|  |
| --- |
| , RD Dep. |
| Конспект и раздаточный материал |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial version | Анжелика Кравчук |  |  |  |
| 1.1 | Review and corrections. | Владимир Тихон |  |  |  |

Contents

[1. Урок 1: Объявление переменных и присваивание значений 3](#_Toc301268626)

[1.1. Что такое переменные? 3](#_Toc301268627)

[1.2. Типы данных 4](#_Toc301268628)

[1.3. Объявление и присваивание переменных 5](#_Toc301268629)

[1.4. Область видимости переменной 8](#_Toc301268630)

[1.5. Преобразование типов данных 9](#_Toc301268631)

[1.6. Константы и переменные только для чтения 11](#_Toc301268632)

[2. Урок 2: Использование выражений и операций в С# 12](#_Toc301268633)

[2.1. Выражения 12](#_Toc301268634)

[2.2. Операции 14](#_Toc301268635)

[2.3. Приоритет операций 15](#_Toc301268636)

[2.4. Рекомендации по выполнению строковой конкатенации 17](#_Toc301268637)

[3. Урок 3: Использование операторов выбора 18](#_Toc301268638)

[3.1. Использование сокращенной формы оператора if (one-way if) 18](#_Toc301268639)

[3.2. Использование полной формы оператора if (either-or if) 20](#_Toc301268640)

[3.3. Использование лесенки if else if…(multiple-outcome if) 21](#_Toc301268641)

[3.4. Использование оператора Switch 22](#_Toc301268642)

[3.5. Рекомендации по использованию операторов выбора 23](#_Toc301268643)

[4. Урок 4: Использование операторов цикла 24](#_Toc301268644)

[4.1. Типы операторов цикла 25](#_Toc301268645)

[4.2. Использование оператора while 26](#_Toc301268646)

[4.3. Использование оператора do 27](#_Toc301268647)

[4.4. Использование оператоа for 28](#_Toc301268648)

[4.5. Операторы break и continue 29](#_Toc301268649)

[5. Урок 5: Создание и использование массивов 30](#_Toc301268650)

[5.1. Что такое массив? 31](#_Toc301268651)

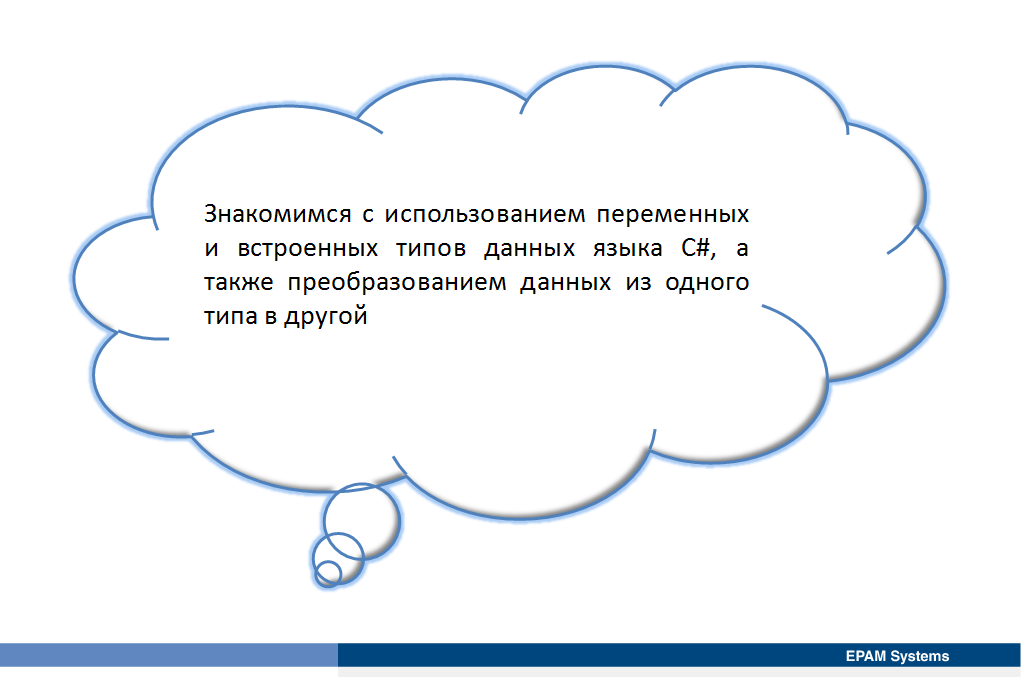
[5.2. Создание и инициализация массивов 32](#_Toc301268652)

[5.3. Общие свойства и методы, предоставляемые массивами 34](#_Toc301268653)

[5.4. Доступ к данным в массиве 35](#_Toc301268654)

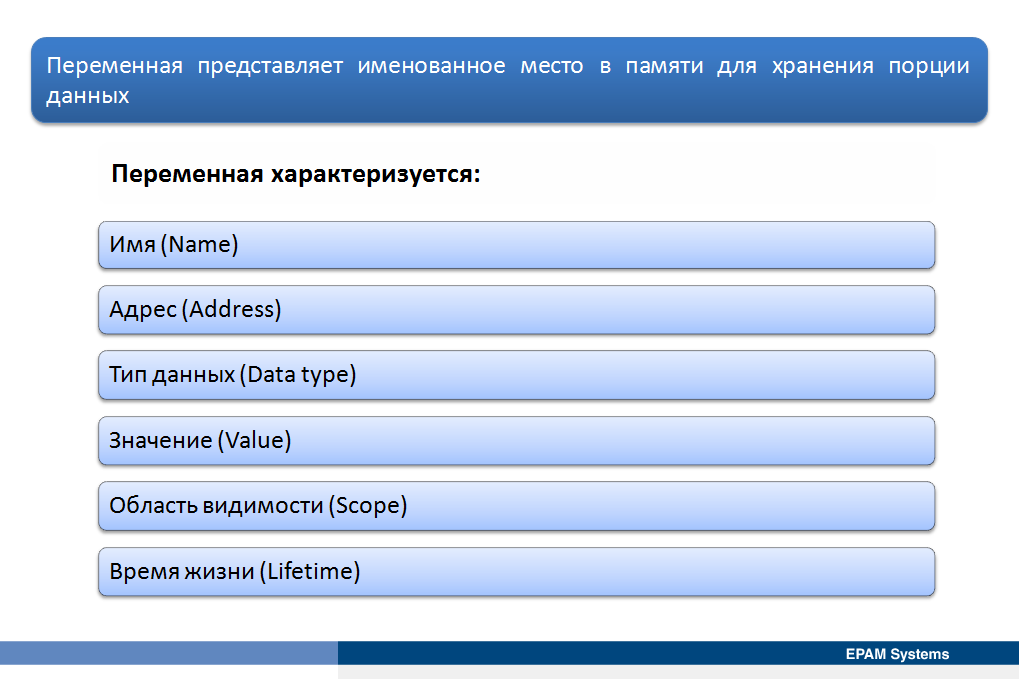
[5.5. Демонстрация. Создание и использование массивов 36](#_Toc301268655)

# Урок 1: Объявление переменных и присваивание значений



Все приложения используют данные, которые могут быть предоставлены с помощью пользовательского интерфейса, из базы данных, услуг сети или других источников. Для хранения и использования данных в приложениях следует ознакомиться с тем, как определять и использовать переменные и типы данных в C#. В уроке описано, как в языке C# используются переменные и встроенные типы данных, а также объясняется, как преобразовать данные, которые хранятся в переменной одного типа данных, в другой.

## Что такое переменные?



Переменная представляет именованное место в памяти для хранения порции данных. Приложение может получить доступ к данным с помощью переменной, с которой они связаны. Часто при выполнении расчетов или передаче данных между пользователем, приложением и базой данных необходимо временно хранить значения. Например, можно получить несколько значений из базы данных, сравнить их, и выполнить различные операции над ними в зависимости от результатов сравнения. Переменные хранят значения, которые приложение может изменить во время его работы.

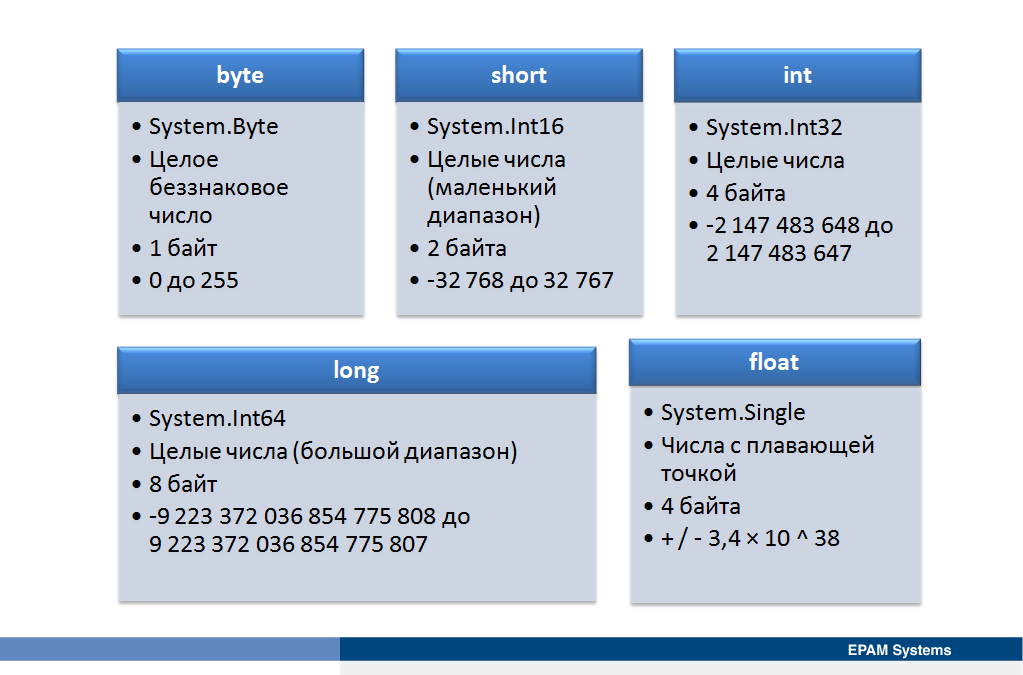
На переменную можно взглянуть с разных сторон. Переменная характеризуется:

* *Имя (Name).* Уникальный идентификатор, который ссылается на переменную в коде.
* *Адрес (Address).* Ячейка памяти переменной.
* *Тип данных (Data type).* Тип данных, которые может хранить переменная (определяет размер и представление данных в памяти).
* *Значение (Value).* Значение по адресу переменной.
* *Область видимости (Scope).* Определенные участки кода, которые могут получать доступ и использовать переменную.
* *Время жизни (Lifetime).* Период времени, который переменная является действительной и доступной для использования.

Переменные можно использовать во многих случаях, в том числе:

* Как счетчик для циклов.
* В качестве временного хранилища для значения свойства.
* В качестве контейнера для хранения значения, возвращаемого из функции.

## Типы данных

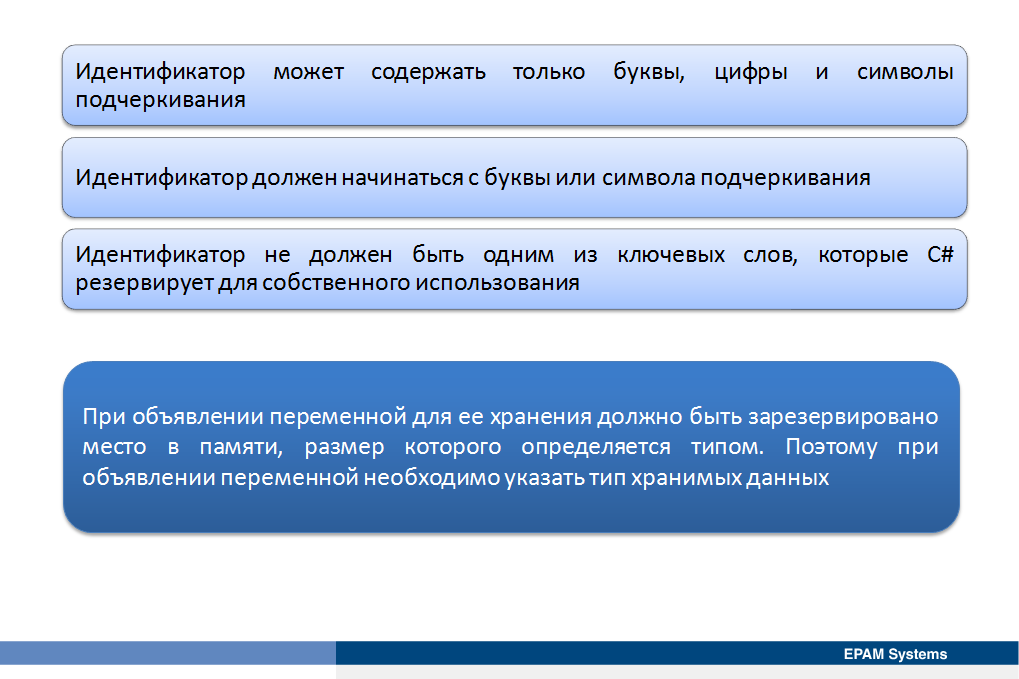


Как и в любом языке программирования, в С# поставляется собственный набор основных типов данных, которые должны применяться для представления локальных переменных, переменных экземпляра, возвращаемых значений и входных параметров. Однако в отличие от других языков программирования, в С# эти ключевые слова представляют собой нечто большее, чем просто распознаваемые компилятором лексемы. Они, по сути, представляют собой сокращенные варианты обозначения полноценных типов из пространства имен System. В следующей таблице перечислены эти системные типы данных, отвечающие требованиям общеязыковой спецификации CLS (Common Language Specification), вместе с охватываемыми ими диапазонами значений, соответствующими ключевыми словами на С#.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Тип** | **FCL-типы** | **Описание** | **Размер**  **(в байтах)** | **Диапазон** |
| **byte** | **System.Byte** | Целое беззнаковое число | 1 | 0 до 255 |
| **short** | **System.Int16** | Целые числа  (маленький диапазон) | 2 | -32 768 до 32 767 |
| **int** | **System.Int32** | Целые числа | 4 | -2 147 483 648 до 2 147 483 647 |
| **long** | **System.Int64** | Целые числа  (больший диапазон) | 8 | -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 |
| **float** | **System.Single** | Числа с плавающей точкой | 4 | + / - 3,4 × 10 ^ 38 |
| **double** | **System.Double** | Числа двойной точности  (более точные)  с плавающей точкой | 8 | + / -1,7 × 10 ^ 308 |
| **decimal** | **System.Desimal** | Денежный значения | 16 | 28 значащих цифр |
| **char** | **System.Char** | Один символ | 2 | N/A |
| **bool** | **System.Bool** | Логический | 1 | true или false |
| **string** | **System.String** | Последовательность символов | 2 на  симол | N/A |
| **object** | **System.Object** | Служит базовым классом для всех типов в мире .NET | Позволяет сохранять любой тип в объектной переменной | |

Каждый из числовых типов, такой как short или int отображается на соответствующую структуру в пространстве имен System.

