Глава 7. Асинхронное выполнение

В наши дни программистам не рекомендуется слишком часто использовать окна сообщений. Но конечно, читатели прекрасно знают, насколько полезными они могут быть для оперативного вывода важной информации или получения жизненно важных указаний «Да/Нет/Отменить».

В Windows Runtime поддержка окон сообщений представлена классом `MessageDialog`, который обладает достаточной гибкостью: вы можете снабдить до трех кнопок любым текстом на свое усмотрение. Однако в этом классе нет метода `Show` — этот ожидаемый вроде бы метод был заменен методом `ShowAsync`.

Суффикс `Async` означает «асинхронный» (asynchronous). Эти пять букв играют исключительно важную роль в Windows Runtime. Изменяется не только имя; изменяется способ использования метода, а в конечном итоге и философия программирования для современных операционных систем — таких, как Windows 8.

Программные потоки и пользовательский интерфейс

Программы для Windows 8, как и приложения для более ранних версий Windows, по своей структуре напоминают конечный автомат. После инициализации программа обычно «дремлет» в памяти, ожидая событий. Очень часто события сигнализируют о взаимодействии пользователя с программой, но иногда они сообщают об изменениях системного уровня — например, изменении ориентации экрана.

Очень важно, чтобы приложения обрабатывали события как можно быстрее, а затем возвращали управление операционной системе, чтобы ожидать новых событий. Если приложение недостаточно быстро обрабатывает события, оно перестает быстро реагировать на действия пользователя и раздражает его. По этой причине приложения должны передавать продолжительные операции во вторичные программные потоки исполнения. Поток, обслуживающий пользовательский интерфейс, не должен отягощаться интенсивными вычислениями.

Но что, если вызов метода самой среды Windows Runtime занимает много времени? Разработчик должен предвидеть проблему и вынести вызов во вторичный поток?

Нет, это выглядит неразумно. По этой причине разработчики Microsoft в ходе проектирования Windows Runtime постарались выявить методы, при вызове которых до возвращения управления приложению может пройти более 50 миллисекунд. К этой категории относилось примерно 10–15 % методов Windows Runtime. Такие методы были сделаны асинхронными, то есть сами методы запускают вторичные потоки для выполнения продолжительных вычислений. Они очень быстро возвращают управление приложению, а позднее оповещают его о завершении своей работы.

Асинхронные методы чаще всего встречаются при выполнении файловых операций ввода/вывода или при обращении к Интернету. Однако они также задействованы в отображении диалоговых окон, реализованных в Windows 8, например окон сообщений `MessageDialog` и окон выбора файлов, которые встретятся нам позднее в этой главе. Все асинхронные методы в Windows Runtime имеют суффикс `Async` и определяются по похожей схеме. К счастью, в результате появления мощных библиотек .NET и расширений языка программирования C# работа с асинхронными методами стала куда менее обременительной.

Скорее всего, важность навыков асинхронного программирования в ближайшие годы только возрастет. На массовых компьютерах прошлого все программные потоки работали на одном процессоре. Операционная система быстро переключала потоки, создавая иллюзию их одновременного выполнения. Однако в последнее время компьютеры стали часто оснащаться несколькими процессорами, обычно размещаемыми на одной микросхеме в многоядерной конфигурации. Такие аппаратные решения позволяют разным потокам выполняться на разных процессорах.

Некоторые вычислительные задачи — например, обработка массивов — могут использовать многопроцессорную архитектуру, запуская несколько вычислительных задач параллельно. Для поддержки асинхронных и параллельных вычислений в .NET была включена поддержка TAP (Task-based Asynchronous Pattern), в которой центральное место занимает класс `Task` из пространства имен `System.Threading.Tasks`. К этой части .NET можно обращаться из приложений Windows Runtime, написанных на C# и Visual Basic, а по гибкости и мощи она существенно превосходит асинхронные средства самой среды Windows Runtime.

