# Общие сведения О .NET Framework

.NET Framework представляет собой новую революционную платформу, которая была создана Microsoft для разработки приложений. Эта платформа была создана для разработки приложений исключительно на базе операционной системы Windows. Хотя выпущенная Microsoft версия .NET Framework рассчитана на использование под управлением Windows, очень быстро становятся доступными альтернативные версии. Одним из примеров является версия Mono, которая распространяется с открытым исходным кодом (вместе с компилятором С#) и способна работать на базе нескольких операционных систем, включая некоторые разновидности таких систем, как Linux и Маc. Многие подобные проекты сейчас еще находятся на стадии разработки и в принципе к моменту прочтения настоящей книги уже могут стать доступными. Помимо этого еще существует и версия Microsoft .NET Compact Framework, которая, по сути, представляет собой усеченный вариант полной версии .NET Framework и может применяться на устройствах типа персональных цифровых помощников (personal digital assistant — PDA) и даже в некоторых смартфонах. Одним из главных стимулов к применению платформы .NET Framework является возможность использовать ее в качестве средства интеграции разных операционных систем.

Помимо этого, приведенное выше определение .NET Framework не содержит никаких указаний касательно типов приложений, которые могут создаваться с помощью этой платформы. Объясняется это тем, что никаких ограничений подобного рода не существует: .NET Framework позволяет создавать приложения для Windows, Web-приложения, приложения типа Web-служб и практически все остальные типы приложений.

Платформа .NET Framework была спроектирована так, чтобы ее можно было использовать из любого языка, включая язык С#, а также языки С++, Visual Basic, JScript и даже некоторые более старые языки, вроде COBOL.

## Что входит в состав .NET Framework

Платформа .NET Framework по большей части состоит из гигантской библиотеки кода, который можно использовать из клиентских языков (вроде С#) путем применения различных приемов объектно-ориентированного программирования (Object-Oriented Programming), или ООП. Эта библиотека поделена на модули, которые применяются в зависимости от того, какие результаты требуется получить. Например, в одном модуле содержатся компоновочные блоки для приложений Windows, в другом — для программирования сетевого обмена, в третьем — для разработки Web-прило­жений. Некоторые из модулей содержат более специфические подмодули, вроде модуля для разработки Web-приложений, который содержит подмодуль для написания Web-служб.

По замыслу разные операционные системы должны поддерживать некоторые или все эти модули, в зависимости от их характеристик. Устройство PDA, например, будет включать поддержку для всех ключевых функциональных возможностей .NET, но вряд ли будет нуждаться в наличии более специфических модулей.

В одном из разделов библиотеки .NET Framework содержатся определения ряда базовых *типов.* Типы отвечают за способ представления данных, и указание некоторых наиболее фундаментальных из них (например, типа "32-битное целое число со знаком") способствует функциональной совместимости между языками, использующими .NET Framework. Это носит название *системы общих типов* (Common Type System — CTS).

Помимо библиотеки в состав .NET Framework еще также входит и так называемая *общеязыковая исполняющая среда .NET* (Common Language Runtime — CLR), которая отвечает за обслуживание процесса выполнения всех тех приложений, которые разрабатываются с помощью библиотеки .NET.

## Написание приложений с помощью .NET Framework

Под написанием приложения с помощью .NET Framework подразумевается просто написание кода с использованием одного из языков, поддерживающих .NET Framework, и библиотеки кода NET.

Чтобы код на языке С# мог выполняться, он обязательно должен преобразовываться в код на том языке, который понимает целевая операционная система. Такой код называется *родным кодом* (native code), а сам процесс преобразования — *компиляцией.* Компиляция выполняется механизмом, который называется *компилятором,* и в .NET Framework состоит из двух этапов.

## Язык MSIL и JIТ-компилятор

При компиляции кода, в котором используется библиотека .NET, он не преобразовывается сразу же в родной код конкретной операционной системы. Вместо этого он сначала преобразуется в код *MSIL* (Microsoft Intermediate Language — промежуточный язык Microsoft). Этот код не является специфическим ни для какой-либо операционной системы, ни для языка С#. Другие языки, например, Visual Basic .NET, на первом этапе тоже компилируются в код на этом языке. В случае разработки приложений на С# такой процесс компиляции выполняется VS

Очевидно, что далее для запуска приложения требуется выполнить еще кое-какую работу. За это отвечает так называемый *JIT-компилятор* (Just-in-Time compiler — оперативный компилятор), который компилирует MSIL в родной код, отвечающий требованиям конкретной операционной системы и архитектуры целевого компьютера. Только после этого этапа у операционной системы появляется возможность запустить приложение. Аббревиатура *JIT* в названии компилятора указывает на то, что он выполняет компиляцию MSIL-кода только при возникновении соответствующей необходимости.

Раньше код часто компилировался в несколько приложений, каждое из которых предназначалось для конкретной операционной системы и архитектуры ЦП. Такой подход обычно применялся в качестве одного из средств оптимизации (например, для принуждения кода работать быстрее на базе микросхем AMD), но в некоторых случаях играл даже критическую роль (например, при необходимости сделать так, чтобы приложения могли работать как в средах Win9x, так и в средах WinNT/2000). Сейчас в нем нет никакой необходимости, поскольку JIT-компиляторы используют MSIL-код, который не зависит ни от типа компьютера, ни от типа операционной системы, ни от типа ЦП. Существуют несколько JIT-компиляторов, каждый из которых рассчитан на конкретную архитектуру, и при создании требуемого родного кода применяться будет только тот из них, который подходит в данном случае.

Вся прелесть этого механизма состоит в том, что он требует приложения гораздо меньшего количества усилий со стороны разработчика: на самом деле он даже позволяет ему вообще не думать о касающихся специфики системы деталях и концентрировать все внимание на более интересных функциональных возможностях кода.

## Сборки

При компиляции приложении создаваемый MSIL-код сохраняется в *сборке* (assembly). С состав сборок входят как исполняемые файлы приложений, которые имеют расширение .**ехе** и могут запускаться прямо в среде Windows безо всяких других программ, так и файлы библиотек, которые имеют расширение .**dll** и предназначены для использования другими приложениями.

Помимо MSIL-кода в сборки еще также включается *метаинформация* (т.е. информация о содержащихся в сборке данных, также называемая *метаданными)* и файлы необязательных *ресурсов* (файлы дополнительных данных, которые могут применяться в MSIL коде, вроде звуковых файлов и файлов изображений). Метаинформация делает сборки полностью самоописательными. Благодаря ей, для использования сборок больше никакой информации не требуется, а это означает исключение вероятности возникновений ситуаций, вроде невозможности добавления требуемых данных в системный реестр и тому подобного, каковые часто представляли проблему при выполнении разработки с помощью других платформ.

Из-за этого развертывание приложений часто сводится просто к копированию файлов в каталог на удаленном компьютере. Поскольку никакой дополнительной информации на целевых системах не требуется, далее пользователь может просто запускать исполняемый файл из этого каталога и (при условии, что в системе установлена CLR-среда .NET) приступить к работе с приложением.