## Объявление и присваивание переменных



Прежде чем использовать переменную, ее нужно объявить, указав ее имя и характеристики. Имя переменной называется идентификатором. Существуют правила, касающиеся идентификаторов, которые можно использовать:

* Идентификатор может содержать только буквы, цифры и символы подчеркивания.
* Идентификатор должен начинаться с буквы или символа подчеркивания.
* Идентификатор не должен быть одним из ключевых слов, которые C# резервирует для собственного использования.

C# при именовании учитывает регистр, а, следовательно, идентификаторы MyData и MYDATA, определяют различные переменные, хотя это и не является хорошим стилем программироваия.

Для именования переменных следует использовать осмысленные имена, поскольку это делает код более простым для понимания. При этом необходимо придерживаться соглашения об именовании.

При объявлении переменной для ее хранения должно быть зарезервировано место в памяти, размер которого определяется типом. Поэтому при объявлении переменной необходимо указать тип хранимых данных. В одном объявлении можно с помощью запятой указать несколько переменных, при этом все объявленные переменные будут иметь один и тот же тип. Синтаксис объявления переменных показан ниже.

DataType variableName;

// or

DataType variableName1, variableName2;

int variableName;

После объявления переменной можно присвоить значение для его дальнейшего использования в приложении с помощью оператора присваивания. Значение переменной можно изменить столько раз, сколько это необходмо. Операция присваивания «=» присваивает значение переменной.

variableName = value;

Значение правой части выражения присваивается переменной в левой части выражения. В следующем примере объявлена целочисленная переменная **price, которой п**рисваивается значение **10**.

int price = 10;

Далее существующей целочисленной переменной **price** присваивается число 20.

price = 20;

Присвоить значение переменным можно при их объявлении (инициализация). В следующем примере кода показан синтаксис объявления переменной с одновременным присваиванием.

DataType variableName = value;

Тип выражения при присваивании должен соответствовать типу переменной, иначе программа не будет компилироваться. Например, код в следующем примере не будет компилироваться, потому что нельзя присвоить строковое значение целочисленной переменной.

int numberOfEmployees;

numberOfEmployees = "Hello";

При объявлении переменной, пока ей не присвоено значение, она содержит случайное значение. Такое поведение являлось источником ошибок в программах C и C++, которые создавали переменную и могли случайно использовать ее в качестве источника информации, прежде чем присвоить ей значение. C# не позволяет использовать неинициализированную переменную. Необходимо присвоить значение переменной прежде, чем использовать его, в противном случае, программа не будет компилироваться.

При объявлении переменных, можно использовать ключевое слово var вместо указания явного типа данных, таких как int или string. Когда компилятор встречает ключевое слово var, он автоматически выводит лежащий в основе тип данных на основе первоначального значения, используемого для инициализации переменной. Поэтому, при объявлении необходимо инициализировать переменную, определив ее следующим образом

var price = 20;

В примере переменная price неявно типизированная переменная. Однако, ключевое слово var не означает, что переменной price можно позже присвоить значение другого типа. Тип переменной price фиксирован, так же, как если бы явно было объявлено, что это целочисленная переменная.

Неявную типизацию можно использовать для любых типов, включая массивы, обобщенные типы и пользовательские специальные типы.

С использованием ключевого слова var связаны различные ограничения. Самое первое и важное из них состоит в том, что неявная типизация применима только для локальных переменных в контексте какого-то метода или свойства. Применять ключевое слово var для определения возвращаемых значений, параметров или данных полей специального типа не допускается.

class ThisWillNeverCompile

{

// var не может применяться для определения полей!

private var mylnt = 10;

// var не может применяться для определения

// возвращаемого значения или типа параметра!

public var MyMethod(var x, var y) { }

}

Однако, неявно типизированную локальную переменную можно возвращать вызывающему методу, при условии, что возвращаемый тип этого метода совпадает с типом, лежащим в основе определенных с помощью var данных.

static int GetAnIntValue()

{

var retVal = 9;

return retVal;

}

Локальным переменным, объявленным с помощью ключевого слова var, не допускается присваивать в качестве начального значения null. Однако присваивание значения null локальной переменной с выведенным после начального присваивания типом вполне допустимо (если переменная относится к ссылочному типу).

var myObj = null;

// Ошибка! Присваивание null в качестве начального

// значения не допускается!

var myObj = (int?)null;

myObj = 78;

var myCar = new Car();

// Все в порядке, поскольку Car

// является переменной ссылочного типа!

myCar = null;

Значение неявно типизированной локальной переменной может быть присвоено другим переменным, причем как неявно, так и явно типизированным.

var myInt2 = 0;

var anotherInt = myInt;

string myString2 = "Wake up!";

var myData = myString;

И, наконец, определять неявно типизированную локальную переменную как допускающую значение null с использованием лексемы «?» в С# нельзя.

Следует отметить, что неявная типизация локальных переменных приводит к получению строго типизированных данных. Таким образом, применение ключевого слова var в С# не приводит к созданию переменнах, которые на протяжении своего существования в программе могут хранить значения разных типов (динамическая типизация[[1]](#footnote-1)). Выведение типа позволяет языку С# оставаться строго типизированным и оказывает влияние только на объявление переменных во время компиляции. После этого данные трактуются как объявленные с выведенным типом; присваивание такой переменной значения другого типа будет приводить к возникновению ошибок на этапе компиляции.

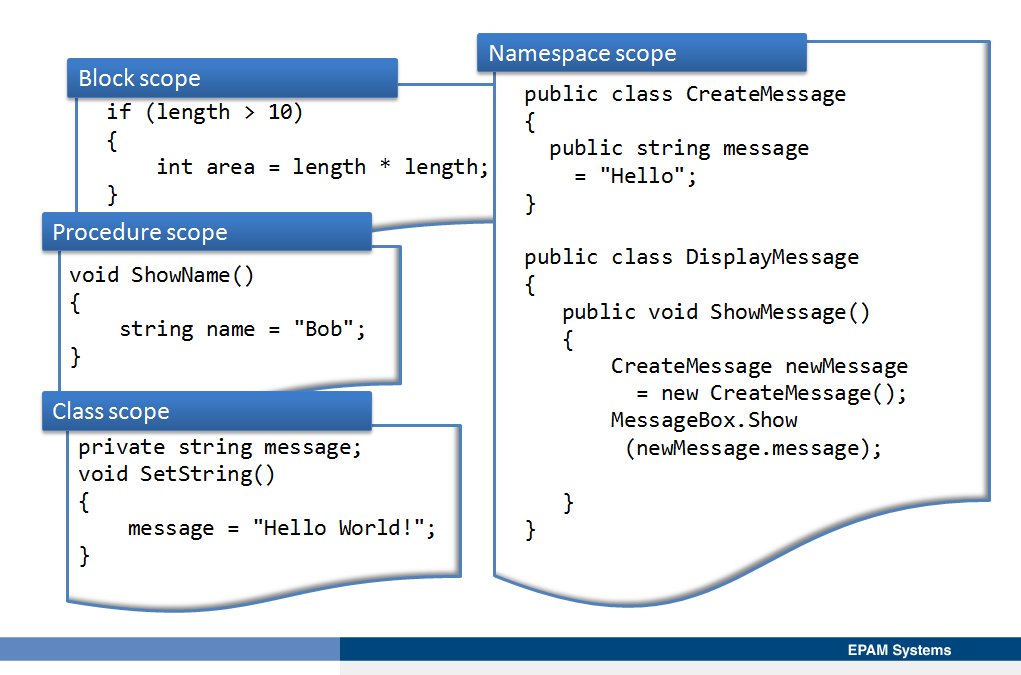
Использование ключевого слова var для объявления локальных переменных просто так особой пользы не приносит, а в действительности может даже вызвать путаницу у тех, кто будет изучать данный код, поскольку лишает возможности быстро определить тип данных и, следовательно, понять, для чего предназначена переменная. Поэтому если точно известно, что переменная должна относиться к типу int, лучше сразу объявить ее с указанием этого типа. Неявно типизированные переменные полезны, когда не известно или трудно явно установить тип выражения, которое будет присвоено переменной. Например, в технологии LINQ применяются так называемые выражения запросов (query expression), которые позволяют получать динамически создаваемые результирующие наборы на основе формата запроса. В таких выражениях неявная типизация чрезвычайно полезна, так как в некоторых случаях явное указание типа просто не возможно.

<http://go.microsoft.com/fwlink/?LinkId=192890>

<http://go.microsoft.com/fwlink/?LinkId=192891>

<http://go.microsoft.com/fwlink/?LinkId=192892>

## Область видимости переменной



Область видимости переменной определяется частью программы, которая может получить доступ к этой переменной. При попытке сослаться на переменную вне ее области видимости, компилятор будет генерировать ошибку.

Переменные могут иметь один из следующих уровней области видимости:

* Блок (Block).
* Процедура (Procedure).
* Класс (Class).
* Пространство имен (Namespace).

Эти уровни охватывают диапазон от узкой (Block) до широкой (Namespace) областей видимости.

**Область видимости Block**. Блоком является набор операторов, заключенных в пределах инициирования и определения операторов объявления. Объявленная в блоке переменная, может быть использована только внутри этого блока. В следующем примере кода показано, как объявить локальную переменную area с областью видимости уровня блока.

if (length > 10)

{

int area = length \* length;

}

**Область видимости Procedure.** Переменные, которые объявлены в рамках процедуры не доступны вне пределов этой процедуры. Только процедура, которая содержит объявление переменной может ее использовать. При объявлении переменных в блоке или процедуре переменные известны как локальные. В следующем примере показано объявление локальной переменной с именем name области видимости уровня процедуры.

void ShowName()

{

string name = "Bob";

MessageBox.Show("Hello " + name);

}

**Область видимости Class.** Если необходимо, чтобы время жизни локальной переменной выходило за рамки жизни процедуры, следует объявить переменную на уровне класса. При объявлении переменных в классе или структуре, но вне процедуры, переменные известны как переменные класса. Определить область видимости для переменных класса можно используя модификаторы доступа. В следующем примере показано объявление private переменную message с областью видимости уровня класса.

...

private string message;

void SetString()

{

message = "Hello World!";

}

void ShowString()

{

MessageBox.Show(message);

}

...

**Область видимости Namespace**. При объявлении переменных на уровне класса с помощью ключевого слова **public** переменные доступны для всех процедур в пределах всего пространства имен. В следующем примере показано, как объявить **public** переменную message в одном классе, но к которой можно получить доступ в другом классе.

public class CreateMessage

{

public string message = "Hello";

}

public class DisplayMessage

{

public void ShowMessage()

{

CreateMessage newMessage = new CreateMessage();

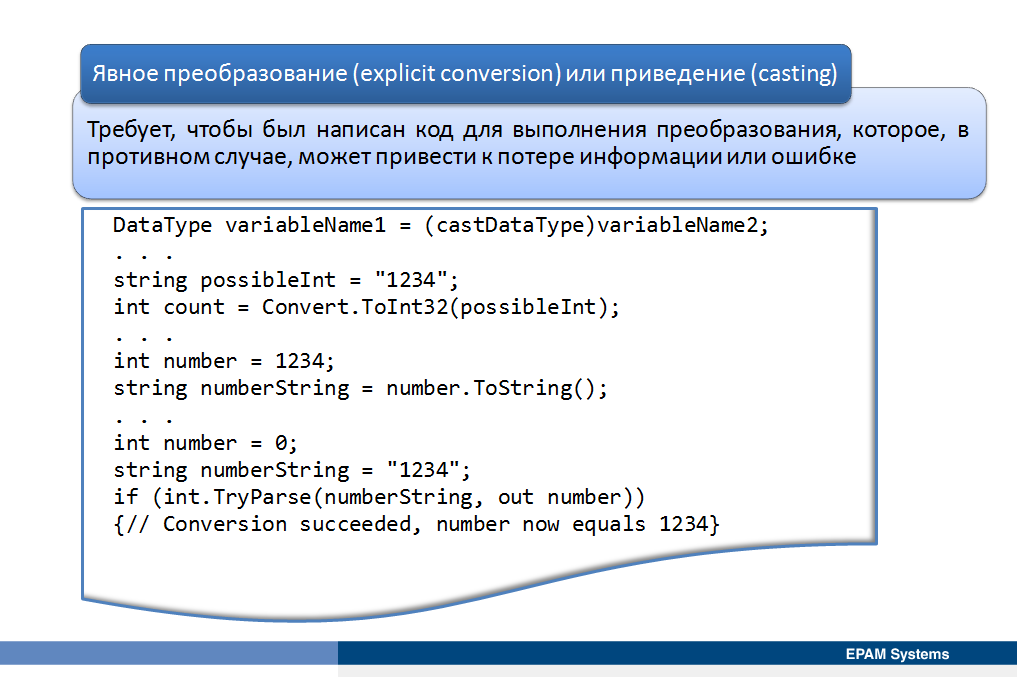
MessageBox.Show(newMessage.message);

}

}

<http://go.microsoft.com/fwlink/?LinkId=192893>.

## Преобразование типов данных



При разработке приложения, может понадобиться преобразовать данные из одного типа в другой. Преобразования необходимы, когда значение одного типа должно быть присоено переменной другого типа. Например, может понадобиться преобразовать строковое значение «99», прочитанное из текстового файла в целое значение 99 для его хранения в целочисленной переменной. Процесс преобразования значение одного типа данных в другой называется преобразование (conversion) или приведение (casting).

