицы, выводится так же, как текст абзацев. А текст в ячей-

ках заголовка выводится полужирным шрифтом и выровненным по центру. Хотя, разумеется, мы можем изменить его оформление, задав соответствующие стили CSS.

Объединение ячеек таблиц

Теперь поговорим об одной интересной особенности таблиц HTML, которая может нам пригодиться. Это так называемое *объединение ячеек* таблиц.

Знакомиться с объединением ячеек лучше всего на примере — простой таблице, HTML-код которой приведен далее.

```
<table>
  <tr>
    <td>1</td>
    <td>2</td>
    <td>3</td>
  </tr>
  <tr>
    <td>4</td>
    <td>5</td>
    <td>6</td>
  </tr>
  <tr>
    <td>7</td>
    <td>8</td>
    <td>9</td>
  </tr>
</table>
```

Платформа Metro выведет данную таблицу так, как представлено на рис. 8.2.

1	2	3
4	5	6
7	8	9

Рис. 8.2. Изначальная таблица, чьи ячейки подвергнутся объединению

1	2+3	
4+7	5	6
	8	9

Рис. 8.3. Таблица, показанная на рис. 8.2, после объединения некоторых ячеек (объединенные ячейки обозначены сложением их номеров)

А теперь рассмотрим таблицу на рис. 8.3.

Здесь выполнено объединение некоторых ячеек. Видно, что объединенные ячейки словно слились в одну. Как это сделать?

Специально для этого теги <td> и <th> поддерживают два весьма примечательных необязательных атрибута. Первый — colspan — объединяет ячейки по горизонтали, второй — rowspan — по вертикали.

Чтобы объединить несколько ячеек по горизонтали в одну, нужно выполнить следующие шаги:

1. Найти в коде HTML тег <td> (<th>), соответствующий первой из объединяемых ячеек (если считать ячейки слева направо).
2. Вписать в него атрибут colspan и указать для него в качестве значения количество объединяемых ячеек, считая и самую первую из них.
3. Удалить теги <td> (<th>), создающие остальные объединяемые ячейки данной строки.

Давайте объединим ячейки 2 и 3 нашей таблицы. Исправленный фрагмент кода, создающий первую строку этой таблицы, выглядит так:

```
. . .
<tr>
  <td>1</td>
  <td colspan="2">2+3</td>
</tr>
. . .
```

Объединить ячейки по вертикали чуть сложнее. Вот шаги, которые нужно для этого выполнить.

1. Найти в коде HTML строку (тег <tr>), в которой находится первая из объединяемых ячеек (если считать строки сверху вниз).
2. Найти в коде этой строки тег <td> (<th>), соответствующий первой из объединяемых ячеек.
3. Вписать в него атрибут rowspan и задать для него в качестве значения количество объединяемых ячеек, считая и самую первую из них.
4. Просмотреть последующие строки и удалить из них теги <td> (<th>), создающие остальные объединяемые ячейки.

Нам нужно объединить ячейки 4 и 7. Далее приведен исправленный фрагмент HTML-кода, создающего вторую и третью строки.

```
. . .
<tr>
  <td rowspan="2">4+7</td>
  <td>5</td>
  <td>6</td>
</tr>
<tr>
  <td>8</td>
  <td>9</td>
</tr>
. . .
```

Обратим внимание, что мы удалили из третьей строки тег <td>, создающий ячейку 7, поскольку она объединилась с ячейкой 4.

На этом о таблицах все. О выводе и форматировании текста — тоже.

Что дальше?

В этой главе мы занимались исключительно текстом. Мы структурировали его, оформляли средствами HTML и CSS, вводили недопустимые символы и создавали таблицы. Так что, если у нас возникнет нужда написать приложение для работы с текстом, мы знаем, как это делается.

Следующая глава будет посвящена разметке — совокупности правил, описывающих местоположение различных элементов интерфейса на экране. Тема эта весьма велика и довольно сложна, но настоящего программиста сложностями не испугаешь!

ГЛАВА 9

Разметка

В предыдущей главе мы занимались выводом и форматированием текста. И узнали, что языки HTML и CSS предоставляют для этого весьма богатый инструментарий. (Что и неудивительно — ведь они изначально создавались для работы с текстом.)

Мы знаем, что все элементы интерфейса, создаваемые с помощью тегов HTML, делятся на блочные и встроенные. Блочные элементы выстраиваются по вертикали сверху вниз, как абзацы в тексте, а встроенные — по горизонтали слева направо, как символы в абзаце. Если мы посмотрим на интерфейс приложений `SimpleCalc` и `DateCalc`, созданных нами ранее, то сразу убедимся в этом.

Но есть ли способы разместить их как-то иначе? Можем ли мы указать местоположение определенного элемента интерфейса сами?

Безусловно. Для этого мы создадим *разметку* (layout) — набор правил, описывающих местоположение элементов интерфейса нашего приложения. Как и с помощью каких инструментов она создается, будет рассказано в этой главе.

Блочные контейнеры, или блоки

В *главе 8* мы познакомились с контейнерами HTML, а именно со встроенными контейнерами. Их назначение — привязка стилей к фрагментам блочных элементов — оказалось чисто утилитарным.

Есть и другая разновидность контейнеров — *блочные*, или *блоки*. Они нам давно знакомы; еще в *главе 2*, создавая наше первое Metro-приложение, мы использовали такой контейнер. И мы знаем, что блочные контейнеры создаются с помощью парного тега `<div>` и ведут себя как блочные элементы (что, собственно, следует из их названия).

Но область применения у таких контейнеров заметно шире. Далее перечислены типичные случаи, когда без блоков не обойтись.

- Объединение нескольких блочных элементов с целью привязать к ним стиль.

```
<div class="some-style">
  <p>Платформа Metro входит в состав Windows 8.</p>
```

```
    <p>Платформа Metro служит для разработки и выполнения приложений,
    изначально предназначенных для планшетов.</p>
  </div>
```

- Превращение встроенного элемента в блочный.

```
<div>
  <input type="button" id="btnCalculate" value="=" />
</div>
```

- Создание разметки.

Вот о последнем случае, в основном, и пойдет рассказ. В самом деле, все поддерживаемые платформой Metro виды разметки, что мы рассмотрим, применимы только к блочным контейнерам.

Сеточная разметка

Сеточная разметка (grid layout) — пожалуй, самый универсальный вид разметки, применимый к абсолютному большинству приложений. При ее использовании в пространстве элемента, в котором организуется разметка *(контейнера разметки)*, формируется подобие таблицы *(сетка разметки)*, в ячейки которых и помещаются отдельные элементы интерфейса.

В качестве контейнера разметки может использоваться сам интерфейс приложения (тег `<body>`), блочный контейнер или ячейка таблицы. Элементы, которые будут позиционированы с применением этой разметки, должны быть блочными.

Сеточная разметка лучше всего подходит в тех случаях, когда набор элементов интерфейса строго определен, количество их ограничено, а местоположение в процессе функционирования приложения меняться не должно. Если же это не так, лучше применить другой вид разметки или вообще отказаться от нее.

Создание сетки разметки

Первое, что нам нужно сделать для реализации сеточной разметки, — собственно указать, что она должна быть реализована. Это следует сделать в элементе, который станет контейнером разметки.

Создание сеточной разметки выполняется с помощью атрибута стиля `display`. Все, что нужно сделать, — задать для него значение `-ms-grid`.

```
#divGrid { display: -ms-grid; }
. . .
<div id="divGrid">
</div>
```

Здесь мы создаем блок `divGrid` и указываем для него сеточную разметку. Теперь данный блок станет контейнером разметки.

Следующий шаг — формирование сетки разметки путем указания размеров составляющих ее строк и столбцов. Для этого предназначены два атрибута стиля, описанные далее.

Атрибут стиля `-ms-grid-columns` служит для указания значений ширины отдельных столбцов в сетке разметки. Сколько значений указано в нем, столько столбцов будет создано. Отдельные значения разделяются пробелами.

В качестве значений ширины можно использовать следующие величины.

- Числовые значения, заданные в любых поддерживаемых CSS единицах измерения размеров (пикселы, миллиметры и др.). Они задают фиксированные значения ширины столбцов.

  ```
  -ms-grid-columns: 100px 200mm 3cm;
  ```

  Сетка разметки будет включать три столбца; первый столбец получит ширину 100 пикселов, второй — 200 миллиметров, третий — 3 сантиметра (сантиметры обозначаются символами `cm`).

- Значение `auto`. Оно указывает, что данный столбец будет иметь такую ширину, чтобы полностью вместить свое содержимое, и не более.

  ```
  -ms-grid-columns: 100px auto 3cm;
  ```

  Здесь второй столбец сетки разметки будет иметь такую ширину, чтобы полностью вместить свое содержимое. Остальные столбцы получат фиксированные значения ширины.

- Значения вида *<доля>*`fr`. Оно задает ширину столбца, равную *доле* от оставшегося пространства контейнера разметки; сама эта доля указывается в виде целого числа.

  ```
  -ms-grid-columns: 100px auto 3fr 1fr;
  ```

  Первый столбец получит фиксированную ширину в 100 пикселов, а второй — такую, чтобы полностью вместить свое содержимое. Оставшееся пространство в контейнере разметки будет разделено на 4 (3 + 1) части; третий столбец получит ширину, равную трем таким частям (3/4 от оставшегося пространства), а четвертый — ширину, равную одной части (1/4 оставшегося пространства).

Аналогично, для задания высоты строк в сетке разметки используется атрибут стиля `-ms-grid-rows`. Он ведет себя так же и поддерживает те же значения, что и атрибут стиля `-ms-grid-columns`.

```
#divGrid {
  -ms-grid-columns: 100px 3fr 1fr;
  -ms-grid-rows: 50px auto 50px;
}
```

Позиционирование элементов интерфейса в ячейках сетки разметки

Следующий наш шаг — собственно расстановка элементов интерфейса в ячейках созданной нами ранее сетки разметки. Выполняется он в два этапа.

На первом этапе мы должны поместить все элементы интерфейса, что будем позиционировать с применением сеточной разметки, внутрь контейнера разметки (что

вполне понятно). Сразу же проверим, являются ли все эти элементы блочными (поскольку разметка действует только на блочные элементы); если какой-то элемент является встроенным, превратить его в блочный можно, заключив в блок.

```
<div id="divGrid">
  <div id="div1"> . . . </div>
  <div id="div2"> . . . </div>
  <div id="div3"> . . . </div>
  <div id="div4"> . . . </div>
</div>
```

Здесь мы создали в контейнере разметки — блоке `divGrid` четыре блока: `div1`, `div2`, `div3` и `div4`. Их мы и будем позиционировать с применением сеточной разметки.

Логично предположить, что позиционирование элемента в контейнере разметки выполняется указанием ячейки сетки разметки, в которой он должен находиться. Так и делают — указывают для этого элемента номера строки и столбца, что формируют эту ячейку.

А выполняется это с помощью атрибутов стиля `-ms-grid-column` и `-ms-grid-row`. Первый указывает номер столбца, а второй — номер строки, на пересечении которых находится нужная ячейка. Нумерация строк и столбцов сетки разметки начинается с единицы, а сами номера задаются в виде целых чисел.

```
#div1 {
  -ms-grid-row: 1;
  -ms-grid-column: 1;
}
```

Помещаем блок `div1` в ячейку сетки разметки, образованную пересечением первой строки и первого столбца.

```
#div4 {
  -ms-grid-row: 2;
  -ms-grid-column: 3;
}
```

А блок `div4` устроится в ячейке, что находится на пересечении второй строки и третьего столбца.

Значение атрибутов стиля `ms-grid-column` и `-ms-grid-row` по умолчанию — 1. То есть, чтобы поместить какой-либо элемент в ячейку первой строки или первого столбца, мы можем не указывать соответствующий атрибут стиля. Так, в нашем случае для блока `div1` мы вообще можем не создавать стиль.

Часто бывает, что какой-то элемент должен занимать не одну ячейку сетки разметки, а сразу несколько соседних ячеек. Для таких случаев язык CSS припас нам атрибуты стиля `-ms-grid-column-span` и `-ms-grid-row-span`. Они задают количество ячеек, расположенных, соответственно, правее и ниже указанной ранее, которые должен занять данный элемент. Эти значения задаются в виде целых чисел.

```
#div4 { -ms-grid-row-span: 2; }
```

Предписываем блоку `div4` занять еще одну ячейку, расположенную ниже той, что мы указали ранее.

```
#div1 {
  -ms-grid-row-span: 2;
  -ms-grid-column-span: 3;
}
```

А блок `div1` растянется на две ячейки вниз и на три — вправо. В результате он займет шесть ячеек в сетке разметки.

Значение атрибутов стиля `-ms-grid-column-span` и `-ms-grid-row-span` по умолчанию — 1. То есть, если мы не укажем какой-либо из этих атрибутов стиля, элемент займет только одну ячейку (что мы и наблюдали ранее).

Выравнивание элементов в ячейках сетки разметки

По умолчанию элементы занимают ячейки сетки разметки полностью. Однако мы можем изменить это поведение, указав для этих элементов выравнивание.

Атрибут стиля `-ms-grid-column-align` задает выравнивание элемента в ячейке сетки разметки по горизонтали. Формат его написания таков:

```
-ms-grid-column-align: start | center | end | stretch
```

Как видим, этот атрибут стиля поддерживает четыре возможных значения:

- `start` — элемент выравнивается по левому краю ячейки для систем письма "слева направо" и по правому краю для систем письма "справа налево";
- `center` — элемент выравнивается по центру ячейки;
- `end` — элемент выравнивается по правому краю ячейки для систем письма "слева направо" и по левому краю для систем письма "справа налево";
- `stretch` — элемент растягивается по всей ширине ячейки (значение по умолчанию).

Атрибут стиля `-ms-grid-row-align` задает для элемента выравнивание по вертикали. Формат его написания таков же, как у атрибута стиля `-ms-grid-column-align`, и поддерживает он те же самые четыре значения:

- `start` — элемент выравнивается по верхнему краю ячейки;
- `center` — элемент выравнивается по центру ячейки;
- `end` — элемент выравнивается по нижнему краю ячейки;
- `stretch` — элемент растягивается по всей высоте ячейки (значение по умолчанию).

```
#div1 {
  -ms-grid-column-align: center;
  -ms-grid-row-align: start;
}
```

Задаем для блока `div1` горизонтальное выравнивание по центру и вертикальное — по верху ячейки.

```
#div4 {
  -ms-grid-column-align: end;
  -ms-grid-row-align: end;
}
```

А для блока `div4` устанавливаем горизонтальное выравнивание по правой границе ячейки и вертикальное — по ее нижней границе.

Пример: окончательная версия арифметического калькулятора

Настала пора попрактиковаться с сеточной разметкой. Возьмем написанное в *главе 6* приложение простейшего калькулятора `SimpleCalc` и доведем до ума его интерфейс. Пусть наш виртуальный калькулятор напоминает настоящий, "железный".

Откроем в Visual Studio проект `SimpleCalc`. Переключимся на файл default.html и исправим код, описывающий интерфейс, чтобы он выглядел так:

```
<div id="divOutput"><input type="text" id="txtOutput" value="0"
readonly /></div>
<div id="divC"><input type="button" id="btnC" value="C" /></div>
<div id="divCE"><input type="button" id="btnCE" value="CE" /></div>
<div id="div1"><input type="button" id="btn1" value="1" /></div>
<div id="div2"><input type="button" id="btn2" value="2" /></div>
<div id="div3"><input type="button" id="btn3" value="3" /></div>
<div id="div4"><input type="button" id="btn4" value="4" /></div>
<div id="div5"><input type="button" id="btn5" value="5" /></div>
<div id="div6"><input type="button" id="btn6" value="6" /></div>
<div id="div7"><input type="button" id="btn7" value="7" /></div>
<div id="div8"><input type="button" id="btn8" value="8" /></div>
<div id="div9"><input type="button" id="btn9" value="9" /></div>
<div id="div0"><input type="button" id="btn0" value="0" /></div>
<div id="divDot"><input type="button" id="btnDot" value="." /></div>
<div id="divPlus"><input type="button" id="btnPlus" value="+" /></div>
<div id="divMinus"><input type="button" id="btnMinus" value="-" /></div>
<div id="divMult"><input type="button" id="btnMult" value="*" /></div>
<div id="divDiv"><input type="button" id="btnDiv" value="/" /></div>
<div id="divEqual"><input type="button" id="btnEqual" value="=" /></div>
```

Мы поместили поле ввода и каждую кнопку в блок, чтобы превратить их в блочные элементы.

Далее откроем файл default.css и введем в него код, определяющий стили, которые создадут сеточную разметку и укажут местоположение для каждого из созданных ранее блочных элементов. Этих стилей будет довольно много, так что давайте рассмотрим их по очереди.

```
body {
  display: -ms-grid;
  -ms-grid-columns: 1fr 80px 80px 80px 80px 80px 1fr;
  -ms-grid-rows: 1fr 80px 80px 80px 80px 80px 1fr;
}
```

Создаем сеточную разметку сразу для всего интерфейса приложения (тега `<body>`). Все равно весь интерфейс приложения будет основан на сеточной разметке.

Сетка разметки будет содержать семь столбцов и семь строк. Начнем со столбцов. Центральные пять получат размеры 80×80 пикселов. Каждый из двух боковых столбцов получит ширину, равную половине от остающегося свободного пространства; в результате интерфейс калькулятора будет располагаться в центре экрана. Точно так же дело обстоит и со строками.

```
div {
  -ms-grid-column-align: center;
  -ms-grid-row-align: center;
}
```

Для содержимого всех блоков, что составят интерфейс нашего приложения, устанавливаем горизонтальное и вертикальное выравнивание по центру.

```
input[type=button] {
  min-width: 70px;
  min-height: 70px;
  font-size: 24pt;
}
```

Задаем для всех кнопок размеры 70×70 пикселов и размер шрифта 24 пункта.

```
#txtOutput {
  width: 300px;
  font-size: 24pt;
}
```

Для поля ввода `txtOutput` указываем ширину в 300 пикселов и размер шрифта также 24 пункта.

```
#divOutput {
  -ms-grid-row: 2;
  -ms-grid-column: 2;
  -ms-grid-column-span: 5;
}
```

Помещаем блок с полем ввода в ячейку, образованную пересечением второго столбца и второй строки сетки разметки, и растягиваем ее еще на четыре ячейки вправо. Так что поле ввода, в котором отображается результат вычислений, будет растянуто на всю ширину набора кнопок.

```
#divC {
  -ms-grid-row: 3;
  -ms-grid-column: 6;
}
```

```
#divCE {
  -ms-grid-row: 4;
  -ms-grid-column: 6;
}
#div1 {
  -ms-grid-row: 5;
  -ms-grid-column: 2;
}
#div2 {
  -ms-grid-row: 5;
  -ms-grid-column: 3;
}
#div3 {
  -ms-grid-row: 5;
  -ms-grid-column: 4;
}
#div4 {
  -ms-grid-row: 4;
  -ms-grid-column: 2;
}
#div5 {
  -ms-grid-row: 4;
  -ms-grid-column: 3;
}
#div6 {
  -ms-grid-row: 4;
  -ms-grid-column: 4;
}
#div7 {
  -ms-grid-row: 3;
  -ms-grid-column: 2;
}
#div8 {
  -ms-grid-row: 3;
  -ms-grid-column: 3;
}
#div9 {
  -ms-grid-row: 3;
  -ms-grid-column: 4;
}
#div0 {
  -ms-grid-row: 6;
  -ms-grid-column: 2;
}
#divDot {
  -ms-grid-row: 6;
  -ms-grid-column: 3;
}
```

```
#divPlus {
  -ms-grid-row: 3;
  -ms-grid-column: 5;
}
#divMinus {
  -ms-grid-row: 4;
  -ms-grid-column: 5;
}
#divMult {
  -ms-grid-row: 5;
  -ms-grid-column: 5;
}
#divDiv {
  -ms-grid-row: 6;
  -ms-grid-column: 5;
}
#divEqual {
  -ms-grid-row: 6;
  -ms-grid-column: 6;
}
```

Позиционируем блоки с кнопками в соответствующих им ячейках сетки разметки.

Сохраним все исправленные файлы и запустим приложение на выполнение. Вот теперь наш калькулятор выглядит вполне профессионально (рис. 9.1).

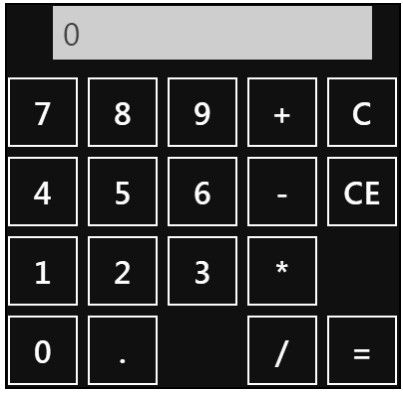


Рис. 9.1. Интерфейс окончательной версии приложения `SimpleCalc`

Гибкая разметка

Если с применением сеточной разметки можно сформировать интерфейс Metro-приложения целиком, то *гибкая разметка* (flexbox layout) для этого вряд ли подойдет. С ее помощью можно форматировать только списки из каких-либо позиций (по крайней мере, для этого данный тип разметки подходит лучше всего). Но, поскольку

ку интерфейс многих приложений включает те или иные списки, гибкая разметка весьма востребована.

Как уже было сказано, гибкая разметка выстраивает элементы интерфейса в подобие списка. Обычно элементы выстраиваются по горизонтали, слева направо; если свободного места больше нет, они переносятся на следующую строку. Хотя, конечно, поведение разметки можно изменить.

Как и в случае сеточной разметки, в качестве контейнера для гибкой разметки может использоваться сам интерфейс приложения (тег `<body>`), блочный контейнер или ячейка таблицы. А сами позиционируемые с ее помощью элементы должны быть блочными.

Создание гибкой разметки

Создать гибкую разметку много проще, чем сеточную. Для этого достаточно указать, что для форматирования содержимого какого-то элемента будет использована гибкая разметка.

Делается это с помощью уже знакомого нам атрибута стиля `display`, которому присваивается значение `-ms-box`.

```
<div id="divItems">
  <div> . . . </div>
  <div> . . . </div>
  <div> . . . </div>
  <div> . . . </div>
  . . .
</div>
. . .
#divItems { display: -ms-box; }
```

Создаем блок `divItems`, помещаем в него элементы, которые будут форматироваться с применением гибкой разметки, и задаем для него гибкую разметку.

По умолчанию любой блок занимает всю ширину родительского элемента. Поэтому, чтобы успешно создать гибкую разметку, мы укажем для блоков какое-либо значение ширины, отличное от `100%`.

```
#divItems > div { width: 30%; }
```

Пусть ширина блоков, что находятся внутри контейнера разметки, будет составлять 30% от ширины родителя.

К сожалению, автоматический перенос элементов на другую строку при нехватке свободного пространства по горизонтали изначально не производится. Нам придется специально указать платформе Metro, чтобы она это делала.

Управление переносом элементов по строкам выполняется с помощью атрибута стиля `-ms-box-lines`. Он указывается, опять же, в контейнере разметки в таком формате:

```
-ms-box-lines: single|multiple
```

Доступных значений всего два:

- `single` — перенос элементов по строкам не выполняется; размещаемые элементы всегда будут располагаться в одну строку (значение по умолчанию);
- `multiple` — перенос элементов по строкам выполняется.

```
#divItems { -ms-box-lines: multiple; }
```

Вот теперь содержимое блока `divItems` будет при необходимости выстраиваться в несколько строк.

Дополнительные параметры гибкой разметки

По умолчанию элементы в контейнере разметки выстраиваются по горизонтали, а строки этих элементов — по вертикали. Однако мы можем изменить это поведение.

Для указания направления, в котором будут выстраиваться размечаемые элементы, служит атрибут стиля `-ms-box-orient`. Он указывается в контейнере разметки в следующем формате:

```
-ms-box-orient: horizontal|vertical|inline-axis|block-axis
```

Здесь доступны четыре значения:

- `horizontal` — размечаемые элементы располагаются в контейнере разметки по горизонтали слева направо (это значение по умолчанию);
- `vertical` — размечаемые элементы располагаются в контейнере разметки по вертикали сверху вниз;
- `inline-axis` — размечаемые элементы располагаются в контейнере разметки в том направлении, в котором выводятся символы в тексте. Так, для языков с направлением письма "слева направо" они выводятся в направлении слева направо, а для языков с направлением письма "сверху вниз" — в направлении сверху вниз;
- `block-axis` — размечаемые элементы выводятся в контейнере разметки в том направлении, в котором выстраиваются строки текста. Так, для языков, где строки выстраиваются сверху вниз, элементы будут выстроены именно в этом направлении.

```
#divItems { -ms-box-orient: vertical; }
```

Задаем вертикальное расположение элементов для блока `divItems`.

Идем далее. По умолчанию элементы в контейнере разметки растягиваются в направлении, перпендикулярном тому, в котором они выстроены, чтобы занять все его свободное пространство. Например, элементы в нашем блоке `divItems` будут растянуты по вертикали.

Изменить это поведение мы можем с помощью атрибута стиля `-ms-box-align`. Он позволяет указать выравнивание размечаемых с помощью гибкой разметки элементов относительно оси, перпендикулярной той, по которой эти элементы выстраи-

ваются и которая указывается атрибутом стиля `-ms-box-orient`. Атрибут стиля `-ms-box-align` указывается в контейнере разметки в таком формате:

```
-ms-box-align: before|after|middle|stretch|baseline
```

Пять значений, поддерживаемых этим атрибутом стиля, перечислены далее:

- `before` — элементы выравниваются по той стороне контейнера разметки, которая повернута на 90° против часовой стрелки относительно направления их выстраивания. Так, если элементы выстраиваются по горизонтали слева направо, то они будут выровнены по верхней стороне контейнера разметки, если они выстраиваются по вертикали сверху вниз — то по его левой стороне;
- `after` — элементы выравниваются по той стороне контейнера разметки, которая повернута на 90° по часовой стрелке относительно направления их выстраивания. Так, если элементы выстраиваются по горизонтали слева направо, то они будут выровнены по нижней стороне контейнера разметки, если они выстраиваются по вертикали сверху вниз — то по его правой стороне;
- `middle` — элементы центрируются по оси, перпендикулярной направлению их выстраивания. Так, если элементы выстраиваются по горизонтали, то они будут центрироваться по вертикали, если они выстраиваются по вертикали, то будут центрироваться по горизонтали;
- `stretch` — элементы растягиваются по оси, перпендикулярной направлению их выстраивания, чтобы занять все пространство контейнера разметки (это значение по умолчанию). Так, если элементы выстраиваются по горизонтали, то они будут растягиваться по вертикали, если они выстраиваются по вертикали, то будут растягиваться по горизонтали;

> ***Внимание!***
>
> Если для размечаемых элементов заданы максимальная ширина или максимальная высота (атрибуты стиля `max-width` или `max-height` соответственно; подробнее о них будет рассказано потом), то эти элементы не будут растянуты, а окажутся выровненными по той стороне контейнера разметки, которая повернута на 90° против часовой стрелки относительно направления их выстраивания (как если бы мы задали значение `before`).

- `baseline` — если элементы выстраиваются по горизонтали, то они будут выравниваться по базовой линии контейнера разметки; если элементы выстраиваются по вертикали, они будут центрироваться по горизонтали. (*Базовой* называется воображаемая линия, на которой располагается текст.)

```
#divItems { -ms-box-align: before; }
```

Задаем для блока `divItems` выравнивание по верхнему краю контейнера.

Иногда может понадобиться выровнять размещаемые с помощью гибкой разметки элементы по направлению, в котором они выстраиваются. Так, если элементы выстраиваются по горизонтали, может возникнуть потребность выровнять их по правому краю контейнера разметки или центрировать в нем.

Нам придет на помощь атрибут стиля `-ms-box-pack`. Он указывается в контейнере разметки и имеет следующий формат:

```
-ms-box-pack: start|end|center|justify
```

Этот атрибут стиля поддерживает четыре возможных значения:

- `start` — элементы выравниваются относительно стороны контейнера разметки, от которой они начинают выстраиваться (значение по умолчанию). Так, если элементы выстраиваются по горизонтали слева направо, они будут выровнены по левой стороне контейнера разметки, а если они выстраиваются по вертикали сверху вниз, то будут выровнены по его верхней стороне;
- `end` — элементы выравниваются относительно стороны контейнера разметки, по направлению к которой они выстраиваются. Так, если элементы выстраиваются по горизонтали слева направо, они будут выровнены по правой стороне контейнера разметки, а если они выстраиваются по вертикали сверху вниз, то будут выровнены по его нижней стороне;
- `center` — элементы центрируются по направлению их выстраивания;
- `justify` — элементы равномерно распределяются в направлении их выстраивания, чтобы заполнить все пространство контейнера разметки. При этом величина просвета между ними устанавливается таким, чтобы они заполнили это пространство целиком.

```
#divItems { -ms-box-pack: end; }
```

Задаем выравнивание по правому краю для блока `divItems`.

Пример: прототип приложения для чтения каналов RSS

В качестве практического задания давайте создадим приложение для чтения каналов RSS. Точнее, его прототип — полноценное приложение мы на данный момент не осилим, т. к. еще не умеем загружать эти каналы из Интернета.

Создадим в Visual Studio новый проект `RSSReader`. Переключимся на файл default.html и впишем в тег `<body>` такой код:

```
<div id="divHeader">
  <h1>Канал новостей</h1>
</div>
<div id="divList">
  <div>
    <p>18.01.2012</p>
    <h3>Опубликованы минимальные требования к устройствам под
    управлением Windows 8</h3>
  </div>
  <div>
    <p>17.01.2012</p>
    <h3>Microsoft рассказала о ReFS, новой файловой системе Windows
    Server 8</h3>
```

```
  </div>
  . . .
</div>
<div id="divContent">
</div>
```

Здесь мы создали три блока. Блок `divHeader` станет заголовком для канала новостей, блок `divList` будет содержать список новостей, а блок `divContent` — текст выбранной пользователем новости. Эти блоки мы потом разместим с применением сеточной разметки.

Внутри блока `divList` мы поместили несколько новостей (чем больше — тем лучше; поскольку мы создаем прототип приложения, мы можем просто придумать эти новости). Каждая новость будет представлять собой блок, включающий в себя дату ее опубликования и заголовок.

На очереди — файл default.css. Откроем его и создадим несколько стилей, описанных далее.

```
body {
  display: -ms-grid;
  -ms-grid-columns: 1fr 1fr;
  -ms-grid-rows: auto 1fr;
}
```

Форматируем весь интерфейс приложения с применением сеточной разметки. Она будет включать в себя две строки и два столбца. Первая строка займет столько места, чтобы вместить свое содержимое, вторая — все остальное место. Столбцы же поделят все пространство экрана поровну.

```
#divHeader { -ms-grid-column-span: 2; }
#divList {
  -ms-grid-row: 2;
  display: -ms-box;
  -ms-box-lines: multiple;
  -ms-box-align: before;
}
#divContent {
  -ms-grid-column: 2;
  -ms-grid-row: 2;
}
```

Помещаем блок `divHeader` в первую строку сетки разметки и растягиваем ее на обе ячейки этой строки. Блок `divList` мы поместим в первую ячейку второй строки, а блок `divContent` — во вторую ее ячейку.

Помимо этого, для блока `divList` мы задаем гибкую разметку с автоматическим переносом содержимого по строкам и выравниванием по верхнему краю контейнера разметки.

```
#divList > div { width: 30%; }
```

А для всех блоков, что непосредственно вложены в блок `divList`, указываем ширину, равную 30% от ширины родителя. Теперь список новостей будет состоять из трех колонок.

Логику для данного приложения мы пока писать не будем. Так что сохраним все исправленные файлы, запустим приложение и посмотрим, как оно выглядит (рис. 9.2).

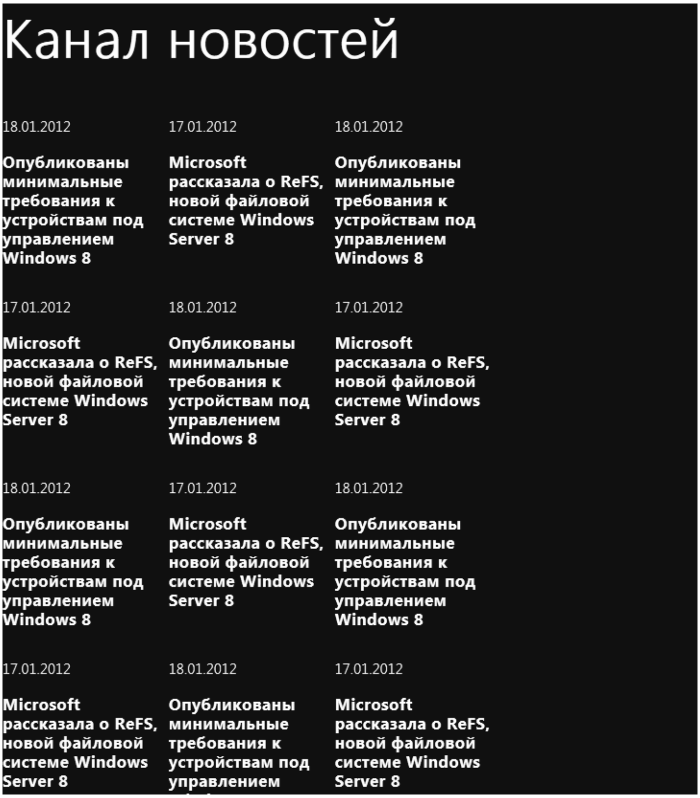


Рис. 9.2. Интерфейс прототипа приложения `RSSReader`

Дополнительные параметры элементов интерфейса

Настала пора немного отвлечься от разметки и рассмотреть средства CSS, которые позволят нам указать дополнительные параметры элементов интерфейса. Это параметры размеров, отступов, рамки, отображения и выравнивания.

Параметры размеров

С атрибутами стиля CSS, задающими размеры элементов интерфейса, мы давно знакомы и не раз ими пользовались. Это атрибуты стиля `width` и `height`. Первый позволяет указать ширину элемента интерфейса, а второй — его высоту.

```
width: auto|<ширина>
height: auto|<высота>
```

Значения ширины и высоты могут быть заданы в любой единице измерения, поддерживаемой языком CSS. Если указать значение `auto`, управлять размерами элемента интерфейса будет сама платформа Metro.

```
input[type=button] {
  width: 50px;
  height: 50px
}
```

Задаем для всех кнопок размеры 40×40 пикселов.

По умолчанию все элементы интерфейса (за несколькими исключениями, о которых мы поговорим потом) будут иметь ширину в 100%, т. е. растянутся на всю ширину родителя, и такую высоту, чтобы полностью вместить свое содержимое.

Далеко не всегда стоит жестко указывать для элемента интерфейса ширину и высоту, особенно если содержимое этого элемента будет создано в процессе работы приложения, и размер этого содержимого заранее неизвестен. Однако возможность задать для данного элемента минимальные и максимальные значения ширины и высоты, вероятно, нам пригодится.

Атрибуты стиля `min-width` и `min-height` позволяют задать минимальную ширину и высоту элемента интерфейса:

```
min-width: <ширина>
min-height: <высота>
```

Аналогичные атрибуты стиля `max-width` и `max-height` позволяют задать максимальную, соответственно, ширину и высоту элемента интерфейса:

```
max-width: <ширина>
max-height: <высота>
```

Значение ширины и высоты может быть задано в любой из поддерживаемых CSS единиц измерения.

```
#divList > div {
  min-width: 22%;
  max-width: 45%;
}
```

Параметры отступов

Язык CSS предлагает средства для создания отступов двух видов.

- Отступ между воображаемой границей элемента интерфейса и его содержимым — *внутренний* отступ. Такой отступ принадлежит данному элементу интерфейса, находится внутри его.
- Отступ между воображаемой границей данного элемента интерфейса и воображаемыми границами соседних элементов — *внешний* отступ. Такой отступ не принадлежит данному элементу интерфейса, находится вне его.

Атрибуты стиля `padding-left`, `padding-top`, `padding-right` и `padding-bottom` позволяют задать величины внутренних отступов, соответственно, слева, сверху, справа и снизу элемента интерфейса:

```
padding-left|padding-top|padding-right|padding-bottom: <отступ>|auto
```

Значение `auto` задает величину отступа по умолчанию; обычно оно равно нулю.

```
#divHeader { padding-bottom: 5mm; }
```

Задаем для блока `divHeader` внутренний отступ снизу, равный 5 мм.

Атрибут стиля `padding` позволяет сразу указать величины внутренних отступов со всех сторон элемента Web-страницы:

```
padding: <отступ 1> [<отступ 2> [<отступ 3> [<отступ 4>]]]
```

- Если указано одно значение, оно задаст величину отступа со всех сторон элемента интерфейса.
- Если указаны два значения, первое установит величину отступа сверху и снизу, а второе — слева и справа.
- Если указаны три значения, первое определит величину отступа сверху, второе — слева и справа, а третье — снизу.
- Если указаны четыре значения, первое задаст величину отступа сверху, второе — справа, третье — снизу, а четвертое — слева.

```
#divContent { padding: 1cm; }
```

Задаем для блока `divContent` внутренние отступы в 1 см.

Атрибуты стиля `margin-left`, `margin-top`, `margin-right` и `margin-bottom` позволяют задать величины внешних отступов, соответственно, слева, сверху, справа и снизу:

```
margin-left|margin-top|margin-right|margin-bottom: <отступ>|auto
```

Значение `auto` задает величину отступа по умолчанию, как правило, равное нулю.

```
#divList > div {
  margin-left: 10px;
  margin-right: 10px;
}
```

Внешние отступы мы также можем указать с помощью атрибута стиля `margin`. Он задает величины отступа одновременно со всех сторон элемента интерфейса:

```
margin: <отступ 1> [<отступ 2> [<отступ 3> [<отступ 4>]]]
```

Этот атрибут стиля ведет себя так же, как его "коллега" `padding`.

```
#divList > div { margin: 0px 10px; }
```

> ***Внимание!***
>
> Задавая отступы, внутренние или внешние, нужно помнить, что они увеличивают размеры элемента интерфейса. Поэтому если мы применим отступы к блокам, то должны будем соответственно изменить их размеры, иначе дизайн приложения будет нарушен.

Параметры рамки

CSS также позволяет нам создать сплошную, пунктирную или точечную рамку по воображаемой границе элемента интерфейса.

Атрибуты стиля `border-left-width`, `border-top-width`, `border-right-width` и `border-bottom-width` задают толщину, соответственно, левой, верхней, правой и нижней сторон рамки:

```
border-left-width|border-top-width|border-right-width|
border-bottom-width: thin|medium|thick|<толщина>
```

Мы можем указать либо числовое значение толщины рамки, либо одно из предопределенных значений: `thin` (тонкая), `medium` (средняя) или `thick` (толстая).

Атрибуты стиля `border-left-color`, `border-top-color`, `border-right-color` и `border-bottom-color` задают цвет, соответственно, левой, верхней, правой и нижней сторон рамки:

```
border-left-color|border-top-color|border-right-color|
border-bottom-color: transparent|<цвет>
```

Значение `transparent` задает "прозрачный" цвет, сквозь который будет "просвечивать" фон родительского элемента.

Атрибуты стиля `border-left-style`, `border-top-style`, `border-right-style` и `border-bottom-style` задают стиль линий, которыми будет нарисована, соответственно, левая, верхняя, правая и нижняя сторона рамки:

```
border-left-style|border-top-style|border-right-style|
border-bottom-style: none|hidden|dotted|dashed|solid|double|groove|
ridge|inset|outset
```

Здесь доступны следующие значения:

- `none` и `hidden` — рамка отсутствует (обычное поведение);
- `dotted` — пунктирная линия;
- `dashed` — штриховая линия;
- `solid` — сплошная линия;
- `double` — двойная линия;
- `groove` — "вдавленная" трехмерная линия;
- `ridge` — "выпуклая" трехмерная линия;
- `inset` — трехмерная "выпуклость";
- `outset` — трехмерное "углубление".

```
h1 {
  border-bottom-width: 5px;
  border-bottom-color: red;
  border-bottom-style: double;
}
```

Задаем для всех заголовков первого уровня двойную рамку красного цвета, состоящую только из нижней части. Фактически мы подчеркнем все такие заголовки двойной красной линией.

Атрибуты стиля `border-left`, `border-top`, `border-right` и `border-bottom` позволяют задать все параметры, соответственно, для левой, верхней, правой и нижней сторон рамки:

```
border-left|border-top|border-right|border-bottom:
<толщина> <стиль> <цвет>
```

Во многих случаях эти атрибуты стиля оказываются предпочтительнее:

```
h1 { border-bottom: 5px double red; }
```

"Глобальный" атрибут стиля `border` позволяет задать все параметры сразу для всех сторон рамки:

```
border: <толщина> <стиль> <цвет>
#divList > div { border: thin dotted black; }
```

> ***Внимание!***
>
> Рамки также увеличивают размеры элемента интерфейса. Поэтому, если мы создадим рамки у блоков, то должны будем соответственно изменить их размеры, иначе дизайн приложения будет нарушен.

Параметры отображения

Очень часто в процессе работы приложения какие-то элементы его интерфейса должны скрываться, а какие-то, наоборот, выводиться на экран. Указать, должен ли элемент интерфейса отображаться на экране, позволят нам два еще незнакомых нам атрибута стиля.

Атрибут стиля `visibility` позволяет указать, будет ли элемент интерфейса отображаться на экране:

```
visibility: visible|hidden
```

Он может принимать два значения:

- `visible` — элемент отображается на экране (это обычное поведение);
- `hidden` — элемент не отображается на экране, однако под него все еще выделено место. Другими словами, вместо элемента на экране видна пустая "прореха".

```
#divContent { visibility: hidden; }
```

Скрываем до поры до времени блок `divContent`.

Атрибут стиля `display` нам уже знаком — с его помощью мы создавали сеточную и гибкую разметки. Помимо этого, он позволяет в случае необходимости вообще скрыть элемент интерфейса, словно бы его и никогда не было:

```
display: none|inline|block
```

Здесь мы видим еще три значения, поддерживаемые этим атрибутом стиля:

- `none` — элемент интерфейса вообще не отображается на экране;
- `inline` — элемент интерфейса отображается как встроенный (значение по умолчанию для встроенных элементов);
- `block` — элемент интерфейса отображается как блочный (значение по умолчанию для блочных элементов).

Если нам потребуется скрыть какой-то элемент интерфейса, мы укажем для атрибута стиля `display` значение `none`.

```
#divContent { display: none; }
```

А чтобы вновь вывести его на экран, мы укажем для атрибута стиля `display` значение `inline`, если этот элемент встроенный, и `block`, если он блочный.

```
#divContent { display: block; }
```

> ***Внимание!***
> Если элемент интерфейса скрыт, методы, вычисляющие его размеры, могут работать некорректно. (Эти методы будут рассмотрены далее.)

Параметры выравнивания

Как задать выравнивание для содержимого элемента интерфейса, который позиционируется на экране с применением сеточной разметки, мы уже знаем. Но как сделать это для других элементов интерфейса?

Выравнивание по горизонтали можно указать с помощью знакомого нам по *главе 8* атрибута стиля `text-align`:

```
#divList > div { text-align: right; }
```

А для указания выравнивания по вертикали мы используем атрибут стиля `vertical-align`:

```
vertical-align: top|middle|bottom
```

Этот атрибут стиля поддерживает три значения: `top` (выравнивание по верхнему краю родителя), `middle` (по центру) и `bottom` (по нижнему краю). Значение по умолчанию — `top`.

```
#divList > div { vertical-align: middle; }
```

Свободно позиционируемые элементы интерфейса

Закончив рассмотрение дополнительных параметров элементов интерфейса, вернемся к разметке.

Обычно все элементы интерфейса, что мы создаем в HTML-коде, находятся на единой поверхности. Их расположением и размерами управляет платформа Metro,

руководствуясь либо созданной нами разметкой, либо, если мы ее не создали, своей внутренней логикой. Мы не можем произвольно поменять месторасположение какого-либо элемента интерфейса без изменения HTML-кода и (или) разметки. Также мы не можем поместить какой-либо элемент над всеми остальными элементами, составляющими интерфейс приложения, чтобы этот элемент перекрыл их.

Совсем другое дело — *свободно позиционируемые элементы интерфейса*. От обычных элементов их отличает следующее:

- мы можем задать для свободно позиционируемого элемента какие угодно местоположение и размеры и впоследствии как угодно изменять их;
- свободно позиционируемые элементы находятся "выше" обычных элементов интерфейса. При этом они могут полностью или частично перекрывать как обычные элементы интерфейса, так и другие свободно позиционируемые элементы.

В свободно позиционируемый элемент может быть превращен любой блок. Для этого используются особые атрибуты стиля CSS, которые мы скоро изучим.

Как правило, свободно позиционируемые элементы применяются для создания специальных элементов интерфейса, которые появляются только в определенный момент и служат либо для получения данных от пользователя (что-то наподобие диалоговых окон), либо для информирования его о чем-либо (окна-предупреждения). Для создания основного интерфейса их применять нецелесообразно.

Превратить обычный блок в свободно позиционируемый элемент нам поможет атрибут стиля `position`:

```
position: static|absolute
```

Из всех значений этого атрибута стиля нас интересуют всего два:

- `static` — элемент интерфейса ведет себя как обычно (это поведение по умолчанию);
- `absolute` — элемент интерфейса становится свободно позиционируемым.

```
<div id="divInfo">
  <h1>Пауза</h1>
</div>
. . .
#divInfo { position: absolute; }
```

Мы создали блок `divInfo` с надписью "Пауза" и превратили его в свободно позиционируемый элемент. Данный блок мы выведем на экран, как только пользователь поставит воспроизведение фильма на паузу.

Атрибуты стиля `left` и `top` задают, соответственно, горизонтальную и вертикальную координаты левого верхнего угла свободно позиционируемого элемента относительно левого верхнего угла его родителя:

```
left|top: <значение>|auto
```

Значения координат можно указывать в любых единицах измерения, поддерживаемых стандартом CSS. Значение `auto` отдает управление координатами свободно позиционируемого элемента платформе Metro.

```
#divInfo {
  left: 200px;
  top: 100px;
  width: 300px;
  height: 200px;
}
```

Указываем координаты, а заодно и размеры свободно позиционируемого элемента `divInfo`.

Как говорилось ранее, свободно позиционируемые элементы могут перекрывать друг друга. При этом элемент, определенный в HTML-коде позже, перекрывает элемент, определенный раньше. Однако мы можем сами задать порядок их перекрытия друг другом, указав так называемый *z-индекс*. Он представляет собой целое число, указывающее номер в порядке перекрытия; при этом элементы с бóльшим *z*-индексом перекрывают элементы с меньшим *z*-индексом. *Z*-индекс задается атрибутом стиля с "говорящим" именем `z-index`:

```
z-index: <номер>|auto
```

Как уже говорилось, *z*-индекс указывается в виде целого числа. Значение `auto` возвращает управление порядком перекрытия платформе Metro.

```
#divInfo { z-index: 99; }
```

Указываем для элемента `divInfo` *z*-индекс, равный 99, чтобы он гарантированно перекрыл все уже присутствующие на экране свободно позиционируемые элементы.

Программное управление местоположением, размерами и видимостью элементов интерфейса

Когда пользователь работает с приложением, в различные моменты времени он выполняет разные действия. Так, в один момент ему может потребоваться ввести интернет-адрес канала RSS, в другой — сохранить в файле текст выбранной в канале новости, в третий — выполнить отписку от какого-то канала.

Значит ли это, что на экране одновременно должны присутствовать и поле для ввода интернет-адреса, и кнопка сохранения файла, и список каналов с кнопкой отписки? Разумеется, нет — в этом случае интерфейс окажется перегруженным настолько, что приложением станет невозможно пользоваться.

Гораздо удобнее, когда элементы интерфейса, необходимые в строго определенные моменты, до поры до времени скрыты и появляются на экране, только когда в них

возникает нужда. Например, поле для ввода интернет-адреса канала RSS изначально можно сделать скрытым и выводить на экран при нажатии кнопки **Подписаться на канал**.

Если мы собираемся использовать в своем приложении свободно позиционируемые элементы (о них говорилось чуть ранее), нам может понадобиться изменить их местоположение и, возможно, размеры. В самом деле, мы не можем знать параметры экрана планшета, на котором будет работать наше приложение, а значит, мы не можем позиционировать эти элементы заранее на все случаи жизни.

Как все это сделать, мы сейчас и выясним. Заодно изучим несколько новых инструментов платформы Metro, которые нам потом очень пригодятся.

Программная привязка стилевых классов

Самый простой способ изменить местоположение, размеры или видимость элемента интерфейса — указать эти параметры в специально созданном стилевом классе и в процессе работы приложения привязать его к данному элементу. В этом нам помогут три метода объекта `WinJS.Utilities`, описанные далее.

Объект `WinJS.Utilities` предоставляет нам методы, выполняющие различные действия над интерфейсом приложения. Единственный экземпляр этого объекта создается платформой Metro и хранится в одноименной переменной.

Метод `addClass` позволяет привязать стилевой класс к элементу интерфейса.

```
WinJS.Utilities.addClass(<элемент>, <стилевой класс>)
```

Первым параметром указывается экземпляр объекта, представляющий элемент интерфейса, вторым — наименование стилевого класса без начальной точки в виде строки.

Метод `addClass` возвращает в качестве результата элемент интерфейса, к которому был привязан стилевой класс.

Метод `removeClass`, наоборот, убирает привязку стилевого класса к элементу интерфейса.

```
WinJS.Utilities.removeClass(<элемент>, <стилевой класс>)
```

Он принимает те же параметры и возвращает тот же результат, что и метод `addClass`.

Метод `toggleClass` привязывает стилевой класс к элементу интерфейса, если он еще не привязан, и удаляет его привязку в противном случае.

```
WinJS.Utilities.toggleClass(<элемент>, <стилевой класс>)
```

Этот метод принимает те же параметры и возвращает тот же результат, что и метод `addClass`.

Еще нам может пригодиться метод `hasClass`. Он позволяет узнать, привязан ли к какому-то элементу интерфейса данный стилевой класс.

```
WinJS.Utilities.hasClass(<элемент>, <стилевой класс>)
```

Этот метод принимает те же параметры, что и метод `addClass`, и возвращает `true`, если указанный стилевой класс привязан к элементу интерфейса, и `false` в противном случае.

```
<div>
  <input type="check" id="chkShowHide" />
  <label for="chkShowHide">Показать или скрыть</label>
</div>
<div id="divForm"> . . . </div>
. . .
.hidden-item { display: none; }
. . .
var chkShowHide, divForm;
function chkShowHideClick() {
  if (chkShowHide.checked) {
    WinJS.Utilities.removeClass(divForm, "hidden-item");
  } else {
    WinJS.Utilities.addClass(divForm, "hidden-item");
  }
}
document.addEventListener("DOMContentLoaded", function () {
  chkShowHide = document.getElementById("chkShowHide");
  divForm = document.getElementById("divForm");
  WinJS.Utilities.addClass(divForm, "hidden-item");
  chkShowHide.addEventListener("click", chkShowHideClick);
});
```

Здесь мы создаем флажок `chkShowHide` и блок `divForm`, который будет содержать элементы управления для ввода неких данных. Если флажок `chkShowHide` установлен, блок `divForm` будет выведен на экран, и наоборот.

Далее мы создаем стилевой класс `.hidden-item`, который запретит вывод на экран элемента интерфейса, к которому он привязан.

Напоследок мы программно привязываем к блоку `divForm` стилевой класс `.hidden-item`, ведь поскольку флажок `chkShowHide` изначально не установлен, значит, данный блок должен быть скрыт. И привязываем к событию `click` флажка `chkShowHide` обработчик, который будет выводить или скрывать этот блок при установке и сбросе флажка.

Программное указание параметров оформления

Другой способ задать новые значения для параметров оформления у элемента интерфейса — указать их непосредственно. Пожалуй, этот способ применяется чаще всего.

Все объекты, представляющие элементы интерфейса, поддерживают свойство `style`. Это свойство хранит экземпляр объекта `CSSStyleDeclaration`, представляющего определение стиля.

Это особенный стиль. Он не является ни стилевым классом, ни стилем переопределения тега, ни именованным стилем. Этот стиль действует только на данный элемент интерфейса, не затрагивая другие, и относится к встроенным стилям, о которых упоминалось еще в *главе 3*. Встроенные стили имеют самый высокий приоритет среди всех остальных разновидностей стилей; заданные в них значения атрибутов стиля гарантированно перекроют все, указанные в любых других стилях.

Стало быть, мы можем указать произвольные параметры оформления для какого-то элемента интерфейса, например, его местоположение, размеры и видимость, просто задав для атрибутов встроенного стиля нужные значения. Выполняется это с помощью свойств объекта `CSSStyleDeclaration`, которые представляют атрибуты встроенного стиля.

Имена данных свойств похожи на имена атрибутов стиля, которым они соответствуют. Давайте рассмотрим несколько примеров:

- свойство `left` соответствует атрибуту стиля `left`;
- свойство `backgroundColor` соответствует атрибуту стиля `background-color`;
- свойство `borderLeftColor` соответствует атрибуту стиля `border-left-color`;
- свойство `msGridColumns` соответствует атрибуту стиля `-ms-grid-columns`.

Как видим, правила именования свойств вполне понятны.

Все свойства объекта `CSSStyleDeclaration`, представляющие атрибуты стиля, принимают значения в виде строк в том виде, в котором они указываются у самих атрибутов стиля.

```
var divHeader = document.getElementById("divHeader");
divHeader.style.height = "200px";
```

Задаем новое значение высоты для блока `divHeader`. Обратим внимание, что мы указали здесь единицу измерения, в нашем случае — пикселы (`px`); если этого не сделать, платформа Metro не сможет правильно обработать заданное нами значение.

```
var divInfo = document.getElementById("divInfo");
divInfo.style.left = "400px";
divInfo.style.top = "300px";
```

Задаем новое местоположение для свободно позиционируемого элемента `divInfo`.

```
var divContent = document.getElementById("divContent");
divContent.style.display = "none";
```

Скрываем блок `divContent`.

```
divHeader.style.border = "thin dotted black";
```

А для блока `divHeader` еще и указываем параметры рамки.

Получение местоположения и размеров элементов интерфейса

Если мы собираемся указывать местоположение и размеры какого-то элемента интерфейса программно, нам может понадобиться выяснить местоположение и размеры других элементов интерфейса. Для этого мы используем методы объекта `WinJS.Utilities`, описанные далее.

Методы `getRelativeLeft` и `getRelativeTop` возвращают, соответственно, горизонтальную и вертикальную координаты левого верхнего угла заданного элемента интерфейса относительно левого верхнего угла указанного родителя.

```
getRelativeLeft|getRelativeTop(<элемент>, <родитель>)
```

Первым параметром указывается сам элемент, вторым — его родитель, относительно которого вычисляются координаты. Результат возвращается в виде числового значения в пикселах.

```
var x = WinJS.Utilities.getRelativeLeft(divInfo, document.body);
var y = WinJS.Utilities.getRelativeTop(divInfo, document.body);
```

Получаем координаты свободно позиционируемого элемента `divInfo` относительно всего интерфейса приложения.

Методы `getContentWidth` и `getContentHeight` позволяют узнать, соответственно, ширину и высоту свободного пространства в указанном элементе интерфейса без учета внутренних отступов и рамки. В качестве единственного параметра они принимают элемент интерфейса и возвращают числовое значение ширины или высоты в пикселах.

```
var avalWidth = WinJS.Utilities.getContentWidth(document.body);
var availHeight = WinJS.Utilities.getContentHeight(document.body);
```

Получаем ширину и высоту всего интерфейса приложения, фактически — размеры свободного пространства на экране.

А методы `getTotalWidth` и `getTotalHeight` возвращают полную, соответственно, ширину и высоту элемента интерфейса с учетом внешних отступов. В остальном они аналогичны уже рассмотренным нами методам `getContentWidth` и `getContentHeight`.

```
var width = WinJS.Utilities.getTotalWidth(divInfo);
var height = WinJS.Utilities.getTotalHeight(divInfo);
```

Получаем полные размеры свободно позиционируемого элемента `divInfo`.

> ***Внимание!***
>
> Если элемент интерфейса скрыт, методы, вычисляющие размеры элемента интерфейса, могут работать некорректно. В частности, некоторые из размеров элемента, возвращенные ими, вполне могут оказаться нулевыми.

Пример: дальнейшее совершенствование видеопроигрывателя

Настала пора применить полученные знания на практике. В качестве "подопытного кролика" на этот раз используем видеопроигрыватель `VideoPlayer`.

Давайте сделаем так, чтобы при постановке воспроизведения на паузу на экране появлялась надпись "Пауза". (Конечно, здесь лучше подойдет графическое изображение символа паузы, но пока сойдет и так.) Причем надпись эта будет появляться в центре экрана и частично перекрывать сам видеопроигрыватель.

Откроем в Visual Studio проект `VideoPlayer`. Переключимся на файл default.html и вставим перед закрывающим тегом `</body>` такой фрагмент кода:

```
<div id="divInfo">
  <h1>Пауза</h1>
</div>
```

Это обычный блок с именем `divInfo`, включающий в себя заголовок первого уровня "Пауза". Данный блок мы потом превратим в свободно позиционируемый элемент.

Теперь откроем файл default.css и добавим в него следующий код:

```
#divInfo {
  position: absolute;
  left: 0px;
  top: 0px;
  width: 300px;
  height: 200px;
  text-align: center;
  vertical-align: middle;
}
.hidden-element { display: none; }
```

Сначала мы создали именованный стиль, который превратит только что созданный блок `divInfo` в свободно позиционируемый элемент. Для этого блока мы указали такие размеры, чтобы его содержимое — заголовок "Пауза" — поместился в него полностью. Изначально мы поместили его в точку с координатами (0, 0) — верхний левый угол экрана — все равно окончательно позиционировать мы его будем программно. Еще мы задали для содержимого данного блока горизонтальное и вертикальное выравнивание по центру.

Также мы создали стилевой класс `.hidden-element`, что сделает элемент интерфейса, к которому он привязан, невидимым на экране.

Отметим, что мы не привязываем этот стилевой класс к блоку `divInfo` изначально, прямо в HTML-коде. Дело в том, что позднее мы будем получать размеры данного блока, а, как мы уже знаем, размеры скрытых элементов интерфейса вычисляются платформой Metro некорректно.

Настала пора внести исправления в код логики. Переключимся на файл default.js и сразу же дополним объявление всех необходимых глобальных переменных следующим образом:

```
var btnOpen, vidMain, divInfo;
```

Мы добавили сюда переменную, где будет храниться блок `divInfo`.

Теперь внесем в обработчик события `DOMContentLoaded` правки, после которых он будет выглядеть следующим образом:

```
document.addEventListener("DOMContentLoaded", function () {
  WinJS.UI.processAll();

  btnOpen = document.getElementById("btnOpen");
  vidMain = document.getElementById("vidMain");
  divInfo = document.getElementById("divInfo");

  var width = WinJS.Utilities.getTotalWidth(divInfo);
  var height = WinJS.Utilities.getTotalHeight(divInfo);
  var availWidth = WinJS.Utilities.getContentWidth(document.body);
  var availHeight = WinJS.Utilities.getContentHeight(document.body);
  divInfo.style.left = (availWidth - width) / 2 + "px";
  divInfo.style.top = (availHeight - height) / 2 + "px";
  WinJS.Utilities.addClass(divInfo, "hidden-element");

  btnOpen.addEventListener("click", btnOpenClick);
  vidMain.addEventListener("pause", vidMainPause);
  vidMain.addEventListener("playing", vidMainPlaying);
});
```

Сначала мы запускаем процесс формирования элементов управления Metro и получаем доступ ко всем элементам интерфейса, с которыми будем взаимодействовать в дальнейшем. Ничего особо нового здесь для нас нет.

Далее мы вычисляем размеры блока `divInfo`. Конечно, можно этого не делать, а "забить" в код значения размеров, указанные в именованном стиле `#divInfo`. Но в этом случае, если мы в процессе отладки приложения захотим изменить размеры блока `divInfo`, нам придется менять их в двух местах — в описании оформления и в коде логики. Понятно, что в таком случае возрастает вероятность допустить ошибку.

После всего этого мы вычисляем ширину и высоту всего интерфейса приложения (а на самом деле — свободного пространства экрана) и, пользуясь этими значениями, а также полученными ранее размерами блока `divInfo`, устанавливаем этот блок в центр экрана. И не забываем программно привязать к блоку `divInfo` стилевой класс `.hidden-element`, чтобы сделать его изначально скрытым.

Напоследок мы привязываем обработчики к событиям кнопки **Открыть** (`btnOpen`) и видеопроигрывателя `vidMain`.

Видеопроигрыватель поддерживает событие `pause`, возникающее при постановке воспроизведения на паузу. Объявление функции `vidMainPause`, которая станет обработчиком этого события, приведено далее.

```
function vidMainPause() {
  WinJS.Utilities.removeClass(divInfo, "hidden-element");
}
```

Она удалит привязку стилевого класса `.hidden-element` к блоку `divInfo`, сделав его видимым.

Событие `playing` возникает в видеопроигрывателе при начале или возобновлении воспроизведения фильма. Функция `vidMainPlaying`, которую мы сейчас объявим, станет обработчиком этого события.

```
function vidMainPlaying() {
  WinJS.Utilities.addClass(divInfo, "hidden-element");
}
```

Эта функция, наоборот, привязывает стилевой класс `.hidden-element` к блоку `divInfo`, тем самым скрывая его.

Сохраним все исправленные файлы и запустим приложение. Откроем какой-либо видеофайл, дождемся, пока он не начнет воспроизводиться, и поставим воспроизведение на паузу, щелкнув на соответствующей кнопке в видеопроигрывателе. На экране должно появиться сообщение "Пауза" (рис. 9.3).

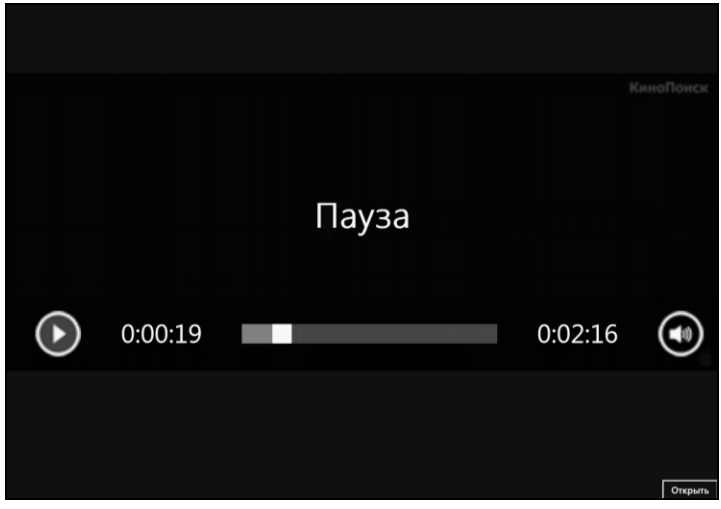


Рис. 9.3. Интерфейс усовершенствованного видеопроигрывателя после постановки воспроизведения на паузу

Дополнительные инструменты разметки

Осталось рассмотреть пару дополнительных инструментов, которые позволят нам реализовать различные специальные эффекты разметки. Обычно нужда в них возникает при выводе больших объемов текста.

Верстка текста в несколько колонок

Одна из замечательных возможностей платформы Metro — верстка текста в несколько колонок (multi-column layout). В сочетании с горизонтальной прокруткой (как реализовать прокрутку, будет рассказано далее) такой текст выглядит особенно оригинально.

Чтобы реализовать многоколоночную верстку текста, нам следует задать либо ширину колонок, либо их количество. Эти параметры указываются для элемента интерфейса, в котором будет выводиться многоколоночный текст.

Атрибут стиля `column-width` служит для задания оптимальной ширины колонок текста:

```
column-width: auto|<ширина колонок>
```

В качестве значения ширины может быть указана величина, заданная с помощью любых единиц измерения, которые поддерживаются стандартом CSS. Если задать значение `auto`, платформа Metro сама будет управлять шириной колонок; это, кстати, значение данного атрибута стиля по умолчанию.

```
#divContent { column-width: 200px; }
```

Указываем, что содержимое блока `divContent` будет верстаться в колонки шириной в 200 пикселов.

Атрибут стиля `column-count` применяется для задания либо оптимального, либо максимального количества колонок текста (подробности *см. далее*):

```
column-count: auto|<количество колонок>
```

В качестве его значения указывается ненулевое целое число. Если же задать значение `auto` (это значение по умолчанию для данного атрибута стиля), платформа Metro сама будет управлять количеством колонок.

```
#divContent { column-count: 3; }
```

Мы можем задать только ширину колонок, только их количество или и то, и другое одновременно.

- Если мы зададим только ширину колонок (атрибут стиля `column-width`), платформа Metro выведет столько колонок заданной ширины, чтобы заполнить все пространство родителя.

  Отметим, что реальная ширина колонок может оказаться несколько меньшей или большей. Так, если в родителе остается свободное пространство, Metro может расширить колонки, чтобы его заполнить. А если для родителя задана ширина меньшая, чем ширина колонок, колонки станут у́же, чтобы "втиснуться" в него.

- Если мы зададим только количество колонок (атрибут стиля `column-count`), Metro выведет именно столько колонок, сколько мы укажем; ширина их будет такая, чтобы заполнить все пространство родителя.

- Если мы укажем и ширину, и количество колонок, платформа Metro выведет столько колонок, сколько мы укажем, или меньше. Ширина колонок будет та-

кой, какую мы указали, с небольшими вариациями. В любом случае, Metro постарается и соблюсти заданные нами параметры колонок, и заполнить пространство родителя целиком.

- Если мы не укажем ни ширины, ни количества колонок, текст будет выведен как обычно — в одну колонку (это поведение по умолчанию).

Мы также можем использовать атрибут стиля `columns` для одновременного задания ширины и количества колонок:

```
columns: [<ширина колонок>] [<количество колонок>]
```

Если указано только значение ширины, количество колонок будет задано равным `auto` (т. е. платформа Metro сама будет управлять количеством колонок). Если же там задать только количество, ширина станет равной `auto`.

```
#divContent { columns: 200px 2; }
```

Задаем ширину колонок в 200 пикселов и количество колонок, равное двум.

Между отдельными колонками текста в любом случае будет выводиться небольшой просвет. Мы имеем возможность указать его ширину, для чего воспользуемся атрибутом стиля `column-gap`:

```
column-gap: normal|<ширина просвета между колонками>
```

Значение ширины может быть указано в любой единице измерения, поддерживаемой CSS. Значение `normal` устанавливает ширину просвета по умолчанию, равную ширине одной латинской буквы "m" (или, если использовать нотацию языка CSS, `1em`).

```
#divContent { column-gap: 5mm; }
```

Устанавливаем ширину просвета в 5 мм.

Также мы имеем возможность создать *разделители* — линии, визуально отделяющие колонки текста друг от друга. Для этого применяются атрибуты стиля, перечисленные далее:

- `column-rule-color` — задает цвет разделителей;
- `column-rule-style` — задает стиль разделителей. Поддерживает те же значения, что уже знакомые нам атрибуты стиля, указывающие стиль для линий рамки *(см. ранее в этой главе)*;
- `column-rule-width` — задает толщину разделителей. Поддерживает те же значения, что и знакомые нам атрибуты стиля, предназначенные для задания толщины рамки.

```
#divContent {
  column-rule-color: grey;
  column-rule-style: dotted;
  column-rule-width: thin;
}
```

Задаем тонкие серые разделители в виде точечных линий.

Чтобы указать все параметры разделителей за один раз, мы можем использовать атрибут стиля `column-rule`:

```
column-rule: <толщина> <стиль> <цвет>
#divContent { column-rule: thin dotted grey; }
```

По умолчанию все элементы интерфейса, для родителя которых задана многоколоночная верстка, занимают одну колонку. Однако мы имеем возможность поместить какой-либо элемент сразу в нескольких колонках, растянув его на всю ширину родителя. Таким элементом может быть, скажем, заголовок статьи.

Сделать это можно, указав для данного элемента атрибут стиля `column-span`. Формат его написания таков:

```
column-span: 1 | all
```

Он поддерживает всего два значения:

- `1` — элемент интерфейса размещается на одной колонке (значение по умолчанию);
- `all` — элемент интерфейса размещается сразу на всех колонках.

Как видим, мы можем поместить элемент либо только на одной, либо сразу на всех колонках. Растянуть его, скажем, только на две колонки мы не можем.

```
#divContent h1 { column-span: all; }
```

Задаем для заголовков первого уровня, что находятся в блоке `divContent`, размещение сразу на всех колонках.

Реализация прокрутки

Часто бывает так, что содержимое не помещается в элементе интерфейса, и либо размеры этого элемента увеличиваются, чтобы все вместить, либо какая-то часть его содержимого оказывается скрытой от пользователя. Такую ситуацию называют *переполнением*. Особенно часто переполнение возникает в элементах, в которых выводится текст.

Чтобы дать пользователю возможность в случае переполнения добраться до скрытой части содержимого, мы реализуем в элементе интерфейса средства для прокрутки. В этом нам помогут четыре особых атрибута стиля CSS.

Атрибуты стиля `overflow-x` и `overflow-y` задают поведение элемента интерфейса в случае переполнения, соответственно, по горизонтали и вертикали:

```
overflow-x|overflow-y: visible|scroll|auto|hidden
```

Здесь мы видим четыре доступных значения:

- `visible` — соответствующий размер элемента увеличится, чтобы полностью вместить все содержимое (обычное поведение). Однако в этом случае часть элемента может оказаться за краем экрана или другого элемента интерфейса, и добраться до скрытой части содержимого не получится;

- `scroll` — в элементе появятся полосы прокрутки, с помощью которых можно будет добраться до скрытой части содержимого. Эти полосы прокрутки будут присутствовать в элементе всегда, даже если в них нет нужды;
- `auto` — то же самое, что `scroll`, но полосы прокрутки появятся в блоке, только если в них возникнет необходимость;
- `hidden` — не помещающееся в элемент содержимое будет скрыто, и добраться до него не получится.

```
#divContent {
  overflow-x: auto;
  overflow-y: hidden;
}
```

Задаем для блока `divContent` горизонтальную прокрутку.

Атрибут стиля `overflow` позволит нам задать поведение элемента интерфейса в случае переполнения сразу по обоим направлениям. Он поддерживает те же значения, что атрибуты стиля `overflow-x` и `overflow-y`.

```
#divContent { overflow: scroll; }
```

Пример: усовершенствованный прототип приложения для чтения каналов RSS

Напоследок попрактикуемся в многоколоночной верстке текста и создании прокрутки. Реализуем все это в прототипе приложения для чтения каналов RSS.

Откроем в Visual Studio проект `RSSReader`. Переключимся на файл default.html и зададим в качестве содержимого блока `divContent` какой-либо большой текст, разбитый на абзацы и заголовки.

```
<div id="divContent">
  <p>18.01.2012</p>
  <h2>Опубликованы минимальные требования к устройствам под
  управлением Windows 8</h2>
  <p>В прошлом месяце Microsoft опубликовала свою документацию по
  аппаратным требованиям для сертификации Windows 8. Этот документ
  содержит рекомендации от Microsoft для разработки систем, которые
  соответствуют референсным по производительности, качеству и прочим
  критериям, для обеспечения оптимального удобства работы Windows 8.</p>
  . . .
</div>
```

Этот текст должен быть действительно большим, чтобы платформа Metro разбила его на несколько колонок и чтобы при этом гарантированно возникла потребность в прокрутке. Если мы не найдем подходящий текст, то придумаем его сами — в конце концов, это всего лишь прототип приложения.

Откроем файл default.css и исправим созданный ранее именованный стиль `#divContent`, чтобы он выглядел так:

```
#divContent {
  -ms-grid-column: 2;
  -ms-grid-row: 2;
  column-count: 2;
  overflow-x: auto;
  overflow-y: hidden;
}
```

Комментарии здесь излишни — все нам уже знакомо.

Сохраним все файлы и запустим приложение. На рис. 9.4 представлена правая часть экрана; видно, что текст разбит на колонки, а в нижней части заметна полоса прокрутки. Это значит, что наш прототип работает!

Рис. 9.4. Интерфейс прототипа приложения `RSSReader` после усовершенствования

Что дальше?

Эта глава получилась большой и насыщенной. Мы узнали, как создается разметка, разобрались с параметрами оформления, имеющими отношение к разметке, поговорили о свободно позиционируемых элементах, попробовали свои силы в программном управлении оформлением элементов интерфейса, научились разбивать текст на колонки и реализовывать прокрутку. После всего этого, пожалуй, потребуется небольшой перерыв.

В следующей главе, собравшись с силами, мы займемся программным созданием элементов интерфейса. Если мы собираемся создавать приложения, выводящие на экран объемистые данные в качестве результата работы, нам это очень пригодится.

ГЛАВА 10

Программное формирование элементов интерфейса

В предыдущей главе мы занимались разметкой: сетчатой, гибкой, разметкой с применением свободно позиционируемых элементов и многоколоночной верстки текста с возможностью прокрутки. Заодно мы познакомились со средствами CSS для задания параметров элементов, напрямую влияющих на разметку, и инструментами, позволяющими задать эти параметры программно.

Что ж, основные принципы формирования интерфейса Metro-приложений мы освоили. Их нам будет достаточно в большинстве случаев.

Но не во всех. Зачастую невозможно определить, какие элементы интерфейса должны присутствовать на экране изначально. Так, приложение для чтения каналов RSS первоначально не "знает", какие новости вывести на экран, — для этого оно должно сначала загрузить из Интернета файл с содержимым канала и обработать его. Поэтому часто приходится формировать определенные элементы интерфейса программно.

Программное создание элементов интерфейса

Платформа Metro предоставляет нам два способа сделать это: простой, но чреватый ошибками, и сложный, но более надежный. Мы рассмотрим оба.

Простой способ: прямое указание HTML-кода

Самый простой способ создать новое содержимое в каком-либо элементе интерфейса — сформировать строку с соответствующим HTML-кодом и присвоить его свойству `innerHTML` нужного элемента. Это свойство поддерживается всеми объектами, представляющими элементы интерфейса.

```
var divHeader = document.getElementById("divHeader");
var s = "<h1>Другой канал новостей</h1>";
divHeader.innerHTML = s;
```

Здесь мы получаем доступ к блоку `divHeader` из прототипа приложения для чтения каналов RSS и задаем для него в качестве нового содержимого заголовок первого уровня "Другой канал новостей".

Достоинство у этого способа одно — простота. Недостатков же два. Во-первых, в HTML-коде довольно легко допустить ошибку, в результате чего новое содержимое будет отображаться некорректно. Во-вторых, предыдущее содержимое элемента интерфейса будет полностью заменено, так что добавить туда что-либо просто так не получится.

Обычно первый способ применяется для указания нового содержимого небольших элементов интерфейса либо для вывода их нового текстового содержимого. Если же требуется создать большой фрагмент содержимого, удобнее использовать другой способ, который сложнее, но быстрее и безопаснее.

Сложный способ: сборка элементов интерфейса

Сложный способ заключается в том, что новое содержимое как бы собирается из отдельных элементов интерфейса, каждый из которых создается программно, с применением особых методов. Процесс создания новых элементов интерфейса можно разбить на несколько этапов.

Собственно создание элемента интерфейса

Это первый этап. Сначала нам нужно создать сам новый элемент интерфейса. Для этого мы используем метод `createElement` объекта `HTMLDocument`. (Вспомним: этот объект представляет все описание интерфейса Metro-приложения вместе со служебными тегами; единственный экземпляр этого объекта создается самой платформой Metro и доступен через переменную `document`.)

```
document.createElement(<имя тега>)
```

В качестве единственного параметра этого метода указывается имя тега, на основе которого должен быть создан элемент интерфейса, без символов < и > в виде строки.

Метод `createElement` возвращает экземпляр объекта, представляющий вновь созданный элемент интерфейса.

Следует сразу отметить, что созданный с применением этого метода элемент существует только в памяти компьютера. На экран он в любом случае выведен не будет, хотя бы потому, что мы еще не указали, где его следует поместить. Как вывести его на экран, мы узнаем потом.

```
var oP = document.createElement("p");
```

Создаем абзац.

```
var oInput = document.createElement("input");
```

Создаем элемент управления HTML.

Создание текстового содержимого

Если создаваемый нами элемент интерфейса должен включать текстовое содержимое, нашим следующим шагом будет его формирование.

Проще всего сделать это, присвоив строку с текстовым содержимым свойству `textContent` созданного элемента. Это свойство поддерживается всеми объектами, представляющими элементы управления.

```
oP.textContent = "Microsoft объяснила поддержку датчиков в Windows 8";
```

Задаем для только что созданного абзаца в качестве содержимого текст новости.

Задание параметров элемента интерфейса

Теперь мы зададим параметры для созданного элемента интерфейса. Выполняется это с помощью набора особых свойств. Фактически они представляют атрибуты тега, что формирует данный элемент интерфейса, и поддерживаются объектом, который его представляет.

```
oInput.type = "button";
```

Указываем кнопку в качестве типа создаваемого элемента управления. Для этого мы присваиваем свойству `type` объекта `HTMLInputElement` строку `"button"`. Данное свойство представляет атрибут `type` тега `<input>`.

```
oInput.value = "Закрыть";
```

Задаем надпись для кнопки, присвоив свойству `value` объекта `HTMLInputElement` строку `"Закрыть"`. Это свойство представляет атрибут `value` тега `<input>`.

```
oInput.disabled = true;
```

Пусть кнопка будет изначально недоступной.

Как видим, каждый атрибут тега имеет "пару" в виде свойства соответствующего объекта. Свойств очень много, и их конкретный набор зависит от элемента интерфейса, что мы создали.

Однако все объекты, представляющие элементы интерфейса, в обязательном порядке поддерживают два свойства, которые мы сейчас рассмотрим.

Свойство `className` хранит имя привязанного к элементу стилевого класса в виде строки. Фактически оно представляет атрибут тега `class`. Имя класса указывается без символа точки в начале.

```
oP.className = "other-class";
```

Привязываем к абзацу стилевой класс `.other-class`.

Свойство `id` соответствует атрибуту тега `id` и позволяет получить или задать имя для элемента интерфейса (и привязать к нему именованный стиль). Значение этого свойства указывается в виде строки.

```
oInput.id = "btnClose";
```

Задаем имя для кнопки.

Вывод созданного элемента интерфейса на экран

Последний шаг, который нам следует сделать, — собственно вывести созданный элемент интерфейса на экран.

Это позволяет сделать метод `appendChild`. Он помещает выводимый на экран элемент в самый конец содержимого другого элемента, который в результате станет его родителем. Этот метод поддерживается всеми объектами, представляющими элементы интерфейса.

```
<будущий родитель>.appendChild(<выводимый элемент — будущий потомок>)
```

Метод `appendChild` вызывается у того элемента интерфейса, который должен стать родителем для выводимого на экран элемента. А сам выводимый на экран элемент задается в качестве единственного параметра этого метода.

```
var divList = document.getElementById("divList");
divList.appendChild(oP);
```

Выводим только что созданный абзац на экран, добавив его в конец блока `divList`; в результате данный абзац станет последним потомком блока.

```
document.body.appendChild(oInput);
```

Помещаем созданную нами кнопку в самый конец интерфейса приложения (тега `<body>`).

Созданные таким образом элементы интерфейса немедленно появятся на экране и станут полноправной частью объектной модели документа DOM (о ней говорилось в *главе 5*). С точки зрения платформы Metro они ничем не отличаются от элементов, определенных в HTML-коде. Мы можем выполнять над такими элементами все действия, описанные в предыдущих главах: привязывать обработчики событий, задавать местоположение и размеры, делать недоступными для ввода и пр.

Программное удаление элементов интерфейса

В процессе работы приложения нам может потребоваться не только программно создавать какие-то элементы его интерфейса, но и программно их удалять. Например, в приложении для чтения каналов RSS перед тем, как программно выводить список статей, обязательно следует удалить из этого списка уже присутствующие там позиции. Можно ли это сделать? Конечно!

Метод `removeChild` удаляет заданный потомок из указанного элемента интерфейса.

```
<родитель>.removeChild(<удаляемый потомок>);
```

Метод `removeChild` вызывается для элемента интерфейса, в котором находится удаляемый нами элемент. Единственным параметром этому методу передается элемент, который должен быть удален.

```
var btnClose = document.getElementById("btnClose");
document.body.removeChild(btnClose);
```

Получаем доступ к ранее созданной и помещенной на экран кнопке `btnClose` и удаляем ее.

Если нам нужно удалить сразу все содержимое какого-либо элемента интерфейса, мы можем пойти двумя путями. Во-первых, мы можем получить доступ к этому элементу и присвоить его свойству `innerHTML` пустую строку.

```
divList.innerHTML = "";
```

Удаляем все содержимое блока `divList`, т. е. очищаем список новостей.

Во-вторых, мы можем вызвать метод `empty` объекта `WinJS.Utilities`.

```
WinJS.Utilities.empty(<родитель, чьих потомков следует удалить>)
```

Единственным параметром этому методу передается элемент интерфейса, чьи потомки должны быть удалены.

```
WinJS.Utilities.empty(divList);
```

Как только мы удалим какой-либо элемент интерфейса, он сразу исчезнет и с экрана, и из объектной модели документа DOM. При этом также будет удалено все его содержимое и все привязанные к нему обработчики событий.

Пример: усовершенствованный прототип приложения для чтения каналов RSS

В качестве практического занятия давайте доработаем приложение для чтения каналов RSS таким образом, чтобы оно формировало список новостей программно и программно же выводило текст новости, на которую указал пользователь.

Откроем в Visual Studio проект `RSSReader`. Переключимся на файл default.html и удалим все содержимое блоков `divList` и `divContent`.

После этого переключимся на файл default.js. Сначала вставим следующий код, объявляющий необходимые переменные и массив, который хранит список новостей:

```
var divList, divContent;
var arrNews = [];
arrNews[0] = [];
arrNews[0][0] = "17.01.2012";
arrNews[0][1] = "Microsoft рассказала о ReFS, новой файловой системе
↳Windows Server 8";
arrNews[0][2] = "В своем блоге B8 компания Microsoft поделилась
↳информацией о новой файловой системе Windows Server 8, известной под
↳названием ReFS и призванной заменить NTFS, которая используется в
↳Windows с 1993 года.";
arrNews[1] = [];
arrNews[1][0] = "18.01.2012";
arrNews[1][1] = "Опубликованы минимальные требования к устройствам под
↳управлением Windows 8";
arrNews[1][2] = "В прошлом месяце Microsoft опубликовала свою документацию
↳ по аппаратным требованиям для сертификации Windows 8. Этот документ
```

```
↳ содержит рекомендации от Microsoft для разработки систем, которые
↳ соответствуют референсным по производительности, качеству и прочим
↳ критериям, для обеспечения оптимального удобства работы Windows 8.";
. . .
```

Для хранения списка новостей мы объявили массив `arrNews`. Каждый его элемент будет хранить одну новость и также представлять собой массив, первый элемент которого сохранит дату публикации новости (для простоты запишем ее в виде строки), второй — заголовок, а третий — содержимое.

Далее создадим обработчик события `DOMContentLoaded`.

```
document.addEventListener("DOMContentLoaded", function () {
  divList = document.getElementById("divList");
  divContent = document.getElementById("divContent");
  fillList();
});
```

Здесь мы получаем доступ к блокам `divList` и `divContent` и вызываем функцию `fillList()`, которая заполнит список новостей.