Конечно, сохранять все необходимое для запуска приложения в одном месте вовсе необязательно. Разработчик может писать и какой-нибудь код, выполняющий задачи, требуемые нескольким приложениям. В подобных ситуациях зачастую удобнее помещать допускающий многократное использование код в то место, к которому смогут получать доступ все приложения. В .NET Framework таким местом является *глобальный кэш сборок* (Global Assembly Cache — GAC). Помещение в него кода осуществляется очень просто, а именно — копированием содержащей нужный код сборки в каталог, где расположен этот кэш.

## Управляемый код

Роль CLR-среды не заканчивается после компиляции кода в MSIL-код и преобразования его JIT-компилятором в родной код. Код, который пишется с помощью .NET Framework, при выполнении (т.е. на этапе, который обычно называется *временем выполнения)* находится *под управлением* CLR-среды. Это означает, что CLR-среда следит за приложениями, осуществляя необходимое управления памятью, обеспечивая безопасность, позволяя выполнять межъязыковую отладку и т.д. Приложения, которые во время выполнения не попадают под контроль CLR-среды, называются *неуправляемыми* (unmanaged), и некоторые языки могут применяться для написания таких приложений, например, с целью доступа к низкоуровневым функциям операционной системы. В языке С#, однако, разрешается писать только код, выполняющийся в управляемой среде, т.е. использовать управляемые средства CLR и позволять среде .NET самостоятельно обрабатываться любые операции взаимодействия с операционной системой.

## Сборка мусора

Одной из наиболее важных функциональных возможностей управляемого кода является средство *сборки мусора* (garbage collection). Оно гарантирует в .NET уверенность в полном очищении использованной приложением памяти после завершения его эксплуатации. До появления .NET об этом по большей части нужно было заботиться самим программистам, и даже несколько простых ошибок в коде могли приводить к таинственному исчезновению крупных блоков памяти в результате их выделения в неправильном месте. Это обычно выливалось в постепенное замедление работы компьютера и в конечном итоге завершалось полным крахом системы.

Механизм сборки мусора в .NET функционирует следующим образом: он инспектирует память компьютера время от времени и удаляет из нее все, что уже больше не нужно. Никаких установленных временных рамок для выполнения этой операции нет; она может происходить как тысячу раз в секунду, так и каждые несколько секунд или через любой другой промежуток времени, но в одном можно быть уверенным точно — она обязательно произойдет.

Для программистов это означает появление кое-каких последствий. Из-за того, что данная операция выполняется автоматически через непредсказуемые промежутки времени, необходимо разрабатывать приложения с учетом этого факта. Код, требующий большого количества памяти для выполнения, должен осуществлять очистку после себя самостоятельно, а не ожидать того, когда ее выполнит механизм сборки мусора.

## Что собой представляет язык С#

Язык С# является одним из тех, которые можно использовать для создания приложений, подлежащих запуску в .NET CLR. Он эволюционировал из языков С и С++ и был создан Microsoft специально для работы с платформой .NET. Благодаря тому, что язык С# разрабатывался недавно, в его состав вошли многие из наилучших функциональных возможностей других языков, но без присущих им проблем.

Разрабатывать приложения с помощью языка С# легче, чем с помощью С++, потому что синтаксис этого языка является более простым. Однако при этом С# все равно остается мощным языком, и существует очень мало вещей, которые может потребоваться делать на С++ из-за того, что их нельзя сделать на С#. Несмотря на это, те функциональные возможности языка С#, которые предлагаются параллельно более усовершенствованным функциональным возможностям языка С++, вроде прямого получения доступа к системной памяти и манипулирования ею, могут реализоваться только применением кода с пометкой **unsafe.** Этот усовершенствованный прием программирования является потенциально опасным (о чем и говорит ключевое слово **unsafe),** поскольку существует вероятность перезаписывания критических для системы блоков памяти.

Иногда код на С# является немного более многословным, чем на С++. Объясняется это тем, что С# (в отличие от С++) представляет собой *безопасный к типам* (type-safe) язык. Это означает, что после присваивания типу каких-то данных он уже далее не может преобразовываться в другой несвязанный тип. Следовательно, к преобразованию между типами должны применяться строгие правила, из-за чего для выполнения той же самой задачи на С# часто требуется писать больше кода, чем на С++, но, тем не менее, это дает определенные преимущества: код получается более надежным, процесс отладки — более простым, a .NET может всегда в любое время определить тип того или иного фрагмента данных. Таким образом, в С# невозможно делать, например, следующее: взять область памяти размером 4 байта под данные длинной 10 байтов и интерпретировать это как X; однако это вовсе не является недостатком.

Язык С# представляет собой лишь один из многих языков, доступных для разработки .NET-приложений. Его главное преимущество состоит в том, что он единственный с самого начала разрабатывался специально для .NET Framework и из-за этого может быть главным кандидатом на использование в тех версиях .NET, которые переносятся на другие операционные системы. Чтобы языки вроде .NET-версии Visual Basic оставались насколько возможно похожими на своих предшественников и при этом совместимыми с CLR, некоторые средства из библиотеки кода .NET в них поддерживаются не полностью. В отличие от них, С# позволяет пользоваться каждым из тех средств, которые предлагаются в этой библиотеке. В состав последней версии .NET было добавлено несколько улучшений для языка С#, частично по просьбам конечных разработчиков, что существенно увеличивает его мощность.

## Приложения, которые можно писать на С#

В .NET Framework не существует никаких ограничений касательно того, приложения какого типа можно создавать, как уже рассказывалось ранее. С# использует .NET Framework и потому тоже не имеет никаких ограничений касательно возможных типов приложений. Однако ниже перечислены некоторые наиболее распространенные из них.

**□ Приложения Windows.** К приложениям этого типа относятся приложения, которые имеют знакомый пользователям Windows внешний вид и поведение, как, например, приложение Microsoft Office. Достигается такой внешний вид и поведение применением модуля .NET Framework под названием Windows Forms (Формы Windows), по сути, представляющего собой библиотеку *элементов управления* (вроде кнопок, панелей инструментов, меню и т.п.), с помощью которых можно создавать пользовательский интерфейс Windows (UI).

* **Web-приложения.** К приложениям этого типа относятся Web-страницы, вроде тех, что могут просматриваться посредством любого Web-браузера. В состав .NET Framework входит мощная система для генерации Web-содержимого динамическим образом, обеспечивающая персонализацию, безопасность и мно­гое другое. Называется она [ASP.NET](http://ASP.NET) (Active Server Pages .NET), и благодаря ей, язык С# можно использовать для создания приложений [ASP.NET](http://ASP.NET) с применением Web-форм.
* **Web-службы.** Этот тип является новым и интересным способом для создания разнообразных распределенных приложений. С помощью Web-служб через Internet можно обмениваться практически любыми данными, используя тот же самый простой синтаксис, независимо от того, какой язык применялся для создания Web-службы, и того, на какой системе она расположена.

Приложениям любого из этих типов может также требоваться доступ к базам данным, обеспечиваться который может либо с помощью такого средства .NET Framework, как [ADO.NET](http://ADO.NET) (Active Data Object .NET), либо с помощью такого средства самого языка С#, как LINQ (Language Integrated Query — язык интегрированных запросов). Помимо этого еще могут быть задействованы и многие другие ресурсы, а именно — инструменты для создания сетевых компонентов, вывода графических объектов, выполнения сложных математических операций и т.д.

# Создание консольного приложения C#

Для создания консольного приложения на языке С# нужно выполнить следующие действия.