В .NET Framework существует два типа преобразования:

* *Неявное преобразование* (implicit conversion). Неявное преобразование выполняется автоматически общеязыковой средой выполнения (CLR) согласно операциям, которые гарантированно завершатся успешно без потери информации.
* *Явное преобразование* (explicit conversion) *или приведение* (casting). Приведение требует, чтобы был написан код для выполнения преобразования, которое, в противном случае, может привести к потере информации или ошибке.

Явное преобразование уменьшает возможность ошибки в коде и делает его более эффективным. В C# запрещены неявные преобразования, которые приводят к потере точности. Однако следует помнить, что и некоторые явные преобразования могут привести к непредсказуемым результатам.

**Неявное преобразование.** Неявное преобразование происходит при автоматическом преобразовании значения из одного типа данных в другой. Преобразование не требует никакого специального синтаксиса в исходном коде. C# позволяет только безопасные неявные преобразования, такие, как расширение числа. В следующем примере показано неявное преобразование из целочисленной переменной в тип long.

int a = 4;

long b;

b = a; // Implicit conversion of int to long

Такое преобразование всегда успешно и никогда не приведет к потере информации. Однако, обратное преобразование не верно, нельзя неявно преобразовать тип long в тип int, потому что это преобразование рискует потерей информации (значение типа long может выходить за пределы диапазона, который поддерживает тип int). В следующей таблице приведены неявные преобразования типов, которые поддерживаются в C#.

|  |  |
| --- | --- |
| **Из** | **В** |
| **sbyte** | **short, int, long, float, double, decimal;** |
| **byte** | **short, ushort, int, uint, long, ulong, float, double, decimal** |
| **short** | **int, long, float, double, decimal** |
| **ushort** | **int, uint, long, ulong, float, double, decimal** |
| **int** | **long, float, double, decimal** |
| **uint** | **long, ulong, float, double, decimal** |
| **long, ulong** | **float, double, decimal** |
| **float** | **double** |
| **char** | **ushort, int, uint, long, ulong, float, double, decimal** |

**Явные преобразования.** В C# можно использовать операцию приведения для выполнения явного преобразования. Приведение типов указывает на тип преобразования в круглых скобках.

DataType variableName1 = (castDataType)variableName2;

Таким способом можно выполнять только значимые преобразования, такие как преобразование long к int. Нельзя использовать приведение, если формат данных физически измененяется, например, если нужно конвертировать строку в целое число. Для выполнения таких преобразований, используются методы класса System.Convert. Одно из преимуществ подхода с применением класса System.Convert связано с тем, что он позволяет выполнять преобразования между типами данных нейтральным к языку образом, следовательно, все языки, ориентированные на CLR могут использовать этот класс. Этот класс можно использовать для облегчения преобразований, поскольку Microsoft IntelliSense помогает найти метод преобразования, который нужен. Класс System.Convert предоставляет методы, которые могут преобразовать данные базового типа в другой базовый тип данных. Эти методы имеют имена, такие как ToDouble, ToInt32, ToString и так далее. В следующем примере тип string преобразуется к типу int.

string possibleInt = "1234";

int count = Convert.ToInt32(possibleInt);

В дополнение к методу Convert.ToString, типы реализуют собственный метод ToString.

int number = 1234;

string numberString = number.ToString();

Некоторые из встроенных типов данных в C# обеспечивают метод TryParse, который позволяет определить, является ли преобразование удачным, перед его выполнением.

int number = 0;

string numberString = "1234";

if (int.TryParse(numberString, out number))

{

// Conversion succeeded, number now equals 1234

}

else

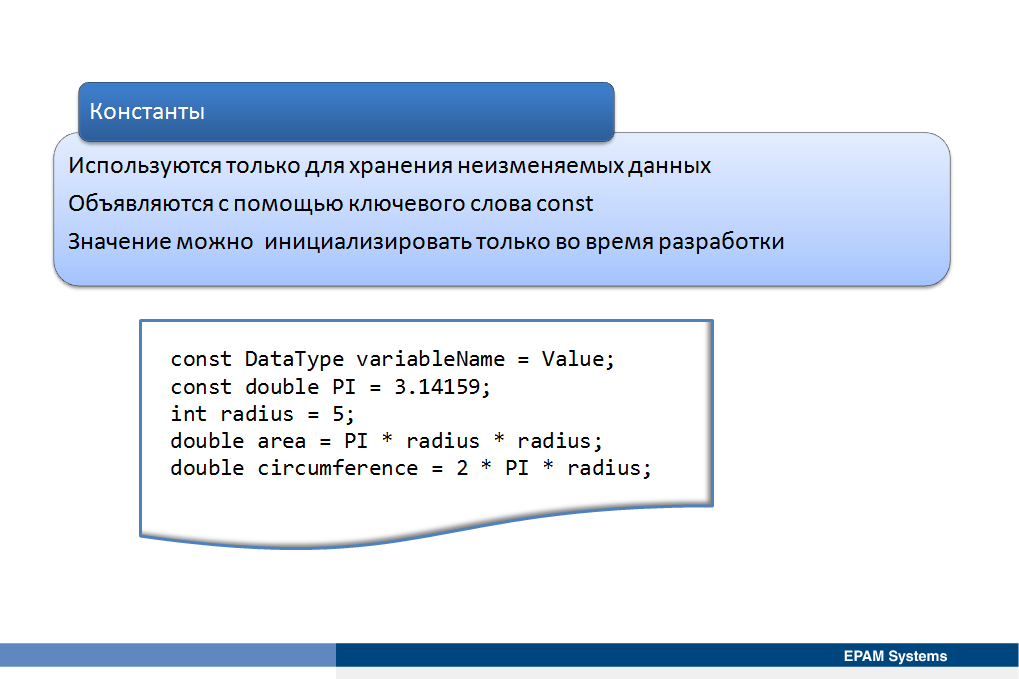
{

// Conversion failed, number now equals 0

}

<http://go.microsoft.com/fwlink/?LinkId=192894>

## Константы и переменные только для чтения



Константы и переменные только для чтения (read-only) позволяют хранить данные, которые могут хранить любые переменные в C#. Однако, при этом они могут использоваться только для хранения данных, которые не изменяются. Переменные только для чтения или константы можно использовать, например, для таких значений, как:

* Количество часов в дне.
* Скорость света.
* Число градусов в окружности.

Объявляется переменная только для чтения с помощью ключевого слова readonly:

readonly DataType variableName = Value;

Константы объявляются с помощью ключевого слова const:

const DataType variableName = Value;

Между использованием констант и переменных только для чтения существует различие. При использовании в приложении константа инициализируется только при объвлении. Таким образом определить и инициализировать значение константы можно только во время разработки и нельзя изменить ее значение во время работы приложения. Переменную только для чтения можно инициализировать при объявлении или в конструкторе класса, содержащем переменную только для чтения. Следовательно, поля с модификатором readonly могут иметь различные значения в зависимости от использованного конструктора.

В следующем примере объявляется переменная только для чтения currentDateTime для хранения текущих даты и времени. В этом примере используется структура DateTime и свойство Now, которое позволяет вычислить текущую дату и время во время выполнения. Попытка использовать этот подход с константой приведет к ошибке времени компиляции.

readonly string currentDateTime = DateTime.Now.ToString();

В следующем примере объявлена константа PI для расчета площади и окружности с радиусом 5.

const double PI = 3.14159;

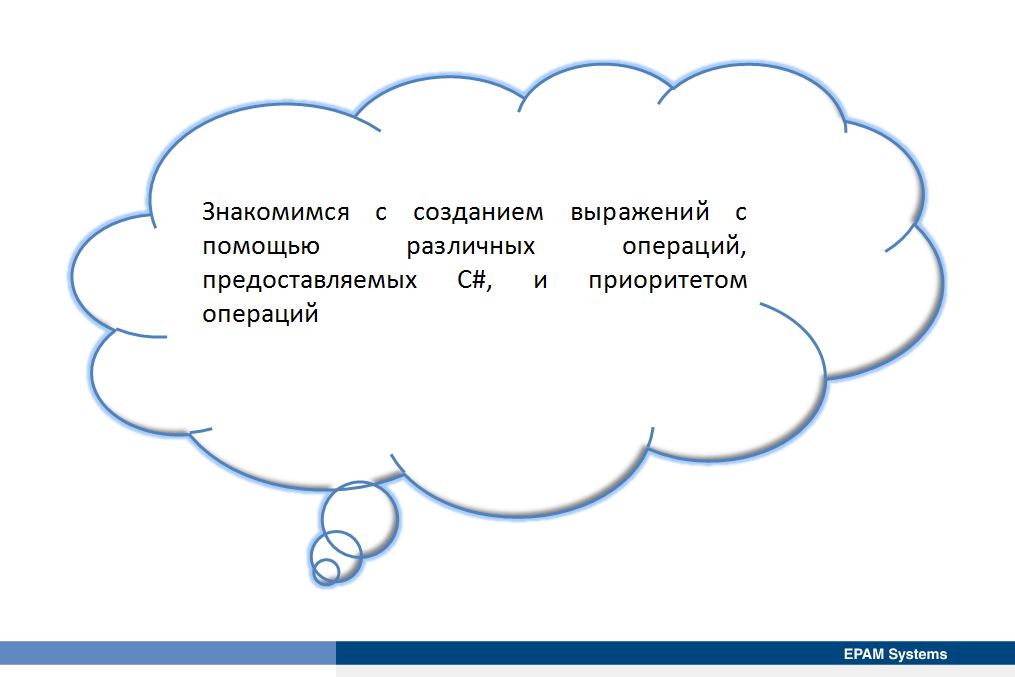
int radius = 5;

double area = PI \* radius \* radius;

double circumference = 2 \* PI \* radius;

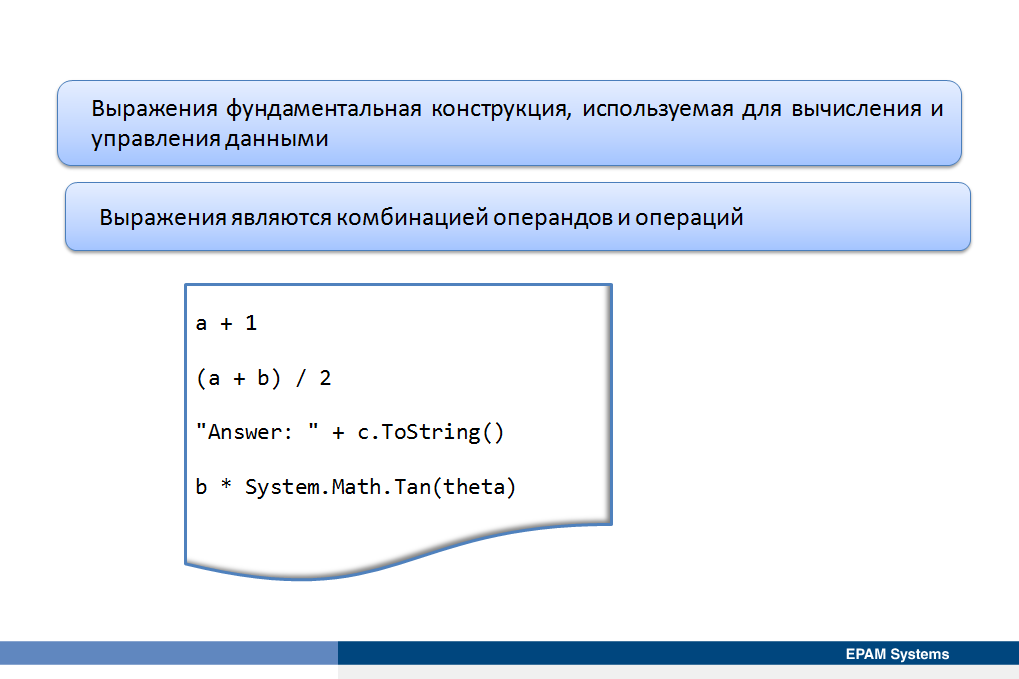
<http://go.microsoft.com/fwlink/?LinkId=192895>

# Урок 2: Использование выражений и операций в С#



Значения, присваиваемые переменным, могут быть постоянными, но чаще, это значения, которые являются результатом выражений, вычисляемых во время выполнения. В уроке описано создание выражений с помощью различных операций, предоставляемых C#, приоритет операций и возможность его изменения в выражениях с помощью скобок, даны рекомендации при работе со строками.

## Выражения



Выражения являются центральными компонентами практически каждого приложения C#, поскольку они являются фундаментальной конструкцией, используемой для вычисления и управления данными. Выражения являются комбинацией операндов и операций. Указанные термины можно определить следующим образом:

* *Операнды* (Operands). Операнды это значения, например, числа и строки, над которыми выполняются операции. Операндами могут быть постоянные значения (литералы), переменные, свойства или результаты вызова метода.
* *Операции* (Operators)*[[2]](#footnote-2)*. Операции определяют действия, выполняемые над операндами. Операции существуют для всех основных математических операций в дополнение к более сложным операциям, таким как логическое сравнение или манипуляции битами данных.

Во время работы приложения все выражения вычисляются в одно значение. Тип значения, которое дает выражение, зависит от типов операндов и операций, которые при этом используются. В приложениях C# не существует никаких ограничений на длину выражения, хотя на практике, она ограничивается памятью компьютера и терпением при вводе. Как правило, рекомендуется использовать достаточно короткие выражения и собирать результаты обработки выражения по частям. Это облегчает понимание того, что делает код, кроме того, делает его проще для отладки в том случае, когда приложение работает нет так, как хотелось бы.