Код, объявляющий функцию `fillList()`, таков:

```
function fillList() {
  var oDiv, oPDate, oPHeader;
  WinJS.Utilities.empty(divList);
  for (var i = 0; i < arrNews.length; i++) {
    oDiv = document.createElement("div");
    oPDate = document.createElement("p");
    oPDate.textContent = arrNews[i][0];
    oDiv.appendChild(oPDate);
    oPHeader = document.createElement("p");
    oPHeader.textContent = arrNews[i][1];
    oDiv.appendChild(oPHeader);
    oDiv.id = "div" + i;
    oDiv.addEventListener("click", divItemClick);
    divList.appendChild(oDiv);
  }
}
```

Здесь мы очищаем блок `divList` от уже имеющегося там содержимого и для каждого элемента массива `arrNews` выполняем следующие действия:

1. Создаем блок, который будет представлять собой позицию списка новостей.
2. Создаем абзац даты публикации, задаем для него строку с этой датой в качестве текстового содержимого и добавляем данный абзац в ранее созданный блок.
3. Проделываем те же самые действия для создания абзаца с заголовком новости.
4. В качестве имени блока задаем строку вида `div<`*индекс массива, хранящего новость, которую представляет данный блок*`>`.

5. Привязываем к блоку функцию `divItemClick()` в качестве обработчика события `click`.
6. Добавляем блок, представляющий позицию в списке новостей, в блок `divList`.

Осталось объявить функцию `divItemClick()`.

```
function divItemClick(evt) {
  var sID = evt.currentTarget.id;
  var sIndex = sID.substr(3);
  var iIndex = parseInt(sIndex);
  divContent.textContent = arrNews[iIndex][2];
}
```

Эта функция выполнит следующие действия:

1. Получит с единственным параметром `evt` экземпляр объекта `MSGestureEvent`, хранящий сведения о событии. (Об этом объекте и его свойствах рассказывалось в *главе 5*.)
2. Обратится к свойству `currentTarget` экземпляра объекта `MSGestureEvent`, чтобы получить элемент интерфейса, в котором выполняется обработка возникшего события. Этим элементом в нашем случае будет блок, представляющий новость, на которую указал пользователь.

   Здесь нужно отметить один важный момент. Когда пользователь указывает пальцем на новость, событие `click` возникает не в самом блоке, а в одном из находящихся в нем абзацев — дате публикации новости или ее заголовке. И только потом данное событие в результате процесса всплытия *(см. главу 5)* возникает в блоке.

   Если мы обратимся к свойству `target` экземпляра объекта `MSGestureEvent`, то получим элемент интерфейса, в котором это событие возникло изначально, — один из этих абзацев. Следовательно, мы не сможем потом узнать индекс элемента массива `arrNews`, в котором хранится соответствующая новость. Поэтому мы используем свойство `currentTarget`.
3. Получает из свойства `id` полученного элемента имя, которое мы для него задали. Вспомним, что это имя имеет формат `div<`*`индекс массива, хранящего новость, которую представляет данный блок`*`>`.
4. Выделит из полученного ранее имени индекс массива, что хранит данные о соответствующей новости. Для этого она вызовет метод `substr` объекта `String`, о котором говорилось в *главе 4*, передав ему в качестве единственного параметра число 3 — номер символа, с которого начинается нужный нам индекс.
5. Преобразует полученный индекс в числовой вид вызовом встроенной функции `parseInt()` (об этой функции также говорилось в *главе 4*).
6. Выведет содержимое новости в блок `divContent`.

Сохраним все исправленные файлы и запустим приложение на выполнение. Попробуем нажать на какую-либо новость в списке и посмотрим, будет ли на экран выведено ее содержимое. Если это случится, значит, мы все сделали правильно.

Что дальше?

В этой главе мы учились создавать элементы интерфейса программно. Мы изучили два способа сделать это: простой, но небезопасный, и сложный, но более устойчивый к ошибкам. И в качестве практики в очередной раз усовершенствовали прототип приложения для чтения каналов RSS.

Эта глава является заключительной в части, посвященной основам разработки пользовательского интерфейса Metro-приложений. Начиная со следующей части, мы будем рассматривать более сложные и специальные вопросы Metro-программирования.

Так, следующая глава будет посвящена инструментам, позволяющим вывести на экран графические изображения. Прочитав ее, мы сможем сделать свои Metro-приложения еще привлекательнее.

ЧАСТЬ III

Работа с графикой и мультимедиа

ГЛАВА 11

Вывод графических изображений

Предыдущая часть книги рассказывала о базовых средствах формирования пользовательского интерфейса Metro-приложений. Мы познакомились с элементами управления HTML и Metro, инструментами для вывода, форматирования и оформления текста, научились формировать разметку и создавать часть интерфейса программно. Этих знаний нам хватит для разработки простейших приложений; собственно, несколько таких приложений мы уже создали.

Эта часть будет посвящена выводу графики и работе с мультимедиа. И начнем мы с рассмотрения инструментов, позволяющих вывести на экран графическое изображение.

Графика в Metro-приложениях может играть разные роли. Это могут быть графические кнопки (кнопки, содержащие изображение вместо текстовой надписи), элементы оформления (например, фигурные рамки), графический фон, какие-то части интерфейса (указатели, предупреждения и пр.). В конце концов, наше приложение может предназначаться для просмотра изображений, загружаемых из локальной памяти компьютера или Интернета!

Графические форматы, поддерживаемые платформой Metro

Начать следует с перечисления форматов графических файлов, которые поддерживаются платформой Metro:

- GIF (Graphics Interchange Format, формат обмена графикой);
- JPEG (Joint Photographic Experts Group, группа экспертов в области неподвижной фотографии);
- PNG (Portable Network Graphics, перемещаемая сетевая графика);
- BMP (Windows BitMaP, битовая матрица Windows);
- WMF (Windows MetaFile, метафайл Windows);

- EMF (Windows Enhanced Metafile, расширенный метафайл Windows);
- SVG (Scalable Vector Graphics, масштабируемая векторная графика).

Как видим, здесь присутствуют и популярные форматы интернет-графики (GIF, JPEG и PNG), и системные графические форматы Windows (BMP, WMF и EMF).

Никакие другие графические форматы не поддерживаются.

Средства HTML для вывода графических изображений

Для вывода на экран графического изображения, хранящегося в файле, применяется одинарный тег `<img>`. Единственный обязательный атрибут `src` этого тега служит для указания ссылки на файл с изображением.

```
<img src="/images/picture.jpg" />
```

Этот HTML-код выведет на экран изображение, хранящееся в файле picture.jpg, что находится в папке images.

Графические изображения представляют собой встроенные элементы. Это значит, что мы можем вставить изображение прямо в абзац.

```
<p>Платформа Metro входит в состав
<img src="/images/microsoft_logo.png" /> Windows 8.</p>
```

Необязательные атрибуты `width` и `height` тега `<img>` позволяют задать, соответственно, ширину и высоту, которые изображение будет иметь после вывода на экран. Значения ширины и высоты указываются в виде целых чисел в пикселах.

```
<img src="/images/picture.jpg" width="400" height="300" />
```

Теперь графическое изображение будет иметь размеры 400×300 пикселов.

Если атрибуты тега `width` и `height` не указывать, изображение получит свои изначальные размеры, записанные в файле, где оно хранится.

А теперь — важный момент! Как мы только что выяснили, графические изображения, применяемые в Metro-приложениях, хранятся в отдельных файлах. Эти файлы загружаются и обрабатываются платформой Metro одновременно с загрузкой и обработкой файлов с описанием интерфейса, оформления и логики. И нет никакой гарантии, что к моменту, когда приложение будет готово к работе (т. е. когда возникнет событие `DOMContentLoaded`), изображения будут успешно загружены и выведены на экран.

В подобных случаях нам может пригодиться необязательный атрибут тега `alt`. Он позволяет задать так называемый *текст замены* — текст, который выводится на экран в то место, где должно находиться еще не загруженное изображение. Как только изображение будет загружено, оно появится на экране, а текст замены пропадет.

```
<img src="/images/picture.jpg"
alt="Здесь должна быть красивая картинка" />
```

Хотя, разумеется, текст замены должен быть более осмысленным и, по возможности, коротким.

Графическое изображение представляется объектом `HTMLImageElement`. Он поддерживает свойства `src`, `width` и `height`, соответствующие одноименным атрибутам тега. Это значит, что мы можем программно управлять параметрами графического изображения и даже в процессе работы приложения заменить одно изображение другим.

```
<img id="imgPicture" src="/images/picture1.jpg" />
. . .
var imgPicture = document.getElementById("imgPicture");
imgPicture.src = "/images/picture2.jpg ";
```

Программно заменяем изображение picture1.jpg изображением picture2.jpg.

Еще объект `HTMLImageElement` поддерживает два события, которые могут быть нам полезны:

- `load` — возникает сразу же после успешной загрузки и вывода изображения;
- `error` — возникает, если в процессе загрузки изображения появилась ошибка. Такое может случиться, например, если указанный файл с изображением отсутствует на диске, недоступен по сети или имеет неподдерживаемый формат.

```
imgPicture.addEventListener("load", function() { . . . });
```

Теперь мы сможем отследить момент, когда изображение будет наконец-то загружено и выведено на экран.

Реализация масштабирования графики с помощью жестов

Раз уж зашла речь о выводе изображений, давайте рассмотрим средства, которые позволят нам реализовать масштабирование изображения с помощью жестов. Эти средства очень просты; нам даже не придется создавать никакой дополнительной логики — достаточно указать несколько атрибутов стиля.

Сразу отметим, что эти средства можно применять только для блочных элементов. Следовательно, поскольку изображение является встроенным элементом, нам потребуется заключить его в блок.

Надо сказать, что данные средства можно применять для масштабирования любого другого содержимого, например текста. Просто чаще всего такое проделывают с изображениями.

Реализация масштабирования

Первое, что нам потребуется сделать, — реализовать в блоке возможность прокрутки содержимого. Если блок не является частью сеточной или гибкой разметки, то, скорее всего, придется задать для него размеры.

```
<div id="divZoom">
  <img src="/images/sample.jpg" />
</div>
. . .
#divZoom {
  width: 100%;
  height: 100%;
  overflow: auto;
}
```

Помещаем изображение в блок `divZoom`, растягиваем его на всю площадь родителя и задаем возможность прокрутки содержимого сразу во всех направлениях.

Следующий наш шаг — собственно указание, что пользователь может масштабировать содержимое данного элемента с помощью жестов. Выполняется это с помощью атрибута стиля `-ms-content-zooming`:

```
-ms-content-zooming: none|zoom
```

Значение `none` указывает, что пользователь не может масштабировать изображение жестами (это поведение по умолчанию), а значение `zoom` — что может.

```
#divZoom { -ms-content-zooming: zoom; }
```

На этом реализация масштабирования закончена.

Еще мы можем задать для блока с масштабируемым содержимым дополнительные параметры, влияющие на процесс масштабирования. Рассмотрим эти параметры и атрибуты стиля, "ответственные" за них.

Атрибуты стиля `-ms-content-zoom-boundary-min` и `-ms-content-zoom-boundary-max` задают, соответственно, минимальное и максимальное значения масштаба, которые может установить пользователь для содержимого данного элемента.

```
-ms-content-zoom-boundary-min|-ms-content-zoom-boundary-max: <масштаб>
```

Здесь допускаются только относительные значения, заданные в процентах. Значения по умолчанию — `100%` и `300%` соответственно.

```
#divZoom {
  -ms-content-zoom-boundary-min: 10%;
  -ms-content-zoom-boundary-max: 200%;
}
```

Теперь пользователь сможет менять масштаб изображения строго в диапазоне 10–200%.

Атрибут стиля `-ms-content-zoom-boundary` позволит нам указать сразу и минимальное, и максимальное значения масштаба:

```
-ms-content-zoom-boundary: <минимальный масштаб> <максимальный масштаб>
#divZoom { -ms-content-zoom-boundary: 10% 200%; }
```

По умолчанию пользователь может установить для содержимого элемента произвольное значение масштаба (плавное масштабирование). Но часто бывает удобнее

скачкообразное масштабирование, когда масштаб может принимать одно из фиксированных значений (например, `50%`, `100%`, `200%` и т. д.). При этом пользователь может как иметь возможность все-таки установить произвольное значение масштаба, так и не иметь ее.

Атрибут стиля `-ms-content-zoom-snap-type` позволяет указать режим скачкообразного масштабирования:

```
-ms-content-zoom-snap-type: none|proximity|mandatory
```

Он поддерживает три значения:

- `none` — скачкообразное масштабирование не применяется; содержимое будет масштабироваться плавно;
- `proximity` — применяется скачкообразное масштабирование, причем пользователь все же имеет возможность выбрать произвольное значение масштаба, находящееся между указанными нами фиксированными значениями;
- `mandatory` — применяется скачкообразное масштабирование, но пользователь не может выбрать произвольное значение масштаба.

```
#divZoom { -ms-content-zoom-snap-type: proximity; }
```

Сам набор фиксированных значений масштаба указывается при помощи атрибута стиля `-ms-content-zoom-snap-points`:

```
-ms-content-zoom-snap-points: <параметры интервала между значениями>|
↳<список произвольных значений>
```

Здесь можно указать либо начальное значение и интервал между остальными значениями, либо список произвольных значений.

- Начальное значение и интервал между остальными — предыдущими и последующими — значениями указывается с применением функции CSS `snapInterval` в таком формате:

  ```
  snapInterval(<начальное значение>,<интервал>)
  ```

  Оба значения могут быть заданы в любых единицах измерения, поддерживаемых CSS.

  ```
  #divZoom { -ms-content-zoom-snap-points: snapInterval(10%,20%); }
  ```

  Начальное значение будет равно 10% от исходного масштаба, а остальные значения будут отличаться друг от друга на 20%. В результате пользователь сможет установить для изображения значения масштабов `10%`, `30%`, `50%` и т. д.

- Список произвольных значений указывается с помощью функции CSS `snapList()` в таком формате:

  ```
  snapList(<значение 1>,<значение 2> . . .)
  ```

  Значения задаются в любых единицах измерения, поддерживаемых CSS.

  ```
  #divZoom { -ms-content-zoom-snap-points:
             snapList(10%,50%,75%,100%,200%); }
  ```

После этого пользователь сможет установить для нашего изображения значения масштаба `10%`, `50%`, `75%`, `100%` и `200%`.

С помощью атрибута стиля `-ms-content-zoom-snap` мы можем указать сразу все параметры списка значений, которые может принимать масштаб:

```
-ms-content-zoom-snap: <режим скачкообразного масштабирования>
↳<список значений масштаба>
#divZoom { -ms-content-zoom-snap: proximity snapInterval(50%,10%); }
```

Дополнительные возможности прокрутки

Как мы уже выяснили, возможность прокрутки является необходимым условием для реализации масштабирования с помощью жестов. Как реализуется прокрутка, мы также знаем *(см. главу 9)*.

Однако платформа Metro позволяет нам задать дополнительные параметры прокрутки. Какие именно — мы сейчас выясним.

Прежде всего, прокрутка, как и масштабирование, может быть плавной или скачкообразной. Для указания параметров скачкообразной прокрутки используются три атрибута стиля:

- `-ms-scroll-snap-type` — задает режим прокрутки сразу и по горизонтали, и по вертикали;
- `-ms-scroll-snap-points-x` — задает набор значений прокрутки по горизонтали;
- `-ms-scroll-snap-points-y` — задает набор значений прокрутки по вертикали.

Значения этих атрибутов стиля задаются в таких же форматах, как и для рассмотренных нами ранее атрибутов стиля `-ms-content-zoom-snap-type` и `-ms-content-zoom-snap-points`.

Значение по умолчанию атрибутов стиля `-ms-scroll-snap-points-x` и `-ms-scroll-snap-points-y` — `snapInterval(0px,100%)`, т. е. начало и конец содержимого.

```
#divZoom {
  -ms-scroll-snap-type: mandatory;
  -ms-scroll-snap-points-x: snapInterval(0px,100px);
  -ms-scroll-snap-points-y: snapInterval(0px,200px);
}
```

Атрибуты стиля `-ms-scroll-snap-x` и `-ms-scroll-snap-y` позволяют задать сразу все параметры скачкообразной прокрутки отдельно для горизонтального и вертикального направлений соответственно.

```
-ms-scroll-snap-x|-ms-scroll-snap-y:
↳<режим скачкообразной прокрутки> <список значений прокрутки>
#divZoom {
  -ms-scroll-snap-x: mandatory snapInterval(0px,100px);
  -ms-scroll-snap-y: proximity snapInterval(0px,200px);
}
```

Как видим, с помощью этих атрибутов стиля мы можем указать для разных направлений различные режимы скачкообразной прокрутки.

По умолчанию прокрутка содержимого может выполняться в произвольном направлении. Но платформа Metro позволяет нам реализовать прокрутку строго по горизонтальной и вертикальной координатным осям; при этом случайные движения пальца в направлении, перпендикулярном направлению прокрутки, будут игнорироваться. Активизировать такой режим мы можем с помощью атрибута стиля `-ms-scroll-rails`.

```
-ms-scroll-rails: none|railed
```

Значение `none` разрешает свободную прокрутку (поведение по умолчанию), а значение `railed` задает прокрутку по координатным осям.

```
#divZoom { -ms-scroll-rails: railed; }
```

Создание графического фона

Помимо вывода графического изображения в качестве полноправного элемента интерфейса, платформа Metro позволяет нам задать его в качестве фона другого элемента интерфейса или всего интерфейса приложения. Сейчас мы рассмотрим инструменты, которые позволяют это сделать.

Сначала нам потребуется указать файл с изображением, которое станет графическим фоном *(фоновое изображение)*. Для этого служит атрибут стиля `background-image`:

```
background-image: none|<ссылка на файл с изображением>
```

Ссылка на файл с изображением записывается с применением функции CSS `url`:

```
url(<собственно ссылка на файл>)
```

Значение `none` атрибута стиля `background-image` убирает графический фон (это поведение по умолчанию).

```
#divHeader { background-image: url("/images/background.png"); }
```

Задаем графический фон для блока `divHeader`.

Графический фон имеет приоритет перед обычным сплошным фоном, цвет которого мы учились задавать в *главе 8*. Другими словами, если мы зададим графический фон, он заменит собой сплошной фон.

Атрибут стиля `background-size` позволяет задать размер фонового изображения:

```
background-size: auto|contain|cover|<числовое значение>
```

Здесь мы можем использовать как произвольное числовое значение размера, заданное в любой из единиц измерения CSS, так и три предопределенных значения:

- `auto` — фоновое изображение будет иметь свои изначальные размеры, взятые из графического файла (это значение по умолчанию);

- `contain` — фоновое изображение будет масштабироваться до наибольшего из размеров элемента интерфейса, чтобы заполнить элемент целиком. Однако при этом часть изображения может оказаться за границами элемента;
- `cover` — фоновое изображение будет масштабироваться до наименьшего из размеров элемента интерфейса, чтобы целиком поместиться в границы элемента. Однако при этом часть элемента может оказаться не заполненной фоновым изображением.

В любом случае фоновое изображение будет масштабироваться с сохранением своих пропорций.

```
#divHeader { background-size: contain; }
```

С помощью атрибута стиля `background-position` можно указать позицию фонового изображения в элементе интерфейса:

```
background-position: <горизонтальная позиция> [<вертикальная позиция>]
```

Горизонтальная позиция фонового изображения задается в следующем формате:

```
<числовое значение>|left|center|right
```

Здесь можно указать позицию как в виде числа (с использованием любой единицы измерения CSS), так и в виде одного из предопределенных значений:

- `left` — фоновое изображение прижимается к левому краю элемента интерфейса (это обычное поведение);
- `center` — располагается по центру;
- `right` — прижимается к правому краю.

Формат задания *вертикальной позиции* фонового изображения таков:

```
<числовое значение>|top|center|bottom
```

Здесь также поддерживаются и числовые значения, и предопределенные:

- `top` — фоновое изображение прижимается к верхнему краю элемента интерфейса (это обычное поведение);
- `center` — располагается по центру;
- `bottom` — прижимается к нижнему краю.

Если для какого-либо элемента указана только позиция фонового изображения по горизонтали, его вертикальная позиция принимается равной `center`.

Значение по умолчанию атрибута стиля `background-position` — `0% 0%`, т. е. фоновое изображение будет располагаться в левом верхнем углу элемента интерфейса, для которого оно задано.

```
#divHeader (
  background-size: auto;
  background-position: center top;
}
```

Если фоновое изображение меньше, чем элемент интерфейса, для которого оно задано, платформа Metro будет повторять это изображение, пока не "замостит" им весь элемент. Параметры этого повторения задает атрибут стиля `background-repeat`:

```
background-repeat: no-repeat|repeat|repeat-x|repeat-y
```

Здесь доступны четыре значения:

- `no-repeat` — фоновое изображение не будет повторяться никогда; в этом случае часть фона элемента интерфейса останется незаполненной им;
- `repeat` — фоновое изображение будет повторяться по горизонтали и вертикали (обычное поведение);
- `repeat-x` — фоновое изображение будет повторяться только по горизонтали;
- `repeat-y` — фоновое изображение будет повторяться только по вертикали.

```
#divHeader ( background-repeat: no-repeat; }
```

Когда пользователь прокручивает содержимое элемента, вместе с ним будет прокручиваться и графический фон. Однако мы можем зафиксировать фон на месте, для чего воспользуемся атрибутом стиля `background-attachment`:

```
background-attachment: scroll|fixed
```

Значение `scroll` заставляет графический фон прокручиваться вместе с содержимым элемента (это поведение по умолчанию). Значение `fixed` фиксирует фон на месте, и он не будет прокручиваться.

```
#divContent { background-attachment: fixed; }
```

Осталось рассмотреть только "глобальный" атрибут стиля `background`, который позволит нам задать сразу все параметры фона:

```
background: [<цвет фона>] [<фоновое изображение>]
↳[<параметры повторения>] [<параметры фиксации>]
↳ [<параметры расположения>]
```

Если какой-то параметр не задан, он получит значение по умолчанию.

```
#divHeader {
  background: url("/images/background.png") no-repeat center top;
}
```

Что дальше?

В этой главе мы изучали средства платформы Metro для вывода графических изображений как в виде отдельного элемента интерфейса, так и в качестве графического фона. Заодно мы выяснили, как реализовать масштабирование с помощью жестов.

Следующая глава будет посвящена обработке мультимедийных данных — звука и видео. Наконец-то мы вплотную займемся нашим видеопроигрывателем!

ГЛАВА 12

Мультимедиа

В предыдущей главе мы выводили на экран графические изображения в виде отдельных элементов интерфейса и графического фона. А еще мы реализовывали масштабирование содержимого элементов интерфейса с помощью жестов и убедились, насколько просто это делается в Metro.

Эта глава будет посвящена выводу мультимедиа — звука и видео. И если со звуком мы еще не работали, то видеопроигрыватель стал первым Metro-приложением, созданным нами. Так что предмет отчасти нам знаком.

> ***НА ЗАМЕТКУ***
>
> Список форматов мультимедийных файлов и кодирования звука и видео, поддерживаемых Windows 8 в стандартной поставке, можно посмотреть на Web-странице **http://msdn.microsoft.com/en-us/library/windows/apps/hh767373.aspx**.

Базовые средства для воспроизведения мультимедиа

Если нам нужно всего лишь воспроизвести звуковой или видеофайл, мы используем базовые средства, предоставляемые языком HTML. Эти средства очень просты, в чем мы сейчас убедимся.

Воспроизведение звука

Для создания специального элемента интерфейса, предназначенного для воспроизведения звукового файла, — аудиопроигрывателя — используется парный тег `<audio>`. Ссылка на сам звуковой файл и параметры его воспроизведения указываются в атрибутах данного тега; содержимого же он никогда не имеет.

```
<audio src="/media/sound.mp3" controls></audio>
```

Создаем аудиопроигрыватель для воспроизведения звукового файла sound.mp3, хранящегося в папке media, причем аудиопроигрыватель будет содержать элементы для управления воспроизведением.

На экране такой аудиопроигрыватель выглядит так, как показано на рис. 12.1. Он содержит кнопку запуска и приостановки воспроизведения, регулятор, показывающий позицию воспроизведения, текстовые поля, выводящие текущую позицию воспроизведения и продолжительность звукового файла, и регулятор громкости с кнопкой отключения звука. В общем, все то, что мы привыкли видеть в обычных программах аудиопроигрывателей.

Рис. 12.1. Аудиопроигрыватель с элементами для управления воспроизведением звукового файла

Уже понятно, что ссылка на сам звуковой файл указывается в атрибуте `src` тега `<audio>`. Это единственный обязательный атрибут данного тега.

Если в теге `<audio>` присутствует атрибут без значения `controls`, аудиопроигрыватель выведет на экран элементы для управления воспроизведением файла. В этом случае аудиопроигрыватель будет представлять собой блочный элемент интерфейса.

Если же атрибут тега `controls` не указывать, аудиопроигрыватель вообще не будет присутствовать на экране.

Указание атрибута тега без значения `autoplay` запустит загрузку и воспроизведение звукового файла сразу же после формирования элемента аудиопроигрывателя. В противном случае файл сам загружаться и воспроизводиться не начнет; воспроизведение должен запустить либо пользователь, либо само приложение — в коде логики. (Программное управление мультимедийными элементами интерфейса мы рассмотрим потом.)

Теперь предположим, что мы не желаем запускать воспроизведение звукового файла автоматически, но все же хотим загрузить этот файл. Это может понадобиться, во-первых, чтобы в дальнейшем запустить его воспроизведение без задержки на загрузку, а во-вторых, чтобы сразу узнать его характеристики, в частности продолжительность. Можно ли это сделать?

Можно. Нам поможет необязательный атрибут тега `preload`. Он может принимать три значения:

- `none` — файл изначально загружаться не будет (поведение по умолчанию). Его загрузка начнется только после того, как пользователь (либо само приложение) запустит его воспроизведение. В результате мы не сможем сразу же узнать характеристики файла, а его воспроизведение начнется после задержки, необходимой для загрузки достаточной порции мультимедийных данных;
- `metadata` — будет загружена только заголовочная часть файла, хранящая сведения о нем. В результате мы сможем получить характеристики файла (как это сделать, будет рассказано потом), однако воспроизведение файла все равно может начаться с задержкой;
- `auto` — файл будет загружен целиком. В таком случае будут доступны сведения об этом файле, а его воспроизведение начнется без задержки.

Как уже должно быть понятно, если в теге <audio> присутствует атрибут autoplay, то атрибут preload можно не указывать.

```
<audio src="/media/sound.mp3" controls preload="auto"></audio>
```

Осталось рассмотреть только атрибут тега без значения loop, который вызывает зацикливание звукового файла. По умолчанию файл будет воспроизведен всего один раз.

```
<audio src="/media/sound.mp3" autoplay controls loop></audio>
```

Как поется в песне, эта музыка будет вечной...

Воспроизведение видео

Для формирования на экране элемента видеопроигрывателя применяется очень похожий тег — <video>. Он также парный и тоже никогда не имеет содержимого, а все данные, необходимые для успешного воспроизведения видео, задаются в его атрибутах.

```
<video src="/movie.mp4" autoplay controls></video>
```

Создаем видеопроигрыватель для воспроизведения файла movie.mp4. При этом на экране будут присутствовать элементы управления его воспроизведением (см. рис. 2.7), а загрузка и воспроизведение файла начнутся сразу после формирования на экране элемента видеопроигрывателя.

Отметим, что видеоролик будет масштабироваться таким образом, чтобы, с одной стороны, не нарушить своих пропорций, а с другой — заполнить максимум пространства видеопроигрывателя и не вылезти при этом за его пределы.

Тег <video> поддерживает все атрибуты, знакомые нам по тегу <audio>: src, autoplay, controls, preload и loop.

Помимо этого, тег <video> поддерживает следующие необязательные атрибуты:

- width — ширина видеопроигрывателя. Значение ширины указывается в виде целого числа в пикселах. Если ширина не задана, видеопроигрыватель будет иметь такую же ширину, как и постер (о постере — чуть позже); если постер отсутствует, он будет иметь ширину 300 пикселов;
- height — высота видеопроигрывателя также в виде целого числа в пикселах. Если высота не указана, видеопроигрыватель будет иметь такую же высоту, как и постер, а если постер отсутствует — 150 пикселов.

  ```
  <video src="/movie.mp4" controls width="640" height="480"></video>
  ```

- poster — ссылка на графический файл с *постером*, изображением, которое будет выводиться перед началом воспроизведения видеофайла. Если постер не указан, выводится первый кадр видеоролика.

  ```
  <video src="/movie.mp4" poster="/images/poster.jpg"></video>
  ```

Программное управление воспроизведением мультимедиа

Аудиопроигрыватель HTML представляется объектом `HTMLAudioElement`, а видеопроигрыватель — объектом `HTMLVideoElement`. Эти объекты поддерживают большое количество свойств, методов и событий, с которыми мы сейчас познакомимся.

Свойства

Начнем с рассмотрения свойств объекта `HTMLAudioElement`.

- `src` — хранит ссылку на мультимедийный файл в виде строки. Соответствует атрибуту тега `src`.
- `autoplay`. Значение `true` указывает платформе Metro начать загрузку и воспроизведение файла сразу после вывода на экран аудио- или видеопроигрывателя, значение `false` — не делать этого. Соответствует атрибуту тега `autoplay`.
- `controls`. Значение `true` указывает платформе Metro вывести на экран элементы для управления воспроизведением файла, значение `false` — не выводить их. Соответствует атрибуту тега `controls`.
- `loop`. Значение `true` вызывает зацикливание файла, а значение `false` — его однократное воспроизведение. Соответствует атрибуту тега `loop`.
- `currentTime` — хранит значение текущей позиции воспроизведения файла в виде числа с плавающей точкой в секундах. Присвоив этому свойству новое значение, мы можем переместить позицию воспроизведения в другое место ролика (выполнить его прокрутку).
- `volume` — хранит значение текущей громкости в виде числа с плавающей точкой от 0 (звук совсем не слышен) до 1 (максимальная громкость; значение по умолчанию). С помощью этого свойства мы можем управлять громкостью воспроизведения звука.
- `muted`. Значение `true` отключает звук, значение `false` — включает (поведение по умолчанию). (Разумеется, для аудироликов отключать звук смысла нет, но для видеороликов — есть.)
- `duration` — возвращает продолжительность ролика в виде числа с плавающей точкой в секундах. Если продолжительность ролика еще не удалось получить, возвращается значение `NaN`. Для потоковых медиаресурсов возвращается значение `Infinity`.
- `seeking` — возвращает `true`, если в данный момент пользователь изменяет текущую позицию воспроизведения ролика (выполняет его прокрутку), и `false` во всех остальных случаях.
- `paused` — возвращает `true`, если воспроизведение ролика приостановлено, и `false`, если оно продолжается.

- `ended` — возвращает `false`, если воспроизведение ролики еще продолжается, и `true`, если оно уже закончилось.

Объект `HTMLVideoElement` также поддерживает все эти свойства плюс еще несколько, перечисленных далее.

- `width` — хранит ширину видеопроигрывателя в виде целого числа в пикселах. Соответствует атрибуту тега `width`.
- `height` — хранит высоту видеопроигрывателя в виде целого числа в пикселах. Соответствует атрибуту тега `height`.
- `poster` — хранит ссылку на файл с постером в виде строки. Соответствует атрибуту тега `poster`.
- `videoWidth` — возвращает ширину изображения самого видеоролика, хранящуюся в видеофайле, в виде целого числа в пикселах. Если получить значение ширины невозможно, возвращается 0.
- `videoHeight` — возвращает высоту изображения самого видеоролика, хранящуюся в видеофайле, в виде целого числа в пикселах. Если получить значение высоты невозможно, возвращается 0.

Как видим, здесь есть свойства, соответствующие одноименным атрибутам тега, свойства, позволяющие управлять воспроизведением, и чисто информационные свойства, что предоставляют различные сведения о загруженном звуковом или видеофайле и состоянии элемента-проигрывателя.

```
var vidMain = document.getElementById("vidMain");
var iDuration = vidMain.duration;
var iWidth = vidMain.videoWidth;
var iHeight = vidMain.videoHeight;
vidMain.width = iWidth;
vidMain.height = iHeight;
vidMain.volume = 0.5;
```

Задаем для видеопроигрывателя размеры, соответствующие размерам изображения видеоролика, и половинное значение громкости.

Методы

Объекты `HTMLAudioElement` и `HTMLVideoElement` поддерживают два интересных для нас метода. Оба этих метода не принимают параметров и не возвращают результата.

- `play` — начинает или возобновляет, если было приостановлено, воспроизведение ролика. Если для атрибута тега `preload` было задано значение `none`, вызов этого метода также инициирует загрузку файла.
- `pause` — приостанавливает воспроизведение ролика.

```
vidMain.pause();
```

События

Объекты `HTMLAudioElement` и `HTMLVideoElement` поддерживают довольно много событий, которые могут быть для нас интересны. Мы рассмотрим их по группам.

К первой группе относятся события, возникающие в процессе загрузки звукового или видеофайла. Они возникают строго в том порядке, в котором перечислены далее.

- `loadstart` — однократно возникает перед началом загрузки звукового или видеофайла.
- `durationchange` — однократно возникает после того, как платформа Metro получит значение продолжительности загружаемого ролика.
- `loadedmetadata` — однократно возникает после того, как платформа Metro получит значения ширины и высоты загружаемого ролика.
- `loadeddata` — однократно возникает, когда объем загруженных мультимедийных данных становится достаточным для того, чтобы, по крайней мере, запустить воспроизведение ролика. Однако это еще не гарантирует того, что ролик начнет воспроизводиться без приостановок на подгрузку данных.
- `canplay` — возникает всякий раз, когда объем загруженных мультимедийных данных становится достаточным для того, чтобы успешно начать или возобновить воспроизведение ролика. Но, опять же, это еще не гарантирует того, что ролик впоследствии будет воспроизводиться без приостановок на подгрузку данных.
- `canplaythrough` — возникает всякий раз, когда мультимедийные данные начинают загружаться со скоростью, достаточной для дальнейшего воспроизведения ролика без приостановок на их подгрузку.
- `progress` — периодически возникает в процессе загрузки остальной части файла.
- `load` — возникает после завершения загрузки файла.
- `error` — возникает, если файл так и не удалось загрузить либо если данный файл имеет неподдерживаемый формат.

Как видим, эти события позволяют довольно точно отследить все моменты, связанные с загрузкой мультимедийного файла.

Следующие события возникают в процессе воспроизведения ролика.

- `playing` — однократно возникает сразу после начала воспроизведения ролика. Воспроизведение может быть запущено либо самим пользователем, либо программно, вызовом метода `play`.
- `timeupdate` — возникает всякий раз, когда текущая позиция воспроизведения ролика изменяется.
- `volumechanged` — возникает при изменении уровня громкости, а также отключении и включении звука.
- `pause` — возникает при приостановке воспроизведения ролика либо пользователем, либо программно, вызовом метода `pause`.

- `seeking` — периодически возникает в процессе перемещения пользователем регулятора, указывающего текущую позицию воспроизведения ролика (прокрутке ролика). Этот регулятор входит в состав стандартных элементов управления воспроизведением, выводимых платформой Metro.
- `seeked` — возникает после того, как пользователь переместит регулятор текущей позиции воспроизведения в новое положение.
- `waiting` — возникает, если пользователь переместит регулятор текущей позиции воспроизведения в позицию, в которой воспроизведение невозможно из-за того, что необходимые мультимедийные данные еще не загружены.
- `ended` — возникает после завершения воспроизведения ролика.

События, перечисленные далее, возникают в случае появления проблем при загрузке файла. (Причиной этих проблем могут быть неполадки в работе сети, долговременной памяти планшета и т. п.)

- `suspend` — возникает всякий раз, когда мультимедийные данные временно перестают загружаться.

  Если загрузка данных успешно возобновляется, то снова последовательно возникают события `canplay` и `canplaythrough`, за которыми следует серия событий `progress`.
- `stalled` — возникает через три секунды после загрузки последней порции мультимедийных данных. Как правило, возникновение этого события означает, что файл загрузить до конца не удастся.

```
var d = new Date(0, 0, 0, 0, 0, 0, 0);
var divPosition = document.getElementById("divPosition");
var iPos, s;
vidMain.addEventListener("timeupdate", function() {
  iPos = vidMain.currentTime;
  d.setHours(0, 0, iPos);
  s = d.getHours() + ":" + d.getMinutes() + ":" + d.getSeconds();
  divPosition.textContent = s;
});
```

Здесь мы привязываем к событию `timeupdate` видеопроигрывателя `vidMain` обработчик. Этот обработчик будет выводить в специально созданном блоке `divPosition` текущую позицию воспроизведения фильма в привычном нам формате времени *<часы>*:*<минуты>*:*<секунды>*.

Сначала мы создаем "нулевое" значение даты и времени (экземпляр объекта `Date`): нулевой год, нулевой месяц, нулевое число, ноль часов, ноль минут и ноль секунд. Это значение понадобится нам для преобразования значения текущей позиции воспроизведения, выраженного в секундах, в обычный формат времени, в котором мы и будем потом выводить это значение на экран.

В теле обработчика события `timeupdate` мы получаем текущую позицию воспроизведения. Выполняется это обращением к свойству `currentTime` нашего видеопроигрывателя.

Но как преобразовать это значение в обычный формат времени? Очень просто!

Объект `Date` поддерживает метод `setHours`, который позволяет задать для значения даты и времени новое время.

```
<значение даты и времени>.setHours(<часы>[, <минуты>[, <секунды>
↳[, <миллисекунды>]]])
```

В качестве параметров этого метода указываются количества часов, минут, секунд и миллисекунд, которые и составят новое время; все эти параметры задаются в виде целых чисел. Причем обязательным является только количество часов; остальные параметры могут быть опущены. Результата метод `setHours` не возвращает.

Данный метод имеет интересную и полезную особенность. Если мы зададим в качестве его параметра слишком большое количество, скажем, минут, он корректно преобразует их в часы. Так, если мы укажем 96 минут, то "на выходе" получим значение времени в 1 час и 36 минут. То же самое будет происходить с "лишними" часами, секундами и миллисекундами, которые будут переведены, соответственно, в дни, минуты и секунды.

Следовательно, указав в качестве третьего параметра метода значение текущей позиции воспроизведения в секундах, а в качестве значений остальных параметров — нули, мы получим это же значение, но уже выраженное в привычном для нас формате времени — в часах, минутах и секундах.

Напоследок нам останется только преобразовать полученное таким образом значение времени в строковый вид и вывести его на экран. Здесь нам пригодятся методы `getHours`, `getMinutes` и `getSeconds` объекта `Date`, возвращающие, соответственно, количество часов, минут и секунд, составляющих данное значение времени.

И еще один момент. Для хранения промежуточных результатов вычисления в теге функции-обработчика мы используем не локальные переменные, а глобальные, объявленные вне тела этой функции. Тому есть две причины. Во-первых, объявление любой переменной отнимает определенное время. Во-вторых, после завершения выполнения тела функции локальные переменные должны быть удалены из памяти, и их удаление также отнимает время. В результате производительность нашего приложения снизится, что будет особенно заметно на маломощных компьютерах.

Поддержка видеофайлов с несколькими звуковыми дорожками

Практически все современные форматы видеофайлов позволяют хранить в одном файле сразу несколько дорожек звука. Обычно эта возможность используется для распространения фильмов со звуковым сопровождением на нескольких языках; каждая звуковая дорожка содержит перевод фильма на один из заявленных языков.

Платформа Metro предлагает встроенные средства, позволяющие выбрать звуковую дорожку, которая будет воспроизводиться при просмотре фильма. Причем можно

выбрать как произвольную дорожку, так и найти дорожку со звуковым сопровождением на определенном языке и указать именно ее.

Объект `HTMLVideoElement` поддерживает свойство `audioTracks`. Оно возвращает экземпляр объекта-коллекции `AudioTracks`, представляющий набор звуковых дорожек, хранящихся в загруженном видеофайле. (О коллекциях рассказывалось в *главе 5*.)

```
var oATs = vidMain.audioTracks;
```

Объект `AudioTracks` поддерживает свойство `length`, характерное для всех коллекций. Оно возвращает количество элементов в коллекции.

Свойство `selectedTrack` хранит номер звуковой дорожки, которая воспроизводится в данный момент. Не забываем, что нумерация элементов коллекции начинается с нуля.

```
oATs.selectedTrack = oATs.length – 1;
```

Выбираем для воспроизведения последнюю звуковую дорожку файла.

Метод `language` возвращает строку с обозначением языка, на котором выполнено звуковое сопровождение фильма, что хранится на указанной дорожке. Он принимает единственный параметр — номер звуковой дорожки, для которой нужно узнать язык.

```
var sLanguage = oATs.language(1);
```

Получаем обозначение языка звукового сопровождения, что хранится на второй дорожке. (Не забываем, что нумерация элементов коллекций начинается с нуля.)

> ***Внимание!***
>
> Полный список обозначений языков можно посмотреть на Web-странице **http://msdn.microsoft.com/en-us/library/ms533052(VS.85).aspx**.

```
if (oATs.length > 1) {
  for (var i = 0; i < oATs.length; i++) {
    if (oATs.language(i) == "ru") {
      oATs.selectedTrack = i;
      break;
    }
  }
}
```

Перебираем все звуковые дорожки и выбираем ту, что хранит звуковое сопровождение на русском языке.

Пример: усовершенствованный видеопроигрыватель

В качестве практического занятия давайте серьезно займемся видеопроигрывателем `VideoPlayer`. Сделаем для него собственные элементы управления воспроизведением (кнопки открытия файла, запуска и приостановки воспроизведения, переключа-

тель наличия звука, регулятор громкости и текущей позиции воспроизведения) вместо стандартных. И заодно реализуем вывод текущей позиции воспроизведения и общей продолжительности ролика.

Откроем в Visual Studio проект `VideoPlayer`. Переключимся на файл default.html и заменим созданный ранее в теге `<body>` код следующим:

```
<div id="divUpper">
  <div id="divProgress">
    <input type="range" id="sldProgress" min="0" step="1" value="0"
    disabled />
  </div>
  <div id="divTiming"></div>
</div>
<div id="divMiddle" data-win-control="WinJS.UI.ViewBox">
  <video id="vidMain" preload="auto"></video>
</div>
<div id="divLower">
  <div id="divButtons">
    <input type="button" id="btnPlay" value="Пуск" disabled />
    <input type="button" id="btnPause" value="Пауза" disabled />
    <input type="button" id="btnOpen" value="Открыть" />
  </div>
  <div id="divMute" data-win-control="WinJS.UI.ToggleSwitch"
  data-win-options="{labelOn: '', labelOff: '', checked: true}"></div>
  <div id="divVolume">
    <input type="range" id="sldVolume" min="1" max="100" step="1"
    value="50" />
  </div>
</div>
```

Интерфейс новой версии приложения будет состоять из трех блоков. Эти блоки мы разместим на экране с применением сеточной разметки.

- Верхний блок `divUpper` будет содержать регулятор текущей позиции воспроизведения и текстовое поле, показывающее текущую позицию воспроизведения и продолжительность ролика в обычном формате времени.
- Средний, самый большой, блок `divMiddle` мы превратим в панель вывода Metro. Он включит в себя собственно видеопроигрыватель.
- Нижний блок `divLower` объединит кнопки управления воспроизведением, переключатель звука и регулятор громкости.

Блок `divUpper` получит в качестве содержимого два блока, которые мы также разместим с применением сеточной разметки. Левый блок `divProgress` включит регулятор текущей позиции воспроизведения `sldProgress`. Правый блок `divTiming` будет содержать текст, показывающий текущую позицию воспроизведения и продолжительность ролика.

А блок `divLower` будет содержать три блока. Левый блок `divButtons` объединит в себе все кнопки для управления воспроизведением: **Пуск** (`btnPlay`), **Пауза**

(btnPause) и **Открыть** (btnOpen). Средний блок divMute мы превратим в переключатель Metro — он будет включать и отключать звуковое сопровождение. А правый блок divVolume вместит в себя регулятор громкости sldVolume.

Регулятор sldProgress и кнопки btnPlay и btnStop мы изначально сделаем недоступными, поскольку сразу после запуска приложения они пользователю не понадобятся — воспроизводить-то пока что нечего.

Поскольку регуляторы HTML позволяют задавать только целочисленные значения, для регулятора sldVolume мы установили шкалу от 0 до 100. Впоследствии, в логике приложения, мы будем делить заданное с помощью данного регулятора значение на 100 и частное от этого деления указывать видеопроигрывателю. Заодно зададим для регулятора изначальное значение, равное 50, т. е. половину от максимальной громкости.

Переключатель Metro, в который мы превратим блок divMute, сделаем изначально включенным, поскольку звуковое сопровождение по умолчанию должно воспроизводиться.

Еще мы укажем, чтобы видеопроигрыватель начинал загрузку видеофайла сразу после того, как пользователь его откроет (значение auto для атрибута preload тега <video>). Это позволит, во-первых, сразу после открытия файла получить его продолжительность, а во-вторых, начать воспроизведение без задержки.

Теперь зададим оформление. Откроем файл default.css и удалим созданные нами ранее стили divButtons, divVideo, divInfo и .hidden-element. Вместо них создадим несколько новых стилей, описанных далее.

```
body {
  display: -ms-grid;
  -ms-grid-columns: 1fr;
  -ms-grid-rows: auto 1fr auto;
}
```

Задаем для всего интерфейса приложения сеточную разметку из одного столбца и трех строк, из которых первая и третья получат такую высоту, чтобы вместить свое содержимое, а вторая займет все оставшееся пространство.

```
#divUpper {
  display: -ms-grid;
  -ms-grid-columns: 1fr 150px;
  -ms-grid-rows: 1fr;
}
```

Так как мы не указали для блока divUpper строку и столбец сеточной разметки, где он должен находиться, он поместится в ячейке, образованной пересечением первой строки и первого столбца (самая верхняя ячейка).

Для блока divUpper мы также задаем сеточную разметку. Она будет включать одну строку и два столбца: правый получит ширину в 150 пикселов, а левый займет все оставшееся пространство.

Для блока `divProgress` мы не будем создавать стиль. Это излишне, т. к. он все равно поместится в ячейке сетки разметки, образованной пересечением первой строки и первого столбца, — в левой части блока `divUpper`.

```
#divTiming {
  -ms-grid-column: 2;
  -ms-grid-column-align: end;
}
```

Помещаем блок `divTiming` во второй столбец сетки разметки, т. е. в правую часть блока `divUpper`. Заодно укажем для него выравнивание содержимого по правому краю.

```
#divMiddle { -ms-grid-row: 2; }
```

Помещаем блок `divMiddle` в ячейку, образованную пересечением второй строки и первого столбца сетки разметки (в центр экрана).

```
#divLower {
  -ms-grid-row: 3;
  display: -ms-grid;
  -ms-grid-columns: 1fr auto auto;
  -ms-grid-rows: 1fr;
}
```

А блок `divLower` мы поместим в ячейку, образованную пересечением третьей строки и первого столбца сетки разметки (в нижнюю часть экрана).

Помимо этого, зададим для содержимого блока `divLower` сеточную разметку из одной строки и трех столбцов. Второй и третий столбцы будут иметь такую ширину, чтобы вместить свое содержимое, а первый займет все остальное пространство родителя.

```
#divButtons {
  -ms-grid-column-align: start;
  padding-left: 10px;
}
```

Блок `divButtons` сам займет ячейку сетки разметки, образованную пересечением первой строки и первого столбца. Нам следует только задать для него выравнивание содержимого по левому краю и внутренний отступ слева, равный 10 пикселам; так мы отодвинем содержимое данного блока — кнопки — от левого края экрана.

```
#divMute { -ms-grid-column: 2; }
```

Блок `divMute` мы поместим в ячейку, находящуюся на пересечении первой строки и второго столбца.

```
#divVolume { -ms-grid-column: 3; }
```

А блок `divVolume` займет ячейку, что находится на пересечении первой строки и третьего столбца.

```
#divProgress, #divTiming, #divButtons, #divMute, #divVolume {
  -ms-grid-row-align: center;
}
```

Для блоков `divProgress`, `divTiming`, `divButtons`, `divMute` и `divVolume` указываем вертикальное выравнивание по центру. Такой прием позволит несколько уменьшить высоту этих блоков и, следовательно, освободить еще немного места под видеопроигрыватель.

```
#sldProgress, #sldVolume { padding: 10px; }
```

Для обоих регуляторов задаем внутренние отступы со всех сторон, равные 10 пикселам.

```
#sldProgress { width: calc(100% – 20px); }
```

А для регулятора `sldProgress` также указываем, чтобы он занял все пространство родителя по ширине за вычетом значений отступов слева и справа.

Оформление готово. Займемся теперь самым сложным — логикой. Переключимся на файл default.js и сразу же удалим функции `vidMainPlaying()` и `vidMainPause()`. Они нам более не понадобятся, а хороший программист никогда не будет засорять логику ненужным кодом.

Исправим код, объявляющий необходимые переменные, чтобы он выглядел так:

```
var btnPlay, btnPause, btnOpen, vidMain, sldProgress, divTiming,
swtMute, sldVolume, sCurrent, sDuration;
var d = new Date(0, 0, 0, 0, 0, 0, 0);
```

После этого займемся обработчиком события `DOMContentLoaded`. Его код будет таким:

```
document.addEventListener("DOMContentLoaded", function () {
  WinJS.UI.processAll();

  btnPlay = document.getElementById("btnPlay");
  btnPause = document.getElementById("btnPause");
  btnOpen = document.getElementById("btnOpen");
  vidMain = document.getElementById("vidMain");
  sldProgress = document.getElementById("sldProgress");
  sldVolume = document.getElementById("sldVolume");
  divTiming = document.getElementById("divTiming");
  swtMute = document.getElementById("divMute").winControl;

  btnPlay.addEventListener("click", btnPlayClick);
  btnPause.addEventListener("click", btnPauseClick);
  btnOpen.addEventListener("click", btnOpenClick);
  vidMain.addEventListener("canplay", vidMainCanPlay);
  vidMain.addEventListener("playing", vidMainPlaying);
  vidMain.addEventListener("pause", vidMainPause);
  vidMain.addEventListener("timeupdate", vidMainTimeUpdate);
  sldProgress.addEventListener("change", sldProgressChange);
```

```
  sldVolume.addEventListener("change", sldVolumeChange);
  swtMute.addEventListener("change", swtMuteChange);

  vidMain.volume = 0.5;
});
```

Здесь мы получаем доступ ко всем нужным элементам интерфейса и привязываем к событиям обработчики. Напоследок не забудем установить для видеопроигрывателя значение громкости, равное 0,5; оно соответствует значению, ранее указанному нами в регуляторе `sldVolume`.

Созданный ранее обработчик события `click` кнопки `btnOpen`, выполняющий открытие видеофайла, мы менять не будем.

Начнем с функции `vidMainCanPlay()` — обработчика события `canplay` видеопроигрывателя `vidMain`.

```
function vidMainCanPlay() {
  btnPlay.disabled = false;
  btnPause.disabled = true;
  sldProgress.disabled = false;
  sldProgress.value = 0;
  sldProgress.max = vidMain.duration;
  d.setHours(0, 0, vidMain.duration);
  sDuration = d.getHours() + ":" + d.getMinutes() + ":" + d.getSeconds();
  showTiming();
}
```

Делаем кнопку `btnPlay` и регулятор `sldProgress` доступными, а кнопку `btnPause` — недоступной, после чего устанавливаем регулятор `sldProgress` в нулевое положение и указываем для него максимальное значение, равное продолжительности фильма. Теперь пользователь сможет запустить фильм на воспроизведение.

Напоследок мы вычисляем значение продолжительности фильма описанным ранее способом, сохраняем это значение в объявленной ранее глобальной переменной `sDuration` и вызываем функцию `showTiming`, которая выведет на экран текущую позицию воспроизведения и продолжительность фильма.

Сразу же объявим функцию `showTiming()`, чтобы не забыть это сделать потом.

```
function showTiming() {
  d.setHours(0, 0, vidMain.currentTime);
  sCurrent = d.getHours() + ":" + d.getMinutes() + ":" + d.getSeconds();
  divTiming.textContent = sCurrent + " / " + sDuration;
}
```

Здесь все нам уже знакомо.

Раз уж мы взялись за видеопроигрыватель, давайте напишем остальные обработчики его событий — функции `vidMainPlaying()`, `vidMainPause()` и `vidMainTimeUpdate()`.

```
function vidMainPlaying() {
  btnPlay.disabled = true;
```

```
  btnPause.disabled = false;
}
function vidMainPause() {
  btnPlay.disabled = false;
  btnPause.disabled = true;
}
function vidMainTimeUpdate() {
  sldProgress.value = vidMain.currentTime;
  showTiming();
}
```

Первая функция делает кнопку `btnPlay` недоступной, а кнопку `btnPause` — доступной, чтобы пользователь смог поставить воспроизведение на паузу. Вторая функция, наоборот, делает кнопку `btnPlay` доступной, а кнопку `btnPause` — недоступной; таким образом, пользователь может возобновить воспроизведение фильма. Третья функция устанавливает регулятор `sldProgress` в положение, соответствующее текущей позиции воспроизведения фильма, и выводит текущую позицию на экран.

Настал черед кнопок. Объявление функций `btnPlayClick()` и `btnPauseClick()`, вызываемых при щелчках на кнопках `btnPlay` и `btnPause` соответственно, будет таким:

```
function btnPlayClick() {
  vidMain.play();
}
function btnPauseClick() {
  vidMain.pause();
}
```

Как говорится, комментарии излишни.

Теперь нашего внимания требуют регуляторы `sldProgress` и `sldVolume`. Напишем для них обработчики событий `change`.

```
function sldVolumeChange() {
  vidMain.volume = sldVolume.value / 100;
}
function sldProgressChange() {
  vidMain.currentTime = sldProgress.value;
}
```

Здесь тоже особо нечего пояснять. Стоит только отметить, что мы ранее установили для регулятора `sldVolume` шкалу от 0 до 100, поэтому полученное от него значение сначала делим на 100.

Напоследок напишем функцию `swtMuteChange`, которая будет вызываться при изменении состояния переключателя `swtMute`.

```
function swtMuteChange() {
  vidMain.muted = !swtMute.checked;
}
```

Здесь мы присваиваем свойству `muted` видеопроигрывателя инвертированное значение свойства `checked` переключателя `swtMute`. Дело в том, что у нас включенный переключатель означает наличие звука, а отключенный — его отсутствие.

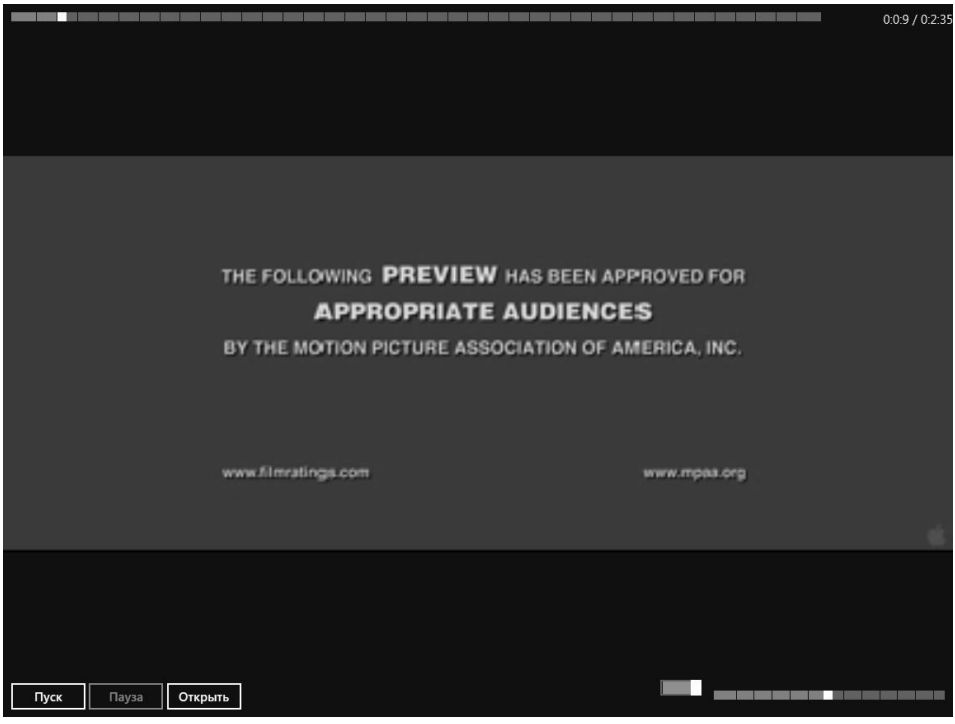


Рис. 12.2. Интерфейс видеопроигрывателя `VideoPlayer` после очередного усовершенствования

Все! Сохраним все измененные файлы и запустим приложение на выполнение. Интерфейс приложения должен выглядеть так, как показано на рис. 12.2.

Откроем какой-либо видеофайл и запустим его на воспроизведение. Попробуем поставить воспроизведение на паузу, возобновить его, изменить позицию воспроизведения и громкость, отключить и снова включить звук. Если мы все сделали правильно, эти функции должны работать.

Наш видеопроигрыватель все более обретает черты законченного приложения!

Что дальше?

В этой главе мы испытывали средства платформы Metro для работы с мультимедиа и нашли их весьма мощными. Заодно в очередной раз усовершенствовали свой видеопроигрыватель.

Следующая часть книги будет посвящена более сложным и развитым элементам интерфейса, предлагаемым Metro. И первая же глава этой части расскажет нам о фрагментах — независимых частях интерфейса приложения, хранящихся в отдельных файлах. Фрагменты помогут нам разбить интерфейс приложения на логические части и создать, таким образом, "многооконные" приложения.

ЧАСТЬ IV

Создание сложных элементов интерфейса

ГЛАВА 13

Фрагменты

В предыдущей главе мы занимались средствами платформы Metro, предназначенными для воспроизведения мультимедиа. И заодно в очередной раз усовершенствовали свое первое Metro-приложение — видеопроигрыватель.

В этой главе мы продолжим заниматься интерфейсом Metro-приложений. Мы начнем рассмотрение более сложных и развитых инструментов, предлагаемых данной платформой. И первыми из этих инструментов, что мы рассмотрим, станут фрагменты.

Введение во фрагменты

Интерфейс современных приложений очень сложен. Он включает в себя множество элементов управления, выполняющих различные задачи. И многие из этих элементов управления необходимы пользователю и вообще имеют смысл только в определенных случаях.

Если вывести на экран все элементы управления, имеющиеся в приложении, одновременно, экран окажется заполненным ими, а что-то, скорее всего, на нем просто не поместится. В любом случае, приложением невозможно будет пользоваться.

Чтобы избежать этого, на экран в данный момент времени выводится только часть интерфейса приложения — та, что необходима пользователю прямо сейчас. Остальная, бóльшая, часть интерфейса либо скрывается, либо не выводится вообще; последнее предпочтительнее, т. к. скрытые элементы управления все равно отнимают системные ресурсы, которых всегда не хватает.

В традиционных Windows-приложениях различные части интерфейса выводятся в отдельных окнах. Главное окно содержит основные элементы управления, необходимые все время, пока приложение работает; остальные части интерфейса располагаются в других окнах — вспомогательных, диалоговых, панелях инструментов, окнах-предупреждениях и пр. Эти окна выводятся на экран только тогда, когда в них возникает необходимость.

Но как быть нам, Metro-разработчикам? Ведь в Metro-приложениях окна вообще не предусмотрены. (Как говорилось еще в *главе 1*, окна занимают слишком много места на экране, а управлять ими с помощью жестов неудобно.)

Конечно, какие-то элементы интерфейса можно до поры до времени скрыть и вывести только тогда, когда они понадобятся пользователю. Но мы уже знаем, что даже скрытые элементы интерфейса, точнее представляющие их экземпляры объектов в структуре DOM, отнимают системные ресурсы, которые у планшетов еще более ограничены, чем у традиционных ПК. Следовательно, так можно поступать только с теми элементами, потребность в которых возникает относительно часто.

Поэтому здесь может быть полезнее другой подход — вынос частей интерфейса, которые должны присутствовать на экране только в определенные моменты времени, за пределы основного интерфейса и оформление их в виде фрагментов.

Фрагмент в терминологии Metro — это независимая часть интерфейса приложения, включающая в себя также его оформление и логику, что обеспечивает функционирование входящих в эту часть элементов интерфейса. Или, другими словами, часть приложения, выделенная в отдельную структурную единицу, выполняющая определенный набор действий и независимая как от самого приложения, так и от других фрагментов.

Фрагмент описывается тремя файлами. Первый файл — <*имя фрагмента*>.html — включает собственно описание интерфейса этого фрагмента. Второй файл — <*имя фрагмента*>.css — хранит его оформление. А третий файл — <*имя фрагмента*>.js — содержит описание логики, заставляющей этот фрагмент работать.

Преимущество фрагментов очевидно — части интерфейса, необходимые только время от времени, выносятся за пределы основного приложения, загружаются только по требованию и не занимают попусту системные ресурсы. Недостатки: потребность в дополнительном коде, собственно выполняющем вывод и удаление фрагментов, и специальных средствах для взаимодействия фрагментов с основным приложением и друг с другом. Впрочем, оба этих недостатка вполне преодолимы, в чем мы вскоре убедимся.

Создание фрагментов

Создать фрагмент очень просто. Сам Visual Studio сделает это за нас.

Прежде всего, отыщем в иерархическом списке панели **SOLUTION EXPLORER** (подробнее о ней говорилось в *главе 2*) пункт, представляющий проект нашего приложения, и выберем его. Щелкнем на этом пункте правой кнопкой мыши и выберем в подменю **Add** появившегося на экране контекстного меню пункт **New Item**. (Также можно выбрать пункт **Add New Item** меню **Project** или нажать комбинацию клавиш <Ctrl>+<Shift>+<A>.) На экране появится диалоговое окно **Add New Item** (рис. 13.1).

Сразу обратим внимание на средний список, где выбирается тип составной части, которую мы хотим добавить в проект: таблица стилей, файл логики, текстовый

файл и т. д. Поскольку нам нужно добавить новый фрагмент, мы выберем пункт **Page Control**.

В поле ввода **Name** укажем имя добавляемой составной части, фактически — имя файла, в которой она будет храниться.

Чтобы добавить в проект новую составную часть, следует нажать кнопку **Add**. Кнопка **Cancel** отменит ее добавление.

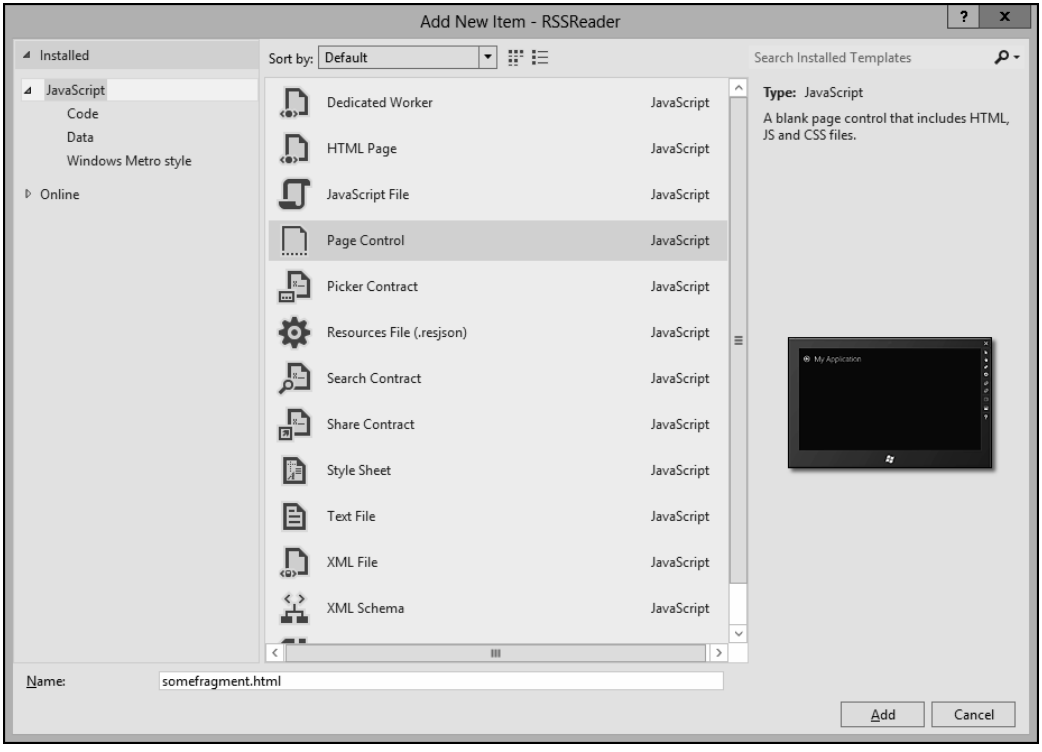


Рис. 13.1. Диалоговое окно **Add New Item**

Содержимое вновь созданного фрагмента

Как уже говорилось, фрагмент хранится в трех файлах, содержащих описание его интерфейса, оформление и логику. Давайте посмотрим, что именно находится в этих файлах.

Файл с описанием интерфейса <*имя фрагмента*>.html хранит такой же код, что изначально присутствует в файле default.html. Единственное исключение — содержимое тега `<body>`; в случае фрагмента оно довольно велико, но так же бесполезно.

То же самое касается и файла с оформлением <*имя фрагмента*>.css. Имеющиеся в нем стили мы так или иначе потом удалим.

Теперь откроем файл логики *<имя фрагмента>*.js. Здесь мы увидим вот что:

```
(function () {
  'use strict';
  . . .
})();
```

Имеет смысл только код, что представлен на этом примере. Остальной код, созданный Visual Studio (обозначен многоточием), мы можем удалить.

И еще. Хорошо видно, что код логики данного фрагмента находится в собственном анонимном пространстве имен. (О пространствах имен, в том числе анонимных, рассказывалось в *главе 5*.) Это значит, что объявленные в нем переменные и функции не будут "видимы" в коде логики основного приложения и других фрагментов. И наоборот, логика основного приложения и других фрагментов не "увидит" всего того, что мы объявим в логике этого фрагмента.

Организация фрагментов

Правила хорошего тона Metro-программирования требуют, чтобы все файлы, хранящие код фрагментов, находились в отдельной папке. Обычно эта папка носит имя fragments.

Visual Studio всегда помещает вновь созданные составные части прямо в папку проекта. (О папке проекта см. в *главе 2*.) Так что нам придется самим создать в проекте новую папку и переместить все файлы фрагмента в нее.

Создать новую папку в проекте несложно. Если мы хотим создать новую папку прямо в папке проекта, то щелкнем правой кнопкой мыши на пункте в списке панели **SOLUTION EXPLORER**, что соответствует проекту. Если мы собираемся создать папку в другой папке, то щелкнем правой кнопкой на пункте, что представляет эту папку. В любом случае на экране появится контекстное меню, в котором мы раскроем подменю **Add** и выберем в нем пункт **New Folder.**

Сразу после этого в списке появится новый пункт, представляющий вновь созданную папку. Вместо его имени будет присутствовать поле ввода; введем в него имя создаваемой папки и нажмем клавишу <Enter>.

Переместить файлы в папку также очень просто. Это можно сделать мышью, точно так же, как мы перемещаем файлы из папки в папку в окне Проводника.

И еще один момент. Если мы откроем файл *<имя фрагмента>*.html, где описывается интерфейс фрагмента, то увидим следующие ссылки на файлы с его оформлением и логикой:

```
<link href="somefragment.css" rel="stylesheet">
<script src="somefragment.js"></script>
```

Нам следует соответственно изменить ссылки на данные файлы. Так, если мы сохраним все файлы фрагментов в папке fragments, эти ссылки будут выглядеть следующим образом:

```
<link href="/fragments/somefragment.css" rel="stylesheet">
<script src="/fragments/somefragment.js"></script>
```

Так мы исключим вероятность того, что платформа Metro не "найдет" все эти файлы. (Вероятность, правда, небольшая, но лучше ее исключить полностью.)

Создание интерфейса и оформления фрагмента

Интерфейс и оформление фрагмента создается точно так же, как и у основного приложения. HTML-код, описывающий интерфейс, мы вписываем внутри тега `<body>`, а стили, задающие его оформление, определяем в таблице стилей.

Но здесь нужно иметь в виду несколько моментов, касающихся природы фрагментов и принципов их обработки платформой Metro.

- Элементы интерфейса, включенные в состав фрагмента, становятся частью интерфейса основного приложения и его структуры DOM. Из этого следует, что имена элементов интерфейса фрагмента не должны совпадать с именами элементов интерфейса основного приложения.
- Стили, определенные в файле оформления фрагмента, становятся частью общего оформления основного приложения. Поэтому нужно внимательно следить, чтобы одни стили не переопределили, целиком или частично, другие, иначе при загрузке фрагмента весь интерфейс может измениться.
- Также нужно проследить, чтобы и основное приложение, и все входящие в его состав фрагменты использовали одну и ту же таблицу стилей, задающую базовое оформление. Так, если приложение использует таблицу стилей ui-dark.css, то все фрагменты также должны использовать эту таблицу стилей. В противном случае все приложение при загрузке фрагментов начнет мигать, как испорченный светофор, что уж точно не понравится пользователям.

Создание логики фрагмента

А теперь поговорим о том, что касается логики фрагмента. Здесь нас ждет еще несколько важных моментов, которые ни в коем случае не следует упускать из виду.

Инициализация фрагмента

Код инициализации фрагмента, в основном, пишется по тем же правилам, что и код инициализации основного приложения (подробнее об этом говорилось в *главе 5*). За двумя отличиями, которые мы сейчас рассмотрим.

Во-первых, не существует никакого события, возникающего по окончании загрузки фрагмента. Следовательно, мы не сможем отследить этот момент в самом фрагменте. (Зато мы можем сделать это в коде, который загружает этот фрагмент, но разговор об этом пойдет позже.)

Поэтому код инициализации фрагмента оформляется как тело обычной функции — не обработчика события:

```
function initialization() {
  //Код инициализации фрагмента
}
```

Во-вторых, если мы используем в интерфейсе фрагмента какие-либо элементы управления Metro, то инициализация их вызовом метода `processAll` объекта `WinJS.UI` без параметров, как мы делали это ранее, скорее всего, не будет лучшим решением. (Подробнее об элементах управления Metro говорилось в *главе 7*.) Дело в том, что данный метод в случае вызова без параметров будет обрабатывать весь интерфейс приложения, а это займет слишком много времени.

Вместо этого мы можем вызвать метод `processAll`, указав ему в качестве параметра элемент интерфейса — родитель всех элементов управления Metro, что мы хотим инициализировать.

```
WinJS.UI.processAll(<блок-родитель для блоков, которые станут
↳элементами управления Metro>);
<div id="divCont">
  <div id="divDate" data-win-control="WinJS.UI.DatePicker"></div>
  <div id="divTime" data-win-control="WinJS.UI.TimePicker"></div>
</div>
. . .
function initialization() {
  var divCont = document.getElementById("divCont");
  WinJS.UI.processAll(divCont);
  . . .
}
```

Вызываем метод `processAll`, передав ему блок `divCont` — родителя обоих элементов управления Metro, что мы хотим инициализировать.

Обеспечение взаимодействия между фрагментом и основным приложением

В процессе работы основное приложение должно взаимодействовать с фрагментом: заносить изначальные данные во входящие в его состав элементы управления; получать данные, введенные в них пользователем; управлять фрагментом, вызывая объявленные в его логике функции; наконец, запустить его инициализацию после загрузки. И фрагмент также должен иметь возможность взаимодействовать с основным приложением, чтобы, например, получить от него какие-то данные.

Но, как мы уже знаем, логика основного приложения и логика фрагмента работают внутри различных пространств имен и, таким образом, изолированы друг от друга. Если бы у этих пространств имен были имена, мы бы могли обратиться из логики основного приложения к переменной или функции, объявленной в логике фрагмента, указав имя пространства имен этого фрагмента. Но, к сожалению, оба этих пространства имен анонимны.

Умный в гору не пойдет — умный гору обойдет. Мы тоже попробуем решить эту проблему не лобовым штурмом, а атакой с фланга.

- Создадим для фрагмента еще одно пространство имен — с именем. Код, который выполнит создание этого пространства имен, должен находиться в составе логики фрагмента.
- Создадим в новом пространстве имен набор свойств, что будут соответствовать всем переменным и функциям фрагмента, к которым мы хотим иметь доступ "извне" — из основного приложения и других фрагментов. Код, создающий все эти свойства и методы, также должен находиться в составе логики фрагмента.

Оба этих действия нам поможет выполнить метод `define` объекта `WinJS.Namespace`. (Этот объект содержит методы, предназначенные для создания пространств имен. Единственный его экземпляр создается платформой Metro и доступен через переменную `WinJS.Namespace`.)

```
WinJS.Namespace.define(<имя пространства имен>,
<состав свойств этого пространства имен>)
```

Этот метод принимает два параметра. Первым параметром передается имя создаваемого пространства имен в виде строки.

Второй параметр требует более обстоятельного разговора. Он должен представлять собой экземпляр объекта `Object` с набором свойств и методов. Значением каждого его свойства станет переменная или функция фрагмента, к которой требуется доступ "извне". Обычно для создания такого экземпляра объекта используется конфигуратор (подробнее о них — в *главе 4*).

Рассмотрим самый простой пример — создание доступной "извне" функции, которая выполнит инициализацию фрагмента. Объявим ее в логике фрагмента:

```
function initialization() { . . . }
```

Теперь там же вызовем метод `define` объекта `WinJS.Namespace`.

```
WinJS.Namespace.define("SomeFragment", {
  initialization: initialization
});
```

Здесь мы создали для нашего фрагмента пространство имен `SomeFragment` со свойством `initialization`, которому присвоили в качестве значения объявленную ранее функцию `initialization()`.

Отметим, что мы дали свойству созданного пространства имен то же имя, что и присвоенной ему функции. Это обычная практика в Metro-программировании.

Далее мы можем вызвать функцию инициализации фрагмента из логики основного приложения:

```
SomeFragment.initialization();
```

Но что делать, если нам потребуется получить доступ из логики фрагмента к переменным и функциям, объявленным в логике основного приложения? То же самое.

Объявляем в логике основного приложения все нужные переменные и функции:

```
var someVar;
function someFunc() { . . . }
```

Там же создаем пространство имен:

```
WinJS.Namespace.define("BaseApplication", {
  someVar: someVar,
  someFunc: someFunc
});
```

И получаем к ним доступ из логики фрагмента:

```
var i = BaseApplication.someVar;
BaseApplication.someFunc();
```

Как видим, мы можем давать доступ "извне" только к определенным переменным и функциям фрагмента; при этом все остальные переменные и функции будут доступны лишь в коде его логики. Это позволяет сделать логику фрагментов более защищенной от случайного изменения со стороны как основного приложения, так и других фрагментов.

Загрузка фрагмента

Что ж, наш фрагмент готов. Осталось только вставить его в основное приложение, или, если пользоваться терминологией Metro, выполнить его *загрузку*.

Сначала нам следует создать элемент интерфейса, который станет родителем для загружаемого фрагмента. Обычно таким элементом выступает блок. (Кстати, его будет удобно использовать в вызове метода `processAll` для инициализации всех элементов управления Metro, входящих в состав фрагмента.)

```
<div id="divBase"></div>
```

Создаем блок `divBase`, в котором и будет выводиться наш фрагмент.

Выполнить загрузку фрагмента позволит метод `render` объекта `WinJS.UI.Fragments`. (Этот объект содержит методы, предназначенные для управления фрагментами. Единственный его экземпляр создается платформой Metro и доступен из переменной `WinJS.UI.Fragments`.)

```
WinJS.UI.Fragments.render(<ссылка на файл фрагмента>,
<элемент, который станет родителем для загружаемого фрагмента>)
```

Первым параметром этому методу передается ссылка на HTML-файл фрагмента в виде строки. Вторым параметром указывается элемент — будущий родитель для загружаемого блока.

Метод `render` возвращает в качестве результата обязательство (экземпляр объекта `WinJS.Promise`; подробнее — в *главе 5*). С помощью метода `then` обязательства мы укажем функцию, которая будет выполнена после завершения загрузки фрагмента.

```
var divBase = document.getElementById("divBase");
WinJS.UI.Fragments.render("/fragments/somefragment.html", divBase).
↳then(function() {
  SomeFragment.initialize();
});
```

Здесь мы загружаем созданный нами ранее фрагмент, хранящийся в файле somefragment.html, и выводим его в созданный ранее блок `divBase`. Как только фрагмент будет загружен и выведен на экран, мы вызываем его функцию инициализации.

Теперь фрагмент готов к работе.

Удаление фрагмента

Удалить ненужный более фрагмент можно с применением любого способа, рассмотренного нами в *главе 10*.

- Мы можем присвоить свойству `innerHTML` элемента интерфейса — родителя этого фрагмента пустую строку:

  ```
  divBase.innerHTML = "";
  ```

- Мы можем вызвать метод `empty` объекта `WinJS.Utilities`, передав ему в качестве параметра элемент — родитель фрагмента:

  ```
  WinJS.Utilities.empty(divBase);
  ```

В любом случае загруженный ранее фрагмент будет удален и с экрана, и из памяти компьютера. Его оформление и логика также будут удалены из памяти.

Пример: прототип "многооконного" приложения для чтения каналов RSS

Настала пора практики. Давайте превратим прототип приложения для чтения каналов RSS в "многооконный". Мы реализуем вывод списка новостей и содержимого выбранной пользователем новости в отдельных фрагментах.

Откроем в Visual Studio проект `RSSReader`. Откроем файл default.html, удалим из тега `<body>` написанный ранее код и введем вот такой:

```
<div id="divHeader">
  <h1>Канал новостей</h1>
</div>
<div id="divBase"></div>
```

Блок `divHeader`, как и ранее, будет служить для вывода названия канала RSS. А в блоке `divBase` мы будем выводить фрагменты.

Откроем теперь файл default.css, удалим созданные ранее стили и зададим следующие:

```
body {
  display: -ms-grid;
  -ms-grid-columns: 1fr;
  -ms-grid-rows: auto 1fr;
}
#divBase { -ms-grid-row: 2; }
```

Комментировать здесь нечего — сеточная разметка нам уже привычна.

Самые значительные изменения претерпит логика. Переключимся на файл default.js и удалим весь созданный нами ранее код, за исключением того, что объявляет и заполняет элементами массив `arrNews`.

После этого объявим нужные нам переменные:

```
var divBase, selectedNew;
```

Переменная `selectedNew` будет хранить номер выбранной пользователем новости. Фрагмент, выводящий список новостей, занесет в нее этот номер, а фрагмент, выводящий содержимое выбранной новости, получит его оттуда.

Обработчик события `DOMContentLoaded` будет очень прост:

```
document.addEventListener("DOMContentLoaded", function() {
  divBase = document.getElementById("divBase");
  loadList();
});
```

Функция `loadList()` загрузит фрагмент, выводящий список новостей. Таким образом, этот список появится на экране сразу после запуска приложения.

Сразу же объявим функцию `loadList()`:

```
function loadList() {
  WinJS.Utilities.empty(divBase);
  WinJS.UI.Fragments.render("/fragments/list.html", divBase).
    ↳then(function() {
    List.fillList();
  });
}
```

Здесь нам также все знакомо. Следует только отметить четыре момента:

- фрагмент, выводящий список новостей, будет храниться в файле list.html, вложенном в папку fragments;
- для этого фрагмента мы потом создадим пространство имен `List`;
- инициализацию этого фрагмента будет выполнять функция `fillList()`; поскольку это единственная задача, которую выполняет функция инициализации, мы можем назвать ее именно так;
- самый важный момент — перед собственно выводом загруженного фрагмента мы очищаем блок `divBase`. Это нужно на тот случай, если пользователь после

просмотра содержимого выбранной новости пожелает вернуться к списку новостей.

Функция `loadContent()` будет выводить фрагмент, отображающий содержимое выбранной пользователем новости:

```
function loadContent() {
  WinJS.Utilities.empty(divBase);
  WinJS.UI.Fragments.render("/fragments/content.html", divBase).
  ↳then(function() {
    Content.showContent();
  });
}
```

Этот фрагмент будет храниться в файле content.html, вложенном в папку fragments. Мы создадим для него пространство имен `Content`, инициализировать его будет функция `showContent()`, а перед его выводом мы также очистим блок `divBase`.

Осталось создать для основного приложения пространство имен:

```
WinJS.Namespace.define("BaseApplication", {
  arrNews: arrNews,
  selectedNew: selectedNew,
  loadList: loadList,
  loadContent: loadContent
});
```

Оно будет называться `BaseApplication` и предоставит доступ "извне" к массиву новостей, переменной, где хранится номер выбранной пользователем новости, и обеим функциям, загружающим фрагменты.

Теперь можно приступать к созданию фрагментов.

Начнем с фрагмента, выводящего список новостей. Дадим ему имя list.html, как условились ранее. Создадим в папке проекта папку fragments и переместим все три файла, где хранится вновь созданный фрагмент, в нее.

Откроем файл list.html, удалим весь код, что уже присутствует в теге `<body>`, и введем туда вот что:

```
<div id="divList"></div>
```

Блок `divList` будет использоваться для вывода списка новостей.

Далее откроем файл list.css, также удалим все его содержимое и создадим два следующих стиля:

```
#divList {
  display: -ms-box;
  -ms-box-lines: multiple;
  -ms-box-align: before;
}
#divList > div { width: 30%; }
```

Они нам уже знакомы по предыдущей версии этого приложения.

На очереди — логика фрагмента. Откроем файл list.js и удалим весь ненужный код. Сразу же объявим функцию `fillList()`.

```
function fillList() {
  var oDiv, oPDate, oPHeader;
  var divList = document.getElementById("divList");
  WinJS.Utilities.empty(divList);
  for (var i = 0; i < BaseApplication.arrNews.length; i++) {
    oDiv = document.createElement("div");
    oPDate = document.createElement("p");
    oPDate.textContent = BaseApplication.arrNews[i][0];
    oDiv.appendChild(oPDate);
    oPHeader = document.createElement("p");
    oPHeader.textContent = BaseApplication.arrNews[i][1];
    oDiv.appendChild(oPHeader);
    oDiv.id = "div" + i;
    oDiv.addEventListener("click", divItemClick);
    divList.appendChild(oDiv);
  }
}
```

И здесь нам все знакомо по предыдущей версии приложения.

Объявим функцию — обработчик события `divItemClick`.

```
function divItemClick(evt) {
  var sID = evt.currentTarget.id;
  var sIndex = sID.substr(3);
  BaseApplication.selectedNew = parseInt(sIndex);
  BaseApplication.loadContent();
}
```

Она занесет номер выбранной новости в переменную `selectedNew` основного приложения и сразу же вызовет функцию `loadContent()`, объявленную там же.

И создадим для данного фрагмента пространство имен `List`:

```
WinJS.Namespace.define("List", {
  fillList: fillList
});
```

Осталось создать фрагмент, который выведет содержимое выбранной новости. Дадим ему имя content.html и также поместим в папку fragments.

Откроем файл content.html и заменим весь код в теге `<body>` следующим:

```
<div id="divContent">
  <div id="divText"></div>
  <div id="divButtons">
    <input type="button" id="btnBack" value="Назад" />
  </div>
</div>
```

Здесь мы создали блок `divContent`, а в нем — еще два блока: `divText`, в котором собственно и будет выводиться текст новости, и `divButtons`, с кнопкой **Назад**. Нажатие этой кнопки снова выведет на экран список новостей.

В файле content.css определим следующие стили:

```
#divContent {
  width: 100%;
  height: 100%;
  display: -ms-grid;
  -ms-grid-columns: 1fr;
  -ms-grid-rows: 1fr 32px;
}
#divText {
  column-count: 2;
  overflow-x: auto;
  overflow-y: hidden;
}
#divButtons {
  -ms-grid-row: 2;
  -ms-grid-column-align: end;
}
```

Здесь также нечего комментировать.