1. В меню **Файл** выберите команду **Создать проект**. Откроется диалоговое окно **Создание проекта**. В этом диалоговом окне выводится список различных типов приложений по умолчанию, которые можно создавать с помощью Visual C#, экспресс-выпуск.
2. В качестве типа проекта выберите **Консольное приложение** и измените имя приложения на **Список файлов**. Можно оставить расположение по умолчанию или указать новый путь по своему усмотрению.
3. Нажмите кнопку **ОК**. Visual C#, экспресс-выпуск создаст новую папку для проекта с таким же именем, как у проекта. Откроется основное окно Visual C#, экспресс-выпуск и панель кода для ввода и изменения исходного кода приложения C#.

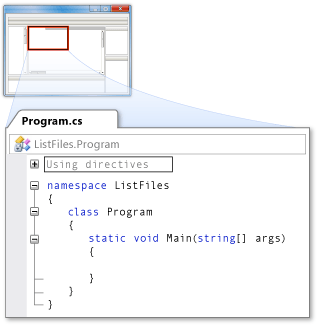


Рис 1.1 Создание консольного приложения

Панель инструментов содержит значки для создания, загрузки и сохранения проектов, редактирования исходного кода, построения приложения, скрытия и отображения других окон, являющихся частью среды Visual C#, экспресс-выпуск. На правом краю панели инструментов находятся пять значков для открытия важных окон, таких как **Обозреватель решений** и **Панель элементов**. При наведении указателя мыши на любой из этих значков появляется всплывающая подсказка.

1. Убедитесь, что отображается **Обозреватель решений**, щелкнув вкладку **Обозреватель решений** в правой части экрана или значок **Обозреватель решений** в панели инструментов. Панель **Обозреватель решений** является очень полезной, на ней отображаются различные файлы, являющиеся частью проекта. Самым важным файлом проекта является "Program.cs", который содержит исходный код приложения.

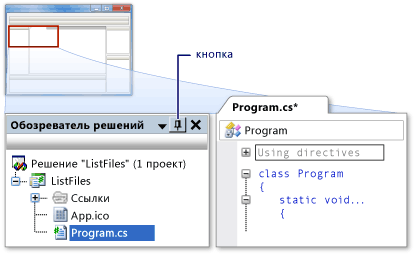


Рис 1.2 Обозреватель решений

1. Введите имя класса **Console** в **редакторе кода**. Если **обозреватель решений** по прежнему скрывает панель **Код**, щелкните панель **Код** для его скрытия. Теперь щелкните правую открытую фигурную скобку ({) внутри метода Main и нажмите ВВОД для перехода на новую строку. Обратите внимание, что редактор делает отступы автоматически.

При вводе имени класса или ключевого слова C# имеется выбор: либо закончить ввод слова самостоятельно, либо позволить сделать это встроенному в панель **Код** инструменту IntelliSense. Например, при вводе символа "c" появится всплывающий список слов, предлагаемых IntelliSense, из которого можно выбрать набираемое слово. В этом случае, слово "Console" пока еще не видно. Либо прокрутите вниз список, либо продолжайте вводить слово "console". Когда слово "console" будет выделено в списке, нажмите клавишу ВВОД, илиTAB, или дважды щелкните его мышью, **Console** будет добавлено в код.

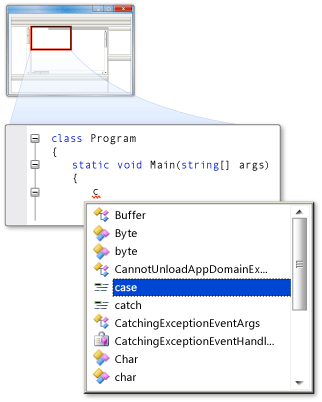


Рис 1.3 Использование IntelliSense

1. Введите точку и имя метода **WriteLine**. Как только пользователь введет точку после **Console**, IntelliSense отобразит другой список. Список содержит имена всех возможных методов и свойств, являющихся частью класса **Console**. Необходимый метод **WriteLine** можно увидеть в конце списка. Либо закончите ввод слова **WriteLine**, либо нажмите клавишу СТРЕЛКА ВНИЗ для выбора, затем ВВОД, или TAB, или дважды щелкните его мышью. Метод **WriteLine** будет добавлен в код.

ExpressConsole

Рис 1.4 Использование IntelliSense

Введите открывающую скобку. Появится сообщение в виде всплывающей подсказки с подписями методов, являющееся еще одной функцией IntelliSense. В этом случае будет отображено 19 различных подписей, которые можно просмотреть с помощью клавиш СТРЕЛКА ВВЕРХ и СТРЕЛКА ВНИЗ.

1. Введите строку "Привет мир!!!" Введите сообщение в двойных кавычках и добавьте закрывающую скобку. Появится красное волнистое подчеркивание, напоминающее о том, что чего-то не хватает. Введите точку с запятой (;) и подчеркивание исчезнет. Добавьте в программу строчку **Console.ReadLine();** Она предназначена для приостановки выполнения программы до нажатия клавиши ВВОД. Если не добавить эту строку, окно с командной строкой сразу исчезнет и пользователь не сможет увидеть вывод результатов выполнения. Если создаваемая служебная программа всегда будет использоваться в консоли командной строки, вызов метода **ReadLine()** можно опустить.
2. Выполните программу. Теперь программа завершена, готова для компиляции и выполнения. Для этого либо нажмите клавишу F5, либо щелкните значок **Пуск** в панели инструментов.Запуск файлов списка VJS ExpressПосле компиляции и выполнения откроется окно **Консоль** с отображением списка файлов и их размеров. Нажмите клавишу ВВОД для выхода из программы.

# Основы С#

**"Hello World" в стиле C#**

Для упорядочения и оформления кода в языке C# используются классы. В действительности весь выполняемый код C# должен содержаться в классе, что справедливо и для короткой программы типа "Hello World!". Ниже приведен полный текст программы, отображающей в окне консоли сообщение "Hello World!".

Листинг программы "Hello World!"

|  |  |
| --- | --- |
| 001:  002:  003:  004:  005:  006:  007:  008:  009:  010:  011:  012: | using System;  // A "Hello World!" program in C#  namespace HelloWorld  {  class Hello  {  static void Main()  {  System.Console.WriteLine("Hello World!");  }  }  } |

## Структура программы на С#

### Директивы using и пространства имен

При создании консольного приложения с Visual C#, экспресс-выпуск, в первой линии в редакторе кода содержится директива using с перечислением нескольких пространств имен .NET Framework. Пространство имен позволяет, в некотором смысле, сгруппировать вместе классы и структуры, что ограничивает их область действия и позволяет избежать конфликта имен с другими классами и структурами. При создании программы в Visual C# Express пространство имен создается автоматически. Для использования в программе классов из других пространств имен необходимо указать их с директивой using. При создании нового приложения наиболее часто используемые пространства имен .NET Framework включены в список по умолчанию. При использовании классов из других пространств имен в библиотеке классов необходимо добавить директиву using для пространства имен к исходному файлу.

Когда редактор кода определит, что объявленный класс или структура не встречается в пространствах имен из текущего списка директивы **using**, он предложит пространство имен, содержащее этот класс или структуру.

Примечания

Следующая после оператора **using** строка содержит комментарий. Комментарии являются полезными для включения в них примечаний для себя или других программистов.