Для того, чтобы строить выражения, как простые, так и сложные, можно комбинировать основные строительные блоки операндов и операций. В самом простейшем случае, можно использовать один операнд для выражения, как показано ниже.

a

Такой операнд может показаться не очень полезным, но, по сути, он необходим. Например, если нужно присвоить значение переменной, потребуется выражение такого типа, на основании которого можно создавать более сложные выражения с помощью операций, как показано в следующем примере.

a + 1

Операция сложения «+» может работать с различными типами данных, и результат выражения будет зависеть от типов данных операндов. Например, если операндом a является целое число, результатом выражения также будет целое число со значением на 1 больше a. Если a это переменная типа double, результатом будет double со значением на 1 больше a. Однако, во втором случае (double) компилятор C# должен будет генерировать код для преобразования постоянного целого значения 1 в постоянное double значение 1, после чего будет вычислено выражение. Правилом является то, что тип выражения совпадает с типом операндов, хотя при этом один или несколько операндов, возможно, необходимо преобразовать, чтобы обеспечить их совместимость. Например, выражение в следующем примере кода содержит два целочисленных операнда, поэтому результатом будет целое число 2 (не 2,5).

5 / 2

При преобразовании одного из операндов в тип double, C# компилятор преобразует другой операнд в double, и результатом будет значение типа double. Следовательно, выражение в следующем примере кода дает double значение 2,5.

5.0 / 2

Продолжить создание выражений с дополнительными значениями и операциями можно как показано в следующем примере.

a + b - 2

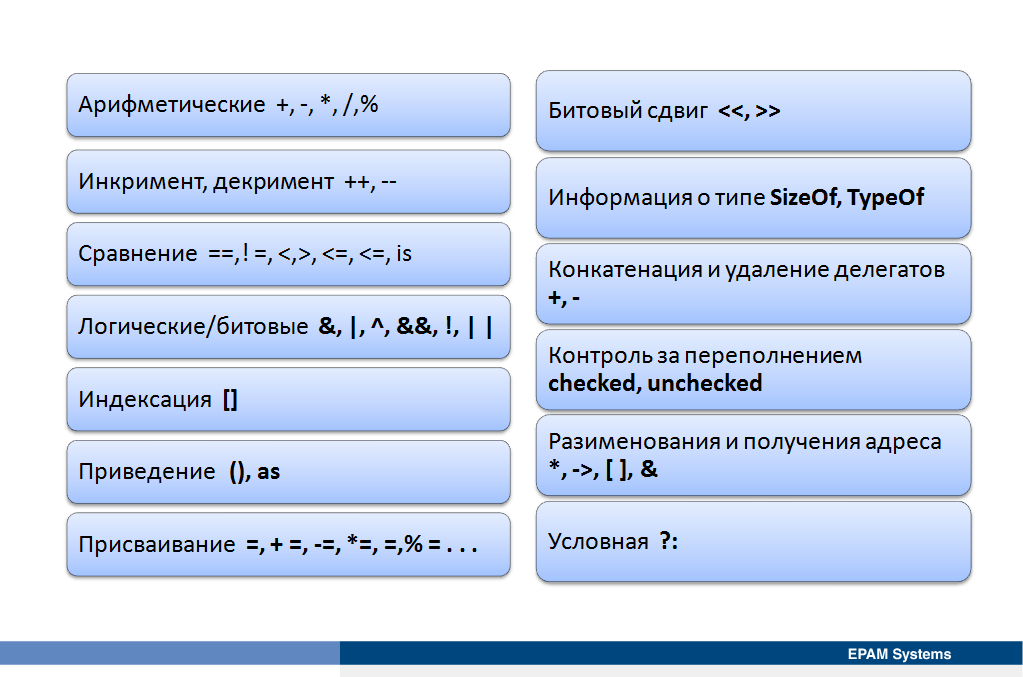
Это выражение для суммы переменных a и b, из результата которого вычтено 2. Некоторые операции, такие как сложение «+», могут быть использованы для вычисления выражений, которые имеют диапазон типов. Например, выражение в следующем примере кода использует операцию сложения для объединения двух строк.

"Answer: " + c.ToString();

Здесь операция сложения использует операнд, который является результатом вызова метода ToString. Этот метод преобразует значение переменной в строку независимо от ее типа.

Библиотека классов .NET Framework содержит много дополнительных методов, используемых для выполнения математических и строковых операций над данными.

## Операции



Операции и операнды объединяются вместе в выражениях. C# предоставляет широкий спектр операций, которые можно использовать для выполнения большинства фундаментальных математических и логических операций.

Операции можно разделить на следующие три категории:

* *Унарные*. Операции с одним операндом. Например, можно использовать операцию вычитания «-» как унарную. Для этого нужно поместите знак «-» непосредственно перед операндом, и он преобразует его значение так, как будто его текущее значение умножается на -1.
* *Бинарные*. Операции с двуми операндами. Это наиболее распространенный тип операций, к которму относится, например, операция умножения «\*».
* *Тернарные*. Существует только одна тернарная операция в C#. Это операция «? :», которая используется в условных выражениях.

В следующей таблице приведены сгруппированные по типу операции, которые можно использовать в C#.

|  |  |
| --- | --- |
| **Operator type** | **Operators** |
| Арифметические | **+, -, \*, /,%** |
| Инкримент, декримент | **++, --** |
| Сравнение | **==,! =, <,>, <=, <=, is** |
| Объединение (конкатенация) строк | **+** |
| Логические/битовые | **&, |, ^, &&, !, | |** |
| Индексация (отсчет начинается с элемента 0) | **[]** |
| Приведение (сasting) | **(), as** |
| Присваивание | **=, + =, -=, \*=, =,% =, & =, | =, =, <=,>> =,?** |
| Битовый сдвиг | **<<, >>** |
| Информация о типе | **SizeOf, TypeOf** |
| Конкатенация и удаление делегатов | **+, -** |
| Контроль за переполнением | **checked, unchecked** |
| Разименования и получения адреса | **\*, ->, [ ], &** |
| Условная | **?:** |

**Инкримент и декримент переменных.** Если к переменной нужно добавить 1, можно использовать операцию сложения «+», как показано в следующем примере.

count = count + 1;

Однако, добавление 1 к переменной настолько распространенная операция, что C# предоставляет для этой цели собственную операцию – операцию инкремента «++». Чтобы увеличить значение переменной на 1, можно написать оператор следующего вида.

count++;

Кроме того, С# предлагает операцию декремента «--», которую можно использовать для вычитания 1 из переменной, как показано ниже.

count--;

Операции инкремента и декремента являются унарными.

**Использование составных операций присваивания.** Если нужно добавить 42 к значению переменной, можно объединить операции присваивания и сложения. Например, в следующем примере к переменной answer 42 добавляется. После выполнения этой операции значение ответа станет на 42 больше.

answer = answer + 42;

Однако, добавление значения к переменной настолько распространено, что C# позволяет выполнить эту задачу сокращенным образом с помощью составной операции составного присваивания «+=». Чтобы добавить 42 к переменной answer, можно написать

answer += 42;

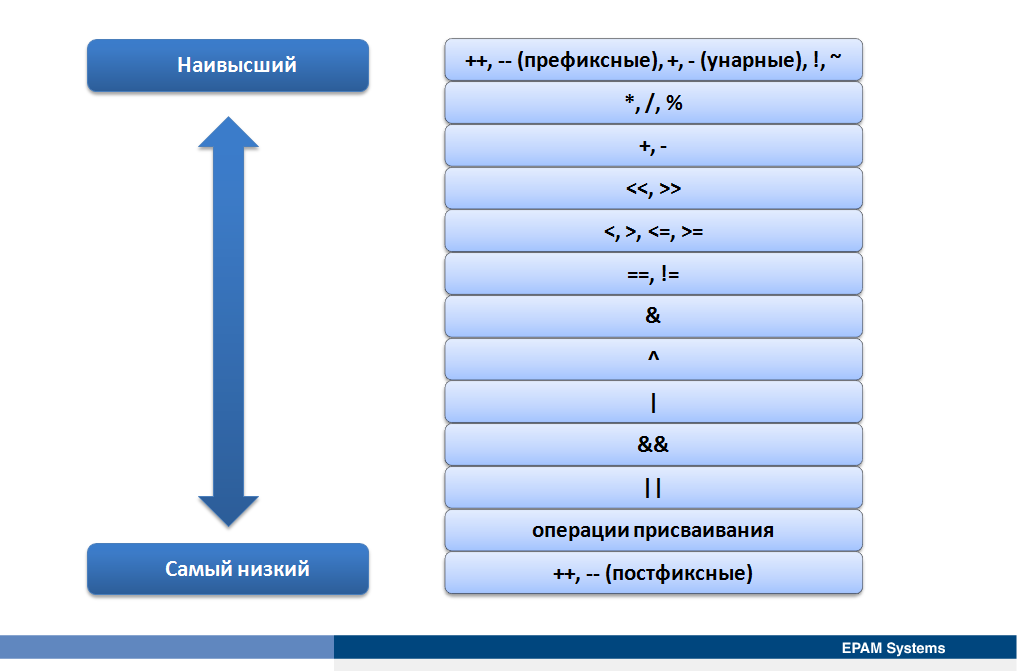
Такое сокращение можно использовать, чтобы объединить все арифметические операции с операцией присваивания, как показано в следующей таблице. Все вместе эти операции известны как составные операции присваивания.

|  |  |
| --- | --- |
| **Заменить** | **на** |
| variable = variable \* number; | variable \*= number; |
| variable = variable / number; | variable /= number; |
| variable = variable % number; | variable %= number; |
| variable = variable + number; | variable += number; |
| variable = variable - number; | variable -= number; |

<http://go.microsoft.com/fwlink/?LinkId=192896>

http://go.microsoft.com/fwlink/?LinkId=192897

## Приоритет операций



Выражение может содержать сложный ряд операций и операндов. Порядок, в котором операции обрабатываются и вычисляются зависит от самих операций. Не всегда существует простой лево-направленный поток выражений.

Операции, которые используются для создания выражения имеют приоритет, определяющий порядок, в котором они вычисляются. Некоторые операции имеют более высокий приоритет, чем остальные, а значит, выполнятся в первую очередь. Например, в следующем примере, операция деления выполняется до операции сложения.

a = b + 1 / 2;

В таблице приведен приоритет операций от высокого уровня до самого низкого.

|  |  |
| --- | --- |
| **Приоритет** | **Операция** |
| Наивысший  Самый низкий | ++, -- (префиксные), +, - (унарные), !, ~ |
| \*, /, % |
| +, - |
| <<, >> |
| <, >, <=, >= |
| ==, != |
| & |
| ^ |
| | |
| && |
| || |
| операции присваивания |
| ++, -- (постфиксные) |

Кроме того, операции имеют конкретную ассоциативность, определяющую порядок, в котором они выполняются по отношению к другим операциям, имеющим такой же приоритет. При использовании операций с одинаковым приоритетом ассоциативность операций используется для определения порядка выполнения. Операции либо право-ассоциативны, либо лево-ассоциативны. Лево-ассоциативные операции выполняются слева направо, например, операция деления «/»:

a / 5 / b

Здесь, **a** делится на 5, а затем результат делится делится на b. Все бинарные операции лево-ассоциативны, кроме операции присваивания, которая право-ассоциативна:

a = b = c

Здесь значение **c** присваивается b, а затем значение b присваивается a (множественное присваивание). Кроме того, стоит отметить, что для многих операторов, ассоциативность не всегда важна.

a + 5 + b

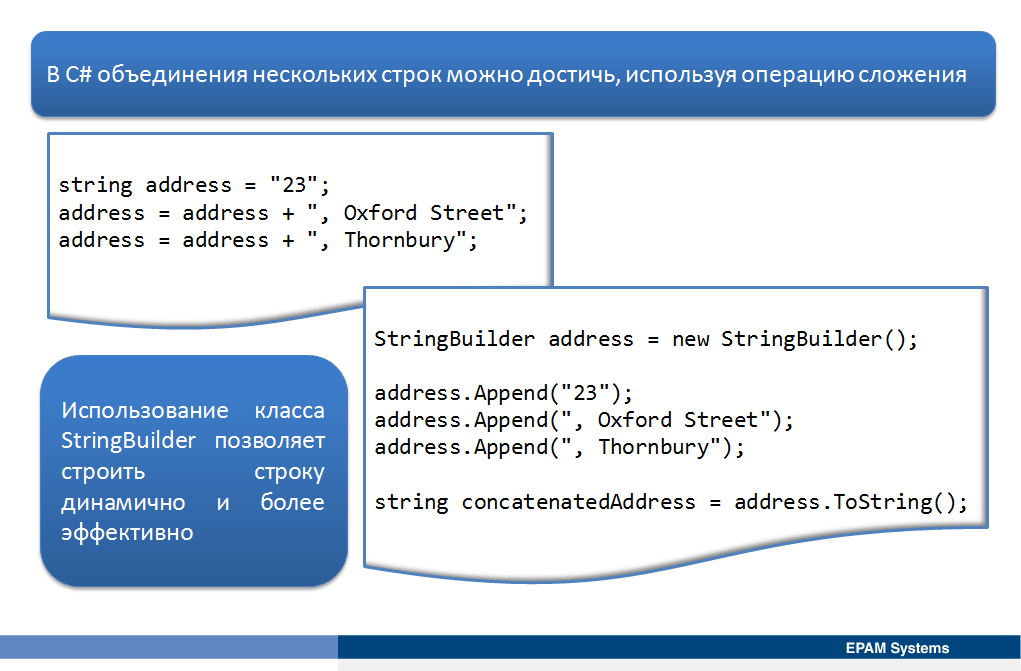
В этом примере не существует никакой разницы для результата при обработке выражения слева направо или справа налево. Однако, поскольку операция сложения определяется как лево-ассоциативный, это может иметь значение в более сложных случаях, например, при перегрузке операций.

Для управления порядком выполнения и изменения приоритета в выражениях можно использовать круглые скобки. Любая часть выражения, которая заключена в скобки обрабатывается до частей выражения, которые находятся вне скобок:

a = (b + 1) / 2;

Здесь (b + 1) – часть выражения, которая обрабатываются в первую очередь, и результат этой операции делится на 2, чтобы определить значение, присвоенное **a**.

## Рекомендации по выполнению строковой конкатенации



В C# объединения нескольких строк можно достичь просто используя операцию сложения. Однако, это считается плохим стилем программирования, поскольку строки являются неизменяемым типом (immutable types). Это означает, что каждый раз при объединении строки, в памяти создается новая строка, а старая строка удаляется. Например, в следующем пример кода в процессе работы создается пять строковых значений.

string address = "23";

address = address + ", Oxford Street";

address = address + ", Thornbury";

Альтернативный подход заключается в использовании класса StringBuilder, который позволяет строить строку динамично и более эффективно. В следующем примере кода показано, как можно использовать класс StringBuilder.