Работа с MessageDialog

Чтобы лучше понять, как используются асинхронные функции, рассмотрим класс `MessageDialog`. Конструктор `MessageDialog` получает строку сообщения и (возможно) заголовок; по умолчанию в окне отображается одна кнопка с текстом «Close». Такого окна достаточно для вывода важной информации для пользователя. Также можно определить до трех пользовательских кнопок при помощи объектов `UICommand`. Ниже приведен фрагмент кода из проекта `HowToAsync1`:

```
MessageDialog msgdlg = new MessageDialog("Choose a color", "How To Async #1");
msgdlg.Commands.Add(new UICommand("Red", null, Colors.Red));
msgdlg.Commands.Add(new UICommand("Green", null, Colors.Green));
msgdlg.Commands.Add(new UICommand("Blue", null, Colors.Blue));
```

В первом аргументе конструктора `UICommand` передается текст, выводимый на кнопке, а в третьем — идентификатор типа `object` (то есть произвольная информация, которая может использоваться для идентификации кнопки). Я решил воспользоваться значением `Color`, которое представляет кнопка. Второй аргумент будет рассмотрен чуть позже.

Класс `UICommand` реализует интерфейс `IUICommand`. Когда класс `MessageDialog` сообщает вашей программе, какая кнопка была нажата, он делает это при помощи объекта типа `IUICommand`.

Асинхронная обработка начинается с вызова `ShowAsync`. Метод вызывается без аргументов и быстро возвращает управление приложению. Само окно сообщения обслуживается вторичным программным потоком. Вызов выглядит так:

```
IAsyncOperation<IUICommand> asyncOp = msgdlg.ShowAsync();
```

`ShowAsync` возвращает объект, реализующий обобщенный интерфейс `IAsyncOperation`. В обобщенном аргументе передается интерфейс `IUICommand`; это означает, что `MessageDialog` возвращает объект типа `IUICommand`, хотя и не сразу. Он не может вернуть значение, пока пользователь не нажмет одну из кнопок, что приведет к закрытию окна сообщения, а ведь оно еще даже не появилось на экране!

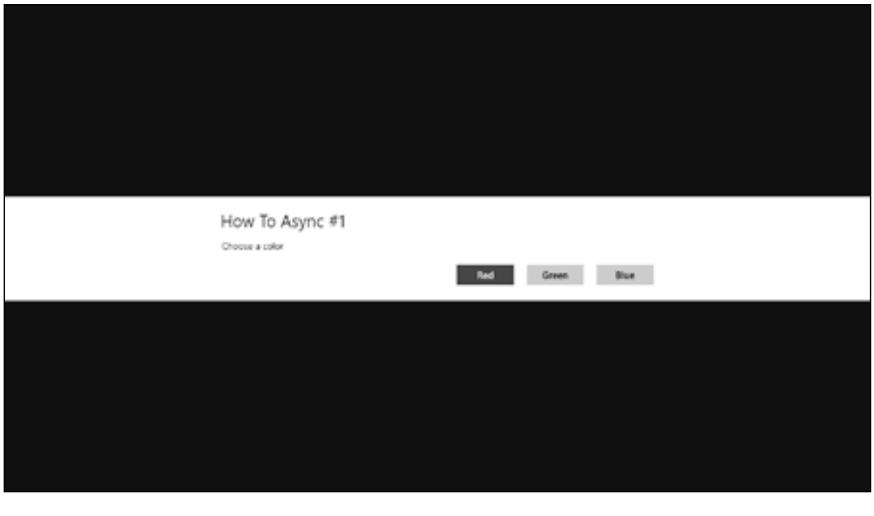
`IAsyncOperation<T>` наследует от интерфейса `IAsyncInfo`, который определяет методы с именами `Cancel` и `Close` и свойства с именами `Id`, `Status` и `ErrorCode`. Интерфейс `IAsyncOperation<T>` дополнительно определяет свойство с именем `Completed`, которое является делегатом типа `AsyncOperationCompletedHandler<T>`.

Свойству `Completed` задается метод обратного вызова, определенный в вашем коде. Хотя `Completed` определяется как свойство, оно скорее работает как событие — в том отношении, что оно оповещает вашу программу о чем-то представляющем интерес для нее. (Различие состоит в том, что событие может иметь несколько обработчиков, а свойство — только один.)