// A "Hello World!" program in C#

Символы // преобразуют остальную часть строки в комментарий. Можно также закомментировать блок текста, поместив его между символами /\* и \*/.

### Классы

В C# классы используются для оформления кода: весь выполняемый код C# должен содержаться в классе.

### Main()

В программе на C# должен присутствовать метод **Main**, в котором начинается и заканчивается управление. В методе **Main** создаются объекты и выполняются другие методы. Метод **Main** является статическим методом, расположенным внутри класса или структуры. В предыдущем примере "Hello World!" он расположен в классе с именем **Program** .

Метод **Main** можно объявить одним из следующих способов:

Он возвращает значение void.

static void Main()

{

//...

}

Он также может возвращать значение типа int.

static int Main()

{

//...

return 0;

}

Он может принимать аргументы, что может быть полезно для программ командной строки.

static void Main(string[] args)

{

//...

}

или

static int Main(string[] args)

{

//...

return 0;

}

Параметр метода **Main** является массивом значений типа string, представляющим аргументы командной строки, используемые для вызова программы. Обратите внимание, что в отличие от C++, массив не содержит имя исполняемого (EXE) файла.

### Консольный ввод и вывод

Программы консоли на C# обычно используют службы ввода/вывода, предоставляемые классом .NET Framework Console. Оператор **Console.WriteLine("Hello, World!");** использует метод **WriteLine**. Он выводит свой строковый параметр в окно командной строки с новой строки. Другие методы **Console** используются для различных операций ввода и вывода. Класс **Console** является членом пространства имен **System.** Если в начало программы поместить директиву **using System**;, то нужно будет указать классы **System,** как в следующем примере.

System.Console.WriteLine("Hello World!");

Метод **WriteLine** очень полезен и часто используется при создании консольных приложений.

Метод **WriteLine** может отображать строки.

Console.WriteLine("Hello World!");

Метод **WriteLine** может также отображать числа.

int x = 42;

Console.WriteLine(x);

Если необходимо отобразить несколько элементов, используйте {0} для представления первого элемента, {1} — для второго элемента и так далее, следующим образом.

int year = 1066;

string battle = "Battle of Hastings";

Console.WriteLine("The {0} took place in {1}.", battle, year);

Результат будет выглядеть следующим образом.

The Battle of Hastings took place in 1066.

### Пространство имен

Пространства имен представляют собой способ организации различных типов, присутствующих в программах C#. Их можно сравнить с папкой в компьютерной файловой системе. Подобно папкам, пространства имен определяют для классов уникальные полные имена. Программа C# содержит одно или несколько пространств имен, каждое из которых либо определено программистом, либо определено как часть написанной ранее библиотеки классов.

Например, пространство имен System содержит класс Console, который включает методы для чтения и записи в окне консоли. В пространство имен System входит ряд других пространств, например System.IO и System.Collections. Только в платформе .NET Framework представлено более 80 пространств имен, в каждом из которых находятся тысячи классов, поскольку пространства имен используются для того, чтобы избежать путаницы с типами и методами, имеющими одинаковые имена.

При написании класса вне объявления пространства имен компилятор предоставит ему заданное по умолчанию пространство имен.

### Ускоренные методы для пространства имен

Для использования метода WriteLine, определенного в классе Console, который содержится в пространстве имен System, следует использовать строку кода, как показано далее.

System.Console.WriteLine("Hello, World!");

Необходимость помнить, что всем методам, содержащимся в Console, должно предшествовать **System** быстро становится утомительной, поэтому в начало исходного файла C# целесообразно вставить директиву using, как показано далее.

using System;

Добавление **using** **System**; устанавливает, что предполагается пространство имен System и впоследствии можно написать следующее.

Console.WriteLine("Hello, World!");

### Создание собственного пространства имен

Обычно пространства имен используются при работе с большими программами. Собственные пространства имен предоставляют определенный уровень контроля над методами и типами с одинаковыми именами. Например, предположим, что выполняется написание приложения, загружающего с диска статистические данные и файлы изображений. Можно создать два новых пространства имен, одно с именем **Images**, а другое — **StatisticalData**. Поскольку используются два отдельных пространства, все имена методов, определенные в каждом из них, будут уникальными, даже если отдельные классы имеют одинаковые имена. Это значит, что в обоих пространствах имен будет находиться класс с именем **FileHandling** и метод с именем **Load**. Для указания нужного класса можно обратиться к **StatisticalData**.**FileHandling** или **Images**.**FileHandling**.

Для каждого пространства имен в проекте Visual C# Express рекомендуется создать отдельную папку.

Пример

В следующем примере определяются два пространства имен, в каждом их которых содержится класс с именем **FileHandling**. Указав пространство имен, можно быстро отличить классы от методов.

|  |  |
| --- | --- |
| 001:  002:  003:  004:  005:  006:  007:  008:  009:  010:  011:  012:  013:  014:  015:  016:  017:  018:  019:  020:  021:  022:  023:  024:  024:  025:  024:  024:  024: | namespace StatisticalData  {  class FileHandling  {  public void Load() {}  // code to load statistical data  }  }  namespace Images  {  class FileHandling  {  public void Load() {}  // code to load an image file  }  }  class Program  {  static void Main()  {  StatisticalData.FileHandling data = new  StatisticalData.FileHandling();  data.Load();  Images.FileHandling image = new Images.FileHandling();  image.Load();  }  } |

## Переменные и константы

### Переменные

Переменная представляет числовое или строковое значение или объект класса. Значение, хранящееся в переменной, может измениться, однако имя остается прежним. Переменная представляет собой один тип **поля**. Следующий код является простым примером объявления целочисленной переменной, присвоения ей значения и последующего присвоения нового значения.

Объявления

int x = 1; // x holds the value 1

x = 2; // now x holds the value 2

В C# переменные объявляются с определенным типом данных и надписью. Если ранее вам приходилось работать со слабо типизированными языками, например JScript, вы привыкли использовать один тип "var" для всех переменных, то в C# необходимо указать тип переменной: int, float, byte, short или другой из более чем 20 различных типов данных. Тип указывает, помимо всего прочего, точный объем памяти, который следует выделить для хранения значения при выполнении приложения. При преобразовании переменной из одного типа в другой язык C# следует определенным правилам

int answer = 42;

string greeting = "Hello, World!";

double bigNumber = 1e100;

System.Console.WriteLine("{0} {1} {2}", answer, greeting, bigNumber);

### Константы

Константа является другим типом поля. Она хранит значение, присваиваемое по завершении компиляции программы, и никогда после этого не изменяется. Константы объявляются помощью ключевого слова const; их использование способствует повышению удобочитаемости кода.

const int speedLimit = 55;

const double pi = 3.14159265358979323846264338327950;

Переменная **readonly** аналогична константе, однако значение ей присваивается при запуске программы. Это дает возможность задать значение на основе каких-либо других условий, неизвестных до выполнения программы. Однако после первого присваивания значение не может быть снова изменено пока выполняется программа.

## Встроенные типы данных

C# является строго типизированным языком. Тип переменной должен быть определен до присвоения ей значения, как показано в следующих примерах.

int a = 1;

string s = "Hello";

XmlDocument tempDocument = new XmlDocument();

Обратите внимание, что тип должен быть определен как для простых встроенных типов, таких как int, так и для сложных или пользовательских типов, таких как XmlDocument.