StringBuilder address = new StringBuilder();

address.Append("23");

address.Append(", Oxford Street");

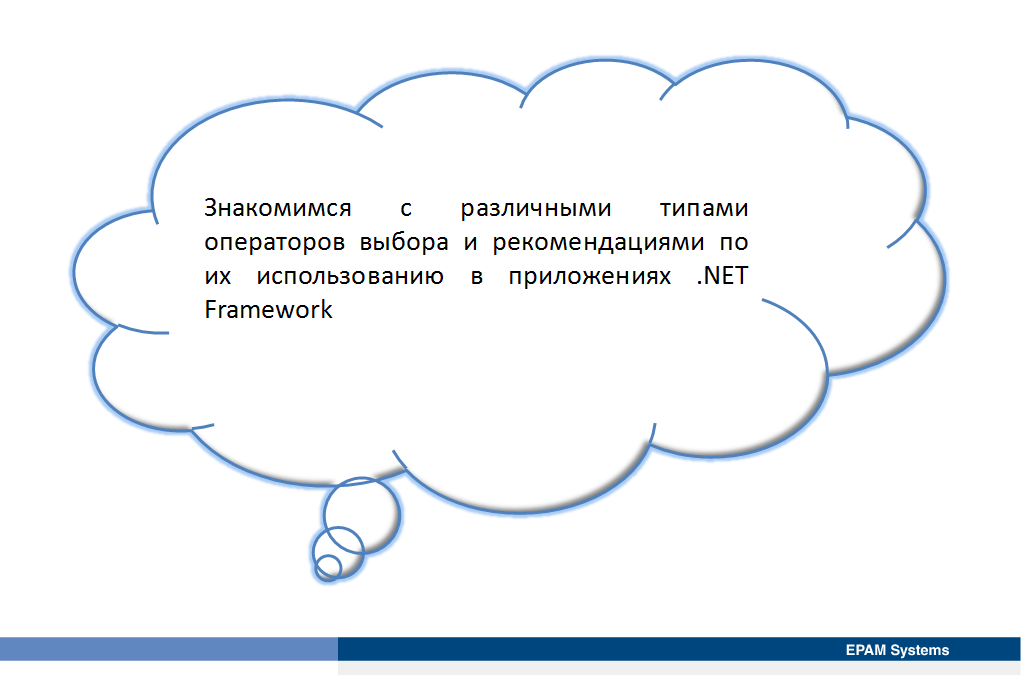
address.Append(", Thornbury");

string concatenatedAddress = address.ToString();

Класс StringBuilder находится в пространстве имен System.Text.

<http://go.microsoft.com/fwlink/?LinkId=192898>

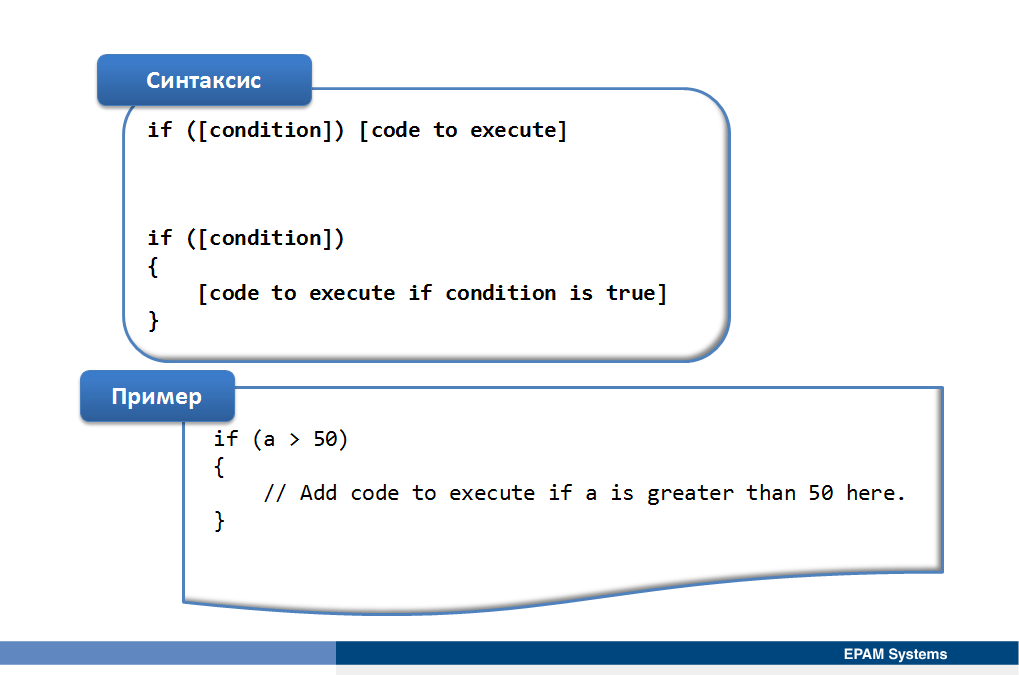
# Урок 3: Использование операторов выбора



Операторы (statements) – это процедурные кирпичики, из которых построены все программы С#. Операторы могут объявлять локальную переменную или константу, содержать вызов метода, создание объекта, присваивание значения переменной, свойствам или полям. Управляющие операторы, например, оператор for, могут создавать циклы, или принимать решения о выполнении и ответвлении блока кода, например, операторы if или switch. Операторы, как правило, заканчиваются точкой с запятой.

По умолчанию, C# выполняет операторы в программе в последовательном порядке. Однако при написании программ часто нужно управлять выполнением кода в зависимости от значения выражения или логического условия. Для достижения этой цели C# предоставляет операторы выбора. Этот урок знакомит с различными типами операторов выбора и дает рекомендации по их использованию в приложениях .NET Framework.

## Использование сокращенной формы оператора if (one-way if)



Сокращенная форма оператора if являются очень полезной, когда нужно выполнить код, основанный на условии. Cинтаксис сокращенной формы оператора if имеет вид

if ([condition]) [code to execute]

В случае если значение выражения ([condition]) истино, то выполняется код [code to execute]. Следует отметить, что условие [condition] всегда должно быть заключено в круглые скобки. Если код [code to execute] состоит из более чем одного оператора, то он помещается в операторные скобки. Это расширяет синтаксис использования оператора if следующим образом:

if ([condition])

{

[code to execute if condition is true]

}

Такая форма оператора if используется часто, даже если выполняется только одна строка кода, поскольку делает код легче читаемым и расширяемым. Пример выполнения кода, когда переменная имеет значение больше 50, имеет следующий вид:

if (a > 50)

{

// Add code to execute if a is greater than 50 here.

}

C# также предлагает две логические операции: логическая операция **AND**, которая представлена знаком операции «&&» и логическая операция **OR**, которая представлена знаком операции «||». В совокупности они известны как условные логические операции. Назначением этих операций является объединение двух логических выражений или значений в одном логическом результате. Эти бинарные операции похожи на операции равенства и отношения поскольку значение выражения, в котором они появляются либо истина, либо ложь. Отличие их состоит в том, что значения, которыми они оперируют также должны быть либо истина, либо ложь.

Результатом операции «&&» будет значении true, только если оба операнда, которыми она оперирует являются true. В следующем примере значение true переменной validPercentage присваивается только тогда, когда значение percent больше или равно 0, а значение percent меньше или равно 100.

bool validPercentage;

if (percent >= 0) && (percent <= 100)

{

validPercentage = true;

}

Достичь того же результата можно путем присваивания значения логического выражения непосредственно переменной validPercentage:

validPercentage = (percent >= 0) && (percent <= 100);

Результатом операции «||» будет значении true, если одно из логических выражений, с которыми она работает, равно true. Операция «||» используется для определения является ли любой из операндов в сочетании логическим выражением истинным. В следующем примере кода значение true переменной invalidPercentage присваивается, если значение percent меньше 0 или значение percent больше 100.

bool invalidPercentage;

if ((percent < 0) || (percent > 100))

{

invalidPercentage = true;

}

Иногда, при оценке выражения, которое использует операции «&&» и «||», для определения тогового результата не обязательно оценивать оба операнда. В следующем примере, если значение переменной age больше или равно 20, значение всего выражения является ложным, независимо от того, является ли значение переменной height больше 180.

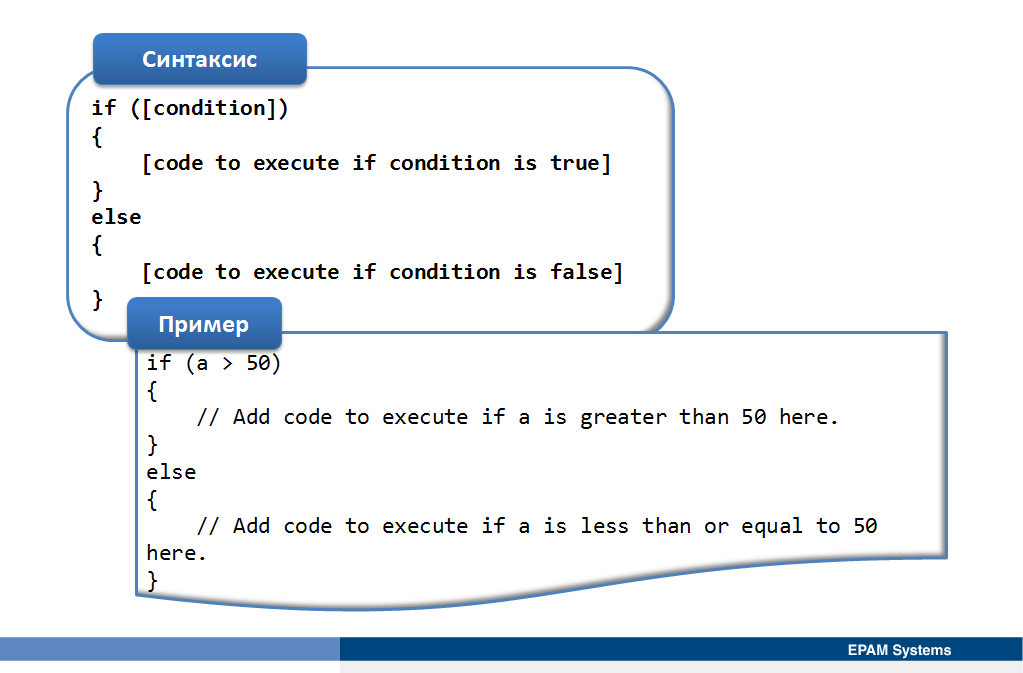
(age < 20) && (height > 180)

Аналогично, если переменная price в следующем примере имеет значение больше или равное 25, значение всего выражения истинно, независимо от того превышает ли значение переменной weight 100.

(price >= 25) || (weight > 100)

Операции «&&» и «||» в C# распознают эти ситуации и в случаях подобным этим, оценка останавливается, как только результат может быть определен. Это означает, что выражение (height > 180) в первом случае и выражения (weight > 100) во втором случае не будут оцениваться. Такое поведение называется замыканием (short-circuiting).

## Использование полной формы оператора if (either-or if)



Для того, чтобы обеспечить выполнения дополнительного блока кода, когда условие [condition] имеет значение false, используетcя ключевое слово **else, а синтаксис оператора if имеет вид**

if ([condition])

{

[code to execute if condition is true]

}

else

{

[code to execute if condition is false]

}

Например,

if (a > 50)

{

// Add code to execute if a is greater than 50 here.

}

else

{

// Add code to execute if a is less than or equal to 50 here.

}

В некоторых простых случаях можно использовать тернарную операцию «?:» как альтернативу использованию полной формы оператора if. Синтаксис использования операции «?:» в этом случае имеет вид

Type result = [condition] ? [true expression] : [false expression]

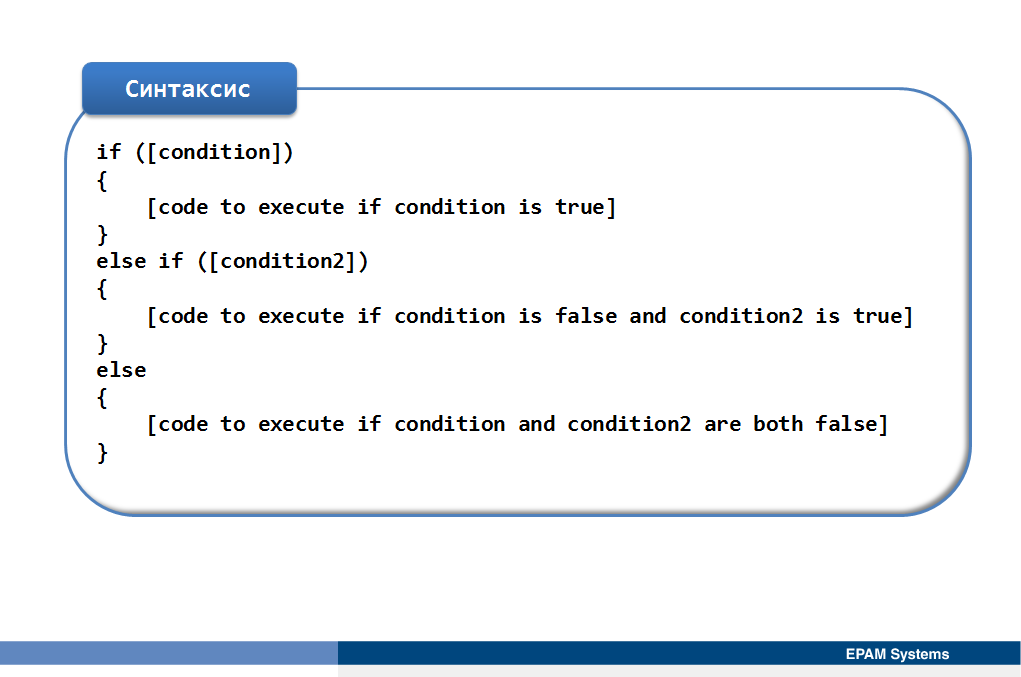
Если выражение [condition] истинно, выполняется [true expression], в противном случае выполняется [false expression]. В следующем примере показан пример использования тернарной операции «?:» для проверки значения строки и возвращения ответа.

string carColor = "green";

string response = (carColor == "red") ? "You have a red car" : "You do not have a red car";

http://go.microsoft.com/fwlink/?LinkId=192904

## Использование лесенки if else if…(multiple-outcome if)



Несколько операторов if можно объединить для создания леснки if else if… (multiple-outcome operator) следующим образом:

if ([condition])

{

[code to execute if condition is true]

}

else if ([condition2])

{

[code to execute if condition is false and condition2 is true]

}

else

{

[code to execute if condition and condition2 are both false]

}

Здесь важно отметить, что если условие [condition] истино, первый блок кода выполняется, независимо от значения условия [condition2]. В этом случае оставшийся код пропускается, и условие [condition2] не вычисляется. Это улучшает производительность, поскольку не требует времени для вычисления каждого условия. Ускорить работу кода можно, гарантируя, что наиболее часто выполняющиеся условия будут проверяться в первую очередь. Следующий код показывает пример оператора if, использующего эту конструкцию.

if (a > 50)

{

// Add code to execute if a is greater than 50 here.