Логика этого фрагмента будет совсем простой. Вот что мы введем в файле content.js:

```
function showContent() {
  var divText = document.getElementById("divText");
  var btnBack = document.getElementById("btnBack");
  divText.innerHTML =
  BaseApplication.arrNews[BaseApplication.selectedNew][2];
  btnBack.addEventListener("click", function () {
    BaseApplication.loadList();
  });
}
WinJS.Namespace.define("Content", {
  showContent: showContent
});
```

Функция инициализации `showContent()` выведет в блоке `divText` содержимое выбранной новости и привяжет к кнопке **Назад** обработчик, который при щелчке на этой кнопке снова загрузит фрагмент со списком новостей. Напоследок мы создаем для этого фрагмента пространство имен `Content`.

Сохраним все измененные файлы и проверим приложение в действии.

Что дальше?

В этой главе мы занимались фрагментами Metro — независимыми частями интерфейса приложения, хранящимися в отдельных файлах. Фрагменты помогут нам в создании приложений со сложным интерфейсом и будут неплохой альтернативой традиционным окнам.

Следующая глава будет посвящена спискам Metro — еще не рассмотренному нами типу элементов управления, поддерживаемых этой платформой. По сравнению со знакомыми нам по *главе 6* списками HTML, они предлагают значительно более богатые возможности по выводу и форматированию данных, взятых из любого источника. Более того, списки Metro сделают за нас бóльшую часть работы по выводу данных — нам останется только их создать.

ГЛАВА 14

Списки Metro

В предыдущей главе мы познакомились с фрагментами — если так можно выразиться, самыми простыми из сложных элементов интерфейса, предлагаемых платформой Metro. Фрагменты позволят нам разделить интерфейс приложения на независимые части и выводить эти части на экран, только когда в них возникнет необходимость.

В этой главе мы продолжим изучать элементы управления Metro. Знакомство с ними состоялось еще в *главе 7*, но тогда мы рассмотрели только самые примитивные из этих элементов. Пришла пора рассмотреть более сложные.

Это списки Metro. Обрабатывающие практически любые данные, выводящие их практически в любом формате и берущие бóльшую часть работы по выводу и форматированию данных на себя. Невероятно мощные, фантастически гибкие и пугающе "самостоятельные"...

Списки Metro помогут нам вывести все, что угодно: содержимое канала RSS, набор графических изображений, сложные данные, полученные в результате вычисления или загруженные из Интернета. Списки Metro займут свое место в очень многих приложениях, если не в большей их части. Списки Metro — самые сложные и развитые элементы управления из всех, что предоставляет нам эта платформа. Все остальные элементы интерфейса посторонитесь — идут списки Metro!

Замечания о создании и инициализации списков Metro

Прежде чем начать работу со списками Metro, нам следует кое-что вспомнить и уяснить одну вещь. Все это связано с созданием и инициализацией списков Metro.

Из *главы 7* мы знаем, что любой элемент управления Metro создается на основе блока. Мы создаем в HTML-коде обычный блок, указываем в качестве значения атрибута `data-win-control` формирующего его тега `<div>` имя объекта, представляющего нужный нам элемент управления, а остальное — дело базовой логики Metro.

Как только интерфейс приложения будет загружен, нам понадобится вызвать метод `processAll` объекта `WinJS.UI`. После этого все элементы управления Metro будут инициализированы и приведены в рабочее состояние. Обычно метод `processAll` вызывается в обработчике события `DOMContentLoaded`, самым первым выражением в его теле.

Проблема в том, что метод `processAll` на самом деле не инициализирует элементы управления Metro, а лишь запускает процесс их инициализации. И процесс этот продолжается даже после того, как данный метод завершит свою работу и начнут выполняться выражения, следующие за его вызовом. Так что, если сразу после вызова метода `processAll` будет идти выражение, получающее доступ к элементу управления Metro, а данный элемент еще не был инициализирован, возникнет ошибка.

Ранее, работая с простыми элементами управления Metro, мы не сталкивались с такой проблемой. Простые элементы Metro инициализируются очень быстро и становятся работоспособными еще до того, как дойдет черед выполняться следующему после вызова метода `processAll` выражению. Но списки Metro — элементы значительно более сложные, и инициализация их может затянуться.

Метод `processAll` возвращает в качестве результата обязательство. С помощью метода `then` мы можем задать для него функцию, которая будет выполнена после инициализации всех элементов управления Metro, что присутствуют в приложении. Тело этой функции — идеальное место для кода, который будет получать доступ к элементам Metro, привязывать к их событиям обработчики и заполнять эти элементы данными.

```
<div id="divControl" data-win-control=" . . . "></div>
. . .
var ctrControl;
WinJS.UI.processAll().then(function() {
  ctrControl = document.getElementById("divControl").winControl;
  . . .
});
```

Список Metro, выводящий несколько позиций

Самый впечатляющий из списков Metro — это список, выводящий одновременно несколько позиций. Такие списки применяются повсеместно.

Создание списка, выводящего несколько позиций

Список Metro, выводящий несколько позиций, представляется объектом `WinJS.UI.ListView`. Имя этого объекта нам следует указать в качестве значения для атрибута `data-win-control` тега `<div>`, что создаст элемент-основу.

```
<div id="divList" data-win-control="WinJS.UI.ListView"></div>
```

По умолчанию список Metro выстраивает свои пункты сначала по вертикали, а потом — по горизонтали и реализует при этом горизонтальную прокрутку. Получает-

ся нечто похожее на таблицу, которая заполняется сначала по столбцам, а потом — по строкам.

Однако мы можем изменить это поведение, воспользовавшись свойством `layout` объекта `WinJS.UI.ListView`. В качестве значения оно принимает экземпляр одного из объектов, перечисленных далее:

- экземпляр объекта `WinJS.UI.GridLayout` указывает списку выстроить пункты сначала по вертикали, потом — по горизонтали с горизонтальной же прокруткой (вывод в виде таблицы; поведение по умолчанию);
- экземпляр объекта `WinJS.UI.ListLayout` указывает списку выстроить пункты по вертикали и реализовать вертикальную прокрутку (вывод в виде списка). При этом список Metro выглядит более традиционно.

```
var ctrList = document.getElementById("divList").winControl;
ctrList.layout = new WinJS.UI.ListLayout();
```

Но в большинстве случаев удобнее указать расположение пунктов списка прямо в теге `<div>`, который создает для него элемент-основу. (Как мы знаем из *главы 7*, для указания параметров элемента управления Metro применяется атрибут тега `data-win-options`.) Но как задать в качестве значения для свойства списка экземпляр объекта?

С помощью записи вот такого формата:

<имя свойства>: {type: *<имя объекта>*}

Имя объекта, на основе которого будет создан экземпляр, указывается без кавычек.

```
<div id="divList" data-win-control="WinJS.UI.ListView"
data-win-options="{layout: {type: WinJS.UI.ListLayout}}"></div>
```

Также по умолчанию список Metro позволяет выбрать сразу несколько пунктов. И мы также можем изменить это поведение.

Свойство `selectionMode` позволит нам указать режим выбора пунктов в списке. Оно поддерживает три предопределенных значения, которые должны быть указаны в виде строк:

- `none` — пользователь не сможет выбрать ни один пункт в списке;
- `single` — пользователь сможет выбрать только один пункт;
- `multi` — пользователь может выбрать сколько угодно пунктов (поведение по умолчанию).

```
<div id="divList" data-win-control="WinJS.UI.ListView"
data-win-options="{selectionMode: 'single'}"></div>
```

Для задания режима выбора пунктов в коде логики удобнее использовать перечисление `WinJS.UI.SelectionMode`. Поддерживаемые им элементы `none`, `single` и `multi` соответствуют перечисленным ранее строковым значениям.

```
ctrList.selectionMode = WinJS.UI.SelectionMode.single;
```

Создание пунктов списка

Пустой, не содержащий пунктов, список никому не нужен. Поэтому следующим нашим шагом будет заполнение готового списка Metro пунктами.

Подготовка массива данных

Данные, которые будут выводиться в списке Metro, должны представлять собой массив (он так и называется *массив данных*). И формируется он по особым правилам:

- каждый элемент массива данных представляет один пункт списка;
- каждый элемент массива данных должен хранить экземпляр объекта `Object` (об этом объекте рассказывалось в *главе 4*);
- каждый экземпляр объекта `Object`, хранящийся в массиве данных, должен включать набор свойств, которые будут хранить отдельные значения, выводимые в пункте списка.

```
var arrNews = [];
arrNews[0] = {
  date: new Date(2012, 1, 17),
  header: "Microsoft рассказала о ReFS, новой файловой системе Windows
  ↳Server 8",
  content: "В своем блоге B8 компания Microsoft поделилась информацией о
  ↳новой файловой системе Windows Server 8, известной под названием ReFS
  ↳и призванной заменить NTFS, которая используется в Windows
  ↳с 1993 года."
};
```

Здесь мы объявили согласно приведенным ранее правилам массив данных `arrNews` и создали первый его элемент, представляющий новость из канала RSS. Как видим, каждая новость будет содержать дату публикации (свойство `date`), заголовок (свойство `header`) и содержимое (свойство `content`).

Создание источника данных, получение адаптера и привязка его к списку

Итак, массив данных готов. Можно создавать на его основе *источник данных* — особую структуру, которая позволит списку получать из массива данные и формировать на их основе пункты.

Источник данных, созданный на основе массива данных, представляется объектом `WinJS.Binding.List`. Нам следует создать экземпляр этого объекта, передав ему в качестве параметра созданный ранее массив данных:

```
var dsrNews = new WinJS.Binding.List(arrNews);
```

После этого мы обратимся к свойству `dataSource` созданного источника данных, чтобы получить так называемый *адаптер*. Это экземпляр особого объекта, который можно рассматривать как посредник между источником данных и списком.

После чего останется только привязать полученный адаптер к списку Metro — и дело сделано. Для этого достаточно присвоить адаптер свойству `itemDataSource` списка.

```
ctrList.itemDataSource = dsrNews.dataSource;
```

После этого список сам сформирует пункты на основе заданных нами данных и выведет их на экран.

Использование шаблонов для оформления пунктов списка

Только пункты этого списка будут содержать совсем не то, что нам нужно. А именно — исходный код на языке JavaScript, с помощью которого мы создали соответствующие этим пунктам элементы массива данных.

Дело в том, что список Metro не "знает", какие значения из элемента массива данных, описывающего пункт, ему следует выводить на экран и какой формат для этого применить. Поэтому он поступает по умолчанию — преобразует каждый элемент массива в текстовое представление, т. е. в JavaScript-код, который его формирует. (Кстати, для этого применяется поддерживаемый всеми объектами метод `toString`, о котором упоминалось в *главе 4*.)

Нам необходимо указать, какой формат списку Metro следует использовать для формирования своих пунктов. Или, если придерживаться терминологии Metro, задать для списка *шаблон* — описание этого формата.

Обычно шаблон Metro формируется на языке HTML прямо в файле, где описывается интерфейс приложения (или фрагмента). HTML-код, создающий шаблон, обязательно должен находиться перед кодом, формирующим список, к которому он будет привязан; в противном случае список его не "найдет".

Шаблон — это особый элемент управления Metro. Он создается так же, как остальные элементы управления Metro, и представляется объектом `WinJS.Binding.Template`.

```
<div id="divListTemplate" data-win-control="WinJS.Binding.Template">
</div>
```

Внутри элемента-основы шаблона формируются элементы интерфейса, в которых, собственно, и будут выводиться необходимые данные. Теги, формирующие эти элементы интерфейса, должны быть пустыми.

Для каждого из этих элементов интерфейса нам нужно указать два свойства. Во-первых, это свойство экземпляра объекта `Object` — элемента массива данных, откуда будет взято значение, что будет выводиться в этом элементе. Во-вторых, это свойство самого элемента интерфейса, которому будет присвоено данное значение. Например, значение свойства `header` элемента массива данных мы можем присвоить свойству `textContent` абзаца, а значение свойства `src` элемента массива данных — свойству `src` элемента — графического изображения.

Указать эти свойства мы можем в атрибуте тега `data-win-bind`, значение которого записывается в следующем формате:

```
<свойство элемента интерфейса>: <свойство экземпляра объекта Object —
↪элемента массива данных>
<div id="divListTemplate" data-win-control="WinJS.Binding.Template">
  <p data-win-bind="textContent: date"></p>
  <p data-win-bind="textContent: header"></p>
</div>
```

Здесь мы создали шаблон с двумя абзацами. Свойству `textContent` первого из них будет присвоено значение свойства `date` элемента массива данных, в результате чего данный абзац получит в качестве текстового содержимого дату публикации новости, которая перед выводом будет автоматически преобразована в строковый вид. А во втором абзаце будет выведен заголовок новости (значение свойства `header` элемента массива данных).

Мы можем задать оформление для элементов интерфейса, входящих в шаблон, привязав к ним стили.

```
<div id="divListTemplate" data-win-control="WinJS.Binding.Template">
  <p data-win-bind="textContent: date" class="new-date"></p>
  <p data-win-bind="textContent: header" class="new-header"></p>
</div>
. . .
.new-date { text-align: right; }
.new-header { font-size: 12pt; }
```

Здесь мы указали для абзаца с датой публикации выравнивание текста по правому краю, а для абзаца с заголовком — размер шрифта в 12 пунктов.

Осталось лишь привязать готовый шаблон к списку. Для этого достаточно присвоить имя его элемента-основы свойству `itemTemplate` списка:

```
<div id="divList" data-win-control="WinJS.UI.ListView"
data-win-options="{itemTemplate: divListTemplate}"></div>
```

Отметим две вещи. Во-первых, мы присваиваем этому свойству имя элемента-основы шаблона; получив его, список Metro сам "отыщет" созданный на его основе шаблон. Во-вторых, мы указываем это имя без кавычек.

Вот теперь наш список будет выводить то, что нам нужно.

Еще платформа Metro позволяет создать шаблон в виде функции (так называемый *шаблон-функцию*). Эта функция будет формировать пункты списка программно. (О программном формировании интерфейса *см. в главе 10*.)

В качестве единственного параметра она должна принимать обязательство. Для этого обязательства мы в вызове метода `then` укажем функцию, которая в качестве параметра получит экземпляр объекта `Object`, представляющий формируемый пункт. Его свойство `data`, в свою очередь, будет хранить экземпляр объекта `Object` — соответствующий данному пункту элемент массива данных.

Результатом, который будет возвращать функция, заданная для этого обязательства, станет полностью созданный элемент интерфейса, который сформирует пункт списка. Этот элемент интерфейса должен представлять собой блок, в котором бу-

дут находиться элементы, представляющие отдельные части этого пункта: абзацы, заголовки, другие блоки и др.

Помимо этого, шаблон-функция должна вернуть в качестве результата обязательство, что она получила как параметр.

Лучше один раз увидеть, чем сто раз услышать. Поэтому давайте рассмотрим пример шаблона-функции.

```
function listTemplate(itemPromise) {
  return itemPromise.then(function(item) {
    var oItem = item.data;
    var oDiv = document.createElement("div");
    var oP = document.createElement("p");
    oP.className = "new-date";
    oP.textContent = oItem.date.getDate() + "." +
      (oItem.date.getMonth() + 1) + "." + oItem.date.getFullYear();
    oDiv.appendChild(oP);
    oP = document.createElement("p");
    oP.className = "new-header";
    oP.textContent = oItem.header;
    oDiv.appendChild(oP);
    return oDiv;
  });
}
```

Его можно использовать как заготовку для создания других шаблонов-функций.

Для формирования значения даты в привычном нам формате *<число>.<месяц>.<год>* мы использовали три метода объекта `Date`. Метод `getDate`, знакомый нам по *главе 7*, возвращает число, метод `getMonth` — номер месяца от 0 до 11 (январь — декабрь), а метод `getFullYear` — год в "полном" формате (четыре цифры). Все они не принимают параметров и возвращают результат в виде целого числа.

Осталось только привязать готовый шаблон-функцию к списку. Сделать это можно только в коде логики.

```
ctrList.itemTemplate = listTemplate;
```

Шаблон-функция имеет перед обычным, описанным в HTML-коде шаблоном определенные преимущества. Во-первых, мы можем использовать в таком шаблоне значения, полученные в результате каких-либо вычислений (так, в приведенном ранее примере мы вычисляли значение даты в привычном нам формате). Во-вторых, в зависимости от выполнения какого-либо условия мы можем менять содержимое пунктов списка. В HTML-шаблоне ничего этого сделать не получится.

Фильтрация пунктов списка

Источник данных предоставляет нам интересную и полезную возможность — *фильтрацию* пунктов списка. То есть вывод в списке только тех пунктов, что удовлетворяют определенному условию — *критерию* фильтрации.

Если мы хотим реализовать в своем списке фильтрацию, нам не обойтись без метода `createFiltered` источника данных.

```
<источник данных>.createFiltered(<функция, задающая критерий фильтрации>)
```

Единственным параметром он принимает функцию, которая реализует критерий фильтрации. В качестве единственного параметра данная функция примет экземпляр объекта — элемент массива данных, а вернет логическое значение: `true`, если данный элемент должен быть выведен в списке, и `false`, если не должен.

Метод `createFiltered` возвращает в качестве результата новый источник данных — уже отфильтрованный.

```
var oDate = new Date(2012, 2, 17);
function filterList(item) {
  return (item.date.getTime() == oDate.getTime());
}
var dsrFiltered = dsrNews.createFiltered(filterList);
ctrList.itemDataSource = dsrFiltered.dataSource;
```

Выводим в списке только новости, опубликованные 17 января 2012 года.

Метод `getTime` объекта `Date` возвращает количество миллисекунд, прошедших между значением даты и полночью 1 января 1970 года. Фактически он позволяет получить числовое представление даты, которое можно использовать в операциях сравнения (что мы и сделали).

Самое интересное, что фильтрация, которую мы реализуем, не затронет сам массив данных. Это значит, что все элементы, не удовлетворяющие заданному нами критерию фильтрации, хоть и не будут присутствовать в новом источнике данных, но все же останутся в составе массива.

Сортировка пунктов списка

Другая возможность, предоставляемая нам источником данных Metro, — сортировка пунктов списка. Отсортированные по какому-либо критерию списки легче читаются.

Выполнить сортировку нам поможет метод `createSorted` источника данных.

```
<источник данных>.createSorted(<функция сравнения>)
```

Единственным параметром этому методу передается так называемая *функция сравнения*. Она принимает два параметра — два сравниваемых элемента массива данных — и возвращает в качестве результата:

- отрицательное число, если первый элемент "меньше" второго;
- 0, если оба этих элемента "равны";
- положительное число, если первый элемент "больше" второго.

Метод `createSorted` возвращает в качестве результата новый — отсортированный — источник данных.

```
function compareItems(item1, item2) {
  return item1.date.getTime() — item2.date.getTime();
}
var dsrSorted = dsrNews.createSorted(compareItems);
ctrList.itemDataSource = dsrSorted.dataSource;
```

Выводим в списке новости, отсортированные по дате публикации.

Отметим, что в теле функции сравнения мы вычитаем числовое представление даты публикации второй новости из числового представления даты публикации первой новости. (Как говорилось ранее, эти представления можно получить вызовом метода `getTime`.) Это самый простой способ получить значение, указывающее, какое из сравниваемых значений "больше", а какое — "меньше".

Сортировка, что мы реализуем в источнике данных, также не будет затрагивать исходный массив данных.

Группировка пунктов списка

Вдоволь налюбовавшись списком Metro, самостоятельно фильтрующим и сортирующим данные, давайте дадим ему задачу потруднее. А именно, заставим его группировать пункты по какому-либо признаку, в нашем случае — по дате публикации новостей.

В процессе *группировки* пункты списка разбиваются на отдельные группы по значению указанного нами признака — *критерия группировки*. Таким критерием может быть значение какого-либо свойства элемента в массиве данных, фрагмент этого значения или результат вычисления, выполненного на основе сразу нескольких значений. Например, если мы выберем в качестве критерия группировки дату публикации новости, то пункты списка будут разбиты на группы, каждая из которых будет соответствовать определенному значению этой даты.

Группы будут выведены отсортированными. Критерий их сортировки также задаем мы.

При выводе на экран каждой группе пунктов будет предшествовать *заголовок*. В заголовке обычно указывается значение критерия, общее для пунктов, что входят в группу (например, значения даты публикации).

> ***Внимание!***
>
> Группировка в списке Metro станет полноценно работать только в случае вывода пунктов в виде таблицы. Если пункты выводятся в виде списка, заголовки групп отображаться не будут.

Чтобы реализовать группировку, нам сначала придется объявить три функции.

Первая функция будет определять значение критерия группировки для каждого элемента массива данных. Она примет в качестве единственного параметра сам этот элемент и вернет соответствующее ему значение критерия группировки обязательно в виде строки.

```
function getGroupKeyValue(item) {
  return item.date.getTime().toString();
}
```

Используем в качестве критерия группировки числовое представление даты публикации новости, преобразованное в строку.

Вторая функция будет подготавливать данные, которые понадобятся для формирования групп и будут впоследствии выводиться в их заголовках. В качестве единственного параметра она также примет элемент массива данных, а в качестве результата вернет экземпляр объекта `Object` со свойствами, которые будут хранить все необходимые для формирования заголовков групп значения.

```
function getGroupData(item) {
  return {
    date: item.date.getDate() + "." + (item.date.getMonth() + 1) + "." +
      item.date.getFullYear()
  };
}
```

Экземпляр объекта `Object`, который вернет эта функция, будет включать свойство `date` — дату публикации новости в виде строки. Ее-то мы и выведем в заголовке группы.

Третья функция будет представлять уже знакомую нам функцию сравнения *(см. ранее)*. Она получит в качестве параметров два значения критерия группировки.

```
function sortGroups(group1, group2) {
  return parseInt(group1) – parseInt(group2);
}
```

Объявив все эти три функции, мы вызовем метод `createGrouped` источника данных:

<источник данных>.createGrouped(*<первая функция>*, *<вторая функция>*, *<третья функция>*)

В качестве параметров он примет три объявленные нами функции. Результатом, который вернет этот метод, станет новый источник данных, уже сгруппированных.

```
var dsrGrouped = dsrNews.createGrouped(getGroupKeyValue, getGroupData,
sortGroups);
```

Полученный источник данных мы привяжем к списку известным нам способом:

```
ctrList.itemDataSource = dsrGrouped.dataSource;
```

Помимо этого, нам потребуется получить еще один источник данных — содержащий все группы. Он хранится в свойстве `groups` источника сгруппированных данных. Извлеченный из него адаптер мы присвоим свойству `groupDataSource` списка Metro.

```
ctrList.groupDataSource = dsrGrouped.groups.dataSource;
```

Останется только создать шаблон, на основе которого будут формироваться заголовки групп, и привязать его к списку. Такой шаблон создается по тем же правилам, что и шаблон для обычных пунктов списка. За одним исключением: данные для вывода он будет брать из экземпляра объекта `Object`, возвращаемого второй по счету из объявленных нами функций.

```
<div id="divGroupTemplate" data-win-control="WinJS.Binding.Template">
  <p data-win-bind="textContent: date"></p>
</div>
```

Чтобы привязать шаблон заголовка группы к списку, присвоим его свойству `groupHeaderTemplate` списка:

```
<div id="divList" data-win-control="WinJS.UI.ListView"
data-win-options="{itemTemplate: divListTemplate,
↳groupHeaderTemplate: divGroupTemplate}"></div>
```

Как и в случае фильтрации и сортировки, реализованная нами группировка не будет затрагивать исходный массив данных.

Получение выбранных пунктов

Что ж, мы заполнили список пунктами и, возможно, реализовали их фильтрацию, сортировку или группировку. Теперь нам нужно как-то выяснить, какие пункты списка выбраны пользователем.

Для этого служит свойство `selection` списка. Оно возвращает экземпляр особого объекта-коллекции, представляющего набор всех выбранных в списке пунктов.

```
var oSel = ctrList.selection;
```

Этот объект поддерживает метод `count`, не принимающий параметров и возвращающий в качестве результата количество выбранных пунктов в виде целого числа.

```
if (oSel.count() > 0) {
  //Получаем выбранные пункты
}
```

Получить выбранные пункты мы можем, вызвав метод `getItems` этого же объекта. Он не принимает параметров и возвращает в качестве результата обязательство, для которого мы зададим в вызове метода `then` функцию, что выполнится после завершения процесса получения выбранных пунктов. В качестве единственного параметра она получит массив, каждый элемент которого будет экземпляром объекта `Object` со свойствами `index` и `data`; первое свойство будет хранить номер выбранного пункта, а второе — соответствующий этому пункту элемент массива данных.

> ***Внимание!***
>
> Следует иметь в виду, что таким образом мы получим не индексы элементов в массиве данных, соответствующих выбранным пунктам, а номера самих выбранных пунктов. А если мы реализовали в списке фильтрацию, сортировку или группировку, эти значения не будут идентичными!

```
var arrSelectedNews = [];
oSel.getItems().then(function(selectedItems) {
  for (var i = 0; i < selectedItems.length; i++) {
    arrSelectedNews.push(selectedItems[i].data);
  }
});
```

Получаем все пункты, выбранные пользователем, и помещаем их в массив `arrSelectedNews`.

Объект-коллекция, представляющий набор выбранных пунктов списка, поддерживает несколько полезных методов, о которых мы сейчас поговорим.

Метод `set` позволяет сделать выбранными пункты списка с указанными номерами. В качестве единственного параметра он принимает массив номеров выбираемых пунктов и не возвращает результата.

```
oSel.set([1, 3, 10]);
```

Делаем выбранными второй, четвертый и одиннадцатый пункты списка. (Не забываем, что нумерация пунктов начинается с нуля.)

Метод `selectAll` делает все пункты списка выделенными, а метод `clear`, наоборот, снимает выделение со всех пунктов. Оба метода не принимают параметров и не возвращают результата.

```
oSel.selectAll();
```

Событие `selectionchanged` списка возникает, когда состав выбранных в данный момент пунктов изменяется, т. е. когда пользователь выбирает другой пункт или отменяет выбор у выбранного ранее. Мы можем использовать это событие, чтобы отслеживать выбор пунктов в списке.

Впрочем, если список позволяет выбрать только один пункт, наша задача серьезно упрощается. При выборе любого пункта в списке возникает событие `iteminvoked`, обработчик которого мы можем использовать для отслеживания выбора пункта и его получения.

Выяснить, какой пункт был выбран, можно из экземпляра объекта `CustomEvent`, который передается в функцию-обработчик этого события единственным параметром. Объект `CustomEvent` представляет события, возникающие в элементах управления Metro, и поддерживает все свойства и методы, доступные в объекте `Event` *(см. главу 5)*.

А еще он поддерживает свойство `detail`. Это свойство хранит набор дополнительных данных о событии в виде экземпляра объекта `Object`, свойства которого и содержат эти самые данные.

В случае события `iteminvoked` данный экземпляр объекта `Object` содержит свойство `itemPromise`. Оно хранит обязательство, для которого мы в вызове метода `then` укажем функцию, что выполнится после получения выбранного пункта. Эта функция примет в качестве параметра экземпляр объекта `Object`, свойство `index` которого будет хранить номер выбранного пункта, а свойство `data` — соответствующий ему элемент массива данных.

```
ctrList.addEventListener("iteminvoked", function(evt) {
  evt.detail.itemPromise.then(function(selected) {
    var selectedIndex = selected.index;
```

```
    var selectedItem = selected.data;
    . . .
  });
});
```

Еще нам может пригодиться метод `getAt` источника данных. Он принимает в качестве единственного параметра номер пункта списка в виде числа и возвращает в качестве результата соответствующий этому пункту элемент массива данных.

```
var oItem = dsrList.getAt(0);
```

Получаем элемент массива данных, соответствующий первому пункту списка.

Источник данных также поддерживает свойство `length`, знакомое нам по массивам. Оно возвращает количество пунктов в списке.

```
var oItem = dsrList.getAt(dsrList.length — 1);
```

Получаем элемент массива данных, соответствующий последнему пункту списка.

Реализация правки данных, выводимых в списке

Не всякое приложение в продолжение всей своей работы манипулирует статичным набором данных. В большинстве случаев эти данные меняются, в том числе и самим пользователем.

Можно, конечно, вносить изменения прямо в массив данных, но это очень неудобно. Куда удобнее пользоваться методами объекта `WinJS.Binding.List`, которые мы сейчас рассмотрим.

Давно знакомый нам метод `push` позволяет добавить в список новый пункт. В качестве единственного параметра он принимает добавляемый пункт, заданный в том формате, в котором указываются элементы массива данных. А возвращает он новое значение количества пунктов в списке.

```
dsrList.push({
  date: new Date(2012, 1, 8),
  header: "Анонсирована линейка игр для Windows 8",
  content: "Microsoft активно готовится к релизу бета-версии Windows 8
  ↳(в терминологии Microsoft, Windows 8 Consumer Preview), в котором
  ↳пользователям станет доступен магазин приложений Windows Store. И вот
  ↳сетевому изданию TheVerge стал известен список игр, которые можно
  ↳будет загрузить из Windows Store."
});
```

Отметим, что новый элемент будет добавлен в самый конец списка.

Метод `setAt` позволит изменить содержимое указанного нами пункта.

<источник данных>`.setAt(`*<номер изменяемого пункта>*`, `*<новое содержимое>*`)`

Первым параметром указывается номер пункта списка, содержимое которого следует изменить, а вторым — само новое содержимое в том же формате, что используется для добавления нового пункта.

```
dsrList.setAt(0, {
  date: new Date(2012, 1, 6),
  header: "В сети появились скриншоты Windows 8 build 8220",
  content: "На выходных в сети появились скриншоты сборки Windows 8
  ↳build 8220, являющейся одним из кандидатов для предстоящей публичной
  ↳бета-версии, которая должна стать доступной до конца текущего месяца."
});
```

Заменяем содержимое первого пункта списка.

Удалить пункт списка мы можем вызовом метода `splice`.

<источник данных>.splice(*<номер первого из удаляемых пунктов>*, *<количество удаляемых пунктов>*)

Первым параметром передается номер первого из удаляемых пунктов списка, вторым — количество следующих подряд пунктов, которые нужно удалить. Оба значения задаются в виде целых чисел.

```
dsrList.splice(2, 1);
```

Удаляем третий пункт списка.

```
dsrList.splice(dsrList.length – 1, 1);
```

Удаляем последний пункт списка.

Метод `move` позволяет переместить пункт с указанным номером на новое место.

<источник данных>.move(*<текущий номер перемещаемого пункта>*, *<новый номер перемещаемого пункта>*)

Первым параметром ему передается номер пункта, который следует переместить, а вторым — номер, который он получит после перемещения. Результата этот метод не возвращает.

```
dsrList.move(dsrList.length – 1, 0);
```

Перемещаем последний пункт списка в самое его начало.

Как уже говорилось, все манипуляции с пунктами списка затрагивают массив данных. Так, если мы добавим в список новый пункт, источник данных добавит в массив соответствующий элемент.

Прочие возможности

Осталось рассмотреть остальные возможности списка Metro, выводящего несколько позиций, которые могут нам пригодиться.

Метод `ensureVisible` списка выполняет прокрутку списка таким образом, чтобы нужный нам пункт стал видимым. В качестве единственного параметра этот метод принимает номер пункта в виде целого числа и не возвращает результата.

```
ctrList.ensureVisible(dsrList.length – 1);
```

Прокручиваем список в самый конец.

Теперь рассмотрим возможности по оформлению списков. Реализуются они созданием стилевых классов со строго определенными именами.

Стилевой класс `.win-listview` задает оформление для самого списка. Он автоматически привязывается к самому элементу-основе.

```
.win-listview {
  width: 100%;
  height: 100%;
}
```

Растягиваем все списки Metro, что входят в интерфейс нашего приложения, на все пространство их родителей.

Стилевой класс `.win-item` задает оформление для пунктов списка.

```
#divList .win-item {
  width: calc(25% — 10px);
  padding-left: 10px;
}
```

Задаем для пунктов нашего списка отступ слева, равный 10 пикселам, и ширину, равную 25% от ширины списка минус заданное ранее значение отступа слева.

Стилевой класс `.win-groupheader` задает оформление для заголовков групп.

```
#divList .win-groupheader { height: 50px; }
```

Пример: прототип приложения для чтения каналов RSS, использующий список Metro

Когда мы создавали прототип приложения для чтения каналов RSS, то для вывода списка новостей применили обычный блок. Нам пришлось программно формировать в нем блоки, представляющие позиции списка, помещать в них абзацы со сведениями о новостях, а также привязывать к ним обработчики событий `click`. Все это потребовало написания довольно объемистого кода.

Но мы тогда еще не знали о существовании списков Metro, выводящих несколько позиций. А ведь такой список сам сделает за нас львиную часть работы. Так давайте же возьмем его в дело!

Откроем в Visual Studio проект `RSSReader`. Откроем файл default.js, отыщем код, объявляющий массив `arrNews`, и переделаем его таким образом, чтобы объявляемый массив удовлетворял всем требованиям списков Metro.

```
var arrNews = [];
arrNews[0] = {
  date: new Date(2012, 0, 17),
  title: "Microsoft рассказала о ReFS, новой файловой системе Windows
  ↳Server 8",
  content: "В своем блоге B8 компания Microsoft поделилась информацией
  ↳о новой файловой системе Windows Server 8, известной под названием
```

```
  ↳ReFS и призванной заменить NTFS, которая используется в Windows
  ↳с 1993 года."
};
arrNews[1] = {
  date: new Date(2012, 0, 18),
  title: "Опубликованы минимальные требования к устройствам под
  ↳управлением Windows 8",
  content: "В прошлом месяце Microsoft опубликовала свою документацию по
  ↳аппаратным требованиям для сертификации Windows 8. Этот документ
  ↳содержит рекомендации от Microsoft для разработки систем, которые
  ↳соответствуют референсным по производительности, качеству и прочим
  ↳критериям, для обеспечения оптимального удобства работы Windows 8."
};
. . .
```

Далее изменим код, объявляющий необходимые переменные, таким образом:

```
var divBase, selectedNew = -1;
```

Значение –1 в JavaScript-программировании традиционно означает, что в списке не выбран ни один пункт. А у нас изначально так оно и есть.

Добавим код, создающий источник данных с сортировкой новостей по дате их публикации:

```
var dsrTemp = new WinJS.Binding.List(arrNews);
function compareItems(item1, item2) {
  return item1.date.getTime() — item2.date.getTime();
}
var dsrNews = dsrTemp.createSorted(compareItems);
```

И модифицируем код, создающий пространство имен `BaseApplication`.

```
WinJS.Namespace.define("BaseApplication", {
  dsrNews: dsrNews,
  selectedNew: selectedNew,
  loadList: loadList,
  loadContent: loadContent
});
```

Мы добавили в это пространство имен переменную, в которой хранится созданный нами источник данных, и удалили оттуда массив данных.

Теперь откроем файл list.html и исправим код, находящийся внутри тега `<body>`, чтобы он выглядел так:

```
<div id="divList" data-win-control="WinJS.UI.ListView"
data-win-options="{layout: {type: WinJS.UI.ListLayout},
↳selectionMode: 'single'}"></div>
```

Мы создали список Metro, в котором указали расположение пунктов по вертикали и возможность выбора только одного пункта.

Откроем файл list.css, удалим все его содержимое и создадим там стили, описанные далее.

```
#divList {
  width: 100%;
  height: 100%;
}
#divList .win-item { padding-left: 10px; }
```

Мы растянули список на все пространство родителя и указали для его пунктов отступ слева в 10 пикселов.

```
.new-date {
  font-size: 14pt;
  font-style: italic;
}
.new-title { font-size: 14pt; }
```

Для заголовка и даты публикации новости мы указали размер шрифта, равный 14 пунктам, а для даты публикации — еще и курсивное начертание.

Откроем файл list.js и сразу добавим код, объявляющий переменную, в которой будет храниться список новостей.

```
var ctrList;
```

Добавим объявление шаблона-функции:

```
function listTemplate(itemPromise) {
  return itemPromise.then(function(item) {
    var oItem = item.data;
    var oDiv = document.createElement("div");
    var oP = document.createElement("p");
    oP.className = "new-date";
    oP.textContent = oItem.date.getDate() + "." +
    (oItem.date.getMonth() + 1) + "." + oItem.date.getFullYear();
    oDiv.appendChild(oP);
    oP = document.createElement("p");
    oP.className = "new-header";
    oP.textContent = oItem.title;
    oDiv.appendChild(oP);
    return oDiv;
  });
}
```

Исправим объявление функции `fillList`, после чего оно будет выглядеть так:

```
function fillList() {
  WinJS.UI.processAll().then(function() {
    ctrList = document.getElementById("divList").winControl;
    ctrList.itemDataSource = BaseApplication.dsrNews.dataSource;
    ctrList.itemTemplate = listTemplate;
```

```
    if (BaseApplication.selectedNew != -1) {
      ctrList.selection.set([BaseApplication.selectedNew]);
    }
    ctrList.addEventListener("iteminvoked", ctrListItemInvoked);
  });
}
```

Здесь мы привязываем к списку созданные ранее источник данных и шаблон-функцию, выясняем, была ли ранее в списке выбрана какая-либо новость, и, если это так, делаем ее выбранной в данный момент. (Это необходимо в случае, если пользователь прочитает текст выбранной новости и снова вернется к списку новостей.) Напоследок мы привязываем к событию `iteminvoked` списка обработчик.

Этот обработчик нам также следует объявить.

```
function ctrListItemInvoked(evt) {
  evt.detail.itemPromise.then(function(selected) {
    BaseApplication.selectedNew = selected.index;
    BaseApplication.loadContent();
  });
}
```

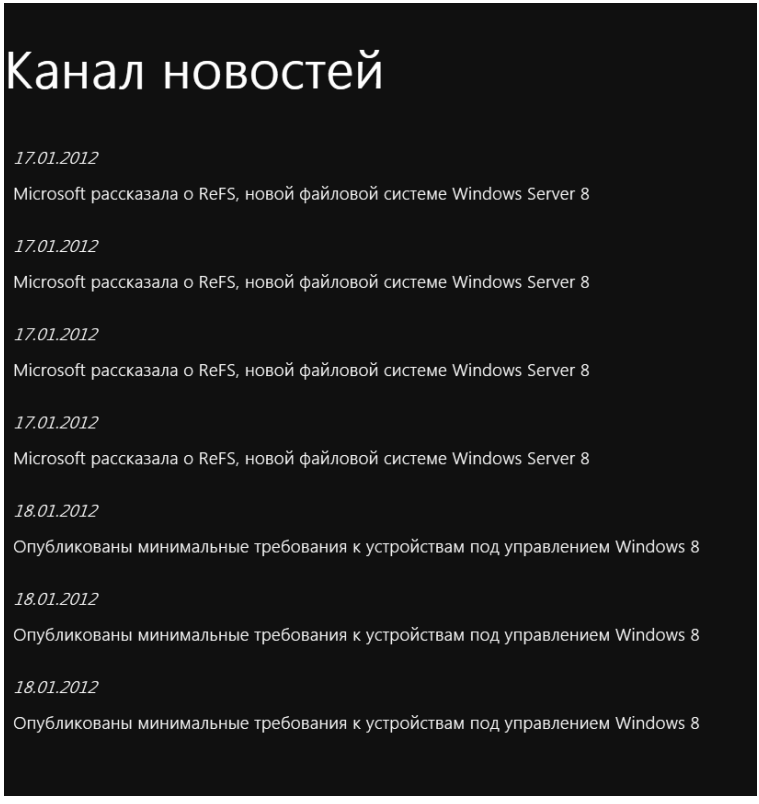


Рис. 14.1. Список Metro, выводящий новости, в очередной версии прототипа приложения для чтений каналов RSS

Здесь мы получаем номер выбранного пользователем пункта и инициируем загрузку фрагмента content.html, который выведет содержимое соответствующей новости.

Функцию `divItemClick` можно удалить — она нам больше не понадобится.

Осталось открыть файл content.js, найти в нем объявление функции `showContent`, а в нем — выражение, выводящее на экран содержимое новости:

```
divText.innerHTML =
BaseApplication.arrNews[BaseApplication.selectedNew][2];
```

Его следует исправить таким образом:

```
divText.innerHTML =
BaseApplication.dsrNews.getAt(BaseApplication.selectedNew).content;
```

Закончив с правками, сохраним все файлы и запустим приложение. Полюбуемся на список новостей (рис. 14.1), выберем в нем какой-либо пункт и ознакомимся с содержимым соответствующей новости. После этого снова вернемся к списку новостей и попробуем выбрать другой пункт.

Список-слайдшоу Metro

Список-слайдшоу Metro, в отличие от рассмотренного нами ранее списка, одновременно выводит на экран только одну позицию. Для переключения между позициями применяются кнопки **Предыдущая** (Previous) и **Следующая** (Next); эти кнопки автоматически формируются при создании списка. Обычно такие списки используют для организации слайдшоу, из-за чего они и получили свое название.

Создание списка-слайдшоу

Список-слайдшоу Metro представляется объектом `WinJS.UI.FlipView`. Имя этого объекта мы укажем в качестве значения для атрибута `data-win-control` тега `<div>`, что создаст элемент-основу.

```
<div id="divSlideshow" data-win-control="WinJS.UI.FlipView"></div>
```

Объект `WinJS.UI.FlipView` поддерживает уже знакомые нам свойства `itemDataSource` и `itemTemplate`, предназначенные, соответственно, для указания источника данных и шаблона.

```
<div id="divSlideshowTemplate" data-win-control="WinJS.Binding.Template">
  <h1 data-win-bind="textContent: title"></h1>
  <img data-win-bind="src: src" />
</div>
<div id="divSlideshow" data-win-control="WinJS.UI.FlipView"
data-win-options="{itemTemplate: divSlideshowTemplate}"></div>
```

Создаем список-слайдшоу, предназначенный для вывода коллекции графических изображений. Он будет выводить, помимо самого изображения, еще и его заголовок.

```
var arrPictures = [];
arrPictures[0] = {
  title: "Первое изображение",
  src: "/images/pictures/1.jpg"
};
arrPictures[1] = {
  title: "Второе изображение",
  src: "/images/pictures/2.jpg"
};
. . .
var dsrPictures = new WinJS.Binding.List(arrPictures);
var ctrSlideshow;
WinJS.UI.processAll().then(function() {
  ctrSlideshow = document.getElementById("divSlideshow").winControl;
  ctrSlideshow.itemDataSource = dsrSlideshow.dataSource;
}
```

Создаем и привязываем к списку источник данных.

Свойство `orientation` объекта `WinJS.UI.FlipView` позволяет задать ориентацию списка. Оно принимает в качестве значения строки `horizontal` (горизонтальная ориентация; значение по умолчанию) и `vertical` (вертикальная ориентация).

```
<div id="divSlideshow" data-win-control="WinJS.UI.FlipView"
data-win-options="{orientation: 'vertical'}"></div>
```

Работа со списком-слайдшоу

Рассмотрим несколько свойств, методов и событий объекта `WinJS.UI.FlipView`, которые могут нам пригодиться.

Свойство `currentPage` хранит номер пункта списка, что отображается на экране в данный момент, в виде целого числа.

```
<div id="divSlideshow" data-win-control="WinJS.UI.FlipView"
data-win-options="{currentPage: 2}"></div>
```

Создаем список, который изначально выведет на экран третий пункт.

```
ctrSlideshow.currentPage = 2;
```

Выводим третий пункт списка программно.

Методы `previous` и `next` выводят на экран, соответственно, предыдущий и следующий пункты списка. Эти методы не принимают параметров.

А возвращают они в качестве значения `true`, если переход на предыдущий или следующий пункт успешно начат, и `false`, если его невозможно выполнить. Такое может случиться, если в списке выводится самый первый или самый последний пункт либо если список уже выполняет переход на другой пункт. Обычно возвращаемый этими методами результат игнорируют.

```
ctrSlideshow.next();
```

Событие pageselected возникает в списке, когда он успешно выводит на экран какой-либо пункт.

```
ctrSlideshow.addEventListener("pageselected", function() {
  var iNumber = ctrSlideshow.currentPage + 1;
  divOutput.textContent = "Выведен пункт № " + iNumber + " из " +
  arrPictures.length;
});
```

Здесь мы привязываем к событию pageselected списка-слайдшоу обработчик, который будет выводить в блоке divOutput номер отображаемого в данный момент пункта в привычном нам формате — начинающимся с единицы — и общее количество пунктов списка.

Осталось рассмотреть возможности по оформлению списков-слайдшоу, реализуемые путем создания стилевых классов с определенными именами.

Стилевой класс .win-flipview задает оформление для самого списка-слайдшоу.

```
.win-flipview {
  width: 100%;
  height: 100%;
}
```

Стилевой класс .win-item задает оформление для пунктов списка-слайдшоу.

```
.win-flipview .win-item { vertical-align: top; }
```

Что дальше?

В этой главе мы работали со списками Metro — исключительно мощными элементами управления, позволяющими вывести любые данные в любом указанном нами формате, возможно, с применением фильтрации, сортировки и группировки. Пожалуй, это одни из наиболее востребованных элементов интерфейса.

Следующая глава будет посвящена другой группе элементов управления Metro, чье назначение — удобная организация различных элементов интерфейса и вывод их на экран по запросу пользователя. Это панели инструментов и всплывающие элементы, без которых тоже редко когда можно обойтись.

ГЛАВА 15

Панели инструментов, всплывающие элементы и меню

В предыдущей главе мы познакомились с самыми развитыми из элементов управления платформы Metro — списками. Первый из этих списков позволяет выводить на экран сразу несколько позиций и применяется во всех случаях, когда следует отобразить набор каких-то однотипных данных. Второй список может отображать одновременно только одну позицию и идеально подходит для создания слайдшоу.

Еще в *главе 13* мы узнали, что далеко не все элементы интерфейса, применяемые в приложении, выводятся на экран одновременно. Более того, на экране постоянно присутствует только очень небольшой их набор; остальные выводятся лишь тогда, когда в них возникает необходимость.

Для реализации такого подхода платформа Metro предлагает три пути. Путь первый — использование фрагментов (о них рассказывалось в той же *главе 13*); он наилучшим образом подходит для вывода на экран больших фрагментов интерфейса. Путь второй — временное скрытие элементов интерфейса, пока они не понадобятся пользователю; его лучше выбирать, если данные элементы являются частью основного интерфейса, и выносить их во фрагменты нежелательно.

Третий же путь — применение особых элементов управления Metro, специально предназначенных для организации других элементов интерфейса. Они сами управляют местоположением находящихся в них элементов и позволяют скрывать и выводить их на экран по запросу пользователя или логики самого приложения.

К таким элементам управления относятся панели инструментов, всплывающие элементы и меню. Настала пора познакомиться с ними.

Панели инструментов Metro

Панели инструментов Metro (в терминологии этой платформы — app bar) выглядят и ведут себя аналогично их "тезкам" из обычных Windows-приложений. Они располагаются в верхней или нижней части экрана и включают в себя набор графических кнопок и, возможно, других элементов интерфейса: полей ввода, регуляторов, индикаторов прогресса, блоков с текстом и пр.

Панели инструментов могут как постоянно присутствовать на экране, так и скрываться, пока в них не возникнет нужда. Панели инструментов могут выводиться на экран как самим пользователям — путем вытягивания из-за края экрана, — так и программно. Скрываться они также могут и программно, и автоматически, по истечении определенного промежутка времени или после нажатия на экран.

В панели инструментов рекомендуется выносить элементы интерфейса, потребность в которых у пользователя должна возникать достаточно часто. В качестве примера можно привести кнопки управления воспроизведением в приложении видеопроигрывателя, кнопки подписки на канал RSS, сохранения выбранной новости в файл или обновления содержимого канала в приложении чтения новостей RSS и пр.

Создание панелей инструментов, содержащих только кнопки

Чаще всего встречаются панели инструментов, содержащие только кнопки. Другие элементы интерфейса в таких панелях инструментов, к сожалению, использовать нельзя — это ограничение самой платформы Metro.

Создание самих панелей инструментов

Сначала нам следует создать саму панель инструментов. Она представляется объектом `WinJS.UI.AppBar`, чье имя мы укажем в качестве значения атрибута `data-win-control` тега `<div>`, создающего элемент-основу.

```
<div id="divAppBar" data-win-control="WinJS.UI.AppBar"></div>
```

HTML-код, создающий панель инструментов, может находиться в любом месте кода интерфейса. Панель инструментов — элемент достаточно "умный", он сам займет место на экране, которое мы для него укажем.

Объект `WinJS.UI.AppBar` поддерживает ряд свойств, которые помогут нам указать местоположение и поведение панели инструментов.

Свойство `placement` позволяет задать местоположение панели инструментов. Оно принимает в качестве значений две предопределенные строки: `top` (панель инструментов находится в верхней части экрана) и `bottom` (в нижней части экрана; поведение по умолчанию).

```
<div id="divTopAppBar" data-win-control="WinJS.UI.AppBar"
data-win-options="{placement: 'top'}"></div>
```

Создаем верхнюю панель инструментов.

Свойство `sticky` включает или отключает автоматическое скрытие панели инструментов по истечении определенного промежутка времени. Значение `true` отключает автоматическое скрытие, а значение `false` — включает (поведение по умолчанию).

Помимо этого, панель инструментов поддерживает свойство `disabled`.

Создание обычных кнопок

На втором этапе в панелях инструментов формируются кнопки. Они являются особыми элементами управления Metro и представляются объектом `WinJS.UI.AppBarCommand`. Создаются эти кнопки с помощью давно знакомых нам тегов `<button>` *(см. главу 6)*, которые помещаются в тег `<div>`, формирующий элемент-основу.

```
<div id="divAppBar" data-win-control="WinJS.UI.AppBar">
  <button data-win-control="WinJS.UI.AppBarCommand"></button>
</div>
```

Для каждой кнопки мы укажем параметры, воспользовавшись специально предназначенными для этого свойствами объекта `WinJS.UI.AppBarCommand`.

Свойство `id` служит для указания имени кнопки. Оно аналогично одноименному атрибуту тега, которым мы пользуемся давным-давно.

Свойство `label` указывает надпись для кнопки.

Свойство `icon` задает графическое обозначение, или *глиф*, для кнопки. Проще всего задать его как элемент перечисления `WinJS.UI.AppBarIcon` — оно содержит огромный набор предопределенных глифов, среди которых легко можно найти подходящий. Список всех элементов этого перечисления и соответствующих им глифов приведен на Web-странице **http://msdn.microsoft.com/en-us/library/windows/apps/hh770557.aspx**.

Свойство `tooltip` позволяет задать для кнопки текст всплывающей подсказки. Если таковой не указана, во всплывающей подсказке будет выводиться текст надписи.

Свойство `section` указывает *секцию* панели инструментов, в которой будет находиться данная кнопка. Таких секций платформа Metro предусматривает две:

- левая, включающая кнопки, которые должны быть доступны только в определенных случаях (например, только после открытия документа, только после выделения текста и пр.);
- правая, включающая кнопки, которые должны быть доступны всегда (к ним можно отнести кнопки создания нового документа, открытия файла и др.).

Значение свойства `selection` указывается в виде одной из строк: `section` (поместить кнопку в левую секцию) или `global` (в правую).

Еще кнопка панели инструментов поддерживает свойство `disabled`.

```
<div id="divAppBar" data-win-control="WinJS.UI.AppBar">
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnPlay', label: 'Пуск', icon: 'play',
  ↳section: 'selection', tooltip: 'Воспроизведение',
  ↳disabled: true}"></button>
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnStop', label: 'Стоп', icon: 'stop',
  ↳section: 'selection', tooltip: 'Стоп', disabled: true}"></button>
```

```
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnOpen', label: 'Открыть',
  ↳icon: 'openfile', section: 'global', tooltip: 'Открыть'}"></button>
</div>
```

Создаем панель инструментов с тремя кнопками: **Пуск**, **Стоп** и **Открыть**. Первые две кнопки будут включены в состав левой секции панели инструментов, а третья — в состав правой.

Создание кнопок-выключателей

Еще платформа Metro позволяет нам создать в панели инструментов так называемые *кнопки-выключатели*. Такая кнопка при первом нажатии на нее активизирует определенную функцию (или, как еще говорят, включается), а при втором нажатии — деактивирует (выключается). В качестве кнопки-выключателя можно выполнить, скажем, кнопку приглушения звука в приложении видеопроигрывателя.

Объект `WinJS.UI.AppBarCommand`, представляющий кнопку панели инструментов, поддерживает свойство `type`. Значение этого свойства задается в виде одной из предопределенных строк.

По умолчанию данное свойство имеет значение `button`, предписывающее Metro создать обычную кнопку. Чтобы создать кнопку-выключатель, достаточно задать этому свойству значение `toggle`.

Свойство `selected` кнопки хранит логическое значение `true`, если кнопка-выключатель включена, и `false`, если она выключена. Значение этого свойства по умолчанию — `false`.

```
<button data-win-control="WinJS.UI.AppBarCommand"
data-win-options="{id: 'btnMute', label: 'Тихо!', icon: 'mute',
↳section: 'global', tooltip: 'Тихий режим', type: 'toggle'}"></button>
```

Создаем кнопку-выключатель **Тихо!**.

Создание разделителей

Еще платформа Metro позволяет нам визуально отделить на панели инструментов одну кнопку от другой, вставив между ними так называемые *разделители*. Такой разделитель выглядит как тонкая вертикальная линия.

Разделитель Metro представляется тем же объектом, что и кнопка, — `WinJS.UI.AppBarCommand`. Он создается с помощью одинарного тега `<hr>`, а его свойство `type` должно иметь строковое значение `separator`.

```
<button data-win-control="WinJS.UI.AppBarCommand"
data-win-options="{section: 'global' . . .}"></button>
<hr data-win-control="WinJS.UI.AppBarCommand"
data-win-options="{type: 'separator', section: 'global'}" />
<button data-win-control="WinJS.UI.AppBarCommand"
data-win-options="{section: 'global' . . .}"></button>
```

Создаем разделитель между двумя кнопками. (Формирующий его код выделен полужирным шрифтом.)

Создание универсальных панелей инструментов

Универсальные панели инструментов могут содержать любые элементы интерфейса: блоки, абзацы, поля ввода, списки, обычные кнопки, регуляторы и др. Но, к сожалению, рассмотренные ранее кнопки, что представляются объектом `WinJS.UI.AppBarCommand`, вставить в такие панели нельзя — они не будут работать. Так что или — или...

Объект `WinJS.UI.AppBar`, представляющий панель инструментов, поддерживает свойство `layout`. Значение этого свойства должно представлять собой одну из предопределенных строк:

- `commands` — создает обычную панель инструментов, способную содержать только кнопки (значение по умолчанию);
- `custom` — создает универсальную панель инструментов.

```
<div id="divTopAppBar" data-win-control="WinJS.UI.AppBar"
data-win-options={layout: 'custom', position: 'top'}>
  <input type="range" id="sldRange" />
</div>
```

Создаем универсальную панель инструментов, включающую регулятор.

Работа с панелями инструментов и кнопками

А теперь рассмотрим несколько полезных для нас свойств, методов и событий объекта `WinJS.UI.AppBar`.

Метод `getCommandById` принимает в качестве параметра строку, содержащую имя кнопки, и возвращает экземпляр объекта `WinJS.UI.AppBarCommand`, представляющий эту кнопку, или `null`, если кнопки с таким именем в панели инструментов нет.

```
var ctrAppBar = document.getElementById("divAppBar").winControl;
var btnPlay = ctrAppbar.getCommandById("btnPlay");
btnPlay.addEventListener("click", function() { . . . });
```

Получаем кнопку **Пуск** и привязываем к ней обработчик события.

Метод `show` выводит панель инструментов на экран, а метод `hide` ее скрывает. Они не принимают параметров и не возвращают результатов.

```
ctrAppBar.show();
```

Свойство `hidden` возвращает `true`, если панель инструментов скрыта, и `false`, если она выведена на экран.

```
if (ctrAppBar.hidden) {
  //Панель инструментов скрыта
}
```

Метод showCommands выводит на экран указанные кнопки.

```
<панель инструментов>.showCommands(<выводимые кнопки>)
```

В качестве параметра передается массив кнопок, которые следует вывести на экран. В этот массив могут входить как строки с именами кнопок, так и экземпляры объекта WinJS.UI.AppBarCommand, представляющие эти кнопки.

Метод showCommands не возвращает результата.

```
ctrAppBar.showCommands([btnPlay, "btnStop"]);
```

Выводим на экран кнопки **Пуск** и **Стоп**. Отметим, что для кнопки **Пуск** мы указали экземпляр объекта, что ее представляет, а для кнопки **Стоп** — имя в виде строки.

Метод hideCommands, напротив, скрывает указанные нами кнопки. Он принимает те же параметры, что и метод showCommands, и также не возвращает результата.

```
ctrAppBar.hideCommands(["btnStop"]);
```

Скрываем кнопку **Стоп**.

Метод showOnlyCommands выводит на экран указанные кнопки и скрывает все остальные. Вызывается он так же, как и метод showCommands.

```
ctrAppBar.showOnlyCommands(["btnOpen"]);
```

Выводим кнопку **Открыть** и скрываем все остальные кнопки.

Теперь поговорим о событиях панели инструментов. Их четыре:

- beforeshow — возникает перед выводом панели инструментов на экран;
- aftershow — возникает после вывода панели инструментов на экран;
- beforehide — возникает перед скрытием панели инструментов;
- afterhide — возникает после скрытия панели инструментов.

```
ctrAppBar.addEventListener("beforeshow", function() { . . . });
```

Теперь мы можем выполнить какие-либо действия перед выводом панели инструментов на экран.

Объект WinJS.UI.AppBarCommand, представляющий кнопку, также поддерживает свойство hidden.

Пример: видеопроигрыватель, использующий панели инструментов

В качестве практического занятия давайте в очередной раз модифицируем приложение видеопроигрывателя, вынеся все присутствующие на экране элементы управления в две панели инструментов — верхнюю и нижнюю. После этого на экране останется только элемент видеопроигрывателя, и ничто не будет отвлекать пользователя от просмотра фильма.

Откроем в Visual Studio проект VideoPlayer. Откроем файл default.html, удалим весь код, находящийся внутри тега <body>, и введем туда новый код:

```
<div id="divVideo" data-win-control="WinJS.UI.ViewBox">
  <video id="vidMain" preload="auto"></video>
</div>
<div id="divTopAppBar" data-win-control="WinJS.UI.AppBar"
data-win-options="{layout: 'custom', placement: 'top'}">
  <div id="divProgress">
    <input type="range" id="sldProgress" min="0" step="1" value="0"
    disabled />
  </div>
  <div id="divTiming"></div>
</div>
<div id="divBottomAppBar" data-win-control="WinJS.UI.AppBar">
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnPlay', label: 'Пуск', icon: 'play',
  ↳section: 'selection'}"></button>
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnPause', label: 'Пауза', icon: 'pause',
  ↳section: 'selection'}"></button>
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnStop', label: 'Стоп', icon: 'stop',
  ↳section: 'selection'}"></button>
  <hr data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'dvd1', section: 'selection',
  ↳type: 'separator'}" />
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnClose', label: 'Закрыть',  icon: 'clear)',
  ↳section: 'selection'}"></button>
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnOpen', label: 'Открыть', icon: 'openfile',
  ↳section: 'global'}"></button>
  <hr data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{section: 'global', type: 'separator'}" />
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnMute', label: 'Тихо!', icon: 'mute',
  ↳section: 'global', type: 'toggle'}"></button>
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnVolumeDown', label: 'Тише', icon: 'down',
  ↳section: 'global'}"></button>
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnVolumeUp', label: 'Громче', icon: 'up',
  ↳section: 'global'}"></button>
</div>
```

Из прежних элементов интерфейса здесь остались только панель вывода Metro и находящийся в ней видеопроигрыватель. Остальные элементы мы создали заново.

Это, прежде всего, универсальная панель инструментов `divTopAppBar`, расположенная в верхней части экрана. Она включит в себя два блока, которые мы разместим с применением сеточной разметки:

- левый — содержащий регулятор позиции воспроизведения;
- правый — выводящий текущую позицию воспроизведения и продолжительность фильма (хронометраж).

Вторая панель инструментов — `divBottomAppBar` — будет располагаться в нижней части экрана и содержать только кнопки. Состав этих кнопок таков:

- **Пуск**, **Пауза**, **Стоп, Закрыть** и разделитель, находящиеся в левой секции;
- **Открыть**, **Тихо!**, **Тише**, **Громче** и еще один разделитель, находящиеся в правой секции.

Настала очередь оформления. Откроем файл default.css, удалим все его содержимое и создадим несколько стилей, описанных далее.

```
#divVideo {
  width: 100%;
  height: 100%;
}
```

Растягиваем панель вывода Metro на весь экран.

```
#divTopAppBar {
  display: -ms-grid;
  -ms-grid-columns: 1fr 300px;
  -ms-grid-rows: 1fr;
}
```

Создаем сеточную разметку в верхней панели инструментов.

```
#divProgress { -ms-grid-row-align: center; }
```

Для блока с регулятором позиции воспроизведения задаем вертикальное выравнивание по центру.

```
#divTiming {
  -ms-grid-column: 2;
  font-size: 24pt;
  text-align: right;
  vertical-align: middle;
}
```

Задаем для блока, в котором будет выводиться хронометраж, размер шрифта в 24 пункта, горизонтальное выравнивание по правому краю и вертикальное — по центру.

```
#sldProgress { width: 100%; }
```

И растягиваем регулятор позиции воспроизведения на всю ширину блока.

Осталась самая трудная часть работы — написание логики. Переключимся на файл default.js, отыщем выражение, объявляющее необходимые переменные, и изменим его таким образом:

```
var btnPlay, btnPause, btnStop, btnOpen, btnClose, btnMute,
btnVolumeDown, btnVolumeUp, vidMain, sldProgress, divTiming,
ctrTopAppBar, ctrBottomAppBar, sCurrent, sDuration;
```

Традиционно после объявления переменных мы занимаемся обработчиком события `DOMContentLoaded`, выполняющим инициализацию приложения. Код обновленного обработчика будет таким:

```
document.addEventListener("DOMContentLoaded", function () {
  WinJS.UI.processAll().then(function() {
    ctrTopAppBar = document.getElementById("divTopAppBar").winControl;
    ctrBottomAppBar = document.getElementById("divBottomAppBar").
    ↳winControl;
    btnPlay = ctrBottomAppBar.getCommandById("btnPlay");
    btnPause = ctrBottomAppBar.getCommandById("btnPause");
    btnStop = ctrBottomAppBar.getCommandById("btnStop");
    btnOpen = ctrBottomAppBar.getCommandById("btnOpen");
    btnClose = ctrBottomAppBar.getCommandById("btnClose");
    btnMute = ctrBottomAppBar.getCommandById("btnMute");
    btnVolumeDown = ctrBottomAppBar.getCommandById("btnVolumeDown");
    btnVolumeUp = ctrBottomAppBar.getCommandById("btnVolumeUp");
    vidMain = document.getElementById("vidMain");
    sldProgress = document.getElementById("sldProgress");
    divTiming = document.getElementById("divTiming");

    btnPlay.addEventListener("click", btnPlayClick);
    btnPause.addEventListener("click", btnPauseClick);
    btnStop.addEventListener("click", btnStopClick);
    btnOpen.addEventListener("click", btnOpenClick);
    btnClose.addEventListener("click", btnCloseClick);
    btnMute.addEventListener("click", btnMuteClick);
    btnVolumeDown.addEventListener("click", btnVolumeClick);
    btnVolumeUp.addEventListener("click", btnVolumeClick);
    vidMain.addEventListener("click", vidMainClick);
    vidMain.addEventListener("canplay", vidMainCanPlay);
    vidMain.addEventListener("playing", vidMainPlaying);
    vidMain.addEventListener("pause", vidMainPause);
    vidMain.addEventListener("timeupdate", vidMainTimeUpdate);
    sldProgress.addEventListener("change", sldProgressChange);

    ctrBottomAppBar.hideCommands([btnPlay, btnPause, btnStop, "dvd1",
    btnClose]);
    ctrTopAppBar.show();
    ctrBottomAppBar.show();

    vidMain.volume = 0.5;
  });
});
```

Здесь мы получаем доступ ко всем необходимым элементам интерфейса и привязываем к ним обработчики событий. После этого мы скрываем все содержимое левой секции; в результате на экране останутся только те кнопки, что находятся в правой секции. Напоследок выводим обе панели инструментов на экран и задаем исходное значение громкости.

Отметим, что мы также привязали обработчик к событию `click` видеопроигрывателя. Этот обработчик будет выводить и скрывать панели инструментов.

Исправим функцию `vidMainCanPlay()` следующим образом:

```
function vidMainCanPlay() {
  ctrBottomAppBar.showCommands([btnPlay, btnStop, "dvd1", btnClose]);
  sldProgress.value = 0;
  sldProgress.max = vidMain.duration;
  sldProgress.disabled = false;
  d.setHours(0, 0, vidMain.duration);
  sDuration = d.getHours() + ":" + d.getMinutes() + ":" + d.getSeconds();
  showTiming();
}
```

Здесь мы выводим на экран кнопки **Пуск**, **Стоп** и **Закрыть** и левый разделитель; теперь пользователь сможет запустить фильм на воспроизведение. Остальные действия нам уже знакомы по предыдущей версии приложения.

Напишем объявление функции `btnStopClick()`, которая будет вызвана при нажатии кнопки **Стоп**:

```
function btnStopClick() {
  vidMain.pause();
  sldProgress.value = 0;
  vidMain.currentTime = 0;
  showTiming();
}
```

Видеопроигрыватель HTML не предоставляет стандартной возможности остановить воспроизведение фильма. Но для нас это не проблема. Мы поставим воспроизведение на паузу и установим позицию воспроизведения фильма в начало. Теперь пользователь сможет начать просмотр фильма с самого начала.

На очереди — объявление функции `btnCloseClick()`, которая выполнится в ответ на нажатие кнопки **Закрыть**:

```
function btnCloseClick() {
  vidMain.src = "";
  sldProgress.disabled = true;
  ctrBottomAppBar.hideCommands([btnPlay, btnPause, btnStop, "dvd1",
  btnClose]);
  divTiming.textContent = "";
}
```

Стандартного средства для закрытия загруженного файла видеопроигрыватель HTML также не предоставляет. Но где наша не пропадала! Мы присвоим свойству `src` видеопроигрывателя пустую строку, сделаем недоступным регулятор позиции воспроизведения, скроем кнопки **Пуск**, **Пауза**, **Стоп** и **Закрыть** вместе с левым разделителем и очистим блок, где выводится хронометраж фильма. Фактически мы вернем приложение в исходное состояние.

Код, объявляющий функцию `btnMuteClick()`, очень прост:

```
function btnMuteClick() {
  vidMain.muted = btnMute.selected;
}
```

Эта функция просто присваивает свойству `muted` видеопроигрывателя инвертированное значение свойства `selected` кнопки `btnMute`. Так что, если данная кнопка включена, звук будет приглушен, и наоборот.

Исправим функции `vidMainPlaying()` и `vidMainPause()`, чтобы объявляющий их код выглядел так:

```
function vidMainPlaying() {
  ctrBottomAppBar.showCommands([btnPause]);
  ctrBottomAppBar.hideCommands([btnPlay]);
  ctrTopAppBar.hide();
  ctrBottomAppBar.hide();
}
function vidMainPause() {
  ctrBottomAppBar.showCommands([btnPlay]);
  ctrBottomAppBar.hideCommands([btnPause]);
}
```

Здесь мы просто скрываем одну кнопку и делаем видимой другую. Функция `vidMainPlaying()`, помимо этого, скрывает обе панели инструментов, чтобы не мозолить глаза пользователю.

Объявим функцию `btnVolumeClick()` — обработчик события `click` кнопок `btnVolumeDown` и `btnVoluemUp`:

```
function btnVolumeClick (evt) {
  var newVolume;
  if (evt.target.winControl == btnVolumeDown) {
    newVolume = vidMain.volume - 0.05;
    if (newVolume < 0) {
      newVolume = 0;
    }
  } else {
    newVolume = vidMain.volume + 0.05;
    if (newVolume > 1) {
      newVolume = 1;
    }
  }
```

```
  btnVolumeDown.disabled = (newVolume == 0);
  btnVolumeUp.disabled = (newVolume == 1);
  vidMain.volume = newVolume;
}
```

Мы просто уменьшаем или увеличиваем значение громкости и при достижении им пределов (0 и 1) делаем соответствующие кнопки недоступными.

Последнее, что мы должны сделать, — объявить функцию `vidMainClick()`, которая станет обработчиком события `click` видеопроигрывателя.

```
function vidMainClick() {
  if ((ctrTopAppBar.hidden) || (ctrBottomAppBar.hidden)) {
    ctrTopAppBar.show();
    ctrBottomAppBar.show();
  } else {
    ctrTopAppBar.hide();
    ctrBottomAppBar.hide();
  }
}
```

Если хотя бы одна панель инструментов скрыта, обе они будут выведены на экран; в противном случае данная функция их скроет.

Функции `swtMuteChange()` и `sldVolumeChange()`, сохранившиеся со старых времен, можно удалить — больше они нам не понадобятся.

Сохраним все файлы и запустим приложение. Откроем какой-либо видеофайл, запустим его на воспроизведение, попытаемся поставить на паузу, отключить и включить звук, остановить воспроизведение и закрыть файл. Посмотрим на интерфейс приложения (рис. 15.1), чтобы удостовериться, все ли мы правильно сделали.

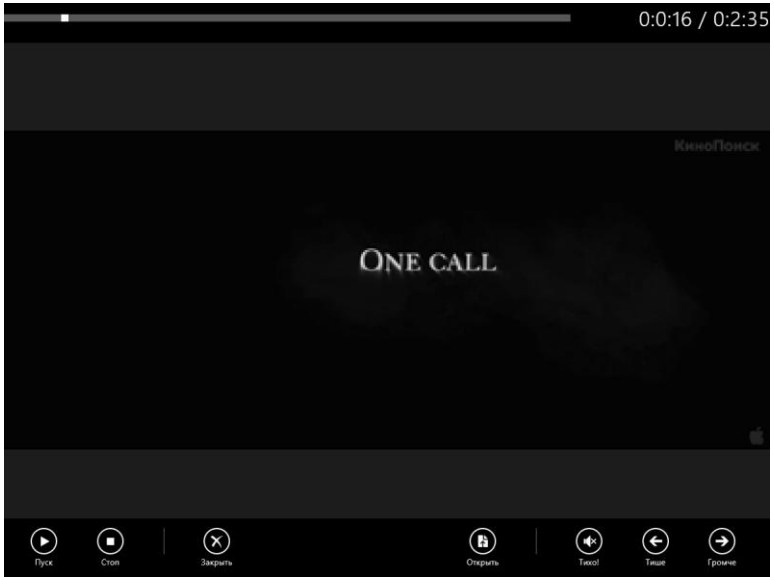


Рис. 15.1. Интерфейс видеопроигрывателя, использующего панели инструментов

Что ж, очередная версия нашего видеопроигрывателя полностью соответствует всем правилам построения интерфейса, что рекомендуются к использованию Microsoft. Так что, когда мы, в конце концов, опубликуем наше приложение в магазине Windows Store (об этом будет рассказано в *главе 28*), у тестировщиков не возникнет к нам никаких претензий.

Всплывающие элементы Metro

Всплывающие элементы (flyout) Metro по своему назначению аналогичны окнам в обычных Windows-приложениях. Они содержат какие-либо элементы интерфейса (кнопки, поля ввода, списки, фрагменты текста и др.), появляются на экране по запросу пользователя и располагаются поверх основного интерфейса.

Всплывающие элементы могут выводиться на экран как программно, так и автоматически, после нажатия на кнопку в панели инструментов. Скрываться они также могут как программно, так и автоматически, после того как пользователь нажмет на любой элемент интерфейса, не являющийся частью всплывающего элемента.

Во всплывающих элементах рекомендуется располагать элементы интерфейса, нужда в которых возникает не столь часто, как в элементах, что обычно помещаются в панели инструментов. В качестве примеров можно привести форму для ввода сведений о канале RSS, на который пользователь хочет подписаться, сведения о воспроизводящемся видеофайле и пр.

Создание всплывающего элемента

Всплывающий элемент представляется объектом `WinJS.UI.Flyout`. Его имя следует указать в атрибуте `data-win-control` тега `<div>`, создающего элемент-основу.

```
<div id="divFlyout" data-win-control="WinJS.UI.Flyout"></div>
```

Внутри этого тега создаются элементы интерфейса, которые станут частью всплывающего элемента.

```
<div id="divFlyout" data-win-control="WinJS.UI.Flyout">
  <div>
    <label for="txtRSSAddress">Интернет-адрес канала RSS</label>
    <br />
    <input type="url" id="txtRSSAddress" />
  </div>
  <div>
    <input type="button" id="btnSubscribe" value="Подписаться" />
  </div>
</div>
```

HTML-код, создающий всплывающий элемент, может располагаться в любом месте кода интерфейса.

Во всплывающих элементах общего назначения, предназначенных для ввода данных, не должны находиться кнопки простого закрытия или отмены ввода. (Такие

кнопки в традиционных Windows-приложениях обычно имеют надпись **Отмена** или **Закрыть**.) Поскольку всплывающие элементы скрываются автоматически, после нажатия на любой элемент интерфейса, который не входит в их состав, потребность в таких кнопках отсутствует.

Работа со всплывающими элементами

Для вывода всплывающего элемента на экран применяется метод `show` объекта `WinJS.UI.Flyout`:

```
<всплывающий элемент>.show(<элемент интерфейса, возле которого он будет
↳выведен>[, <месторасположение>[, <выравнивание>]])
```

Первым параметром данный метод принимает элемент интерфейса, возле которого он будет выведен на экран. Этот параметр может представлять собой либо строку с именем нужного элемента интерфейса, заданного с помощью атрибута тега `id`, либо экземпляр объекта, представляющего этот элемент.

Вторым, необязательным, параметром указывается местоположение всплывающего элемента относительно элемента, заданного в первом параметре. Его значение должно представлять собой одну из следующих строк:

- `auto` — месторасположением всплывающего элемента управляет сама платформа Metro (значение по умолчанию);
- `top` — всплывающий элемент выводится выше указанного элемента;
- `bottom` — ниже;
- `left` — левее;
- `right` — правее.

Третьим, также необязательным, параметром задается выравнивание всплывающего элемента. Этот параметр имеет смысл только в том случае, если всплывающий элемент выводится выше или ниже указанного. Его значением должна быть одна из следующих строк: `left` (выравнивание по левому краю), `center` (выравнивание по центру; значение по умолчанию) и `right` (выравнивание по правому краю).

Метод `show` не возвращает значения.

```
var ctrFlyout = document.getElementById("divFlyout").winControl;
ctrFlyout.show("btnAdd", "bottom", "right");
```

Выводим созданный ранее всплывающий элемент `divFlyout` ниже и правее кнопки `btnAdd`.

Помимо этого, объект `WinJS.UI.Flyout` поддерживает свойство `hidden`, метод `hide` и события `beforeshow`, `aftershow`, `beforehide` и `afterhide`, знакомые нам по объекту `WinJS.UI.AppBar` *(см. ранее в этой главе)*.

```
ctrFlyout.hide();
```

Скрываем всплывающий элемент сами, не дожидаясь, пока это сделает пользователь.

Пример: вывод сведений о видеофайле в приложении видеопроигрывателя

В качестве примера возьмем наш многострадальный, неоднократно переделанный видеопроигрыватель и переделаем его еще раз. Пусть при нажатии на блоке, в котором показывается хронометраж, на экране появляется всплывающий элемент со сведениями об открытом видеофайле, а именно размерами его изображения.

Снова откроем в Visual Studio проект `VideoPlayer`. Откроем файл default.html и добавим в код, уже присутствующий в теге `<body>`, следующий фрагмент:

```
<div id="divInfo" data-win-control="WinJS.UI.Flyout">
  <p>Ширина: <span id="spnWidth"></span></p>
  <p>Высота: <span id="spnHeight"></span></p>
</div>
```

Здесь мы создаем всплывающий элемент и указываем в качестве его содержимого два абзаца, предназначенные, соответственно, для вывода значений ширины и высоты изображения. Для вывода собственно этих значений мы создали в каждом абзаце встроенный контейнер.

Теперь переключимся на файл default.js и объявим переменные для хранения всплывающего элемента и обоих встроенных контейнеров.

```
var ctrInfo, spnWidth, spnHeight;
```

Далее найдем обработчик события `DOMContentLoaded`. Добавим в его код следующие выражения (выделены полужирным шрифтом):

```
document.addEventListener("DOMContentLoaded", function () {
  WinJS.UI.processAll().then(function() {
    . . .
    ctrInfo = document.getElementById("divInfo").winControl;
    spnWidth = document.getElementById("spnWidth");
    spnHeight = document.getElementById("spnHeight");
    divTiming.addEventListener("click", divTimingClick);
    ctrInfo.addEventListener("beforeshow", ctrInfoBeforeShow);
  });
});
```

Они получат доступ ко всем вновь созданным элементам интерфейса и привяжут к событию `click` блока `dimTiming` функцию-обработчик `divTimingClick()`, а к событию `beforeshow` всплывающего элемента — функцию-обработчик `ctrInfoBeforeShow()`.

Объявим функцию `divTimingClick()`.

```
function divTimingClick() {
  if (vidMain.videoWidth > 0) {
    ctrInfo.show(divTiming, "bottom", "right");
  }
}
```

Видеопроигрыватель предоставит нам значения ширины и высоты изображения у видеоролика только в том случае, если файл с этим роликом в нем открыт; в противном случае оба этих значения будут равны нулю. Так что сначала мы проверяем, доступны ли эти значения (не равно ли нулю значение ширины изображения), и, только удостоверившись в этом, выводим всплывающий элемент на экран ниже блока `divTiming` с выравниванием по правому краю.

Основную работу по получению и выводу размеров изображения выполнит функция `ctrInfoBeforeShow()`.

```
function ctrInfoBeforeShow() {
  spnWidth.textContent = vidMain.videoWidth;
  spnHeight.textContent = vidMain.videoHeight;
}
```

Осталось только сохранить все исправленные файлы, запустить приложение, открыть какой-либо видеофайл и проверить вновь добавленную функцию в действии. На рис. 15.2 представлен результат, который мы должны увидеть на экране.

Рис. 15.2. Всплывающий элемент, выводящий ширину и высоту изображения у видеоролика, в приложении видеопроигрывателя

Вывод всплывающего элемента после нажатия кнопки на панели инструментов

Предположим, что нам требуется реализовать в своем приложении вывод всплывающего элемента на экран после нажатия кнопки в панели инструментов. Тогда нам крупно повезло — платформа Metro предлагает для этого стандартные средства, не требующие никакого программирования.

Создадим в панели инструментов кнопку и сделаем с ней две вещи. Во-первых, зададим для ее свойства `type` строковое значение `flyout`. Во-вторых, укажем в другом ее свойстве — `flyout` — всплывающий элемент, который требуется выводить на экран. Этот элемент может быть задан либо в виде строки с именем, либо в виде представляющего его экземпляра объекта.

Помимо этого, нам следует переместить HTML-код, создающий всплывающий элемент, таким образом, чтобы он находился перед кодом, формирующим саму панель инструментов с кнопкой, которая будет выводить его на экран. Если этого не сделать, всплывающий элемент не будет выведен.

```
<button data-win-control="WinJS.UI.AppBarCommand"
data-win-options="{id: 'btnAdd', label: 'Подписаться', icon: 'add',
↳section: 'global', type: 'flyout', flyout: 'divFlyout'}"></button>
```

После нажатия этой кнопки на экране появится всплывающий элемент `divFlyout`.

В качестве практики попробуйте реализовать вывод всплывающего элемента, отображающего сведения о видеофайле, по нажатию специальной кнопки на нижней панели инструментов. Благо сложного программирования для этого не потребуется.

Меню

Меню Metro схоже по внешнему виду и назначению с аналогичным элементом интерфейса традиционных Windows-приложений. Его можно рассматривать как разновидность всплывающего элемента, поскольку оно ведет себя сходным образом.

Само меню представляется объектом `WinJS.UI.Menu`. Имя этого объекта мы зададим в атрибуте `data-win-control` тега `<div>`, создающего элемент-основу.

```
<div id="divMenu" data-win-control="WinJS.UI.Menu"></div>
```

HTML-код, создающий меню, может находиться в любом месте кода интерфейса.

Объект `WinJS.UI.Menu` поддерживает уже знакомые нам свойства `disabled` и `hidden`, методы `getCommandById`, `show`, `hide`, `showCommands`, `hideCommands` и `showOnlyCommands` и события `beforeshow`, `aftershow`, `beforehide` и `afterhide`.

Пункты меню создаются с помощью тегов `<button>`, а разделители — с помощью тегов `<hr>`. Все они представляются объектом `WinJS.UI.MenuCommand`, который поддерживает знакомые нам свойства `id`, `label`, `type`, `selected`, `flyout`, `disabled` и `hidden`.

```
<div id="divMenu" data-win-control="WinJS.UI.Menu">
  <button data-win-control="WinJS.UI.MenuCommand"
  data-win-options="{id: 'btnSave', label: 'Сохранить'}"></button>
  <button data-win-control="WinJS.UI.MenuCommand"
  data-win-options="{id: 'btnSaveAll', label: 'Сохранить все'}"></button>
  <hr data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{type: 'separator'}" />
  <button data-win-control="WinJS.UI.MenuCommand"
  data-win-options="{id: 'btnDelete', label: 'Удалить'}"></button>
</div>
```

Создаем меню с пунктами **Сохранить**, **Сохранить все** и **Удалить** и разделителем между двумя последними пунктами.

Создание *пунктов-выключателей*, аналогичных по функциональности кнопкам-выключателям, и пунктов, выводящих всплывающие элементы и другие меню, осуществляется так же, как создание кнопок панелей инструментов.

```
var ctrMenu = document.getElementById("divMenu").winControl;
ctrMenu.hideCommands(["btnSave", "btnDelete"]);
ctrMenu.show();
```

Скрываем пункты **Сохранить** и **Удалить** и выводим меню на экран.

Как уже говорилось ранее, мы можем реализовать автоматический вывод меню на экран после нажатия кнопки в панели инструментов. Делается это так же, как и в случае обычных всплывающих элементов.

Что дальше?

В этой главе мы рассмотрели еще три полезных нам элемента управления Metro: панель инструментов, всплывающий элемент и меню. Панели инструментов помогут нам организовать элементы интерфейса, потребность в которых возникает постоянно, всплывающие элементы мы можем использовать для размещения тех элементов, что необходимы лишь эпизодически, а зачем обычно применяется меню, мы и так знаем.

На этом мы заканчиваем рассмотрение возможностей платформы Metro по созданию интерфейса приложений и переходим к знакомству с ее внутренними механизмами. И следующая же глава книги будет посвящена механизмам, позволяющим нам работать с файлами: выбирать их, читать, создавать, записывать, копировать, удалять, получать сведения о них и др. Нам это очень пригодится, поскольку подавляющее большинство любых приложений, так или иначе, работают с файлами.

ЧАСТЬ V

Работа с файлами и сетью

ГЛАВА 16

Работа с файлами

В предыдущей главе мы учились работать с панелями инструментов и всплывающими элементами Metro, а также узнали, насколько полезными могут быть эти штуки.

Эта глава будет посвящена работе с внутренними механизмами платформы Metro, обеспечивающими работу с файлами. Эти механизмы весьма развиты и позволяют нам открывать и создавать произвольные файлы, выполнять чтение из файлов и запись в них, получать список файлов, хранящихся в указанной папке, получать сведения о конкретном файле, а также копировать, перемещать и удалять их. Так что при написании приложений, работающих с файлами, никаких проблем у нас возникнуть не должно.

По ходу дела мы познакомимся с системой безопасности, реализуемой платформой Metro, а именно правами Metro-приложений. И, разумеется, выясним, как эти права задаются.

Диалог открытия файла

Очень многие приложения дают пользователю возможность выбрать файл для открытия. Для этого платформа Metro предоставляет особый диалог, называемый *диалогом открытия файла*. Он появляется на экране, перекрывая интерфейс приложения, и содержит список файлов, хранящихся в указанной нами папке, и кнопки **Открыть** (Open) и **Отмена** (Cancel).