Вот как это делается:

```
asyncOp.Completed = OnMessageDialogShowAsyncCompleted;
```

Если метод вашей программы, который вызвал `ShowAsync` и задал обработчик `Completed`, содержит дополнительный код, этот код начнет выполняться далее. Только после того, как метод, вызвавший `ShowAsync`, вернет управление операционной системе, на экране появляется объект `MessageDialog`.



С этим объектом `MessageDialog` работает поток, созданный специально для этой цели. И хотя пользовательский интерфейс программы блокируется на время отображения `MessageDialog`, поток пользовательского интерфейса продолжает выполнять свою работу.

Обратите внимание: кнопка Red окрашена по цвету отличается от других кнопок. Это кнопка по умолчанию, которая сработает при нажатии пользователем клавиши Enter. Кнопку по умолчанию можно сменить при помощи свойства DefaultCommandIndex объекта MessageDialog. Свойство CancelCommandIndex управляет тем, какая кнопка будет активизирована при нажатии пользователем клавиши Esc.

Когда пользователь нажимает кнопку, окно сообщения закрывается и вызывается метод обратного вызова Completed вашей программы. В первом аргументе метода передается объект, возвращенный ShowAsync, но я присвоил ему другое имя (asyncInfo), потому что теперь он содержит полезную информацию:

```
void OnMessageDialogShowAsyncCompleted(IAsyncOperation<IUICommand> asyncInfo,
                                       AsyncStatus asyncStatus)
{
    // Получение значения Color
    IUICommand command = asyncInfo.GetResults();
    Color clr = (Color)command.Id;
    ...
}
```

Аргумент IAsyncOperation содержит свойство Status типа AsyncStatus — перечисления, состоящего из четырех значений: Started, Completed, Canceled и Error. Значение воспроизводится во втором аргументе обработчика Completed. Если произойдет ошибка — что неактуально для класса MessageDialog, но безусловно возможно при файловом вводе/выводе или обращении к Интернету, — свойство ErrorCode объекта IAsyncOperation содержит объект типа Exception.

В общем случае перед вызовом GetResults следует убедиться в том, что операция завершилась со статусом Completed. Метод GetResults возвращает объект, тип которого соответствует типу обобщенного аргумента IAsyncOperation; в нашем случае это объект типа IUICommand, обозначающий нажатую кнопку. Из него можно получить свойство Id, которое передается в третьем аргументе конструктора UICommand. В нашем примере оно может быть преобразовано в значение Color.

Например, этот цвет может быть использован программой для задания фоновой кисти Grid:

```
contentGrid.Background = new SolidColorBrush(clr);
```

Стоп, не так быстро!

Когда ваша программа вызывает ShowAsync, класс MessageDialog создает вторичный поток для отображения окна сообщения и кнопок. Когда пользователь нажимает кнопку, вызывается обработчик Completed в вашем коде, но он выполняется во вторичном потоке, из которого нельзя обратиться к объектам пользовательского интерфейса!

Для любого конкретного окна может быть только один поток приложения, который обрабатывает ввод пользователя и отображает элементы управления и графику, взаимодействующие с этим потоком. Соответственно этот «поток пользовательского интерфейса» (или «UI-поток») играет очень важную роль в работе приложений Windows, потому что все взаимодействия с пользователем должны осуществляться через этот поток. Но только код, выполняемый в этом потоке, может обращаться к элементам и элементам управления, образующим пользовательский интерфейс.

Это ограничение можно обобщить: класс DependencyObject не является потоково-безопасным. К любому объекту класса, производного от DependencyObject, может обращаться только поток, создавший этот объект.

В этом конкретном примере значение Color может быть получено вторичным потоком, потому что Color является структурой, а следовательно, не наследует от DependencyObject. Однако любой код, применяющий значение Color к пользовательскому интерфейсу, должен выполняться в UI-потоке.

К счастью, решение существует. Для компенсации своей потоковой небезопасности класс DependencyObject содержит свойство Dispatcher, которое возвращает объект типа CoreDispatcher. Свойство Dispatcher является исключением из правила, запрещающего обращение к DependencyObject из другого потока. Свойство HasThreadAccess класса CoreDispatcher позволяет узнать, возможно ли обращение к конкретному объекту DependencyObject из программного потока, в котором выполняется код.