}

else if (a > 10)

{

// Add code to execute if a is greater than 10 and less than or

// equal to 50 here.

}

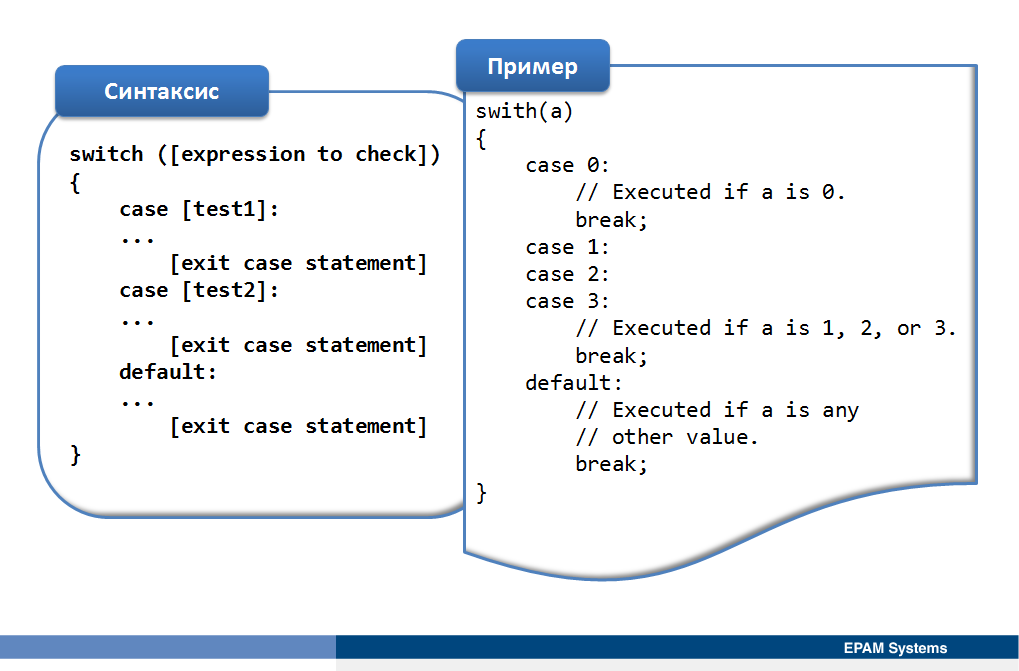
else

{

// Add code to execute if a is less than or equal to 50 here.

}

## Использование оператора Switch



Оператор switch позволяет выполнить один из нескольких блоков кода в зависимости от значения переменной или выражения. Эти блоки кода обеспечивают очень простую легкую для чтения конструкцию и предлагают альтернативный подход к использованию оператора if else if...:

switch ([expression to check])

{

case [test1]:

...

[exit case statement]

case [test2]:

...

[exit case statement]

default:

...

[exit case statement]

}

В операторе switch необходимо в круглых скобках указать выражение для проверки [expression to check] и определить значения [testX] для сравнения с переменной. Сравнения выполняются по очереди, таким образом, если значение [expression to check] совпадет со значением [test1], выполнится первый блок кода, если со значением [test2], выполнится второй блок кода и так далее. Не существует никаких ограничений на количество сравнений, которые можно включить в оператор swith, кроме памяти компьютера. Если совпадений не обнаружено, выполняется блок кода **default**. Блок **default** не является обязательным. Тип значения, который возвращает тестируемое выражение[expression to check] должен быть целым числом (включая char), строкой или логическим значением, а значения, задаваемые операторами case должны соответствовать этому типу. Каждое сравнение ([testX]) является одним значением. Проверить нескольких значений можно с помощью нескольких последовательных операторов case.

swith(a)

{

case 0:

// Executed if a is 0.

break;

case 1:

case 2:

case 3:

// Executed if a is 1, 2, or 3.

break;

default:

// Executed if a is any other value.

break;

}

Каждый блок кода в операторе switch должен заканчиваться оператором, который явно завершает конструкцию ([exit case statement]). Если опустить этот оператор, созникнет ошибка компиляции. В качестве таких оператов можно использовать:

* **break; О**ператор **break** завершает выполнение выбранного оператора.
* **goto case [testX];** Оператор **goto** передает управление на выполнение указанного блока кода в операторе **switch**.
* **return;** Оператор приводит к завершению оператора **switch** и содержащихся в нем методов. С помощью оператора можно передать возвращаемое значение.
* **throw;** Оператор throw генерирует исключительную ситуацию.

Рекомендуется по мере возможности использовать оператор break. Использование **goto case** или **return** может привести к коду, который трудно поддерживать. В следующем примере показано использование оператора **switch** для проверки значения строки.

switch(carColor.ToLower())

{

case "red":

// Red car

break;

case "blue":

// Blue car

break;

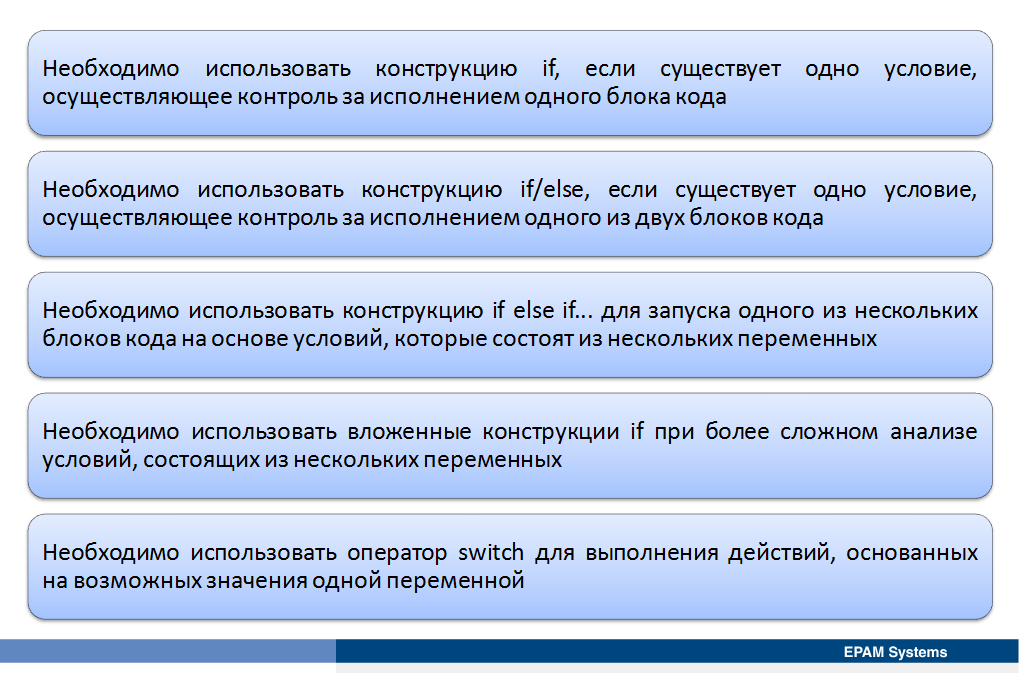
default:

// Unknown car

break;

}

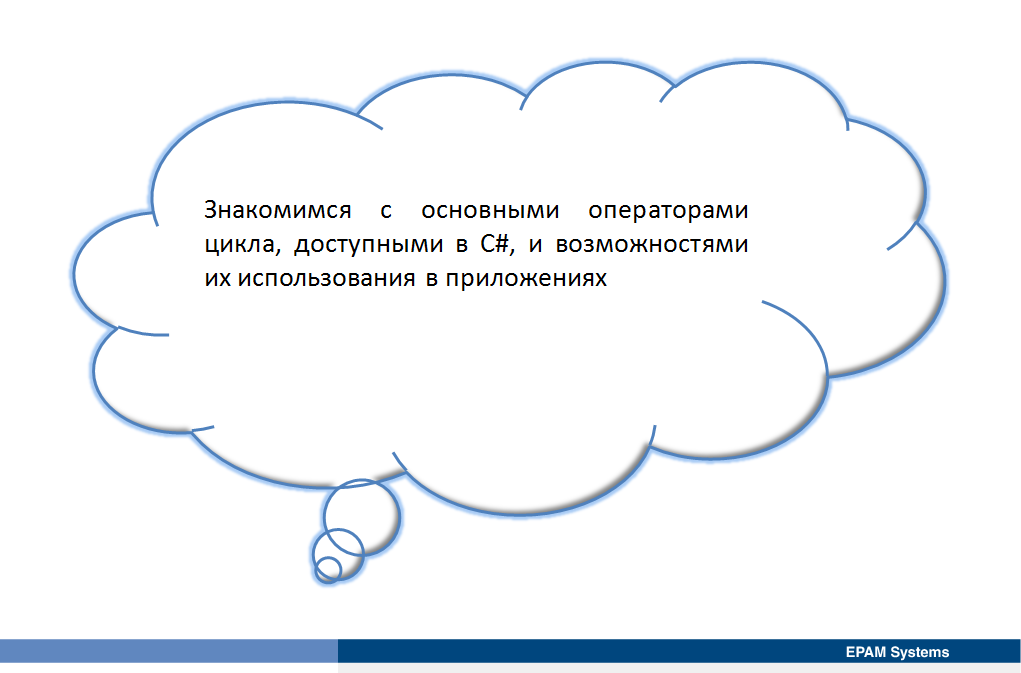
## Рекомендации по использованию операторов выбора



Существует несколько программных конструкций, которые можно использовать для реализации условных операторов. Выбирать программную конструкцию следует основываясь на функцииональности, которую необходимо реализовать. Рекомендации по выбору операторов выбора заключаются в следующем:

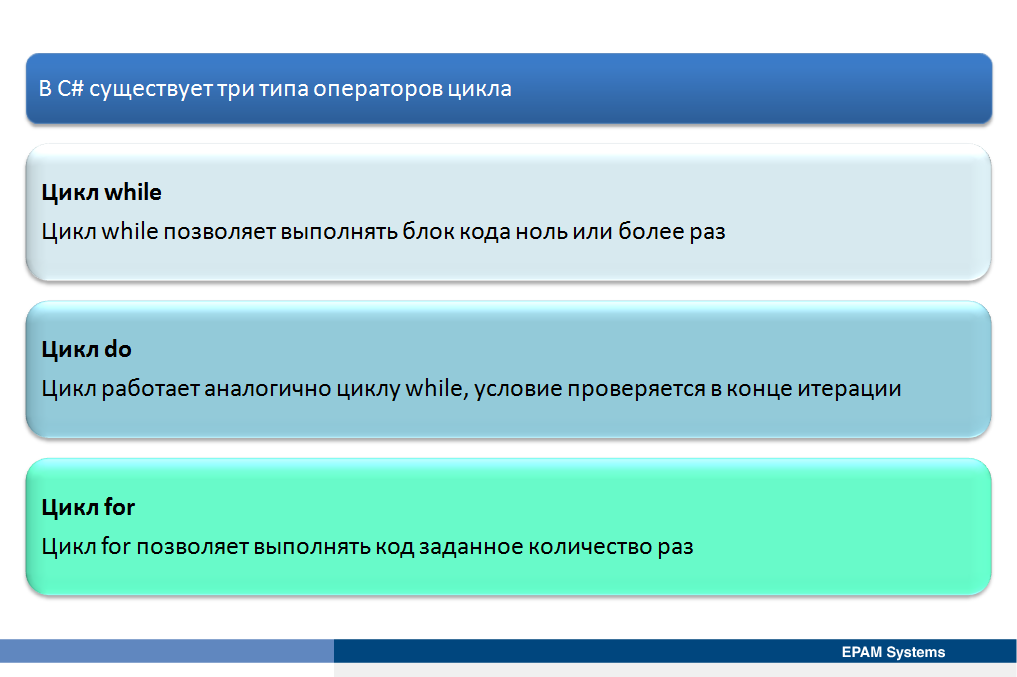
* Необходимо использовать конструкцию **if**, если существует одно условие, осуществляющее контроль за исполнением одного блока кода. Типичным примером является получение реакции пользователя на вопрос «да/нет». Конструкция **if** может использоваться для выполнения блока кода, если пользователь отвечает «да».
* Необходимо использовать конструкцию **if/else**, если существует одно условие, осуществляющее контроль за исполнением одного из двух блоков кода. Эта конструкция подходит для использования, если предлагать пользователю выбор из двух вариантов.
* Необходимо использовать конструкцию **if else if...** для запуска одного из нескольких блоков кода на основе условий, которые состоят из нескольких переменных. Хорошим примером использования этой конструкции является проверка х- и у-координат точек, определенных в прямоугольной области поверхности. Эту конструкцию можно также использовать для проверки одной переменной, которая имеет значение в определенном диапазоне или диапазонах.
* Использование вложенных конструкций **if** выполняет более сложный анализ условий, которые состоят из нескольких переменных. Такая структура дает максимальную гибкость, но часто приводит к коду, который трудно читать. Эту конструкцию можно использовать, чтобы протестировать несколько переменных и условий.
* Необходимо использовать оператор **switch** для выполнения действий, основанных на возможных значения одной переменной. Эту конструкцию можно использовать вместо вложенных конструкций **if else if...**, чтобы сделать код более понятным при тестировании значения одной переменной.