C# поддерживает следующие встроенные типы данных.

|  |  |
| --- | --- |
| Тип данных | Диапазон |
| byte | 0 .. 255 |
| sbyte | -128 .. 127 |
| short | -32,768 .. 32,767 |
| ushort | 0 .. 65,535 |
| int | -2,147,483,648 .. 2,147,483,647 |
| uint | 0 .. 4,294,967,295 |
| long | -9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807 |
| ulong | 0 .. 18,446,744,073,709,551,615 |
| float | -3,402823e38 .. -3,402823e38 .. |
| double | -1,79769313486232e308 .. 1,79769313486232e308 |
| decimal | -79228162514264337593543950335 .. 79228162514264337593543950335 |
| char | Символ Юникода. |
| string | Строка символов Юникода. |
| bool | true или false |
| object | Объект. |

Эти типы данных называются псевдонимами для предопределенных типов в пространстве имен System. Все эти типы, за исключением object и string, являются типами значений.

### Использование встроенных типов данных

Встроенные типы данных используются в программах C# несколькими способами.

Как переменные.

int answer = 42;

string greeting = "Hello, World!";

Как константы.

const int speedLimit = 55;

const double pi = 3.14159265358979323846264338327950;

Как возвращаемые значения и параметры.

long CalculateSum(int a, int b)

{

long result = a + b;

return result;

}

Можно также определять собственные типы данных, используя [Классы](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.ru/dv_csexpresscon/html/a94e71ac-71b3-432a-8070-09e7b314ff41.htm), [Перечисления](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.ru/dv_csexpresscon/html/a4a6c0b6-878e-429d-9d88-61bb16cf1a4a.htm) или [Структуры](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.ru/dv_csexpresscon/html/823e7974-101c-4d89-a778-5930fcfa8bf6.htm).

### Преобразование типов данных

Преобразование между типами данных можно сделать неявно, в этом случае преобразование автоматически выполняет компилятор, или явно с помощью приведения, в этом случае принудительное преобразование выполняет программист, учитывая риск возможной потери данных.

Пример.

int i = 0;

double d = 0;

i = 10;

d = i; // An implicit conversion

d = 3.5;

i = (int) d; // An explicit conversion, or "cast"

### Типы значений и ссылочные типы

В отличие от некоторых языков программирования, C # имеет две разновидности типов данных: для значения и для ссылки. Если производительность приложения имеет существенное значение или есть заинтересованность в том, как C# управляет данными и памятью, важно знать различия между этими типами.

Если в объявлении переменной используется один из основных встроенных типов данных или определенная пользователем структура данных, значит мы имеем дело с типом значения. Исключение составляет тип данных **string**, который является ссылочным типом.

Тип значения хранит свое содержимое в памяти, выделенной в стеке. Например, значение 42 в таком случае хранится в области памяти, называемой стеком.

int x = 42;

При выходе переменной x из области действия в связи с завершением выполнения метода, в котором она была объявлена, значение удаляется из стека.

Использование стека является эффективным, но ограниченное время существования типов значений делает их менее подходящими для совместного использования данных между различными классами.

В отличие от этого ссылочный тип, такой как экземпляр класса или массив, размещается в другой области памяти, называемой кучей. В следующем примере пространство, необходимое для массива из десяти целых чисел, размещается в куче.

int[] numbers = new int[10];

Эта память не возвращается к куче при завершении метода, она освобождается только когда система сборки мусора C# определит, что она больше не нужна. Объявление ссылочных типов увеличивает расход ресурсов, но их преимущество заключается в том, что они могут быть доступны из других классов.

### Упаковка и распаковка

Упаковкой называется процесс преобразования типа значения в ссылочный тип. Для упаковки переменной необходимо создать ссылочную переменную, указывающую на новую копию в куче. Ссылочная переменная является объектом, следовательно для нее можно использовать все методы, наследуемые каждым объектом, например **ToString().** В следующем коде показано, как это происходит.

int i = 67; // i is a value type

object o = i; // i is boxed

System.Console.WriteLine(i.ToString()); // i is boxed

Распаковка применяется для классов, предназначенных для работы с объектами: например, использование ArrayList для хранения целых чисел. Для хранения целых чисел в ArrayList используется упаковка. При извлечении целого числа должна быть применена распаковка.

System.Collections.ArrayList list =

new System.Collections.ArrayList(); // list is a reference type

int n = 67; // n is a value type

list.Add(n); // n is boxed

n = (int)list[0]; // list[0] is unboxed

### Проблемы производительности

Давайте рассмотрим ситуацию более детально. При передаче данных методу как параметров типов значений копия каждого параметра создается в стеке. Очевидно, если параметр, о котором идет речь, является крупным типом данных, таким как пользовательская структура с множеством элементов, или метод выполняется много раз, это может негативно сказаться на производительности.

В таких ситуациях рекомендуется передавать ссылку на тип, используя ключевое слово **ref**. В C# это эквивалентно способу передачи указателя в переменную в функцию в С++. Как и в случае с C++, метод имеет возможность изменить содержимое переменной, что не всегда может быть безопасным. Программист должен находить компромиссное соотношение между безопасностью и производительностью.

int AddTen(int number) // parameter is passed by value

{

return number + 10;

}

void AddTen(ref int number) // parameter is passed by reference

{

number += 10;

}

Ключевое слово **out** имеет сходство с ключевым словом **ref**, но оно указывает компилятору, что метод должен присвоить значение параметру, иначе возникнет ошибка компиляции.

void SetToTen(out int number)

{

// If this line is not present, the code will not compile.

number = 10;

}

## Операторы

Синтаксис операторов в C# сходен с синтаксисом других языков программирования в стиле языка C. Операторы используются для выполнения вычислений, назначения значений, проверки на равенство и неравенство и т. д.

### Операторы равенства и назначения

В C# оператор знака равенства (=) имеет ту же функциональность, что и в C и C++.

|  |  |
| --- | --- |
| Оператор | Назначение |
| = | Присваивание значения. |
| == | Проверка на равенство. |

Пример

int x = 100;

if (x == 100)

{

System.Console.WriteLine("X is equal to 100");

}

### Логические и математические операторы

Далее представлен список основных математических операторов, указанных в порядке приоритета. Для упорядочения по другим принципам используйте скобки.

|  |  |
| --- | --- |
| Оператор | Назначение |
| \*, /, % | Умножение, деление, остаток от деления |
| +, - | Сложение, вычитание |
| & | Логическое И |
| ^ | Логическое исключающее ИЛИ |
| | | Логическое ИЛИ |

Пример

int x = 1;

int y = x + 10 \* 100; // multiplication first y = 1001

int z = (x + 10) \* 100; // addition first z = 1100

### Операторы увеличения и уменьшения

Поддерживаются сочетания клавиш в стиле языков C/C++, включая постфиксные и префиксные операторы, как показано в следующих примерах.

|  |  |
| --- | --- |
| Оператор | Назначение |
| v ++ | Увеличение переменной v на 1. |
| v += n | Увеличение переменной v на n. |
| v \*= n | Умножение переменной v на n. |
| v -= n | Вычитание n из переменной v. |