Если обращение невозможно (и даже если возможно), вы можете поместить фрагмент кода в очередь на исполнение потока, создавшего объект.

Чтобы поставить код в очередь на выполнение в потоке пользовательского интерфейса, следует вызвать метод RunAsync, определяемый CoreDispatcher. Это еще один асинхронный метод, а при вызове ему передается метод, который должен быть выполнен в потоке пользовательского интерфейса:

```
void OnMessageDialogShowAsyncCompleted(IAsyncOperation<IUICommand> asyncInfo,
                                       AsyncStatus asyncStatus)
{
    ...
    this.Dispatcher.RunAsync(CoreDispatcherPriority.Normal,
                             OnDispatcherRunAsyncCallback);
}

void OnDispatcherRunAsyncCallback()
{
    contentGrid.Background = new SolidColorBrush(clr);
}
```

Обычно объект CoreDispatcher, полученный из свойства Dispatcher, не сохраняется как переменная; метод RunAsync просто вызывается для самого свойства Dispatcher, как в приведенном примере. Метод обратного вызова, переданный методу RunAsync, может безопасно обращаться к элементам пользовательского интерфейса. Однако следует заметить, что передать произвольную информацию этому методу невозможно; это означает, что метод OnMessageDialogShowAsyncCompleted должен сначала сохранить значение Color как поле.

Неважно, из какого элемента пользовательского интерфейса будет получен объект CoreDispatcher; все объекты пользовательского интерфейса создаются в одном UI-потоке, поэтому все они работают идентично.

Хотя это не показано в приведенном коде, метод RunAsync класса CoreDispatcher возвращает объект IAsyncAction:

```
IAsyncAction asyncAction = this.Dispatcher.RunAsync(CoreDispatcherPriority.Normal,
                                                    OnDispatcherRunAsyncCallback);
```

Объект `IAsyncAction` очень похож на объект `IAsyncOperation`, возвращаемый методом `ShowAsync` класса `MessageDialog`. Оба объекта реализуют интерфейс `IAsyncInfo`. Серьезное различие между ними заключается в том, что `IAsyncOperation` используется для асинхронных методов, которые должны возвращать что-то программе (отсюда и обобщенный аргумент), тогда как `IAsyncAction` используется для асинхронных методов, не возвращающих информации.

Иерархия интерфейсов выглядит так:

```
IAsyncInfo
 IAsyncAction
 IAsyncActionWithProgress<TProgress>
 IAsyncOperation<TResult>
 IAsyncOperationWithProgress<TResult, TProgress>
```

Некоторые асинхронные методы могут возвращать информацию о ходе выполнения (прогрессе) асинхронных операций; они используют специальные интерфейсы.

Как бы то ни было, вы можете задать в обработчике `Completed` объект `IAsyncAction`, возвращенный методом `RunAsync` объекта `CoreDispatcher`, и использовать его для обращения к пользовательскому интерфейсу:

```
void OnMessageDialogShowAsyncCompleted(IAsyncOperation<IUICommand> asyncInfo,
                                       AsyncStatus asyncStatus)
{
    ...
    IAsyncAction asyncAction =
                this.Dispatcher.RunAsync(CoreDispatcherPriority.Normal,
                OnDispatcherRunAsyncCallback);
    asyncAction.Completed = OnDispatcherRunAsyncCompleted;
}

void OnDispatcherRunAsyncCompleted(IAsyncAction asyncInfo, AsyncStatus asyncStatus)
{
    contentGrid.Background = new SolidColorBrush(clr);
}
```

Этот конкретный обработчик `Completed` выполняется в потоке пользовательского интерфейса. Присутствие дополнительного метода не имеет особого смысла; просто задать второй аргумент метода `RunAsync` равным `null` нельзя, поэтому нужен второй метод.