# Урок 4: Использование операторов цикла



Для реализации логики приложения .NET Framework часто необходимо многократно выполнять части кода либо определенное количество раз, либо пока справедливо условие. Для этого можно использовать операторы цикла, предоставляемые C#. В уроке представлены три основных оператора цикла, доступные в C# и объясняется, как их можно использовать в приложениях.

## Типы операторов цикла



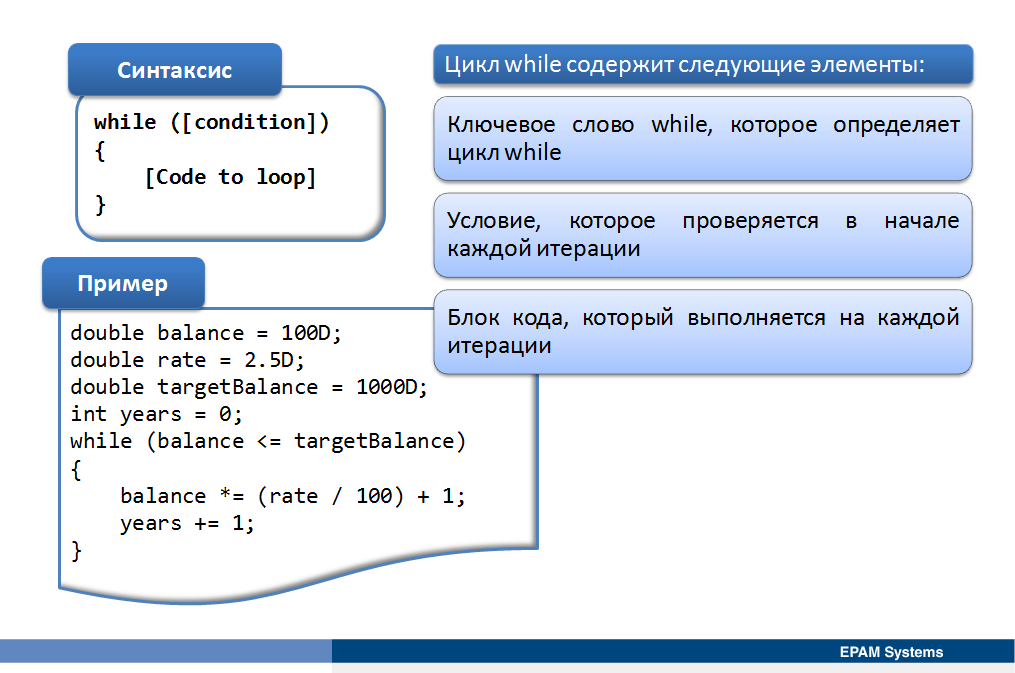
В C# существует три типа операторов цикла, которые можно использовать. Каждый из этих операторов работает по-своему и преследует различные цели.

**Цикл while.** Цикл while позволяет выполнять блок кода ноль или более раз. Цикл while не использует переменную-счетчик, хотя можно ее реализовать, определив вне цикла и манипулируя ею на каждой итерации. В начале каждой итерации цикла while, проверяется логическое условие. Если это условие истинно, выполняется тело цикла, если условие ложно, цикл завершается. Циклы while могут быть очень полезными, если заранее не известно, сколько раз нужно выполнять тело цикла.

**Цикл do.** Цикл работает аналогично циклу while за исключением одной детали. В цикле do условие проверяется в конце итерации, а не в ее начале. Это означает, что тело цикла do всегда выполняется по крайней мере один раз, в отличие от цикла while, который может не выполнится ни разу. Циклы do являются очень полезными, когда заранее не известно, сколько раз должен выполниться код и еще не определено условие. Циклы do можно использовать, например, для осуществления пользовательского ввод до тех пор, пока не будет предоставлена достоверная информация.

**Цикл for.** Цикл for позволяет выполнять код заданное количество раз. Для этого в цикле определяется переменная-счетчик, значение которой изменяется на каждой итерации. Когда переменная-счетчик достигает предельного значения, определяемого вне цикла, цикл завершается. Код в теле цикла for может использовать значение переменной-счетчика. Например, можно использовать цикл for для обработки элементов массива. Можно также использовать вложенные циклы for с различными счетчиками для обработки многомерных массивов или проверки пикселей по указанным координатам.

## Использование оператора while



Цикл while позволяет выполнять блок кода ноль или более раз. В начале каждой итерации цикл while вычисляет выражение. Если результат выражения истина, начинается следующая итерация, если оно ложно, цикл завершается. Цикл while содержит следующие элементы:

* Ключевое слово while, которое определяет цикл while.
* Условие, которое проверяется в начале каждой итерации.
* Блок кода, который выполняется на каждой итерации.

Цикл while имеет следующий синтаксис.

while ([condition])

{

[Code to loop]

}

Условием [condition] может быть любое выражение, результатом которого является логическое значение. Каждый раз, когда начинается итерация, в том числе первая, выполняется выражение. Если выражение истинно, выполняется итерация, в противном случае, цикл завершается.

Условие вычисляется для каждой итерации, перед началом итерации. Условие не контролируется во время выполнения итерации, поэтому последняя итерация всегда завершена до завершения цикла.

Следующий пример кода показывает простой расчет, который можно использовать, чтобы определить, сколько лет потребуется банковскому счету для превышения указанного значения с указанной процентной ставкой.

double balance = 100D;

double rate = 2.5D;

double targetBalance = 1000D;

int years = 0;

while (balance <= targetBalance)

{

balance \*= (rate / 100) + 1;

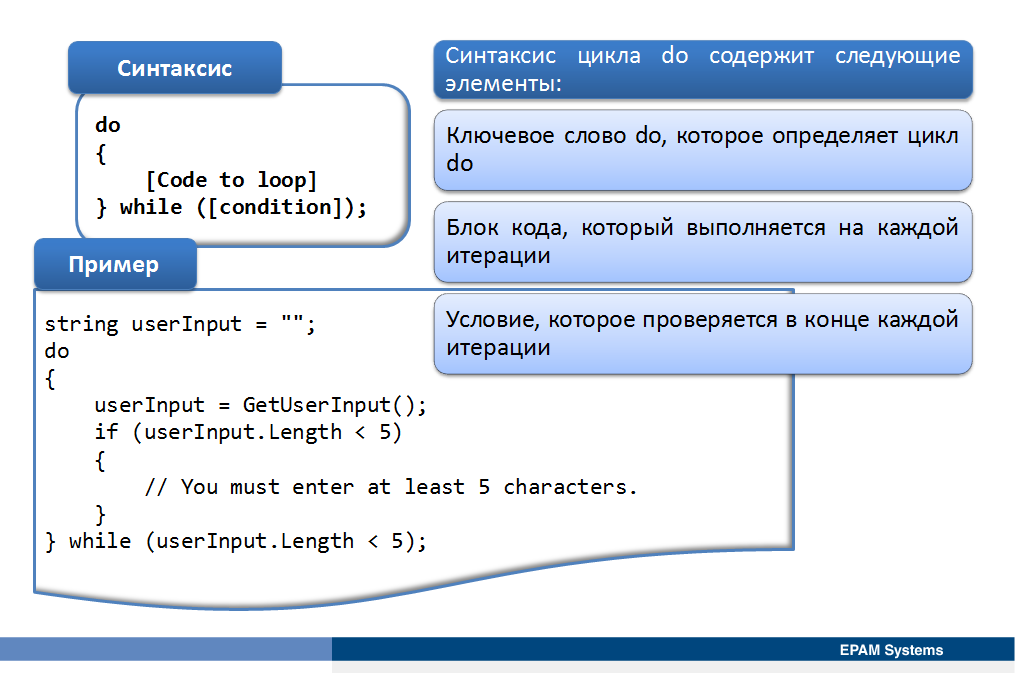
years += 1;

}

Здесь условие balance <= targetBalance проверяется перед каждой итерацией. Если значение переменной balance больше значения targetBalance перед началом цикла, итерации не будут выполняться и значение years останется равным значению 0.

При использовании цикла **while** код должен изменять булево условие, в противном случае цикл будет выполняться бесконечное число раз.

## Использование оператора do



Цикл do позволяет выполнять блок кода один или несколько раз. В конце каждой итерации цикла do вычисляется логическое выражение. Если это выражение истинно, начинается другая итерация. Если оно ложно, цикл завершается. Синтаксис цикла do содержит следующие элементы:

* Ключевое слово do, которое определяет цикл do.
* Блок кода, который выполняется на каждой итерации.
* Условие, которое проверяется в конце каждой итерации.

Цикл do имеет следующий синтаксис:

do

{

[Code to loop]

} while ([condition]);

Как и в цикле while условием [condition] может быть любое выражение, результатом которого является логическое значение. Каждый раз при завершении итерации вычисляется выражение. Если выражение истинно, выполняется следующая итерация, в противном случае цикл завершается.

Типичным примером использования цикла do является анализ ввода пользователя. Следующий пример показывает код, который требует от пользователя ввести строку длиной по крайней мере пять символов.

string userInput = "";

do

{

userInput = GetUserInput();

if (userInput.Length < 5)

{

// You must enter at least 5 characters.

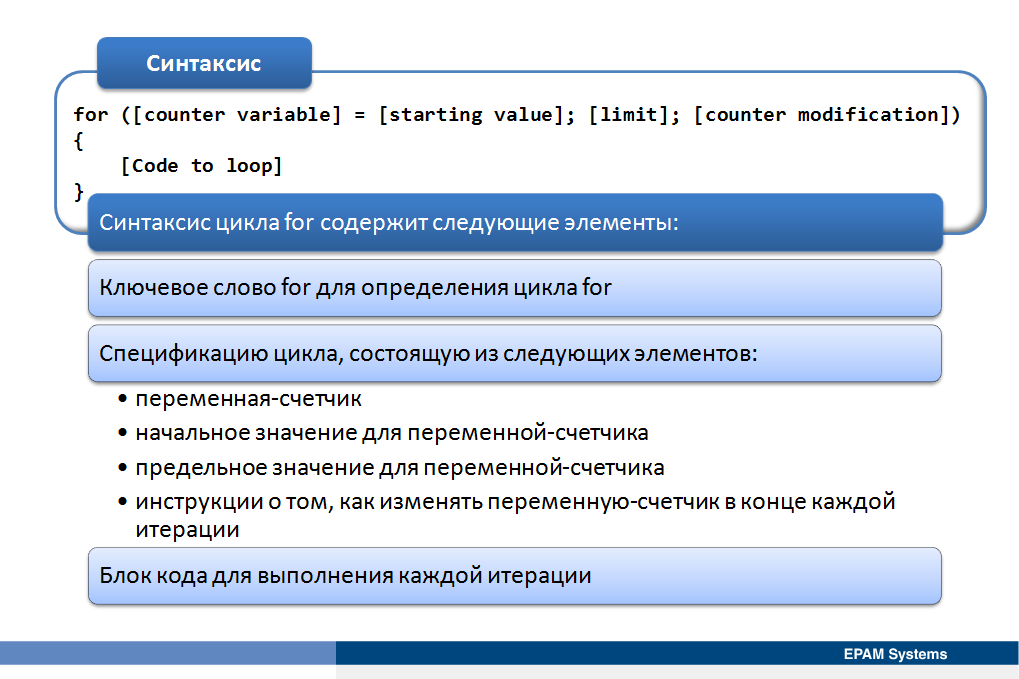
}

} while (userInput.Length < 5);

Метод GetUserInput осуществляет пользовательский ввод и возвращает строку (код метода не приводится).

При использовании цикла do код должен изменить булево условие, в противном случае цикл будет выполняться бесконечное число раз.

## Использование оператоа for



Цикл for позволяет выполнять блок кода заданное количество раз, отслеживая число выполняемых итераций с помощью переменной-счетчика. Синтаксис цикла for содержит следующие элементы:

* Ключевое слово for для определения цикла for.
* Спецификацию цикла, состоящую из следующих элементов:

1. переменная-счетчик (это может быть переменная, которая уже определена или переменная, которая определяется как часть спецификации цикла);
2. начальное значение для переменной-счетчика;
3. предельное значение для переменной-счетчика;
4. инструкции о том, как изменять переменную-счетчик в конце каждой итерации;

* Блок кода для выполнения каждой итерации.

Цикл for имеет следующий синтаксис.

for ([counter variable] = [starting value]; [limit]; [counter modification])

{

[Code to loop]

}

В таблице описаны цели заполнителей кода, представленного выше.

|  |  |
| --- | --- |
| **Заполнитель** | **Использование** |
| [counter variable] | Идентификатор существующей числовой переменной или определение новой числовой переменной. |
| [starting value] | Число, присваиваемое переменной-счетчику на первой итерации. |
| [limit] | Условие, проверяемое в начале каждой итерации. Если условие оценивается как истинное, цикл продолжается. Если оно оценивается как ложное, цикл завершается. |
| [counter modification] | Операции для выполнения в конце каждой итерации. |

Следующий пример кода показывает простой цикл for, выполняющий 10 итераций. Переменая-счетчик создается на первой итерации, устанавливается в 0 и увеличивается в конце каждой итерации на 1. Когда значение i достигает 10, цикл завершается (при i равно 10 цикл не работает).

for (int i = 0; i < 10; i++)

{

// Code to loop, which can use i.

}

Следующий код выполняется пять раз со значениями i для каждой итерации 0, 2, 4, 6 и 8.

for (int i = 0; i < 10; i = +2)

{

// Code to loop, which can use i.

}

В приведенных примерах, управляющая переменная i создается как часть конструкции for. Область видимости этой переменой – тело цикла for. Когда цикл заканчивается, переменная i не доступна. Если необходимо изучить значение управляющей переменной за пределами цикла, нужно объявить переменную перед началом цикла и использовать ее, как показано в следующем примере кода.

int j;

for (j = 0; j < 10; j++)

{

// Code to loop, which can use j.