Собственно, мы познакомились с этим диалогом еще в *главе 2*, когда создавали наше первое Metro-приложение — видеопроигрыватель. Так что тема нам уже частично знакома.

Подготовка диалога открытия файла

Диалог открытия файла в платформе Metro представляется объектом `Window.Storage.Pickers.FileOpenPicker`. Нам потребуется создать экземпляр этого объекта и сразу же присвоить его какой-либо переменной.

```
var oFOP = new Windows.Storage.Pickers.FileOpenPicker();
```

Далее мы зададим для диалога открытия файла нужные параметры. Для этого используется набор особых свойств, поддерживаемых объектом `Window.Storage.Pickers.FileOpenPicker`.

Свойство `suggestedStartLocation` позволяет указать, какая системная папка или библиотека Windows 8 должна быть изначально открыта в этом диалоге. В качестве значения этого свойства следует использовать один из следующих элементов перечисления `Windows.Storage.Pickers.PickerLocationId`:

- `documentsLibrary` — библиотека Документы;
- `computerFolder` — системная папка Компьютер;
- `desktop` — системная папка Рабочий стол;
- `downloads` — системная папка Загрузки;
- `homeGroup` — системная папка Домашняя сетевая группа;
- `musicLibrary` — библиотека Музыка;
- `picturesLibrary` — библиотека Изображения;
- `videosLibrary` — библиотека Видео.

```
oFOP.suggestedStartLocation =
Windows.Storage.Pickers.PickerLocationId.videosLibrary;
```

Изначально выводим в диалоге библиотеку Видео.

Свойство `fileTypeFilter` позволяет указать расширения файлов, которые будут перечислены в диалоге открытия и, соответственно, могут быть выбраны пользователем. Значение этого свойства — особый массив, элементы которого представляют собой строки, хранящие расширения файлов.

Чтобы указать нужные нам расширения, удобнее всего использовать метод `replaceAll` данного массива. В качестве параметра он принимает массив строк, хранящих расширения файлов, и заменяет им содержимое текущего массива. Не забываем при этом, что расширения файлов должны быть указаны с начальными точками.

```
oFOP.fileTypeFilter.replaceAll([".avi", ".mp4", ".wmv"]);
```

Указываем для нашего диалога открытия файла допустимые расширения avi, mp4 и wmv.

Свойство `viewMode` позволяет указать, в каком виде в диалоге открытия файла будут перечислены позиции, соответствующие отдельным файлам. Значение этого свойства должно представлять собой один из элементов перечисления `Windows.Storage.Pickers.PickerViewMode`:

- `list` — каждая позиция будет содержать миниатюру и имя файла;
- `thumbnail` — каждая позиция будет содержать только миниатюру файла.

```
oFOP.viewMode = Windows.Storage.Pickers.PickerViewMode.list;
```

Пусть каждая позиция в диалоге открытия файла содержит и миниатюру, и имя файла. По мнению автора, такие позиции удобнее в большинстве случаев.

Свойство commitButtonText позволяет указать надпись для кнопки открытия файла. Надпись должна быть задана в виде строки. По умолчанию эта кнопка имеет надпись **Открыть** (Open).

```
oFOP.commitButtonText = "Воспроизвести";
```

Вывод диалога и получение выбранного в нем файла

Подготовив диалог открытия файла, мы можем вывести его на экран и получить файл, который пользователь в нем выбрал. Для этого применяются два метода объекта Window.Storage.Pickers.FileOpenPicker, которые мы сейчас рассмотрим.

Метод pickSingleFileAsync выводит диалог на экран и дает возможность пользователю выбрать в нем один файл. Этот метод не принимает параметров и возвращает в качестве результата обязательство. Для этого обязательства мы с помощью метода then зададим функцию, которая будет выполнена после того, как пользователь выберет файл. В качестве единственного параметра эта функция получит:

- экземпляр объекта Windows.Storage.StorageFile, если пользователь выбрал файл в диалоге;
- значение null, если пользователь отказался от выбора файла.

Объект Windows.Storage.StorageFile представляет любой файл.

```
oFOP.pickSingleFileAsync().then(function(file) {
  if (file) {
    //Файл был выбран
  } else {
    //Файл не был выбран
  }
});
```

Чтобы проверить, был ли выбран файл, мы используем условное выражение, в качестве условия которого подставим полученный функцией результат. Если он представляет собой экземпляр объекта, платформа Metro автоматически преобразует его в значение true, и условие выполнится. Если же он представляет собой null, данное значение будет преобразовано в false, и условие не выполнится. (Подробнее о преобразовании данных различных типов говорилось в *главе 4*.)

Метод pickMultipleFilesAsync выводит диалог на экран и позволяет пользователю выбрать в нем сразу несколько файлов. Этот метод также не принимает параметров и также возвращает в качестве результата обязательство. В качестве единственного параметра функция, которую мы укажем для этого обязательства с помощью метода then, получит экземпляр объекта-коллекции, хранящего список всех выбранных пользователем файлов.

Этот объект-коллекция поддерживает свойство size, которое возвращает количество выбранных пользователем файлов в виде целого числа. А получить доступ к отдельному элементу коллекции можно тем же способом, что применяется для доступа к элементам обычного массива.

Отметим, что, даже если пользователь не выбрал в диалоге ни одного файла, функция все равно получит экземпляр этого объекта-коллекции. Просто в данном случае он не будет содержать файлов, и его свойство `size` вернет значение 0.

```
var arrFiles = [];
oFOP.pickMultipleFileAsync().then(function(files) {
  if (files.size > 0) {
    for (var i = 0; i < files.size; i++) {
      arrFiles.push(files[i]);
    }
  }
});
```

Получаем список всех выбранных пользователем файлов и помещаем их в массив `arrFiles`.

Вывод выбранного пользователем файла на экран

Многие приложения выводят выбранный пользователем файл на экран в специально предназначенном для этого элементе интерфейса. Например, графический файл выводится в элементе графического изображения (теге `<img>`), аудиофайл воспроизводится в элементе аудиопроигрывателя, а видеофайл — в элементе видеопроигрывателя. Это нам знакомо еще по *главе 2* и по нашему первому Metro-приложению.

Однако здесь есть одна тонкость. Если мы просто присвоим выбранный пользователем, скажем, видеофайл свойству `src` видеопроигрывателя, приложение завершится по ошибке. То же самое будет в случае графического изображения и аудиофайла.

Дело в том, что диалог открытия выдает выбранный в нем файл в виде экземпляра объекта `Windows.Storage.StorageFile`, который не поддерживается ни одним из элементов интерфейса, предоставляемых платформой Metro; ни элемент графического изображения, ни аудио-, ни видеопроигрыватель его просто не "поймут".

Нам придется преобразовать экземпляр объекта `Windows.Storage.StorageFile` в вид, поддерживаемый данными элементами. В этом нам поможет метод `createObjectURL` объекта `URL`. (Объект `URL` предоставляет различные инструменты для работы с файлами и ссылками на них. Единственный его экземпляр создается самой платформой Metro и доступен через переменную `URL`.)

Метод `createObjectURL` принимает в качестве единственного параметра файл в виде экземпляра объекта `Windows.Storage.StorageFile` и возвращает представляющий его экземпляр объекта `Blob`. А объект `Blob` можно рассматривать как массив данных, хранящий содержимое данного файла и представленный в виде, который поддерживается всеми элементами интерфейса. То, что нам и нужно!

```
oFOP.pickSingleFileAsync().then(function(file) {
  if (file) {
    vidMain.src = URL.createObjectURL(file);
  }
});
```

Получаем выбранный пользователем видеофайл, преобразуем его в экземпляр объекта `Blob`, присваиваем свойству `src` видеопроигрывателя и смотрим фильм.

Заметим, что данное преобразование необходимо только для файлов, которые мы хотим вывести на экран в специально созданных для этого элементах интерфейса. Если же мы собираемся обрабатывать данный файл в логике приложения, без вывода его на экран, выполнять это преобразование не нужно.

Чтение из файла

Итак, выбранный пользователем файл — экземпляр объекта `Windows.Storage.StorageFile` — мы получили. Давайте теперь прочитаем его содержимое.

Открытие файла для чтения

Прежде чем начать чтение из файла, нам следует его открыть. Для этого объект `Windows.Storage.StorageFile` предоставляет нам метод `openAsync`, запускающий процесс открытия файла.

В качестве единственного параметра он принимает один из элементов перечисления `Windows.Storage.FileAccessMode`, указывающий режим открытия файла. Поскольку мы хотим открыть файл для чтения, то используем элемент `read` этого перечисления.

Метод `openAsync` возвращает в качестве результата обязательство. Для этого обязательства мы, как обычно, укажем в вызове метода `then` функцию, которая будет выполнена по завершении открытия файла.

Данная функция в качестве единственного параметра получит так называемый *поток* — структуру, представляющую все данные, что хранятся в открытом файле. Такой поток является экземпляром объекта `Windows.Storage.Streams.InMemoryRandomAccessStream`.

В теле этой функции и будут выполняться дальнейшие операции, связанные с чтением файла.

```
oFOP.pickSingleFileAsync().then(function(file) {
  if (file) {
    file.openAsync(Windows.Storage.FileAccessMode.read).
    ↳then(function(stream) {
      //Поток получен
    });
  }
});
```

Объект `Windows.Storage.Streams.InMemoryRandomAccessStream`, представляющий поток, поддерживает свойство `size`. Оно возвращает размер потока (а фактически — самого файла) в виде целого числа в байтах. Эта величина нам впоследствии пригодится.

Получение потока чтения и читателя данных

Что ж, поток, представляющий все хранящиеся в файле данные, мы получили. Однако этот поток, называемый *общим*, не позволяет нам прочитать данные. Поэтому следующим нашим шагом будет выделение из него другого потока, специально предназначенного для чтения данных, — *потока чтения*.

Сделать это мы можем вызовом метода `getInputStreamAt` объекта `Windows.Storage.Streams.InMemoryRandomAccessStream` (который, как мы помним, представляет общий поток). Формат его вызова таков:

```
<общий поток>.getInputStreamAt(<начальная позиция>)
```

В качестве единственного параметра этот метод принимает целочисленное значение, указывающее позицию в получаемом потоке чтения, с которой начнется чтение данных. Фактически это номер байта, с которого потом начнется считывание.

Метод `getInputStreamAt` возвращает в качестве результата то, что нам нужно, — поток чтения. Он представляет собой экземпляр особого объекта.

```
var oReadStream = stream.getInputStreamAt(0);
```

Получаем поток чтения с изначальной позицией, равной нулю. В результате считывание данных из такого потока начнется с самого его начала.

На основе полученного потока чтения мы сформируем вспомогательную структуру данных, называемую *читателем*. Она как раз и предоставит нам все необходимые инструменты для чтения данных из файла.

Читатель представляется объектом `Windows.Storage.Streams.DataReader`. Нам следует создать экземпляр этого объекта, передав ему в качестве параметра полученный ранее поток чтения.

```
var oDataReader = new Windows.Storage.Streams.DataReader(oReadStream);
```

Получив читателя, мы вызовем у него метод `loadAsync`, который запустит процесс загрузки данных из файла в оперативную память. Единственным параметром данному методу передается объем считываемых данных в виде целого числа; обычно ему передают размер общего потока — значение его свойства `size`, — считывая тем самым все содержимое файла.

Метод `loadAsync` возвращает в качестве результата обязательство. Мы привяжем к нему с помощью метода `then` не принимающую параметров функцию, которая выполнится после окончания загрузки данных из файла, когда данные будут готовы к собственно считыванию. В теле этой функции мы и будем их читать.

```
oDataReader.loadAsync(stream.size).then(function() {
  //Выполняем чтение данных
});
```

Собственно чтение из файла

Уф!.. Пожалуй, разработчики платформы Metro здесь несколько перемудрили... Слишком многое нужно сделать, чтобы наконец-то получить возможность прочитать содержимое файла.

Чтение текстовых файлов

Проще всего прочитать текстовый файл. Для этого достаточно вызвать метод `readString` читателя, передав ему в качестве единственного параметра длину считываемой строки. Если нам следует прочитать все содержимое текстового файла, мы передадим этому методу полученный из свойства `size` размер общего потока. Метод `readString` вернет в качестве результата строку, считанную из файла.

```
<div id="divOutput"></div>
. . .
var oReadStream, oDataReader;
var divOutput = document.getElementById("divOutput");
oFOP.pickSingleFileAsync().then(function(file) {
  if (file) {
    file.openAsync(Windows.Storage.FileAccessMode.read).
    ↳then(function(stream) {
      oReadStream = stream.getInputStreamAt(0);
      oDataReader = new Windows.Storage.Streams.DataReader(oReadStream);
      oDataReader.loadAsync(stream.size).then(function() {
        divOutput.textContent = oDataReader.readString(stream.size);
        //Закрытие читателя (см. далее)
      });
    });
  }
});
```

Открываем текстовый файл, выбранный пользователем, и выводим его содержимое в блоке `divOutput`.

Чтение двоичных файлов

Еще Metro дает нам возможность читать содержимое двоичных файлов и предоставляет для этого богатые возможности.

Прежде всего, это набор методов объекта `Windows.Storage.Streams.DataReader`, перечисленных далее. Все эти методы не принимают параметров, а в качестве результата возвращают прочитанное из файла значение соответствующего типа:

- `readInt32` — читает целочисленное значение;
- `readDouble` — читает числовое значение с плавающей точкой;
- `readBoolean` — читает логическое значение;
- `readDateTime` — читает значение даты и времени (экземпляр объекта `Date`);
- `readByte` — читает байт и возвращает его в качестве числового значения;

- `readGuid` — читает значение глобального уникального идентификатора (GUID) и возвращает его в виде строки.

Для чтения строк можно использовать уже знакомый нам метод `readString`. Единственное, что следует помнить в этом случае, — мы должны точно знать длину читаемой строки.

Все перечисленные ранее методы не только читают из файла соответствующее значение, но и подготавливают поток для чтения следующего записанного в нем значения. Так что если мы будем вызывать эти методы последовательно, то получим все значения, записанные в файле друг за другом.

```
var oReadStream, oDataReader, iValue1, iValue2, fValue3;
oFOP.pickSingleFileAsync().then(function(file) {
  if (file) {
    file.openAsync(Windows.Storage.FileAccessMode.read).
    ↳then(function(stream) {
      oReadStream = stream.getInputStreamAt(0);
      oDataReader = new Windows.Storage.Streams.DataReader(oReadStream);
      oDataReader.loadAsync(stream.size).then(function() {
        iValue1 = oDataReader.readInt32();
        iValue2 = oDataReader.readInt32();
        fValue3 = oDataReader.readDouble();
        //Закрытие читателя (см. далее)
      });
    });
  }
});
```

Читаем из выбранного пользователем двоичного файла последовательно записанные в нем два целых числа и одно число с плавающей точкой.

Метод `readBytes` пригодится в случае, если мы захотим прочитать все содержимое файла. Он принимает в качестве единственного параметра пустой массив и заполняет его целыми числами, представляющими отдельные байты информации, что хранится в файле. Результата он не возвращает.

```
var arrBytes = [];
oDataReader.readBytes(arrBytes);
```

Получаем в массиве `arrBytes` все содержимое файла.

Еще нам должно пригодиться свойство `unconsumedBufferLength`. Оно возвращает длину еще не прочитанной из файла информации в байтах в виде целого числа.

```
var oReadStream, oDataReader, arrValues;
oFOP.pickSingleFileAsync().then(function(file) {
  if (file) {
    file.openAsync(Windows.Storage.FileAccessMode.read).
    ↳then(function(stream) {
      oReadStream = stream.getInputStreamAt(0);
      oDataReader = new Windows.Storage.Streams.DataReader(oReadStream);
```

```
      oDataReader.loadAsync(stream.size).then(function() {
        while (oDataReader.unconsumedBufferLength > 0) {
          arrValues.push(oDataReader.readInt32());
        }
        //Закрытие читателя (см. далее)
      });
    });
  }
});
```

Читаем из двоичного файла все хранящиеся там целочисленные значения и помещаем их в массив `arrValues`.

Закрытие читателя

Закончив все операции чтения из файла, мы обязательно должны закрыть читателя. В результате открытый нами файл будет закрыт.

Закрытие читателя выполняется вызовом его метода `close`, который не принимает параметров и не возвращает результата.

```
oDataReader.close();
```

Запись в файл

Что ж, читать из файла мы научились. На очереди — запись в файл. Выполняется она примерно так же, как и чтение.

Диалог сохранения файла

Первое, что нам потребуется сделать, — выяснить имя и месторасположение файла, в котором пользователь хочет сохранить информацию.

Для выбора сохраняемого файла платформа Metro также предоставляет специальный диалог, который носит название *диалога сохранения файла*. Он также появляется на экране, перекрывая интерфейс приложения, и содержит список файлов, уже имеющихся в указанной папке, и кнопки **Сохранить** (Save) и **Отмена** (Cancel).

Диалог сохранения файла представляется объектом `Window.Storage.Pickers.FileSavePicker`. Нам следует создать экземпляр этого объекта и присвоить его переменной.

```
var oFSP = new Windows.Storage.Pickers.FileSavePicker();
```

Далее мы зададим для диалога сохранения файла нужные параметры, воспользовавшись различными свойствами объекта `Window.Storage.Pickers.FileSavePicker`.

Свойства `suggestedStartLocation` и `commitButtonText` уже нам знакомы. Они задают, соответственно, папку, содержимое которой изначально будет выведено в диалоге сохранения файла, и надпись для кнопки сохранения.

```
oFSP.suggestedStartLocation =
Windows.Storage.Pickers.PickerLocationId.documentsLibrary;
```

Свойство `fileTypeChoices` хранит массив расширений, которые будут перечислены в диалоге сохранения и которые пользователь сможет выбрать для сохраняемого файла. Но этот массив несколько иного рода, чем тот, что присутствует в диалоге открытия файла. Вместе с собственно расширениями он хранит наименования соответствующих им типов файлов.

Изначально этот массив пуст, так что мы должны сразу же его заполнить. Для этого следует использовать метод `insert` данного массива, который добавляет в него новый элемент.

```
<массив расширений>.insert(<наименование типа файлов>, <расширения>)
```

Первым параметром этому методу передается строка с наименованием типа файлов, а вторым — массив строк, каждая из которых хранит одно из расширений, соответствующих этому типу. Расширения должны быть указаны с начальными точками.

```
oFSP.fileTypeChoices.insert("Текстовые файлы", [".txt"]);
oFSP.fileTypeChoices.insert("Служебные файлы", [".ini", ".log"]);
```

Свойство `defaultFileExtension` служит для указания расширения файла по умолчанию; оно будет добавлено к имени файла, если пользователь не укажет расширение сам. Расширение задается в виде строки с начальной точкой.

```
oFSP.defaultFileExtension = ".txt";
```

Свойство `suggestedFileName` задает имя файла, которое будет подставлено в диалог сохранения изначально. Оно указывается в виде строки и не должно включать расширение.

```
oFSP.suggestedFileName = "Текстовый файл";
```

Метод `pickSaveFileAsync` выводит диалог на экран. Он не принимает параметров и возвращает в качестве результата обязательство. Для этого обязательства мы с помощью метода `then` зададим функцию, которая будет выполнена после того, как пользователь укажет файл и получит в качестве единственного параметра:

- экземпляр объекта `Windows.Storage.StorageFile`, если пользователь указал файл для сохранения;
- значение `null`, если пользователь отказался от сохранения файла.

```
oFSP.pickSaveFileAsync().then(function(file) {
  if (file) {
    //Файл был указан
  } else {
    //Файл не был указан
  }
});
```

Открытие файла для записи, получение потока записи и писателя данных

Получив файл, указанный пользователем, мы можем открыть его для чтения. Выполняется это с помощью уже знакомого нам метода `openAsync` объекта `Windows.Storage.StorageFile`. Только в этом случае нам следует указать в качестве его параметра один из следующих элементов перечисления `Windows.Storage.FileAccessMode`:

- `readWrite` — файл будет доступен и для чтения, и для записи. Записываемые в файл данные будут доступны только после завершения записи;
- `readWriteUnsafe` — файл будет доступен и для чтения, и для записи. Записываемые в файл данные будут доступны сразу же после выполнения операций записи;
- `readWriteNoCopyOnWrite` — файл будет доступен и для чтения, и для записи. Записываемые в файл данные будут доступны только после завершения записи. Поток записи, что мы получим, будет изначально пуст.

Элемент `readWrite` указывают, если требуется дописать данные в файл. Если же мы хотим полностью заменить содержимое файла новым, то лучшим вариантом будет указание элемента `readWriteNoCopyOnWrite`.

```
oFSP.pickSaveFileAsync().then(function(file) {
  if (file) {
    file.openAsync(Windows.Storage.FileAccessMode.readWriteNoCopyOnWrite).
    ↳then(function(stream) {
      //Поток получен
    });
  }
});
```

Получив общий поток, мы выделим из него *поток записи*, представляющий записываемые данные. Для этого мы воспользуемся методом `getOutputStreamAt` объекта `Windows.Storage.Streams.InMemoryRandomAccessStream`. Формат его записи такой же, как у знакомого нам метода `getInputStreamAt`.

Если мы укажем 0 в качестве параметра метода `getOutputStreamAt`, запись начнется с начала файла, следовательно, файл будет перезаписан. Если мы укажем общий размер потока (значение его свойства `size`), вновь записанные данные будут добавлены в конец файла.

Если же мы при открытии файла в качестве параметра метода `openAsync` указали элемент `readWriteNoCopyOnWrite` перечисления `Windows.Storage.FileAccessMode`, нет разницы, какой параметр указать в вызове метода `getOutputStreamAt`. Как мы уже знаем, в таком случае поток записи будет пуст, и файл в любом случае окажется перезаписанным.

```
var oReadStream = oStream.getInputStreamAt(0);
```

Когда мы занимались чтением из файла, то после получения потока чтения получали читателя данных. В случае записи в файл нам следует получить *писателя*, посредством которого мы и будем выполнять сами операции записи.

Писатель данных представляется объектом `Windows.Storage.Streams.DataWriter`. Нам следует создать экземпляр этого объекта, передав ему поток записи в качестве параметра.

```
var oDataWriter = new Windows.Storage.Streams.DataWriter(oWriteStream);
```

Собственно запись в файл

Теперь мы можем записать в файл любую информацию — как текстовую, так и двоичную.

Запись строковых данных

Как и в случае чтения, проще всего выполнить запись строковых данных, создав тем самым текстовый файл. Мы применим для этого метод `writeString` объекта `Windows.Storage.Streams.DataWriter`, передав ему в качестве единственного параметра саму записываемую строку. В качестве результата этот метод вернет длину записанной строки в байтах (не в символах!) в виде целого числа.

```
<textarea id="txtInput"></textarea>
. . .
var oWriteStream, oDataWriter;
var txtInput = document.getElementById("txtInput");
oFSP.pickSaveFileAsync().then(function(file) {
  if (file) {
    file.openAsync(Windows.Storage.FileAccessMode.readWriteNoCopyOnWrite).
    ↳then(function(stream) {
      oWriteStream = stream.getOutputStreamAt(0);
      oDataWriter = new Windows.Storage.Streams.DataWriter(oWriteStream);
      oDataWriter.writeString(txtInput.value);
      //Выполняем завершение записи (см. далее)
    });
  }
});
```

Получаем файл, выбранный пользователем для сохранения, открываем его и записываем туда текст, введенный в область редактирования `txtInput`.

Еще нам может пригодиться метод `measureString` объекта `Windows.Storage.Streams.DataWriter`. Он принимает в качестве единственного параметра строку и возвращает ее длину в байтах в виде целого числа.

Запись двоичных данных

Раз уж платформа Metro позволяет нам читать из файлов двоичные данные, следовательно, она должна позволять и записывать их. Так и есть.

Для записи двоичных данных применяются методы объекта `Windows.Storage.Streams.DataWriter`, описанные далее. Они принимают в качестве единственного параметра значение, которое следует записать в файл, и не возвращают результата:

- `writeInt32` — записывает целочисленное значение;
- `writeDouble` — записывает числовое значение с плавающей точкой;
- `writeBoolean` — записывает логическое значение;
- `writeDateTime` — записывает значение даты и времени (экземпляр объекта `Date`);
- `writeByte` — записывает полученное числовое значение в виде байта;
- `writeGuid` — записывает полученное в виде строки значение глобального уникального идентификатора (GUID).

Для записи строк мы используем уже знакомый нам метод `writeString`.

Все эти методы, помимо записи переданного им значения в файл, подготавливают поток для записи следующего значения. Так что если мы будем вызывать эти методы последовательно, то все указанные в их вызовах значения будут записаны в файл друг за другом.

```
var oWriteStream, oDataWriter;
oFSP.pickSaveFileAsync().then(function(file) {
  if (file) {
    file.openAsync(
    ↳Windows.Storage.FileAccessMode.readWriteNoCopyOnWrite).
    ↳then(function(stream) {
      oWriteStream = stream.getOutputStreamAt(0);
      oDataWriter = new Windows.Storage.Streams.DataWriter(oWriteStream);
      oDataWriter.writeInt32(20);
      oDataWriter.writeDouble(1.34);
      oDataWriter.writeDateTime(new Date());
      //Выполняем завершение записи (см. далее)
    });
  }
});
```

Записываем в выбранный пользователем файл последовательно целое число, число с плавающей точкой и значение текущих даты и времени.

Метод `writeBytes` позволяет записать в файл содержимое массива чисел, которые будут трактоваться как байты, т. е. одномоментно создать все содержимое двоичного файла. Он принимает в качестве единственного параметра массив с записываемыми данными и не возвращает результата.

```
var arrBytes = [32, 45, 0, 128, 94, 89, 7, 22];
oDataWriter.writeBytes(arrBytes);
```

Завершение операции записи и закрытие читателя

Записав в файл все нужные данные, мы обязательно должны выполнить операцию завершения записи. Если этого не сделать, данные не будут записаны в файл и впоследствии потеряются.

Сначала мы уведомим о завершении записи писателя, вызвав его метод `storeAsync`. Этот метод запускает процесс передачи всех записанных данных потоку записи. Он не принимает параметров и возвращает в качестве результата обязательство, для которого мы с помощью метода `then` укажем не принимающую параметров функцию, что будет выполнена после завершения передачи данных.

В теле этой функции мы вызовем метод `flushAsync` потока записи. Он запустит собственно процесс переноса записанных данных из оперативной памяти в файл. Этот метод также не принимает параметров и возвращает в качестве результата обязательство.

Для этого обязательства мы с помощью метода `then` укажем не принимающую параметров функцию, что будет выполнена после завершения записи данных. В теле этой функции мы вызовем метод `close` писателя, который выполнит закрытие как самого писателя, так и открытого нами файла.

```
oDataWriter.storeAsync().then(function() {
  oWriteStream.flushAsync().then(function() {
    oDataWriter.close();
  });
});
```

Пример: простейший текстовый редактор

Чтобы применить полученные знания на практике, давайте создадим простейшее приложение текстового редактора. Оно позволит пользователю создавать, открывать, править и сохранять обычные текстовые файлы.

Создадим в Visual Studio новый проект `TextEditor`. Откроем файл default.html и введем в тег `<body>` такой код:

```
<textarea id="txtEdit"></textarea>
<div id="divAppBar" data-win-control="WinJS.UI.AppBar"
data-win-options="{placement: 'top'}">
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnNew', label: 'Создать', icon: 'document',
  ↳section: 'global'}"></button>
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnOpen', label: 'Открыть', icon: 'openfile',
  ↳section: 'global'}"></button>
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnSave', label: 'Сохранить',
  ↳icon: 'savelocal', section: 'global'}"></button>
</div>
```

Мы создаем область редактирования, в которой и будет выводиться содержимое текстового файла, и верхнюю панель инструментов с кнопками **Создать**, **Открыть** и **Сохранить.**

Откроем файл default.css и создадим в нем стили, перечисленные далее.

```
#txtEdit {
  margin-top: 68px;
  width: 100%;
  height: calc(100% — 68px);
}
```

Задаем для области редактирования внешний отступ сверху, равный 68 пикселам. Он нужен для того, чтобы пользователь смог вытащить панель инструментов, проведя по экрану пальцем. (Проведение пальца по области редактирования вызовет выделение ее содержимого.) Кроме этого, верхний отступ исключит перекрытие области редактирования выведенной на экран панелью инструментов. Напоследок мы растягиваем область редактирования на всю ширину и высоту экрана за вычетом установленного ранее значения отступа сверху.

Далее переключимся на файл default.js и сразу же добавим код, объявляющий необходимую переменную:

```
var txtEdit;
```

Напишем код инициализации приложения:

```
document.addEventListener("DOMContentLoaded", function () {
  WinJS.UI.processAll().then(function() {
    txtEdit = document.getElementById("txtEdit");
    var ctrAppBar = document.getElementById("divAppBar").winControl;
    var btnNew = ctrAppBar.getCommandById("btnNew");
    var btnOpen = ctrAppBar.getCommandById("btnOpen");
    var btnSave = ctrAppBar.getCommandById("btnSave");
    btnNew.addEventListener("click", btnNewClick);
    btnOpen.addEventListener("click", btnOpenClick);
    btnSave.addEventListener("click", btnSaveClick);
  });
});
```

Реализуем создание нового текстового документа:

```
function btnNewClick() {
  txtEdit.value = "";
}
```

Для этого достаточно просто очистить область редактирования.

Реализуем открытие файла:

```
function btnOpenClick() {
  var oFOP = new Windows.Storage.Pickers.FileOpenPicker();
  var oReadStream, oDataReader;
  oFOP.suggestedStartLocation =
```

```
  Windows.Storage.Pickers.PickerLocationId.documentsLibrary;
  oFOP.fileTypeFilter.replaceAll([".txt"]);
  oFOP.viewMode = Windows.Storage.Pickers.PickerViewMode.list;
  oFOP.pickSingleFileAsync().then(function(file) {
    if (file) {
      file.openAsync(Windows.Storage.FileAccessMode.read).
      ↳then(function(stream) {
        oReadStream = stream.getInputStreamAt(0);
        oDataReader = new
        ↳Windows.Storage.Streams.DataReader(oReadStream);
        oDataReader.loadAsync(stream.size).then(function() {
          txtEdit.value = oDataReader.readString(stream.size);
          oDataReader.close();
        });
      });
    }
  });
}
```

Осталось реализовать сохранение файла:

```
function btnSaveClick() {
  var oFSP = new Windows.Storage.Pickers.FileSavePicker();
  var oWriteStream, oDataWriter;
  oFSP.suggestedStartLocation =
  Windows.Storage.Pickers.PickerLocationId.documentsLibrary;
  oFSP.fileTypeChoices.insert("Текстовые файлы", [".txt"]);
  oFSP.suggestedFileName = "Текстовый файл";
  oFSP.defaultFileExtension = ".txt";
  oFSP.pickSaveFileAsync().then(function(file) {
    if (file) {
      file.openAsync(
      ↳Windows.Storage.FileAccessMode.readWriteNoCopyOnWrite).
      ↳then(function(stream) {
        oWriteStream = stream.getOutputStreamAt(0);
        oDataWriter = new
        Windows.Storage.Streams.DataWriter(oWriteStream);
        oDataWriter.writeString(txtEdit.value);
        oDataWriter.storeAsync().then(function() {
          oWriteStream.flushAsync().then(function() {
            oDataWriter.close();
          });
        });
      });
    }
  });
}
```

Здесь все нам уже знакомо.

Сохраним все исправленные файлы и запустим приложение на выполнение. Попробуем открыть в нем какой-либо файл, исправить его и сохранить, а потом — создать и сохранить новый текстовый документ.

Конечно, наше приложение еще трудно назвать полноценным текстовым редактором; скорее, это прототип, написанный в образовательных целях. Но ведь мы пока еще только учимся...

Получение списка файлов и папок

Следующее, что мы изучим, будет получение списка всех файлов и папок, которые хранятся в указанной нами папке. Выполнить такую операцию средствами платформы Metro проще простого! (Чего не скажешь о чтении из файлов и записи в них...)

Замечания о программном доступе к библиотекам. Права приложения

Но сначала рассмотрим один очень важный вопрос. Он касается программного доступа к библиотекам Windows и реализованной в платформе Metro системе безопасности.

Дело в том, что Metro изначально не позволяет приложениям получать доступ к персональным данным пользователя, в частности, содержимому системных библиотек. То есть если наше приложение попытается программно обратиться к содержимому любой библиотеки, попытка не увенчается успехом, и выполнение приложения будет прервано по ошибке.

> ***На заметку***
>
> Кстати, то же самое произойдет, если приложение попытается прочитать содержимое подключаемого носителя (например, флэш-диска), получить данные со встроенных датчиков (GPS, электронного компаса, акселерометра и др.) или изображение с фото- или видеокамеры. Впрочем, более обстоятельный разговор об этом будет впереди.

Все это сделано для того, чтобы не дать вредоносным приложениям похитить персональные данные пользователя или скомпрометировать его иным способом. Хотя все Metro-приложения перед публикацией в магазине Windows Store проходят тщательную проверку, лишний раз обезопаситься не помешает.

Чтобы все-таки получить доступ к библиотекам или устройствам, приложение должно обладать определенными правами, носящими название *прав приложения*. Так, если наше приложение должно программно работать с библиотекой Изображения, оно должно иметь право на доступ к этой библиотеке.

Права приложения задаются на этапе его разработки прямо в среде Visual Studio. Найдем в списке панели **SOLUTION EXPLORER** (подробнее о ней говорилось в *главе 2*) файл package.appxmanifest и дважды щелкнем на нем. В главном окне появится еще одно окно документа, в котором наглядно, в графическом виде, будут

представлены все параметры открытого нами проекта и, в частности, права приложения.

В этом окне документа будет присутствовать набор вкладок, каждая из которых служит для указания определенной группы параметров. Права приложения указываются на вкладке **Capabilities**, поэтому сразу переключимся на нее.

Здесь мы увидим список **Capabilities**, каждый пункт которого — флажок (рис. 16.1). Эти пункты представляют собой доступные в платформе Metro права приложения; установленный флажок означает, что приложение обладает соответствующим правом, сброшенный — отменяет данное право.

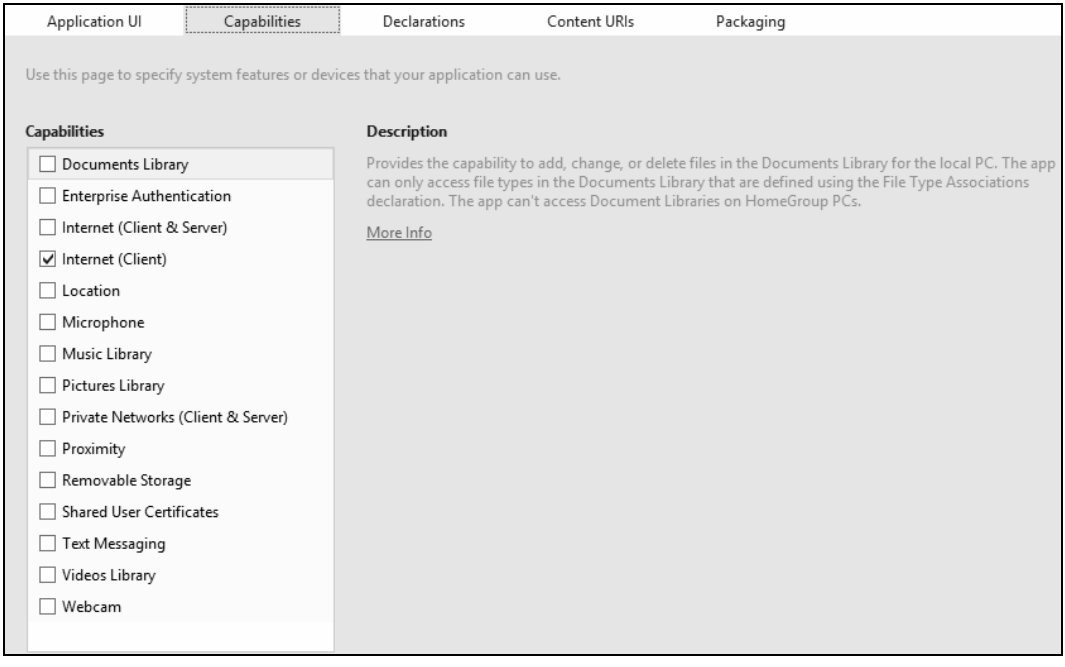


Рис. 16.1. Содержимое вкладки **Capabilities** окна для задания параметров приложения

На данном этапе нам будут интересны следующие пункты списка **Capabilities** и соответствующие им права:

- **Document Library** — доступ к библиотеке Документы, к файлам тех типов, которые указаны в параметрах приложения;
- **Music Library** — доступ к библиотеке Музыка;
- **Pictures Library** — доступ к библиотеке Изображения;
- **Private Networks (Client & Server)** — доступ к домашней сетевой группе;
- **Videos Library** — доступ к библиотеке Видео.

Чтобы дать приложению какое-либо из этих прав, достаточно установить нужный флажок.

Если разрешен доступ к библиотекам Музыка, Изображения или Видео, то доступ к домашней сетевой группе предоставляется автоматически, и специально давать приложению такое право не требуется.

Если разрешен доступ к библиотеке Документы, обязательно следует указать типы файлов, которые сможет открывать и создавать приложение. При этом доступ к файлам неуказанных типов будет невозможен. (Библиотек Музыка, Изображения и Видео это не касается; если приложение имеет к ним доступ, оно может обрабатывать хранящиеся там файлы любых типов.)

Задать сведения о доступных типах файлов можно на вкладке **Declarations** (рис. 16.2). Переключимся на нее.

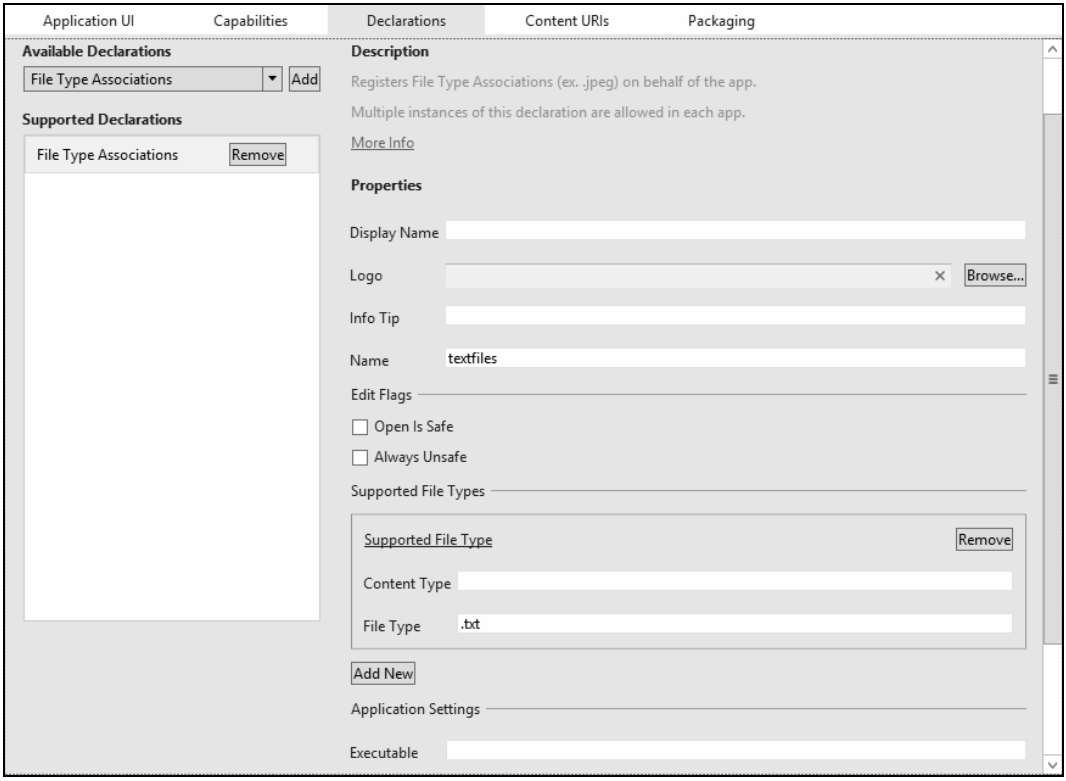


Рис. 16.2. Содержимое вкладки **Declarations** окна для задания параметров приложения

Здесь мы сразу увидим список **Supported Declarations**, в котором перечисляются заданные нами типы файлов. Изначально он пуст.

Чтобы создать новый тип файлов, мы выберем в раскрывающемся списке **Available Declarations** пункт **File Type Associations** и нажмем кнопку **Add**. В списке **Supported Declarations** появится пункт **File Type Associations**, представляющий вновь созданный тип. Выберем его.

В поле ввода **Name**, расположенном в группе элементов управления **Properties**, введем имя созданного типа файлов. Это имя должно содержать только строчные

(маленькие) латинские буквы, цифры и знаки подчеркивания. Например, для текстовых файлов мы зададим имя `textfiles`.

Ниже расположена группа элементов управления **Supported File Types**, в которой, в свою очередь, находится набор групп, описывающих конкретные расширения файлов, что относятся к данному типу. Расширение файла вводится в поле ввода **File Type**, обязательно с начальной точкой, например `.txt`.

Чтобы добавить в этот набор новую группу — новое расширение файлов, следует нажать кнопку **Add New**. Чтобы удалить ненужную группу и, соответственно, расширение, достаточно нажать присутствующую в ее заголовке кнопку **Remove**.

Мы можем добавить в список **Supported Declarations** сколько угодно типов файлов и указать для каждого из них сколько угодно расширений. Так, мы можем дать приложению текстового редактора поддержку еще одного типа файлов — служебных — с расширениями ini и log.

Закончив задание прав приложения, следует закрыть окно документа, в котором выводятся его параметры, и сохранить его содержимое.

> ***Внимание!***
>
> Даже если приложение имеет все нужные права, при первой его попытке получить доступ к библиотеке или устройству платформа Metro запросит разрешение пользователя на это, выведя соответствующее предупреждение. Причем следует иметь в виду, что пользователь может и отказаться.
>
> Если приложение использует диалоги открытия и сохранения файла, его права на доступ к библиотекам указывать необязательно. Платформа Metro в этом случае будет исходить из того, что пользователь сам предоставил доступ к выбранному им в диалоге файлу. Кстати, это же касается диалога выбора папки, о котором скоро пойдет речь.

Получение доступа к библиотекам

Получить программный доступ к системным библиотекам Windows очень просто. Достаточно воспользоваться перечислением `Windows.Storage.KnownFolders`, содержащим следующие элементы:

- `documentsLibrary` — библиотека Документы;
- `homeGroup` — домашняя сетевая группа;
- `musicLibrary` — библиотека Музыка;
- `picturesLibrary` — библиотека Изображения;
- `videosLibrary` — библиотека Видео.

Каждый из этих элементов хранит основную папку соответствующей библиотеки. Она, как и любая другая папка в платформе Metro, представляется в виде экземпляра объекта `Windows.Storage.StorageFolder`.

```
var libImages = Windows.Storage.KnownFolders.picturesLibrary;
```

Получаем библиотеку Изображения.

Диалог выбора папки

Немногим сложнее получить доступ к произвольной папке, выбранной пользователем. Нам поможет стандартный *диалог выбора папки* платформы Metro.

Диалог выбора папки представляется объектом `Window.Storage.Pickers.FolderPicker`. Следовательно, мы должны сначала создать экземпляр этого объекта и присвоить его какой-либо переменной.

```
var oFP = new Windows.Storage.Pickers.FolderPicker();
```

Объект `Window.Storage.Pickers.FolderPicker` поддерживает уже знакомые нам свойства `suggestedStartLocation`, `fileTypeFilter`, `viewMode` и `commitButtonText`. С их помощью мы зададим параметры диалога.

```
oFP.suggestedStartLocation =
Windows.Storage.Pickers.PickerLocationId.picturesLibrary;
oFP.fileTypeFilter.replaceAll([".gif", ".jpg", ".png"]);
oFP.viewMode = Windows.Storage.Pickers.PickerViewMode.thumbnail;
```

Для вывода диалога и получения выбранной в нем папки мы вызовем у него метод `pickSingleFolderAsync`. Этот метод не принимает параметров и возвращает в качестве результата обязательство, для которого мы с помощью метода `then` зададим функцию. Данная функция будет выполнена после того, как пользователь выберет папку и в качестве единственного параметра получит:

- экземпляр объекта `Windows.Storage.StorageFolder`, представляющий выбранную пользователем в диалоге папку;
- значение `null`, если пользователь отказался от выбора папки.

```
oFP.pickSingleFolderAsync().then(function(folder) {
  if (folder) {
    //Папка была выбрана
  } else {
    //Папка не была выбрана
  }
});
```

Собственно получение списка файлов и папок

Платформа Metro позволяет нам получить как простой список файлов и папок, хранящихся в указанной нами папке, так и более сложный, отсортированный по заданному нами же критерию.

Получение простого списка файлов и папок

Проще всего получить простой список содержимого указанной нами папки, без всякой сортировки по алфавиту или иным критериям. Для этого служат два метода объекта `Windows.Storage.StorageFolder`, который, как мы уже знаем, представляет папку:

- `getFilesAsync` — получение списка всех файлов;
- `getFoldersAsync` — получение списка всех папок.

Оба метода в данном случае вызываются без параметров и возвращают в качестве результата обязательство. Функция, которую мы для него укажем в вызове метода `then`, выполнится после завершения формирования списка файлов или папок и получит в качестве результата коллекцию, которая и представляет данный список. Это та же самая коллекция, что получается в результате вызова метода `pickMultipleFilesAsync` *(см. ранее в этой главе)*; она поддерживает свойство `size` и способ доступа к элементам, характерный для массивов.

```
var arrFiles = [];
libImages.getFilesAsync().then(function(files) {
  if (files.size > 0) {
    for (var i = 0; i < files.size; i++) {
      arrFiles.push(files[i]);
    }
  }
});
```

Получаем список всех файлов, что хранятся в библиотеке Изображения, и помещаем их в массив `arrFiles`.

Получение списка файлов с сортировкой

Получить список файлов, отсортированный по заданному критерию, немного сложнее. Мы вызовем знакомый нам метод `getFilesAsync`, передав ему в качестве единственного параметра один из элементов перечисления `Windows.Storage.Search.CommonFileQuery`, задающий критерий сортировки:

- `defaultQuery` — сортировка отсутствует (поведение по умолчанию);
- `orderByName` — сортировка по наименованию файла (это может быть заголовок, записанный в метаданных, или, если он отсутствует, имя файла);
- `orderByTitle` — сортировка по заголовку файла, записанному в его метаданных (это может быть заголовок документа, название музыкальной композиции или фильма, тема почтового сообщения и др.). Поддерживается только для файлов, хранящихся в библиотеке или домашней сетевой группе;
- `orderByMusicProperties` — сортировка по музыкальным метаданным (названию композиции и альбома, имени исполнителя и др.). Поддерживается только для файлов, хранящихся в библиотеке Музыка или домашней сетевой группе;
- `orderBySearchRank` — сортировка сначала по рейтингу во встроенной в Windows поисковой подсистеме, а потом — по дате последнего изменения;
- `orderByDate` — сортировка по дате файла (это может быть дата съемки для файла с фотографией, дата последней правки документа, дата последнего изменения файла и др.).

Если указан элемент `defaultQuery`, в результирующем списке будут присутствовать только файлы, непосредственно хранящиеся в указанной нами папке. Если же ука-

зать любой другой элемент, список включит также и все файлы, хранящиеся во вложенных папках.

```
var arrFiles = [];
libImages.getFilesAsync
(Windows.Storage.Search.CommonFileQuery.orderBySearchRank).
then(function(files) {
  if (files.size > 0) {
    for (var i = 0; i < files.size; i++) {
      arrFiles.push(files[i]);
    }
  }
});
```

Получаем список всех файлов в библиотеке Изображения и вложенных в нее папках, отсортированный по их рейтингу и дате последнего изменения.

Получение списка файлов с группировкой

Несложно получить и список файлов, хранящихся в указанной папке, с группировкой по заданному нами критерию. Достаточно вызвать метод `getFoldersAsync`, передав ему в качестве единственного параметра один из элементов перечисления `Windows.Storage.Search.CommonFolderQuery`, задающий критерий группировки:

- `defaultQuery` — группировка отсутствует (поведение по умолчанию);
- `groupByYear` — группировка файлов по году их даты (это может быть дата съемки для файла с фотографией, дата последней правки документа, дата последнего изменения файла и др.);
- `groupByMonth` — группировка файлов по месяцу их даты;
- `groupByArtist` — группировка аудиофайлов по имени исполнителя;
- `groupByAlbum` — группировка аудиофайлов по названию альбома;
- `groupByAlbumArtist` — группировка аудиофайлов сначала по имени исполнителя, а потом — по названию альбома;
- `groupByComposer` — группировка аудиофайлов по автору произведения;
- `groupByGenre` — группировка аудиофайлов по жанру;
- `groupByPublishedYear` — группировка аудио- и видеофайлов по году выпуска альбома или фильма;
- `groupByRating` — группировка графических, аудио- и видеофайлов по рейтингу;
- `groupByTag` — группировка файлов по ключевым словам;
- `groupByAuthor` — группировка файлов документов по автору;
- `groupByType` — группировка файлов по наименованию их типа.

Отметим, что группировка поддерживается только для файлов, хранящихся в библиотеке или домашней сетевой группе.

Если мы зададим элемент `defaultQuery`, то получим "на выходе" коллекцию всех папок, хранящихся в указанной нами папке, как если бы мы вызвали метод `getFoldersAsync` без параметров.

Если же мы зададим любой другой элемент перечисления `Windows.Storage.Search.CommonFolderQuery`, т. е. активизируем группировку, метод `getFoldersAsync` выдаст нам список папок, сформированных платформой Metro и существующих только в памяти компьютера *(виртуальных папок)*. Каждая из этих папок будет соответствовать определенному значению критерия группировки и включит файлы, относящиеся к этому значению.

Для примера предположим, что в нашей библиотеке Музыка хранятся композиции групп "Deep Purple" и "Heart". (Также предположим, что MP3-файлы с этими композициями хранят корректно заполненные теги; в противном случае ничего не выйдет.) Если мы выполним группировку по имени исполнителя, то получим список из двух виртуальных папок: Deep Purple и Heart. Первая папка будет хранить файлы с композициями "Deep Purple", а вторая — файлы с композициями "Heart".

Также заметим, что в процессе формирования списка файлов с группировкой платформа Metro будет просматривать не только основную папку самой указанной нами библиотеки, но и все папки, вложенные в нее. Так что результирующий список будет включать и файлы, хранящиеся во вложенных папках.

```
var arrFolders = [];
libMusic.getFoldersAsync
(Windows.Storage.Search.CommonFolderQuery.groupByArtist).
then(function(folders) {
  if (folders.size > 0) {
    for (var i = 0; i < folders.size; i++) {
      folders[i].getFilesAsync().then(function(files) {
        if (files.size > 0) {
          var arrFiles = [];
          for (var j = 0; j < files.size; j++) {
            arrFiles.push(files[j]);
          }
          arrFolders.push(arrFiles);
        }
      });
    }
  }
});
```

Получаем список файлов из библиотеки Музыка, сгруппированный по исполнителю, и формируем на его основе массив `arrFolders`. Каждый элемент этого массива будет соответствовать полученной в результате группировки виртуальной папке и хранить массив файлов, которые в этой папке находятся.

Получение сведений о файлах и папках

Так, список файлов и папок мы получили. Теперь давайте узнаем, как получить сведения об отдельном файле или папке — имя, расширение, размер, дату создания и изменения и пр.

Получение основных сведений о файле

Объект `Windows.Storage.StorageFile`, представляющий файл, поддерживает ряд свойств, которые позволят нам узнать различные параметры этого файла. Самые полезные из этих свойств перечислены далее:

- `name` — возвращает имя файла с расширением в виде строки;
- `displayName` — возвращает имя файла без расширения в виде строки;
- `fileType` — возвращает расширение файла с начальной точкой в виде строки;
- `path` — возвращает полный путь к файлу в виде строки;
- `dateCreated` — возвращает дату и время создания файла в виде экземпляра объекта `Date`;
- `displayType` — возвращает наименование типа файла в виде строки;
- `contentType` — возвращает MIME-тип файла в виде строки.

```
var sFileName;
oFOP.pickSingleFileAsync().then(function(file) {
  if (file) {
    sFileName = file.fileName;
  }
});
```

Получаем имя выбранного пользователем файла.

А метод `getBasicPropertiesAsync` запускает процесс получения остальных параметров файла — его размера и даты последнего изменения. Он не принимает параметров, а в качестве результата возвращает обязательство, для которого мы с помощью метода `then` укажем функцию, которая выполнится, когда параметры файла будут получены.

Данная функция получит в качестве параметра экземпляр объекта `BasicProperties`, представляющий остальные параметры файла. Нас интересуют следующие свойства этого объекта:

- `size` — возвращает размер файла в байтах в виде целого числа;
- `dateModified` — возвращает дату и время последнего изменения файла в виде экземпляра объекта `Date`.

```
var iFileSize;
oFOP.pickSingleFileAsync().then(function(file) {
  if (file) {
```

```
    file.getBasicPropertiesAsync().then(function(properties) {
      iFileSize = properties.size;
    });
  }
});
```

Получаем размер выбранного файла.

Получение миниатюры файла

Многие приложения, выводящие список файлов, против каждой позиции отображают небольшое изображение — *миниатюру*. Такой миниатюрой может быть содержимое графического файла или файла документа, первый кадр видеофайла или какое-либо стороннее изображение, обозначающее данный тип файлов.

Ранее, чтобы получить миниатюру файлов, разработчик должен был писать довольно громоздкий код. Но нам этого делать не придется — платформа Metro предоставляет встроенные средства для получения миниатюры файла, весьма гибкие и простые в использовании.

Объект `Windows.Storage.StorageFile`, представляющий файл, поддерживает метод `getThumbnailAsync`, который запускает процесс получения миниатюры данного файла. Формат его вызова таков:

```
<файл>.getThumbnailAsync(<вид миниатюры>[, <ширина миниатюры>
↳[, <параметры миниатюры>]])
```

Единственный обязательный параметр этого метода задает вид получаемой миниатюры. Он задается как один из элементов перечисления `Windows.Storage.FileProperties.ThumbnailMode`:

- `picturesView` — миниатюра размером 190×130 пикселов для списка графических файлов;
- `videosView` — миниатюра размером 190×130 пикселов для списка видеофайлов (представляет собой первый кадр фильма);
- `musicView` — миниатюра размером 40×40 пикселов для списка звуковых файлов (обложка альбома, если она есть);
- `documentsView` — миниатюра размером 40×40 пикселов для списка файлов документов (либо первая страница содержимого, либо сохраненная в файле миниатюра, если она есть);

> ***На заметку***
>
> При указании любого из этих элементов мы получим миниатюру, оптимизированную для использования в списке Metro, который выводит пункты сначала по вертикали, а потом — по горизонтали с горизонтальной прокруткой. (Подробнее о списках Metro говорилось в *главе 14*).

- `listView` — миниатюра размером 40×40 пикселов, оптимизированная для списка Metro, который выводит свои пункты по вертикали с вертикальной прокруткой.

- `singleItem` — большая миниатюра, оптимизированная для использования в качестве отдельного элемента интерфейса. Минимальный размер самой длинной стороны такой миниатюры равен 256 пикселам.

Вторым, необязательным, параметром указывается размер самой длинной стороны получаемой миниатюры в виде целого числа в пикселах. Если параметр опущен, будет создана миниатюра с размерами по умолчанию.

Третьим, также необязательным, параметром задаются дополнительные параметры получаемой миниатюры. Они указываются в виде одного из элементов перечисления `Windows.Storage.FileProperties.ThumbnailOptions`; все его элементы описаны далее:

- `none` — дополнительные параметры отсутствуют;
- `returnOnlyIfCached` — возвращать только миниатюры, либо уже хранящиеся в файле, либо те, что уже находятся в системном кэше миниатюр Windows. Если таковые отсутствуют, миниатюра не будет сформирована;
- `resizeThumbnail` — увеличить или уменьшить миниатюру, полученную из файла или системного кэша, чтобы она вписалась в указанные нами размеры;
- `useCurrentScale` — увеличить миниатюру, полученную из файла или системного кэша, соответственно разрешающей способности экрана (поведение по умолчанию).

Метод `getThumbnailAsync` возвращает в качестве результата обязательство, для которого мы в вызове метода `then` укажем функцию. Эта функция будет выполнена по завершению процесса получения миниатюры и получит в качестве параметра следующее:

- представляющий эту миниатюру экземпляр объекта `Windows.Storage.FileProperties.StorageItemThumbnail`, если миниатюру удалось получить;
- значение `null`, если ее получить не удалось.

С помощью уже знакомого нам метода `createObjectURL` объекта `URL` мы можем преобразовать полученный экземпляр объекта `Windows.Storage.FileProperties.StorageItemThumbnail` в экземпляр объекта `Blob` и вывести миниатюру на экран.

```
<img id="imgThumbnail" />
. . .
var imgThumbnail = document.getElementById("imgThumbnail");
oFOP.pickSingleFileAsync().then(function(file) {
  if (file) {
    file.getThumbnailAsync(
    ↳Windows.Storage.FileProperties.ThumbnailMode.singleItem, 512,
    ↳Windows.Storage.FileProperties.ThumbnailOptions.resizeThumbnail).
    ↳then(function(thumbnail) {
      imgThumbnail.src = URL.createObjectURL(thumbnail);
    });
  }
});
```

Получаем миниатюру выбранного пользователем файла и выводим ее на экран. Для миниатюры мы задаем вид, оптимизированный для вывода в качестве отдельного элемента интерфейса, размер длинной стороны в 512 пикселов и масштабирование миниатюр под указанный нами размер.

Получение сведений о папке и миниатюры содержимого папки

Объект `Windows.Storage.StorageFolder`, который представляет папку, поддерживает ряд свойств, позволяющих нам узнать основные параметры этой папки. Это уже знакомые нам свойства `displayName`, `name`, `path`, `dateCreated` и `properties`.

Помимо этого, платформа Metro позволяет нам получить миниатюру всего содержимого папки. Такая миниатюра представляет собой набор миниатюр отдельных файлов, хранящихся в папке, которые выглядят как сложенные в "стопку".

Получение миниатюры содержимого папки выполняется с помощью того же метода `getThumbnailAsync`, который на этот раз вызывается у папки.

```
<img id="imgThumbnail" />
. . .
var imgThumbnail = document.getElementById("imgThumbnail");
oFP.pickSingleFolderAsync().then(function(folder) {
  if (folder) {
    folder.getThumbnailAsync(
    ↳Windows.Storage.FileProperties.ThumbnailMode.singleItem, 512).
    ↳then(function(thumbnail) {
      imgThumbnail.src = URL.createObjectURL(thumbnail);
    });
  }
});
```

Пример: приложение для просмотра графических файлов

Для практики давайте создадим приложение, позволяющее просматривать графические файлы. Оно будет формировать список файлов, хранящихся в библиотеке Изображения, в виде миниатюр и выводить увеличенное содержимое выбранного файла в отдельном блоке.

Создадим в Visual Studio проект `ImageViewer`. Сразу же дадим новому приложению права на доступ к библиотеке Изображения, установив флажок **Pictures Library** в списке **Capabilities** на одноименной вкладке (см. рис. 16.1) окна параметров приложения. Если этого не сделать, наше приложение не сможет получить доступ к данной библиотеке.

Откроем файл default.html и введем в тег `<body>` вот такой код:

```
<div id="divFileName"></div>
<div id="divViewer" data-win-control="WinJS.UI.ViewBox">
  <img id="imgViewer" />
</div>
<div id="divList" data-win-control="WinJS.UI.ListView"
data-win-options="{selectionMode: 'single'}"></div>
```

Мы создаем три блока, которые потом разместим на экране с применением сеточной разметки. Верхний блок `divFileName` будет служить для вывода имени выбранного в списке файла. Средний, самый большой, блок `divViewer` мы превратим в панель вывода Metro; она получит в качестве содержимого элемент графического изображения `imgViewer`, в котором будет выводиться содержимое выбранного файла. А нижний блок `divList` потом превратится в список Metro, перечисляющий графические файлы.

Откроем файл default.css и создадим в нем несколько стилей, перечисленных далее.

```
body {
  display: -ms-grid;
  -ms-grid-columns: 1fr;
  -ms-grid-rows: 40px 1fr 30px 170px;
}
```

Здесь мы создаем сеточную разметку. Отметим, что она будет включать четыре строки: первая, вторая и четвертая вместят в себя созданные нами блоки, а третья создаст просвет между блоками `divViewer` и `divList`.

```
#divFileName {
  font-size: 16pt;
  text-align: center;
}
```

Для верхнего блока, в котором будет выводиться имя выбранного файла, задаем размер шрифта в 16 пунктов и выравнивание по центру.

```
#divViewer {
  -ms-grid-row: 2;
  -ms-grid-column-align: center;
  -ms-grid-row-align: center;
}
```

Для среднего блока указываем, в том числе, выравнивание по центру. После этого выбранное пользователем изображение будет выведено в середине данного блока.

```
#divList {
  -ms-grid-row: 4;
  height: 170px;
}
```

По умолчанию список Metro имеет слишком большую высоту — 400 пикселов. Так что нам придется задать для него значение высоты, совпадающее с высотой последней строки сеточной разметки, в которой он будет находиться.

```
#divList .win-item {
  width: 190px;
  height: 130px;
}
```

Для пунктов списка `divList` указываем размеры 190×130 пикселов. Такие же размеры будут иметь миниатюры, которые мы станем выводить в этом списке.

Закончив со стилями, переключимся на файл default.js. Сначала, как обычно, напишем код, объявляющий необходимые переменные:

```
var divFileName, imgViewer, ctrList, arrFiles = [], dsrFiles;
```

Напишем код инициализации:

```
document.addEventListener("DOMContentLoaded", function() {
  WinJS.UI.processAll().then(function() {
    divFileName = document.getElementById("divFileName");
    imgViewer = document.getElementById("imgViewer");
    ctrList = document.getElementById("divList").winControl;
    ctrList.addEventListener("iteminvoked", ctrListItemInvoked);
    ctrList.itemTemplate = listTemplate;
    fillList();
  });
});
```

Здесь мы, помимо всего прочего, привязываем обработчик к событию `iteminvoked` списка, задаем для него шаблон-функцию `listTemplate()` и вызываем функцию `fillList()`, которая заполнит список.

Сразу же объявим функцию `fillList()`:

```
function fillList() {
    var libImages = Windows.Storage.KnownFolders.picturesLibrary;
    libImages.getFilesAsync().then(function(files) {
    if (files.size > 0) {
      for (var i = 0; i < files.size; i++) {
        arrFiles.push(files[i]);
      }
    }
    dsrFiles = new WinJS.Binding.List(arrFiles);
    ctrList.itemDataSource = dsrFiles.dataSource;
  });
}
```

Мы получаем доступ к библиотеке Изображения, помещаем все имеющиеся в нем файлы в массив, формируем на основе этого массива источник данных и привязываем к нему список.

Не забудем объявить шаблон-функцию `listTemplate()`:

```
function listTemplate(itemPromise) {
  return itemPromise.then(function(item) {
    var oItem = item.data;
```

```
    var oDiv = document.createElement("div");
    var oImg = document.createElement("img");
    oItem.getThumbnailAsync(
    ↳Windows.Storage.FileProperties.ThumbnailMode.picturesView).
    ↳then(function(thumbnail) {
      oImg.src = URL.createObjectURL(thumbnail);
    });
    oDiv.appendChild(oImg);
    return oDiv;
  });
}
```

Здесь мы создаем блок, который станет пунктом списка, и помещаем внутри него элемент графического изображения, в котором выводим миниатюру соответствующего ему файла.

Осталось только объявить функцию-обработчик события `iteminvoked` списка.

```
function ctrListItemInvoked(evt) {
  evt.detail.itemPromise.then(function(selected) {
    var oFile = selected.data;
    imgViewer.src = URL.createObjectURL(oFile);
    divFileName.textContent = oFile.name;
  });
}
```

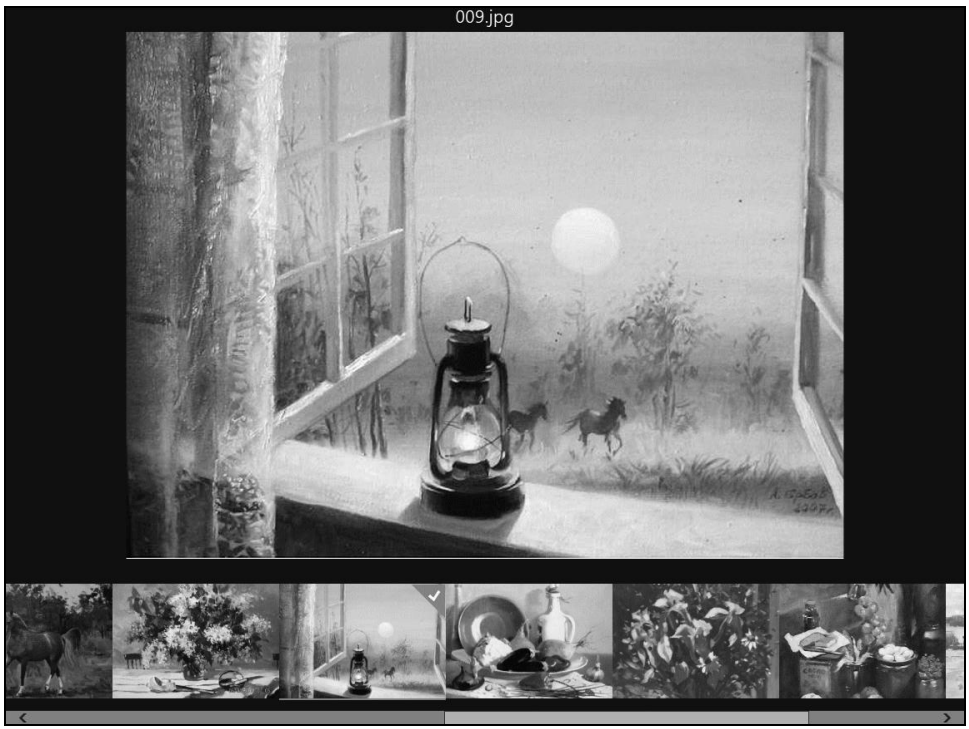


Рис. 16.3. Интерфейс приложения для просмотра графических файлов

Получаем выбранный в списке файл, выводим его содержимое в элементе графического изображения `imgViewer`, а его имя — в верхнем блоке `divFileName`.

Сохраним все файлы и запустим приложение. Дождемся, когда список файлов будет сформирован, выберем какой-либо файл и посмотрим на его содержимое (рис. 16.3). Если мы все сделали правильно, все должно работать.

Действия над файлами и папками

А что еще платформа Metro позволяет нам сделать с файлами и папками? Можем ли мы, скажем, скопировать файл в другое место или создать новую папку? Разумеется!

> ***Внимание!***
>
> Чтобы все эти действия успешно выполнились, приложение должно иметь права на доступ к содержимому соответствующей библиотеки.

Действия над файлами

Доступные нам действия над файлами выполняются вызовом особых методов. Все эти методы поддерживаются объектом `Windows.Storage.StorageFile`, если в их описании не указано иное.

Метод `copyAsync` запускает процесс копирования файла в другую папку и, возможно, под другим именем.

```
<файл>.copyAsync(<папка назначения>[, <имя копии>
[, <что делать в случае конфликта имен>]])
```

Первым, обязательным, параметром указывается папка назначения в виде экземпляра объекта `Windows.Storage.StorageFolder`.

Вторым, необязательным, параметром задается имя копии файла — в виде строки. Если оно не указано, копия файла будет иметь то же имя, что и оригинальный файл.

Третьим, также необязательным, параметром указывается действие, выполняемое платформой Metro в случае конфликта имен, или, другими словами, если в папке назначения уже есть файл с таким же именем. Значение этого параметра должно представлять собой один из элементов перечисления `Windows.Storage.NameCollisionOption`:

- `generateUniqueName` — файл будет успешно скопирован, и его копия получит уникальное имя, образованное добавлением к изначальному имени порядкового номера;
- `replaceExisting` — файл будет успешно скопирован, и его копия заменит уже имеющийся файл;
- `failIfExists` — файл скопирован не будет, и в процессе копирования возникнет ошибка (поведение по умолчанию).

Метод `copyAsync` возвращает в качестве результата обязательство, для которого мы в вызове метода `then` укажем две функции. Первая функция выполнится после успешного копирования и получит в качестве параметра экземпляр объекта `Windows.Storage.StorageFile`, представляющий файл, что был получен в результате копирования. Вторая функция выполнится, если в процессе копирования возникнет ошибка (например, если в папке назначения уже есть файл с таким же именем, и мы указали, чтобы в таком случае файл не копировался).

Мы можем использовать эти функции для выполнения каких-либо завершающих действий или для информирования пользователя об успешном или неудачном копировании файла.

```
var oDestination = Windows.Storage.KnownFolders.documentsLibrary;
oFOP.pickSingleFileAsync().then(function(file) {
  if (file) {
    file.copyAsync(oDestination).then(function(copy) {
      //Файл успешно скопирован
    }, function() {
      //Копирование не было выполнено
    });
  }
});
```

Копируем выбранный пользователем файл в библиотеку Документы.

Даже если мы не собираемся информировать пользователя об успешном копировании файла, мы все равно должны указать первую функцию, хотя бы пустую. В противном случае файл не будет скопирован.

```
oFOP.pickSingleFileAsync().then(function(file) {
  if (file) {
    file.copyAsync(oDestination).then(function() { });
  }
});
```

Метод `moveAsync` запускает процесс перемещения файла в другое месторасположение. Формат его вызова такой же, как у метода `copyAsync`.

```
oFOP.pickSingleFileAsync().then(function(file) {
  if (file) {
    file.moveAsync(oDestination, file.fileName,
    Windows.Storage.NameCollisionOption.replaceExisting).
    ↳then(function() { });
  }
});
```

Метод `renameAsync` запускает процесс переименования файла.

```
<файл>.renameAsync(<новое имя>[, <что делать в случае конфликта имен>]])
```

Первым, обязательным, параметром указывается новое имя файла в виде строки, вторым — действие, выполняемое в случае конфликта имен. Возвращаемый результат — также обязательство.

```
oFOP.pickSingleFileAsync().then(function(file) {
  if (file) {
    file.renameAsync(file.name + "2" + file.fileType,
    Windows.Storage.NameCollisionOption.replaceExisting).
    ↳then(function() { });
  }
});
```

Переименовываем выбранный пользователем файл, добавив к его имени двойку.

Метод `deleteAsync` запускает процесс удаления файла. В качестве единственного параметра он принимает дополнительные параметры, указанные в виде одного из элементов перечисления `Windows.Storage.StorageDeleteOption`:

- `default` — если файл находится вне хранилища приложения, удалить его в Корзину; в противном случае удалить файл без помещения в Корзину (о хранилищах приложения разговор пойдет в конце этой главы);
- `permanentDelete` — в любом случае удалить файл без помещения в Корзину.

Метод `deleteAsync` также возвращает в качестве результата обязательство.

```
oFOP.pickSingleFileAsync().then(function(file) {
  if (file) {
    file.deleteAsync(
    ↳Windows.Storage.StorageDeleteOption.tryMoveToRecycleBin).
    ↳then(function() { });
  }
});
```

Удаляем выбранный пользователем файл в Корзину.

Метод `getFileAsync` объекта `Windows.Storage.StorageFolder` запускает процесс получения файла, находящегося в данной папке и имеющего указанное нами имя.

<папка>.getFileAsync(*<имя файла>*)

Единственным параметром он принимает имя открываемого файла в виде строки.

В качестве результата он опять же вернет привычное нам обязательство. И мы, как обычно, укажем для него в вызове метода `then` функцию, которая будет выполнена после завершения процесса получения файла. Она примет в качестве единственного параметра:

- экземпляр объекта `Windows.Storage.StorageFile`, представляющий полученный файл;
- значение `null`, если таковой файл не существует.

```
var libDocuments = Windows.Storage.KnownFolders.documentsLibrary;
var sFileName = "test.txt";
libDocuments.getFileAsync(sFileName).then(function(file) {
  if (file) {
    file.openAsync(Windows.Storage.FileAccessMode.read).
    ↳then(function(stream) {
```

```
      //Выполняем чтение из файла
    });
  }
})
```

Открываем файл test.txt, хранящийся в библиотеке Документы, и читаем его содержимое.

Метод `getFileFromPathAsync` запускает процесс получения произвольного файла, находящегося по указанному нами пути.

```
Windows.Storage.StorageFile.getFileFromPathAsync(<путь к файлу>)
```

Отметим, что он вызывается у самого объекта `Windows.Storage.StorageFile`, а не у его экземпляра. В качестве единственного параметра он принимает полный путь к файлу в виде строки, а возвращает обязательство того же типа, что и знакомый нам метод `getFileAsync`.

```
var sFilePath = "c:\\test.txt";
Windows.Storage.StorageFile.getFileFromPathAsync(sFilePath).
↳then(function(file) {
  if (file) {
    file.openAsync(Windows.Storage.FileAccessMode.read).
    ↳then(function(stream) {
      //Выполняем чтение из файла
    });
  }
})
```

Открываем файл test.txt, хранящийся в корневой папке диска C:, и выполняем чтение его содержимого. (Отметим, что мы продублировали символ обратного слеша в строке с путем к получаемому файлу. Зачем это делается, говорилось в *главе 4*.)

Метод `createFileAsync` объекта `Windows.Storage.StorageFolder` запускает процесс создания файла в данной папке.

```
<папка>.createFileAsync(<имя файла>
↳[, <что делать в случае конфликта имен>])
```

Первым, обязательным, параметром передается само имя создаваемого файла в виде строки.

Второй, необязательный, параметр позволяет указать, какое действие Metro должна выполнить в случае, если файл с таким именем уже существует в папке. Это действие указывается элементом перечисления `Windows.Storage.CreationCollisionOption`:

- `generateUniqueName` — созданный файл получит уникальное имя, образованное добавлением к указанному нами имени порядкового номера;
- `replaceExisting` — существующий файл будет заменен вновь созданным;
- `failIfExists` — файл создан не будет, и в процессе работы возникнет ошибка (поведение по умолчанию);

- `openIfExists` — в результате выполнения метода будет получен существующий файл (вероятно, самое часто выполняемое действие).

Метод `createFileAsync` возвращает в качестве результата обязательство, для которого мы в вызове метода `then` укажем одну или две функции. Первая функция выполнится после успешного создания файла и получит в качестве параметра экземпляр объекта `Windows.Storage.StorageFile`, представляющий созданный файл. Вторая, необязательная, функция выполнится, если в процессе создания файла возникнет ошибка (например, если в папке назначения уже есть файл с таким же именем, и мы указали, чтобы в таком случае файл не создавался).

```
var oFolder = Windows.Storage.KnownFolders.documentsLibrary;
oFolder.createFileAsync("test.txt",
Windows.Storage.CreationCollisionOption.openIfExists).
↳then(function(file) {
  file.openAsync(
  ↳Windows.Storage.FileAccessMode.readWriteNoCopyOnWrite).
  ↳then(function(stream) {
    //Выполняем запись в файл
  });
});
```

Создаем в библиотеке Документы файл test.txt или получаем уже существующий там файл с таким именем, после чего выполняем в него запись.

Действия над папками

Все доступные нам действия над папками выполняются вызовами соответствующих методов объекта `Windows.Storage.StorageFile`.

Прежде всего, это методы `renameAsync` и `deleteAsync`, выполняющие, соответственно, переименование и удаление папки. Как они вызываются, мы уже знаем.

Метод `getFolderAsync` запускает процесс получения папки, находящейся в данной папке и имеющей указанное нами имя.

<папка>`.getFolderAsync(`*<имя папки>*`)`

Единственным параметром он принимает имя открываемого файла в виде строки.

Результатом, возвращенным этим методом, будет обязательство. В вызове метода `then` мы укажем для него функцию, которая будет выполнена после завершения процесса получения папки. Она получит в качестве единственного параметра:

- экземпляр объекта `Windows.Storage.StorageFolder`, представляющий полученную папку;
- значение `null`, если папка с таким именем не существует.

```
var libDocuments = Windows.Storage.KnownFolders.documentsLibrary;
var sFolderName = "test ";
libDocuments.getFolderAsync(sFolderName).then(function(folder) {
  if (folder) {
```

```
    folder.getFilesAsync().then(function(files) {
      if (files.size > 0) {
        //Выполняем действия над файлами
      }
    });
  }
})
```

Открываем папку test библиотеки Документы и выполняем какие-либо действия над хранящимися в ней файлами.

А метод `getFolderFromPathAsync` запускает процесс получения произвольной папки по заданному нами пути.

```
Windows.Storage.StorageFolder.getFolderFromPathAsync(<путь к папке>)
```

Он вызывается у самого объекта `Windows.Storage.StorageFolder`, а не у его экземпляра. В качестве единственного параметра он принимает путь к получаемой папке в виде строки и возвращает обязательство того же типа, что и метод `getFolderAsync`.

```
var sFolderPath = "c:\\test";
Windows.Storage.StorageFolder.getFolderFromPathAsync(sFolderPath).
↳then(function(folder) {
  if (folder) {
    folder.getFilesAsync().then(function(files) {
      if (files.size > 0) {
        //Выполняем действия над файлами
      }
    });
  }
})
```

Получаем доступ к папке test, хранящейся в корневой папке диска C:, и выполняем какие-либо действия над хранящимися в ней файлами.

Метод `createFolderAsync` запускает процесс создания новой папки. Формат его вызова такой же, как у метода `createFileAsync`.

```
var oFolder = Windows.Storage.KnownFolders.documentsLibrary;
oFolder.createFolderAsync("test",
Windows.Storage.CreationCollisionOption.openIfExists).
↳then(function(folder) {
  folder.createFileAsync("test.txt",
  Windows.Storage.CreationCollisionOption.openIfExists).
  ↳then(function(file) {
    file.openAsync(
    ↳Windows.Storage.FileAccessMode.readWriteNoCopyOnWrite).
    ↳then(function(stream) {
      //Выполняем запись в файл
    });
  });
});
```

Создаем в библиотеке Документы папку test или получаем уже существующую там папку с тем же именем. После этого создаем в данной папке файл test.txt и выполняем запись в него.

Хранилища приложения

Многие приложения хранят в файлах какие-либо данные, необходимые им для работы. К таким данным относятся, например, настройки приложения, сведения о его состоянии (подробнее о состоянии приложения мы поговорим в *главе 23*), промежуточные результаты вычислений и пр.

Специально для хранения этих данных платформа Metro предусматривает так называемые *хранилища приложения*. Они представляют собой папки, которые создаются самой Metro при первом обращении к соответствующему хранилищу. Для доступа к ним приложению не требуются никакие особые права; подразумевается, что хранилище — неотъемлемая собственность приложения, и оно может распоряжаться им по своему усмотрению.

Платформа Metro предоставляет приложению три различных хранилища. Давайте их рассмотрим.

- *Локальное*, или *непереносимое*. Служит для хранения данных, которые не требуется синхронизировать между всеми устройствами — традиционными компьютерами и планшетами, — принадлежащими пользователю. Объем локального хранилища не ограничен (точнее, ограничен только объемом долговременной памяти).

  Локальное хранилище обычно применяется для хранения файлов с временными данными, промежуточными результатами, состоянием и настройками, относящимися только к данной копии приложения.

- *Переносимое.* В отличие от локального хранилища, оно применяется для хранения данных, которые требуется синхронизировать между устройствами.

  Следует помнить, что объем переносимого хранилища ограничен. Точнее, хранить в нем можно данные любого объема, но по достижении определенного, установленного самой платформой Metro предела синхронизация данных перестает выполняться. Она будет возобновлена только после удаления избыточных данных.

  Переносимое хранилище — идеальное место для хранения файлов с информацией о состоянии приложения и настройками, которые оказывают влияние на все копии приложения. При одном условии: все эти данные должны быть невелики по объему.

- *Временное.* Применяется для кратковременного хранения любых данных. Размер временного хранилища не ограничен.

  У временного хранилища есть одна особенность. Время от времени платформа Metro просматривает его содержимое и удаляет все файлы, которые давно не открывались, с целью освободить место на диске. Обязательно следует иметь в ви-

ду, что эта чистка производится без какого-либо предупреждения; говоря другими словами, приложение не "знает", хранится ли еще записанный им файл, или он уже удален.

Временное хранилище обычно используется для кэширования (временного хранения) файлов, загруженных из сети. Также его можно использовать для записи данных, которые актуальны только в течение краткого промежутка времени (промежуточные результаты вычислений, рабочие данные, слишком объемные, чтобы постоянно держать их в оперативной памяти, резервные копии документов и пр.).

Получить доступ к хранилищам приложения можно с помощью объекта `Windows.Storage.ApplicationData`. Сначала мы обратимся к его свойству `current`, которое хранит экземпляр данного объекта, представляющий все хранилища приложения.

```
var oAppData = Windows.Storage.ApplicationData.current;
```

Этот экземпляр объекта поддерживает свойства, позволяющие получить доступ к конкретному хранилищу:

- `localFolder` — локальное хранилище;
- `roamingFolder` — переносимое хранилище;
- `temporaryFolder` — временное хранилище.

Значения всех этих свойств представляют собой экземпляры объекта `Windows.Storage.StorageFolder`, т. е. папки.

```
var oRF = oAppData.roamingFolder;
oRF.createFileAsync("settings.ini",
Windows.Storage.CreationCollisionOption.openIfExists).
↳then(function(file) {
  file.openAsync(
  ↳Windows.Storage.FileAccessMode.readWriteNoCopyOnWrite).
  ↳then(function(stream) {
    //Выполняем запись в файл
  });
});
```

Создаем в переносимом хранилище файл settings.ini и выполняем запись в него.

Свойство `roamingStorageQuota` объекта `Windows.Storage.ApplicationData` возвращает максимально допустимый объем данных, которые приложение может записать в переносимое хранилище, в виде целого числа в килобайтах. А свойство `roamingStorageUsage` того же объекта возвращает объем уже хранящихся в нем данных, также в виде целого числа и в килобайтах.

```
if (oAppData.roamingStorageQuota – oAppData.roamingStorageUsage > 1) {
  //Выполняем запись в файл
}
```

Выполняем запись в созданный в переносимом хранилище файл, только если в нем остался, по меньшей мере, 1 Кбайт свободного места.

Как только содержимое переносимого хранилища изменяется, возникает событие `datachanged` объекта `Windows.Storage.ApplicationData`. Обработчик этого события можно использовать для отслеживания изменений в переносимом хранилище, внесенных другой копией того же приложения, установленной на другом компьютере, который принадлежит этому же пользователю. С учетом того, что переносимое хранилище обычно применяется для хранения настроек, мы таким образом можем реализовать автоматическую синхронизацию сделанных в настройках изменений среди всех копий данного приложения.

```
oAppData.addEventListener("datachanged", function() {
  var oRF = oAppData.roamingFolder;
  oRF.getFileAsync("settings.ini").then(function(file) {
    file.openAsync(Windows.Storage.FileAccessMode.read).
    ↳then(function(stream) {
      //Выполняем чтение настроек из файла
    });
  });
});
```

Этот обработчик события `datachanged` откроет файл settings.ini, хранящийся в переносимом хранилище, и прочитает его содержимое.

Что дальше?

В этой главе мы учились работать с файлами. Мы узнали, как выяснить у пользователя, какой файл он хочет открыть или сохранить, как выполнить чтение и запись, как получить список папок и файлов, хранящихся в указанной нами папке, как получить сведения о файле и папке и как выполнить над ними различные действия (копирование, перемещение, удаление и др.). Глава оказалась очень объемной, но так ведь и сама тема немаленькая...