Ниже приведен полный код проекта `HowToAsync1`. Файл XAML содержит всего одну кнопку, предназначенную для вызова `MessageDialog`:

Проект: HowToAsync1 | **Файл**: MainPage.xaml (фрагмент)

```
<Grid Name="contentGrid"
      Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Button Content="Show me a MessageDialog!"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Click="OnButtonClick" />

</Grid>
```

В файле фонового кода ничего удивить уже не должно:

Проект: HowToAsync1 | **Файл**: MainPage.xaml.cs (фрагмент)

```
public sealed partial class MainPage : Page
{
    Color clr;
```

```
    public MainPage()
    {
        this.InitializeComponent();
    }

    void OnButtonClick(object sender, RoutedEventArgs args)
    {
        MessageDialog msgdlg = new MessageDialog("Choose a color",
                               "How To Async #1");
        msgdlg.Commands.Add(new UICommand("Red", null, Colors.Red));
        msgdlg.Commands.Add(new UICommand("Green", null, Colors.Green));
        msgdlg.Commands.Add(new UICommand("Blue", null, Colors.Blue));

        // Отображение MessageDialog с обработчиком Completed
        IAsyncOperation<IUICommand> asyncOp = msgdlg.ShowAsync();
        asyncOp.Completed = OnMessageDialogShowAsyncCompleted;
    }

    void OnMessageDialogShowAsyncCompleted(IAsyncOperation<IUICommand> asyncInfo,
                                           AsyncStatus asyncStatus)
    {
        // Получение значения Color
        IUICommand command = asyncInfo.GetResults();
        clr = (Color)command.Id;

        // Использование Dispatcher для выполнения в UI-потоке
        IAsyncAction asyncAction =
                     this.Dispatcher.RunAsync(CoreDispatcherPriority.Normal,
                     OnDispatcherRunAsyncCallback);
    }

    void OnDispatcherRunAsyncCallback()
    {
        // Назначение фоновой кисти
        contentGrid.Background = new SolidColorBrush(clr);
    }
}
```

Необязательный второй аргумент конструктора `UICommand` задает метод обратного вызова как тип делегата `UICommandInvokedHandler`:

```
void OnMessageDialogCommand(IUICommand command)
{
    ...
}
```

Метод обратного вызова выполняется в UI-потоке. Вероятно, по этой причине он представляет более простой способ получения кнопки, нажатой пользователем.

Методы обратного вызова как лямбда-функции

Потребность в более элегантных средствах работы с методами обратного вызова стала одной из причин, из-за которых в C# 3.0 добавилась поддержка анонимных методов,

также называемых «лямбда-функциями» или «лямбда-выражениями». Всю логику обратного вызова в HowToAsync1 можно переместить в лямбда-функции в обработчике Click, а значение Color не нужно сохранять в поле. Данная возможность продемонстрирована в проекте HowToAsync2:

```
Проект: HowToAsync2 | Файл: MainPage.xaml.cs (фрагмент)
void OnButtonClick(object sender, RoutedEventArgs args)
{
    MessageDialog msgdlg = new MessageDialog("Choose a color", "How To Async #2");
    msgdlg.Commands.Add(new UICommand("Red", null, Colors.Red));
    msgdlg.Commands.Add(new UICommand("Green", null, Colors.Green));
    msgdlg.Commands.Add(new UICommand("Blue", null, Colors.Blue));

    // Отображение MessageDialog с обработчиком Completed
    IAsyncOperation<IUICommand> asyncOp = msgdlg.ShowAsync();
    asyncOp.Completed = (asyncInfo, asyncStatus) =>
        {
            // Получение значения Color
            IUICommand command = asyncInfo.GetResults();
            Color clr = (Color)command.Id;

        // Использование Dispatcher для выполнения в UI-потоке
            IAsyncAction asyncAction =
                        this.Dispatcher.RunAsync(CoreDispatcherPriority.Normal,
                        () =>
                {
                    // Назначение фоновой кисти
                    contentGrid.Background = new SolidColorBrush(clr);
                });
        };
}
```

Хотя все операции были перемещены в один обработчик Click, очевидно, что этот код не выполняется весь одновременно. Обработчик Completed объекта MessageDialog выполняется только после закрытия окна сообщения, а метод обратного вызова класса CoreDispatcher выполняется только тогда, когда поток пользовательского интерфейса доступен для выполнения кода.