Пример

int x = 0;

int y = x++; // x is 1, y is 0

System.Console.WriteLine("{0} {1}", x, y);

int z = ++x; // x is 2, z is 2

System.Console.WriteLine("{0} {1}", x, z);

### Реляционные операторы

Следующие операторы сравнивают два значения и возвращают логический результат.

|  |  |
| --- | --- |
| Оператор | Назначение |
| == | Проверка на равенство. |
| != | Проверка на неравенство. |
| > | Больше. |
| < | Меньше. |
| >= | Больше или равно. |
| <= | Меньше или равно. |

Пример

int x = int.Parse(System.Console.ReadLine());

if (x > 100)

{

System.Console.WriteLine("X is greater than 100");

}

**Логические условные операторы**

Логические операторы используются для создания более гибких условных инструкций путем объединения нескольких предложений.

|  |  |
| --- | --- |
| Оператор | Назначение |
| && | Условное И. |
| || | Условное ИЛИ. |
| ! | Условное НЕТ. |

Пример

int x = int.Parse(System.Console.ReadLine());

if ((x >= 100) && (x <= 200))

{

System.Console.WriteLine("X is between 100 and 200");

}

### Несколько дополнительных математических операторов

Для выполнения более сложных математических операций, например в тригонометрии, используется класс платформ Math. В этом примере используются методы Sin (вычисление синуса) и Sqrt (вычисление квадратного корня) и константа PI.

Пример

double d = System.Math.Sin(System.Math.PI/2);

double e = System.Math.Sqrt(144);

В следующей таблице представлен список операторов C#, сгруппированных в порядке приоритета. Операторы в каждой группе имеют одинаковый приоритет.

|  |  |
| --- | --- |
| Категория оператора | Операторы |
| Основные | x.y  f(x)  a[x]  x++  x--  new  typeof  checked  unchecked  -> |
| Унарные | +  -  !  ~  ++x  --x  (T)x  true  false  &  sizeof |
| Мультипликативные | \*  /  % |
| Аддитивные | +  - |
| Сдвиг | <<  >> |
| Относительные и тестирующие тип | <  >  <=  >=  is  as |
| Операторы равенства | ==  != |
| Логическое AND | & |
| Логическое XOR | ^ |
| Логическое OR | | |
| Условное AND | && |
| Условное OR | || |
| Условные | ?: |
| Операторы присваивания | =  +=  -=  \*=  /=  %=  &=  |=  ^=  <<=  >>=  ?? |

## Решения и ветвление

Изменение потока управления в программе в ответ на какой либо ввод или вычисляемое значение является важной частью языка программирования. C# предоставляет возможность изменить поток управления либо безусловным способом — за счет перехода к новому расположению в коде, либо условным — за счет выполнения теста.

### Оператор if

В самой простой форме условного ветвления используется конструкция **if.** Предложение **else** можно использовать с конструкцией **if**, а конструкции **if** могут быть вложенными.

|  |  |
| --- | --- |
| 001:  002:  003:  004:  005:  006:  007:  008:  009:  010:  011:  012:  013:  014:  015:  016:  017:  018:  019:  020:  021:  022:  023:  024:  025: | using System;  class Program  {  static void Main()  {  int x = 1;  int y = 1;  if (x == 1)  Console.WriteLine("x == 1");  else  Console.WriteLine("x != 1");  if (x == 1)  {  if (y == 2)  {  Console.WriteLine("x == 1 and y == 2");  }  else  {  Console.WriteLine("x == 1 and y != 2");  }  }  }  } |

Операторы, следующие за ключевыми словами **if** и **else**, могут представлять одну строку кода, как показано в первом операторе **if-else** в предыдущем примере кода, или блок операторов, заключенных в скобки, как показано во втором операторе **if-else**. Вложение операторов **if-else** является допустимым, однако в программировании лучшим способом считается использование оператора **switch**.

### Оператор switch

В зависимости от значения заданного выражения оператор switch может выполнять несколько действий. Код между оператором **case** и ключевым словом break выполняется при соблюдении данного условия. Если требуется, чтобы поток управления продолжался в другом операторе **case**, используйте ключевое слово **goto**.

|  |  |
| --- | --- |
| 001:  002:  003:  004:  005:  006:  007:  008:  009:  010:  011:  012:  013:  014:  015:  016:  017:  018:  019:  020:  021:  022:  023:  024:  025: | using System;  class Program  {  static void Main()  {  int x = 3;  switch (x)  {  case 1:  Console.WriteLine("x is equal to 1");  break;  case 2:  Console.WriteLine("x is equal to 2");  break;  case 3:  goto default;  default:  Console.WriteLine("x is equal to neither 1 nor 2");  break;  }  }  } |

Выражение, применяемое оператором **switch** для определения возможности выполнения, должно использовать [Встроенные типы данных](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.ru/dv_csexpresscon/html/387ed62b-5adf-4cbd-8ce1-4f1f6734af36.htm), такие как **int** или **string**; более сложные пользовательские типы использовать нельзя.

В отличие от Visual Basic, условие в C# должно быть постоянным значением. Например, сравнение выражения и диапазона значений не допускается.

## Циклы

Циклом называется один или несколько операторов, повторяющихся заданное число раз или до тех пор, пока не будет выполнено определенное условие. Выбор типа цикла зависит от задачи программирования и личных предпочтений кодирования. Одним из основных отличий C# от других языков, таких как C++, является цикл **foreach**, разработанный для упрощения итерации по массиву или коллекции.

### Циклы foreach

В C# представлен новый способ создания циклов, который может быть неизвестен программистам на C++ и C: цикл foreach. Вместо просто создания структуры данных, такой как коллекция переменной для индексирования массива или другой, цикл **foreach** выполняет более тяжелую работу.

// An array of integers

int[] array1 = {0, 1, 2, 3, 4, 5};

foreach (int n in array1)

{

System.Console.WriteLine(n.ToString());

}

// An array of strings

string[] array2 = {"hello", "world"};

foreach (string s in array2)

{

System.Console.WriteLine(s);

}

### Циклы for

Далее показано создание нескольких циклов с использованием ключевого слова for.

// An array of integers

int[] array1 = {0, 1, 2, 3, 4, 5};

for (int i=0; i<6; i++)

{

System.Console.WriteLine(array1[i].ToString());

}

// An array of strings

string[] array2 = {"hello", "world"};

for (int i=0; i<2; i++)

{

System.Console.WriteLine(array2[i]);

}

### Циклы while

В следующих примерах показаны варианты цикла while.

// An array of integers

int[] array1 = {0, 1, 2, 3, 4, 5};

int x = 0;

while (x < 6)

{

System.Console.WriteLine(array1[x].ToString());

x++;

}

// An array of strings

string[] array2 = {"hello", "world"};

int y = 0;

while (y < 2)

{

System.Console.WriteLine(array2[y]);

y++;

}

### Циклы do-while

В следующих примерах показаны варианты цикла **do-while**.

// An array of integers

int[] array1 = {0, 1, 2, 3, 4, 5};

int x = 0;

do

{

System.Console.WriteLine(array1[x].ToString());

x++;

} while(x < 6);

// An array of strings

string[] array2 = {"hello", "world"};

int y = 0;

do

{

System.Console.WriteLine(array2[y]);

y++;

} while(y < 2);

## Массивы и коллекции

Хранение группы связанных элементов данных является основным требованием большинства программных приложений; для этого существуют два основных способа: массивы и коллекции.