Следующая глава будет посвящена работе с каналами RSS. Этот способ распространения информации в Интернете стал настолько популярным, что платформа Metro получила специальные инструменты для его поддержки. Заодно мы доделаем свое приложение для чтения каналов RSS — хватит ему пребывать на стадии прототипа!

ГЛАВА 17

Работа с каналами новостей RSS и Atom

В предыдущей главе мы рассматривали внутренние механизмы платформы Metro, позволяющие нам работать с файлами. Мы познакомились с диалогами открытия и сохранения файла, диалогом выбора папки, средствами для файлового чтения и записи, перечисления содержимого указанной нами папке, копирования, перемещения, удаления и пр. И хотя некоторые из этих средств оказались несколько мудреными, они вполне покроют все наши потребности в работе с файлами.

В последние несколько лет стала широко использоваться такая форма распространения информации в Интернете, как каналы новостей, созданные по стандартам RSS и Atom. На данный момент их популярность настолько велика, что платформа Metro обзавелась встроенными механизмами для их поддержки, которым и будет посвящена данная глава.

Стандарты RSS и Atom предоставляют примерно одинаковые возможности по созданию и наполнению каналов новостей. Неудивительно, что загрузка содержимого каналов, созданных на основе этих обоих стандартов, выполняется сходным образом.

Подготовка интернет-адреса

Получение интернет-адреса канала новостей, который хочет загрузить пользователь, не должно вызвать у нас проблем. Обычно для этого используют всплывающие элементы с полем ввода и кнопкой подписки на канал; как все это сделать, мы давно знаем.

Итак, строку с интернет-адресом канала новостей мы получили. И здесь мы столкнемся с проблемой: механизмы Metro, работающие с каналами новостей, не "переваривают" интернет-адреса, заданные в виде строк.

Они манипулируют интернет-адресами, представляющими собой экземпляры объекта `Windows.Foundation.Uri`. Поэтому нам придется создать экземпляр данного объекта, передав ему в качестве параметра строку с интернет-адресом.

```
var sFeedURL = "http://www.thevista.ru/rss.php";
var oFeedURL = new Windows.Foundation.Uri(sFeedURL);
```

Мы взяли интернет-адрес канала RSS с Web-сайта **http://www.thevista.ru/** и создали на его основе экземпляр объекта `Windows.Foundation.Uri`.

Кстати, этот объект поддерживает свойство `absoluteUri`. Оно возвращает интернет-адрес, на основе которого был создан его экземпляр, в виде строки. Это свойство нам очень пригодится впоследствии.

Создание клиента новостей и загрузка содержимого канала

Получив интернет-адрес в поддерживаемом платформой Metro формате, мы можем приступить к собственно загрузке содержимого данного канала новостей. Выполняется это в два этапа.

На первом этапе мы создадим так называемого *клиента новостей*, который и выполнит загрузку канала. Он представляется экземпляром объекта `Windows.Web.Syndication.SyndicationClient`, который нам потребуется создать.

```
var oFeedClient = new Windows.Web.Syndication.SyndicationClient();
```

На втором этапе мы вызовем у готового клиента новостей метод `retrieveFeedAsync`, запускающий процесс загрузки содержимого канала.

```
<клиент новостей>.retrieveFeedAsync(<интернет-адрес канала>)
```

Единственным параметром этому методу передается интернет-адрес канала в виде экземпляра объекта `Windows.Foundation.Uri`.

Метод `retrieveFeedAsync` возвращает в качестве результата обязательство, для которого мы укажем в вызове метода `then` функцию. Она выполнится после успешной загрузки содержимого указанного нами канала новостей и получит в качестве результата экземпляр объекта `Windows.Web.Syndication.SyndicationFeed`, представляющий этот канал.

Вся остальная работа с загруженным каналом новостей будет протекать в теле этой функции.

```
oFeedClient.retrieveFeedAsync(oFeedURL).then(function(feed) {
  //Выполняем чтение загруженного канала
});
```

Надо сказать, что мы можем указать в вызове метода `then` еще две функции (*см. в главе 5* формат вызова данного метода). Первая из них выполнится при возникновении ошибки, а вторая будет периодически вызываться в процессе загрузки канала. Но можно этого и не делать, как во многих случаях и поступают.

Получение сведений о канале новостей

Объект `Windows.Web.Syndication.SyndicationFeed`, представляющий канал новостей, поддерживает несколько полезных для нас свойств, которые позволят нам узнать основные сведения о канале. Эти свойства перечислены далее.

- `title` — возвращает заголовок канала.
- `subtitle` — возвращает подзаголовок канала.
- `rights` — возвращает сведения об авторских правах.

  Значением всех этих свойств является экземпляр особого объекта, который и хранит соответствующие свойству данные. Чтобы получить их в строковом виде, мы обратимся к свойству `text` этого объекта.

```
var sFeedTitle = feed.title.text;
```

- `iconUri` — возвращает интернет-адрес значка канала.
- `imageUri` — возвращает интернет-адрес графического логотипа канала.

  Значением этих свойств является экземпляр уже знакомого нам объекта `Windows.Foundation.Uri`. Если канал не содержит значка или логотипа, соответствующее свойство будет хранить значение `null`.

```
<img id="imgLogo" />
. . .
var imgLogo = document.getElementById("imgLogo");
var oLogoURL = feed.imageUri;
if (oLogoURL) {
  var sLogoURL = oLogoURL.absoluteUri;
  imgLogo.src = URL.createObjectURL(sLogoURL);
}
```

  Проверяем, имеет ли канал новостей логотип, и, если имеет, выводим в его элементе графического изображения `imgLogo`.

- `lastUpdatedTime` — возвращает дату и время последнего изменения содержимого данного канала в виде экземпляра объекта `Date`.
- `id` — возвращает внутренний идентификатор канала в виде строки.

Получение отдельных новостей

Загрузив канал, мы можем получить все новости, составляющие его содержимое. Для этого мы воспользуемся свойством `items` объекта `Windows.Web.Syndication.SyndicationFeed`, представляющего канал. Это свойство хранит экземпляр того же объекта-коллекции, что возвращается методами, выполняющими получение списка файлов *(см. главу 16)*.

```
var oItems = feed.items;
var arrItems = [];
```

```
for (var i = 0; i < oItems.size; i++) {
  arrItems.push(oItems[i]);
}
```

Получаем список новостей, имеющихся в загруженном ранее канале, и помещаем их в массив `arrItems`.

Каждая новость — элемент полученной ранее коллекции — представляется экземпляром объекта `Windows.Web.Syndication.SyndicationItem`. Перечисленные далее свойства этого объекта хранят сведения о новости.

- `title` — возвращает заголовок новости.
- `summary` — возвращает содержимое новости RSS.
- `rights` — возвращает сведения об авторских правах создателей новости.

  Значением всех этих свойств является тот же экземпляр объекта, что возвращается одноименными свойствами канала *(см. ранее)*.
- `content` — возвращает содержимое новости Atom.

  Значением этого свойства является экземпляр объекта `Windows.Web.Syndication.SyndicationContent`. Чтобы получить содержимое новости Atom в строковом виде, следует обратиться к свойству `text` данного объекта.
- `authors` — возвращает список авторов новости.

  Значением этого свойства будет тот же экземпляр объекта-коллекции, что возвращается методами, выполняющими получение списка файлов *(см. главу 16)*. Каждый из элементов этой коллекции представляет собой экземпляр объекта `Windows.Web.Syndication.SyndicationPerson`, представляющий автора новости. Свойство `name` данного объекта возвращает имя автора, а свойство `email` — его адрес электронной почты; оба значения представляют собой строки.
- `links` — возвращает список всех интернет-адресов, имеющихся в составе новости.

  Значением этого свойства будет аналогичный экземпляр объекта-коллекции. Каждый из его элементов представляет собой экземпляр объекта `Windows.Web.Syndication.SyndicationLink`, представляющий интернет-адрес из состава новости. Свойство `uri` данного объекта возвращает сам интернет-адрес в виде экземпляра объекта `Windows.Foundation.Uri`.
- `publishedDate` — возвращает дату и время публикации новости в виде экземпляра объекта `Date`.

Теперь дадим необходимые пояснения.

Содержимое новости можно получить либо из свойства `content`, либо из свойства `summary`. Обычно сначала проверяют, хранится ли что-либо в свойстве `content` (не равно ли ее значение `null`); если же там ничего нет, используют значение свойства `summary`.

В списке интернет-адресов, имеющихся в составе новости, самым первым идет интернет-адрес, ведущий на Web-страницу с полным текстом новости. Поэтому, как правило, в приложениях для чтения каналов RSS используют только первый интернет-адрес из списка.

```
var arrItems = [], oI, sAuthors, sContent;
for (var i = 0; i < feed.items.size; i++) {
  oI = feed.items[i];
  sAuthors = "";
  for (var j = 0; j < oI.authors.size; j++) {
    if (sAuthors != "") {
      sAuthors += ", ";
    }
    sAuthors += oI.authors[j].name;
  }
  if (oI.content) {
    sContent = oI.content.text;
  } else {
    sContent = oI.summary.text;
  }
  arrItems.push({
    title: oI.title.text,
    authors: sAuthors,
    content: sContent,
    url: oI.links[0].uri.absoluteUri,
    date: oI.publishedDate
  });
}
```

Перебираем все новости из загруженного канала и для каждой добавляем в массив `arrItems` описывающий ее элемент. Этот элемент включит заголовок новости; строку со списком ее авторов, чьи имена разделены запятыми; содержимое; первый интернет-адрес и дату публикации.

Полученный массив, кстати, можно использовать в качестве массива данных для формирования списка Metro. Чем мы сейчас и займемся.

Пример: окончательная версия приложения для чтения каналов RSS

Вооружившись всеми необходимыми знаниями, давайте доведем до ума наше приложение для чтения каналов RSS. А еще лучше, чтобы в который раз не вносить масштабные правки в старый код, создадим новое приложение, выполняющее ту же задачу.

Пусть проект нового приложения носит имя `FeedReader`. Откроем файл default.html и введем в тег `<body>` следующий код, для удобства разбив его пустыми строками на части:

```
<div id="divListTemplate" data-win-control="WinJS.Binding.Template">
  <h2 data-win-bind="textContent: title"></h2>
  <p data-win-bind="textContent: authors" class="item-authors"></p>
  <div data-win-bind="textContent: content" class="item-content"></div>
  <p data-win-bind="textContent: date" class="item-date"></p>
</div>

<div id="divHeader">
  <h1 id="hHeader"></h1>
</div>
<div id="divList" data-win-control="WinJS.UI.ListView"
data-win-options="{layout: {type: WinJS.UI.ListLayout},
↳itemTemplate: divListTemplate, selectionMode: 'single'}"></div>
<div id="divContent"></div>

<div id="divURL" data-win-control="WinJS.UI.Flyout">
  <div>
    <label for="txtURL">Интернет-адрес</label>
    <br />
    <input type="url" id="txtURL" />
  </div>
  <div>
    <input type="button" id="btnURL" value="Подписаться на канал" />
  </div>
</div>

<div id="divAppBar" data-win-control="WinJS.UI.AppBar"
data-win-options="{placement: 'top'}">
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnRefresh', label: 'Обновить',
  ↳icon: 'refresh', section: 'selection'}"></button>
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnSubscribe', label: 'Подписаться',
  ↳icon: 'add', section: 'global', type: 'flyout',
  ↳flyout: 'divURL'}"></button>
</div>
```

Здесь мы, прежде всего, создаем три блока, которые разместим на экране с применением сеточной разметки:

- верхний — `divHeader` — включит в свой состав заголовок первого уровня `hHeader`, который мы используем для вывода заголовка канала RSS;
- левый — `divList` — мы превратим в список Metro, выводящий перечень новостей;
- правый — `divContent` — мы используем для вывода Web-страницы, содержащей полный текст выбранной в списке новости. Загрузкой Web-страниц мы будем заниматься в *главе 18*, а пока оставим этот блок пустым.

Помимо этого, мы создаем еще два элемента интерфейса:

- панель инструментов с кнопками **Подписаться** (выводит на экран всплывающий элемент, позволяющий подписаться на канал) и **Обновить** (обновляет содержимое канала, загружая его повторно);
- всплывающий элемент с полем ввода **Интернет-адрес** (канала RSS, который пользователь хочет загрузить) и кнопкой **Подписаться на канал** (собственно загружает канал с указанным интернет-адресом). Здесь пользователь сможет указать интернет-адрес канала новостей, который он хочет просмотреть.

Наконец, мы создаем шаблон для списка, в котором будет выводиться содержимое канала новостей.

Откроем файл default.css и создадим в нем стили, перечисленные далее.

```
body {
  display: -ms-grid;
  -ms-grid-columns: 1fr 1fr;
  -ms-grid-rows: auto 1fr;
}
#divHeader {
  -ms-grid-column-span: 2;
  -ms-grid-column-align: center;
}
#divList {
  -ms-grid-row: 2;
  height: 100%;
}
#divContent {
  -ms-grid-row: 2;
  -ms-grid-column: 2;
}
#divList .win-item { padding: 10px; }
```

Следующие три стиля будут привязаны к элементам интерфейса, находящимся в шаблоне.

```
.item-authors {
  font-style: italic;
  text-align: right;
}
```

Этот стилевой класс будет привязан к абзацу со списком авторов новости.

```
.item-content {
  padding-top: 10px;
  padding-bottom: 10px;
}
```

Этот стилевой класс будет привязан к блоку, в котором выводится содержимое новости. Он устанавливает внутренние отступы сверху и внизу, равные 10 пикселам.

Так мы немного отодвинем содержимое новости от списка ее авторов и даты публикации.

```
.item-date {
  text-align: right;
  padding-right: 20px;
}
```

А этот стиль будет привязан к абзацу с датой публикации новости.

Переключимся на файл default.js и напишем код, создающий все необходимые переменные:

```
var hHeader, ctrList, ctrURL, txtURL, btnSubscribe, btnRefresh,
ctrAppBar, oFeedURL = null, arrFeed, dsrFeed;
```

Переменная `oFeedURL` будет хранить экземпляр объекта `Windows.Foundation.Uri`, представляющий интернет-адрес канала новостей. Сразу же присвоим ей значение `null`, говорящее, что подписка на канал новостей еще не выполнена. Переменная `arrFeed` будет хранить массив данных с новостями из канала, а переменная `dstFeed` — созданный на основе этого массива источник данных.

Создадим код инициализации:

```
document.addEventListener("DOMContentLoaded", function() {
  WinJS.UI.processAll().then(function() {
    hHeader = document.getElementById("hHeader");
    ctrList = document.getElementById("divList").winControl;
    ctrURL = document.getElementById("divURL").winControl;
    txtURL = document.getElementById("txtURL");
    ctrAppBar = document.getElementById("divAppBar").winControl;
    btnSubscribe = ctrAppBar.getCommandById("btnSubscribe");
    btnRefresh = ctrAppBar.getCommandById("btnRefresh");
    var btnURL = document.getElementById("btnURL");
    btnRefresh.addEventListener("click", btnRefreshClick);
    btnURL.addEventListener("click", btnURLClick);
    ctrAppBar.hideCommands([btnRefresh]);
  });
});
```

Помимо всего прочего, этот код изначально скроет кнопку **Обновить**, поскольку обновлять пока нечего.

Начнем с объявления функции, которая будет выполнена после нажатия кнопки **Подписаться на** канал всплывающего элемента:

```
function btnURLClick() {
  oFeedURL = new Windows.Foundation.Uri(txtURL.value);
  ctrURL.hide();
  ctrAppBar.showCommands([btnRefresh]);
  fillList();
}
```

Здесь мы получаем интернет-адрес, занесенный в поле ввода, создаем на его основе экземпляр объекта `Windows.Foundation.Uri`, присваиваем его переменной `oFeedURL`, убираем с экрана всплывающий элемент, выводим на экран кнопку **Обновить** и вызываем функцию, которая загрузит содержимое канала и выведет его содержимое в списке.

Сразу же объявим эту функцию:

```
function fillList() {
  var oI, sAuthors, sContent, sDate;
  arrFeed = [];
  var oFeedClient = new Windows.Web.Syndication.SyndicationClient();
  oFeedClient.retrieveFeedAsync(oFeedURL).then(function(feed) {
    hHeader.textContent = feed.title.text;
    for (var i = 0; i < feed.items.size; i++) {
      oI = feed.items[i];
      sAuthors = "";
      for (var j = 0; j < oI.authors.size; j++) {
        if (sAuthors != "") {
          sAuthors += ", ";
        }
        sAuthors += oI.authors[j].name;
      }
      if (oI.content) {
        sContent = oI.content.text;
      } else {
        sContent = oI.summary.text;
      }
      sDate = oI.publishedDate.getDate() + "." +
      (oI.publishedDate.getMonth() + 1) + "." +
      oI.publishedDate.getFullYear();
      arrFeed.push({
        title: oI.title.text,
        authors: sAuthors,
        content: sContent,
        url: oI.links[0].uri.absoluteUri,
        date: sDate
      });
    }
    dsrFeed = new WinJS.Binding.List(arrFeed);
    ctrList.itemDataSource = dsrFeed.dataSource;
  });
}
```

Осталось только объявить функцию, которая будет выполнена по нажатию на кнопке **Обновить**:

```
function btnRefreshClick() {
  fillList();
}
```

Сохраним все файлы и запустим приложение на выполнение. Подпишемся на какой-либо канал новостей и посмотрим, что отобразится в списке (рис. 17.1). Наконец попробуем обновить содержимое канала.

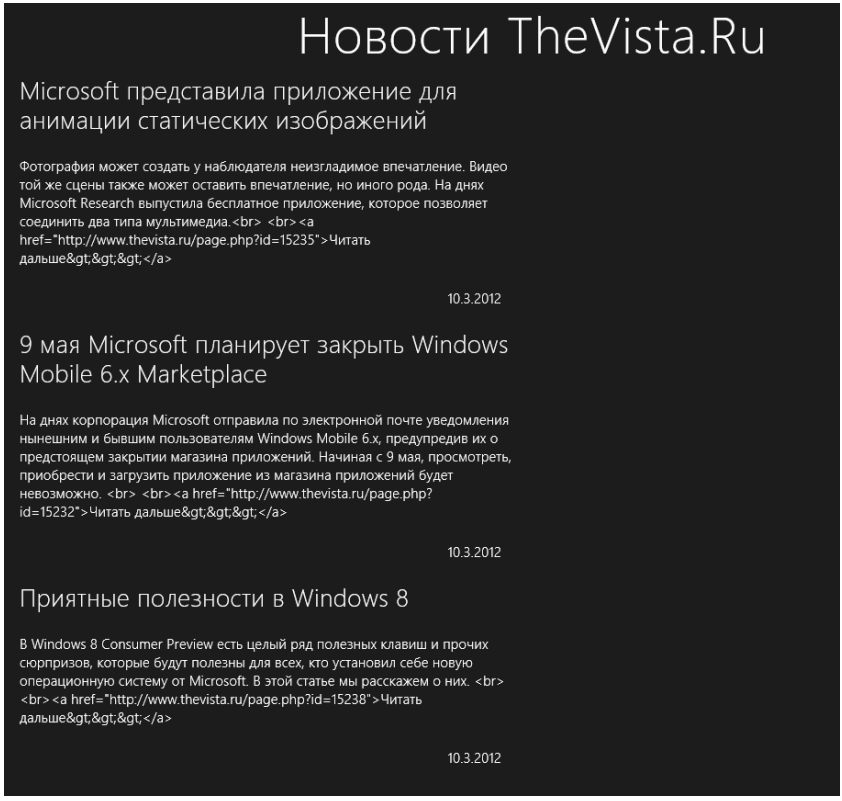


Рис. 17.1. Интерфейс второго приложения для чтения каналов новостей

Что дальше?

В этой главе мы изучали механизмы платформы Metro, призванные обеспечить поддержку каналов новостей RSS и Atom. И наконец-то создали работоспособное приложение для чтения этих каналов.

Следующая глава будет посвящена загрузке из сети информации другого рода: Web-страниц, файлов и результатов работы Web-сервисов, а именно — популярного поискового ресурса Microsoft Bing. В конце концов, планшеты немыслимы без постоянного подключения к Интернету...

ГЛАВА 18

Загрузка данных из сети

В предыдущей главе мы начали знакомиться с сетевыми возможностями платформы Metro. Мы узнали о внутренних механизмах этой платформы, призванных обеспечить поддержку каналов новостей RSS и Atom — популярного на данный момент средства распространения информации в Интернете.

Эта глава будет посвящена разговору об инструментах Metro, позволяющих загружать из сети другие данные: Web-страницы, файлы и результаты работы Web-сервисов. Заодно мы рассмотрим еще не знакомый нам элемент интерфейса — гиперссылки; в некоторых случаях они могут пригодиться.

Вывод Web-страниц

И начнем мы с рассмотрения средств, позволяющих загрузить Web-страницу и вывести ее на экран прямо в интерфейсе нашего приложения.

Фреймы

Для этого предназначены фреймы HTML. *Фрейм* — это особый элемент интерфейса, который можно рассматривать как Web-обозреватель, встроенный прямо в Metro-приложение. Мы можем открыть во фрейме любую Web-страницу, просто указав ее интернет-адрес.

Фрейм создается с помощью парного тега `<iframe>`. Содержимое у этого тега никогда не указывается.

```
<iframe id="frmMain"></iframe>
```

Атрибут `src` этого тега позволяет задать интернет-адрес Web-страницы, которая должна быть открыта во фрейме изначально.

```
<iframe id="frmMain" src="http://www.microsoft.com/"></iframe>
```

Фрейм представляется объектом `HTMLIFrameElement`. Этот объект поддерживает свойство `src`, соответствующее одноименному атрибуту тега `<iframe>`.

```
var frmMain = document.getElementById("frmMain");
frmMain.src = "http://www.thevista.ru/";
```

Открываем в нашем фрейме другую Web-страницу.

Событие `load` возникает во фрейме сразу же после окончания загрузки Web-страницы.

Следует отметить, что Web-страница, открытая во фрейме, полностью изолируется как от приложения, так и от самого устройства. Это значит, что Web-сценарии, входящие в состав Web-страницы, не смогут получить доступ к данным приложения, равно как и к датчикам, встроенным в планшет. Такой подход позволяет защитить данные пользователя от вредоносных Web-страниц.

Пример: окончательная версия приложения для чтения каналов новостей

Теперь мы можем реализовать в приложении, работающем с каналами новостей, вывод Web-страницы с полным текстом новости при выборе ее в списке.

Откроем в Visual Studio проект `FeedReader`. Откроем файл default.html, найдем в нем код, создающий блок `divContent`, и изменим его следующим образом:

```
<div id="divContent">
  <iframe id="frmContent"></iframe>
</div>
```

Мы вставили в этот блок фрейм `frmContent`, в котором и будет выводиться Web-страница с полным текстом выбранной новости.

Далее откроем файл default.css и создадим в нем новый стиль:

```
#frmContent {
  width: 100%;
  height: 100%;
}
```

Растянем этот фрейм на все пространство блока.

Переключимся на файл default.js и введем код, объявляющий необходимую переменную:

```
var frmContent;
```

Найдем код инициализации и вставим в его конец следующие выражения (выделены полужирным шрифтом):

```
document.addEventListener("DOMContentLoaded", function() {
  WinJS.UI.processAll().then(function() {
    . . .
    frmContent = document.getElementById("frmContent");
    ctrList.addEventListener("iteminvoked", ctrListItemInvoked);
  });
});
```

И объявим функцию, которая выполнится после выбора пункта в списке:

```
function ctrListItemInvoked(evt) {
  evt.detail.itemPromise.then(function(selected) {
    frmContent.src = selected.data.url;
  });
}
```

Она откроет во фрейме Web-страницу с полным содержимым выбранной в списке новости.

Сохраним исправленные файлы и запустим приложение на выполнение. Подпишемся на какой-либо канал новостей, дождемся, когда список будет заполнен, и выберем какую-либо новость. На экране должна появиться соответствующая новости Web-страница (рис. 18.1).

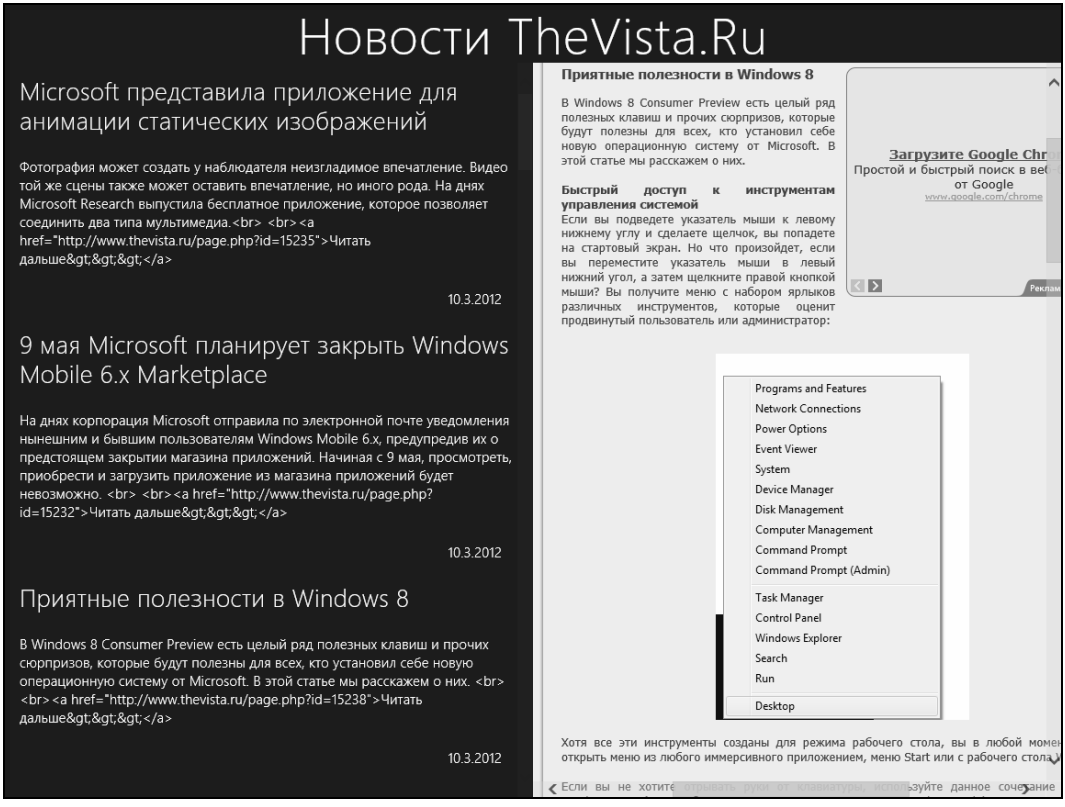


Рис. 18.1. Интерфейс второго приложения для чтения каналов новостей после модификации

Фоновая загрузка файлов

Приложения для загрузки файлов относятся к категории самых востребованных. И останутся таковыми, пока пользователи загружают из Сети файлы.

Неудивительно, что платформа Metro предоставляет встроенные средства для обеспечения загрузки файлов. И средства эти настолько просты в использовании, что пользоваться ими одно удовольствие!

Самое интересное, что загрузка файлов выполняется даже тогда, когда запустившее ее приложение неактивно. Поэтому ее часто называют фоновой.

Подготовительные действия

Перед тем как начать загрузку какого-либо файла, нам следует выполнить две подготовительные операции. Первая — создание интернет-адреса этого файла в том формате, в котором его "понимает" платформа Metro. Да-да, в виде знакомого нам по *главе 17* экземпляра объекта `Windows.Foundation.Uri`.

```
var sURL = "http://www.somesite/ru/files/file.zip";
var oURL = new Windows.Foundation.Uri(sURL);
```

Вторая подготовительная операция — создание файла, в котором будут сохранены загружаемые из удаленного файла данные *(конечный файл)*. Нам придется сделать это самим.

Microsoft рекомендует сохранять загружаемые из сети файлы в папке Загрузки. Эта папка представляется экземпляром объекта `Windows.Storage.DownloadsFolder`, созданным самой платформой Metro и доступным через одноименную переменную.

```
var oDownloads = Windows.Storage.DownloadsFolder;
```

Чтобы создать файл или папку в папке Загрузки, никаких дополнительных прав приложению давать не нужно.

Объект `Windows.Storage.DownloadsFolder` поддерживает два метода. Метод `createFileAsync` запускает процесс создания файла, а метод `createFolderAsync` — папки. Имя файла или папки, заданное в виде строки, передается соответствующему методу в качестве единственного параметра.

Оба этих метода возвращают в качестве результата обязательство, для которого мы зададим в вызове метода `then` функцию, которая выполнится по завершению создания файла или папки. Эта функция получит в качестве единственного параметра экземпляр объекта `Windows.Storage.StorageFile` или `Windows.Storage.StorageFolder`, представляющий вновь созданный файл или папку.

```
oDownloads.createFileAsync("file.zip").then(function(file) {
  //Новый файл создан
  //Выполняем загрузку файла
});
```

Сама загрузка файла будет выполняться в теле данной функции.

Загрузка файла

Чтобы загрузить файл, нам потребуется *загрузчик* — особая структура данных, инициирующая процесс загрузки. Загрузчик представляется экземпляром объекта

с длинным именем `Windows.Networking.BackgroundTransfer.BackgroundDownloader`, который мы создадим в первую очередь.

```
var oDownloader = new
Windows.Networking.BackgroundTransfer.BackgroundDownloader();
```

Имея загрузчик, мы можем создать *загрузку* — структуру, которая, собственно, и будет загружать файл. Для этого мы вызовем метод `createDownload` загрузчика.

```
<загрузчик>.createDownload(<интернет-адрес>, <конечный файл>)
```

Первым параметром этому методу передается полученный ранее интернет-адрес загружаемого файла, а вторым — конечный файл.

Метод `createDownload` возвращает экземпляр объекта `Windows.Networking.BackgroundTransfer.DownloadOperation`, представляющий загрузку.

```
var oDO = oDownloader.createDownload(oURL, file);
```

Наконец, мы запустим загрузку файла, вызвав не принимающий параметров метод `startAsync` только что полученной загрузки.

Метод `startAsync` возвращает в качестве результата обязательство, для которого мы укажем в вызове метода `then` три функции. Первая функция выполнится после успешной загрузки файла, вторая — в случае возникновения ошибки, а третья будет периодически выполняться в процессе загрузки. Эти функции мы можем использовать, чтобы информировать пользователя об успешной или неудачной загрузке и выводить сведения о ее прогрессе.

Помимо этого, нам следует сохранить возвращенное методом `startAsync` обязательство в какой-либо переменной. Это понадобится, если мы захотим реализовать прерывание загрузки.

```
var oP = oDO.startAsync().then(function() { . . . },
function() { . . . }, function() { . . . });
```

В одном загрузчике мы можем запустить сколько угодно загрузок. И все они будут работать параллельно, не мешая друг другу.

Приостановка, возобновление и прерывание загрузки

Платформа Metro предоставляет нам удобные средства для управления загрузками: приостановки, возобновления и прерывания. Сейчас мы о них поговорим.

Для приостановки загрузки достаточно вызвать метод `pause` этой самой загрузки. Данный метод не принимает параметров и не возвращает результата.

```
oDO.pause();
```

Возобновить ранее приостановленную загрузку мы можем вызовом ее метода `resume`. Он также не принимает параметров и не возвращает результата.

```
oDO.resume();
```

Если ресурс, с которого выполняется загрузка файла, поддерживает возобновление загрузки файлов, она продолжится с того места, на котором и была приостановлена. В противном случае загрузка начнется с самого начала.

Прервать загрузку мы можем вызовом метода `cancel` обязательства, полученного в результате вызова `startAsync`. Этот метод, как и два предыдущих, не принимает параметров и не возвращает результата.

```
oP.cancel();
```

> ***Внимание!***
>
> Прерывание загрузки пользователем платформа Metro расценивает как возникновение ошибки.

Получение сведений о прогрессе загрузки

Как мы уже знаем, третья из функций, что мы зададим для обязательства, возвращенного методом `startAsync`, будет периодически выполняться в процессе загрузки файла. Также мы знаем, что данная функция может использоваться для вывода прогресса загрузки.

Получить сведения о прогрессе загрузки можно из ее свойства `progress`. Оно хранит экземпляр объекта `Windows.Networking.BackgroundTransfer.BackgroundDownloadProgress`, содержащий эти сведения.

Данный объект поддерживает следующие интересные нам свойства:

- `bytesReceived` — возвращает количество загруженных байтов в виде целого числа;
- `totalBytesToReceive` — возвращает размер загружаемого файла в байтах в виде целого числа.

```
<progress id="prgDownload" />
. . .
var prgDownload = document.getElementById("prgDownload");
var oP = oDO.startAsync().then(function() { . . . },
function() { . . . }, function() {
  prgDownload.max = oDO.progress.totalBytesToReceive;
  prgDownload.value = oDO.progress.bytesReceived;
});
```

Используем индикатор прогресса для визуального отображения прогресса загрузки.

Однако если наше приложение выполняет одновременно несколько загрузок, с отображением их прогресса возникнут сложности. Нам придется создавать несколько индикаторов прогресса и отслеживать, какой из них за какой загрузкой "закреплен". Впрочем, разговор об этом выходит за рамки данной книги.

Получение сведений о возникшей ошибке

Вторая по счету функция, та, что выполняется при возникновении ошибки, получает в качестве параметра экземпляр объекта `Error`, описывающий эту ошибку. Он поддерживает свойство `message`, возвращающее текстовое описание ошибки в виде строки, которое мы можем вывести на экран.

```
<p id="pStatus"></p>
. . .
var pStatus = document.getElementById("pStatus");
var oP = oDO.startAsync().then(function() { . . . }, function(err) {
  pStatus.textContent = err.message;
}, function() { . . . });
```

Выводим текстовое описание ошибки в абзаце `pStatus`.

Возобновление загрузок, оставшихся после предыдущего запуска приложения

Если приложение, загружающее файлы, прекратит свою работу, все выполняемые им загрузки будут приостановлены самой платформой Metro и сохранены в памяти компьютера. При последующем запуске этого приложения сохраненные загрузки так же автоматически возобновятся.

Однако здесь возникает проблема. Ранее, создавая загрузку, мы указали для относящегося к ней обязательства три функции, которые будут выполняться по завершению загрузки, в случае ошибки и периодически в процессе загрузки. Когда приложение завершается, эти три функции перестанут работать (что вполне понятно, приложение-то выгружено из памяти). И при повторном запуске приложения нам придется заново указать их.

Сначала нам следует выяснить, остались ли в "наследство" от предыдущего запуска приложения какие-то загрузки. В этом нам поможет метод `getCurrentDownloadsAsync` загрузчика. Отметим сразу, что данный метод вызывается у самого объекта `Windows.Networking.BackgroundTransfer.BackgroundDownloader`, а не у его экземпляра.

Метод `getCurrentDownloadsAsync` не принимает параметров и возвращает в качестве результата обязательство, для которого мы с помощью метода `then` укажем функцию. Она выполнится после завершения процесса получения всех сохраненных загрузок и получит в качестве параметра экземпляр объекта-коллекции, который их содержит.

Это тот же самый объект-коллекция, что возвращается методом `pickMultipleFilesAsync` *(см. главу 16)*. Он поддерживает свойство `size`, хранящее размер коллекции, и метод доступа к отдельным элементам, характерный для массивов.

```
Windows.Networking.BackgroundTransfer.BackgroundDownloader.
↳getCurrentDownloadsAsync().then(function(downloads) {
  if (downloads.size > 0) {
    //Выполняем действия над приостановленными загрузками
  }
});
```

Теперь мы можем снова указать для каждой из сохраненных загрузок три функции, описанные ранее. Сделать это нам позволит метод `attachAsync` загрузки. Он не

принимает параметров и возвращает обязательство, для которого мы укажем в вызове метода `then` те самые три функции.

После этого наше приложение снова сможет обрабатывать выполняемые загрузки.

```
Windows.Networking.BackgroundTransfer.BackgroundDownloader.
↳getCurrentDownloadsAsync().then(function(downloads) {
  for (var i = 0; i < downloads.size; i++) {
    downloads[i].attachAsync().then(function() { . . . },
    function() { . . . }, function() { . . . });
  }
});
```

Загрузка также поддерживает два свойства, которые помогут нам однозначно ее идентифицировать и определить, какой файл она загружает.

Свойство `guid` возвращает строку с уникальным идентификатором данной загрузки. Этот идентификатор не будет меняться ни в коем случае.

Если наше приложение позволяет выполнять сразу несколько загрузок, для хранения сведений о них (имени конечного файла, ссылки на индикатор прогресса, в котором отображается ход загрузки, и др.) мы можем использовать хэш. (О хэшах, или ассоциативных массивах, говорилось в *главе 4*.) А в качестве индексов элементов этого хэша удобно применять уникальные идентификаторы, возвращаемые свойством `guid`.

Свойство `requestedUri` возвращает интернет-адрес загружаемого файла в виде экземпляра объекта `Windows.Foundation.Uri`.

Пример: приложение для загрузки файлов

Осталось попрактиковаться в фоновой загрузке файлов, написав соответствующее приложение. Пусть оно позволяет загружать одновременно только один файл.

Создадим в Visual Studio проект `FileDownloader`. Откроем файл default.html и введем в тег `<body>` следующий код:

```
<p>
  Интернет-адрес загружаемого файла
  <br />
  <input type="url" id="txtURL" />
</p>
<p>
  Имя конечного файла
  <br />
  <input type="text" id="txtFileName" />
</p>
<p>
  <input type="button" id="btnStart" value="Начать загрузку" />
</p>
```

```
<p>
  <progress id="prgProgress" value="0" />
</p>
<p id="pProgress">0%</p>
<p id="pStatus">Готов</p>
<p>
  <input type="button" id="btnPause" value="Приостановить" disabled />
  <input type="button" id="btnResume" value="Возобновить" disabled />
  <input type="button" id="btnCancel" value="Прервать" disabled />
</p>
```

Мы создаем поле ввода для указания интернет-адреса загружаемого файла и имени конечного файла, кнопку **Начать загрузку**, индикатор, который будет визуально отображать прогресс загрузки, два абзаца для вывода прогресса загрузки и состояния приложения и кнопки **Приостановить**, **Возобновить** и **Прервать**.

Не будем возиться с оформлением, а сразу займемся логикой. Переключимся на файл default.js и введем код, объявляющий необходимые переменные:

```
var txtURL, txtFileName, btnStart, prgProgress, pProgress, pStatus,
btnPause, btnResume, btnCancel, oDownloader, oDO, oP;
```

Напишем код инициализации:

```
document.addEventListener("DOMContentLoaded", function () {
  txtURL = document.getElementById("txtURL");
  txtFileName = document.getElementById("txtFileName");
  btnStart = document.getElementById("btnStart");
  prgProgress = document.getElementById("prgProgress");
  pProgress = document.getElementById("pProgress");
  pStatus = document.getElementById("pStatus");
  btnPause = document.getElementById("btnPause");
  btnResume = document.getElementById("btnResume");
  btnCancel = document.getElementById("btnCancel");
  btnStart.addEventListener("click", btnStartClick);
  btnPause.addEventListener("click", btnPauseClick);
  btnResume.addEventListener("click", btnResumeClick);
  btnCancel.addEventListener("click", btnCancelClick);

  oDownloader = new
  Windows.Networking.BackgroundTransfer.BackgroundDownloader();

  Windows.Networking.BackgroundTransfer.BackgroundDownloader.
  ↳getCurrentDownloadsAsync().then(function(downloads) {
    if (downloads.size > 0) {
      oDO = downloads[0];
      oP = oDO.attachAsync().then(complete, error, progress);
      pStatus.textContent = "Идет загрузка";
      txtURL.disabled = true;
      txtFileName.disabled = true;
```

```
      btnStart.disabled = true;
      btnPause.disabled = false;
      btnCancel.disabled = false;
    }
  });
});
```

Здесь мы, помимо получения доступа к элементам интерфейса и привязки обработчиков к событиям, создаем загрузчик и сразу же получаем сохраненную загрузку, если она есть.

Объявим функцию, которая выполнится после нажатия кнопки **Начать загрузку**:

```
function btnStartClick() {
  var oURL = new Windows.Foundation.Uri(txtURL.value);
  var oDownloads = Windows.Storage.DownloadsFolder;
  oDownloads.createFileAsync(txtFileName.value).then(function(file) {
    oDO = oDownloader.createDownload(oURL, file);
    oP = oDO.startAsync().then(complete, error, progress);
    prgProgress.value = 0;
    pStatus.textContent = "Идет загрузка";
    txtURL.disabled = true;
    txtFileName.disabled = true;
    btnStart.disabled = true;
    btnPause.disabled = false;
    btnResume.disabled = true;
    btnCancel.disabled = false;
  });
}
```

Начинаем загрузку файла и подготавливаем приложение для ее приостановки и прерывания. Заодно выводим в абзаце состояния текст, сообщающий о начале загрузки.

Объявим функцию, которая выполнится после завершения загрузки:

```
function complete() {
  setOnComplete();
  pStatus.textContent = "Завершено";
}
```

Она восстанавливает изначальное состояние приложения, вызвав функцию `setOnComplete()`, и выводит соответствующее сообщение в абзаце состояния.

Объявим функцию `setOnComplete()`, которая восстановит изначальное состояние приложения и подготовит его к запуску новой загрузки:

```
function setOnComplete() {
  txtURL.value = "";
  txtFileName.value = "";
  prgProgress.value = 0;
  pProgress.textContent = "0%";
```

```
  txtURL.disabled = false;
  txtFileName.disabled = false;
  btnStart.disabled = false;
  btnPause.disabled = true;
  btnResume.disabled = true;
  btnCancel.disabled = true;
}
```

Объявим функцию, которая выполнится при возникновении ошибки:

```
function error(err) {
  setOnComplete();
  pStatus.textContent = err.message;
}
```

Здесь мы восстанавливаем изначальное состояние и выводим в абзаце состояния текстовое описание ошибки.

На очереди — функция, которая будет выполняться в процессе загрузки файла и выводить ее прогресс:

```
function progress() {
  prgProgress.max = oDO.progress.totalBytesToReceive;
  prgProgress.value = oDO.progress.bytesReceived;
  pProgress.textContent = prgProgress.position * 100 + "%";
}
```

Свойство `position` индикатора прогресса возвращает число от 0 до 1, обозначающее его положение. Благодаря ему мы можем вычислять процентаж загрузки.

Осталось реализовать приостановку, возобновление и прерывание загрузки.

```
function btnPauseClick() {
  oDO.pause();
  pStatus.textContent = "Приостановлено";
  btnPause.disabled = true;
  btnResume.disabled = false;
}
function btnResumeClick() {
  oDO.resume();
  pStatus.textContent = "Идет загрузка";
  btnPause.disabled = false;
  btnResume.disabled = true;
}
function btnCancelClick() {
  oP.cancel();
  setOnComplete();
}
```

Сохраним все файлы и запустим приложение. Запустим загрузку какого-либо достаточно большого файла, приостановим ее, возобновим и подождем, пока она не завершится. Запустим загрузку другого файла и попробуем прервать ее.

Взаимодействие с удаленными Web-сервисами

Мы уже умеем выводить в Metro-приложениях Web-страницы и загружать файлы. Но разговор об интернет-возможностях этой платформы будет неполным без упоминания об инструментах для взаимодействия с удаленными Web-сервисами — отправки запросов и загрузки результатов их исполнения.

Возьмем, например, поисковый сервис Microsoft Bing. Он позволяет исполнять запросы на поиск данных, отправленные сторонними приложениями, и возвращать результаты поиска в удобном формате. Проверим все это в деле?

Формирование запроса

Сначала нам следует сформировать для удаленного Web-сервиса запрос на получение данных, например, для их поиска. В составе этого запроса мы укажем входные параметры: ключевое слово для поиска, количество найденных позиций (например, интернет-адресов), которые будут возвращены в составе одного ответа, тип искомых позиций (Web-страницы, изображения или что-то еще) и различные служебные данные.

Большинство Web-сервисов принимают запросы, сформированные согласно *методу GET*. Он устанавливает следующие правила:

- параметры запроса указываются прямо в составе интернет-адреса Web-сервиса;
- параметры запроса и их значения записываются парами вида *<имя параметра>=<значение параметра>*;
- отдельные пары "параметр — значение" отделяются друг от друга символами амперсанда (`&`);
- параметры запроса отделяются от собственно интернет-адреса Web-сервиса символом вопросительного знака (`?`);
- строковые значения параметров могут содержать только латинские буквы, цифры, символы подчеркивания, точки и двоеточия. Если в значении какого-либо параметра имеются другие символы, например кириллические буквы или пробелы, они должны быть закодированы особым образом.

```
http://api.bing.net/json.aspx?Query=Metro&Sources=Image
```

Здесь мы сформировали согласно правилам метода GET запрос к Web-сервису Bing. Этот запрос включает два параметра: `Query` со значением `Metro` и `Sources` со значением `Image`.

Для кодирования строковых значений параметров в подходящий для использования в запросе вид мы можем использовать встроенную JavaScript-функцию `encodeURIComponent()`. В качестве единственного параметра она принимает строку и возвращает ее в закодированном виде.

```
var sSource = "Метро";
var sEncoded = encodeURIComponent(sSource);
var sQuery = "http://api.bing.net/json.aspx?Query=" + sEncoded +
"&Sources=Image"
```

Здесь мы включили в состав запроса закодированную строку "Метро".

Кстати, функция `decodeURIComponent()` выполняет обратную задачу — преобразование закодированной строки в обычный вид.

Набор параметров, поддерживаемых конкретным Web-сервисом, описывается в разделе его документации, предназначенном для разработчиков.

Отправка запроса и получение ответа

Создав запрос, мы можем его отправить. Для этого предназначен метод `xhr` объекта `WinJS`. (Объект `WinJS` предоставляет несколько служебных свойств и методов. Единственный его экземпляр создается платформой Metro и доступен из одноименной переменной.)

В качестве единственного параметра метод `xhr` принимает экземпляр объекта `Object`, хранящий параметры запроса, которые задаются в различных его свойствах. Нас интересует свойство `url`, в котором указывается строка с полностью сформированным запросом.

Метод `xhr` возвращает обязательство. Для него мы в вызове метода `then` укажем функцию, которая выполнится после получения ответа от Web-сервиса. В качестве единственного параметра она получит экземпляр объекта, представляющий полученный ответ.

```
var oParams = {
  url: sQuery
};
WinJS.xhr(oParams).then(function(response) {
  //Ответ от Web-сервиса получен
  //Выполняем его обработку
});
```

Обработка ответа

Получить результаты обработки нашего запроса можно из свойства `responseText` данного объекта. Это свойство возвращает строку, представляющую собой JavaScript-код, который формирует экземпляр объекта `Object`, хранящий полученный результат.

Такой формат передачи данных по сети — в виде формирующего их JavaScript-кода — сейчас очень популярен. Он носит название *JSON* (JavaScript Object Notation, объектная нотация JavaScript).

Получив этот код, мы его выполним. Да-да, укажем платформе Metro, чтобы она его выполнила и выдала нам сам экземпляр объекта `Object`, что описывается этим

кодом. Для этого мы используем метод `parse` объекта `JSON`. (`JSON` — один из встроенных в JavaScript объектов. Единственный его экземпляр создается платформой Metro и доступен из одноименной переменной.)

Метод `parse` принимает в качестве единственного параметра строку с JavaScript-кодом и возвращает экземпляр объекта `Object` с данными.

```
var oResponse = JSON.parse(response.responseText);
```

Набор свойств, поддерживаемых этим экземпляром объекта, зависит от конкретного Web-сервиса и описывается в его документации для разработчиков.

Введение в Bing API

Теперь кратко рассмотрим возможности Web-сервиса Microsoft Bing, предназначенные для разработчиков приложений, которые будут с ним взаимодействовать. Иначе говоря, его интерфейс разработки приложений, или *API* (Application Programming Interface). Конечно, целиком его мы разбирать не будем — для этого понадобится отдельная книга, — а познакомимся только с некоторыми его возможностями, теми, которые потом задействуем в нашем пробном приложении, предназначенном для поиска изображений по ключевому слову.

> ***Внимание!***
>
> Любое приложение, которое будет работать с Microsoft Bing, обязано в составе каждого запроса указывать свой уникальный идентификатор, так называемый AppID. Этот идентификатор можно бесплатно получить на Web-странице с интернет-адресом **http://www.bing.com/developers/appids.aspx**.

> ***На заметку***
>
> Полное описание Bing API приведено в разделе MSDN, расположенном по интернет-адресу **http://msdn.microsoft.com/en-us/library/dd251056.aspx**.

Параметры запроса

Запрос, отправляемый Bing, должен включать следующие обязательные параметры:

- `Query` — задает ключевое слово для поиска;
- `Sources` — задает тип искомых данных. Для поиска изображений этот параметр должен иметь значение `Image`;
- `AppID` — уникальный идентификатор приложения;
- `Image.Count` — количество результатов, возвращаемых в составе одного ответа;
- `Image.Offset` — номер первого результата, возвращаемого в составе данного ответа. На забываем, что нумерация и в этом случае начинается с нуля.

Для получения ответа в формате JSON запрос следует отправлять на интернет-адрес **http://api.bing.net/json.aspx**. (Bing также может отправлять ответы в формате XML; в этом случае запрос следует отправлять по другому интернет-адресу.)

```
http://api.bing.net/json.aspx?Query=Metro&Sources=Image&
↳AppID=<идентификатор приложения>&Image.Count=10&Image.Offset=30
```

Этот запрос укажет Bing найти все изображения, связанные с ключевым словом "Metro", и вернуть в качестве ответа десять найденных изображений, начиная с 31-го.

Содержание ответа

Возвращаемый Bing ответ после обработки, как уже говорилось ранее, представляет собой экземпляр объекта `Object`. Он имеет весьма сложную структуру и хранит множество данных.

Прежде всего, он содержит свойство `SearchResponse`, которое, собственно, и хранит данные ответа в виде экземпляра объекта `Object`. Этот экземпляр объекта, в свою очередь, содержит свойство `Image`, включающее сведения о найденных изображениях также в виде экземпляра объекта `Object`. Последний содержит три свойства:

- `Total` — общее количество найденных изображений, включая и те, что не вошли в состав данного ответа, в виде целого числа. Отметим, что оно всегда вычисляется приблизительно;
- `Offset` — порядковый номер первого изображения, входящего в состав данного ответа, из всех найденных. Указывается в виде целого числа;
- `Results` — массив, каждый элемент которого представляет собой экземпляр объекта `Object` и хранит данные об одном найденном изображении. Фактически это знакомый нам по *главе 14* массив данных.

```
var iCount = oResponse.SearchResponse.Image.Total;
var arrResults = oResponse.SearchResponse.Image.Results;
```

Получаем общее количество найденных изображений и сами эти изображения.

Экземпляр объекта `Object`, описывающий найденное изображение, поддерживает следующие свойства:

- `Title` — заголовок в виде строки;
- `Url` — интернет-адрес Web-страницы, содержащей данное изображение, в виде строки;
- `MediaUrl` — интернет-адрес файла, хранящего само изображение, в виде строки;
- `Width` — ширина изображения в пикселах в виде целого числа;
- `Height` — высота изображения в пикселах в виде целого числа;
- `Thumbnail` — сведения о миниатюре изображения, сгенерированной самим Bing. Представляются в виде экземпляра объекта `Object`, имеющего следующие свойства:
  - `Url` — интернет-адрес файла с миниатюрой в виде строки;
  - `Width` — ширина миниатюры в пикселах в виде целого числа;
  - `Height` — высота миниатюры в пикселах в виде целого числа.

```
var arrURLs = [];
var arrThumbnailURLs = [];
for (var i = 0; i < arrResults.length; i++) {
  arrURLs.push(arrResults[i].Url);
  arrThumbnailURLs.push(arrResults[i].Thumbnail.Url);
}
```

Помещаем интернет-адреса файлов с самими изображениями и их миниатюрами в массивы `arrURLs` и `arrThumbnailURLs`.

Пример: приложение для поиска изображений по ключевому слову

Что ж, Bing API нам теперь подвластен. Давайте создадим приложение, которое будет искать в Сети изображения по указанному ключевому слову и выводить их миниатюры в списке. А после того как пользователь выберет в списке какую-либо миниатюру, оно выведет соответствующее изображение в отдельном блоке.

Сначала получим AppID для этого приложения. (Как это сделать, было рассказано ранее.) Когда будем его получать, зададим в качестве имени приложения `ImageSearcher`.

После этого создадим в Visual Studio новый проект `ImageSearcher`. Откроем файл default.html и впишем в тег `<body>` такой код:

```
<div id="divListTemplate" data-win-control="WinJS.Binding.Template">
  <div class="image-placement">
    <img data-win-bind="src: thumbnailUrl" />
  </div>
  <div class="title-placement">
    <p data-win-bind="textContent: title"></p>
  </div>
</div>

<div id="divSearch">
  <input type="text" id="txtKeyword" />
  <input type="button" id="btnSearch" value="Искать" />
</div>
<div id="divList" data-win-control="WinJS.UI.ListView"
data-win-options="{layout: {type: WinJS.UI.ListLayout},
↳itemTemplate: divListTemplate, selectionMode: 'single'}"></div>
<div id="divViewer" data-win-control="WinJS.UI.ViewBox">
  <img id="imgViewer" />
</div>
<div id="divNavigation">
  <input type="button" id="btnPrevious" value="Предыдущие 10" disabled />
  <input type="button" id="btnNext" value="Следующие 10" disabled />
</div>
```

Мы создали четыре блока, которые потом разместим на экране с помощью сеточной разметки. Верхний блок включит в себя поле ввода для указания ключевого слова и кнопку **Искать**. Левый блок мы превратим в список Metro; он будет перечислять миниатюры и заголовки найденных изображений. Правый блок превратится в панель вывода Metro, включающую элемент графического изображения, в котором будет выводиться изображение, выбранное в списке. А нижний блок будет содержать кнопки **Предыдущие 10** и **Следующие 10**; с помощью этих кнопок мы будем загружать предыдущую и следующую порцию найденных изображений.

Также мы создали шаблон для списка. Он включит в себя два блока, которые мы также разместим на экране с помощью сеточной разметки. Левый блок включит элемент графического изображения для вывода миниатюры, а в правом блоке будет выводиться заголовок изображения.

Откроем файл default.css и создадим в нем стили, перечисленные далее.

```
body {
  display: -ms-grid;
  -ms-grid-columns: 400px 1fr;
  -ms-grid-rows: 40px 1fr 32px;
}
#divSearch {
  -ms-grid-column-span: 2;
  -ms-grid-column-align: end;
}
#divList {
  -ms-grid-row: 2;
  height: 100%;
}
#divViewer {
  -ms-grid-row: 2;
  -ms-grid-column: 2;
  -ms-grid-column-align: center;
  -ms-grid-row-align: center;
}
#divNavigation {
  -ms-grid-row: 3;
  -ms-grid-column-span: 2;
  -ms-grid-column-align: end;
}
#divList .win-item {
  height:  100px;
  display: -ms-grid;
  -ms-grid-columns: 180px 1fr;
  -ms-grid-rows: 1fr;
}
.image-placement { width: 180px; }
.title-placement { -ms-grid-column: 2; }
```

Переключимся на файл default.js и сразу же введем код, создающий необходимые переменные:

```
var txtKeyword, ctrList, imgViewer, btnPrevious, btnNext, sKeyword,
arrResults, dsrResults;
```

Объявим еще четыре переменные:

```
var sAppID = <идентификатор AppID нашего приложения>;
var iCurrent, iTotal;
var iResultCount = 10;
```

Понятно, что первая переменная будет хранить идентификатор AppID нашего приложения. Вторая сохранит порядковый номер первого изображения, загружаемого в составе текущего запроса, из всех найденных Bing. Третья сохранит общее количество найденных изображений, возвращенное Bing. А четвертая будет хранить количество изображений, которые Bing будет возвращать в состав одного ответа; зададим его равным 10.

Код инициализации будет следующим:

```
document.addEventListener("DOMContentLoaded", function() {
  WinJS.UI.processAll().then(function() {
    txtKeyword = document.getElementById("txtKeyword");
    var btnSearch = document.getElementById("btnSearch");
    ctrList = document.getElementById("divList").winControl;
    imgViewer = document.getElementById("imgViewer");
    btnPrevious = document.getElementById("btnPrevious");
    btnNext = document.getElementById("btnNext");

    btnSearch.addEventListener("click", btnSearchClick);
    ctrList.addEventListener("iteminvoked", ctrListItemInvoked);
    btnPrevious.addEventListener("click", btnPreviousClick);
    btnNext.addEventListener("click", btnNextClick);
  });
});
```

Объявим функцию, которая запустит поиск изображений:

```
function btnSearchClick() {
  sKeyword = txtKeyword.value;
  iCurrent = 0;
  fillList();
}
```

Здесь мы сохраняем введенное пользователем ключевое слово в объявленной ранее переменной, указываем, чтобы изображения загружались, начиная с самого первого из найденных, и вызываем функцию `fillList()`, которая запустит поиск и выведет его результаты в списке.

Сразу же объявим эту функцию:

```
function fillList() {
  var oResponse, arrR;
```

```
  var sQuery = "http://api.bing.net/json.aspx?Query=" +
  encodeURIComponent(sKeyword) + "&Sources=Image&AppID=" + sAppID +
  "&Image.Count=" + iResultCount + "&Image.Offset=" + iCurrent;
  WinJS.xhr({url: sQuery}).then(function(response) {
    arrResults = [];
    oResponse = JSON.parse(response.responseText);
    iTotal = oResponse.SearchResponse.Image.Total;
    arrR = oResponse.SearchResponse.Image.Results;
    for (var i = 0; i < arrR.length; i++) {
      arrResults.push({
        title: arrR[i].Title,
        thumbnailUrl: arrR[i].Thumbnail.Url,
        url: arrR[i].MediaUrl
      });
    }
    dsrResults = new WinJS.Binding.List(arrResults);
    ctrList.itemDataSource = dsrResults.dataSource;
    btnPrevious.disabled = (iCurrent == 0);
    btnNext.disabled = (iCurrent + iResultCount > iTotal);
  });
}
```

Она отправит Bing запрос, получит ответ, сформирует на его основе источник данных и привяжет к нему список. Наконец, она будет управлять доступностью кнопок **Предыдущие 10** и **Следующие 10**.

Объявим функцию, которая выведет на экран изображение, соответствующее выбранному пользователем пункту списка:

```
function ctrListItemInvoked(evt) {
  evt.detail.itemPromise.then(function (selected) {
    imgViewer.src = selected.data.url;
  });
}
```

Наконец, объявим функции, которые будут выполнять переход на предыдущие и следующие 10 найденных изображений:

```
function btnPreviousClick() {
  iCurrent -= iResultCount;
  if (iCurrent < 0) {
    iCurrent = 0;
  }
  fillList();
}
function btnNextClick() {
  iCurrent += iResultCount;
  fillList();
}
```

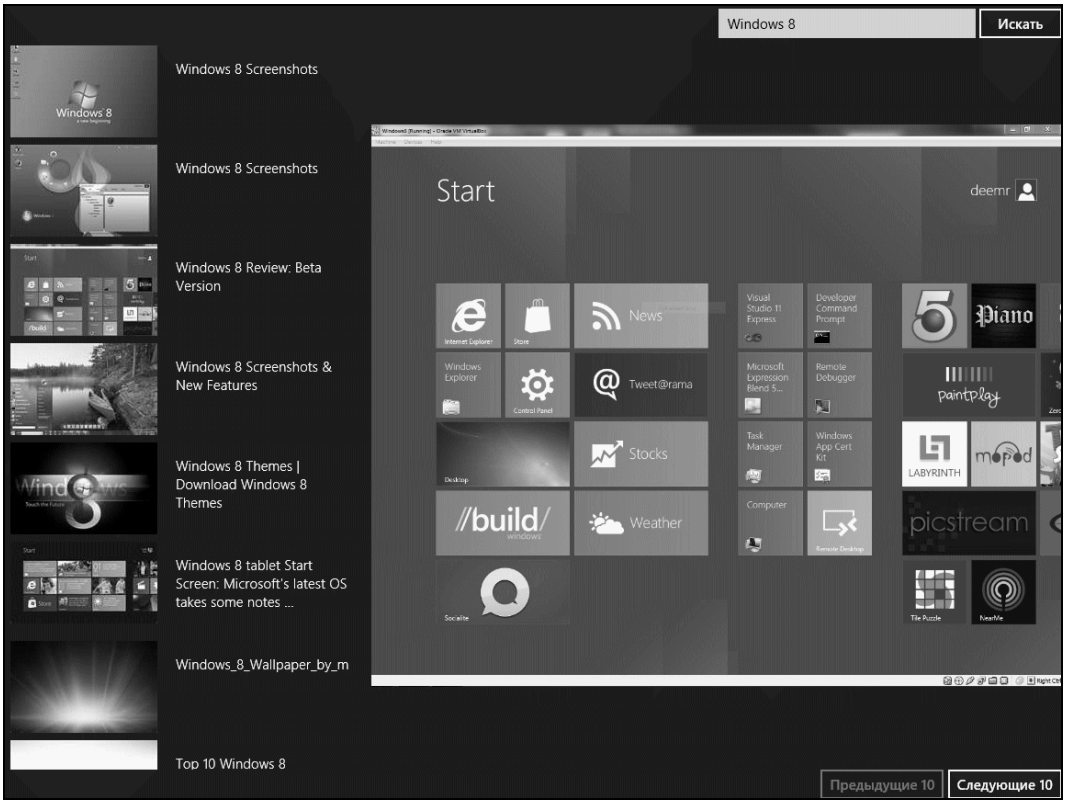


Рис. 18.2. Интерфейс приложения для поиска изображений в Интернете

Сохраним все файлы и запустим приложение на выполнение. Введем в поле ввода какое-либо ключевое слово и запустим поиск. Когда список заполнится, выберем любой его пункт и посмотрим на изображение, что появится на экране (рис. 18.2). Попробуем перейти на следующие и предыдущие 10 изображений из всех найденных Bing. После этого выполним поиск по другому ключевому слову.

Гиперссылки

Осталось рассмотреть один весьма примечательный элемент интерфейса, обычный на Web-страницах, но в Metro-приложениях применяемый нечасто. Это *гиперссылки* — фрагменты текста или графические изображения, при нажатии на которые выполняется переход по определенному интернет-адресу.

Гиперссылка создается с применением парного тега `<a>`. Внутри этого тега помещается текст или изображение, которое и станет гиперссылкой.

```
<p>Перейти на Web-сайт
<a href="http://www.microsoft.ru/">Microsoft</a>.</p>
```

Здесь мы превратили слово "Microsoft" в гиперссылку, указывающую на Web-сайт этой компании.

```
<a href="http://www.microsoft.ru/">
↳<img src="/images/microsoft_logo.jpg" /></a>
```

А этот код превратит в гиперссылку графическое изображение.

Как видим, гиперссылка представляет собой встроенный элемент интерфейса.

Обязательный атрибут `href` тега `<a>` служит для указания интернет-адреса, на который следует выполнить переход.

По умолчанию Web-страница, на которую указывает гиперссылка, будет открыта в обычном Web-обозревателе. Но мы можем сделать так, чтобы она открылась во фрейме, входящем в состав интерфейса нашего приложения. (О фреймах говорилось в начале этой главы.) Для этого платформа Metro предоставляет нам два пути.

Путь первый — указание нужного фрейма прямо в HTML-коде, создающем гиперссылку. Сначала мы укажем имя фрейма, воспользовавшись атрибутом `name` тега `<iframe>`. Обычно там указывают то же имя, что и в атрибуте тега `id`.

```
<iframe id="frmMain" name="frmMain"></iframe>
```

Потом мы зададим это же имя в атрибуте `target` тега `<a>`, что создает гиперссылку:

```
<p>Перейти на Web-сайт
<a href="http://www.microsoft.com/" target="frmMain">Microsoft</a>.</p>
```

Второй путь — программное открытие Web-страницы. Сначала мы привяжем к событию `click` гиперссылки обработчик.

```
<p>Перейти на Web-сайт
<a id="aLink" href="http://www.microsoft.com/">Microsoft</a>.</p>
<iframe id="frmMain" name="frmMain"></iframe>
. . .
var aLink = document.getElementById("aLink");
var frmMain = document.getElementById("frmMain");
aLink.addEventListener("click", function(evt) {
  evt.preventDefault();
  frmMain.src = aLink.href;
});
```

Метод `preventDefault` объекта `Event`, представляющего событие, отменяет действие по умолчанию, которое платформа Metro выполняет при наступлении данного события. В случае гиперссылки таким действием по умолчанию является переход по указанному в ней интернет-адресу. Данный метод не принимает параметров и не возвращает результата.

Гиперссылка представляется объектом `HTMLAnchorElement`. Этот объект поддерживает свойства `href` и `target`, соответствующие одноименным атрибутам тега.

Что дальше?

В этой главе мы рассматривали механизмы платформы Metro, призванные выполнять загрузку из Сети самых разнообразных данных: Web-страниц, файлов и результатов работы Web-сервисов. Еще мы познакомились с гиперссылками — особыми элементами интерфейса, не самыми популярными в Metro-программировании, но в некоторых случаях могущими оказаться полезными.

Следующая глава будет посвящена передаче данных от одного Metro-приложения другому. Для этого Metro предоставляет стандартные средства, удобные и гибкие.

ЧАСТЬ VI

Обмен данными и работа с устройствами

ГЛАВА 19

Обмен данными между Metro-приложениями

Предыдущая глава рассказывала о реализации в Metro-приложениях загрузки данных из Интернета. Мы научились выводить Web-страницы, загружать файлы и работать с удаленными Web-сервисами, в частности с популярным поисковым сервисом Microsoft Bing. И, как обычно, для практики написали пару приложений.

Эта глава будет посвящена обмену данными между Metro-приложениями, работающими на одном устройстве. Мы узнаем, как можно стандартными средствами передать от одного приложения другому текст, графическое изображение или файл. А платформа Metro, как всегда, снабдит нас подходящими инструментами.

Передача данных

Сначала рассмотрим передачу данных от одного приложения другому, чтобы было что потом принимать...

Подготовительные действия

Первое, что нам следует сделать, — указать платформе Metro, что наше приложение может передавать данные другим приложениям, и, стало быть, пользователь может использовать для этого соответствующий стандартный модуль (Share charm) Windows 8.

Сначала мы вызовем не принимающий параметров метод `getForCurrentView` объекта с очень длинным, в стиле платформы Metro, именем `Windows.ApplicationModel.DataTransfer.DataTransferManager`. Отметим, что данный метод следует вызывать у самого объекта, а не у его экземпляра.

Метод `getForCurrentView` возвращает в качестве результата экземпляр упомянутого ранее объекта. Назовем его *диспетчером обмена*.

```
var oDTM = Windows.ApplicationModel.DataTransfer.DataTransferManager.
↳getForCurrentView();
```

Диспетчер обмена поддерживает событие `datarequested`, возникающее, когда пользователь начинает процедуру передачи данных. Функция-обработчик этого события получит в качестве единственного параметра экземпляр объекта `Windows.ApplicationModel.DataTransfer.DataRequestedEventArgs`, хранящий сведения об этом событии.

Данный объект поддерживает свойство `request`, которое хранит *запрос передачи* — структуру, представляющую отправляемые данные. Запрос передачи представляется экземпляром объекта `Windows.ApplicationModel.DataTransfer.DataRequest`.

Из всех свойств запроса передачи нас сейчас интересует только `data`. Оно хранит *набор передаваемых данных* — структуру, хранящую сами данные, что передаются в составе операции обмена. Набор передаваемых данных представляется экземпляром объекта `Windows.ApplicationModel.DataTransfer.DataPackage`.

```
oDTM.addEventListener("datarequested", function(evt) {
  var oDR = evt.request;
  var oDP = oDR.data;
  //Подготавливаем данные для отправки
});
```

Привязываем к событию `datarequested` диспетчера обмена обработчик и получаем в его теле сначала запрос передачи, а потом — набор передаваемых данных.

Собственно передача данных

Перед тем как реализовывать в приложениях возможность обмена данными, нам следует уяснить несколько простых правил:

- платформа Metro позволяет пересылать между приложениями данные следующих видов: текст, интернет-адреса, графические изображения и наборы файлов;
- мы можем включить в состав одного набора передаваемых данных единицы данных, относящиеся к различным видам. Например, мы можем переслать за один раз фрагмент текста, графическое изображение и набор файлов;
- мы можем включить в состав одного набора передаваемых данных только одну единицу данных любого из перечисленных ранее видов. Так, мы можем переслать за раз только один фрагмент текста и только одно графическое изображение; переслать два изображения или два фрагмента текста мы уже не можем;
- набор файлов может включать в себя сколько угодно файлов любых типов.

А только после этого следует знакомиться с собственно средствами, предназначенными для отправки данных.

Передача текста

Проще всего отправить другому приложению фрагмент текста. Для этого служит метод `setText` набора передаваемых данных. В качестве единственного параметра

он принимает строку, содержимое которой следует передать другому приложению, и не возвращает результата.

```
var s = "Платформа Metro";
oDP.setText(s);
```

Передача интернет-адреса

Интернет-адрес отправить также несложно. Сначала мы преобразуем его в экземпляр объекта `Windows.Foundation.Uri`, а потом передадим в качестве единственного параметра не возвращающему результата методу `setUri` набора передаваемых данных.

```
var sURL = "http://www.microsoft.com/";
var oURL = new Windows.Foundation.Uri(sURL);
oDP.setUri(oURL);
```

Передача графических изображений

Отправка графического изображения выполняется в три этапа.

На первом этапе мы получим файл, хранящий это изображение. Как это сделать, мы знаем из *главы 16*.

На втором этапе мы получим поток для этого файла. Это поток особого рода, представляемый объектом `Windows.Storage.Streams.RandomAccessStreamReference`. Получить его можно, вызвав метод `createFromFile` данного объекта (не экземпляра!) и передав ему в качестве параметра открытый файл. Результатом, возвращенным методом `createFromFile`, и будет этот поток.

На третьем этапе мы вызовем метод `setBitmap` набора передаваемых данных. Этот метод принимает в единственном параметре полученный нами поток и не возвращает результата.

```
var libDocuments = Windows.Storage.KnownFolders.documentsLibrary;
var sFileName = "microsoft_logo.jpg";
var oStream;
libDocuments.getFileAsync(sFileName).then(function(file) {
  if (file) {
    oStream = new Windows.Storage.Streams.RandomAccessStreamReference.
    ↳createFromFile(file);
    oDP.setBitmap(oStream);
  }
})
```

Передаем графическое изображение, хранящееся в файле microsoft_logo.jpg из библиотеки Документы.

> ***Внимание!***
>
> Microsoft рекомендуется выполнять отправку графического изображения одновременно и в качестве графического изображения, и в качестве файла. (Отправка файлов описывается чуть позже.)

Передача файлов

Отправить файлы проще, чем графическое изображение. Достаточно только вызвать метод `setStorageItems` набора данных.

```
<набор передаваемых данных>.setStorageItems(<массив файлов>)
```

Единственным параметром этому методу передается массив передаваемых файлов, представленных экземплярами объекта `Windows.Storage.StorageFile`. Это может быть как обычный массив JavaScript, так и коллекция, формируемая методами `pickMultipleFilesAsync`, `getFilesAsync` и т. п.

Метод `setStorageItems` не возвращает результата.

```
var libDocuments = Windows.Storage.KnownFolders.documentsLibrary;
var arrFileNames = ["letter.doc", "microsoft_logo.jpg", "readme.txt"];
var arrPromises = [], arrFiles = [], oP;
for (var i = 0; i < arrFileNames.length; i++) {
  oP = libDocument.getFileAsync(sFileName).then(function(file) {
    arrFiles.push(file);
  });
  arrPromises.push(oP);
}
WinJS.Promise.join(arrPromises).then(function() {
  oDP.setStorageItems(arrFiles);
}
```

Выполняем передачу файлов letter.doc, microsoft_logo.jpg и readme.txt, хранящихся в библиотеке Документы.

Мы уже знаем, что метод `getFileAsync` всего лишь запускает операцию получения файла. Когда эта операция закончится, неизвестно. И уж тем более неизвестно, когда закончатся несколько одновременно выполняющихся операции получения файла (в нашем случае их три).

К счастью, платформа Metro предоставляет нам изящный способ решить эту проблему. Это метод `join` объекта `WinJS.Promise`. Он вызывается у самого объекта `WinJS.Promise`, а не у его экземпляра, принимает в качестве единственного параметра массив обязательств и возвращает также обязательство. Функция, которую мы укажем для него в методе `then`, выполнится только после выполнения всех обязательств, что входят в упомянутый ранее массив.

Посмотрим еще раз на код примера. Мы сохранили все обязательства, возвращенные вызовами метода `getFilesAsync`, в массив `arrPromises`. Этот массив мы передали методу `join`, а в теле функции, указанной для возвращенного им обязательства, мы выполнили отправку полученных файлов.

Такой подход к обработке одновременно выполняющихся операций рекомендует сама Microsoft.

Задание параметров передаваемых данных

Платформа Metro позволяет нам указать некоторые параметры передаваемых данных. Более того, она требует это сделать.

Набор передаваемых данных поддерживает свойство `properties`. Оно содержит экземпляр объекта `Windows.ApplicationModel.DataTransfer.DataPackagePropertySet`, который и хранит эти параметры. Из всех поддерживаемых им свойств нам будут интересны следующие:

- `title` — задает заголовок передаваемых данных в виде строки;
- `description` — задает более развернутое описание передаваемых данных в виде строки;

  НА ЗАМЕТКУ

  Эти свойства являются обязательными к заполнению.

- `thumbnail` — задает миниатюру для передаваемого графического изображения в виде потока — экземпляра объекта `Windows.Storage.Streams.RandomAccessStreamReference`. (Как получить этот поток, было рассказано ранее.) Часто этому свойству присваивают тот же поток, что указывается в вызове метода `setBitmap`.

  Это свойство рекомендуется заполнять в случае, если выполняется передача графического изображения. Впрочем, по опыту автора, задавать его необязательно — обмен данных прекрасно выполняется и без этого;
- `fileTypes` — задает список типов передаваемых файлов. Хранит массив, каждый элемент которого представляет собой расширение, характерное для определенного типа файлов, заданное с начальной точкой в виде строки. Для указания типов файлов удобно использовать метод `replaceAll`, упомянутый в *главе 16*.

  Это свойство рекомендуется указывать, если выполняется передача файлов.

```
var libDocuments = Windows.Storage.KnownFolders.documentsLibrary;
var sFileName = "microsoft_logo.jpg";
var oStream;
oDP.properties.title = "Логотип Microsoft";
oDP.properties.description = "Проверка средств Metro по пересылке
↳файлов";
oDP.properties.fileTypes.replaceAll([".jpg"]);
libDocuments.getFileAsync(sFileName).then(function(file) {
  if (file) {
    oStream = new Windows.Storage.Streams.RandomAccessStreamReference.
    ↳createFromFile(file);
    oDP.setBitmap(oStream);
    oDP.properties.thumbnail = oStream;
    oDP.setStorageItems([file]);
  }
})
```

Отправляем графическое изображение одновременно в качестве графического изображения и файла, задав при этом все описанные ранее параметры.

Отложенная передача и передача по требованию

Когда пользователь начинает передачу данных, приложение должно предоставить все передаваемые данные в течение определенного времени. Однако если эти данные велики по объему (например, представляют собой графическое изображение), их подготовка может затянуться. В таком случае платформа Metro, скорее всего, сочтет, что передающее приложение перестало работать, и завершит его.

Metro предлагает два разных подхода к решению этой проблемы. Сейчас мы поговорим о них.

Отложенная передача

Первый подход заключается в реализации *отложенной передачи*, когда передающее приложение запрашивает у платформы Metro дополнительное время на подготовку данных.

Перед тем как поместить их в набор передаваемых данных, мы вызовем не принимающий параметров метод `getDeferral` запроса передачи. Тем самым мы сообщим Metro, что приложение начало подготовку данных и требует дополнительного времени на ее завершение.

Метод `getDeferral` возвращает экземпляр объекта `Windows.ApplicationModel.DataTransfer.DataRequestDeferral`, представляющий отложенную передачу. Этот экземпляр объекта мы обязательно сохраним в какой-либо переменной.

Когда все передаваемые данные будут помещены в набор, мы вызовем метод `complete` полученного ранее экземпляра объекта `Windows.ApplicationModel.DataTransfer.DataRequestDeferral`. Этот метод не принимает параметров, не возвращает результатов и сообщает Metro, что приложение наконец-то подготовило все нужные данные к передаче.

```
var libDocuments = Windows.Storage.KnownFolders.documentsLibrary;
var sFileName = "microsoft_logo.jpg";
var oStream, oDef;
oDP.properties.title = "Логотип Microsoft";
oDP.properties.description = "Проверка средств Metro по пересылке
↳файлов";
oDef = oDR.getDeferral();
libDocuments.getFileAsync(sFileName).then(function(file) {
  if (file) {
    oStream = new Windows.Storage.Streams.RandomAccessStreamReference.
    ↳createFromFile(file);
    oDP.properties.thumbnail = oStream;
    oDP.setBitmap(oStream);
    oDP.setStorageItems([file]);
    oDef.complete();
  }
})
```

Реализуем отложенную передачу графического изображения.

Передача по требованию

Второй подход — это *передача данных по требованию*. Заключается он в том, что передающее приложение выполнит отправку данных только после того, как принимающее приложение пожелает их получить.

Первое действие, которое нам следует выполнить для реализации передачи по требованию, — указание функции, которая поместит передаваемые данные в набор. Для этого мы применим метод `setDataProvider` набора передаваемых данных.

```
<набор передаваемых данных>.setDataProvider(<вид данных>,
<функция, выполняющая передачу данных>)
```

Первым параметром указывается вид данных, которые будет подготавливать указанная функция. Его значением должен быть один из следующих элементов перечисления `Windows.ApplicationModel.DataTransfer.StandardDataFormats`:

- `text` — фрагмент текста;
- `uri` — интернет-адрес;
- `bitmap` — графическое изображение;
- `storageItems` — набор файлов.

Вторым параметром передается функция, которая, собственно, и будет выполнять отложенную передачу данных указанного ранее формата.

Метод `setDataProvider` не возвращает результата.

Вторым нашим действием будет объявление передающей данные функции — той, что указывается вторым параметром в вызове метода `setDataProvider`.

Эта функция примет в качестве единственного параметра *требование* — структуру, которая представляет запрос на получение данных, пришедший от принимающего их приложения. Требование представляется экземпляром объекта `Windows.ApplicationModel.DataTransfer.DataProviderRequest`.

Требование поддерживает знакомый нам метод `getDeferral`. Он уведомляет платформу Metro о том, что приложение требует дополнительного времени на подготовку данных к отправке.

А еще требование поддерживает метод `setData`. Он выполняет передачу данных, которые были переданы ему в качестве единственного параметра, и не возвращает результата.

```
function sendBitmap(request) {
  var sFileName = "microsoft_logo.jpg";
  var libDocuments = Windows.Storage.KnownFolders.documentsLibrary;
  var oDef = request.getDeferral();
  libDocuments.getFileAsync(sFileName).then(function(file) {
    if (file) {
      var oStream = new Windows.Storage.Streams.
      ↳RandomAccessStreamReference.createFromFile(file);
      request.setData(oStream);
```

```
      oDef.complete();
    }
  })
}
. . .
oDP.properties.title = "Логотип Microsoft";
oDP.properties.description = "Проверка средств Metro по пересылке файлов";
oDP.setDataProvider(Windows.ApplicationModel.DataTransfer.
↳StandardDataFormats.bitmap, sendBitmap);
```

Реализуем передачу по требованию графического изображения.

Мы можем указать сразу несколько функций, каждая из которых будет передавать данные в определенном виде. Так, одна функция может передавать графическое изображение в виде собственно графического изображения, а другая — в виде файла.

Получение данных

Итак, данные мы отправили. Теперь их нужно как-то получить и что-то с ними сделать. Как это выполняется, мы сейчас узнаем.

Подготовительные действия

Прежде чем начать реализацию в приложении получения данных, нам следует выполнить подготовительные действия. Какие — мы сейчас узнаем.

Указание прав приложения на получение данных

Сначала нам следует дать приложению права на получение данных определенного вида. А если мы собираемся дать приложению права на получение наборов файлов, то должны указать, файлы каких форматов оно может принимать.

Откроем окно параметров приложения, как делали это в *главе 16*. И сразу переключимся на вкладку **Declarations** этого окна (см. рис. 16.2).

Выберем в раскрывающемся списке **Available Declarations** пункт **Share Target** и нажмем кнопку **Add**. В списке **Supported Declarations** появится пункт **Share Target**, который мы также выберем. Окно задания параметров приложения изменит свой вид (рис. 19.1).

Доступные приложению виды принимаемых данных задаются в группе элементов управления **Data Formats**.

Чтобы добавить приложению поддержку какого-либо вида данных, следует нажать находящуюся в этой группе кнопку **Add New**. В ответ там появится вложенная группа элементов управления, включающая поле ввода **Data Format**. Укажем в нем поддерживаемый вид данных в виде одного из следующих значений: `text` (фрагменты текста), `uri` (интернет-адреса), `bitmap` (графические изображения) и `storageitems` (наборы файлов).

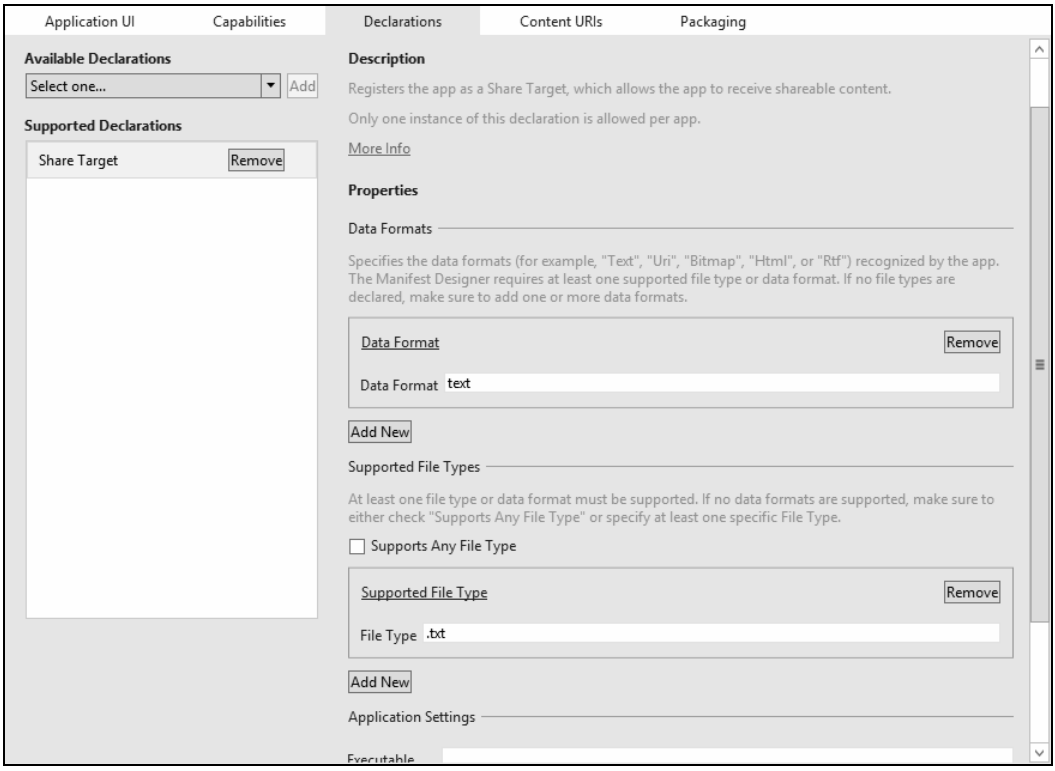


Рис. 19.1. Содержимое вкладки **Declarations** окна для задания параметров приложения при указании поддерживаемых видов данных

Чтобы дать приложению поддержку еще одного вида данных, следует нажать кнопку **Add New**. Чтобы удалить ненужную группу и, соответственно, поддерживаемый приложением вид данных, достаточно нажать присутствующую в заголовке соответствующей вложенной группы кнопку **Remove**.