Это конкретное сочетание двух лямбда-функций не так плохо, но вложенные лямбда-функции быстро создают путаницу. Например, при файловом вводе/выводе часто приходится последовательно выполнять ряд действий, многие из которых асинхронны. Вложенные лямбда-функции громоздятся друг на друга и начинают скрывать реальную структуру кода. Безусловно, лямбда-функции удобны, но часто они плохо читаются. В некоторых ситуациях лямбда-функции даже могут «вывернуть наизнанку» процедурный код и усложнить выполнение простой команды return или обработку исключения.

Задача отчаянно требует другого решения. К счастью, теперь оно существует.

Замечательный оператор await

Ключевое слово C# 5.0 await позволяет работать с асинхронными операциями так, словно они являются относительно обычными вызовами методов без методов обрат-

ного вызова. Ниже приведен код, который я использую для получения объекта `IAsyncOperation`:

```
IAsyncOperation<IUICommand> asyncOp = msgdlg.ShowAsync();
```

В предыдущих версиях для получения объекта `IUICommand`, обозначавшего нажатую кнопку, использовался метод обратного вызова. Оператор `await` фактически извлекает этот объект `IUICommand` прямо из объекта `IAsyncOperation`:

```
IUICommand command = await asyncOp;
```

Очень часто эти две команды объединяются, как показано в программе `HowToAsync3`, функционально эквивалентной первым двум программам:

Проект: HowToAsync3 | **Файл**: MainPage.xaml.cs (фрагмент)

```
async void OnButtonClick(object sender, RoutedEventArgs args)
{
    MessageDialog msgdlg = new MessageDialog("Choose a color", "How To Async #3");
    msgdlg.Commands.Add(new UICommand("Red", null, Colors.Red));
    msgdlg.Commands.Add(new UICommand("Green", null, Colors.Green));
    msgdlg.Commands.Add(new UICommand("Blue", null, Colors.Blue));

    // Отображение MessageDialog
    IUICommand command = await msgdlg.ShowAsync();

    // Получение значения Color
    Color clr = (Color)command.Id;

    // Назначение фоновой кисти
    contentGrid.Background = new SolidColorBrush(clr);
}
```

Удобно, не правда ли?

Ключевое слово `await` является полноценным оператором C#, и оно может встраиваться в более сложные конструкции. Следующая команда выполняет работу последних трех команд из предыдущего фрагмента:

```
contentGrid.Background = new SolidColorBrush((Color)(await msgdlg.ShowAsync()).Id);
```

Программа `HowToAsync3` функционально идентична двум предыдущим программам. При этом ее синтаксис существенно стройнее, и все это благодаря оператору `await`. Все выглядит так, словно оператор `await` обходит все хлопоты с обратными вызовами и возвращает `IUICommand` напрямую. Немного смахивает на фокус, но большинство громоздких подробностей реализации теперь скрыто. Компилятор C# распознает паттерн с методом `ShowAsync`, генерирует метод обратного вызова и вызов `GetResults`.

По сути, оператор `await` разбивает метод, в котором он используется, и превращает его в конечный автомат. Метод `OnButtonClick` выполняется нормально, пока не будет вызван метод `ShowAsync` и не встретится оператор `await`. Несмотря на свое название, оператор `await` не ожидает завершения операции. Вместо этого в этой точке происходит выход из обработчика `Click`, а управление возвращается Windows. После этого может выполняться другой код в UI-потоке программы, как и `MessageDialog`. Когда `MessageDialog` закроется, будет готов результат, а UI-поток будет готов к выполнению кода, выполнение обработчика `Click` продолжится с присваивания объекту `IUICommand`. Далее метод продолжает выполняться до следующего оператора `await`, если он встретится.

В этом конкретном обработчике `Click` других операторов `await` нет: когда объекту `IUICommand` присваивается значение, код выполняется в потоке пользовательского интерфейса, и диспетчер не нужен.

До появления `await` мне всегда казалось, что асинхронные операции в C# нарушают императивную структуру языка. Оператор `await` возвращает эту структуру и превращает асинхронные вызовы в то, что кажется серией обычных последовательных вызовов методов. Но несмотря на простоту использования, следует помнить, что на самом деле метод, в котором встречается `await`, делится на части с использованием обратных вызовов, которые вам просто не видны.