## Массивы

Массивы являются коллекциями объектов одного типа. Поскольку длина массивов практически не ограничена, они могут использоваться для хранения тысяч или даже миллионов объектов, но размер массива должен быть указан при его создании. Каждый элемент массива доступен по числовому индексу, указывающему позицию или ячейку, в которой объект хранится в массиве. Массивы могут хранить как ссылочные типы, так и типы значений.

### Одномерные массивы

Массив является индексированной коллекцией объектов. Одномерный массив объектов объявляется следующим образом.

type[] arrayName;

Часто элементы в массиве инициализируются в это же время, как показано ниже.

int[] array = new int[5];

Значение по умолчанию числовых элементов массива задано равным нулю, а элементы ссылок имеют значение **null**, но значения можно инициализировать при создании массива следующим образом.

int[] array1 = new int[] { 1, 3, 5, 7, 9 };

Или так.

int[] array2 = {1, 3, 5, 7, 9};

Индексация массивов начинается с нуля, поэтому номер первого элемента массива равен 0.

string[] days = {"Sun", "Mon", "Tue", "Wed", "Thr", "Fri", "Sat"};

System.Console.WriteLine(days[0]); // Outputs "Sun"

### Многомерные массивы

Концептуально, многомерный массив с двумя измерениями напоминает сетку. Многомерный массив с тремя измерениями напоминает куб.

// declare multidimension array (two dimensions)

int[,] array2D = new int[2,3];

// declare and initialize multidimension array

int[,] array2D2 = { {1, 2, 3}, {4, 5, 6} };

// write elements in a multidimensional array

for (int i=0; i<2; i++)

{

for (int j=0; j<3; j++)

{

array2D[i,j] = (i + 1) \* (j + 1);

}

}

// read elements in a multidimensional array

for (int i=0; i<2; i++)

{

for (int j=0; j<3; j++)

{

System.Console.Write(array2D[i,j]);

}

System.Console.WriteLine();

}

### Массивы массивов

Одним из вариантов многомерного массива является массив массивов. Массив массивов представляет собой одномерный массив, в котором каждый элемент является массивом. Элементы массива не обязаны иметь одинаковый размер.

Объявить массив массивов можно следующим образом

int[][] jaggedArray = new int[3][];

Создание массива трех массивов. Эти массивы можно инициализировать следующим образом.

jaggedArray[0] = new int[5];

jaggedArray[1] = new int[4];

jaggedArray[2] = new int[2];

### Использование оператора foreach

Оператор foreach часто используется для доступа к каждому элементу, хранимому в массиве.

int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };

foreach (int i in numbers)

{

System.Console.WriteLine(i);

}

**Массивы объектов**

Создание массива объектов в отличие от создания массива простых типов данных, например целочисленных, происходит в два этапа. Сначала необходимо объявить массив, а затем создать объекты для хранения в нем. В этом примере создается класс, определяющий аудио компакт-диск. Затем создается массив для хранения 20 аудио компакт-дисков.

|  |  |
| --- | --- |
| 001:  002:  003:  004:  005:  006:  007:  008:  009:  010:  011:  012:  013:  014:  015:  016:  017:  018:  019:  020:  021:  022:  023:  024:  026:  027:  028:  029:  030:  031:  032:  033:  034:  035:  036:  037:  038:  039:  040:  041:  042:  043:  044:  045:  046:  047: | namespace CDCollection  {  // Define a CD type.  class CD  {  private string album;  private string artist;  private int rating;  public string Album  {  get {return album;}  set {album = value;}  }  public string Artist  {  get {return artist;}  set {artist = value;}  }  public int Rating  {  get {return rating;}  set {rating = value;}  }  }  class Program  {  static void Main(string[] args)  {  // Create the array to store the CDs.  CD[] cdLibrary = new CD[20];  // Populate the CD library with CD objects.  for (int i=0; i<20; i++)  {  cdLibrary[i] = new CD();  }  // Assign details to the first album.  cdLibrary[0].Album = "See";  cdLibrary[0].Artist = "The Sharp Band";  cdLibrary[0].Rating = 10;  }  }  } |

## Коллекции

Массив является одним из многочисленных вариантов хранения набора данных, используемых C#. Вариант выбора зависит от нескольких факторов, например от планируемого способа управления или доступа к элементам. Например, **список** работает, как правило, быстрее массива при добавлении элемента в начало или в середину коллекции. Другие типы классов коллекций, такие как хэш-таблица, связный список и стек, имеют свои преимущества.

В следующем примере показано использование класса List<(Of <(T>)>). Обратите внимание, что в отличие от класса Array, элементы могут вставляться в середину списка. В этом примере показано ограничение списка, который может содержать только текстовые элементы.

|  |  |
| --- | --- |
| 001:  002:  003:  004:  005:  006:  007:  008:  009:  010:  011:  012:  013:  014:  015:  016:  017:  018: | public class TestCollections  {  public static void TestList()  {  System.Collections.Generic.List<string> sandwich = new System.Collections.Generic.List<string>();  sandwich.Add("bacon");  sandwich.Add("tomato");  sandwich.Insert(1, "lettuce");  foreach (string ingredient in sandwich)  {  System.Console.WriteLine(ingredient);  }  }  } |

В этом примере показано три различных способа объявления нескольких видов массивов: одномерного, многомерного и массива массивов.

Пример

// Single-dimensional arrays.

int[] myArray1 = new int [5];

string[] myArray2 = new string[6];

// Multidimensional arrays.

int[,] myArray3 = new int[4,2];

int[,,] myArray4 = new int [4,2,3];

// Jagged array (array of arrays)

int[][] myArray5 = new int[3][];

### Надежное программирование

Убедитесь, что перед использованием массива массивов выполнена инициализация его элементов, как показано далее.

myArray5[0] = new int[7];

myArray5[1] = new int[5];

myArray5[2] = new int[2];

В этом примере показано три различных способа инициализации нескольких видов массивов: одномерного, многомерного и массива массивов.

Пример

// Single-dimensional array (numbers).

int[] n1 = new int[4] {2, 4, 6, 8};

int[] n2 = new int[] {2, 4, 6, 8};

int[] n3 = {2, 4, 6, 8};

// Single-dimensional array (strings).

string[] s1 = new string[3] {"John", "Paul", "Mary"};

string[] s2 = new string[] {"John", "Paul", "Mary"};

string[] s3 = {"John", "Paul", "Mary"};

// Multidimensional array.

int[,] n4 = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };

int[,] n5 = new int[,] { {1, 2}, {3, 4}, {5, 6} };

int[,] n6 = { {1, 2}, {3, 4}, {5, 6} };

// Jagged array.

int[][] n7 = new int[2][] { new int[] {2,4,6}, new int[] {1,3,5,7,9} };

int[][] n8 = new int[][] { new int[] {2,4,6}, new int[] {1,3,5,7,9} };

int[][] n9 = { new int[] {2,4,6}, new int[] {1,3,5,7,9} };

В следующем примере показана передача массива объектов в метод DisplayMyCollection, использующий ключевое слово params для принятия любого числа аргументов.

Пример

class MyBoxingClass

{

public static void DisplayMyCollection(params object[] anArray)

{

foreach (object obj in anArray)

{

System.Console.Write(obj + "\t");

}

// Suspend the screen.