Если наше приложение должно принимать наборы файлов, нам следует указать доступные для него форматы принимаемых файлов. Это выполняется в уже знакомой нам по *главе 16* группе элементов управления **Supported File Types**. А чтобы дать приложению поддержку всех форматов файлов, достаточно установить флажок **Supports Any File Type**.

Закончив задание прав приложения, следует закрыть окно документа, в котором выводятся его параметры, и сохранить его содержимое.

Отслеживание активизации приложения

В процессе передачи данных пользователь выбирает приложение, которое получит эти данные. Как только он его выберет, платформа Metro автоматически активизирует данное приложение, а если оно еще не запущено, выполнит его запуск. В любом случае в экземпляре объекта `WinJS.Application`, представляющем приложение

(об этом объекте упоминалось еще в *главе 5*), возникнет событие `activated`. В обработчике этого события и выполняются все действия по получению данных.

Интересно, что Visual Studio сам формирует для нас "заготовку" обработчика этого события при создании любого проекта. Если мы посмотрим на код, хранящийся в файле default.js какого-либо созданного нами приложения, то без особого труда ее отыщем. Вот она:

```
app.onactivated = function(eventObject) {
  . . .
};
```

Нам останется только удалить тело обработчика и вписать туда нужный нам код. Чем мы сейчас и займемся.

Функция — обработчик события `activated` получает в качестве единственного параметра экземпляр знакомого нам объекта `CustomEvent`. Его свойство `detail` хранит дополнительные сведения о событии в виде экземпляра объекта `Object`.

В случае события `activated` данный экземпляр объекта будет содержать свойство `kind`. Оно хранит один из элементов перечисления `Windows.ApplicationModel.Activation.ActivationKind`, указывающий, каким именно образом было активизировано приложение. Если приложение было активизировано для получения данных, это будет элемент `shareTarget`.

```
app.onactivated = function(eventObject) {
  if (eventObject.detail.kind ===
  Windows.ApplicationModel.Activation.ActivationKind.shareTarget) {
    //Приложение было активизировано для получения данных
    //Получаем эти данные
  }
};
```

Определение вида полученных данных

Как мы уже знаем, передающее приложение может поместить в состав набора передаваемых данных несколько единиц информации, относящихся к различным видам. Мы сами передавали, в частности, графическое изображение как в виде собственно графического изображения, так и в виде хранящего его файла.

Поэтому принимающее приложение должно проверить, какие виды данных оно получило. Сделать это совсем несложно.

Экземпляр объекта `Object`, хранящийся в свойстве `detail`, поддерживает еще и свойство `shareOperaton`. Оно хранит *запрос получения* — структуру данных, описывающую саму операцию получения данных. Запрос получения представляется экземпляром объекта `Windows.ApplicationModel.DataTransfer.ShareTarget.ShareOperation`.

Данный объект поддерживает свойство `data`, которое хранит *набор получаемых данных* — структуру, представляющую принимаемые данные. Набор получаемых данных является экземпляром объекта `Windows.ApplicationModel.DataTransfer.DataPackageView`.

```
var oSO = eventObject.detail.shareOperation;
var oDPW = oSO.data;
```

Набор получаемых данных поддерживает метод `contains`. В качестве единственного параметра он принимает наименование вида данных как элемент перечисления `Windows.ApplicationModel.DataTransfer.StandardDataFormats` и возвращает `true`, если данные этого вида присутствуют в наборе, и `false` в противном случае.

```
if (oDPW.contains(Windows.ApplicationModel.DataTransfer.
↳StandardDataFormats.text)) {
  //Принятые данные содержат фрагмент текста
}
if (oDPW.contains(Windows.ApplicationModel.DataTransfer.
↳StandardDataFormats.storageItems)) {
  //Принятые данные содержат набор файлов
}
```

Собственно получение данных

Теперь рассмотрим процедуру собственно получения данных, относящихся к различным видам.

Получение текста

Получить текст очень просто — вызовом не принимающего параметров метода `getTextAsync` набора получаемых данных.

В качестве результата данный метод вернет обязательство, для которого мы с помощью метода `then` укажем функцию. Она выполнится после завершения процесса получения переданного текста и получит в качестве единственного параметра сам этот текст в виде строки.

```
var sText;
oDPW.getTextAsync().then(function(text) {
  sText = text;
});
```

Получение интернет-адреса

За получение интернет-адреса "отвечает" не принимающий параметров метод `getUriAsync` набора получаемых данных. Он вернет обязательство, мы укажем для него в вызове метода `then` функцию, а она получит в качестве параметра экземпляр объекта `Windows.Foundation.Uri`, хранящий переданный нам интернет-адрес.

```
var oURL, sURL;
oDPW.getUriAsync().then(function(uri) {
  oURI = uri;
  sURI = uri.absoluteUri;
});
```

Получение графического изображения

Получить графическое изображение можно, вызвав метод `getBitmapAsync` набора получаемых данных. Он вызывается точно так же, как рассмотренные нами ранее методы `getTextAsync` и `getUriAsync`, только функция, которую мы зададим для обязательства, получит в качестве параметра поток — экземпляр объекта `Windows.Storage.Streams.RandomAccessStreamReference`.

Чтобы вывести принятое изображение на экран, сначала следует открыть данный поток. Это выполняется вызовом его метода `openReadAsync`. Он не принимает параметров и возвращает в качестве результата обязательство. Функция, что мы укажем для него в вызове метода `then`, выполнится после открытия потока и получит в качестве параметра *типизированный поток*. Этот поток хранит данные определенного типа (в нашем случае — графическое изображение определенного формата) и представляется экземпляром особого объекта.

Далее мы преобразуем типизированный его поток в другой — представляемый экземпляром объекта `msStream`. Для этого служит метод `createBlobFromRandomAccessStream` объекта `MSApp`. (Объект `MSApp` представляет несколько служебных методов для работы с файлами и потоками. Единственный экземпляр этого объекта создается платформой Metro и доступен через одноименную переменную.)

```
MSApp.createBlobFromRandomAccessStream(<MIME-тип данных типизированного
↳потока>, <типизированный поток>)
```

Первым параметром передается строка с MIME-типом данных, хранящихся в потоке; его можно получить из свойства `contentType` типизированного потока. А сам этот поток передается вторым параметром.

Метод `createBlobFromRandomAccessStream` возвращает то, что нам нужно, — поток другого формата, представленный экземпляр объекта `msStream`.

Полученный поток мы можем преобразовать в вид, пригодный к выводу на экран в элементе графического изображения. Для этого мы применим давно знакомый нам метод `createObjectURL` объекта `URL`.

```
<img id="imgOutput" />
. . .
var imgOutput = document.getElementById("imgOutput");
var mssStream;
oDPW.getBitmapAsync().then(function(stream) {
  stream.openReadAsync().then(function(typedStream) {
    mssStream = MSApp.createBlobFromRandomAccessStream
    ↳(typedStream.contentType, typedStream);
    imgOutput.src = URL.createObjectURL(mssStream);
  });
});
```

Выводим полученное графическое изображение в элементе `imgOutput`.

Получение набора файлов

Получение набора файлов производится вызовом не принимающего метода `getStorageItemsAsync` набора получаемых данных. Этот метод вызывается так же, как и все рассмотренные нами ранее методы. Функция, которую мы зададим для обязательства, получит в качестве параметра коллекцию принятых файлов, аналогичную той, что формирует метод `pickMultipleFilesAsync`.

```
var oTF = Windows.Storage.ApplicationData.current.temporaryFolder;
oDPW.getStorageItemsAsync().then(function(files) {
  for (var i = 0; i < files.size; i++) {
    files[i].copyAsync(oTF).then(function() { });
  }
});
```

Получаем файлы и копируем их во временное хранилище (подробнее о хранилищах приложения *см. в главе 16*).

Получение параметров принятых данных

При передаче данных мы задали для них некоторые параметры, в частности заголовок, описание и, возможно, миниатюру. Можно ли прочитать их в приложении, получившем эти данные?

Конечно, можно. Набор получаемых данных поддерживает свойство `properties`, которое хранит представляющий эти параметры экземпляр объекта `Windows.ApplicationModel.DataTransfer.DataPackagePropertySetView`. В нем имеются уже знакомые нам свойства `title`, `description`, `thumbnail` и `fileTypes`.

```
<p id="pTitle"></p>
<p id="pDecription"></p>
<p><img id="imgThumbnail" /></p>
. . .
var pTitle = document.getElementById("pTitle");
var pDescription = document.getElementById("pDescription");
var imgThumbnail = document.getElementById("imgThumbnail");
var mssStream;
pTitle.textContent = oDPW.properties.title;
pDescription.textContent = oDPW.properties.description;
if (oDPW.properties.thumbnail) {
  oDPW.properties.thumbnail.openReadAsync().then(function(typedStream) {
    mssStream = MSApp.createBlobFromRandomAccessStream
    ↳(typedStream.contentType, typedStream);
    imgThumbnail.src = URL.createObjectURL(mssStream);
  });
}
```

Выводим на экран заголовок, описание и миниатюру, если она была указана.

Завершение получения данных

Платформа Metro предусматривает следующий сценарий обмена данными. Пользователь выполняет передачу данных из какого-либо приложения, указывая при этом приложение-получатель. Последнее активизируется в прикрепленном (snapped) виде, получает данные, обрабатывает их определенным образом и по окончании обработки деактивируется. В результате пользователь может продолжать работу в приложении, передавшем данных.

Чтобы реализовать такое поведение в своем Metro-приложении, нам следует по окончании обработки принятых данных вызвать метод `reportCompleted` запроса получения. Этот метод не принимает параметров и не возвращает результата.

```
var sText;
oDPW.getTextAsync().then(function(text) {
  sText = text;
  //Выполняем обработку полученного текста
  oSO.reportCompleted();
});
```

Отложенное получение данных

Как и в случае с передачей данных, их получение должно быть выполнено в течение определенного времени. Если приложение задержится с получением данных (например, начнет копировать куда-либо большой набор полученных файлов), платформа Metro может счесть его неработающим и завершить.

В таком случае приложение должно затребовать у Metro дополнительное время на получение данных. Для этого мы, перед тем как начать процесс получения данных, вызовем метод `reportStarted` запроса получения; данный метод не принимает параметров и не возвращает результата.

Кроме того, после собственно получения данных, но до начала их обработки рекомендуется вызвать метод `reportDataRetrieved` запроса получения, который также не принимает параметров и не возвращает результата. Он сообщит платформе Metro, что данные приняты, и отправившее их приложение может быть приостановлено или вообще завершено.

```
oSO.reportStarted();
var oTF = Windows.Storage.ApplicationData.current.temporaryFolder;
oDPW.getStorageItemsAsync().then(function(files) {
  oSO.reportDataRetrieved();
  for (var i = 0; i < files.size; i++) {
    files[i].copyAsync(oTF).then(function() { });
  }
  oSO.reportComplete();
});
```

Выполняем отложенное получение набора файлов.

Пример: реализация передачи и получения данных

В качестве примера давайте доработаем написанные нами ранее приложения.

- Приложению для просмотра изображений `ImageViewer` дадим возможность передачи данных. Пусть оно передает путь к выбранному в нем графическому файлу в виде текста, само выбранное изображение и хранящий его файл.
- Приложению текстового редактора `TextEditor` дадим возможность приема данных — пути к выбранному файлу.

Кроме того, мы создадим еще одно, тестовое, приложение, которое будет принимать графическое изображение и хранящий его файл. Изображение оно выведет на экран, а файл скопирует во временное хранилище и выведет путь к полученной копии.

Модификация приложения для просмотра изображений

Откроем в Visual Studio решение `ImageViewer`. Переключимся на файл default.js и введем код, объявляющий необходимые переменные:

```
var oDTM, oSelected = null;
```

Далее найдем код инициализации и добавим в него следующие выражения (выделены полужирным шрифтом):

```
document.addEventListener("DOMContentLoaded", function () {
  WinJS.UI.processAll().then(function () {
    . . .
    oDTM = Windows.ApplicationModel.DataTransfer.DataTransferManager.
    ↳getForCurrentView();
    oDTM.addEventListener("datarequested", oDTMDataRequest);
  });
});
```

Исправим код, объявляющий функцию `ctrListItemInvoked()`, чтобы он выглядел так:

```
function ctrListItemInvoked(evt) {
  evt.detail.itemPromise.then(function (selected) {
    oSelected = selected.data;
    imgViewer.src = URL.createObjectURL(oSelected);
    divFileName.textContent = oSelected.name;
  });
}
```

Здесь мы сохраняем выбранный пункт в объявленной ранее переменной `oSelected`.

Объявим обработчик события `datarequest` диспетчера обмена:

```
function oDTMDataRequest(evt) {
  var oDP = evt.request.data;
  if (oSelected) {
    oSelected = selectedItems[0];
    oDP.properties.title = "Выбранное изображение";
    oDP.properties.description = "Изображение, выбранное в приложении
    ↳ImageViewer";
    oDP.properties.fileTypes.replaceAll([".gif", ".jpg", ".png"]);
    oDP.setText(oSelected.path);
    oDP.setStorageItems([oSelected]);
    oDP.setDataProvider(Windows.ApplicationModel.DataTransfer.
    ↳StandardDataFormats.bitmap, sendBitmap);
  }
}
```

Он будет выполнять передачу фрагмента текста и файла.

Объявим функцию, которая выполнит передачу изображения по требованию:

```
function sendBitmap(request) {
  var oDef = request.getDeferral();
  var oStream = new Windows.Storage.Streams.
  ↳RandomAccessStreamReference.createFromFile(oSelected);
  request.setData(oStream);
  oDef.complete();
}
```

Сохраним исправленный файл.

Добавление существующего проекта в решение

Перед тем как Visual Studio сможет запустить Metro-приложение, оно должно выполнить некоторые подготовительные действия. Эти действия включают в себя объединение всех файлов, входящих в проект, в готовое к установке приложение, установку этого приложения в системе Windows и создание для него плитки в меню **Пуск** (Start).

Когда мы запускаем приложение, Visual Studio выполняет все эти действия автоматически. Мы также можем выполнить подготовку Metro-приложения вручную — для этого достаточно вызвать пункт **Deploy Solution** меню **Build**.

Но когда мы начнем испытывать приложения, обменивающиеся данными, то столкнемся с необходимостью выполнить все эти действия сразу над несколькими приложениями: теми, что будут отправлять данные, и теми, что будут их получать. Конечно, можно последовательно, один за другим, открывать проекты всех этих приложений и выполнять их запуск, но это слишком трудоемко. Есть более удобный способ.

Еще в *главе 2* мы познакомились с решением Visual Studio — сущностью, которая позволит объединить сразу несколько проектов. Все проекты, входящие в состав

решения, обрабатываются — открываются, готовятся к запуску и закрываются — как единое целое. Однако запускаться в каждый момент времени будет только один проект — одно приложение (его называют *запускаемым проектом*).

Следовательно, мы можем объединить в рамках одного решения сразу два или даже больше проектов приложений, обменивающихся данными. Проект приложения, которое должно передавать данные, мы сделаем запускаемым. Принимающие данные приложения сразу запускать необязательно — их достаточно только подготовить к запуску.

Сейчас нам нужно добавить уже существующий проект `TextEditor` в состав решения `ImageViewer`. Выберем в списке панели **SOLUTION EXPLORER** "корень" (он, как мы знаем, представляет само решение) и щелкнем на нем правой кнопкой мыши. В появившемся на экране контекстном меню выберем пункт **Add**, а затем — в подменю — пункт **Existing Project**. В стандартном диалоговом окне открытия файла, которое появится на экране, отыщем файл нужного нам проекта и откроем его.

Открытый нами проект будет добавлен в состав решения. В списке панели **SOLUTION EXPLORER** он будет представлен отдельной "ветвью".

Теперь сделаем проект `ImageViewer` запускаемым — ведь именно это приложение будет пересылать данные. Для этого выберем "ветвь" списка, представляющую его, щелкнем на нем правой кнопкой мыши и выберем в появившемся на экране контекстном меню пункт **Set as StartUp Project**.

Пункт списка, соответствующий запускаемому проекту, выделяется полужирным шрифтом.

Модификация приложения — текстового редактора

Так, проект текстового редактора `TextEditor` в решение `ImageViewer` мы добавили. Давайте сразу же его доработаем.

Сначала дадим этому приложению право на получения фрагментов текста. Как это сделать, было описано ранее.

Далее откроем файл default.js данного проекта, найдем в нем код, привязывающий обработчик к событию `activated`, и изменим его следующим образом:

```
app.onactivated = function(eventObject) {
  if (eventObject.detail.kind ===
  Windows.ApplicationModel.Activation.ActivationKind.shareTarget) {
    var oSO = eventObject.detail.shareOperation;
    var oDPW = oSO.data;
    if (oDPW.contains(Windows.ApplicationModel.DataTransfer.
    ↳StandardDataFormats.text)) {
      oDPW.getTextAsync().then(function(text) {
        txtEdit.value = text;
      });
    }
  }
};
```

В этом примере мы не стали вызывать метод `reportCompleted`. В результате приложение текстового редактора после получения данных останется активным, и мы сразу увидим, получило оно данные или нет.

Сохраним исправленный файл, выполним подготовку решения, выбрав пункт **Deploy Solution** меню **Build**, и запустим его. Когда приложение для просмотра графики появится на экране, выберем в списке любой файл и осуществим передачу данных текстовому редактору. Если в его области редактирования появился путь к выбранному файлу, значит, передача данных успешно выполнилась.

Создание нового проекта в составе решения

Теперь создадим в составе решения `ImageViewer` проект тестового приложения, которое будет получать графическое изображение и файл. Этот проект мы назовем `ImageReceiver`.

Выберем "корень" списка в панели **SOLUTION EXPLORER** и щелкнем на нем правой кнопкой мыши. В контекстном меню, которое после этого появится на экране, выберем пункт **Add**, а затем, в появившемся подменю, — пункт **New Project**. На экране появится диалоговое окно **Add New Project**, похожее на уже знакомое нам по *главе 2* окно **New Project** (см. рис. 2.2). Введем в него все нужные данные и нажмем кнопку **OK**.

Вновь созданный проект будет также добавлен в состав решения.

Создание тестового приложения, принимающего данные

Первым же делом дадим приложению `ImageReceiver` право на получения графических изображений и файлов форматов GIF, JPEG и PNG.

Откроем файл default.html нового проекта и введем в тег `<body>` такой код:

```
<div id="divFilePath"></div>
<div id="divViewer" data-win-control="WinJS.UI.ViewBox">
  <img id="imgViewer" />
</div>
<div id="divButtons">
  <input type="button" id="btnComplete" value="Завершить" />
</div>
```

Мы создали три блока, которые потом разместим на экране с применением сеточной разметки. Верхний блок будет использоваться для вывода пути к копии принятого файла. Средний же блок мы превратим в панель вывода Metro; его содержимым станет элемент графического изображения, в котором мы выведем полученное изображение. Нижний блок будет содержать кнопку **Завершить**, которая завершит процесс обмена данными (фактически — вызовет метод `reportComplete`).

Откроем файл default.css и создадим стили, перечисленные далее.

```
body {
  display: -ms-grid;
  -ms-grid-columns: 1fr;
  -ms-grid-rows: 40px 1fr;
}
#divViewer {
  -ms-grid-row: 2;
  -ms-grid-column-align: center;
  -ms-grid-row-align: center;
}
#divButtons {
  -ms-grid-row: 3;
  -ms-grid-column-align: end;
}
```

Переключимся на файл default.js и напишем код, объявляющий необходимые переменные:

```
var divFilePath, imgViewer, oSO;
```

Напишем код инициализации:

```
document.addEventListener("DOMContentLoaded", function() {
  WinJS.UI.processAll().then(function() {
    divFilePath = document.getElementById("divFilePath");
    imgViewer = document.getElementById("imgViewer");
    var btnComplete = document.getElementById("btnComplete");
    btnComplete.addEventListener("click", function() {
      oSO.reportComplete();
    });
  });
});
```

Отыщем код, привязывающий обработчик к событию `activated`, и изменим его, чтобы он выглядел так:

```
app.onactivated = function(eventObject) {
  if (eventObject.detail.kind ===
  Windows.ApplicationModel.Activation.ActivationKind.shareTarget) {
    var oSO = eventObject.detail.shareOperation;
    var oDPW = oSO.data;
    if (oDPW.contains(Windows.ApplicationModel.DataTransfer.
    ↳StandardDataFormats.bitmap)) {
      var mssStream;
      oSO.reportStarted();
      oDPW.getBitmapAsync().then(function(stream) {
        oSO.reportDataRetrieved();
        stream.openReadAsync().then(function(typedStream) {
```

```
            mssStream = MSApp.createBlobFromRandomAccessStream
            ↳(typedStream.contentType, typedStream);
            imgViewer.src = URL.createObjectURL(mssStream);
          });
        });
      }
      if (oDPW.contains(Windows.ApplicationModel.DataTransfer.
      ↳StandardDataFormats.storageItems)) {
        oSO.reportStarted();
        var oTF = Windows.Storage.ApplicationData.current.temporaryFolder;
        oDPW.getStorageItemsAsync().then(function(files) {
          oSO.reportDataRetrieved();
          files[0].copyAsync(oTF).then(function(copy) {
            divFilePath.textContent = copy.path;
          });
        });
      }
    }
};
```

Сохраним все исправленные файлы, подготовим решение, выбрав пункт **Deploy Solution** меню **Build**, и запустим его. Выберем в приложении просмотра графики любой файл и выполним передачу данных тестовому приложению. Если последнее успешно выведет изображение и путь к копии полученного файла, мы все сделали правильно.

Что дальше?

В этой главе мы познакомились с механизмами платформы Metro, предназначенными для передачи данных от одного приложения другому. Мы научились передавать и принимать текст, интернет-адреса, графические изображения и файлы — данные, с которыми пользователи обычно имеют дело.

В следующей главе мы узнаем, как осуществить доступ к содержимому флэш-дисков и встроенным фото- и видеокамерам. Да-да, мы научимся фотографировать и снимать кино!

ГЛАВА 20

Работа с флэш-дисками и камерами

В предыдущей главе мы занимались передачей данных от одного Metro-приложения другому. Мы научились передавать текст, интернет-адреса, графические изображения и наборы файлов.

В этой главе мы займемся взаимодействием с устройствами, подключенными к компьютеру либо входящими в его состав: флэш-дисками и цифровыми фото- и видеокамерами. Мы узнаем, каким образом можно активизировать определенное приложение при подключении флэш-диска с файлами заданного типа и получить со встроенной камеры фотографию или видеоролик.

Работа с флэш-дисками

Когда пользователь подключает к компьютеру флэш-диск, система выводит на экран сообщение, предлагающее выполнить над содержимым этого диска какие-либо действия, используя установленные на компьютере приложения. Так, если подключить диск с видеофайлами, система предложит воспроизвести эти файлы в одном из установленных видеопроигрывателей. Технология, реализующая все это, носит название *AutoPlay*.

Давайте же узнаем, как применять AutoPlay в своих приложениях. Пользователям это, несомненно, придется по вкусу.

Указание прав приложения на доступ к флэш-дискам и набора поддерживаемых им команд

Первое, что нам предстоит сделать, — дать приложению права на доступ к флэш-дискам (изначально ни одно приложение такими правами не обладает) и набор команд, которые оно будет поддерживать.

Откроем окно параметров приложения, как делали это в *главе 16*, и переключимся на его вкладку **Capabilities** (см. рис. 16.1). Установим флажок **Removable Storage**, находящийся в списке **Capabilities**. Тем самым мы дадим приложению права на доступ к флэш-дискам.

Следующее наше действие — задание набора поддерживаемых приложением команд AutoPlay. Например, приложение видеопроигрывателя может поддерживать команду на воспроизведение первого из файлов, находящихся на диске, и команду на добавление всех файлов в список воспроизведения. Список всех этих команд будет выводиться в сообщении, которое платформа Metro выведет на экран при подключении флэш-диска.

Для каждой из создаваемых команд нам следует указать следующие сведения.

- *Событие AutoPlay*, которое возникает в системе в случае наличия на флэш-диске файлов определенного типа. Так, если флэш-диск содержит видеофайлы, возникает событие `PlayVideoFilesOnArrival`, а если там хранятся графические файлы — событие `ShowPicturesOnArrival`.

  Каждая команда, которую мы создадим для приложения, должна быть связана с определенным событием. Например, команду на воспроизведение видеофайла, хранящегося на флэш-диске, мы свяжем с упомянутым ранее событием `PlayVideoFilesOnArrival`.

  > ***ВНИМАНИЕ!***
  >
  > Полный список всех событий AutoPlay, поддерживаемых платформой Metro, можно найти на Web-странице с интернет-адресом **http://msdn.microsoft.com/en-us/library/windows/apps/hh452731.aspx**.

- Уникальный идентификатор команды, который будет передан приложению; по этому идентификатору можно будет выяснить, какую команду выбрал пользователь. Так, в качестве идентификатора команды на воспроизведение видеофайла мы можем указать `play`.
- Название команды, которое будет отображаться в сообщении AutoPlay.

Переключимся на вкладку **Declarations** окна параметров приложения. Выберем в раскрывающемся списке **Available Declarations** пункт **AutoPlay Content** и нажмем кнопку **Add**. В списке **Supported Declarations** появится пункт **AutoPlay Content**; выберем его. Окно задания параметров приложения тотчас изменит свой вид (рис. 20.1).

Поддерживаемые приложением команды AutoPlay указываются в группе элементов управления **Launch Actions**.

Чтобы создать новую команду, следует нажать находящуюся в данной группе кнопку **Add New**. В ответ там появится вложенная группа элементов управления **Launch Action**, где и задаются параметры создаваемой команды. Наименование события, с которым будет связана команда, вводится в поле ввода **Content Event**, идентификатор команды — в поле ввода **Verb**, а ее название — в поле ввода **Action Display Name**.

Чтобы добавить еще одну группу **Launch Action**, дав тем самым приложению поддержку еще одной команды, мы снова нажмем кнопку **Add New**. Чтобы удалить ненужную группу и, соответственно, команду, следует нажать присутствующую в группе кнопку **Remove**.

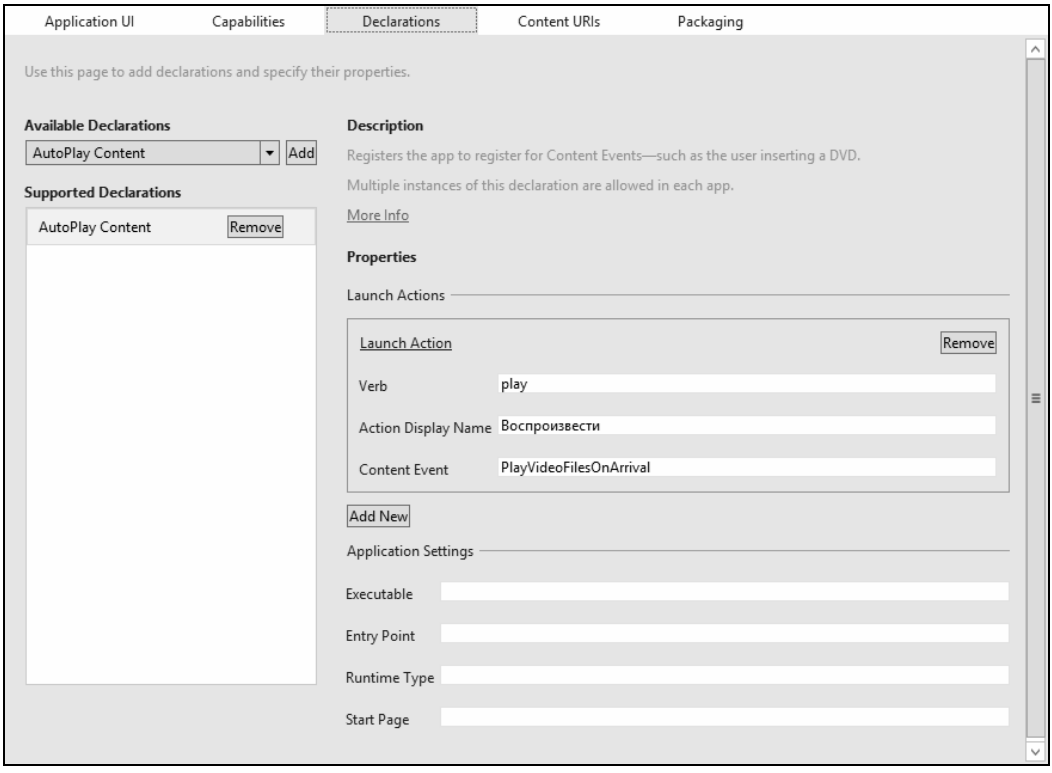


Рис. 20.1. Содержимое вкладки **Declarations** окна для задания параметров приложения при задании параметров поддержки AutoPlay

После этого нам следует обязательно задать типы файлов, с которыми будет работать наше приложение. Как это делается, было описано в *главе 16*: выбираем в раскрывающемся списке **Available Declarations** пункт **File Type Associations**, нажимаем кнопку **Add**, выбираем в списке **Supported Declarations** пункт **File Type Associations** и т. д.

Закончив задание прав приложения, закроем окно документа, в котором выводятся его параметры, и не забудем сохранить его содержимое.

Собственно работа с флэш-дисками

Работа с файлами, хранящимися на подключенном к компьютеру флэш-диске, выполняется в обработчике события `activated` объекта `WinJS.Application`. Мы уже имели дело с этим обработчиком в *главе 19*, когда реализовывали получение данных от другого приложения. Заготовку для этого обработчика создал сам Visual Studio, так что нам останется вписать в него нужный код.

Функция-обработчик события `activated` получит в качестве единственного параметра экземпляр объекта `CustomEvent` со свойством `detail`. Это свойство хранит экземпляр объекта `Object`, свойства которого представляют различные сведения о возникшем событии.

Если приложение было активизировано через команду AutoPlay, свойство `kind` этого экземпляра объекта будет хранить элемент `file` перечисления `Windows.ApplicationModel.Activation.ActivationKind`.

```
app.onactivated = function(eventObject) {
  if (eventObject.detail.kind ===
  Windows.ApplicationModel.Activation.ActivationKind.file) {
    //Приложение было активизировано через команду AutoPlay
    //Получаем файлы, хранящиеся на флэш-диске
  }
};
```

Экземпляр объекта `Object` из свойства `detail` в этом случае будет содержать еще два свойства. Свойство `verb` будет хранить идентификатор выбранной пользователем команды AutoPlay в виде строки. А свойство `files` будет хранить массив, единственным элементом которого станет корневая папка подключенного флэш-диска.

```
app.onactivated = function(eventObject) {
  if (eventObject.detail.kind ===
  Windows.ApplicationModel.Activation.ActivationKind.file) {
    if (eventObject.detail.verb == "play") {
      var oRootFolder = eventObject.detail.files[0];
      //Что-либо делаем с файлами и папками, хранящимися на флэш-диске
    }
  }
};
```

Пример: доработка приложения видеопроигрывателя для поддержки AutoPlay

Практиковаться мы будем на приложении видеопроигрывателя, которым не занимались уже довольно давно. Пусть оно при подключении к компьютеру флэш-диска начинает воспроизводить первый из хранящихся там видеофайлов.

Откроем в Visual Studio проект `VideoPlayer`. Дадим приложению право на доступ к флэш-дискам. Создадим команду `play` и свяжем ее с событием AutoPlay `PlayVideoFilesOnArrival`. И дадим приложению права на работу с типами файлов avi, mp4 и wmv.

Переключимся на файл default.js, отыщем код, привязывающий к событию `activated` обработчик, и изменим его, чтобы он выглядел так:

```
app.onactivated = function(eventObject) {
  if (eventObject.detail.kind ===
  Windows.ApplicationModel.Activation.ActivationKind.file) {
    if (eventObject.detail.verb == "play") {
      var oRootFolder = eventObject.detail.files[0];
      oRootFolder.getFilesAsync().then(function(files) {
        if (files) {
```

```
            vidMain.src = URL.createObjectURL(files[0]);
          }
        });
      }
    }
};
```

Сохраним исправленный файл и запустим приложение. Подключим к компьютеру флэш-диск, содержащий видеофайлы. Когда на экране появится сообщение AutoPlay, выберем зарегистрированную нами команду. Если после этого наше приложение начало воспроизводить первый из хранящихся на диске видеофайлов, значит, мы все сделали правильно.

Получение фото

Получение фото со встроенной камеры средствами платформы Metro выполняется очень просто. Сейчас мы в этом убедимся.

Подготовительные действия

Сначала нам следует дать приложению права на доступ к встроенной фотокамере. Для этого мы откроем окно параметров приложения, переключимся на его вкладку **Capabilities** (см. рис. 16.1) и установим флажок **Webcam** в списке **Capabilities**. После чего закроем данное окно, не забыв сохранить его содержимое.

Собственно получение фото

Работа с фотокамерой в Metro-приложениях осуществляется через специальное диалоговое окно — *окно камеры*. Это окно предоставляет возможность просмотреть изображение, получаемое камерой в данный момент, задать параметры изображения, выбрать файл, в котором оно будет сохранено, и собственно выполнить съемку.

Это диалоговое окно представляется объектом `Windows.Media.Capture.CameraCaptureUI`. Нам следует создать экземпляр данного объекта и присвоить его какой-либо переменной:

```
var oCCU = new Windows.Media.Capture.CameraCaptureUI();
```

Далее мы выведем окно камеры на экран и предоставим пользователю возможность сделать фото. Делается это вызовом метода `captureFileAsync` окна камеры. В качестве единственного параметра он принимает один из элементов перечисления `Windows.Media.Capture.CameraCaptureUIMode`, в нашем случае — элемент `photo`.

Метод `captureFileAsync` возвращает в качестве результата обязательство, для которого мы с помощью метода `then` укажем функцию. Она выполнится после того, как пользователь сделает фото или откажется от этого.

В качестве единственного параметра она получит:

- файл, хранящий фото и представленный экземпляром давно знакомого нам объекта `Windows.Storage.StorageFile`, если фото было сделано;
- `null`, если пользователь отказался от съемки фото.

```
oCCU.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo).
↳then(function(file) {
  if (file) {
    //Фото сделано
    //Выполняем с хранящим его файлом какое-либо действие,
    //например, перемещаем его в другую папку
  } else {
    //Фото не было сделано
  }
});
```

Задание параметров получаемых фотографий

Диалоговое окно камеры, помимо всего прочего, предоставляет пользователю возможность задать различные параметры получаемых фотографий: размеры, соотношение сторон, формат и др. Однако мы можем сами установить для этих параметров какие-либо значения, чтобы избавить пользователя от мук выбора.

Объект `Windows.Media.Capture.CameraCaptureUI`, представляющий окно камеры, поддерживает свойство `photoSettings`. Оно хранит экземпляр объекта с именем прямо-таки рекордной длины `Windows.Media.Capture.CameraCaptureUIPhotoCaptureSettings`, который и представляет параметры фото.

Рассмотрим свойства этого объекта в порядке их значимости.

Свойство `format` позволяет указать формат получаемых фото в виде одного из элементов перечисления `Windows.Media.Capture.CameraCaptureUIPhotoFormat`. Всего этих элементов и, соответственно, поддерживаемых графических форматов три: `jpeg`, `png` и `jpegXR` (формат *JPEG XR* был разработан корпорацией Microsoft для хранения высококачественных фотографий).

Свойство `maxResolution` задает максимальное разрешение получаемого фото в виде одного из элементов перечисления `Windows.Media.Capture.CameraCaptureUIMaxPhotoResolution`. Все его элементы и, соответственно, поддерживаемые Metro разрешения фотографий таковы:

- `highestAvailable` — максимально возможное разрешение;
- `verySmallQvga` и `smallVga` — 320×240 пикселов;
- `mediumXga` — 1024×768 пикселов;
- `large3M` — 1920×1080 пикселов (3 мегапиксела);
- `veryLarge5M` — 5 мегапикселов.

Свойство `allowCropping` позволяет указать, будет ли пользователь иметь возможность обрезки фото. Значение `true` этого свойства дает пользователю такую возможность, а значение `false` — отменяет.

Остальные свойства, что мы рассмотрим, будут иметь смысл только в случае установки максимального допустимого разрешения фотографий и активизированной возможности обрезки.

Свойство `croppedSizeInPixels` устанавливает размер обрезанного фото. Если его указать, пользователь сможет обрезать фото только до заданного нами размера — не больше и не меньше.

Данное свойство хранит экземпляр объекта `Windows.Foundation.Size`, который и представляет эти размеры. Свойство `width` этого объекта служит для задания ширины, а свойство `height` — высоты; обе величины задаются в виде чисел в пикселах.

Если задать нулевые размеры обрезанного фото, пользователь сможет обрезать фото до любого размера.

Свойство `croppedAspectRatio` устанавливает соотношение сторон обрезанного фото. Пользователь изменить его не сможет.

Значение этого свойства также задается в виде экземпляра объекта `Windows.Foundation.Size`. Свойство `width` в этом случае указывает относительную ширину, а свойство `height` — относительную высоту; обе величины задаются в виде чисел.

Если задать для обоих относительных размеров нулевые значения, пользователь сможет обрезать фото до любого соотношения сторон.

Платформа Metro позволяет нам использовать вместо экземпляров объекта `Windows.Foundation.Size` экземпляры объекта `Object` — так проще. В них следует создать свойства `width` и `height`.

```
var oCCU = new Windows.Media.Capture.CameraCaptureUI();
oCCU.photoSettings.format =
Windows.Media.Capture.CameraCaptureUIPhotoFormat.jpeg;
oCCU.photoSettings.maxResolution =
Windows.Media.Capture.CameraCaptureUIMaxPhotoResolution.highestAvailable;
oCCU.photoSettings.allowCropping = true;
oCCU.photoSettings.croppedSizeInPixels = {width: 0, height: 0};
oCCU.photoSettings.croppedAspectRatio = {width: 16, height: 9};
oCCU.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo).
↳then(function(file) {
  if (file) {
    . . .
  }
});
```

Устанавливаем для получаемых фото формат JPEG, максимально доступное разрешение и возможность обрезки, а для обрезанных фото — неограниченные размеры и соотношение сторон 16:9. После чего выводим на экран диалоговое окно камеры.

Получение видео

Получить видео со встроенной камеры так же просто, как и фото. Более того, в обоих случаях используются практически одинаковые инструменты.

Подготовительные действия

Как и в случае с получением фото, чтобы успешно снять видео, приложение должно иметь право на доступ к камере. Как его дать, было рассказано ранее.

Скорее всего, пользователь захочет получить видео со звуковым сопровождением. Чтобы благополучно записать звук, приложение дополнительно должно иметь право на доступ к встроенному микрофону. Дать его можно установкой флажка **Microphone** в списке **Capabilities**, который находится на вкладке **Capabilities** окна параметров приложения.

Собственно получение видео

Как уже говорилось, для получения видео используются те же инструменты, что и для получения фото. Единственное исключение — методу `captureFileAsync` следует передать элемент `video` перечисления `Windows.Media.Capture.CameraCaptureUIMode`.

```
var oCCU = new Windows.Media.Capture.CameraCaptureUI();
oCCU.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.video).
↳then(function(file) {
  if (file) {
    //Видео записано
    //Выполняем с хранящим его файлом какое-либо действие
  } else {
    //Видео не было записано
  }
});
```

Задание параметров получаемых видеороликов

Напоследок рассмотрим, как установить параметры снимаемого видео.

Объект `Windows.Media.Capture.CameraCaptureUI` предоставляет для этой цели свойство `videoSettings`. Оно хранит экземпляр объекта `Windows.Media.Capture.CameraCaptureUIVideoCaptureSettings`, в свойствах которого и задаются нужные параметры.

Свойство `format` указывает формат получаемых видеороликов в виде одного из элементов перечисления `Windows.Media.Capture.CameraCaptureUIVideoFormat`. Поддерживаются всего два элемента и соответствующих им формата: `mp4` и `wmv`.

Свойство `maxResolution` задает максимальное разрешение получаемого видео в виде одного из элементов перечисления `Windows.Media.Capture.CameraCaptureUIMaxVideoResolution`. Поддерживаемые им элементы и, соответственно, значения максималь-

ного разрешения таковы: `highestAvailable` (максимально возможное разрешение), `lowDefinition` (низкое разрешение), `standardDefinition` (стандартное разрешение) и `highDefinition` (высокое разрешение).

Свойство `maxDurationInSeconds` позволяет задать максимальную продолжительность снимаемого видео в секундах в виде целого числа. Если указать ее равной 0, продолжительность видео ограничена не будет.

Свойство `allowCropping` указывает, будет ли пользователь иметь возможность обрезки "картинки" видео. Значение `true` этого свойства дает пользователю такую возможность, а значение `false` — отменяет.

```
var oCCU = new Windows.Media.Capture.CameraCaptureUI();
oCCU.videoSettings.format =
Windows.Media.Capture.CameraCaptureUIVideoFormat.mp4;
oCCU.videoSettings.maxResolution =
Windows.Media.Capture.CameraCaptureUIMaxVideoResolution.highDefinition;
oCCU.videoSettings.maxDurationInSeconds = 0;
oCCU.videoSettings.allowCropping = true;
oCCU.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.video).
↳then(function(file) {
  if (file) {
    . . .
  }
});
```

Задаем для снимаемого видео формат MP4, высокое разрешение, любую продолжительность и возможность обрезки, после чего выводим на экран окно камеры.

Что дальше?

В этой главе рассматривались инструменты платформы Metro для работы с подключенными флэш-дисками и встроенными фото- и видеокамерами. Мы научились реализовывать в своих приложениях поддержку технологии AutoPlay, делать фотографии и снимать видео.

Следующая глава будет посвящена взаимодействию с меню **Пуск** (Start) Windows 8. Мы научимся выводить в плитках этого меню произвольные данные и создавать вторичные плитки. Еще мы познакомимся со всплывающими уведомлениями и узнаем, для чего они могут пригодиться.

ЧАСТЬ VII

Прочие возможности Metro

ГЛАВА 21

Работа с плитками меню *Пуск* и всплывающими уведомлениями

В предыдущей главе мы занимались реализацией технологии AutoPlay и работой со встроенными камерами. Так что теперь мы умеем фотографировать и снимать фильмы!

Одно из самых значительных нововведений Windows 8 — обновленное меню **Пуск** (Start). Теперь оно занимает весь экран компьютера и представляет собой набор *плиток*, каждая из которых соответствует одному из установленных в системе приложений, как написанных для платформы Metro, так и традиционных.

На первый взгляд, плитки — это замена пунктам старого меню **Пуск**. Однако они далеко не так просты, как кажется на первый взгляд. Любое Metro-приложение может вывести на "свою" плитку какую-либо информацию текстового или графического характера. Еще любое Metro-приложение может программно создать произвольное количество так называемых вторичных плиток, активирующих какие-либо функции этого приложения или указывающих на какой-либо из открытых в нем документов.

Наконец, вместо знакомых нам по предыдущим версиям Windows окон-предупреждений теперь предусматриваются *всплывающие уведомления*. Они появляются в правом нижнем углу экрана, несут текстовое сообщение (обычно уведомляющее о чем-либо или описывающее возникшую проблему) и чувствительны к нажатию.

Настало время познакомиться со всем этим.

Вывод информации на плитки меню *Пуск*

В обычном состоянии плитка отображает лишь название и логотип приложения, указанные в его параметрах. (Как это сделать, мы узнаем в *главе 27*.) Но мы можем программно вывести в ней практически любую информацию, включающую фрагменты текста и изображения. Например, приложение для просмотра графики может выводить в плитке изображение, выбранное в данный момент.

Выбор шаблона для плитки

Для вывода информации в плитках меню **Пуск** мы используем один из предопределенных в платформе Metro шаблонов. Каждый шаблон задает, во-первых, вид информации, которая будет выводиться в плитке (только текст, только изображения или и то, и другое), а во-вторых, устанавливает количество единиц информации обоих видов, выводимых одновременно. Все поддерживаемые Metro шаблоны перечислены на Web-странице с интернет-адресом **http://msdn.microsoft.com/en-us/library/windows/apps/hh761491.aspx**.

К сожалению, вывести на плитках данные в произвольном формате мы не можем. Но это не страшно — Metro приготовила нам целых тридцать шаблонов, среди которых мы уж точно найдем подходящий.

Шаблоны для вывода данных в плитках описаны на языке *XML* (eXtensible Markup Language, расширяемый язык разметки). Он очень похож на язык HTML; для форматирования данных там также используются теги, а для указания их параметров — атрибуты тегов. Так, для форматирования фрагментов текста в XML-коде шаблонов применяется парный тег `<text>`, а для вставки графических изображений — одинарный тег `<image>`, атрибут `src` которого указывает ссылку на графический файл. Каждый шаблон имеет уникальное имя.

Для примера давайте рассмотрим XML-код шаблона `TileWideImageAndText01`. Этот шаблон предназначен для широких плиток и позволяет вывести графическое изображение и фрагмент текста под ним.

```
<tile>
  <visual lang="en-US">
    <binding template="TileWideImageAndText01">
      <image id="1" src="image1.png"/>
      <text id="1">Text field 1</text>
    </binding>
  </visual>
</tile>
```

Здесь мы видим тег `<image>`, выводящий изображение, и тег `<text>`, который выводит текст. (Теги `<tile>`, `<visual>` и `<binding>` являются служебными.)

Выбрав подходящий шаблон, мы его получим. Для этого следует использовать метод `getTemplateContent` объекта `Windows.UI.Notifications.TileUpdateManager`. (Данный объект предоставляет ряд служебных методов для работы с плитками и шаблонами; единственный его экземпляр создается самой платформой Metro.)

Метод `getTemplateContent` принимает в качестве единственного параметра один из элементов перечисления `Windows.UI.Notifications.TileTemplateType`. Эти элементы соответствуют поддерживаемым Metro шаблонам плиток; их список можно найти на Web-странице с интернет-адресом **http://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.notifications.tiletemplatetype.aspx**. Например, шаблону `TileWideImageAndText01` соответствует элемент `tileWideImageAndText01`.

А возвращает этот метод XML-код шаблона в виде экземпляра объекта `Windows.Data.Xml.Dom.XmlDocument`. С ним-то мы и будем иметь дело.

```
var template = Windows.UI.Notifications.TileTemplateType.
↳tileWideImageAndText01;
var oTileTemplate = Windows.UI.Notifications.TileUpdateManager.
↳getTemplateContent(template);
```

Заполнение шаблона данными

Теперь мы можем заполнить полученный шаблон данными.

Сначала нам потребуется получить доступ к различным элементам шаблона. Для этого мы можем использовать знакомый нам еще по *главе 5* метод `getElementsByTagName`. Его следует вызвать у экземпляра объекта `Windows.Data.Xml.Dom.XmlDocument`, представляющего шаблон.

Элемент шаблона, возвращаемый методом `getElementsByTagName`, представляется экземпляром объекта `Windows.Data.Xml.Dom.XmlElement`.

```
var oTexts = oTileTemplate.getElementsByTagName("text");
var oImages = oTileTemplate.getElementsByTagName("image");
var txt1 = oTexts[0];
var img1 = oImages[0];
```

Задать текстовое содержимое тега `<text>` можно, присвоив его свойству `innerText` полученного элемента. Понятно, что это содержимое должно быть задано в виде строки.

```
txt1.innerText = "Логотип Microsoft";
```

Задать графическое изображение можно, указав для атрибута `src` тега `<image>` ссылку на файл, хранящийся на локальном диске, или интернет-адрес файла, что находится в Интернете. Для указания ссылок на локальные файлы следует использовать формат `ms-appx:///<собственно ссылка на файл>`, причем *ссылка* в этом случае указывается без начального символа слеша. Например:

```
ms-appx:///images/microsoft_logo.png
```

К сожалению, вывести на плитку содержимое потока `Blob`, возвращенного методом `createObjectURL` объекта `URL` (подробнее — в *главе 16*), мы не можем.

Для указания значений атрибутов тега следует использовать метод `setAttribute` элемента шаблона.

```
<элемент шаблона>.setAttribute(<имя атрибута тега>,
<значение атрибута тега>)
```

Первым параметром передается имя атрибута тега, а вторым — его новое значение, оба — в виде строк.

Метод `setAttribute` не возвращает результата.

```
img1.setAttribute("src", "ms-appx:///images/microsoft_logo.png");
```

Вывод информации на плитку

Заполнив шаблон данными, мы можем вывести его на плитку. Выполняется это в три этапа.

На первом этапе мы создадим на основе этого шаблона *уведомление* — структуру данных, представляющую выводимую на плитку стороннюю информацию. Уведомление представляется объектом `Windows.UI.Notifications.TileNotification`. Нам следует создать экземпляр этого объекта, указав в качестве единственного параметра заполненный данными шаблон:

```
var oN = new Windows.UI.Notifications.TileNotification(oTileTemplate);
```

На втором этапе мы получим *диспетчер плиток* — структуру данных, которая будет управлять выводом информации на плитку. Диспетчер плиток представляется экземпляром объекта `Windows.UI.Notifications.TileUpdater`.

Получить его можно вызовом не принимающего параметров метода `createTileUpdaterForApplication` объекта `Windows.UI.Notifications.TileUpdateManager`. Результатом, возвращенным этим методом, будет нужный нам диспетчер плиток.

> ***НА ЗАМЕТКУ***
>
> Вообще-то, необязательно получать диспетчер плиток непосредственно перед выводом на экран каждого уведомления — это можно сделать и раньше. Автор рекомендует получать диспетчер плиток единожды, при запуске приложения — в коде его инициализации — и использовать его для вывода всех уведомлений, что предусмотрены приложением.

```
var oTU = Windows.UI.Notifications.TileUpdateManager.
↳createTileUpdaterForApplication();
```

На третьем этапе мы собственно выведем информацию на плитку. Это выполняется вызовом метода `update` диспетчера плиток. В качестве единственного параметра он принимает выводимое уведомление и не возвращает результата.

```
oTU.update(oN);
```

Вывод информации на плитки разных размеров

Платформа Metro устанавливает для плиток приложений два размера: стандартный (квадратные плитки) и расширенный (широкие плитки). Изначально плитка может быть как квадратной, так и широкой, но пользователь может изменить ее размер уже после установки приложения.

Отсюда следует, что нам перед собственно выводом информации на плитку как-то придется выяснять ее размер. К сожалению, платформа Metro не позволяет нам это сделать. Однако мы можем сформировать выводимую информацию таким образом, чтобы она подошла к плитке любого размера.

Сначала мы получим два шаблона, первый из которых будет предназначен для квадратных плиток, а второй — для широких:

```
var squareTemplate = Windows.UI.Notifications.TileTemplateType.
↳tileSquareBlock;
var oSquareTileTemplate = Windows.UI.Notifications.TileUpdateManager.
↳getTemplateContent(squareTemplate);
```

Получаем шаблон, предназначенный для квадратной плитки. Мы используем простой шаблон `TileSquareBlock`, позволяющий вывести два фрагмента текста.

```
var wideTemplate = Windows.UI.Notifications.TileTemplateType.
↳tileWideImageAndText01;
var oWideTileTemplate = Windows.UI.Notifications.TileUpdateManager.
↳getTemplateContent(wideTemplate);
```

Для вывода информации на широкую плитку мы используем знакомый шаблон `TileWideImageAndText01`.

Далее мы заполним оба этих шаблона данными. Как это делается, мы уже знаем.

Теперь нам потребуется извлечь из шаблона, предназначенного для квадратных плиток, тег `<binding>` вместе с его содержимым и добавить его в содержимое тега `<visual>` "широкого" шаблона. (См. XML-код шаблона, приведенный ранее, — там присутствуют оба этих тега.) Таким образом, мы создадим один универсальный шаблон, подходящий для плиток любого размера.

Перенести определенный элемент из одного фрагмента XML-кода в другой можно вызовом метода `importNode` объекта `Windows.Data.Xml.Dom.XmlDocument` (он, как мы помним, представляет весь XML-код шаблона).

<целевой фрагмент XML-кода>.importNode(*<извлекаемый элемент>*, *<извлекать вместе с потомками>*)

Этот метод вызывается у того фрагмента XML-кода, в который требуется перенести указанный элемент. Первым параметром ему передается сам переносимый элемент XML-кода. Вторым параметром указывается логическое значение `true`, если данный элемент следует перенести вместе с содержимым (то, что нам и нужно в этом случае), и `false`, если он нужен нам "пустым".

Метод `importNode` возвращает в качестве результата перенесенный элемент, уже принадлежащий целевому фрагменту XML-кода.

```
var oBindings = oSquareTileTemplate.getElementsByTagName("binding");
var bnd1 = oBindings[0];
var oBinding = oWideTileTemplate.importNode(bnd1, true);
```

После этого нам останется добавить перенесенный элемент в содержимое другого элемента. Для этого мы применим знакомый нам по *главе 10* метод `appendChild`.

```
var oVisuals = oWideTileTemplate.getElementByTagName("visual");
var vis1 = oVisuals[0];
vis1.appendChild(oBinding);
```

Добавляем перенесенный из "квадратного" шаблона тег `<binding>` в тег `<visual>`.

Напоследок мы создадим на основе содержимого "широкого" шаблона уведомление.

```
var oN = new
Windows.UI.Notifications.TileNotification(oWideTileTemplate);
```

И выведем его на экран. Как это сделать, рассказывалось ранее.

Задание параметров информации, выводимой на плитку

Теперь рассмотрим дополнительные параметры, которые мы можем указать для информации, выводимой на плитку в составе уведомления. Задавать их следует до того, как информация будет выведена на экран.

По умолчанию информация, выведенная приложением на плитку, будет присутствовать на ней постоянно. Но мы можем указать срок, до которого эта информация будет действительна; по истечении этого времени платформа Metro ее удалит.

Срок актуальности выводимой на плитку информации задается с помощью свойства `expirationTime` уведомления. Значением этого свойства должен быть экземпляр объекта `Date`, хранящий нужное значение даты и времени.

```
var curDate = new Date();
var expDate = new Date(curDate.getFullYear(), curDate.getMonth(),
curDate.getDate(), curDate.getHours() + 1, curDate.getMinutes(),
curDate.getSeconds());
var oN = new Windows.UI.Notifications.TileNotification(oTileTemplate);
oN.expirationTime = expDate;
```

Теперь наша информация будет выводиться на плитке в течение часа.

По умолчанию на плитку может выводиться содержимое только одного уведомления. Как только мы отправим на плитку другое уведомление, выводившееся ранее будет удалено. Однако мы можем активизировать так называемую *очередь уведомлений*, когда на плитку будут выводиться, последовательно сменяя друг друга, пять последних уведомлений. Это может пригодиться, если наше приложение активно выводит данные на экран.

Активизировать очередь уведомлений и впоследствии деактивировать ее можно вызовом метода `enableNotificationQueue` диспетчера плиток. В качестве единственного параметра он принимает `true`, если требуется активизировать очередь сообщений, или `false`, если ее нужно деактивировать, и не возвращает результата.

```
var oTU = Windows.UI.Notifications.TileUpdateManager.
↳createTileUpdaterForApplication();
oTU.enableNotificationQueue(true);
```

Не забываем, что одновременно в очереди могут находиться только пять уведомлений. Как только мы выведем на экран новое уведомление, самое старое из находящихся в очереди будет удалено.

Если очередь уведомлений активна, мы можем разбить уведомления на группы, указав для каждой из них перед выводом особую строку — *тег* (не путать с тегами языков HTML и XML!). Вновь выведенное уведомление, для которого был указан тег, будет заменять более старое уведомление с тем же тегом. Если наше приложение активно выводит на экран разноплановые данные, такой подход будет наилучшим.

Для указания тега служит свойство `tag` уведомления. Его значение должно представлять собой строку.

```
var oN = new Windows.UI.Notifications.TileNotification(oTileTemplate);
oN.tag = "pictures";
```

Задаем для созданного уведомления тег `pictures`.

Сброс плитки

Если нам потребуется удалить всю выводимую на плитку информацию, очистить очередь уведомлений (если она активна) и вернуть плитку к изначальному состоянию, мы вызовем метод `clear` диспетчера плиток. Этот метод не принимает параметров и не возвращает результата.

```
var oTU = Windows.UI.Notifications.TileUpdateManager.
↳createTileUpdaterForApplication();
oTU.clear();
```

Наклейки

Наклейка (badge) в терминологии платформы Metro — это небольшой значок, выводящийся в правом нижнем углу плитки и указывающий на состояние приложения. Обычно наклейки используются для вывода вспомогательных данных: количество новостей в канале RSS, состояние видеопроигрывателя (идет воспроизведение, приостановлен) и пр.

Metro предоставляет нам возможность вывести наклейки двух видов: число от 1 до 99 или один из предопределенных значков. Все доступные нам наклейки можно найти на Web-странице **http://msdn.microsoft.com/en-us/library/windows/apps/hh761458.aspx**.

Как и в случае с плитками, сначала следует получить шаблон нужной нам наклейки. Выполняется это вызовом метода `getTemplateContent` объекта `Windows.UI.Notifications.BadgeUpdateManager`. (Данный объект предоставляет методы для работы с наклейками и их шаблонами; единственный его экземпляр создается платформой Metro.)

Метод `getTemplateContent` принимает в качестве единственного параметра один из элементов перечисления `Windows.UI.Notifications.BadgeTemplateType`: `badgeGlyph` (наклейка в виде значка) или `badgeNumber` (наклейка в виде числа). Возвращает он XML-код шаблона в виде экземпляра объекта `Windows.Data.Xml.Dom.XmlDocument`.

```
var template = Windows.UI.Notifications.BadgeTemplateType.badgeGlyph;
var oBadgeTemplate = Windows.UI.Notifications.BadgeUpdateManager.
↳getTemplateContent(template);
```

Код шаблона в этом случае представляет собой одинарный тег `<badge>` с атрибутом `value`. Значение этого атрибута указывает вид значка или число, выводимое в качестве наклейки.

```
var bdg = oBadgeTemplate.getElementsByTagName("badge")[0];
bdg.setAttribute("value", "paused");
```

Указываем для наклейки значок "пауза".

Теперь создадим на основе заполненного шаблона уведомление. Оно будет представляться экземпляром объекта `Windows.UI.Notifications.BadgeNotification`. В качестве параметра мы укажем готовый шаблон наклейки.

```
var oN = new Windows.UI.Notifications.BadgeNotification(oBadgeTemplate);
```

Создадим *диспетчер наклеек*, который будет управлять их выводом. Он представляется экземпляром объекта `Windows.UI.Notifications.BadgeUpdater` и возвращается в качестве результата не принимающим параметров методом `createBadgeUpdaterForApplication` объекта `Windows.UI.Notifications.BadgeUpdateManager`.

```
var oBU = Windows.UI.Notifications.BadgeUpdateManager.
↳createBadgeUpdaterForApplication();
```

И выведем наклейку на экран, вызвав метод `update` диспетчера наклеек. В качестве единственного параметра он принимает уведомление и не возвращает результата.

```
oBU.update(oN);
```

Удалить выведенную на экран наклейку можно вызовом метода `clear` диспетчера наклеек. Этот метод не принимает параметров и не возвращает результата.

```
oBU.clear();
```

Пример: вывод на плитку имени файла, выбранного в приложении для просмотра графики, и общего количества файлов

Практическое занятие будет посвящено очередной доработке приложения для просмотра графики. Пусть оно выводит на свою плитку имя файла, выбранного в списке, и общее количество присутствующих там файлов. Причем имя файла будет выведено прямо на плитке, а количество файлов — с помощью наклейки, имеющей вид числа.

Откроем в Visual Studio решение `ImageViewer`. Откроем файл default.js проекта `ImageViewer`, входящего в это решение. Добавим следующий код, объявляющий необходимые переменные:

```
var squareTemplate = Windows.UI.Notifications.TileTemplateType.
↳tileSquareText04;
```

```
var wideTemplate = Windows.UI.Notifications.TileTemplateType.
↳tileWideText03;
var oTU = Windows.UI.Notifications.TileUpdateManager.
↳createTileUpdaterForApplication();
var badgeTemplate =
Windows.UI.Notifications.BadgeTemplateType.badgeNumber;
var oBU = Windows.UI.Notifications.BadgeUpdateManager.
↳createBadgeUpdaterForApplication();
```

Для вывода имени выбранного файла на квадратную плитку мы используем шаблон `TileSquareText04`, а для вывода на широкую плитку — шаблон `TileWideText03`. Оба этих шаблона позволяют вывести один достаточно длинный фрагмент текста, который при необходимости будет разбит на несколько строк.

Найдем объявление функции `fillList()`, заполняющей список файлов, и добавим в ее конец следующие выражения (выделены полужирным шрифтом):

```
function fillList() {
  var libImages = Windows.Storage.KnownFolders.picturesLibrary;
  libImages.getFilesAsync().then(function (files) {
    . . .
    if ((files.size > 0) && (files.size < 100)) {
      var oBadgeTemplate = Windows.UI.Notifications.BadgeUpdateManager.
      ↳getTemplateContent(badgeTemplate);
      var bdg = oBadgeTemplate.getElementsByTagName("badge")[0];
      bdg.setAttribute("value", files.size.toString());
      var oN = new
      Windows.UI.Notifications.BadgeNotification(oBadgeTemplate);
      oBU.update(oN);
    }
  });
}
```

Поскольку в качестве наклейки можно вывести только числа от 1 до 99, мы перед ее созданием проверяем, не выходит ли значение количества файлов за эти пределы.

После этого отыщем обработчик события `iteminvoked` списка и добавим в его конец следующие выражения (выделены полужирным шрифтом):

```
function ctrListItemInvoked(evt) {
  evt.detail.itemPromise.then(function (selected) {
    . . .
    var oSquareTileTemplate = Windows.UI.Notifications.TileUpdateManager.
    ↳getTemplateContent(squareTemplate);
    var oWideTileTemplate = Windows.UI.Notifications.TileUpdateManager.
    ↳getTemplateContent(wideTemplate);
    var txt1 = oSquareTileTemplate.getElementsByTagName("text")[0];
    var txt2 = oWideTileTemplate.getElementsByTagName("text")[0];
```

```
    txt1.innerText = oSelected.name;
    txt2.innerText = oSelected.name;
    var bnd1 = oSquareTileTemplate.getElementsByTagName("binding")[0];
    var oBinding = oWideTileTemplate.importNode(bnd1, true);
    var vis1 = oWideTileTemplate.getElementsByTagName("visual")[0];
    vis1.appendChild(oBinding);
    var oN = new
    Windows.UI.Notifications.TileNotification(oWideTileTemplate);
    oTU.update(oN);
  });
}
```

Здесь нам все уже знакомо, так что дополнительных комментариев не требуется.

Сохраним исправленный файл и запустим приложение. Сразу же переключимся на меню **Пуск** и проверим, выводится ли на плитке наклейка с количеством файлов. Вернемся к приложению, выберем в списке какое-либо изображение и снова переключимся на меню **Пуск**. В плитке должно появиться имя файла, в котором хранится выбранное изображение.

Вторичные плитки

Когда мы устанавливаем новое Metro-приложение, в меню **Пуск** для него создается плитка. Это так называемая *основная плитка*; она собственно представляет это приложение и служит для его запуска.

Но приложение может создать в меню **Пуск** произвольное количество *вторичных плиток*. Такие плитки служат либо для активации какой-либо из функций приложения, либо для доступа к какому-либо из открытых в приложении документов, писем, новостей, сообщений и пр.

Создание вторичных плиток

Вторичная плитка представляется экземпляром объекта `Windows.UI.StartScreen.SecondaryTile`. При его создании нам следует указать следующие параметры в том порядке, в котором они перечислены.

- Уникальный идентификатор создаваемой вторичной плитки. Он указывается в виде строки и позволяет впоследствии получить доступ к этой плитке, чтобы изменить ее параметры или удалить.
- Короткую надпись, состоящую не более чем из 13 символов. Задается в виде строки и выводится прямо на плитке.
- Длинную надпись. Задается в виде строки и выводится во всплывающей подсказке.
- Параметр вторичной плитки. Задается в виде строки, передается приложению при нажатии на плитку и используется для выяснения, нажатием какой плитки было активизировано приложение.

- Дополнительные параметры плитки. Для их указания используются элементы перечисления `Windows.UI.StartScreen.TileOptions`:
  - `none` — дополнительные параметры отсутствуют;
  - `showNameOnLogo` — вывести на квадратной плитке короткую надпись плитки вместо ее логотипа;
  - `showNameOnWideLogo` — вывести на широкой плитке короткую надпись плитки вместо ее логотипа;
  - `copyOnDeployment` — выполнять синхронизацию набора вторичных плиток между всеми компьютерами, принадлежащими данному пользователю.

  Мы можем складывать элементы этого перечисления, как обычные числа, чтобы объединить даваемый ими эффект.
- Ссылку на графическое изображение, которое будет выводиться на квадратной плитке. Указывается в виде экземпляра объекта `Windows.Foundation.Uri`. Если ссылка не задана, вторичная плитка будет содержать логотип приложения, указанный в его параметрах.
- Ссылку на графическое изображение, которое будет выводиться на широкой плитке. Указывается в виде экземпляра объекта `Windows.Foundation.Uri`.

Последний параметр является необязательным и указывается только в том случае, если требуется создать широкую плитку. Если же его опустить, будет создана квадратная плитка.

```
var tileID = "SecondaryTile.showLastImage";
var tileShortLabel = "Последнее";
var tileLongLabel = "Вывод последнего изображения из имеющихся в списке";
var tileArgument = "showlastimage";
var tileOptions = Windows.UI.StartScreen.TileOptions.showNameOnLogo +
Windows.UI.StartScreen.TileOptions.showNameOnWideLogo;
var tileSquareLogo = new
Windows.Foundation.Uri("ms-appx:///images/logos/last_image_square.png");
var tileWideLogo = new
Windows.Foundation.Uri("ms-appx:///images/logos/last_image_wide.png");
var oST = new Windows.UI.StartScreen.SecondaryTile(tileID,
tileShortLabel, tileLongLabel, tileArgument, tileOptions, tileSquareLogo,
tileWideLogo);
```

Создаем широкую вторичную плитку, которая при нажатии выведет в приложении для просмотра графики самое последнее из имеющихся в списке изображений.

Чтобы вывести созданную вторичную плитку на экран, мы вызовем у нее не принимающий параметров метод `requestCreateAsync`. Он отображает сообщение, предлагающее пользователю поместить данную плитку в меню **Пуск**, причем пользователь может как согласиться, так и отказаться.

Метод `requestCreateAsync` вернет в качестве результата обязательство, для которого мы с помощью метода `then` укажем функцию. Она выполнится, когда пользователь

сделает свой выбор, и получит в качестве единственного параметра логическое значение `true`, если пользователь согласился поместить вторичную плитку в меню **Пуск**, и `false`, если он отказался от этого.

```
oST.requestCreateAsync().then(function(isCreated) {
  if (isCreated) {
    //Вторичная плитка была создана
  } else {
    //Вторичная плитка не была создана
  }
});
```

Обработка нажатий на вторичные плитки

Вторичные плитки используются для активизации либо различных функций приложения, либо перехода к открытым в них документам. Это мы уже знаем. Но до сих пор не выяснили, как это сделать.

Сначала нам нужно выяснить, было ли приложение активизировано нажатием на вторичную плитку и, если это так, на какую именно плитку. Сделать это можно в обработчике события `activated` объекта `WinJS.Application`. (Подробнее об этом событии и его обработчике говорилось в *главе 19*.)

Свойство `kind` экземпляра объекта `Object`, что находится в свойстве `detail` события, будет хранить в этом случае элемент `launch` перечисления `Windows.ApplicationModel.Activation.ActivationKind`. А значением свойства `arguments` того же экземпляра объекта станет параметр вторичной плитки либо, если приложение было запущено щелчком на основной плитке, пустую строку.

```
app.onactivated = function(eventObject) {
  if (eventObject.detail.kind ===
  Windows.ApplicationModel.Activation.ActivationKind.launch) {
    //Приложение было активизировано нажатием на вторичную плитку
    if (eventObject.detail.arguments == "showlastimage" {
      //Приложение было активизировано нажатием на вторичную плитку
      //"Последнее"
    }
  }
};
```

Вывод информации и наклеек на вторичные плитки

Платформа Metro также позволяет нам выводить на вторичные плитки произвольную информацию и размещать на них наклейки. Выполняется это так же, как и в случае основной плитки, за следующими исключениями.

Диспетчер плиток для вторичной плитки создается вызовом метода `createTileUpdaterForSecondaryTile` объекта `Windows.UI.Notifications.TileUpdateManager`. Он принимает в качестве единственного параметра идентификатор нужной вторичной плитки.

```
var oTU = Windows.UI.Notifications.TileUpdateManager.
↳createTileUpdaterForSecondaryTile("SecondaryTile.showLastImage");
```

Диспетчер наклеек для вторичной плитки создается вызовом метода `createBadgeUpdaterForSecondaryTile` объекта `Windows.UI.Notifications.BadgeUpdateManager`. В качестве единственного параметра он также принимает идентификатор вторичной плитки.

```
var oBU = Windows.UI.Notifications.BadgeUpdateManager.
↳createBadgeUpdaterForSecondaryTile("SecondaryTile.showLastImage");
```

Оба диспетчера будут действовать только на ту вторичную плитку, идентификатор которой мы указали в качестве параметра соответствующего метода.

Работа с вторичными плитками

Создав вторичные плитки, мы можем получить к ним доступ и изменить их параметры. Для этого используются свойства и методы объекта `Windows.UI.StartScreen.SecondaryTile`, описанные далее.

Метод `exists` позволяет узнать, присутствует ли в меню **Пуск** вторичная плитка с указанным идентификатором. Он вызывается у самого объекта, принимает в качестве единственного параметра строку с идентификатором искомой плитки и возвращает `true`, если данная плитка присутствует в меню, и `false`, если ее там нет.

```
if (Windows.UI.StartScreen.SecondaryTile.
↳exists("SecondaryTile.showLastImage")) {
  //Плитка SecondaryTile.showLastImage присутствует в меню
} else {
  //Плитка SecondaryTile.showLastImage отсутствует
}
```

Метод `findAllAsync` запускает процесс получения всех вторичных плиток, созданных приложением. Он также вызывается у самого объекта, не принимает параметров и возвращает в качестве результата обязательство. Для него мы в вызове метода `then` укажем функцию, которая выполнится, когда все плитки будут получены. В качестве параметра эта функция примет массив, содержащий все найденные плитки и аналогичный тому массиву, что получается в результате вызова метода `pickMultipleFilesAsync` *(см. главу 16)*.

```
Windows.UI.StartScreen.SecondaryTile.findAllAsync().then(function(tiles) {
  if (tiles.size > 0) {
    //Что-то делаем с полученными вторичными плитками
  }
});
```

Мы можем получить параметры вторичной плитки и, при желании, изменить их. Для этого мы воспользуемся перечисленными далее свойствами:

- `tileId` — хранит идентификатор плитки в виде строки;
- `shortName` — хранит короткую надпись в виде строки;

- `displayName` — хранит длинную надпись в виде строки;
- `arguments` — хранит параметр плитки в виде строки;
- `tileOptions` — хранит дополнительные параметры плитки в виде элемента перечисления `Windows.UI.StartScreen.TileOptions` или их суммы;
- `logo` — хранит ссылку на файл с изображением, предназначенным для квадратной плитки, в виде экземпляра объекта `Windows.Foundation.Uri`;
- `wideLogo` — хранит ссылку на файл с изображением, предназначенным для широкой плитки, в виде экземпляра объекта `Windows.Foundation.Uri`;
- `smallLogo` — хранит ссылку на файл с изображением, которое будет выводиться в левом нижнем углу плитки, если на ней отображается сторонняя информация. Указывается в виде экземпляра объекта `Windows.Foundation.Uri`. Если не задано, будет использован логотип приложения.

```
var tileID = tiles[0].tileId;
tiles[0].shortName = "В конец";
```

Выясняем идентификатор первой плитки из полученных при вызове метода `findAllAsync` и задаем для нее другую короткую надпись.

Если мы изменили параметры плитки, то должны ее обновить. Делается это вызовом не принимающего параметров метода `updateAsync`. В качестве результата он возвращает обязательство, для которого мы с помощью метода `then` укажем функцию. Она выполнится после обновления плитки и получит в качестве параметра значение `true`, если плитка была успешно обновлена, и `false`, если сделать это почему-то не удалось.

```
tiles[0].updateAsync().then(function(updated) {
  if (updated) {
    //Плитка была обновлена
  } else {
    //Плитка не была обновлена
  }
});
```

Удаление вторичных плиток

Чтобы удалить ненужную более вторичную плитку, мы вызовем у нее не принимающий параметров метод `requestDeleteAsync`. Он выводит на экран сообщение, предлагающее пользователю удалить плитку; пользователь может как подтвердить удаление плитки, так и отказаться от этого.

Результатом, возвращенным методом `requestDeleteAsync`, будет обязательство. Функция, которую мы укажем для него в вызове метода `then`, выполнится после того, как пользователь сделает выбор. В качестве единственного параметра она получит `true`, если пользователь удалил плитку, или `false` в противном случае.

```
tiles[0].requestDeleteAsync().then(function(deleted) {
  if (deleted) {
    //Плитка была удалена
  } else {
    //Плитка не была удалена
  }
});
```

Всплывающие уведомления

Всплывающие уведомления в платформе Metro выполняют ту же функцию, что окна-предупреждения в традиционных Windows-приложениях. Такие уведомления появляются в правом верхнем углу экрана, содержат некое сообщение, состоящее из текста и графики, и чувствительны к нажатиям. Если пользователь так и не нажмет на уведомление, оно по истечении определенного времени исчезнет.

Обычно всплывающие уведомления используются, чтобы сообщить пользователю о каком-либо значимом событии: приходе нового письма или сообщения, срабатывании будильника или возникновении ошибки. Нажав на всплывающее уведомление, пользователь может сразу открыть письмо, перейти к сообщению или просто активизировать отправившее его приложение.

Активизация функции вывода всплывающих уведомлений

Если мы собираемся использовать в своем приложении всплывающие уведомления, нам следует явно указать это в параметрах приложения.

Откроем окно параметров приложения, как делали это в *главе 16*, и переключимся на его вкладку **Application UI** (рис. 21.1). Найдем в группе элементов управления **Notifications** раскрывающийся список **Toast Capable** и выберем в нем пункт **Yes**. Закроем окно параметров приложения, сохранив все сделанные изменения.

Выбор шаблона для всплывающего сообщения и заполнение его данными

Для формирования всплывающих уведомлений также используются шаблоны. Полный их список приведен на Web-странице с интернет-адресом **http://msdn.microsoft.com/en-us/library/windows/apps/hh761494.aspx**.

Получение нужного шаблона выполняется вызовом метода `getTemplateContent` объекта `Windows.UI.Notifications.ToastNotificationManager`. (Данный объект предоставляет методы для работы с всплывающими уведомлениями и их шаблонами; его единственный экземпляр создается платформой Metro.)

Метод `getTemplateContent` принимает в качестве единственного параметра элемент перечисления `Windows.UI.Notifications.ToastTemplateType`, соответствующий нуж-

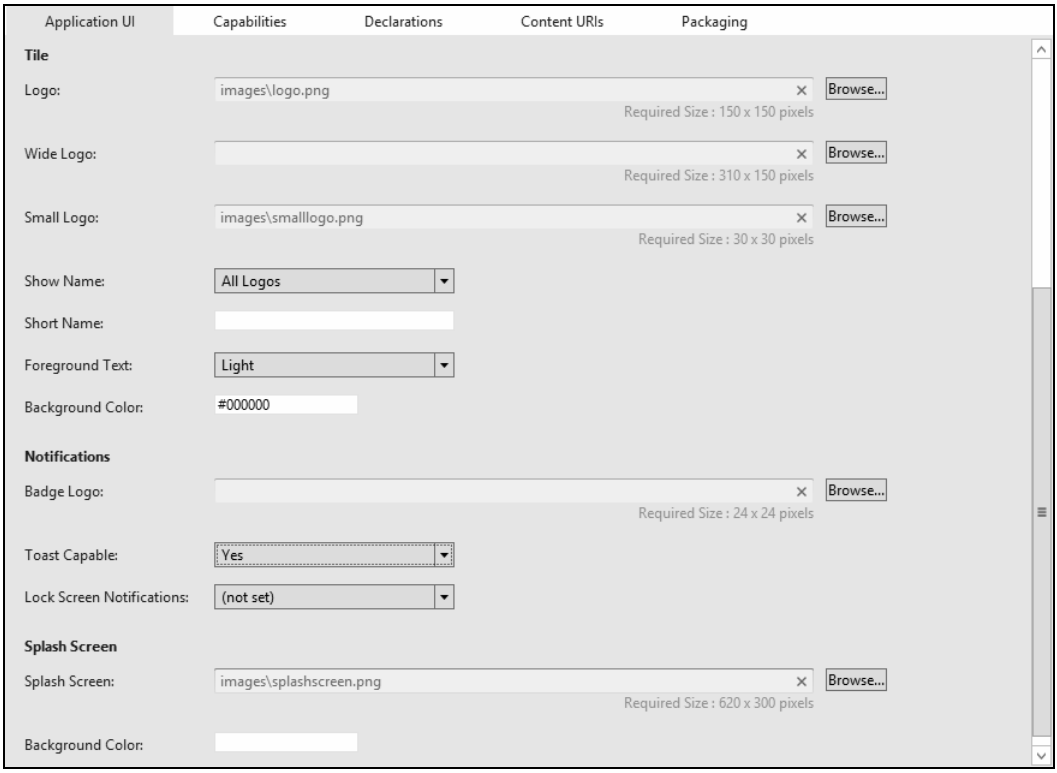


Рис. 21.1. Содержимое вкладки **Application UI** окна, где задаются параметры приложения

ному шаблону. Полный список всех элементов этого перечисления можно найти на Web-странице с интернет-адресом **http://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.notifications.toasttemplatetype.aspx**.

Результатом, возвращаемым данным методом, станет XML-код шаблона в виде экземпляра объекта `Windows.Data.Xml.Dom.XmlDocument`.

```
var template = Windows.UI.Notifications.ToastTemplateType.toastText01;
var oToastTemplate = Windows.UI.Notifications.ToastNotificationManager.
↳getTemplateContent(template);
```

Шаблон `toastText01` позволяет вывести большой фрагмент текста, который в случае необходимости будет разбит на строки.

Теперь мы можем заполнить полученный шаблон данными.

```
var txt1 = oToastTemplate.getElementsByTagName("text")[0];
txt1.innerText = "Ошибка!";
```

Задание воспроизводимого звука

Следующий наш шаг — задание звука, который будет воспроизведен при выводе всплывающего уведомления на экран. Это позволит нам привлечь к нему внимание пользователя.

Воспроизводимый звук указывается в теге <audio>. К сожалению, изначально этот тег в XML-коде шаблона отсутствует, так что нам придется его создать. Для этого мы воспользуемся знакомым нам по *главе 10* методом createElement.

```
var aud1 = oToastTemplate.createElement("audio");
```

Теперь зададим для атрибута src этого тега одно из предопределенных значений, собственно указывающее нужный нам звук. Все доступные нам значения перечислены далее:

- Notification.Default — звук, воспроизводимый по умолчанию;
- Notification.IM — получение нового мгновенного сообщения;
- Notification.Mail — получение нового письма;
- Notification.Reminder — наступление очередного события, указанного в приложении планировщика;
- Notification.SMS — получение нового сообщения SMS;

> ***НА ЗАМЕТКУ***
>
> Все эти звуки не должны зацикливаться при воспроизведении.

- Notification.Looping.Alarm и Notification.Looping.Alarm2 — звук будильника;
- Notification.Looping.Call и Notification.Looping.Call2 — телефонный звонок.

> ***НА ЗАМЕТКУ***
>
> Два последних звука должны зацикливаться.

```
aud1.setAttribute("src", "Notification.Mail");
aud1.setAttribute("loop", "false");
```

Указываем для нашего всплывающего уведомления звук, используемый для индикации получения письма, и отключаем его зацикливание.

Осталось только добавить полностью сформированный тег <audio> в состав содержимого тега <toast>, который, собственно, и определяет всплывающее уведомление. Для этого мы применим метод appendChild.

```
var tst1 = oToastTemplate.getElementsByTagName("toast")[0];
tst1.appendChild(aud1);
```

Вывод всплывающего уведомления

Всплывающее уведомление представляется экземпляром объекта Windows.UI.Notifications.ToastNotification. Мы создадим его, указав в качестве параметра XML-код шаблона, заполненного данными.

```
var oT = new Windows.UI.Notifications.ToastNotification(oToastTemplate);
```

Далее получим *диспетчера всплывающих уведомлений*, который будет управлять их выводом. Он представляется экземпляром объекта Windows.UI.Notifications.

ToastNotifier и возвращается в качестве результата не принимающим параметров методом createToastNotifier объекта Windows.UI.Notifications.ToastNotificationManager.

```
var oTN = Windows.UI.Notifications.ToastNotificationManager.
↳createToastNotifier();
```

Вывод всплывающего уведомления на экран выполняется вызовом метода show полученного диспетчера. В качестве единственного параметра он принимает всплывающее уведомление и не возвращает результата.

```
oTN.show(oT);
```

Реализация отслеживания нажатий на всплывающее уведомление

Как уже говорилось, пользователь может нажать на всплывающее уведомление, чтобы активизировать связанную с ним функцию приложения. Например, нажав на уведомление, выведенное приложением видеопроигрывателя в случае ошибки при открытии файла, пользователь сможет открыть другой файл.

Реализовать отслеживание нажатий на всплывающие уведомления проще всего, привязав к событию activated этого уведомления обработчик. Данное событие возникает как раз при нажатии на уведомление.

Привязать обработчик к событию activated следует до того, как уведомление будет выведено на экран.

```
oT.addEventListener("activated", function() {
  //Пользователь нажал на уведомление
});
```

Дополнительные возможности всплывающих уведомлений

Метод hide диспетчера всплывающих уведомлений позволяет удалить с экрана уведомление, указанное в качестве единственного параметра. Результата этот метод не возвращает.

```
oTN.hide(oT);
```

Убираем выведенное ранее всплывающее уведомление.

Событие dismissed возникает во всплывающем уведомлении, если пользователь так и не нажал на нем, и оно исчезло с экрана. Экземпляр объекта, хранящий сведения о событии и передаваемый в качестве параметра его функции-обработчику, будет содержать свойство reason. Данное свойство будет хранить один из элементов перечисления Windows.UI.Notifications.ToastDismissalReason, приведенных далее:

- userCanceled — пользователь сам убрал уведомление с экрана;
- applicationHidden — уведомление было скрыто приложением с помощью вызова метода hide;

- `timedOut` — уведомление было скрыто по истечении отведенного ему времени.

```
oT.addEventListener("dismissed", function(evt) {
  if (evt.reason ==
  Windows.UI.Notifications.ToastDismissalReason.userCanceled) {
    //Всплывающее уведомление было скрыто самим пользователем
  }
});
```

Пример: вывод сообщения об ошибке открытия видеофайла в приложении видеопроигрывателя

Для примера реализуем в приложении видеопроигрывателя вывод всплывающего уведомления. Пусть оно отображает сообщение об ошибке, возникшей при открытии видеофайла, и при нажатии позволяет открыть другой файл.