В некоторых случаях это создает проблемы. Иногда при вызове метода в вашей программе Windows ожидает, что метод завершается при возвращении управления операционной системе. Если метод содержит оператор `await`, это условие не гарантировано. Метод с `await` возвращает управление Windows до выполнения кода, следующего за оператором `await`.

Чтобы сообщить Windows о том, что метод с оператором `await` еще не завершен, используется объект «отсрочки» (deferral). Вы увидите, как это делается, позднее в этой главе, когда мы займемся обработкой события `Suspending` класса `Application`.

При использовании оператора `await` должны соблюдаться некоторые ограничения. Он недопустим в блоках `catch` или `finally` обработчиков исключений, хотя и может находиться в блоке `try`; собственно, именно так перехватываются ошибки, происходящие в асинхронных методах, или выявляется факт отмены асинхронных операций (как вы вскоре увидите).

Метод, в котором находится оператор `await`, должен быть помечен ключевым словом `async`, как обработчик `Click` в следующем примере:

```
async void OnButtonClick(object sender, RoutedEventArgs args)
{
    // ... Код с операторами await
}
```

Ключевое слово `async` само по себе ничего не делает. В предыдущих версиях C# слово `await` не было ключевым, поэтому программисты могли использовать его для имен переменных, свойств и т. д. Добавление нового ключевого слова `await` в C# нарушит работу этого кода, но ограничение `await` методами, помеченными ключевым словом `async`, решает эту проблему. Модификатор `async` не изменяет сигнатуру — метод все равно остается действительным обработчиком `Click`. Однако ключевое слово `async` (а следовательно, и `await`) не может использоваться в методах, служащих точками входа, а конкретно `Main` и конструкторах классов.

Если вам понадобится вызывать асинхронные методы во время инициализации страницы, вызывайте их в обработчике события `Loaded`, пометив его ключевым словом `async`:

```
public MainPage()
{
    this.InitializeComponent();
    ...
    Loaded += OnLoaded;
}

async void OnLoaded(object sender, RoutedEventArgs arg)
```

```
{
    ...
}
```

Или если вы предпочитаете определить обработчик `Loaded` как анонимный метод:

```
public MainPage()
{
    this.InitializeComponent();
    ...
    Loaded += async (sender, args) =>
        {
            ...
        };
}
```

Заметили `async` перед списком аргументов?

Отмена асинхронных операций

Не все асинхронные операции имеют такую четкую структуру, как вызов метода `ShowAsync` класса `MessageDialog`.

У асинхронных операций имеются три особенности, которые часто усложняют их выполнение:

- **Отмена.** Многие асинхронные операции могут отменяться — действием пользователя, намеренно прерывающего затянувшуюся операцию, или другим способом.
- **Оповещения о прогрессе.** При выполнении значительного объема работы асинхронные операции могут передавать информацию о прогрессе. Пользователю спокойнее, когда он получает такую информацию в виде текста или на индикаторе `ProgressBar`.
- **Ошибки.** В ходе выполнения асинхронной операции может возникнуть проблема — например, при попытке открытия несуществующего файла.

Начнем с первого пункта. Отмена окна сообщения — то есть удаление его с экрана до того, как пользователь нажмет какую-либо кнопку — встречается нечасто, но в некоторых ситуациях она имеет смысл.

Интерфейс `IAsyncInfo` (реализуемый четырьмя стандартными асинхронными интерфейсами Windows Runtime) определяет метод с именем `Cancel` для отмены операций. Как упоминалось ранее, интерфейс `IAsyncInfo` также включает свойство `Status`, принимающее значения из перечисления `AsyncStatus`: `Started`, `Completed`, `Canceled` и `Error`. Для последнего случая `IAsyncInfo` также определяет свойство `ErrorCode` типа `Exception`.

Если вы используете механизм обратного вызова для асинхронных операций, обычно в начале метода обратного вызова следует проверить статус операции и перед вызовом `GetResults` убедиться в том, что операция имеет статус `Completed`, а не `Canceled` или `Error`.

Если используется оператор `await`, поместите команду `await` в блок `try`. При отмене асинхронной операции выдается исключение типа `TaskCanceledException`. Если в ходе асинхронной операции происходит ошибка, она будет обозначена в объекте исключения.