System.Console.ReadLine();

}

static void Main()

{

DisplayMyCollection(101, "Visual C# Basics", 2002);

}

}

В этом примере используется оператор **foreach** для доступа и отображения элементов массива.

Пример

int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};

foreach (int element in numbers)

{

System.Console.WriteLine(element);

}

В этом примере используется оператор **foreach** для доступа и отображения элементов массива.

Пример

int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};

foreach (int element in numbers)

{

System.Console.WriteLine(element);

}

## Перечисления

C# позволяет создать собственный набор именованных констант с помощью ключевого слова **enum**. Эти типы данных позволяют объявить набор имен или других значений литералов, определяющих все возможные значения, которые могут быть назначены переменной.

Например, если в программе ведется работа с днями недели, может потребоваться создать новый тип с именем **DayOfWeek**. Затем можно объявить новую переменную типа **DayOfWeek**, а затем присвоить ей значение. Использование этого типа данных способствует повышению удобочитаемости кода, кроме того, снижается вероятность назначения переменной недопустимого или неожиданного значения.

|  |  |
| --- | --- |
| 001:  002:  003:  004:  005:  006:  007:  008:  009:  010:  011:  012:  013:  014:  015:  016:  017:  018:  019:  020:  021:  022: | public enum DayOfWeek  {  Sunday = 0,  Monday = 1,  Tuesday = 2,  Wednesday = 3,  Thursday = 4,  Friday = 5,  Saturday = 6  }  class Program  {  static void Main()  {  DayOfWeek day = DayOfWeek.Monday;  int i = (int) DayOfWeek.Monday;  System.Console.WriteLine(day); // displays Monday  System.Console.WriteLine(i); // displays 1  }  } |

### Дополнительные способы перечисления

Далее представлено несколько дополнительных функций типов данных **enum**, которые могут быть полезны.

### Отображение значений литералов перечисления

Для в доступа к имени или словам, используемым типе данных enum можно применить метод **ToString(),** как показано далее.

DayOfWeek day = DayOfWeek.Wednesday;

System.Console.WriteLine(day.ToString()); // displays Wednesday

### Установка значений по умолчанию

По умолчанию первым значением в перечисляемом типе является ноль. Можно указать другое начальное значение, как показано далее.

enum Color { Red = 1, Yellow = 2, Blue = 3 };

Фактически, можно определить уникальные целочисленные значения для всех значений.

enum Medal { Gold = 30, Silver = 20, Bronze = 10 };

## Манипулирование строками

Манипулирование строками является одной из наиболее часто выполняемых операций.

Переменная типа string может восприниматься как доступный только для чтения массив переменных char. Это означает, что доступ к отдельным символам можно получать с использованием такого синтаксиса:

string myString = "A string";

char myChar = myString [1];

Однако присваивать отдельные символы подобным образом нельзя. Для получения массива char, пригодного для выполнения записи, можно применять показанный ниже код. В этом коде задействована такая команда переменной массива, как ToCharArray():

string myString = "A string";

char[] myChars = myString.ToCharArray() ;

После этого массивом char можно начинать манипулировать обычным образом. Также еще можно использовать строки в циклах foreach, как показано ниже:

foreach (char character in myString)

{

Console.WriteLine ("{0} ", character);

}

Как и в случае массивов, получать информацию о количестве элементов в строке также можно с помощью myString.Length:

string myString = Console.ReadLine();

Console.WriteLine ("You typed {0} characters.", myString.Length);

Другие базовые приемы манипулирования строками подразумевают использование команд в формате, подобном <строка>.ToCharArray (). К числу наиболее простых, но полезных из них относятся команды <строка>. ToLower() и <строка>. ToUpper(). Эти две команды позволяют преобразовывать строки, соответственно, в нижний и в верхний регистр. Допустим требуется обеспечить выполнение проверки на предмет наличия конкретного ответа от пользователя, например, строки yes. Преобразование вводимой пользователем строки в нижний регистр позволит также выполнять проверку на предмет наличия и таких строк, как YES, Yes, yeS и т.д.

string userResponse = Console.ReadLine();

if (userResponse.ToLower () == "yes")

{

// Выполнение действия в ответ.

}

Эта команда, подобно прочим командам в настоящем разделе, на самом деле не изменяет ту строку, к которой применяется. Вместо этого комбинирование этой команды со строкой приводит к созданию новой строки, которая может далее либо сравниваться с еще какой-нибудь строкой (как это делается здесь), либо присваиваться другой переменной. В роли этой другой переменной может выступать та же самая переменная, над которой выполняются операции:

userResponse = userResponse.ToLower();

Очень важно запомнить этот момент, поскольку использование просто такой строки кода:

userResponse.ToLower() ;

на самом деле не позволит получить никаких особых результатов. Существуют и другие вещи, которые можно делать для упрощения интерпретации вводимых пользователем данных. Что если пользователь случайно введет в начале или в конце входных данных лишний пробел? В таком случае приведенный код работать не будет. Потребуется усечь введенную строку, что можно сделать с помощью команды <строка>.Trim():

string userResponse = Console.ReadLine();

userResponse = userResponse.Trim() ;

if (userResponse.ToLower() == "yes")

{

// Выполнение действия в ответ.

}

Применение такого подхода позволит также распознавать и такие строки, как:

*" YES"*

*"Yes "*

Эти команды можно применять и для удаления любых других символов, указывая их в массиве char, например:

|  |  |
| --- | --- |
| 001:  002:  003:  004:  005:  006:  007:  008: | char[] trimChars = {' ', 'е\ ' s'};  string userResponse = Console.ReadLine();  userResponse = userResponse.ToLower ();  userResponse = userResponse.Trim(trimChars);  if (userResponse == "y")  {  // Выполнение действия в ответ.  } |

Этот код предусматривает устранение любых вхождений символов пробела, буквы е и буквы s в начале или конце строки. В случае отсутствия любых других символов в строке это будет приводить к распознаванию таких строк. Еще можно применять команды <строка>.TrimStart () и <строка>. TrimEnd (), которые будут отсекать символы пробела, соответственно, в начале и конце строки. С ними тоже можно задавать массивы char. Существуют еще две других строковых команды, которые можно использовать для манипулирования пробелами в строках: <строка>. PadLeft () и <строка>. PadRight (). Эти команды позволяют добавлять пробелы слева и справа от строки для принудительного приведения ее к желаемой длине. Применяются они следующим образом:

<строка>. PadX (<желаемая\_длина>) ;

Ниже приведен пример:

myString = "Aligned";

myString = myString.PadLeft(10, ‘-‘);

Выполнение этого кода приведет к добавлению слева от слова Aligned в myString трех символов. Данные команды могут быть очень полезными при выравнивании строк в столбцах, что является особенно удобным для позиционирования строк, содержащих числа.

Как и команды усечения, эти команды тоже можно использовать вторым способом, предоставляя в них символ, который должен применяться для дополнения строки. Указывать разрешено только один символ char, а не массив символов char, как в случае усечения:

myString = "Aligned";

myString = myString.PadLeft(10, '-');

Выполнение этого кода приведет к добавлению в начале myString трех символов -. Существует еще очень много подобных команд для манипулирования строками, но большинство из них полезно только в специфических ситуациях.