Откроем в Visual Studio проект `VideoPlayer` и сразу же укажем для него возможность вывода всплывающих уведомлений. Далее переключимся на файл default.js и объявим необходимые переменные.

```
var toastTemplate = Windows.UI.Notifications.ToastTemplateType.
↳toastText01;
var oTN = Windows.UI.Notifications.ToastNotificationManager.
↳createToastNotifier();
```

Отыщем код инициализации и добавим в его конец следующее выражение (выделено полужирным шрифтом):

```
document.addEventListener("DOMContentLoaded", function () {
  WinJS.UI.processAll().then(function() {
    . . .
    vidMain.addEventListener("error", vidMainError);
  });
});
```

Это выражение привяжет обработчик к событию `error` видеопроигрывателя (подробнее об этом событии говорилось в *главе 12*).

Объявим данный обработчик:

```
function vidMainError() {
  var oToastTemplate = Windows.UI.Notifications.ToastNotificationManager.
  ↳getTemplateContent(toastTemplate);
  var txt1 = oToastTemplate.getElementsByTagName("text")[0];
  txt1.innerText = "При открытии видеофайла возникла ошибка. Нажмите
  ↳здесь, чтобы открыть другой файл.";
  var aud1 = oToastTemplate.createElement("audio");
  aud1.setAttribute("src", "Notification.Default");
  aud1.setAttribute("loop", "false");
```

```
  var tst1 = oToastTemplate.getElementsByTagName("toast")[0];
  tst1.appendChild(aud1);
  var oT = new
  Windows.UI.Notifications.ToastNotification(oToastTemplate);
  oT.addEventListener("activated", btnOpenClick);
  oTN.show(oT);
}
```

При нажатии на всплывающее уведомление будет выполнена та же функция, что выполняется после нажатия на кнопку **Открыть**.

Сохраним все исправления и запустим приложение. Попытаемся открыть какой-либо файл, не являющийся видеофайлом (для этого мы можем взять любой файл и дать ему расширение, скажем, avi), и проверим, появилось ли на экране всплывающее уведомление. Если оно появилось, нажмем на него и посмотрим, что из этого выйдет.

Что дальше?

В этой главе мы учились выводить на плитки меню **Пуск** стороннюю информацию и наклейки, создавать вторичные плитки и использовать всплывающие уведомления.

Следующая глава будет посвящена управлению жизненным циклом Metro-приложений. Мы узнаем, как сохранить важную информацию при приостановке приложения платформой Metro и восстановить ее, когда приложение снова будет запущено. С учетом того, что все неактивные Metro-приложения в Windows 8 приостанавливаются, а при недостатке системных ресурсов — вообще завершаются, для нас эта тема будет очень важна.

ГЛАВА 22

Управление жизненным циклом Metro-приложения

В предыдущей главе мы изучали средства платформы Metro по выводу произвольной информации на плитки меню **Пуск** (Start), созданию вторичных плиток и использованию всплывающих уведомлений. Эти средства позволят нам держать пользователя в курсе того, что происходит с приложением.

Эта глава будет посвящена управлению жизненным циклом Metro-приложения. Мы научимся сохранять важные рабочие данные перед приостановкой приложения и восстанавливать их после его активизации. И, как всегда, платформа Metro максимально облегчит нам эту задачу.

Жизненный цикл Metro-приложения

Принципы, согласно которым платформа Metro, входящая в состав Windows 8, управляет написанными под нее приложениями, заметно отличаются от тех, которыми оперировали предыдущие версии Windows. Частично они рассматривались еще в *главе 1*, и сейчас настало время поговорить о них более обстоятельно. Итак...

- Когда пользователь нажимает на плитку меню **Пуск**, платформа Metro проверяет, было ли соответствующее этой плитке приложение уже запущено. Если это так, приложение выводится на экран и возобновляет свою работу *(активизируется)*; другая копия этого приложения запущена не будет. В противном случае приложение запускается заново и также активизируется.
- Как только пользователь переключается на другое приложение, то, что было до этого момента активным, пропадает с экрана и приостанавливает свою работу *(деактивируется)*. При этом все ее рабочие данные (значения переменных, экземпляры объектов, внутренние структуры данных, используемые самой платформой Metro, и пр.) остаются в оперативной памяти. Таким образом, при повторной активизации этого приложения оно может продолжить свою работу с того места, где оно было приостановлено.

  Поскольку Windows 8 предназначена, в первую очередь, для портативных и мобильных компьютеров (ноутбуков, планшетов и др.), такой подход оправдан.

Работающие в фоновом режиме приложения отнимают системные ресурсы компьютера и расходуют энергию аккумулятора. Следовательно, их приостановка позволит сохранить приемлемый уровень производительности и автономности работы компьютера.

- Деактивированные приложения могут оставаться в таком состоянии неопределенно долго. (Как мы уже знаем из *главы 1*, Metro-приложения не предоставляют пользователю возможность их завершения; по крайней мере, Microsoft не рекомендует ее реализовывать.)
- Если платформа Metro посчитает, что ей существенно не хватает системных ресурсов (в первую очередь — оперативной памяти) для запуска очередного приложения или продолжения работы уже запущенных, она полностью завершит работу какого-либо из деактивированных приложений. При этом все рабочие данные завершенных приложений будут выгружены из памяти.
- Платформа Metro никак не уведомляет приложения об их завершении. (Разработчики не реализовали в ней такого средства, и непонятно, почему.)

Рассмотренный нами набор принципов определяет, как Metro-приложения запускаются, активизируются, деактивируются и завершаются. Или, говоря на жаргоне профессиональных программистов, *жизненный цикл* Metro-приложения.

Проблема потери и устаревания рабочих данных

Прочитав написанное ранее, мы как Metro-разработчики сразу увидим две проблемы, с которыми обязательно столкнемся.

1. Предположим, пользователь запустил какое-либо приложение, некоторое время с ним поработал и переключился на другое, потом — на третье, четвертое и т. д. В какой-то момент платформа Metro столкнулась с нехваткой системных ресурсов и, чтобы их освободить, завершила приложение, что пользователь запустил самым первым. Разумеется, при этом все данные, что были открыты в этом приложении, окажутся утерянными.

   И вот пользователь вдруг вспоминает, что не закончил работу в самом первом приложении. Он активизирует его нажатием на плитку меню **Пуск** и с удивлением обнаруживает, что все данные, с которыми он работал, исчезли. (А это и немудрено — приложение-то было запущено заново.) Понятно, что такой сюрприз пользователю не понравится, и это еще мягко говоря...

2. Если приложение загружает из Интернета какие-либо данные, обновляющиеся достаточно часто, то при его активизации после продолжительного бездействия данные эти окажутся устаревшими. (Это актуально, например, для приложения для чтения каналов новостей — ведь эти каналы обновляются постоянно.) Так что пользователю после активизации этого приложения придется выполнять обновление данных вручную, что потребует от него лишних действий и, таким образом, снизит привлекательность приложения.

Есть ли решение этих проблем? Есть!

- Сразу после деактивации приложения следует сохранить все его рабочие данные. Причем делать это придется при каждой деактивации приложения — кто знает, может быть, платформе Metro вскоре вздумается его завершить.
- Сразу после активизации и возобновления работы приложения следует обновить загруженные из Интернета данные.

Реализуется все это очень просто, в чем мы сейчас убедимся.

Сохранение и восстановление рабочих данных

Для сохранения и последующего восстановления рабочих данных приложения платформа Metro предоставляет нам исключительно удобные стандартные инструменты.

Сохранение рабочих данных

Прежде всего, это событие `checkpoint`. Оно возникает в экземпляре объекта `WinJS.Application`, представляющем наше приложение, перед тем как оно будет деактивировано и, соответственно, приостановлено. Обработчик этого события — идеальное место для сохранения рабочих данных.

Visual Studio при создании любого проекта сам формирует "пустой" обработчик события `checkpoint`.

```
app.oncheckpoint = function (eventObject) {
  . . .
};
```

Нам лишь останется вписать в него код, который и сохранит рабочие данные.

Для этого мы используем свойство `sessionState` того же объекта `WinJS.Application`. Оно хранит "пустой", не содержащий ни единого свойства, экземпляр объекта `Object`. Мы сами сформируем в нем нужные нам свойства и сохраним в них рабочие данные приложения.

Microsoft рекомендует сохранять рабочие данные в специально созданном экземпляре объекта `Object`, который, в свою очередь, сохраняется в каком-либо свойстве экземпляра объекта, что хранится в свойстве `sessionState`.

```
app.oncheckpoint = function (eventObject) {
  var oData = {
    prop1: value1,
    prop2: value2
  };
  app.sessionState.appWorkData = oData;
};
```

Здесь мы так и поступили: создали экземпляр объекта `Object` с двумя свойствами — `prop1` и `prop2`, — сохранили в них рабочие данные и присвоили полученный экземпляр объекта вновь созданному свойству `appWorkData` экземпляра объекта `Object`, что хранится в свойстве `sessionState`.

Восстановление рабочих данных

Восстановление сохраненных ранее рабочих данных при повторном запуске приложения выполняется в обработчике события `activated` экземпляра объекта `WinJS.Application`, представляющего приложение. С этим обработчиком события мы неоднократно имели дело в предыдущих главах.

Сначала мы проверим свойство `kind` экземпляра объекта `Object`, что находится в свойстве `detail` экземпляра объекта, представляющего событие. Это свойство должно хранить элемент `launch` перечисления `Windows.ApplicationModel.Activation.ActivationKind`.

Далее мы проверим, существует ли свойство, что мы создали в экземпляре объекта `Object`, который хранится `sessionState` экземпляра объекта `WinJS.Application`. Если это так, значит, приложение было ранее завершено самой платформой Metro, и в этом свойстве хранятся его рабочие данные, которые мы получим и каким-либо образом используем в приложении.

```
app.onactivated = function (eventObject) {
  if (eventObject.detail.kind ===
  Windows.ApplicationModel.Activation.ActivationKind.launch) {
    if (app.sessionState.appWorkData) {
      var oData = app.sessionState.appWorkData;
      var value1 = oData.prop1;
      var value2 = oData.prop2;
      //Используем сохраненные ранее рабочие данные
    }
  }
};
```

Обновление загруженных данных

Обновить данные, загруженные из Интернета, при повторной активизации приложения также очень просто.

Как только приложение вновь начинает свою работу после активизации, в объекте `Windows.UI.WebUI.WebUIApplication` возникает событие `resuming`. (Этот объект предоставляет несколько событий, возникающих в приложении в разные моменты времени. Единственный его экземпляр создается платформой Metro.) Лучшего места для обновления данных, чем обработчик этого события, мы не найдем.

```
Windows.UI.WebUI.WebUIApplication.onresuming = function() {
  //Обновляем данные
}
```

Активизация и деактивация приложения в среде Visual Studio

В процессе испытания разрабатываемых Metro-приложений нам придется как-то выполнять их активизацию и деактивацию. Для этого мы можем использовать исключительно удобные средства, предоставляемые Visual Studio, о которых сейчас и пойдет разговор.

Как только мы запустим любое Metro-приложение в данной среде разработки, панель инструментов его главного окна изменит свой вид. В числе прочего, там появятся три следующие кнопки:

- **Suspend the application** () — выполняет деактивацию и приостановку приложения;
- **Resume the application** () — выполняет активизацию и возобновление работы приложения;

> ***НА ЗАМЕТКУ***
> Эти две кнопки пригодятся для проверки перезагрузки данных.

- **Suspend and shut down the application** () — выполняет деактивацию и приостановку приложения, после чего завершает его работу. Понадобится при проверке сохранения рабочих данных.

Пример 1: сохранение и восстановление рабочих данных в приложении видеопроигрывателя

В качестве практики давайте доработаем приложение видеопроигрывателя таким образом, чтобы оно сохраняло и восстанавливало путь к открытому в нем видеофайлу и позицию, на которой было прервано его воспроизведение.

Откроем в Visual Studio проект `VideoPlayer`. И сразу же дадим ему права на доступ к библиотеке Видео. Поскольку при восстановлении открытого ранее файла мы будем получать этот файл программно, эти права нам пригодятся.

После этого переключимся на файл default.js и введем код, объявляющий необходимые переменные.

```
var sFilePath = "", iPosition = 0;
```

В переменной `sFilePath` будет храниться путь к открытому файлу; пустая строка будет означать, что никакой файл в приложении пока не открыт. А переменную `iPosition` мы используем для хранения текущей позиции воспроизведения.

Найдем код, объявляющий функцию `btnOpenClick()`, что выполняется при нажатии кнопки **Открыть**, и дополним его таким выражением (выделено полужирным шрифтом):

```
function btnOpenClick(evt) {
  . . .
  oFP.pickSingleFileAsync().then(function (oFile) {
    if (oFile) {
      sFilePath = oFile.path;
      vidMain.src = URL.createObjectURL(oFile);
    }
  });
}
```

Здесь мы сохранили в объявленной ранее переменной путь к открываемому файлу.

В объявление функции `btnCloseClick()`, которая выполнится после нажатия кнопки **Закрыть**, мы добавим такое выражение (выделено полужирным шрифтом):

```
function btnCloseClick() {
  sFilePath = "";
  . . .
}
```

Далее найдем обработчик события `checkpoint` и впишем в него такой код:

```
app.oncheckpoint = function (eventObject) {
  if (sFilePath != "") {
    var oData = {
      filePath: sFilePath,
      position: vidMain.currentTime
    };
    app.sessionState.appWorkData = oData;
  }
};
```

Сначала мы проверяем, был ли в приложении открыт какой-либо файл, и, следовательно, существуют ли рабочие данные, которые следует сохранять. Если это так, мы сохраняем путь к открытому файлу и позицию его воспроизведения.

Отыщем обработчик события `activated` и дополним его следующим кодом (выделен полужирным шрифтом):

```
app.onactivated = function (eventObject) {
  . . .
  if (eventObject.detail.kind ===
  Windows.ApplicationModel.Activation.ActivationKind.launch) {
    if (app.sessionState.appWorkData) {
      var oData = app.sessionState.appWorkData;
      sFilePath = oData.filePath;
      Windows.Storage.StorageFile.getFileFromPathAsync(sFilePath).
      ↳then(function(file) {
        if (file) {
          vidMain.src = URL.createObjectURL(file);
```

```
            iPosition = oData.position;
          }
        });
      }
    }
};
```

Здесь мы извлекаем из сохраненных данных путь к файлу, находим и открываем этот файл и получаем позицию воспроизведения.

Напоследок найдем объявление функции `vidMainCanPlay()` — обработчика события `canplay` видеопроигрывателя. Отыщем присутствующее в его коде выражение

```
sldProgress.value = 0;
```

и заменим его следующим кодом:

```
vidMain.currentTime = iPosition;
sldProgress.value = iPosition;
iPosition = 0;
```

Этот код установит проигрыватель на ту позицию воспроизведения, что была сохранена при завершении приложения.

Сохраним все сделанные правки и запустим приложение. Откроем какой-либо видеофайл, запустим его на воспроизведение, немного подождем, переключимся на Visual Studio и выполним приостановку и завершение приложения, нажав кнопку **Suspend and shut down the application**. Когда после этого мы запустим его снова, оно должно опять открыть этот файл и установить его на позицию, на которой воспроизведение было прекращено. Если так и произошло, мы все сделали правильно.

Пример 2: перезагрузка содержимого канала новостей после активизации приложения

Теперь доработаем приложение для чтения каналов новостей таким образом, чтобы оно перезагружало содержимое канала после повторной активизации.

Откроем проект `FeedReader` и переключимся на файл default.js. Создадим там следующий код:

```
Windows.UI.WebUI.WebUIApplication.onresuming = function(evt) {
  if (oFeedURL) {
    fillList();
  }
}
```

Сначала мы проверяем, подписался ли пользователь на какой-либо канал новостей, и после этого выполняем заполнение списка новостей.

Сохраним исправленный файл и запустим приложение. Подпишемся на канал новостей, переключимся на Visual Studio, приостановим приложение, нажав кнопку **Suspend the application**, и сразу же возобновим его работу кнопкой **Resume the**

application. Когда мы после этого снова вернемся к приложению `FeedReader`, оно должно загрузить канал новостей повторно.

Что дальше?

В этой главе мы занимались управлением жизненным циклом Metro-приложений. Мы научились сохранять рабочие данные, чтобы избежать их потери, и перезагружать данные, полученные из Интернета.

Следующая глава будет посвящена еще одной важной теме — созданию настраиваемых приложений. Ведь одно из главных отличий профессионально написанных приложений от ученических поделок — возможность их настройки.

ГЛАВА 23

Создание настраиваемых Metro-приложений

В предыдущей главе мы рассматривали жизненный цикл Metro-приложений — выясняли, как они запускаются, активируются, деактивируются и завершаются. Также мы научились предотвращать потерю рабочих данных приложения и выполнять их обновление, когда приложение вновь начинает работать.

Эта глава будет посвящена разработке настраиваемых Metro-приложений. Мы познакомимся со стандартными инструментами платформы Metro, предназначенными для хранения настроек в локальном и переносимом хранилищах. И, как обычно, в очередной раз переделаем одно из написанных ранее приложений, дав ему возможность настройки.

Как пользователь будет настраивать Metro-приложение

Начнем с того, что рассмотрим принципы, согласно которым мы будем создавать интерфейсные средства для настройки Metro-приложения.

В традиционных Windows-приложениях для представления параметров настройки используются самые разные элементы интерфейса: специализированные диалоговые окна (чаще всего), отдельные элементы управления, находящиеся в главном окне (обычно в простых приложениях), кнопки на панели инструментов, пункты меню и др. Каких-либо правил, регламентирующих реализацию инструментов настройки, здесь не существует, так что фантазия разработчиков не знает границ.

В Metro-приложениях выбор поскромнее, поскольку набор доступных элементов интерфейса здесь заметно меньше. Мы можем использовать кнопки панели инструментов, пункты меню и элементы управления, являющиеся частью основного интерфейса. Ну и, разумеется, всплывающие элементы; пожалуй, ими пользоваться удобнее всего.

НА ЗАМЕТКУ

Платформа Metro предусматривает инструменты для создания панелей настроек, выводящихся в составе соответствующего стандартного модуля (charm). Однако, на взгляд автора, на данный момент они не очень удобны, поэтому он не рекомендует их использовать.

Все настройки должны действовать на приложение сразу же после их изменения. Пользователь не должен нажимать для применения настроек никаких кнопок **OK** или **Применить**. Так, если в приложении видеопроигрывателя он включит автоматический запуск воспроизведения фильма после его открытия, приложение сразу должно принять это к исполнению и при открытии очередного фильма сразу же начать его воспроизведение.

Хранилища настроек

После того как пользователь изменит настройки приложения, они должны быть сохранены. Вряд ли пользователь обрадуется, когда при следующем запуске приложения обнаружит, что все кропотливо заданные им настройки вернулись к состоянию по умолчанию.

Для хранения настроек приложения платформа Metro предлагает два специальных хранилища, которые так и называются — *хранилища настроек*. Они аналогичны рассмотренным нами в *главе 16* хранилищам приложения, предназначенным для записи файлов.

- *Локальное*, или *непереносимое*. Служит для хранения настроек, которые не требуется синхронизировать между всеми устройствами, принадлежащими данному пользователю. Объем локального хранилища не ограничен.
- *Переносимое*. Применяется для хранения настроек, которые требуется синхронизировать между устройствами. Объем переносимого хранилища ограничен.

Получить доступ к обоим этим хранилищам можно с помощью знакомого нам по *главе 16* объекта `Windows.Storage.ApplicationData`. Как и ранее, мы обратимся к его свойству `current`, которое хранит экземпляр данного объекта, представляющий все хранилища настроек.

```
var oAppData = Windows.Storage.ApplicationData.current;
```

Этот экземпляр объекта поддерживает следующие свойства, позволяющие получить доступ к конкретному хранилищу:

- `localSettings` — локальное хранилище;
- `roamingSettings` — переносимое хранилище.

Каждое из этих свойств хранит так называемый *контейнер настроек* — структуру, объединяющую все записанные нами данные. Контейнер представляет собой экземпляр объекта `Windows.Storage.ApplicationDataContainer`.

```
var oRSC = oAppData.roamingSettings;
```

Сохранение настроек

Сначала мы рассмотрим сохранение настроек. Для этого платформа Metro предоставляет исключительно гибкие и простые в использовании механизмы.

Сохранение простых значений

Проще всего сохранить в контейнере настроек какие-либо простые значения: строки, числа, логические величины, экземпляры объектов и др.

Контейнер поддерживает весьма примечательное свойство `values`. Оно хранит "пустой" (не имеющий элементов) хэш, в котором мы и сохраним все нужные значения. Этот хэш представляет собой экземпляр особого объекта, но ведет себя так же, как обычные хэши, знакомые нам по *главе 4*.

В качестве индекса сохраняемого элемента указывается строковое значение. Его длина не должна превышать 255 символов. Сохраняемое значение может быть любого типа; его объем не должен быть более 8 Кбайт.

```
oRSC.values["autoLoad"] = true;
oRSC.values["feedURL"] = "http://www.thevista.ru/rss.php";
oRSC.values["lastSaveDate"] = new Date();
```

Сохраняем в контейнере, соответствующем переносимому хранилищу, логическое значение, строку и экземпляр объекта `Date`.

Еще хэш, хранящийся в свойстве `values` контейнера, поддерживает несколько полезных для нас методов и одно свойство. Давайте поговорим о них.

Метод `remove` позволяет удалить не нужный более элемент. В качестве единственного параметра он принимает строку с индексом удаляемого элемента и не возвращает результата.

```
oRSC.values.remove("lastSaveDate");
```

Метод `clear` удаляет все элементы из хэша. Он не принимает параметров и не возвращает результата.

```
oRSC.values.clear();
```

Метод `hasKey` позволяет узнать, присутствует ли в хэше элемент с указанным индексом. В качестве единственного параметра он принимает строку с искомым индексом и возвращает `true`, если таковой элемент присутствует в контейнере, и `false`, если его там нет.

```
if (oRSC.values.hasKey("feedURL")) {
  //Элемент с таким индексом присутствует
}
```

А свойство `size` возвращает количество элементов в хэше.

```
var iSettingCount = oRSC.values.size;
```

Создание составных значений

Иногда требуется сохранить сразу несколько значений в составе единой структуры данных. В таком случае мы создадим *составное значение* и поместим все сохраняемые значения настроек в него.

Составное значение представляется объектом `Windows.Storage.ApplicationDataCompositeValue`. Нам следует создать экземпляр этого объекта.

```
var oCV = new Windows.Storage.ApplicationDataCompositeValue();
```

Составное значение ведет себя как хэш. Следовательно, для помещения в него значений мы можем использовать знакомые приемы.

```
oCV["autoPlay"] = true;
oCV["rewindAtEnd"] = false;
```

Сохраняем в составном значении две логические величины.

Важно помнить, что совокупный объем всех значений, хранящихся в составном значении, не должен превышать 64 Кбайт.

Готовое составное значение мы можем сохранить в контейнере, как проделывали это с простыми значениями.

```
oRSC.values["playerSettings"] = oCV;
```

Составное значение поддерживает уже знакомые нам методы `remove`, `clear` и `hasKey` и свойство `size`.

Создание вложенных контейнеров настроек

Если наше приложение поддерживает большое количество настроек, затрагивающих разные аспекты и составляющих сложную иерархию, составных значений для их организации нам может оказаться мало. В этом случае будет удобнее создать несколько вложенных контейнеров и поместить в каждый из них настройки, относящиеся к одной группе.

Вложенный контейнер настроек может хранить произвольное количество как простых, так и составных значений. Он хранится внутри контейнера, представляющего само хранилище настроек (назовем его *основным*).

Создание вложенного контейнера выполняется вызовом метода `createContainer` основного контейнера.

```
<основной контейнер>.createContainer(<имя вложенного контейнера>,
<следует ли создавать вложенный контейнер>)
```

Первым параметром передается уникальное имя создаваемого вложенного контейнера в виде строки. Вторым параметром передается один из элементов перечисления `Windows.Storage.ApplicationDataCreateDisposition`, указывающий, следует ли создавать данный контейнер, если он не существует:

- `always` — если контейнер с указанным именем не существует, он будет создан; в противном случае будет возвращен существующий контейнер;

- `existing` — возвращается уже существующий контейнер с указанным именем; если же такого контейнера нет, он не будет создан.

Метод `createContainer` возвращает вложенный контейнер — либо созданный, либо уже существующий. Он представляется экземпляром объекта `Windows.Storage.ApplicationDataContainer` — того же, что и основной контейнер.

```
var oC = oRSC.createContainer("imageSearcherSettings",
Windows.Storage.ApplicationDataCreateDisposition.always);
```

Получив вложенный контейнер, мы заполним его сохраняемыми данными.

```
oC.values["keyword"] = "Windows 8";
oC.values["loadSearchResult"] = true;
```

Сохраняем во вложенном контейнере строковое и логическое значения.

Мы можем сохранить во вложенном контейнере составные значения и даже создать в нем другие вложенные контейнеры. Выполняется это так же, как и в случае с основным контейнером.

Любой контейнер поддерживает свойство `containers`. Оно хранит экземпляр особого объекта-коллекции, перечисляющей все вложенные контейнеры, что хранятся в данном контейнере. Этот объект предоставляет нам два метода и свойство, которые могут нам пригодиться.

Метод `hasKey` нам уже знаком. Он позволяет узнать, существует ли вложенный контейнер с указанным именем.

```
if (oRSC.containers.hasKey("imageSearcherSettings")) {
  //Вложенный контейнер imageSearcherSettings существует
}
```

Метод `lookup` позволяет получить вложенный контейнер с указанным именем. Имя нужного контейнера передается единственным параметром.

```
if (oRSC.containers.hasKey("imageSearcherSettings")) {
  var oC = oRSC.containers.lookup("imageSearcherSettings");
  //Сохраняем настройки во вложенном контейнере imageSearcherSettings
}
```

Свойство `size` возвращает количество вложенных контейнеров.

Если нам потребуется удалить ненужный вложенный контейнер, мы вызовем метод `deleteContainer`; вызывать его следует у того контейнера, в который вложен удаляемый контейнер. Данный метод принимает в качестве единственного параметра строку с именем удаляемого контейнера и не возвращает результата.

```
oRSC.deleteContainer("imageSearcherSettings");
```

Считывание настроек

Прочитать сохраненные настройки очень просто.

```
var oAppData = Windows.Storage.ApplicationData.current;
var oRSC = oAppData.roamingSettings;
```

```
var bAutoLoad = oRSC.values["autoLoad"];
var sFeedURL = oRSC.values["feedURL"];
```

Считываем значения двух элементов, хранящихся непосредственно в основном контейнере.

Если мы попытаемся прочитать значение по несуществующему индексу, мы получим `null`.

```
var dLastSaveDate = oRSC.values["lastSaveDate"];
```

Элемент с индексом `lastSaveDate` не существует (ранее мы его удалили), поэтому переменная `dLastSaveDate` получит значение `null`.

Для проверки существования элемента в контейнере также можно использовать метод `hasKey`, описанный ранее.

```
var oCV = oRSC.values["playerSettings"];
if (oCV) {
  var bAutoPlay = oCV["autoPlay"];
  var bRewindAtEnd = oCV["rewindAtEnd"];
}
```

Извлекаем из контейнера составное значение и, если оно существует, получаем хранящиеся в нем данные.

```
if (oRSC.containers.hasKey("imageSearcherSettings")) {
  var oC = oRSC.containers.lookup("imageSearcherSettings");
  var sKeyword = oC.values["keyword"];
  var bLoadSearchResult = oC.values["loadSearchResult"];
}
```

Получаем настройки, хранящиеся во вложенном контейнере.

Отслеживание изменения настроек, сохраненных в переносимом хранилище

Собственно, об этом уже говорилось в *главе 16*. Но этот момент так важен, что лучше повториться.

Как только мы записываем какое-либо значение в переносимое хранилище настроек, в экземпляре объекта `Windows.Storage.ApplicationData` возникает событие `datachanged`. Это событие следует использовать для того, чтобы перенести новые значения настроек данного приложения на все остальные его копии, установленные на других компьютерах, что принадлежат этому же пользователю.

```
var oAppData = Windows.Storage.ApplicationData.current;
oAppData.addEventListener("datachanged", function() {
  var oRSC = oAppData.roamingSettings;
  var bAutoLoad = oRSC.values["autoLoad"];
  var sFeedURL = oRSC.values["feedURL"];
  //Применяем полученные настройки
});
```

Благодаря этому мы можем быть уверены, что все копии приложения, установленные на всех компьютерах, используют одни и те же настройки.

Пример: реализация настроек в приложении видеопроигрывателя

Практическое занятие будет посвящено реализации настроек в приложении видеопроигрывателя. Давайте сделаем так, чтобы оно (разумеется, по желанию пользователя) запускало фильм на воспроизведение сразу после его открытия и по окончании воспроизведения перематывало его в начало.

Откроем в Visual Studio проект `VideoPlayer`. Откроем файл default.html и вставим в содержимое тега `<body>` перед кодом, определяющим нижнюю панель инструментов, вот такой код:

```
<div id="divSettings" data-win-control="WinJS.UI.Flyout">
  <div>
    <input type="checkbox" id="chkAutoPlay" />
    <label for="chkAutoPlay">Запускать воспроизведение сразу после
    открытия файла</label>
  </div>
  <div>
    <input type="checkbox" id="chkRewindOnEnd" />
    <label for="chkRewindOnEnd">Перематывать в начало по окончании
    воспроизведения</label>
  </div>
</div>
```

Здесь мы создали всплывающий элемент, в котором и будут задаваться настройки приложения. Назначение входящих в состав этого элемента двух флажков понятно из их надписей.

Теперь создадим в нижней панели инструментов кнопку, которая будет выводить данный всплывающий элемент на экран. Для этого вставим в создающий эту панель инструментов код вот такой фрагмент (выделен полужирным шрифтом):

```
<button data-win-control="WinJS.UI.AppBarCommand"
data-win-options="{id: 'btnOpen', label: 'Открыть', icon: 'openfile',
↳section: 'global'}"></button>
<hr data-win-control="WinJS.UI.AppBarCommand"
data-win-options="{section: 'global', type: 'separator'}" />
<button data-win-control="WinJS.UI.AppBarCommand"
data-win-options="{id: 'btnSettings', label: 'Настройки',
↳icon: 'settings', section: 'global', type: 'flyout',
↳flyout: 'divSettings'}"></button>
<hr data-win-control="WinJS.UI.AppBarCommand"
data-win-options="{section: 'global', type: 'separator'}" />
```

Переключимся на файл default.js и наберем код, объявляющий необходимые переменные:

```
var bAutoPlay = false, bRewindOnEnd = false, ctrSettings, chkAutoPlay,
chkRewindOnEnd;
```

Переменные `bAutoPlay` и `bRewindOnEnd` будут хранить текущие значения настроек — признаки, будет ли выполняться автоматический запуск фильма на воспроизведение и его перемотка в начало, соответственно. Начальные значения этих настроек мы зададим равными `false` (т. е. упомянутые ранее действия изначально выполняться не будут).

Там же поместим код, получающий основной контейнер, что соответствует переносимому хранилищу настроек:

```
var oRSC = Windows.Storage.ApplicationData.current.roamingSettings;
```

Добавим в код инициализации следующие выражения (выделены полужирным шрифтом):

```
document.addEventListener("DOMContentLoaded", function() {
  WinJS.UI.processAll().then(function() {
    . . .
    ctrSettings = document.getElementById("divSettings").winControl;
    chkAutoPlay = document.getElementById("chkAutoPlay");
    chkRewindOnEnd = document.getElementById("chkRewindOnEnd");
    ctrSettings.addEventListener("beforeshow", ctrSettingsBeforeShow);
    chkAutoPlay.addEventListener("click", chkAutoPlayClick);
    chkRewindOnEnd.addEventListener("click", chkRewindOnEndClick);
    vidMain.addEventListener("ended", vidMainEnded);
    var oAppData = Windows.Storage.ApplicationData.current;
    oAppData.addEventListener("datachanged", appDataDataChanged);
    appDataDataChanged();
  });
});
```

Помимо всего прочего, мы сделали здесь две вещи. Во-первых, привязали обработчик к событию `ended` видеопроигрывателя, чтобы получить возможность отследить момент, когда воспроизведение фильма закончится, чтобы перемотать его в начало. Во-вторых, привязали обработчик к событию `datachanged` переносимого хранилища настроек и вызвали этот обработчик, чтобы загрузить настройки сразу при запуске приложения.

Сразу же объявим функцию — обработчик события `datachanged` переносимого хранилища настроек:

```
function appDataDataChanged() {
  if (oRSC.values.hasKey("autoPlay")) {
    bAutoPlay = oRSC.values["autoPlay"];
  }
```

```
  if (oRSC.values.hasKey("rewindOnEnd")) {
    bRewindOnEnd = oRSC.values["rewindOnEnd"];
  }
}
```

Здесь мы проверяем, присутствуют ли в контейнере переносимого хранилища сохраненные ранее настройки, и, если они там есть, считываем их.

Найдем объявление функции `btnOpenClick()` (она выполняется после нажатия кнопки **Открыть**) и вставим в него выражение, выделенное полужирным шрифтом:

```
function btnOpenClick(evt) {
  . . .
  oFP.pickSingleFileAsync().then(function(oFile) {
    if (oFile) {
      . . .
      if (bAutoPlay) {
        vidMain.play();
      }
    }
  });
}
```

Оно запустит фильм на воспроизведение, если пользователь указал это в настройках.

Напишем обработчик события `ended` видеопроигрывателя:

```
function vidMainEnded() {
  if (bRewindOnEnd) {
    vidMain.currentTime = 0;
    sldProgress.value = 0;
    showTiming();
    vidMain.pause();
  }
}
```

Здесь мы, если пользователь изъявил такое желание, перематываем фильм в начало и ставим воспроизведение на паузу.

Напишем обработчик события `beforeshow` всплывающего элемента настроек:

```
function ctrSettingsBeforeShow() {
  chkAutoPlay.checked = bAutoPlay;
  chkRewindOnEnd.checked = bRewindOnEnd;
}
```

Здесь мы устанавливаем флажки, присутствующие в данном всплывающем элементе, согласно текущим значениям настроек.

Осталось только написать обработчики событий `click` обоих флажков, что служат для указания настроек:

```
function chkAutoPlayClick() {
  bAutoPlay = chkAutoPlay.checked;
  oRSC.values["autoPlay"] = bAutoPlay;
}
function chkRewindOnEndClick() {
  bRewindOnEnd = chkRewindOnEnd.checked;
  oRSC.values["rewindOnEnd"] = bRewindOnEnd;
}
```

Мы одновременно и переносим состояние соответствующего флажка в отведенную для хранения настройки переменную, и сохраняем его в переносимом хранилище.

Сохраним все исправленные файлы, запустим приложение и проверим его в действии.

Что дальше?

В этой главе мы учились сохранять и впоследствии считывать значения настроек Metro-приложения. Делалось это совсем несложно, как и многое в платформе Metro.

Следующая глава открывает последнюю часть этой книги, посвященную коммерциализации Metro-приложений. Мы будем учиться создавать локализованные приложения, поддерживающие различные языки и автоматически подстраивающиеся под язык, что указан в настройках операционной системы. А, научившись это делать, можем выходить на международный уровень!

ЧАСТЬ VIII

Коммерциализация и распространение Metro-приложений

ГЛАВА 24

Локализация Metro-приложений

В предыдущей главе мы занимались созданием настраиваемых Metro-приложений. Мы узнали, каким образом сохраняются значения настроек и как их можно впоследствии считать. И тем самым сделали еще один шаг в профессиональное программирование.

Эта, последняя, часть книги будет посвящена коммерциализации и распространению Metro-приложений. И, если с распространением все ясно, понятие коммерциализации требует разъяснений.

Под *коммерциализацией* понимается подготовка приложения к распространению, в том числе и на коммерческой основе (т. е. за деньги). Она включает:

- *локализацию* приложений — перевод его интерфейса на другие языки (если, конечно, предполагается распространять данное приложение на международном рынке);
- обеспечение поддержки устройств с различными параметрами экрана, к которым относятся его разрешение, плотность расположения пикселов, соотношение размеров сторон и пр. Сюда же входит обеспечение работы в ландшафтной и портретной ориентации устройства и в прикрепленном режиме;
- обеспечение поддержки либо покупки самого приложения после окончания срока бесплатного пользования, либо приобретения отдельных частей его функциональности (актуально для платных приложений).

И начнем мы с локализации Metro-приложений.

Как создаются Metro-приложения для международного рынка

Сначала, как водится, — теоретическая часть. Мы узнаем, какие принципы применяются для создания "многоязычных" приложений и какие из них применим мы.

Еще несколько лет назад разработчикам приходилось изготавливать несколько редакций одного и того же приложения, каждая из которых поддерживала один язык.

Понятно, что такой подход очень трудоемок и часто вызывает путаницу — пользователь может по ошибке загрузить редакцию приложения, поддерживающую не тот язык.

Поэтому сейчас применяется другой подход — создание приложения, которое изначально поддерживает сразу несколько языков. Нужный язык либо выбирается при запуске приложения автоматически, либо указывается в его настройках вручную.

Предыдущие версии Windows не предоставляли никаких специальных инструментов для создания локализованных приложений подобного рода, так что разработчикам приходилось для каждого приложения изобретать свой подход. Существуют, правда, разработки третьих фирм (в том числе и бесплатные), которые существенно облегчают и упрощают процесс локализации. Однако они либо подходят только для одного средства разработки, либо дорогостоящи.

Все изменилось с появлением Windows 8. Входящая в ее состав платформа Metro предоставляет стандартные средства для создания локализованных приложений, мощные и исключительно простые в использовании. Ими-то мы и займемся в этой главе.

В общем, локализованные Metro-приложения создаются и работают по следующим правилам:

- каждое приложение включает в свой состав все средства, необходимые для поддержки всех заявленных языков: фрагменты текста, изображения и пр.;
- нужный язык выбирается при запуске приложения автоматически на основе языка, указанного в настройках операционной системы. Выбор языка выполняет сама платформа Metro;
- всю работу по выводу фрагментов текста и изображений, соответствующих выбранному языку, также выполняет платформа Metro.

Как правило, локализацию приложения осуществляют уже после завершения его разработки. Все-таки делать сразу два дела — локализовывать приложение, "допиливая" его по ходу дела, — крайне неудобно.

Процесс локализации Metro-приложения

Теперь можно приступить к рассмотрению процесса локализации Metro-приложений. Выполняется он в том порядке, в котором здесь изложен.

Сначала мы решим, какие языки будет поддерживать наше приложение. Как минимум, оно должно поддерживать русский и английский языки — "родной" язык приложения и язык межнационального общения.

Далее мы внимательно просмотрим все файлы, содержащие описание интерфейса приложения и его логики. Отыщем все фрагменты текста, которые так или иначе выводятся на экран и, соответственно, требуют локализации. К ним относятся как фрагменты, присутствующие в файлах описания интерфейса, так и строковые

значения, что используются в логике приложения и выводятся на экран программно.

Список всех поддерживаемых языков и локализуемых фрагментов текста — та самая "печка", от которой мы будем "плясать".

Создание папок для хранения языковых ресурсов

Первое практическое действие, что мы предпримем, — создание в папке проекта вложенной папки с именем strings. В ней будут храниться файлы с фрагментами текста, переведенные на языки, которые поддерживает приложение. Назовем эти фрагменты текста *языковыми ресурсами* приложения.

> ***НА ЗАМЕТКУ***
>
> *Ресурсами* приложения называются любые данные, включенные в его состав и не относящиеся к описанию его интерфейса, оформления и логики. Так, графические изображения, используемые приложением, являются ресурсами.

В только что созданной папке мы создадим набор других папок, каждая из которых будет хранить языковые ресурсы, относящиеся к одному языку. Эти папки должны иметь имена, совпадающие с наименованием соответствующего языка.

Все поддерживаемые Metro языки с их наименованиями можно найти на Web-странице **http://msdn.microsoft.com/en-us/library/ms533052(v=vs.85).aspx**. Так, русский язык имеет обозначение ru, а американский английский — en-us.

Пусть наше приложение должно поддерживать русский и американский английский языки. Тогда мы создадим такую структуру папок:

```
<папка проекта>
  strings
    en-us
    ru
```

Создание файлов с языковыми ресурсами

Теперь нам следует создать файлы, которые и будут хранить языковые ресурсы, относящиеся к одному языку. Они так и называются — *файлы языковых ресурсов*.

Создать файл языковых ресурсов можно прямо в среде Visual Studio. Отыщем в иерархическом списке панели **SOLUTION EXPLORER** пункт, представляющий проект нашего приложения, и выберем его. Щелкнем на этом пункте правой кнопкой мыши и выберем в подменю **Add** появившегося на экране контекстного меню пункт **New Item**. На экране отобразится диалоговое окно **Add New Item** (см. рис. 13.1). Выберем в среднем списке этого окна пункт **Resources File (.resjson)**, укажем в поле ввода **Name** имя создаваемого файла — resources.resjson — и нажмем кнопку **Add**.

> ***ВНИМАНИЕ!***
>
> Файл языковых ресурсов обязательно должен иметь имя resources.resjson, независимо от языка, на котором записаны хранящиеся в нем фрагменты текста.

Созданный файл языковых ресурсов, как обычно, будет находиться прямо в папке проекта. Нам следует переместить его в папку, что соответствует языку, ресурсы которого он будет хранить. Например, если мы собираемся сохранить в этом файле ресурсы для русского языка, то переместим его в папку strings/ru.

И сразу же наполним файл содержимым — языковыми ресурсами. Они описываются в формате JSON *(см. главу 18)* — в виде JavaScript-кода, формирующего экземпляр объекта `Object`. Отдельные фрагменты текста будут представлять свойства этого экземпляра объекта.

- Имена свойств должны представлять собой строки, содержащие уникальные идентификаторы фрагментов текста. Впоследствии платформа Metro, определив язык операционной системы и открыв соответствующий ему файл языковых ресурсов, сможет по этому идентификатору загрузить нужный фрагмент текста.
- Значениями свойств станут сами фрагменты текста, переведенные на соответствующий язык.

В качестве примера давайте рассмотрим содержимое двух файлов языковых ресурсов, которые войдут в состав приложения для чтения каналов новостей `FeedReader`. Вот что хранит файл с русскими языковыми ресурсами:

```
{
  "txtURLLabel": "Интернет-адрес",
  "btnURLValue": "Подписаться",
  "btnRefreshLabel": "Обновить",
  "btnSubscribeLabel": "Подписаться"
}
```

Здесь мы создали четыре фрагмента текста, которые будут выводиться на экран в качестве надписей для поля ввода, обычной кнопки и двух кнопок панели инструментов.

Далее создадим остальные файлы языковых ресурсов — для всех прочих поддерживаемым приложением языков. Отметим, что в них мы должны использовать те же идентификаторы языковых ресурсов, что и в первом файле.

Например, так будет выглядеть содержимое аналогичного файла с языковыми ресурсами, переведенными на американский английский:

```
{
  "txtURLLabel": "URL",
  "btnURLValue": "Subscribe",
  "btnRefreshLabel": "Refresh",
  "btnSubscribeLabel": "Subscribe"
}
```

В результате для приложения, поддерживающего русский и английский языки, у нас должна получиться такая структура папок и файлов:

```
<папка проекта>
  strings
```

```
en-US
  resources.resjson
  (файл языковых ресурсов, переведенных на американский английский)
ru
  resources.resjson
  (файл языковых ресурсов, переведенных на русский)
```

Указание ссылок на языковые ресурсы в HTML-коде

Создав файлы языковых ресурсов, обратимся к файлам, описывающим интерфейс приложения. Нам следует вновь найти в них все локализуемые фрагменты текста и указать вместо них ссылки на соответствующие им языковые ресурсы.

Ссылка на фрагмент текста указывается в значении атрибута тега `data-win-res` в следующем формате:

```
{<свойство элемента интерфейса>: '<идентификатор фрагмента текста в файле
↳языкового ресурса>'}
```

Для примера возьмем фрагмент HTML-кода приложения `FeedReader`, создающий элементы управления для указания интернет-адреса канала новостей, и укажем в нем ссылки на созданные ранее языковые ресурсы.

```
<div>
  <label for="txtURL" data-win-res="{textContent: 'txtURLLabel'}">
  </label>
  <br />
  <input type="url" id="txtURL" />
</div>
<div>
  <input type="button" id="btnURL"
  data-win-res="{value: 'btnURLValue'}" />
</div>
```

Если требуется указать ссылки на фрагменты текста сразу для нескольких свойств данного элемента интерфейса, эти ссылки записываются друг за другом через запятую.

```
<input type="url" id="txtURL" data-win-res="{value: 'txtURLValue',
↳placeholder: 'txtURLPlaceholder'}" />
```

Локализация графических изображений

Многие приложения выводят на экран графические изображения, специфичные для какого-либо языка. К ним можно отнести национальные флаги, изображения, включающие в свой состав надписи, и т. п.

Понятно, что локализация должна затрагивать и графические изображения подобного рода. Как это сделать?

Почти так же, как и в случае локализуемых фрагментов текста. Мы создаем в папке, где хранятся локализуемые изображения, вложенные папки, соответствующие

всем поддерживаемым приложением языкам. В этих вложенных папках мы помещаем файлы с изображениями, специфичными для данного языка. В HTML-коде в этом случае ничего править не надо.

Предположим, что наше приложение выводит на экран изображения российского и американского национальных флагов. В таком случае мы создадим такую структуру папок и файлов:

```
<папка проекта>
  images
    en-US
      flag.png
      (файл с изображением национального флага США)
    ru
      flag.png
      (файл с изображением национального флага России)
  (нелокализуемые графические файлы)
```

Как уже говорилось, в теге, выводящем изображение флага, никаких изменений делать не нужно:

```
<img src="/images/flag.png" />
```

Если локализуемые изображения хранятся в папках, вложенных в папку images, то папки, соответствующие языкам, мы должны создать именно в них:

```
<img src="/images/flags/flag.png" />
. . .
<папка проекта>
  images
    flags
      en-US
        flag.png
        (файл с изображением национального флага США)
      ru
        flag.png
        (файл с изображением национального флага России)
  (нелокализуемые графические файлы)
```

Локализация строковых значений

Осталось локализовать строковые значения, что присутствуют в коде логики.

Сначала мы получим *диспетчер языковых ресурсов* — структуру данных, которая позволит нам получить из файла языковых ресурсов нужные фрагменты текста. Диспетчер языковых ресурсов представляется экземпляром объекта `Windows.ApplicationModel.Resources.ResourceLoader`, который нам следует создать.

```
var oRL = new Windows.ApplicationModel.Resources.ResourceLoader();
```

Чтобы получить нужный нам фрагмент текста, мы вызовем метод `getString` диспетчера языковых ресурсов. Этот метод в качестве единственного параметра при-

нимает строку с идентификатором получаемого фрагмента текста и возвращает в качестве результата этот фрагмент.

```
<div id="divAppBar" data-win-control="WinJS.UI.AppBar"
data-win-options="{placement: 'top'}">
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnRefresh' . . .}"></button>
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnSubscribe' . . .}"></button>
</div>
. . .
var ctrAppBar = document.getElementById("divAppBar").winControl;
var btnSubscribe = ctrAppBar.getCommandById("btnSubscribe");
var btnRefresh = ctrAppBar.getCommandById("btnRefresh");
btnSubscribe.label = oRL.getString("btnSubscribeLabel");
btnRefresh.label = oRL.getString("btnRefreshLabel");
```

Получаем из файла языковых ресурсов два фрагмента текста и выводим их на экран в качестве надписей для кнопок панели инструментов.

К сожалению, это единственный способ локализовать фрагменты текста, выводящиеся на экран в составе элементов управления Metro. Указать ссылки на них прямо в HTML-коде мы не сможем.

Инициализация языковых ресурсов

И самое последнее, что нам требуется сделать для локализации приложения, — выполнить инициализацию языковых ресурсов. В процессе инициализации платформа Metro просмотрит все HTML-файлы с описанием интерфейса и заменит ссылки на языковые ресурсы соответствующими им фрагментами текста.

Инициализацию языковых ресурсов следует выполнять в основном коде инициализации приложения — в обработчике события `DOMContentLoaded`. Выполняется она вызовом не принимающего параметров и не возвращающего результата метода `processAll` объекта `WinJS.Resources`. (Этот объект предоставляет различные служебные методы для работы с языковыми ресурсами. Единственный его экземпляр создает сама платформа Metro.)

```
document.addEventListener("DOMContentLoaded", function() {
  WinJS.Resources.processAll();
  . . .
});
```

Пример: локализация приложения для чтения каналов новостей

Практическое занятие по локализации мы проведем на примере приложения для чтения каналов новостей, которым уже давно не занимались. Давайте дадим ему поддержку еще и английского языка.

Откроем в Visual Studio проект `FeedReader`. Создадим в папке проекта вложенную папку strings, а в ней — папки ru и en-us.

Создадим в составе проекта файл языковых ресурсов resources.resjson. Зададим для него такое содержимое:

```
{
  "txtURLLabel": "Интернет-адрес",
  "btnURLValue": "Подписаться",
  "btnRefreshLabel": "Обновить",
  "btnSubscribeLabel": "Подписаться"
}
```

И переместим во вложенную папку ru.

Создадим еще один файл языковых ресурсов — на этот раз английских. Содержимое его будет таким:

```
{
  "txtURLLabel": "URL",
  "btnURLValue": "Subscribe",
  "btnRefreshLabel": "Refresh",
  "btnSubscribeLabel": "Subscribe"
}
```

Переместим этот файл во вложенную папку en-us.

Откроем файл default.html, отыщем код, создающий всплывающий элемент и панель инструментов, и изменим его следующим образом (исправленные фрагменты выделены полужирным шрифтом):

```
<div id="divURL" data-win-control="WinJS.UI.Flyout">
  <div>
    <label for="txtURL" data-win-res="{textContent: 'txtURLLabel'}">
    </label>
    <br />
    <input type="url" id="txtURL" />
  </div>
  <div>
    <input type="button" id="btnURL"
    data-win-res="{value: 'btnURLValue'}" />
  </div>
</div>

<div id="divAppBar" data-win-control="WinJS.UI.AppBar"
data-win-options="{placement: 'top'}">
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnRefresh', icon: 'refresh',
  section: 'selection'}"></button>
  <button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'btnSubscribe', icon: 'add', section: 'global',
  type: 'flyout', flyout: 'divURL'}"></button>
</div>
```

Помимо всего прочего, мы удалили у кнопок панели инструментов надписи. Поскольку мы все равно зададим их позднее, в коде логики, здесь они совершенно не нужны.

Переключимся на файл default.js и введем код, объявляющий переменную, что понадобится нам потом:

```
var oRL = new Windows.ApplicationModel.Resources.ResourceLoader();
```

Она будет хранить диспетчер языковых ресурсов.

Отыщем код инициализации приложения и добавим в него следующие выражения (выделены полужирным шрифтом):

```
document.addEventListener("DOMContentLoaded", function() {
  WinJS.Resources.processAll();
  WinJS.UI.processAll().then(function() {
    . . .
    btnSubscribe.label = oRL.getString("btnSubscribeLabel");
    btnRefresh.label = oRL.getString("btnRefreshLabel");
  });
});
```

Сохраним все исправленные файлы, запустим приложение на выполнение и проверим, загружает ли оно языковые ресурсы, соответствующие языку операционной системы.

К сожалению, пользовательские версии Windows 8 не позволяют сменить язык. Поэтому, чтобы проверить поддержку приложением остальных заявленных языков, нам придется установить другие языковые редакции этой системы. Проще всего сделать это, воспользовавшись каким-либо пакетом диспетчера виртуальных машин, например Oracle VirtualBox. Далее мы создадим дистрибутивный пакет приложения (как это сделать, будет рассказано в *главе 27*), перенесем его в соответствующую виртуальную машину и установим в работающей там копии Windows 8, после чего получим возможность запустить приложение и, наконец, проверить его работу.

Что дальше?

В этой главе мы занимались локализацией Metro-приложений — давали им поддержку различных языков. Разработчики платформы Metro заранее подумали об этом и предоставили нам все необходимые инструменты.

Следующая глава будет посвящена обеспечению поддержки Metro-приложениями устройств с различными параметрами экранов. Мы научимся поддерживать экраны с повышенной плотностью пикселов (а, судя по всему, за такими экранами будущее), изменять интерфейс приложений таким образом, чтобы с ними было удобно работать при любом — и ландшафтном, и портретном — положении устройства, и пользоваться преимуществами прикрепленного режима работы.

ГЛАВА 25

Адаптация Metro-приложений для устройств с различными параметрами экрана

В предыдущей главе мы переводили интерфейс наших Metro-приложений на другие языки. Это первое, что следует сделать в процессе коммерциализации приложений.

Второе — это обеспечение работы приложений на устройствах с разными параметрами экрана и в различных режимах (портретная и ландшафтная ориентация экрана; монопольный и прикрепленный режимы). Microsoft планирует устанавливать Windows 8 на самые разные устройства, и если приложение не будет на них работать, грош ему цена.

Тема эта столь велика, что на ее рассмотрение стоит отвести отдельную главу.

Экраны устройств, работающих под Windows 8, и их типичные параметры

В мире выпускается огромное количество различных мобильных устройств: ноутбуков, планшетов, мультимедийных проигрывателей, GPS-навигаторов и др. Экраны этих устройств имеют самые разнообразные характеристики, рассмотреть все многообразие которых просто невозможно.

Однако можно выделить несколько групп типичных устройств, экраны которых имеют примерно сходные параметры, в первую очередь разрешение, соотношение сторон и плотность пикселов. Давайте вкратце поговорим о них.

По разрешению экрана устройства можно разделить на три группы:

- экраны низкого разрешения — от 1024×768 пикселов включительно до 1366×768 пикселов исключительно. На устройствах с такими экранами обеспечивается работа Metro-приложений в полноэкранном режиме (подробнее о режимах работы приложений будет рассказано чуть позже);

Внимание!

Разрешение экрана 1024×768 пикселов является минимальным, на котором гарантируется работа Metro-приложений.

- экраны рекомендуемого разрешения — от 1366×768 пикселов включительно до 1920×1080 пикселов исключительно. На таких устройствах Metro-приложения могут работать в прикрепленном режиме;
- экраны высокого разрешения — от 1920×1080 пикселов включительно и более.

По соотношению сторон экрана устройства делятся также на три группы:

- широкие — 16:9. Обеспечивают максимальный комфорт при просмотре фильмов. Применяются, в основном, в пользовательских планшетах и мультимедийных проигрывателях;
- расширенные — 16:10. Также обеспечивают комфортный просмотр фильмов и, вдобавок, несколько лучше подходят для офисной работы. Применяются в пользовательских планшетах;
- традиционные ("телевизионные") — 4:3. Отлично подходят для офисной работы, вследствие чего часто ставятся в бизнес-планшеты.

На три группы можно разделить устройства и по плотности пикселов:

- экраны со стандартной плотностью пикселов — 96 пиксел/дюйм. Используются в бюджетных устройствах;
- экраны с повышенной плотностью пикселов — 148 пиксел/дюйм. Обеспечивают более высокое качество изображения. Ставятся в более дорогие устройства;
- экраны с высокой плотностью пикселов — 208 пиксел/дюйм. Дают максимальное на данный момент качество изображения. Устанавливаются в устройства высшего ценового диапазона.

Именно на эти группы и следует ориентироваться при разработке приложений, адаптированных под любые экраны.

Еще следует помнить, что устройства могут использоваться в различном положении, или, как еще говорят, ориентации.

- *Ландшафтная* ориентация — это ориентация, при которой длинная сторона экрана устройства располагается горизонтально. В такой ориентации обычно смотрят фильмы.
- *Портретная* ориентация — это ориентация, при которой длинная сторона экрана устройства располагается вертикально. Такая ориентация лучше подходит для чтения больших текстов.

Режимы работы Metro-приложения

Помимо различных экранов, нужно принимать к сведению то, что Metro-приложение может работать в трех различных режимах. Эти режимы различаются долей экранного пространства, которое может занять интерфейс приложения.

- *Полноэкранный режим* (fullscreen), в котором интерфейс приложения полностью занимает весь экран устройства. По умолчанию любое приложение работает именно в этом режиме.
- *Прикрепленный режим* (snapped), в котором приложение занимает небольшое пространство вдоль левой или правой стороны экрана. Перевести приложение в прикрепленный режим может только пользователь.

  Чтобы приложения получили возможность работы в прикрепленном режиме, экран устройства должен иметь ширину не менее 1366 пикселов. При этом ширина интерфейса прикрепленного приложения составит 320 пикселов; еще 22 пиксела займет полоса, отделяющая его от остального пространства экрана.
- *Режим заполнения* (fill), когда приложение делит экран с другим, работающим в прикрепленном режиме, и, соответственно, занимает оставшуюся часть экрана.

Интересно, что оба приложения — и прикрепленное, и работающее в режиме заполнения — будут выполняться одновременно. Таким образом обеспечивается своего рода многозадачность, получить которую другими средствами в платформе Metro невозможно.

Когда следует адаптировать приложения под различные экраны

Из всего сказанного ранее может показаться, что каждое Metro-приложение требует обязательной подстройки под различные экраны. Но это отнюдь не так. Приложений, действительно требующих адаптации под экраны, не так уж и много. Давайте рассмотрим несколько типичных случаев.

- Приложение, которое выводит на экран большие текстовые документы. В ландшафтном режиме этот текст может быть отформатирован в несколько колонок с горизонтальной прокруткой. Напротив, в портретном или прикрепленном режиме текст лучше вывести в одну колонку с вертикальной прокруткой — так будет удобнее.
- Приложение, выводящее в левой части экрана, скажем, список каких-либо позиций, а в правой — данные, относящиеся к позиции, выбранной в списке. В ландшафтном режиме такой вариант расположения элементов интерфейса будет идеален. Но в портретном или прикрепленном режиме удобнее будет разместить список и данные по выбранной позиции вертикально, друг над другом.

  Так, написанные нами "читалка" каналов новостей `FeedReader` и утилита для поиска изображений в сервисе Bing `ImageSearcher` — прямо-таки хрестоматийные примеры такого рода приложений.
- Приложение с большим количеством элементов интерфейса. В прикрепленном режиме следует скрыть те из них, что нужны только в определенные моменты времени. Например, в видеопроигрывателе в этом случае можно скрыть кнопки остановки, закрытия файла и вызова настроек.

- Приложение, выводящее список с большим количеством позиций. В прикрепленном режиме мы можем уменьшить количество выводимых позиций, а на экранах высокого разрешения — наоборот, увеличить.
- Приложения, в чьем интерфейсе активно используются свободно позиционируемые элементы (подробнее о них *см. в главе 9*). Чтобы правильно позиционировать такие элементы, нам требуется знать, по крайней мере, разрешение экрана.

Так или иначе рекомендуется проверить, как приложение будет выглядеть на разных экранах и в различных режимах работы. В этом нам поможет встроенный в Visual Studio симулятор, речь о котором пойдет в конце этой главы.

Медиазапросы CSS

Но как мы можем указать местоположение элементов интерфейса и прочие их параметры для разных экранов и различных режимов работы? Какие механизмы платформы Metro мы для этого задействуем?

Введение в медиазапросы

Для начала — самый простой из них, носящий название *медиазапросов* CSS. Их можно рассматривать как набор условий, указываемый для группы стилей CSS, при которых стили, входящие в эту группу, будут иметь силу. Такими условиями в нашем случае являются различные параметры экрана и режимы работы приложения.

Предположим, что мы создаем приложение, предназначенное для использования только в ландшафтной ориентации экрана. Мы определяем месторасположение элементов его интерфейса и создаем стили, которые разместят элементы так, как нам нужно.

И тут нам дают распоряжение — срочно обеспечить в приложении возможность работы в портретном режиме. Нет ничего проще!

Сначала мы создадим медиазапрос, для которого в качестве условия укажем ландшафтную ориентацию устройства. В него мы переместим созданные ранее стили, устанавливающие месторасположение элементов интерфейса для ландшафтной ориентации.

После этого мы напишем еще один медиазапрос, но укажем для него уже портретную ориентацию. В этот медиазапрос мы включим вновь созданные стили, которые зададут другое расположение элементов интерфейса — подходящее для портретной ориентации.

Отметим, что все эти манипуляции мы выполняем в файле default.css — в таблице стилей, описывающей оформление приложения. Описание его интерфейса и логики мы вообще не трогаем.

И вот, приложение получает конечный пользователь. Когда он работает со своим планшетом в ландшафтной ориентации, приложение использует стили, входящие

в состав первого медиазапроса. Если же он повернет планшет, установив его в портретную ориентацию, приложение задействует стили, что входят в состав второго медиазапроса. Для пользователя это будет выглядеть так, словно приложение само переупорядочило элементы своего интерфейса соответственно новой ориентации устройства.

Точно так же мы сможем добавить приложению и поддержку, например, прикрепленного режима — мы просто создадим еще один медиазапрос.

Мы можем создать медиазапросы для различных разрешений экрана, для разных соотношений их сторон и различной величины плотности пикселов. Язык CSS включает для этого все средства.

Создание медиазапросов

Медиазапросы создаются по следующему шаблону:

```
@media <устройство вывода> and <список условий> {
  <стили, входящие в медиазапрос>
}
```

В качестве устройства вывода мы можем указать экран (`screen`), принтер (`print`) или любое устройство (`all`).

Условия для медиазапроса записываются в следующем формате:

```
(<имя параметра>: <значение параметра>)
```

Отдельные условия в списке отделяются друг от друга пробелом, ключевым словом `and` и еще одним пробелом.

```
@media screen and (-ms-view-state: fullscreen-landscape) {
  . . .
}
```

Этот медиазапрос будет действовать при выводе на экран, при ландшафтной ориентации и в полноэкранном режиме. (Параметр `-ms-view-state` задает одновременно и ориентацию экрана устройства, и режим работы приложения.)

```
@media screen and (-ms-view-state: fullscreen-landscape) and
↳(min-resolution: 148dpi) {
  . . .
}
```

А этот медиазапрос будет действовать при выводе на экран, при ландшафтной ориентации, в полноэкранном режиме приложения и при условии, что минимальная плотность пикселов экрана равна 148 пиксел/дюйм. (Параметр `min-resolution` как раз устанавливает минимальную плотность пикселов.)

Кстати, Visual Studio при создании любого проекта формирует в файле default.css четыре медиазапроса. Все они работают при выводе на экран и задействуются, соответственно, в случаях:

- ландшафтной ориентации и полноэкранного режима;
- режима заполнения;

- прикрепленного режима;
- портретной ориентации и полноэкранного режима.

Эти медиазапросы мы можем увидеть в конце кода изначальной таблицы стилей, приведенном в *разд. "Создание оформления" главы 2*. Так что, если нам потребуется создать медиазапрос такого рода, мы можем воспользоваться этими заготовками.

Написание условий для медиазапросов

Теперь давайте познакомимся с параметрами, применяемыми в условиях медиазапросов, и их назначением. Этих параметров довольно много.

- `min-width` и `max-width` — задают, соответственно, минимальную и максимальную ширину экрана устройства.

```
@media screen and (max-width: 1024px) {
  #divContent { column-count: 3; }
}
@media screen and (max-width: 1366px) {
  #divContent { column-count: 4; }
}
```

  Если ширина экрана устройства не превышает 1024 пиксела, верстаем текст в три колонки. Если же устройство имеет экран шириной от 1025 до 1366 пикселов, текст будет сверстан в четыре колонки.

- `min-height` и `max-height` — задают, соответственно, минимальную и максимальную высоту экрана устройства.

```
@media screen and (min-height: 900px) {
  #divList .win-item { height: 20%; }
}
@media screen and (max-height: 899px) {
  #divList .win-item { height: 33%; }
}
```

  Если ширина экрана устройства составляет 900 пикселов и более, устанавливаем для пунктов списка Metro высоту, равную 1/5 от высоты списка; в результате список будут отображать пять пунктов. В противном случае выводим в списке только три пункта.

- `width` и `height` — задают точное значение, соответственно, ширины и высоты экрана устройства.

```
@media screen and (width: 1920px) {
  #divList .win-item { height: 33%; }
}
```

- `min-aspect-ratio` и `max-aspect-ratio` — задают, соответственно, минимальное и максимальное значения соотношения сторон экрана.

```
@media screen and (max-aspect-ratio: 16/10) {
  #divList .win-item {
    width: 20%;
    height: 33%;
  }
}
```

Если соотношение сторон экрана устройства не превышает 16:10, устанавливаем для пунктов списка Metro ширину в 1/5 от ширины списка и высоту в 1/3 от его высоты.

- `aspect-ratio` — задает точное значение соотношения сторон экрана.

```
@media screen and (device-aspect-ratio: 4/3) {
  #divList .win-item {
    width: 50%;
    height: 33%;
  }
}
```

- `min-resolution` и `max-resolution` — задают, соответственно, минимальную и максимальную величину плотности пикселов экрана в пиксел/дюйм (единица измерения `dpi`).

```
@media screen and (min-resolution: 208dpi) {
  #divList .win-item {
    width: 10%;
    height: 20%;
  }
}
```

Если плотность пикселов у экрана устройства составляет, как минимум, 208 пиксел/дюйм, устанавливаем для пунктов списка Metro ширину в 1/10 от ширины списка и высоту в 1/5 от его высоты.

- `resolution` — задает точное значение плотности пикселов экрана в пиксел/дюйм.

```
@media screen and (resolution: 96dpi) {
  #divList .win-item {
    width: 33%;
    height: 33%;
  }
}
```

- `-ms-view-state` — задает одновременно ориентацию экрана и режим работы приложения. Доступны следующие значения:
  - `fullscreen-landscape` — ландшафтная ориентация и полноэкранный режим;
  - `fullscreen-portrait` — портретная ориентация и полноэкранный режим;
  - `snapped` — прикрепленный режим;
  - `filled` — режим заполнения.

```
@media screen and (-ms-view-state: snapped) {
  #divDetail { display: none; }
}
```

Скрываем блок `divDetail`, если приложение работает в закрепленном режиме.

- `orientation` — задает ориентацию экрана устройства. Доступны значения `portrait` (портретная ориентация) и `landscape` (ландшафтная ориентация).

```
@media screen and (orientation: portrait) {
  #divList .win-item {
    width: 50%;
    height: 33%;
  }
}
@media screen and (orientation: landscape) {
  #divList .win-item {
    width: 33%;
    height: 50%;
  }
}
```

Меняем размеры пунктов списка соответственно ориентации экрана.

- `min-color` и `max-color` — задают, соответственно, минимальное и максимальное количество битов, отводимых на описание цвета одного пиксела.

```
@media screen and (max-color: 2) {
  body {
    color: white;
    background-color: black;
  }
}
```

Если экран устройства поддерживает максимум двухбитный цвет (четыре градации серого), задаем для интерфейса приложения белый цвет, а для его фона — черный.

- `color` — задает точное количество битов, отводимых на описание цвета одного пиксела.

```
@media screen and (color: 1) {
  #divDetail { border: none; }
}
```

Если устройство поддерживает только однобитный цвет (т. е. имеет монохромный экран), убираем у блока `divDetail` рамку.

Пример: адаптация приложения для чтения каналов новостей под портретную ориентацию устройства

Практиковаться в адаптации приложений под различные экраны мы будем на "читалке" каналов новостей. Давайте сделаем так, чтобы при портретной ориентации устройства все элементы интерфейса этого приложения располагались по вертикали.

Откроем в Visual Studio проект `FeedReader`. Откроем файл default.css, найдем стиль переопределения тега `body` и именованные стили `#divHeader`, `#divList` и `#divContent` и поместим их в медиазапрос следующего вида:

```
@media screen and (-ms-view-state: fullscreen-landscape) {
  body { . . . }
  #divHeader { . . . }
  #divList { . . . }
  #divContent { . . . }
}
```

Благо этот медиазапрос уже создан Visual Studio, и нам останется только перенести в него перечисленные ранее стили. А действовать он будет в случае, если устройство находится в ландшафтной ориентации, а приложение работает в полноэкранном режиме.

Далее отыщем другой медиазапрос, также созданный Visual Studio и действующий в случае, если устройство находится в портретной ориентации:

```
@media screen and (-ms-view-state: fullscreen-portrait) {
}
```

И создадим в нем такие стили:

```
@media screen and (-ms-view-state: fullscreen-portrait) {
  body {
    display: -ms-grid;
    -ms-grid-columns: 1fr;
    -ms-grid-rows: auto 1fr 1fr;
  }
  #divHeader { -ms-grid-column-align: center; }
  #divList {
    -ms-grid-row: 2;
    height: 100%;
  }
  #divContent { -ms-grid-row: 3; }
}
```

Сохраним исправленный файл и запустим приложение в симуляторе Visual Studio (разговор о нем пойдет в конце этой главы). Установим симулятор в портретную ориентацию и посмотрим, правильно ли выводится интерфейс приложения.

Адаптация изображений под разные значения плотности пикселов

Если наше приложение выводит в составе своего интерфейса какие-либо графические изображения, то при запуске его на экранах с разными значениями плотности пикселов мы столкнемся с тем, что размеры этих изображений визуально меняются. Что и понятно, ведь растровые графические изображения — просто набор точек, и чем плотнее пикселы расположены на экране, тем меньше размеры выводимых на нем изображений.

Если мы хотим, чтобы изображения выглядели одинаково на экранах с разной плотностью пикселов, нам придется создать несколько версий каждого из этих изображений — по каждой на одно значение плотности. Обычно создаются по три версии изображения — для значений плотности 96, 148 и 208 пиксел/дюйм. Как говорилось в начале этой главы, данные значения выбраны самой Microsoft в качестве стандартных.

Файлам, хранящим версии изображения, даются имена следующего вида:

```
<изначальное имя файла>.scale-<относительное значение плотности пикселов,
↳для которого предназначен этот файл>.<расширение файла>
```

Отметим, что здесь указывается не абсолютное значение плотности в пикселах на дюйм, а относительное, в процентах:

- 100 — для значения плотности 96 пиксел/дюйм;
- 140 — для значения плотности 148 пиксел/дюйм;
- 180 — для значения плотности 208 пиксел/дюйм.

Как вариант, мы можем поместить файлы, содержащие разные версии изображения, во вложенные папки с именами:

- scale-100 — для значения плотности 96 пиксел/дюйм;
- scale-140 — для значения плотности 148 пиксел/дюйм;
- scale-180 — для значения плотности 208 пиксел/дюйм.

Имена этих файлов в таком случае менять не надо.

Платформа Metro при запуске приложения сама определит плотность пикселов экрана и загрузит соответствующий ему графический файл. При этом в тегах `<img>`, вводящих эти изображения, ничего менять не нужно.

Пусть наше приложение выводит на экран графический логотип, хранящийся в файле logo.png.

```
<img src="/images/logo.png" />
```

Тогда у нас есть два пути. Либо мы создадим три файла, хранящих разные версии этого логотипа:

```
<папка проекта>
  images
```

```
    logo.scale-100.png
    logo.scale-140.png
    logo.scale-180.png
```

Либо мы поместим эти файлы во вложенные папки:

```
<папка проекта>
  images
    scale-100
      logo.png
    scale-140
      logo.png
    scale-180
      logo.png
```

Программное определение размеров экрана и ориентации устройства

Очень часто приходится определять размеры экрана и ориентацию устройства программно. Например, если наше приложение выводит часть своего интерфейса в свободно позиционируемом элементе, нам придется определить размеры экрана устройства, чтобы правильно позиционировать этот элемент.

Определить размеры экрана можно с помощью методов `getTotalWidth` и `getTotalHeight` объекта `WinJS.Utilities`. Эти методы и этот объект знакомы нам еще по *главе 9*.

```
var iScreenWidth = WinJS.Utilities.getTotalWidth(document.body);
var iScreenHeight = WinJS.Utilities.getTotalHeight(document.body);
```

Определить ориентацию устройства чуть сложнее.

Сначала мы получим доступ к датчику ориентации, встроенному в устройство. Выполняется это вызовом не принимающего параметров метода `getDefault` объекта `Windows.Devices.Sensors.SimpleOrientationSensor`.

В качестве результата метод `getDefault` возвращает экземпляр этого объекта, представляющий датчик ориентации. Если таковой в устройстве отсутствует, возвращается `null`.

```
var oOS = Windows.Devices.Sensors.SimpleOrientationSensor.getDefault();
if (oOS) {
  // Датчик ориентации получен
}
```

Объект `Windows.Devices.Sensors.SimpleOrientationSensor`, представляющий датчик ориентации, поддерживает метод `getCurrentOrientation`. Он не принимает параметров и возвращает один из элементов перечисления `Windows.Devices.Sensors.SimpleOrientation`, указывающий текущую ориентацию устройства:

- `notRotated` — портретная ориентация;
- `rotated90DegreesCounterclockwise` — ландшафтная ориентация;

- `rotated180DegreesCounterclockwise` — портретная ориентация; устройство перевернуто;
- `rotated270DegreesCounterclockwise` — ландшафтная ориентация; устройство перевернуто;
- `faceup` — устройство расположено горизонтально экраном вверх;
- `facedown` — устройство расположено горизонтально экраном вниз.

```
<div id="divList" data-win-control="WinJS.UI.ListView"></div>
. . .
var ctrList = document.getElementById("divList").winControl;
var orientation = oOS.getCurrentOrientation();
if (orientation == Windows.Devices.Sensors.SimpleOrientation.notRotated)
{
  ctrList.layout = new WinJS.UI.ListLayout();
}
if (orientation == Windows.Devices.Sensors.SimpleOrientation.
↳rotated90DegreesCounterclockwise) {
  ctrList.layout = new WinJS.UI.GridLayout();
}
```

Если устройство имеет портретную ориентацию, задаем для списка `divList` расположение пунктов по вертикали с вертикальной прокруткой. В случае ландшафтной ориентации устанавливаем для этого списка расположение пунктов сначала по вертикали, а потом — по горизонтали с горизонтальной прокруткой.

Да, но вот только пользователь может изменить ориентацию устройства в любой момент времени. Как это отследить?

Как только ориентация устройства изменяется, в датчике ориентации возникает событие `orientationchanged`. Экземпляр объекта, представляющий событие и передаваемый обработчику в качестве единственного параметра, поддерживает свойство `orientation`. Это свойство хранит один из элементов перечисления `Windows.Devices.Sensors.SimpleOrientation`, указывающий новую ориентацию устройства.

```
oOS.addEventListener("orientationchanged", oOSOrientationChanged);
function oOSOrientationChanged(evt) {
  if (evt.orientation ==
  Windows.Devices.Sensors.SimpleOrientation.notRotated) {
    ctrList.layout = new WinJS.UI.ListLayout();
  }
  if (orientation == Windows.Devices.Sensors.SimpleOrientation.
  ↳rotated90DegreesCounterclockwise) {
    ctrList.layout = new WinJS.UI.GridLayout();
  }
}
```

Теперь наш список будет менять порядок расположения пунктов "на лету".

Тестирование приложений при помощи симулятора

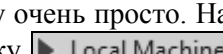
Когда мы запускаем приложение, Visual Studio собирает все файлы проекта воедино, формируя из него установочный пакет. (Профессиональные разработчики называют этот процесс *сборкой*.) Далее он осуществляет установку приложения из этого пакета и выполняет его запуск. В результате приложение работает в том же окружении, что и сам Visual Studio.

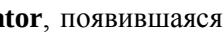
Ранее нас это вполне устраивало. Но сейчас, разрабатывая приложения, адаптирующиеся к различным экранам, мы должны будем проверять его на экранах с разными параметрами. Как это сделать?

Специально для такого случая Visual Studio включает в свой состав так называемый *симулятор*. Его можно рассматривать как компьютер, функционирующий "внутри" нашего компьютера. Он имеет свой "процессор", свою "память", свою копию Windows 8, уже установленную и полностью работоспособную, и свой экран, для которого мы можем сами указать размер и ориентацию.

Если мы запустим приложение в симуляторе, Visual Studio переместит установочный пакет в его память и именно там выполнит установку приложения, которое после этого будет успешно функционировать в "воображаемом" компьютере.

> ***Внимание!***
>
> Чтобы симулятор успешно работал, нам требуется задать пароль для учетной записи, под которой мы входим в систему.

Выбрать симулятор как целевую платформу очень просто. Найдем на панели инструментов главного окна Visual Studio кнопку 

. В ее правой части находится небольшая стрелка, направленная вниз. Нажмем эту стрелку и выберем в появившемся на экране меню пункт **Simulator**. После чего данная кнопка примет такой вид — 

; надпись **Simulator**, появившаяся на ней, означает, что отныне все приложения будут запускаться в симуляторе.

Запустим приложение, как выполняли это ранее. Подождем, пока на экране не появится окно симулятора (рис. 25.1). Выполним вход в копию Windows 8, что установлена в симуляторе, выбрав учетную запись и введя ее пароль. И дождемся, наконец, момента, когда приложение запустится.

Поскольку симулятор представляет собой вполне полноценный компьютер, хоть и сформированный чисто программными средствами, мы можем делать с запущенным в нем приложением все, что захотим. Мы можем нажимать кнопки, вводить текст, устанавливать и сбрасывать флажки, выбирать пункты в списках и пр. Мы даже можем выйти в меню **Пуск** системы, установленной в симуляторе, и запустить оттуда какое-либо другое приложение.

А еще мы можем менять параметры экрана симулятора. Выполняется это с помощью кнопок, расположенных вдоль правого края его окна.

Кнопка **Rotate Simulator clockwise 90 degrees** () выполняет поворот экрана симулятора на 90° по часовой стрелке, а кнопка **Rotate Simulator counter-clockwise 90 degrees** () — на 90° против часовой стрелки. Мы можем пользоваться этими кнопками, чтобы изменить ориентацию симулятора.

Если нажать кнопку **Change Resolution** (), на экране появится меню, из которого мы можем изменить размеры экрана симулятора. Названия пунктов этого меню указаны в формате ***<размер экрана в дюймах>*in. *<ширина экрана в пикселах>* X *<высота экрана в пикселах>***, так что мы сразу поймем, какой пункт выбрать.

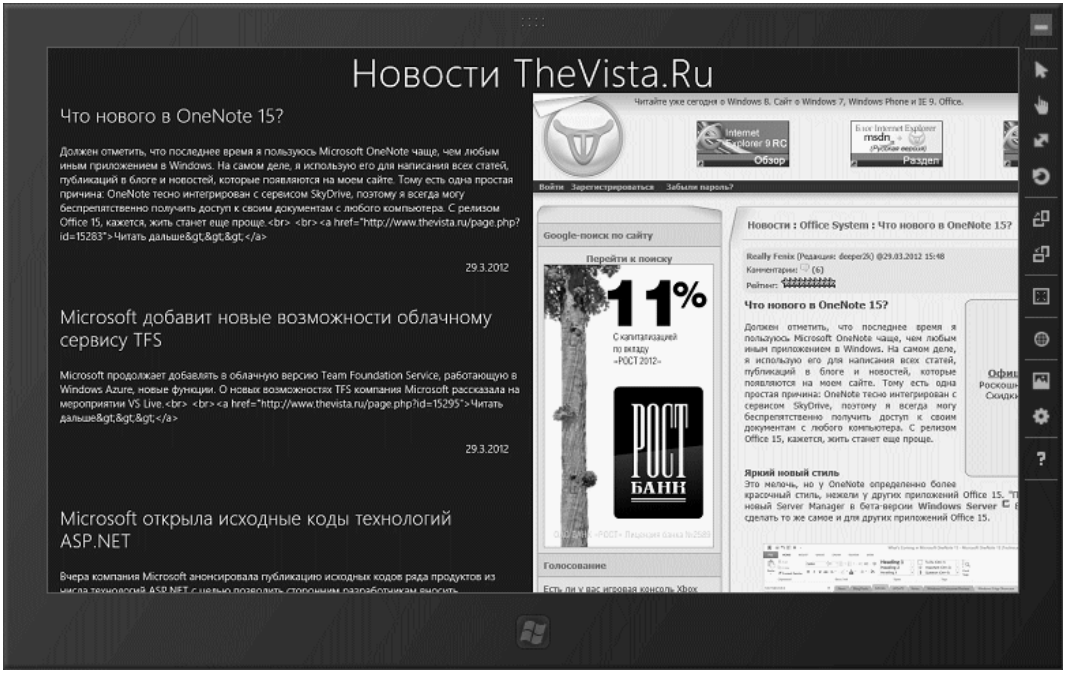
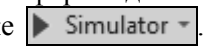


Рис. 25.1. Окно симулятора с запущенным в нем приложением

Закончив работу с симулятором, переключимся в окно Visual Studio и завершим работающее приложение тем же способом, которым пользовались еще в *главе 2*. Через некоторое время симулятор будет закрыт, и его окно исчезнет.

Чтобы снова указать в качестве платформы для запуска приложений свой компьютер, мы опять обратимся к кнопке 

. Нажмем присутствующую в ее правой части кнопку и выберем в появившемся меню пункт **Local Machine**. И все приложения с этого момента будут запускаться как обычно.

Что дальше?

В этой главе мы адаптировали наши приложения под устройства с различными параметрами экранов. Мы узнали, как подстраивать интерфейс приложений под экраны с ландшафтной и портретной ориентацией, под экраны с разными размерами, соотношением сторон и плотностью пикселов. Или, говоря по-простому, сделать приложения "всеядными".

Следующая глава пригодится, когда мы решим, что наше приложение заслужило достойную оплату. Мы выясним, как создать условно-бесплатное приложение с ограниченным сроком пользования или функциональностью. А еще мы реализуем покупку приложения или части его функциональности прямо из его интерфейса (чтобы пользователь не успел передумать).

ГЛАВА 26

Создание коммерческих Metro-приложений

В предыдущей главе мы адаптировали Metro-приложения для работы на устройствах с различными параметрами экрана. Мы разобрались, как модифицировать интерфейс приложений для работы в ландшафтной и портретной ориентации, в прикрепленном режиме и режиме заполнения, на экранах с разным разрешением и плотностью пикселов. Теперь нашим приложением будет удобно пользоваться на планшетах с самыми разными характеристиками.

Заработать на продаже плодов своего труда — вполне законное и закономерное желание. Очень и очень многие Metro-программисты захотят получить за свои приложения нечто большее, чем простое спасибо.

И платформа Metro идет им навстречу. Пользуясь только ее стандартными механизмами, мы можем создать условно-бесплатное приложение и дать пользователю возможность купить его прямо из интерфейса этого приложения, нажав специальную кнопку. Мы даже можем оснастить приложение набором функций, которые будут доступны только за отдельную плату. И все это — без малейшего труда.

Бесплатные, условно-бесплатные и платные приложения

Все приложения (не только разработанные для платформы Metro) по модели их распространения делятся на три группы:

- *бесплатные* — вообще не требуют оплаты;
- *условно-бесплатные* — могут использоваться без оплаты в режиме ограниченной функциональности; полная же функциональность будет доступна только после оплаты;
- *платные* — требуют оплаты непосредственно перед их установкой.

С бесплатными и платными приложениями все понятно. Условно-бесплатные же требуют более глубокого рассмотрения.

Термин "условно-бесплатное приложение" в терминологии маркетинга ПО означает, что данное приложение *может* использоваться без оплаты. Однако в этом случае полноценно пользоваться приложением невозможно. Будучи еще не оплаченным, такое приложение работает в тестовом режиме и, по большому счету, выступает как демонстрационная версия, призванная показать пользователю, что он может получить после оплаты.

Условно-бесплатные приложения могут демонстрировать свои потенциальные возможности в трех разных режимах.

- Режим работы в течение ограниченного периода времени *(тестового периода)*. При этом пользователь может задействовать все функции данного приложения без ограничения, как если бы он его уже оплатил. Однако по завершении тестового периода приложение либо перестает запускаться, либо делает бóльшую часть своей функциональности недоступной, пока пользователь не произведет оплату.
- Режим работы в режиме ограниченной функциональности. Приложением можно пользоваться неограниченное время, но доступен будет только ограниченный набор функций — самых базовых. Чтобы получить доступ к полной функциональности, пользователь должен оплатить приложение.
- Комбинированный режим. В течение тестового периода пользователь имеет доступ к полной функциональности приложения, а по его истечении — только к ограниченной.

Каких-либо особых преимуществ для разработчика приложения ни один из этих режимов не имеет. На практике обычно реализуется режим работы в течение тестового периода.

Кроме того, приложение может иметь набор функций, которые требуется покупать дополнительно, независимо от самого приложения (если это приложение не бесплатное). Такие функции могут реализовываться в любом приложении — платном, условно-бесплатном или бесплатном.

Теперь поговорим о том, за какие приложения имеет смысл брать деньги, и если брать, то как.

- Бесплатно обычно распространяются самые простые приложения, использующие только стандартные средства платформы Metro. К таким можно отнести все приложения, написанные нами в процессе изучения этой книги. Также бесплатно часто распространяются экспериментальные приложения, приложения, поставляемые в комплекте с подключаемыми устройствами (цифровыми камерами, флэш-дисками и др.), и вспомогательные приложения — утилиты.
- Платными обычно делают сложные программные пакеты, на разработку которых были затрачены значительные ресурсы (офисные пакеты, специализированное ПО, развитые игры и др.).
- Условно-бесплатная модель распространения наилучшим образом подходит для приложений, с одной стороны, не таких сложных, чтобы делать их полностью платными, а с другой, реализующих какие-либо ноу-хау, что выходят за рамки стандартных возможностей платформы Metro. К приложению такого рода мож-

но отнести видеопроигрыватель, позволяющий воспроизводить форматы файлов, которые не поддерживаются Windows 8.

- Отдельно приобретаемыми имеет смысл делать функции приложения, опять же, реализующими какие-либо ноу-хау. В качестве примера можно привести приложение для работы с каналами новостей, которое преобразует содержимое канала в полностью оформленный документ Microsoft Word.

Коммерческое предложение Microsoft

Microsoft позволяет публиковать в магазине Windows Store приложения, распространяемые по любой из перечисленных ранее моделей: бесплатные, условно-бесплатные и платные. Также возможно распространение приложений, включающих в себя платные функции.

Минимальная цена приложения или его отдельной функции составляет 1,49 долл. США. Максимальная цена не ограничена.

Microsoft гарантирует прием оплаты за приложения в любой из валют, имеющих хождение на поддерживаемых Windows Store рынках. В число этих валют входит и российский рубль.

Разработчик получает 70% суммы дохода от продажи своего приложения; эти 70% составляют чистую прибыль разработчика. В случае если доход от продажи составит 25 тыс. долл. США и более, разработчик станет получать уже 80% от этой суммы. Остальные, соответственно, 30 и 20% дохода удерживаются Microsoft и идут на поддержку магазина приложений.

Как видим, коммерческое предложение Microsoft весьма выгодно.

Создание условно-бесплатного приложения

За бесплатные приложения платить не нужно — это ясно по определению. За платные приложения платить нужно сразу же — иначе пользователь просто не сможет их загрузить и установить.

А как обстоит дело с условно-бесплатными приложениями? Ведь нам надо и дать пользователю попробовать приложение в деле и заставить его впоследствии раскошелиться.

Ранее весь код, отслеживающий состояние оплаты и соответственно управляющий функциональностью условно-бесплатного приложения, разработчик должен был писать сам. Разумеется, существовали специализированные программные пакеты, но были они весьма дорогостоящие и не всегда удобные в использовании. Сейчас же ничего этого не нужно — ведь у нас есть платформа Metro!

Лицензия приложения

Когда мы опубликуем наше приложение в магазине Windows Store, мы, помимо прочих сведений о приложении, сформируем для него так называемую *лицензию*.

Она включит данные о модели распространения приложения — будет оно бесплатным, условно-бесплатным или платным.

В случае условно-бесплатного приложения (нас интересует именно этот случай) лицензия приложения также продолжительность тестового периода. Если мы хотим реализовать работу приложения в режиме ограниченной функциональности, то укажем очень большой тестовый период — скажем, до 2199 года.

База данных магазина Windows Store сохранит все указанные нами данные о лицензии. Впоследствии в этой базе будет зафиксирован и сам факт покупки приложения пользователем.

Получение сведений о лицензии

Первое, что нам следует сделать при запуске приложения, прямо в его коде инициализации, — запросить у Windows Store лицензию. В частности, нас интересует, работает ли приложение в тестовом режиме (т. е. было ли приложение приобретено), закончился ли уже тестовый период, и если не закончился, то сколько дней осталось до его завершения. Получив все эти данные, мы можем соответственно ограничить функциональность приложения и вывести элементы интерфейса, призывающие пользователя сделать покупку, или, напротив, дать ему доступ к полной функциональности приложения.

Лицензия приложения возвращается свойством `licenseInformation` объекта `Windows.ApplicationModel.Store.CurrentAppSimulator`. (Этот объект предоставляет средства для работы с лицензией приложения и реализации его оплаты. Единственный его экземпляр создается самой платформой Metro.)

> ***Внимание!***
>
> Объект `Windows.ApplicationModel.Store.CurrentAppSimulator` следует использовать только для тестирования приложения. Перед его распространением обязательно нужно заменить его объектом `Windows.ApplicationModel.Store.CurrentApp`.

```
var oLI = Windows.ApplicationModel.Store.CurrentAppSimulator.
↳licenseInformation;
```

Сама лицензия представляется экземпляром объекта `Windows.ApplicationModel.Store.LicenseInformation`. Этот объект поддерживает три свойства, значения которых помогут нам узнать, работает ли приложение в тестовом режиме и сколько еще ему осталось работать.

Прежде всего, нам понадобится свойство `isActive`. Оно возвращает `true`, если лицензия данного условно-бесплатного приложения активна, т. е. тестовый период еще не закончился. Если же тестовый период уже закончился, данное свойство возвращает значение `false`.

Далее мы проверим свойство `isTrial`. Оно возвращает `true`, если данное приложение работает в тестовом режиме, и `false`, если оно уже оплачено.

Дата и время окончания тестового периода хранится в свойстве `expirationDate` в виде экземпляра объекта `Date`. Чтобы получить количество дней, оставшееся до

окончания тестового периода, мы можем вычесть из этого значения текущую дату и разделить получившуюся разность на 86 400 000.

```
if (oLI.isActive) {
  if (oLI.isTrial) {
    //Тестовый период еще не закончился.
    //Делаем приложение ограниченно работоспособным и выводим элементы
    //интерфейса, позволяющие купить приложение
    var iDaysRemaining = (oLI.expirationDate — new Date()) / 86400000;
    //Получаем количество дней, оставшихся до завершения тестового
    //периода, и выводим полученную величину на экран
  } else {
    //Приложение оплачено.
    //Делаем приложение полностью работоспособным и скрываем элементы
    //интерфейса, позволяющие купить приложение
  }
} else {
  //Тестовый период закончился.
  //Делаем приложение неработоспособным и выводим элементы
  //интерфейса, позволяющие купить приложение
}
```

Этот фрагмент кода можно использовать как шаблон для реализации переключения приложения в тестовый либо полнофункциональный режим в зависимости от того, было ли приложение куплено.

Получение сведений о приложении из магазина Windows Store

Хорошо, пользователь узнал, что приложение работает в тестовом режиме и до "часа ноль" осталось столько-то дней. Неплохо было бы дополнить эти сведения суммой, которую ему придется выложить, чтобы продолжать пользоваться приложением.

Конечно, мы можем вписать стоимость приложения вручную. Но зачем это делать, если можно запросить ее в магазине Windows Store!

Объект `Windows.ApplicationModel.Store.CurrentAppSimulator` поддерживает не принимающий параметров метод `loadListingInformationAsync`, который запускает процесс получения информации о приложении из Windows Store. В качестве результата он возвращает обязательство, для которого мы в вызове метода `then` укажем функцию, которая выполнится после получения всех сведений.

Единственным параметром, полученным данной функцией, станет экземпляр объекта `Windows.ApplicationModel.Store.ListingInformation`. Он как раз и содержит некоторые сведения о приложении, хранящиеся в базе данных магазина.

Нам, в первую очередь, понадобится свойство `formattedPrice` этого объекта. Оно возвращает стоимость приложения в валюте текущего рынка с указанием ее наименования в виде строки.

```
Windows.ApplicationModel.Store.CurrentAppSimulator.
↳loadListingInformationAsync().then(function(oListI) {
  var sPrice = oListI.formattedPrice;
  . . .
});
```

Еще нам могут пригодиться два других свойства объекта `Windows.ApplicationModel.Store.ListingInformation`. Свойство `name` хранит название приложения для текущего рынка в виде строки, а свойство `description` — описание приложения также для текущего рынка и в виде строки.

Реализация покупки приложения

Высший шик программирования — реализация возможности покупки приложения прямо из его интерфейса, нажатием одной кнопки. Средствами платформы Metro реализовать такую возможность проще некуда.

Достаточно вызвать метод `requestAppPurchaseAsync` знакомого нам объекта `Windows.ApplicationModel.Store.CurrentAppSimulator`. Этот метод не принимает параметров и возвращает обязательство, для которого мы в вызове метода `then` укажем две функции. Первая функция выполнится после успешного завершения покупки приложения, а вторая — если в процессе покупки возникла какая-либо ошибка.

При вызове метода `requestAppPurchaseAsync` платформа Metro выведет на экран небольшое диалоговое окно (рис. 26.1). Чтобы подтвердить факт покупки приложения, мы нажмем в этом окне кнопку **Continue**. (Кнопка **Cancel** позволит прервать процесс покупки; ее нажатие платформа Metro сочтет за возникновение ошибки.)

```
Windows.ApplicationModel.Store.CurrentAppSimulator.
↳requestAppPurchaseAsync().then(function() {
  //Приложение было успешно куплено
  //Сообщить об этом пользователю
}, function() {
  //В процессе покупки приложения возникла проблема
});
```

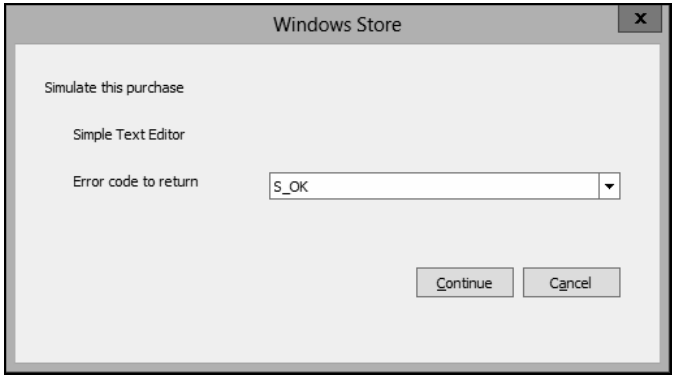


Рис. 26.1. Диалоговое окно, подтверждающее покупку приложения

Отслеживание изменений в лицензии

В процессе работы приложения нам так или иначе придется отслеживать изменения, произошедшие в его лицензии. Например, если тестовый период приложения закончился, нам придется как-то на это реагировать. Или если пользователь успешно купил приложение, нам потребуется перевести его в полнофункциональный режим работы. Как это сделать?

Как только платформа Metro получает от магазина приложений Windows Store сигнал об изменениях в лицензии, в представляющем его экземпляре объекта `Windows.ApplicationModel.Store.LicenseInformation` возникает событие `licensechanged`. Обработчик этого события можно использовать для выяснения, что произошло с лицензией нашего приложения.

```
var oLI = Windows.ApplicationModel.Store.CurrentAppSimulator.
↳licenseInformation;
oLI.addEventListener("licensechanged", function() {
  if (oLI.isActive) {
    if (oLI.isTrial) {
      //Тестовый период еще не закончился
    } else {
      //Приложение было куплено
    }
  } else {
    //Тестовый период закончился
  }
});
```

Реализация покупки отдельных функций приложения

Что ж, создавать условно-бесплатные приложения мы научились. Теперь узнаем, как реализовать в приложении покупку отдельных частей его функциональности.

Сначала нам предстоит выделить в приложении все функции, за которые мы хотим брать деньги, и дать каждой из них идентификатор. Этот идентификатор должен представлять собой строку, содержащую буквы латинского алфавита и цифры, быть уникальным в пределах данного приложения и, по возможности, вразумительным.

Далее мы укажем в составе информации о приложении, что записывается в базе данных магазина Windows Store, лицензию для каждой из его платных функций. Эта лицензия включит идентификатор соответствующей платной функции, а впоследствии — признак, была ли данная функция приобретена пользователем.

В коде самого приложения мы сначала получим лицензию на каждую из платных функций. Для этого мы обратимся к свойству `productLicenses` лицензии самого приложения. Оно хранит экземпляр объекта-коллекции, представляющей все присутствующие в приложении платные функции.

```
var oLI = Windows.ApplicationModel.Store.CurrentAppSimulator.
↳licenseInformation;
var oPLs = oLI.productLicenses;
```

Коллекция, что мы получим из свойства `productLicenses`, поддерживает знакомый нам по *главе 23* метод `lookup`. Его мы можем использовать для извлечения лицензии на платную функцию с определенным идентификатором.

```
var oPL = oPLs.lookup("showVideoInfo");
```

Здесь мы получаем лицензию на платную функцию с идентификатором `showVideoInfo`.

Лицензия на платную функцию представляется экземпляром объекта `Windows.ApplicationModel.Store.ProductLicense`. Свойство `isActive` этого объекта возвращает `true`, если пользователь приобрел данную функцию и, следовательно, имеет к ней доступ, и `false` в противном случае.

```
if (oPL.isActive) {
  //Данная платная функция приобретена пользователем и,
  //следовательно, доступна.
  //Выводим на экран необходимые элементы интерфейса и задействуем
  //их функциональность
} else {
  //Данная платная функция еще не приобретена пользователем и,
  //следовательно, не доступна.
  //Выводим на экран элементы интерфейса, предлагающие приобрести
  //эту функцию
}
```

Чтобы получить из магазина Windows Store стоимость платной функции, мы сначала получим оттуда сведения о стоимости самого приложения. Объект `Windows.ApplicationModel.Store.ListingInformation`, представляющий эти сведения, поддерживает свойство `productListings`, которое хранит экземпляр объекта-коллекции, содержащего сведения о стоимости отдельных платных функций. А эти сведения, в свою очередь, представляются экземплярами объекта `Windows.ApplicationModel.Store.ProductListing`, который поддерживает знакомые нам свойства `formattedPrice` и `name`.

```
oLI.loadListingInformationAsync().then(function(oListI) {
  var oPList = oListI.productListings.lookup("showVideoInfo");
  var sPrice = oPList.formattedPrice;
  . . .
});
```

Получаем стоимость платной функции с идентификатором `showVideoInfo`.

Покупка платной функции выполняется вызовом метода `requestProductPurchaseAsync` объекта `Windows.ApplicationModel.Store.CurrentAppSimulator`. Данный метод принимает в качестве единственного параметра строку с идентификатором приобретаемой функции и возвращает обязательство, для которого в вызове метода `then` следует указать две функции. Первая функция выполнится после успешного

завершения покупки функции, а вторая — если в процессе покупки возникла ошибка.

```
Windows.ApplicationModel.Store.CurrentAppSimulator.
requestProductPurchaseAsync("showVideoInfo").then(function() {
  //Платная функция была успешно куплена
}, function() {
  //В процессе покупки функции возникла проблема
});
```

Тестирование условно-бесплатных приложений

Любое приложение в процессе разработки должно тестироваться, и не раз. Это в полной мере относится и к условно-бесплатным приложениям, и приложениям с платными функциями.

Но как нам протестировать ту часть их кода, что "отвечает" за ограничение функциональности приложения, работающего в тестовом режиме, и его покупку? Ведь для этого нужно опубликовать приложение в Windows Store, а ведь, вполне возможно, мы еще даже не зарегистрированы в этом магазине!

Об этом позаботилась сама платформа Metro. Если мы используем для управления условно-бесплатным приложением "тестовый" объект `Windows.ApplicationModel.Store.CurrentAppSimulator`, фрагмент базы данных, хранящий описание лицензий данного приложения и его платных функций, будет сформирован в оперативной памяти нашего компьютера. А данные для заполнения этой базы будут загружены из особого файла, который мы создадим.

Это обычный текстовый файл, сохраненный в кодировке UTF-8. Его содержимым станет код лицензии, записанный на языке XML (который кратко описывался в *главе 21*). Он будет располагаться в папке LocalState\Microsoft\Windows Store\ApiData, вложенной в папку, где находятся рабочие данные Metro-приложения, и иметь имя WindowsStoreProxy.xml.

Что касается рабочих данных приложения, то они хранятся в папке *<папка пользовательского профиля>*\AppData\Local\Packages*<наименование пакета приложения>*. Наименование пакета приложения можно найти в текстовом поле **Package Family Name**, находящемся на вкладке **Packaging** (рис. 26.2) окна параметров приложения.

Создать файл с описанием лицензии можно в любом текстовом редакторе, поддерживающем кодировку UTF-8. В качестве примера такого редактора можно привести Блокнот, поставляемый в составе всех версий Windows.

Структура тегов, формирующих содержимое файла WindowsStoreProxy.xml, такова (отступы обозначают вложенность тегов друг в друга):

```
<?xml version="1.0" encoding="utf-8" ?>
<CurrentApp>
  <ListingInformation>
```

```
    <App>
      <AppId>
      <LinkUri>
      <AgeRating>
      <CurrentMarket>
      <MarketData>
        <Name>
        <Description>
        <Price>
        <CurrencySymbol>
    [<Product>
      <MarketData>
        <Name>
        <Price>
        <CurrencySymbol>]
  <LicenseInformation>
    <App>
      <IsActive>
      <IsTrial>
      [<ExpirationDate>]
    <Product
      <IsActive>
```

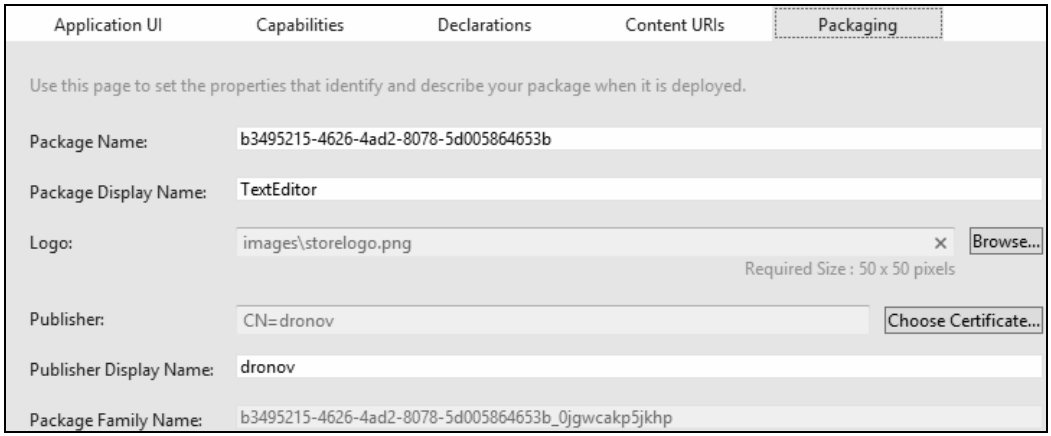


Рис. 26.2. Вкладка **Packaging** окна параметров приложения

Тег `<?xml version="1.0" encoding="utf-8" ?>` обязательно должен присутствовать в самом начале содержимого данного файла. Он задает версию языка XML, применяемую для форматирования данных, и кодировку.

Тег `<CurrentApp>` описывает саму лицензию и также должен присутствовать в обязательном порядке.

Тег `<ListingInformation>`, вложенный в тег `<CurrentApp>`, описывает сведения о самом приложении: его название, описание и цены для различных рынков. В нем также записываются сведения о платных функциях данного приложения.

В тег `<ListingInformation>` вложен тег `<App>`, содержащий сведения о самом приложении. Для форматирования этих сведений применяются следующие теги:

- `<AppId>` — уникальный идентификатор (GUID) приложения. Он формируется самим Visual Studio при создании проекта и выводится в поле ввода **Package Name** вкладки **Packaging** окна параметров приложения (см. рис. 26.2);
- `<LinkUri>` — интернет-адрес раздела магазина Windows Store, где представлено приложение. Для тестовых целей мы используем интернет-адрес, указанный в формате **http://apps.windows.microsoft.com/app/*<GUID приложения, заданный в теге <AppId>>***;
- `<AgeRating>` — возрастной рейтинг приложения. Его значение должно представлять собой целое число, обозначающее минимальный возраст пользователя: 3, 7, 12 или 16. В тестовых целях мы можем указать любой рейтинг;
- `<CurrentMarket>` — текущий рынок приложения. Его значением должно быть наименование языка операционной системы, установленной на нашем компьютере. Наименования всех языков, поддерживаемых платформой Metro, можно найти по интернет-адресу **http://msdn.microsoft.com/en-us/library/ms533052(v=vs.85).aspx**. Так, русскому языку соответствует наименование ru, а американскому английскому — en-us;
- `<MarketData>` — наименование, описание и стоимость приложения для определенного рынка. Обозначение языка, соответствующего этому рынку, указывается в атрибуте `xml:land` данного тега.

  Тег `<MarketData>` содержит следующие вложенные теги:

  - `<Name>` — название приложения;
  - `<Description>` — описание приложения;
  - `<Price>` — стоимость приложения без обозначения валюты;
  - `<CurrencySymbol>` — обозначение валюты.

  Для каждого рынка, на котором предполагается распространять приложение, создается свой тег `<MarketData>`.

Тег `<Product>`, также вложенный в тег `<ListingInformation>`, описывает одну из платных функций, реализованных в приложении. Этот тег является необязательным; если в приложении нет платных функций, создавать его не следует. В атрибуте `ProductId` этого тега задается идентификатор соответствующей платной функции.

Тег `<Product>` должен содержать набор уже знакомых нам тегов `<MarketData>`, задающих наименование и стоимость платной функции для определенного рынка. Тег `<MarketData>`, в свою очередь, должен содержать теги `<Name>`, `<Price>` и `<CurrencySymbol>`, описанные ранее.

Тег `<LicenseInformation>`, вложенный в тег `<CurrentApp>`, описывает сведения о состоянии лицензии приложения: активна ли лицензия, работает ли приложение в тестовом режиме и сколько ему еще осталось так работать.

Тег <App>, вложенный в тег <LicenseInformation>, описывает состояние лицензии самого приложения. В него вложены следующие теги:

- <IsActive> — указывает, активна ли лицензия данного приложения. Содержимое true обозначает, что лицензия приложения активна (т. е. либо тестовый период еще не закончился, либо пользователь купил приложение). А содержимое false обозначает неактивность лицензии;
- <IsTrial> — указывает, работает ли приложение в тестовом режиме. Содержимое true обозначает, что приложение работает в тестовом режиме, а содержимое false — в полнофункциональном;
- <ExpirationDate> — задает дату и время окончания тестового периода. Его содержимое указывается в формате *<год>-<номер месяца>-<число>*T*<часы>*:*<минуты>*:*<секунды>*.*<сотые доли секунд>*Z.

  Дата окончания тестового периода является обязательной к указанию только в случае, если приложение работает в тестовом режиме (тег <IsTrial> имеет содержимое true). В противном случае тег <ExpirationDate> будет проигнорирован.

А тег <Product>, вложенный в тег <LicenseInformation>, описывает сведения о состоянии лицензии одной из платных функций; ее идентификатор указывается в атрибуте ProductId данного тега. Внутри тега <Product> должен находиться уже знакомый нам тег <IsActive>.

В качестве примера давайте рассмотрим вот такой XML-код:

```
<?xml version="1.0" encoding="utf-8" ?>
<CurrentApp>
  <ListingInformation>
    <App>
      <AppId>2B14D306-D8F8-4066-A45B-0FB3464C67F2</AppId>
      <LinkUri>http://apps.microsoft.com/app/2B14D306-D8F8-4066-A45B-
      ↳0FB3464C67F2</LinkUri>
      <CurrentMarket>ru</CurrentMarket>
      <AgeRating>3</AgeRating>
      <MarketData xml:lang="ru">
        <Name>Видеопроигрыватель</Name>
        <Description>Приложение простейшего
        видеопроигрывателя</Description>
        <Price>30</Price>
        <CurrencySymbol>руб.</CurrencySymbol>
      </MarketData>
      <MarketData xml:lang="en-us">
        <Name>Video Player</Name>
        <Description>Simple video playing application</Description>
        <Price>1.49</Price>
        <CurrencySymbol>$</CurrencySymbol>
      </MarketData>
    </App>
```

```
    <Product ProductId="showVideoInfo">
      <MarketData xml:lang="ru">
        <Name>Вывод параметров видеофайла</Name>
        <Price>30</Price>
        <CurrencySymbol>руб.</CurrencySymbol>
      </MarketData>
      <MarketData xml:lang="en-us">
        <Name>Show video file properties</Name>
        <Price>1.49</Price>
        <CurrencySymbol>$</CurrencySymbol>
      </MarketData>
    </Product>
    <Product ProductId="captureScreenshot">
      <MarketData xml:lang="ru">
        <Name>Снятие скриншотов</Name>
        <Price>30</Price>
        <CurrencySymbol>руб.</CurrencySymbol>
      </MarketData>
      <MarketData xml:lang="en-us">
        <Name>Screenshots capturing</Name>
        <Price>1.49</Price>
        <CurrencySymbol>$</CurrencySymbol>
      </MarketData>
    </Product>
  </ListingInformation>
  <LicenseInformation>
    <App>
      <IsActive>true</IsActive>
      <IsTrial>true</IsTrial>
      <ExpirationDate>2012-04-10T12:10:00.00Z</ExpirationDate>
    </App>
    <Product ProductId="showVideoInfo">
      <IsActive>false</IsActive>
    </Product>
    <Product ProductId="captureScreenshot">
      <IsActive>false</IsActive>
    </Product>
  </LicenseInformation>
</CurrentApp>
```

Он описывает лицензию для приложения простейшего видеопроигрывателя. Для российского рынка его стоимость составит 30 руб., для рынка США — 1,49 долл. В данный момент это приложение работает в тестовом режиме, который закончится 10 апреля 2012 года в 12:10:00. Помимо этого, приложение реализует две платные функции; обе в данный момент неактивны.

И еще пара замечаний по поводу тестирования условно-бесплатных приложений.

Как уже говорилось, хранящаяся в памяти компьютера тестовая лицензия формируется на основе содержимого файла WindowsStoreProxy.xml. Однако все изменения, что мы сделаем в этой лицензии в процессе испытания приложения (например, имитация его покупки), не будут сохранены в файле. Так что при следующем запуске приложения мы получим в памяти изначальную копию лицензии.

Чтобы испытать реакцию приложения на изменения, сделанные в лицензии (окончание тестового периода, покупка приложения или платной функции и др.), мы сделаем соответствующие правки в файле WindowsStoreProxy.xml. Разумеется, перед этим мы должны завершить приложение, а впоследствии — вновь запустить его.

Пример 1: условно-бесплатное приложение текстового редактора

Давайте для примера превратим наше приложение простейшего текстового редактора в условно-бесплатное. По истечении тестового периода оно будет выводить всплывающее уведомление, предлагающее пользователю произвести оплату.

Откроем в Visual Studio проект `TextEditor`. Сразу же откроем окно параметров приложения и дадим ему возможность вывода всплывающих уведомлений. (Как это сделать, описывалось в *главе 21*.)

Откроем файл default.html, отыщем код, формирующий панель инструментов, и вставим в него следующий фрагмент (выделен полужирным шрифтом):

```
<button data-win-control="WinJS.UI.AppBarCommand"
data-win-options="{id: 'btnSave', label: 'Сохранить', icon: 'savelocal',
↳section: 'global'}"></button>
<hr data-win-control="WinJS.UI.AppBarCommand"
data-win-options="{id: 'dvd1', section: 'global', type: 'separator'}" />
<button data-win-control="WinJS.UI.AppBarCommand"
data-win-options="{id: 'btnBuy', label: 'Купить',
↳icon: 'shop', section: 'global', type: 'flyout',
↳flyout: 'divBuy'}"></button>
```

Мы добавили разделитель и кнопку **Купить**.

Вставим перед кодом, создающим панель инструментов, следующий фрагмент:

```
<div id="divBuy" data-win-control="WinJS.UI.Flyout">
  <p>Стоимость: <span id="spnPrice"></span></p>
  <p><input type="button" id="btnPay" value="Выполнить оплату" /></p>
</div>
```

Он сформирует всплывающий элемент, который будет выводиться на экран после нажатия кнопки **Купить** панели инструментов. Этот всплывающий элемент получит в качестве содержимого абзац, где будет выводиться стоимость приложения, и кнопку **Выполнить оплату**, собственно запускающую операцию покупки.

Переключимся на файл default.js и введем код, объявляющий необходимые переменные:

```
var spnPrice;
var oLI = Windows.ApplicationModel.Store.CurrentAppSimulator.
↳licenseInformation;
var toastTemplate =
Windows.UI.Notifications.ToastTemplateType.toastText01;
var oTN =
Windows.UI.Notifications.ToastNotificationManager.createToastNotifier();
```

Здесь мы также получаем лицензию и диспетчера всплывающих уведомлений.

Отыщем код инициализации и добавим в его конец следующие выражения (выделены полужирным шрифтом):

```
document.addEventListener("DOMContentLoaded", function() {
  WinJS.UI.processAll().then(function() {
    . . .
    var ctrBuy = document.getElementById("divBuy").winControl;
    spnPrice = document.getElementById("spnPrice");
    var btnPay = document.getElementById("btnPay");
    ctrBuy.addEventListener("beforeshow", ctrBuyBeforeShow);
    btnPay.addEventListener("click", btnPayClick);
    oLI.addEventListener("licensechanged", oLILicenseChanged);
    oLILicenseChanged();
  });
});
```

Здесь мы также привязываем обработчик к событию `licensechanged` лицензии и сразу же вызываем этот обработчик, чтобы задать состояние приложения уже при его запуске.

Сразу же напишем функцию-обработчик события `licensechanged` лицензии.

```
function oLILicenseChanged() {
  if (oLI.isActive) {
    txtEdit.readOnly = false;
    if (oLI.isTrial) {
      var iDaysRemaining = (oLI.expirationDate — new Date()) / 86400000;
      showToast("До окончания тестового периода осталось " +
      iDaysRemaining + " дней");
    } else {
      document.getElementById("divAppBar").winControl.
      ↳hideCommands(["dvd1", "btnBuy"]);
    }
  } else {
    txtEdit.readOnly = true;
    showToast("Тестовый период работы приложения закончился.
    ↳Приобретите приложение, если хотите продолжать его использовать.");
  }
}
```

- Если лицензия приложения активна, делаем область редактирования, где выполняется работа с текстом, доступной для ввода.
  - Если приложение не было оплачено, и тестовый период его работы не завершился, выводим всплывающее уведомление с количеством дней, отведенных пользователю на тестирование.
  - Если приложение было оплачено, скрываем кнопку **Купить** панели инструментов.
- Если лицензия приложения неактивна (т. е. тестовый период закончился, а пользователь не оплатил приложение), делаем область редактирования недоступной для ввода и выводим соответствующее всплывающее уведомление.

Напишем функцию, используемую для вывода всплывающих уведомлений.

```
function showToast(sText) {
  var oToastTemplate = Windows.UI.Notifications.
  ↳ToastNotificationManager.getTemplateContent(toastTemplate);
  var txt1 = oToastTemplate.getElementsByTagName("text")[0];
  txt1.innerText = sText;
  var aud1 = oToastTemplate.createElement("audio");
  aud1.setAttribute("src", "Notification.Default");
  aud1.setAttribute("loop", "false");
  var tst1 = oToastTemplate.getElementsByTagName("toast")[0];
  tst1.appendChild(aud1);
  var oT = new
  Windows.UI.Notifications.ToastNotification(oToastTemplate);
  oT.addEventListener("activated", btnPayClick);
  oTN.show(oT);
}
```

Отметим, что в качестве функции, которая будет выполнена при нажатии на всплывающее уведомление, мы укажем ту же функцию, что выполняется после нажатия на кнопку **Выполнить оплату** всплывающего элемента. Мы объявим эту функцию потом.

Напишем обработчик события `beforeshow` всплывающего элемента, предлагающего оплатить приложение:

```
function ctrBuyBeforeShow() {
  Windows.ApplicationModel.Store.CurrentAppSimulator.
  ↳loadListingInformationAsync().then(function(oListI) {
    spnPrice.textContent = oListI.formattedPrice;
  });
}
```

Он просто выведет стоимость приложения.

Осталось только создать функцию, которая выполнится после нажатия кнопки **Выполнить оплату** всплывающего элемента или всплывающего уведомления.

```
function btnPayClick() {
  Windows.ApplicationModel.Store.CurrentAppSimulator.
  ↳requestAppPurchaseAsync().then(function() {
    showToast("Поздравляем: вы купили приложение.");
  }, function() {
    showToast("В процессе оплаты приложения возникла ошибка.");
  });
}
```

Здесь мы выполняем приобретение приложения и, в зависимости от успеха или неудачи, выводим соответствующее всплывающее уведомление.

Сохраним все исправленные файлы.

Откроем папку *<папка пользовательского профиля>*\AppData\Local\Packages и отыщем там папку, чье имя совпадает с наименованием пакета (как его выяснить, мы уже знаем). В ней создадим папку LocalState\Microsoft\Windows Store\ApiData и поместим туда файл WindowsStoreProxy.xml следующего содержания, сохраненный в кодировке UTF-8:

```
<?xml version="1.0" encoding="utf-8" ?>
<CurrentApp>
  <ListingInformation>
    <App>
      <AppId><GUID приложения></AppId>
      <LinkUri>http://apps.microsoft.com/app/<GUID приложения></LinkUri>
      <CurrentMarket>en-us</CurrentMarket>
      <AgeRating>3</AgeRating>
      <MarketData xml:lang="ru">
        <Name>Текстовый редактор</Name>
        <Description>Приложение простейшего текстового
        редактора</Description>
        <Price>30</Price>
        <CurrencySymbol>руб.</CurrencySymbol>
      </MarketData>
      <MarketData xml:lang="en-us">
        <Name>Simple Text Editor</Name>
        <Description>Simple text editing application</Description>
        <Price>1.49</Price>
        <CurrencySymbol>$</CurrencySymbol>
      </MarketData>
    </App>
  </ListingInformation>
  <LicenseInformation>
    <App>
      <IsActive>true</IsActive>
      <IsTrial>true</IsTrial>
      <ExpirationDate><какая-либо будущая дата></ExpirationDate>
    </App>
```

```
  </LicenseInformation>
</CurrentApp>
```

Автор тестировал данное приложение в английской редакции Windows 8, поэтому установил в качестве содержимого тега `<CurrentMarket>` значение `en-us`. Пользователям русской редакции следует указать для этого тега содержимое `ru`.

Запустим приложение на выполнение. Проверим, появляется ли на экране всплывающее уведомление, выводящее количество дней, что остались до конца тестового периода. Нажмем на этом уведомлении, чтобы имитировать оплату приложения, и посмотрим, что случится. После этого завершим приложение.

Попробуем указать в теге `<ExpirationDate>` файла WindowsStoreProxy.xml другую дату, на этот раз — прошедшую. Снова запустим приложение. На экране должно появиться всплывающее уведомление с сообщением об истечении тестового периода, а область редактирования, где выводится текст, должна стать недоступной для ввода.

Пример 2: приложение видеопроигрывателя с платной функцией

Второе практическое занятие мы посвятим приложению видеопроигрывателя. Мы превратим поддерживаемую им функцию вывода сведений об открытом видеофайле в платную.

Откроем в Visual Studio проект `VideoPlayer`. Переключимся на файл default.html и вставим куда-либо в тег `<body>` следующий код:

```
<div id="divBuy" data-win-control="WinJS.UI.Flyout">
  <p>Стоимость: <span id="spnPrice"></span></p>
  <p><input type="button" id="btnPay" value="Выполнить оплату" /></p>
</div>
```

Он сформирует всплывающий элемент, аналогичный рассмотренному нами ранее.

Переключимся на файл default.js и сразу же введем код, объявляющий необходимые переменные:

```
var ctrBuy, spnPrice;
var oLI = Windows.ApplicationModel.Store.CurrentAppSimulator.
↳licenseInformation;
```

Найдем код инициализации и добавим в него следующие выражения (выделены полужирным шрифтом):

```
document.addEventListener("DOMContentLoaded", function() {
  WinJS.UI.processAll().then(function() {
    . . .
    ctrBuy = document.getElementById("divBuy").winControl;
    spnPrice = document.getElementById("spnPrice");
    var btnPay = document.getElementById("btnPay");
```

```
    ctrBuy.addEventListener("beforeshow", ctrBuyBeforeShow);
    btnPay.addEventListener("click", btnPayClick);
  });
});
```

Отыщем объявление функции `divTimingClick()`, выполняющейся после нажатия на блок хронометража, и изменим его следующим образом:

```
function divTimingClick() {
  if (vidMain.videoWidth > 0) {
    if (oLI.productLicenses.lookup("showVideoInfo").isActive) {
      ctrInfo.show(divTiming, "bottom", "right");
    } else {
      ctrBuy.show(divTiming, "bottom", "right");
    }
  }
}
```

Здесь мы проверяем, приобрел ли пользователь функциональность вывода сведений о видеофайле (она имеет идентификатор `showVideoInfo`). Если это так, мы выводим данные сведения на экран; в противном случае показываем всплывающий элемент, предлагающий купить эту функцию.

Напишем обработчик события `beforeshow` всплывающего элемента, предлагающего оплатить приложение:

```
function ctrBuyBeforeShow() {
  Windows.ApplicationModel.Store.CurrentAppSimulator.
  ↳loadListingInformationAsync().then(function(oListI) {
    spnPrice.textContent = oListI.
    ↳productListings.lookup("showVideoInfo").formattedPrice;
  });
}
```

Он выводит стоимость платной функции `showVideoInfo()`.

Объявим функцию, которая выполнится после нажатия кнопки **Выполнить оплату** всплывающего элемента.

```
function btnPayClick() {
  Windows.ApplicationModel.Store.CurrentAppSimulator.
  ↳requestProductPurchaseAsync("showVideoInfo").then(function() {
    showToast("Поздравляем: вы купили функцию.");
  }, function() {
    showToast("В процессе оплаты функции возникла ошибка.");
  });
}
```

Осталось только позаимствовать из кода приложения видеопроигрывателя *(см. ранее)* функцию `showToast()`, которая будет выводить всплывающие уведомления.

Сохраним все исправленные файлы.

В папке *<папка пользовательского профиля>*\AppData\Local\Packages найдем папку, чье имя совпадает с наименованием пакета нашего приложения. Создадим там папку LocalState\Microsoft\Windows Store\ApiData и поместим в нее файл WindowsStoreProxy.xml, сохраненный в кодировке UTF-8. Содержимое этого файла будет таким:

```
<?xml version="1.0" encoding="utf-8" ?>
<CurrentApp>
  <ListingInformation>
    <App>
      <AppId><GUID приложения></AppId>
      <LinkUri>http://apps.microsoft.com/app/<GUID приложения></LinkUri>
      <CurrentMarket>en-us</CurrentMarket>
      <AgeRating>3</AgeRating>
      <MarketData xml:lang="ru">
        <Name>Видеопроигрыватель</Name>
        <Description>Приложение простейшего
        видеопроигрывателя</Description>
        <Price>30</Price>
        <CurrencySymbol>руб.</CurrencySymbol>
      </MarketData>
      <MarketData xml:lang="en-us">
        <Name>Video Player</Name>
        <Description>Simple video playing application</Description>
        <Price>1.49</Price>
        <CurrencySymbol>$</CurrencySymbol>
      </MarketData>
    </App>
    <Product ProductId="showVideoInfo">
      <MarketData xml:lang="ru">
        <Name>Вывод параметров видеофайла</Name>
        <Price>30</Price>
        <CurrencySymbol>руб.</CurrencySymbol>
      </MarketData>
      <MarketData xml:lang="en-us">
        <Name>Show video file properties</Name>
        <Price>1.49</Price>
        <CurrencySymbol>$</CurrencySymbol>
      </MarketData>
    </Product>
  </ListingInformation>
  <LicenseInformation>
    <App>
      <IsActive>true</IsActive>
      <IsTrial>false</IsTrial>
    </App>
```

```
    <Product ProductId="showVideoInfo">
      <IsActive>false</IsActive>
    </Product>
  </LicenseInformation>
</CurrentApp>
```

Здесь мы указали, что лицензия приложения активна, приложение изначально работает в полнофункциональном режиме, а его платная функция изначально неактивна. Тем самым мы создали изначально бесплатное приложение с платной функцией.

Запустим приложение на выполнение. Откроем какой-либо видеофайл и попытаемся выяснить его параметры, нажав на блок хронометража. На экране должен появиться всплывающий элемент, предлагающий купить платную функцию. Выполним его покупку и снова нажмем на блок хронометража; на экране должен появиться всплывающий элемент с параметрами видеофайла.

Перевод приложения из тестового в рабочий режим

При разработке условно-бесплатных приложений и приложений с платными функциями мы использовали для отладки тестовые средства, а именно объект `Windows.ApplicationModel.Store.CurrentAppSimulator`. Благодаря этому мы получили возможность проверить функциональность, отслеживающую состояние лицензии и выполняющую оплату, не публикуя приложение в магазине Windows Store (и не тратя деньги на его покупку).

Но в процессе подготовки приложения к публикации в Windows Store нам придется реализовать в нем реальные механизмы отслеживания состояния лицензии и покупки. Как это сделать?

Очень просто. Достаточно заменить в коде логики объект `Windows.ApplicationModel.Store.CurrentAppSimulator` объектом `Windows.ApplicationModel.Store.CurrentApp`. Никаких других правок в коде выполнять не нужно.

```
Windows.ApplicationModel.Store.CurrentApp.
↳requestAppPurchaseAsync().then(function() {
  //Приложение было успешно куплено
}, function() {
  //В процессе покупки приложения возникла проблема
});
```

> ***Внимание!***
>
> Приложения, использующие вызовы объекта `Windows.ApplicationModel.Store.CurrentAppSimulator`, не будут опубликованы в Windows Store в любом случае.

Что дальше?

В этой главе мы учились создавать условно-бесплатные приложения и приложения, включающие в свой состав платные функции. Мы узнали, как реализовать получение состояния лицензии такого приложения, как сообщить пользователю об окончании тестового периода и как выполнить покупку приложения прямо из его интерфейса. Еще мы познакомились с тестовыми инструментами платформы Metro, позволяющими проверить работу всех этих механизмов.

Следующая, заключительная, глава будет посвящена последним этапам подготовки приложения к публикации в магазине Windows Store и собственно его публикации. Другими словами, мы наконец-то дадим нашим приложениям путевку в жизнь!

ГЛАВА 27

Распространение Metro-приложений

В предыдущей главе мы занимались созданием коммерческих Metro-приложений, в частности условно-бесплатных и имеющих платные функции. А платформа Metro предоставила нам для этого все необходимые инструменты.

Последняя глава данной книги будет посвящена распространению приложений. Мы узнаем, какие требования предъявляются к приложениям — кандидатам на публикацию в магазине Windows Store, как подготовить приложение к распространению, как провести его окончательное тестирование, какие сведения о приложении следует предоставить при его публикации и, наконец, как его выполнить. И тем самым завершим долгий рассказ о Metro-программировании...

Требования к публикуемым приложениям

Начнем с перечисления требований, которым должно отвечать приложение, предназначенное для публикации в магазине Windows Store.

- Приложение должно представлять собой окончательную редакцию. Отладочные редакции приложений не будут приниматься к публикации ни под каким видом. (Об отладочных и окончательных редакциях приложения будет рассказано далее.)
- Приложение должно иметь уникальное имя и описание, исчерпывающе его характеризующие и максимально полно описывающие выполняемые им функции. Также приложение должно иметь уникальный логотип и изображение заставки, отличные от установленных по умолчанию (белая звезда на черном фоне) самим Visual Studio.
- Приложение должно качественно реализовывать все заявленные функции.
- Приложение должно поддерживать, по крайней мере, один язык из списка, приведенного на Web-странице **http://msdn.microsoft.com/en-us/library/windows/apps/hh694075.aspx#app_langs**. В частности, там указаны русский и английский языки.

- Приложение не должно содержать критических ошибок, могущих привести к аварийному завершению его работы.
- Приложение не должно содержать вредоносного кода и каким-либо образом компрометировать пользователя.
- Приложение не должно включать кода, требующегося исключительно для отладки. К такому коду относятся, например, вызовы объекта `Windows.ApplicationModel.Store.CurrentAppSimulator`, применяемого для тестирования условно-бесплатных приложений (подробности *см. в главе 26*).
- Приложение не должно иметь избыточных прав, не требующихся для его успешного функционирования.
- Дистрибутивный комплект приложения должен включать все необходимые файлы и данные и не должен содержать ничего лишнего.

Невыполнение любого из этих требований может повлечь за собой отказ в публикации приложения. Так что будем внимательны!

Задание общих параметров приложения

Перед публикацией приложения нам следует задать для него общие параметры. Они включают в себя название приложения, его описание, различные логотипы, которые будут выводиться на плитке, на всплывающих уведомлениях и пр., и некоторые другие данные.

Общие параметры приложения указываются на вкладке **Application UI** (рис. 27.1) окна параметров приложения. Откроем данную вкладку и посмотрим, какие элементы управления, находящиеся на ней, нам понадобятся.

Поле ввода **Display Name** служит для указания названия приложения, которое будет выводиться в диспетчере задач и экране поиска. Его длина не должна превышать 255 символов.

Область редактирования **Description** задает развернутое описание приложения, которое будет выводиться во всплывающей подсказке к соответствующей приложению плитке меню **Пуск**. Его длина не должна превышать 2048 символов.

Группа флажков **Supported Initial Rotations** позволяет выбрать положения устройства и режимы работы, являющиеся предпочтительными для работы с приложением. (Подробнее об этом говорилось в *главе 25*.) Всего в этой группе присутствуют четыре флажка:

- **Landscape** — ландшафтная ориентация и полноэкранный режим;
- **Portrait** — портретная ориентация и полноэкранный режим;
- **Landscape-Flipped** — ландшафтная ориентация и прикрепленный режим;
- **Portrait-Flipped** — портретная ориентация и прикрепленный режим.

Установка какого-либо флажка говорит платформе Metro, что данное приложение полностью функционально в соответствующих ориентации и режиме. Напротив,

сброшенный флажок означает, что приложение хоть и может работать в данных ориентации и режиме, однако они не поддерживаются им в полной мере. Изначально ни один флажок не установлен — это значит, что приложение может работать во всех ориентациях и режимах.

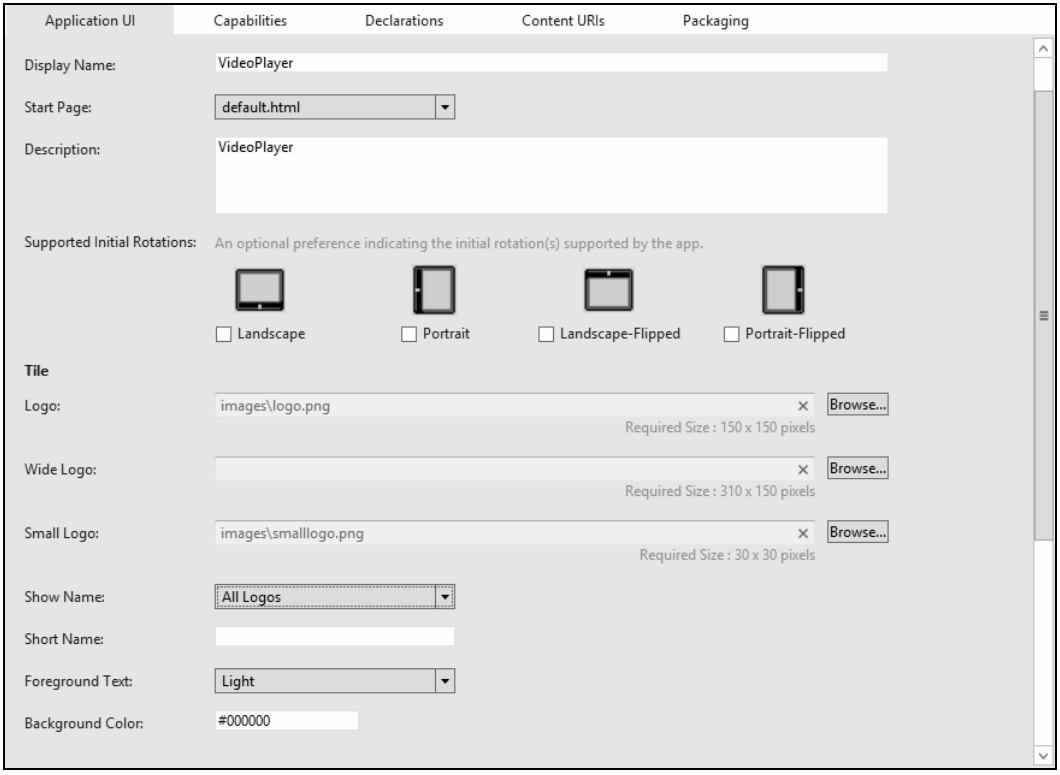


Рис. 27.1. Вкладка **Application UI** окна параметров приложения

Группа элементов управления **Tile** позволяет указать файлы с графическими изображениями, которые будут выводиться на плитках в качестве логотипов приложения. Все элементы управления, находящиеся в этой группе, описаны далее.

В поле ввода **Logo** указывается ссылка на файл с логотипом приложения, который будет отображаться на квадратной плитке. Чтобы не вводить его вручную, мы можем нажать расположенную правее этого поля ввода кнопку **Browse** и выбрать нужный файл в появившемся на экране стандартном диалоговом окне открытия файла.

Файл, хранящий логотип, должен быть включен в состав проекта. Размеры квадратного логотипа — 150×150 пикселов.

Изначально в качестве квадратного логотипа используется файл logo.png из папки images. Данный файл создается самим Visual Studio при создании проекта и содержит изображение по умолчанию — белая звезда на черном фоне.

В поле ввода **Wide Logo** указывается ссылка на файл с логотипом, который будет отображаться на широкой плитке. Указывается только в том случае, если мы хотим создать для приложения широкую плитку. Изначально не указан.

Размеры широкого логотипа — 300×150 пикселов. Хранящий его файл также должен находиться в составе проекта.

Поле ввода **Small Logo** указывает ссылку на файл с маленьким, размерами 30×30 пикселов, логотипом, который отображается в левом нижнем углу плитки в случае, если на ней присутствует информация, выводимая приложением (подробности *см. в главе 21*). Изначально там указан файл smalllogo.png из папки images, созданный Visual Studio и хранящий логотип по умолчанию.

Раскрывающийся список **Show Name** позволяет указать, будет ли на плитке приложения вместо его логотипа выводиться короткое название (мы зададим его чуть позже). Здесь присутствуют четыре пункта:

- **All Logos** — на всех плитках будет выводиться короткое название приложения;
- **No Logos** — на всех плитках будет выводиться логотип приложения;
- **Logo Only** — на квадратной плитке будет выводиться короткое название, а на широкой — логотип приложения;
- **Wide Logo Only** — на квадратной плитке будет выводиться логотип, а на широкой — короткое название приложения.

Что касается короткого названия приложения, то оно указывается в поле ввода **Short Name**. Его длина не должна превышать 13 символов.

Раскрывающийся список **Foreground Text** позволяет задать интенсивность цвета, которым будет выводиться на плитке короткое название приложения. Пункт **Light** устанавливает высокую интенсивность цвета, а пункт **Dark** — низкую.

Поле ввода **Background Color** задает цвет фона плитки. Мы можем указать как значение цвета в формате RGB, так и именованный цвет.

Группа элементов управления **Splash Screen** позволяет указать параметры заставки, которая появляется в самом начале загрузки Metro-приложения (мы наблюдали эти заставки, начиная с *главы 2*). Здесь присутствуют всего два элемента управления.

Поле ввода **Splash Screen** указывает ссылку на файл с изображением заставки. Это изображение должно иметь следующие размеры в пикселах:

- 620×300 — для экранов с плотностью 96 пиксел/дюйм;
- 868×420 — для экранов с плотностью 148 пиксел/дюйм;
- 1116×540 — для экранов с плотностью 208 пиксел/дюйм.

Как создать файлы, хранящие графические изображения, что предназначены для экранов с разной плотностью пикселов, говорилось в *главе 25*.

Изначально в этом поле ввода указан файл splashscreen.png из папки images, созданный Visual Studio и хранящий заставку по умолчанию.

Поле ввода **Background Color** задает цвет фона заставки. Мы можем указать как значение цвета в формате RGB, так и именованный цвет.

Задав все нужные данные, закроем окно параметров приложения и не забудем сохранить его содержимое.

Окончательное тестирование приложения

Задав основные параметры нашего Metro-приложения, мы можем приступать к его окончательному тестированию. Оно включает в себя создание дистрибутивного пакета приложения, выполнение установки приложения и тестирование его, в том числе и с помощью специальных программных средств.

Задание параметров дистрибутивного пакета

Сначала мы зададим некоторые параметры дистрибутивного пакета. Они указываются на вкладке **Packaging** окна параметров приложения (см. рис. 26.2).

В поле ввода **Package Display Name** указывается название дистрибутивного пакета, которое будет выводиться пользователю. Изначально оно совпадает с именем проекта приложения.

Поле ввода **Logo** служит для указания ссылки на файл с логотипом дистрибутивного пакета. Этот логотип должен иметь размеры 50×50 пикселов. Изначально там указан файл storelogo.png из папки images, созданный Visual Studio и хранящий логотип по умолчанию.

Указав эти сведения, сохраним их и закроем окно параметров приложения.

Создание дистрибутивного пакета

Ранее, когда мы тестировали приложения в процессе разработки, Visual Studio формировал их так называемые *отладочные редакции*. Отладочная редакция приложения включает в свой состав дополнительные данные, необходимые для его отладки в среде Visual Studio. Конечному пользователю, который будет работать с нашим приложением, эти данные совершенно не нужны; более того, они увеличивают размер приложения и теоретически могут замедлить его работу.

Поэтому перед созданием дистрибутивного пакета приложения, предназначенного для окончательного тестирования, нам следует указать Visual Studio сформировать его *окончательную редакцию*, или *релиз*. Скажем даже больше — мы обязательно дадим Visual Studio такое указание, поскольку, как говорилось в начале этой главы, отладочные редакции приложений в Windows Store однозначно не публикуются.

Сделать это очень просто. Отыщем в панели инструментов главного окна Visual Studio небольшой раскрывающийся список 

, расположенный правее кнопки, которой мы запускали приложение на выполнение. Изначально в нем выбран пункт **Debug**, говорящий о том, что в настоящее время будут создаваться отладочные редакции приложения.

Чтобы Visual Studio создал окончательную редакцию приложения, нам следует выбрать в этом списке пункт **Release**.

Теперь создадим дистрибутивный пакет. Выберем в меню **Store** пункт **Create App Package**. На экране появится диалоговое окно **Create App Package**, организованное в виде "мастера". Первая страница этого "мастера" показана на рис. 27.2.

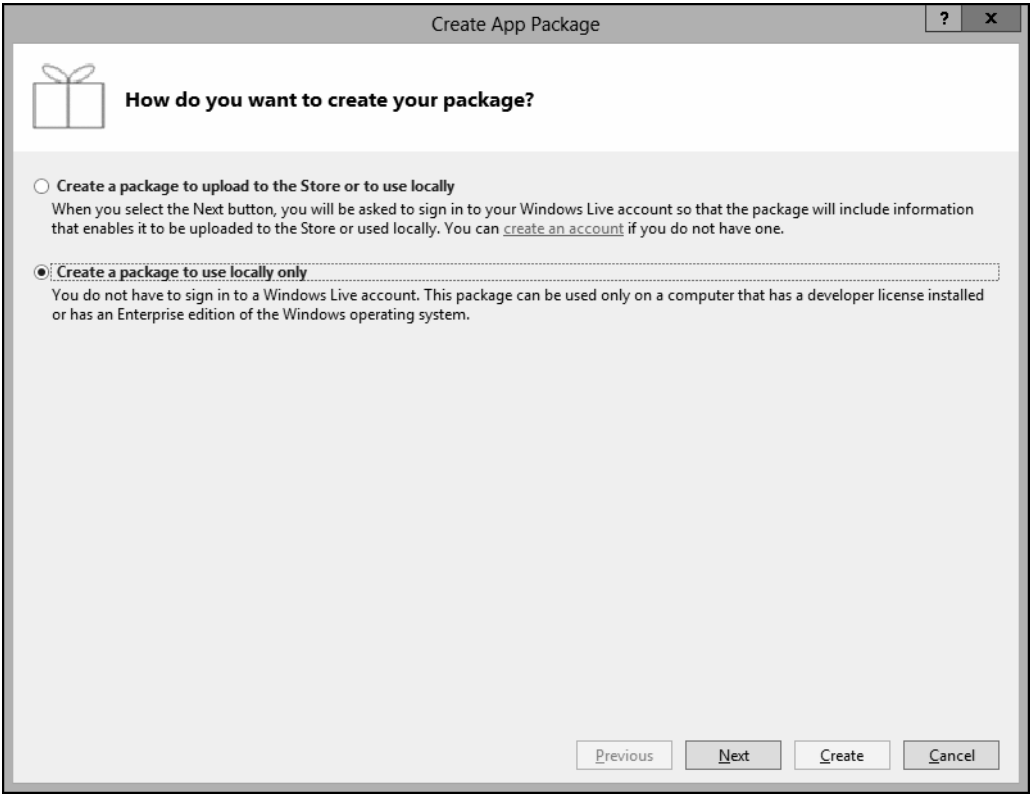


Рис. 27.2. Первая страница диалогового окна **Create App Package**

В этом окне нам следует установить переключатель **Create a package to use locally only**. Тем самым мы укажем, что собираемся использовать дистрибутивный пакет только на своем компьютере, а если и распространять, то своими силами.

Нажмем кнопку **Next**. Окно **Create App Package** переключится на следующую страницу (рис. 27.3).

Здесь мы видим большое поле ввода **Package location**, в котором указывается путь, по которому будет сохранен дистрибутивный пакет. По умолчанию он сохраняется в папке AppPackages, вложенной в папку проекта, и обычно нет большого смысла менять его расположение.

Вот группа элементов управления **Version** будет для нас более полезна. Она позволяет указать версию приложения.

Собственно номер версии указывается в четырех полях ввода, находящихся непосредственно под заголовком группы. Версия задается в формате *<номер версии>.<номер подверсии>.<номер мофицикации>.<номер сборки>*.

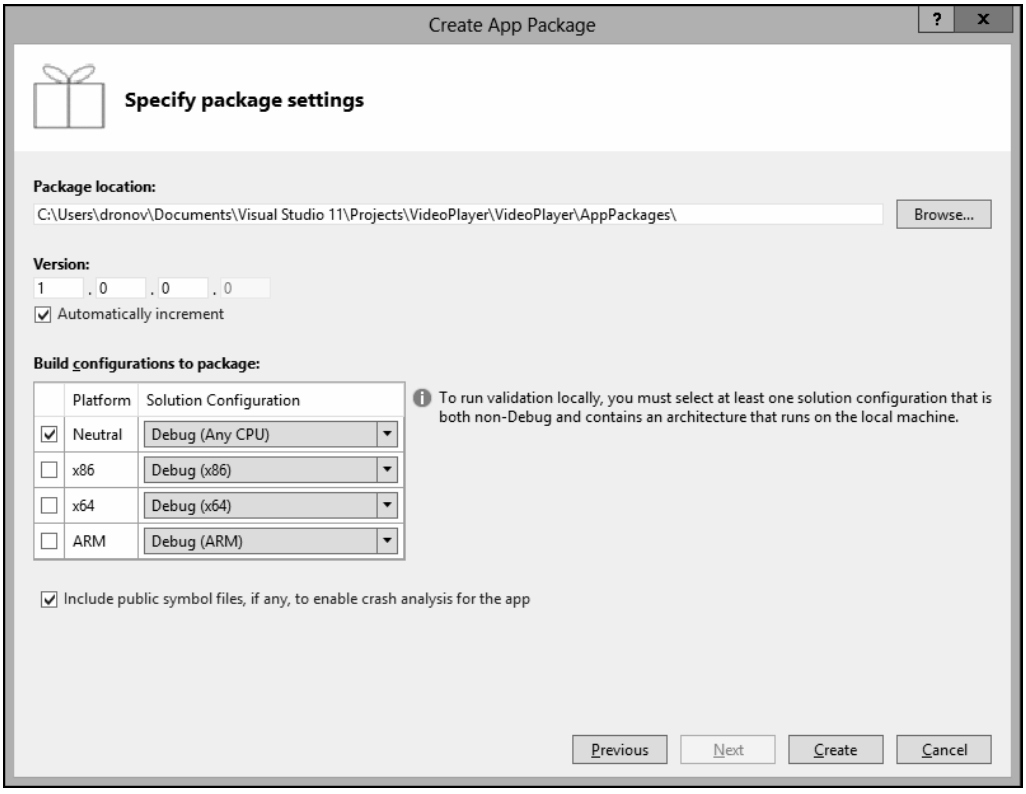


Рис. 27.3. Вторая страница диалогового окна **Create App Package**

Если установлен флажок **Automatically increment** (а он установлен по умолчанию), номер сборки (самый последний компонент версии) будет автоматически увеличиваться на единицу при создании очередного пакета.

Остальные элементы управления, находящиеся на второй странице окна, мы трогать не будем — установленные в них значения являются оптимальными. Поэтому нажмем кнопку **Create**, чтобы запустить процесс создания дистрибутивного пакета.

Когда пакет будет создан, окно **Create App Package** переключится на третью страницу. На ней мы увидим гиперссылку, ведущую в папку, где был сохранен только что созданный дистрибутивный пакет. Нажмем кнопку **OK**, чтобы закрыть это окно.

Теперь перейдем в папку AppPackages, что находится в папке проекта. Там мы увидим другую папку — с именем вида *<имя проекта>_<версия приложения>*_AnyCPU_Test. Именно в ней будут находиться файлы, составляющие дистрибутивный комплект нашего приложения.

Установка приложения из дистрибутивного комплекта

Этот дистрибутивный комплект мы можем использовать для тестирования приложения, так сказать, вручную. Еще мы можем передать этот комплект на тестирова-

ние другим разработчикам; их независимое мнение будет для нас чрезвычайно ценным.

> ***Внимание!***
>
> Установить приложение из дистрибутивного комплекта могут только пользователи, имеющие лицензию разработчика Metro-приложений. Эта лицензия автоматически устанавливается при первом запуске Visual Studio и в дальнейшем автоматически обновляется при устаревании.
>
> При запуске приложения на выполнение Visual Studio каждый раз создает и устанавливает в системе его отладочную версию. Эту версию перед установкой приложения из дистрибутивного пакета обязательно следует удалить, иначе установка не будет выполнена.

Чтобы установить приложение из дистрибутивного пакета, нужно открыть папку, где находятся файлы этого пакета, и найти там командный файл Add-AppxDevPackage.bat. Этот файл следует запустить с повышенными правами, щелкнув на нем правой кнопкой мыши, выбрав в появившемся на экране контекстном меню пункт **Запуск от имени администратора** (Run as Administrator) и положительно ответив на предупреждение UAC.

После этого на экране появится окно командной строки, в котором будет отображаться ход установки. По его окончанию нужно нажать любую клавишу, чтобы закрыть это окно.

Установив приложение, мы можем запустить его и проверить в действии.

Тестирование приложения с помощью Windows App Cert Kit

Чтобы облегчить процесс тестирования Metro-приложений, Microsoft включила в состав Visual Studio особый программный инструмент — *Windows App Cert Kit*. Он позволяет проверить:

- корректно ли сформирован дистрибутивный пакет приложения (содержит ли он все нужные файлы и данные и не включает ли чего лишнего);
- использует ли приложение только стандартные механизмы платформы Metro;
- достаточно ли быстро приложение активизируется и деактивируется;
- содержит ли приложение все необходимые языковые ресурсы;
- не нарушает ли приложение правил безопасности (не имеет ли "лишних" прав и не компрометирует ли пользователя);
- наконец, не содержит ли приложение критических ошибок, могущих привести к его "падению".

Запустить Windows App Cert Kit можно из меню **Пуск**, нажав одноименную плитку. На экране появится предупреждение системы UAC, на которое следует ответить положительно.

Далее мы увидим окно Windows App Cert Kit, организованное в виде "мастера" и изначально открытое на первой странице (рис. 27.4). Поскольку мы собираем-

ся протестировать Metro-приложение, то нажмем здесь кнопку **Validate Metro style App**.

На следующей странице окна мы сможем выбрать Metro-приложение, которое хотим протестировать (рис. 27.5).

Все установленные в системе Metro-приложения перечислены в большом списке, занимающем почти все окно. Найдем там пункт, соответствующий нужному нам

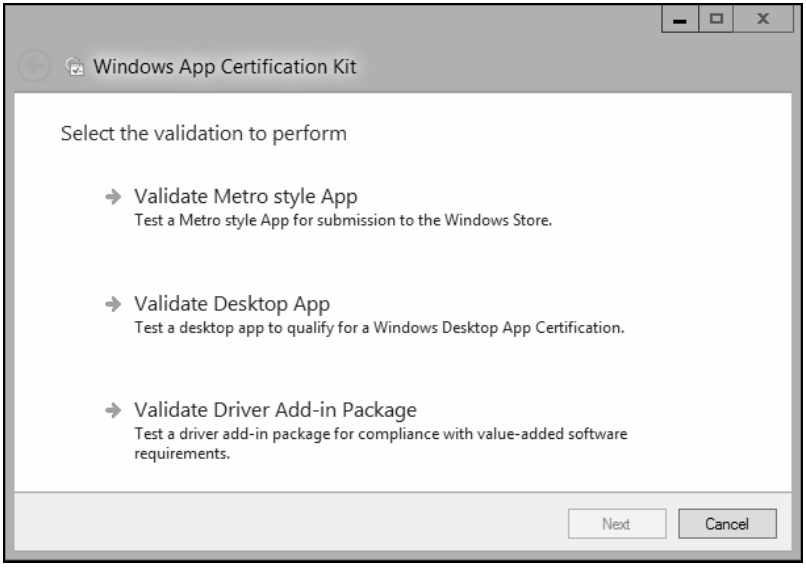


Рис. 27.4. Первая страница окна утилиты Windows App Cert Kit

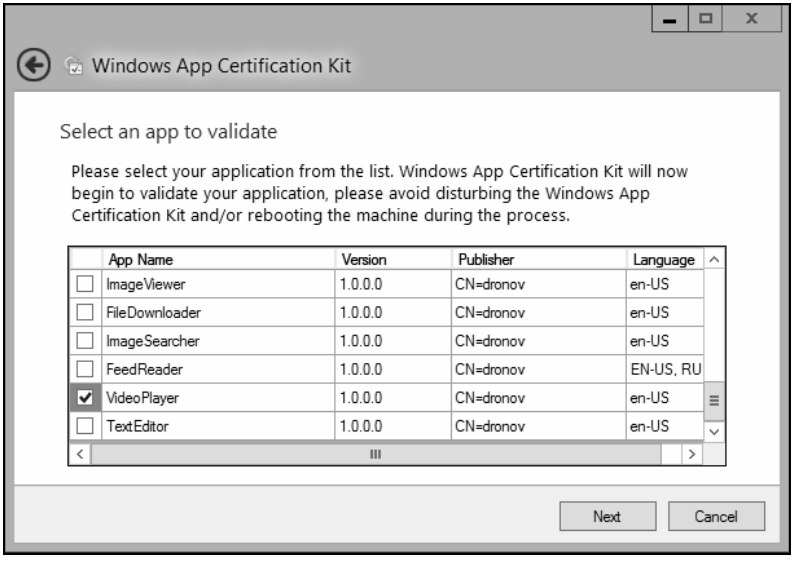


Рис. 27.5. Вторая страница окна утилиты Windows App Cert Kit

приложению, и установим присутствующий в его левой части флажок. После чего нажмем кнопку **Next**.

Сразу после этого Windows App Cert Kit предупредит нас, что сейчас начнется сам процесс тестирования, в течение которого приложение будет несколько раз запускаться и завершаться; при этом все попытки работать с ним недопустимы. Закроем окно-предупреждение нажатием кнопки **OK** и подождем, пока тестирование не закончится.

А когда оно закончится, Windows App Cert Kit выведет на экран стандартное диалоговое окно сохранения файла. В этом окне мы зададим расположение и имя XML-файла, в котором будут сохранены результаты тестирования.

После сохранения результатов тестирования окно утилиты переключится на третью страницу. В ней мы увидим гиперссылку **Click here to view the results**, открывающую XML-файл с результатами в Web-обозревателе.

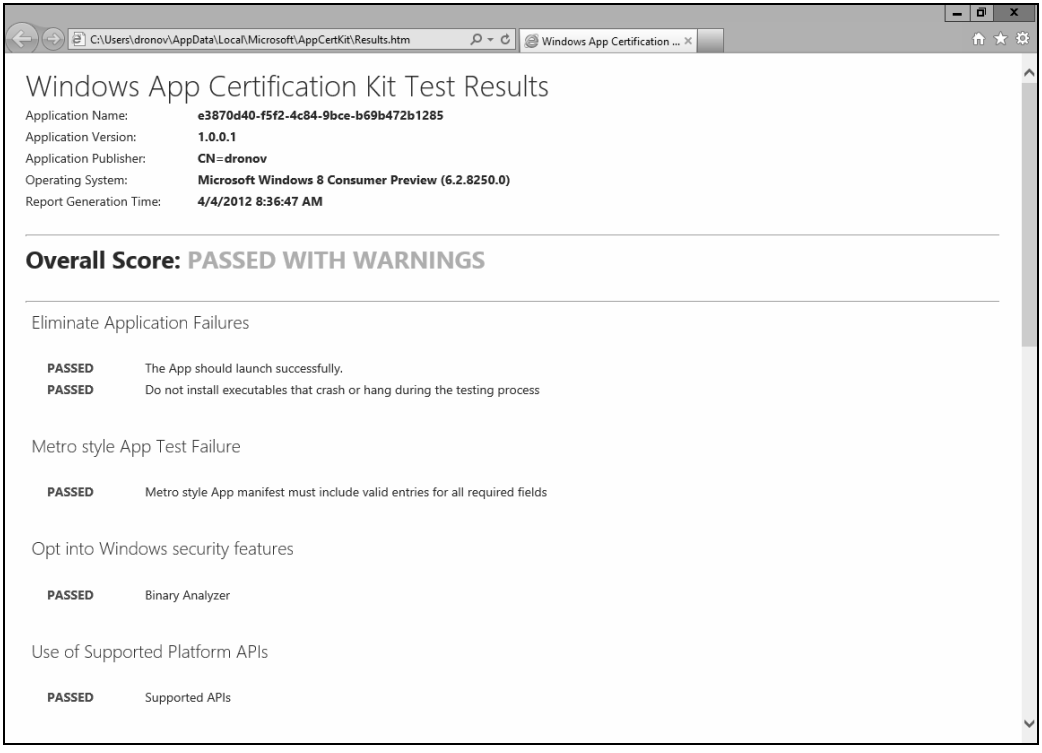


Рис. 27.6. Результаты тестирования Metro-приложения

Пример этих результатов можно увидеть на рис. 27.6. Видно, что приложение прошло все тесты, но у Windows App Cert Kit возникли претензии в плане его производительности: приложение слишком долго активизируется и деактивируется. Дело в том, что автор работал с Windows 8, установленной в виртуальном окружении, следовательно, производительность системы оставляла желать лучшего.

Закончив тестирование приложения, закроем Web-обозреватель и нажмем кнопку **Finish** окна Windows App Cert Kit, чтобы завершить работу этой утилиты.

Регистрация в магазине Windows Store

Чтобы получить возможность публикации приложений в магазине Windows Store, нам потребуется там зарегистрироваться. Сделать это можно прямо в среде Visual Studio.

Выберем в меню **Store** пункт **Open Developer Account**. На экране появится окно Web-обозревателя с открытой в нем Web-страницей Windows Store Dashboard. Там мы сможем как выполнить регистрацию в магазине Windows Store, так и войти в него, использовав полученные при регистрации учетные данные.

Для регистрации в магазине Windows Store нам потребуется указать следующие данные:

- тип регистрации (как индивидуальный разработчик или как организация);
- интернет-адрес нашего Web-сайта (честно говоря, автор не понял, зачем его нужно указывать, ведь этот интернет-адрес пользователи в любом случае не увидят);
- имя или название организации, которое будет отображаться пользователям;
- контактные данные (в первую очередь, адрес электронной почты);
- сведения об организации, если мы регистрируемся как организация.

Далее следует прочитать и принять соглашение для разработчиков.

Через некоторое время на указанный нами адрес электронной почты придет письмо, содержащее регистрационный код. Этот код мы укажем в составе данных, требуемых для завершения регистрации.

Напоследок останется только ввести банковские реквизиты, по которым будут перечисляться деньги, вырученные за продажу приложений.

За регистрацию в магазине Windows Store Microsoft взимает ежегодную плату. Сумма этой платы составляет 1500 руб. за регистрацию в качестве индивидуального разработчика и 3000 руб. за регистрацию в качестве организации. Перечисление денег в счет оплаты регистрации выполняется банковским переводом.

Публикация приложения

Перед тем как собственно выполнять публикацию Metro-приложения в магазине Windows Store, нам потребуется подготовить все необходимые сведения об этом приложении. Эти сведения перечислены далее.

- *Название приложения.* Должно быть уникальным в пределах магазина приложений, отражать назначение и функциональность приложения и, по возможности, сразу привлекать внимание потенциальных пользователей. Длина названия ограничена 256 символами. Обычно оно совпадает с названием приложения, ко-

торое задается в поле ввода **Display Name** на вкладке **Application UI** окна параметров приложения (см. рис. 27.1).

- *Развернутое описание приложения.* Должно максимально полно описывать назначение и ключевые возможности приложения, делая акцент на уникальных функциях, которые отличают это приложение от конкурентов. Длина описания может составлять максимум 10 000 символов.
- *Список ключевых возможностей приложения.* Указывается отдельно от описания и перечисляет самые впечатляющие, с точки зрения разработчика, возможности приложения, отличающие его от конкурентов. Может содержать максимум 20 пунктов, длина каждого из которых ограничена 200 символами.
- *Список ключевых слов.* Эти ключевые слова будут использоваться при поиске приложений в Windows Store. Каждое ключевое слово должно характеризовать какую-либо из ключевых возможностей приложения (например, для "читалки" каналов новостей можно указать ключевые слова "feed", "reader", "rss" и "atom"). Допускается указание до 7 ключевых слов, каждое из которых может содержать до 45 символов.
- *Описание новой версии приложения.* Указывается только для новой версии уже опубликованного ранее приложения. Должно кратко описывать новые возможности приложения, появившиеся в этой версии. Может содержать до 1500 символов.
- *Сведения о правах разработчика.* Могут включать до 200 символов.
- *Текст дополнительного лицензионного соглашения.* Указывается только в том случае, если приложение не полностью удовлетворяет стандартной пользовательской лицензии (ее текст можно найти на Web-странице **http://msdn.microsoft.com/en-us/library/windows/apps/hh694058.aspx**, приложение А). Длина дополнительного лицензионного соглашения не должна превышать 10 000 символов.
- *Изображения, представляющие интерфейс приложения в различные моменты его работы.* Допускается указание от одного до восьми изображений минимального разрешения 1366×768 или 768×1366 пикселов. Изображения должны храниться в файлах формата PNG; размер файлов не должен превышать 2 Мбайт. К каждому изображению следует указать текстовое примечание длиной до 200 символов.
- *Изображения, представляющие интерфейс приложения и предназначенные для публикации в разделе рекомендуемых приложений.* Указываются отдельно от изображений, описанных ранее. Допускается указание от одного до четырех изображений с разрешением 414×180, 414×468, 558×756 или 846×468 пикселов. Изображения должны храниться в файлах формата PNG; размер файлов не должен превышать 2 Мбайт.

> ***Внимание!***
>
> Приложения, для которых не были указаны эти изображения, не могут быть отобраны в раздел рекомендуемых.

- *Список специфических аппаратных требований.* Указывается только в том случае, если приложению для успешной работы требуются аппаратные средства, не являющиеся стандартными для типовых Windows-планшетов. В этом списке допускается указание до 11 пунктов, длина каждого из которых ограничена 200 символами.
- *Интернет-адрес "домашнего" Web-сайта приложения*, содержащего развернутое описание, инструкции по работе с приложением и устранению возникающих проблем и пр. Может содержать до 2048 символов.
- *Контактные данные.* Должны представлять собой либо адрес электронной почты, либо интернет-адрес Web-страницы, где пользователи смогут оставить замечания по работе с приложением и описания возникших проблем. Могут содержать до 2048 символов.
- *Текст дополнительного соглашения об информационной безопасности.* Указывается только в том случае, если приложение не удовлетворяет стандартной пользовательской лицензии (Web-страница **http://msdn.microsoft.com/en-us/library/windows/apps/hh694058.aspx**, приложение A). Не должен содержать более 2048 символов.
- *Рейтинг приложения.* Требуется только для игр.
- *Сведения о том, применяются ли в приложении криптографические алгоритмы*, чей экспорт ограничен действующим на территории страны законодательством.
- *Модель распространения.* Является приложение бесплатным, условно-бесплатным или платным.
- *Стоимость приложения.*
- *Продолжительность тестового периода.* Указывается только для условно-бесплатных приложений.
- *Языки, поддерживаемые приложением.*
- *Предпочитаемая дата публикации приложения.* Доступны две возможности: публикация приложения сразу после окончания его тестирования персоналом Windows Store и публикация не ранее указанной нами даты.
- *Категория и подкатегория, к которым будет отнесено приложение.* Список всех доступных в Windows Store категорий и подкатегорий можно посмотреть на Web-странице **http://msdn.microsoft.com/en-us/library/windows/apps/hh694073.aspx**.
- *Является ли приложение доступным для пользователей с ограниченными возможностями.* (Рассмотрение разработки таких приложений выходит за рамки этой книги.)
- *Список платных функций приложения, если они есть.* Каждый пункт этого списка должен включать:
  - идентификатор платной функции, определенный в коде логики приложения (подробнее *см. в главе 26*);

- стоимость платной функции;
- описание платной функции, включающее не более 100 символов.

❒ *Примечания для персонала Windows Store*, который будет тестировать приложение перед его публикацией. К таким примечаниям можно отнести, например, тестовые учетные данные для подключения к Web-сервису, с которым работает приложение, или описание доступа к его скрытым функциям. Не должно содержать более 500 символов.

Собрав все эти данные, можно приступать к публикации приложения.

Сначала мы создадим дистрибутивный пакет, предназначенный для публикации в Windows Store. Выполняется это так же, как и в случае создания пакета для тестирования, только на первой странице окна **Create App Package** (см. рис. 27.2) следует установить переключатель **Create a package to upload to the Store or to use locally**, а потом — выполнить вход в магазин Windows Store, используя полученные ранее при регистрации учетные данные.

Для загрузки готового дистрибутивного пакета в магазин приложений нужно выбрать пункт **Upload App Package** меню **Store**. На экране появится окно Web-обозревателя, в котором будет открыта Web-страница, предназначенная для указания всех сведений о публикуемом приложении. Поскольку все нужные сведения мы уже собрали, это не составит для нас проблемы. На этой же Web-странице мы сможем указать путь к папке, где находится дистрибутивный комплект, и собственно выполнить публикацию приложения.

Останется только дождаться волнующего момента, когда наше первое Metro-приложение появится в магазине Windows Store. И мы сможем с гордостью именовать себя Metro-программистами!

На сей оптимистической ноте автор и заканчивает свою книгу.

Заключение

Что ж, книга закончена. Перевернута последняя страница, и автор очень скоро попрощается с вами, уважаемые читатели. Но напоследок хочет сказать еще кое-что...

Мы прошли курс разработки Metro-приложений для устройств, в первую очередь планшетов, работающих под управлением Windows 8. Мы познакомились с возможностями платформы Metro по разработке интерфейса, оформления и логики приложений. Мы выучили три компьютерных языка — HTML, CSS и JavaScript, — что обязательно понадобятся нам для работы. Мы выяснили, какие механизмы Metro следует применять при работе с файлами, какие — для взаимодействия с удаленными интернет-сервисами, а какие — для хранения пользовательских настроек. Мы разобрались со средой разработки Visual Studio и использовали ее для создания нескольких пробных Metro-приложений, которые станут нашим заделом на будущее.

И, тем не менее, разговор о платформе Metro еще далеко не закончен. Впереди нас ждет немало интересного.

Когда автор писал эти строки, широкой публике была доступна только тестовая редакция Windows 8 — Windows 8 Consumer Preview. Версия платформы Metro, включенная в ее состав, также не являлась окончательной. Многие компоненты Metro на момент написания книги еще не были завершены, а какие-то из них, возможно, даже еще не были созданы.

Набор возможностей платформы Metro очень велик, настолько велик, что описать его в одной книге, да еще и не слишком толстой, просто невозможно. Потребуется много толстых томов, чтобы все подробно задокументировать.

Кроме того, многие инструменты платформы Metro предназначены для выполнения весьма специфических задач, с которыми столкнется далеко не каждый Metro-разработчик. Естественно, что эти инструменты также остались "за бортом".

- Автор не рассматривал механизмы Metro, призванные обеспечивать взаимодействие со встроенными и подключаемыми устройствами: принтерами, датчиками (освещенности, направления, ускорения и др.), приемниками GPS и т. п.
- Автор не касался встроенных в Metro средств поиска и настройки приложений, средств для сложной обработки мультимедиа, в частности его перекодирования,

инструментов для работы с контактами пользователя и некоторых других возможностей этой платформы.

- Автор не описывал некоторые специфичные инструменты языка HTML: графическую канву, индексированное хранилище, средства кэширования данных и др.
- Автор изложил только те сведения о публикации Metro-приложений, что нашел на Web-сайте Microsoft. В то время, когда писалась книга, магазин Windows Store был закрыт для сторонних разработчиков (зарегистрироваться в нем могли лишь доверенные партнеры Microsoft), так что автору не удалось лично проверить все описанное в книге.
- Автор весьма поверхностно рассказал о самой среде разработки Visual Studio, описав только те ее средства, что потребуются начинающим разработчикам. В частности, автор не рассказал об отладке приложений с использованием пошагового прохода, контрольных точек и др.

Тем не менее данная книга может помочь многим и многим сделать свои первые шаги в мир Metro-программирования и создать свои первые Metro-приложения. Которые — кто знает — вполне возможно, станут популярными и завоюют сердца пользователей во всем мире. Ведь индустрия информационных технологий — самая демократичная отрасль мировой экономики; здесь даже новичок, не имеющий никакого капитала, может добиться оглушительного успеха.

Для таких целеустремленных новичков автор приводит табл. 31, где перечислены некоторые полезные для дальнейшего самообразования интернет-ресурсы.

Таблица 31. *Полезные интернет-ресурсы по теме книги*

Интернет-адрес	Описание
http://msdn.microsoft.com/en-us/windows/apps/	Раздел Web-сайта MSDN, посвященный Metro-программированию. Документация, справочники и готовые примеры приложений. Отсюда же можно загрузить Visual Studio
http://msdn.microsoft.com/library/default.aspx	Основной Web-сайт MSDN (Microsoft Developer Network). В ряде случаев может пригодиться
http://www.thevista.ru/	Web-сайт TheVista.ru, посвященный технологиям Microsoft и, в частности, Windows 8 и Metro. Новости, обзорные статьи и материалы для разработчиков, в том числе и вышедшие из-под пера автора этой книги

Вот и все. Автору осталось только распрощаться с читателями и пожелать им успехов в Metro-программировании.

До свидания!

Владимир Дронов

Предметный указатель

Л

М

H

O

П

P

C

T

У

Ф

Х

Ц

Ч

Ш

Э

Я