}

// j is also available here

Можно использовать вложенные циклы for, каждый из которых определяет свою собственную переменную-счетчик. Эта идиома полезна при обработке многомерных массивов. В следующем примере показано, как использовать два вложенных цикла for для обработки в обратном порядке символов в массиве строк.

string[] strings = new string[]{"One", "Two", "Three", "Four", "Five"};

string result = "";

for (int stringIndex = 0; stringIndex < strings.Length; stringIndex++)

{

for (int charIndex = strings[stringIndex].Length - 1; charIndex >= 0; charIndex--)

{

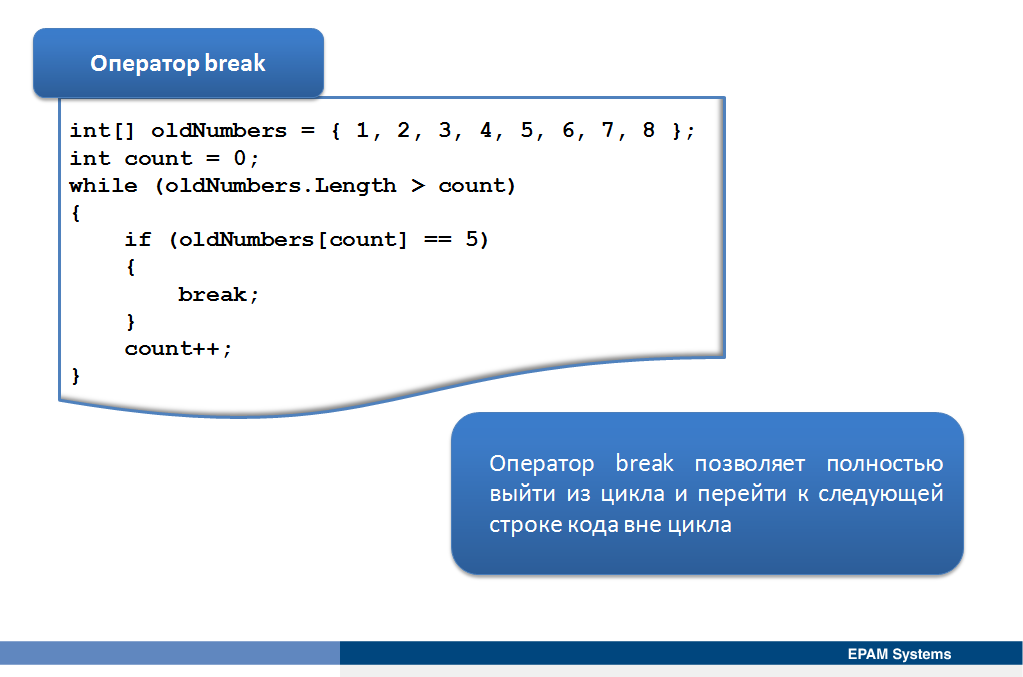
result += strings[stringIndex][charIndex];

}

}

После инициализации переменных внешний цикл for итерируется по значениями счетчика 0, 1, 2, 3, 4 (strings.Length: 5). Для каждого значения этого счетчика соответствующая строка в массиве используется для определения начального значения для счетчика внутреннего цикла. Внутренний цикл перебирает строку посимвольно, начиная с конца строки и продвигаясь в обратном порядке (управляющая переменная charIndex установлена в длину строки и уменьшается в конце каждой итерации. Цикл останавливается, когда charIndex меньше нуля. Тело внутреннего цикла извлекает символ, который индексируется charIndex от строки stringIndex в массиве. Следует отметить, что можно извлечь отдельные символы из строки, используя массивоподобный доступ с помощью индекса. Когда каждый символ в строке обработан, первая итерация внешнего цикла завершена, и внешний цикл начинает вторую итерацию. Этот процесс продолжается, пока не обработается каждый символ в каждой строке. Значением result при завершении работы кода является строка «**enOowTeerhTruoFeviF»**.

## Операторы ****break**** и continue



При использовании операторов циклов while, do и for для изменения поведения цикла можно использовать операторы break и continue. Следует помнить, что эти операторы следует использовать с осторожностью, поскольку они могут привести к коду, который трудно понимать и поддерживать.

**Оператор break.** Оператор break позволяет полностью выйти из цикла и перейти к следующей строке кода вне цикла. Оператор break особенно полезен, если происходит итеририрование по массиву в поисках записи для преждевременного (до выполнения условия окончания) выхода из цикла при ее нахождении. Не следует путать использование оператора break в цикле с его использованием в операторе **switch**. В следующем примере показан досрочный выход из цикла, когда в массиве найдено значение 5.

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

int count = 0;

while (oldNumbers.Length > count)

{

if (oldNumbers[count] == 5)

{

break;

}

count++;

}

При использовании в циклах while, do и for оператор break ведет себя одинаково.

**Оператор continue.** Оператор continueпохож на оператор break за исключением того, что, вместо выхода из цикла полностью, он пропускает оставшийся код текущей итерации, проверяет условие, и, в случае его истинности, начинает следующую итерацию цикла. В следующем примере кода показано, как добавить дополнительную логику в цикл while, который не будет выполняться, пока не найдено значение 5.

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

int count = 0;

while (oldNumbers.Length > count)

{

if (oldNumbers[count] == 5)

{

continue;

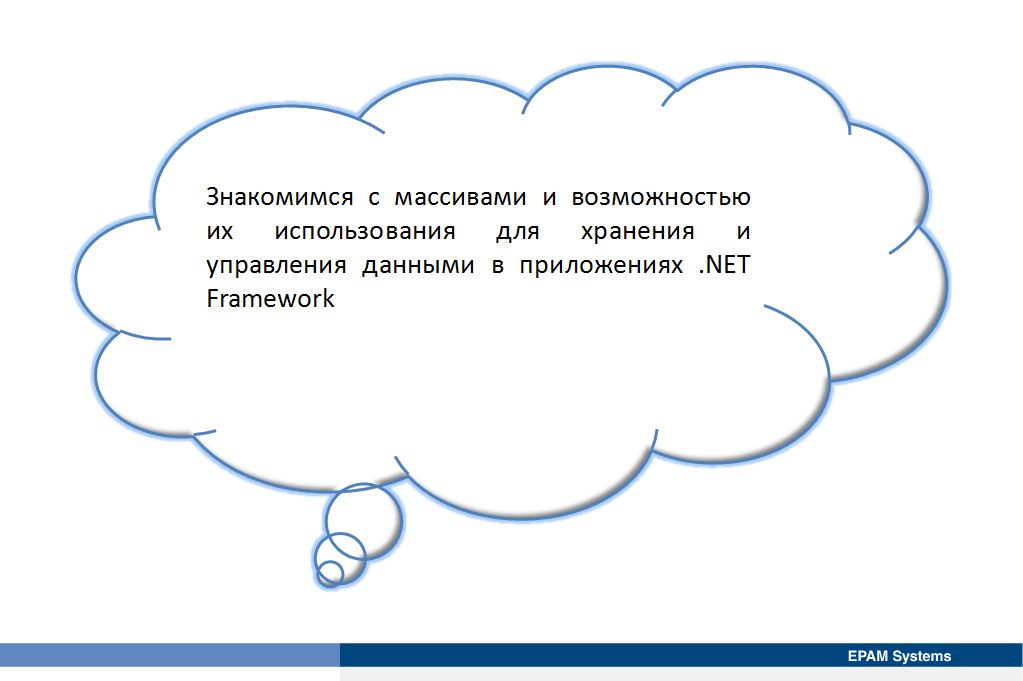
}

// Code that won't be hit when the value 5 is found count++;

}

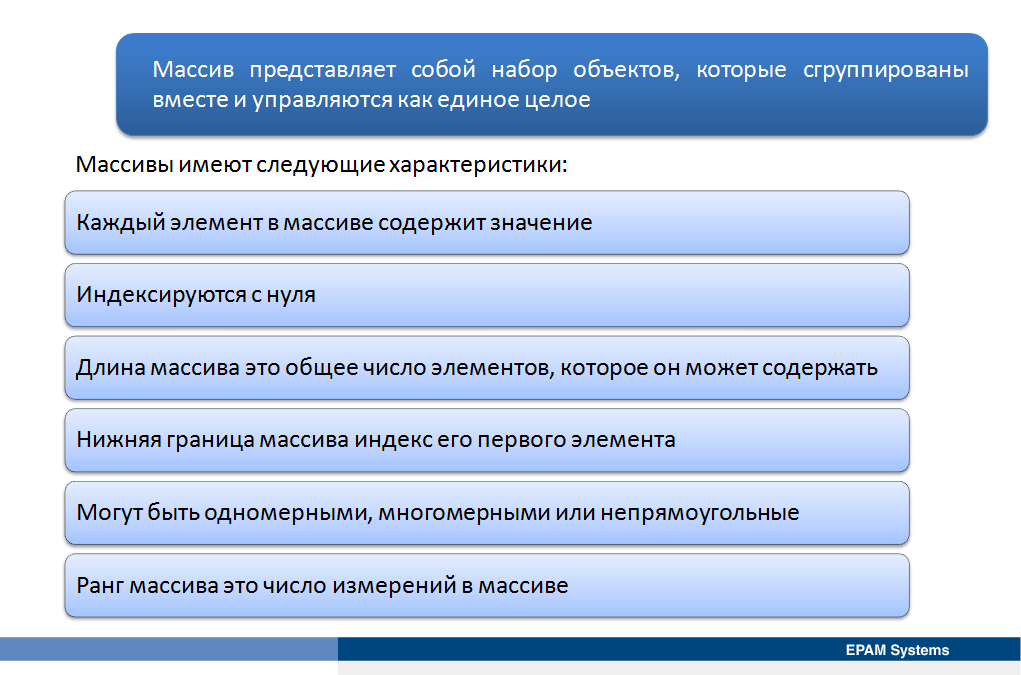
Оператор continue ведет себя одинаково при использовании в циклах while и do. Существует небольшое различие при его использовании в цикле for – в этом цикле оставшийся код текущей итерации пропускается, как и в других циклах, но модификатор переменной счетчика изменяется до проверки условия и начала следующей итерации.

# Урок 5: Создание и использование массивов



Любая переменная в текущий момент времени содержит только одно значение. Однако часто необходимо хранить и обрабатывать множество значений, о котором нельзя знать заранее, насколько большим оно является. Например, когда имеется список клиентов в базе данных, которую нужно получить и обработать. Массивы позволяют считывать и обрабатывать переменное число связанных с ним элементов данных. Урок знакомит с массивами и объясняет, как их можно использовать для хранения и управления данными.

## Что такое массив?

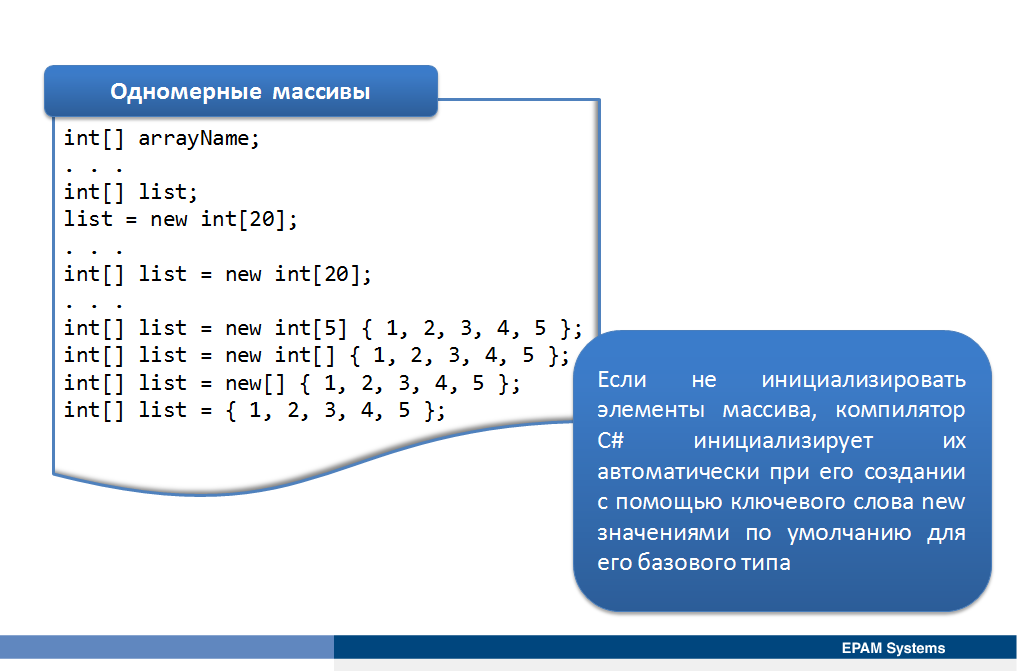


Массив представляет собой набор объектов, которые сгруппированы вместе и управляются как единое целое. О массиве можно думать как о последовательности элементов. Все элементы массива имеют одинаковый тип. Можно создавать простые массивы, которые имеют одно измерение (список), два измерения (таблицу), три измерения (куб) и так далее. Массивы имеют следующие характеристики:

* Каждый элемент в массиве содержит значение.
* Массивы индексируются с нуля. Первым индексом в массиве является 0.
* Длина массива это общее число элементов, которое он может содержать.
* Нижняя граница массива индекс его первого элемента.
* Массивы могут быть одномерными, многомерными или непрямоугольные (массивы массивов).
* Ранг массива это число измерений в массиве.

Массивы определенного типа могут содержать только элементы этого типа. Если нужно манипулировать множеством похожих объектов или типов значений, можно воспользоваться одним из типов коллекций, которые определены в пространстве имен System.Collections.

## Создание и инициализация массивов



При объявлении массива, необходимо указать тип данных, которые он содержит, и его имя. Поскольку массив – это ссылочный пользовательский тип, то его объявление помещает ссылку на массив в область видимости, но не выделяет для него память. Физически CLR создает массив при использовании ключевого слова new.

**Одномерные (single) массивы.** Для объявления одномерного массива необходимо указать тип элементов массива и использовать квадратные скобки [], чтобы указать, что переменная является массивом.

int[] arrayName;

Затем с помощью ключевого слова new следует указать размер массива и тем самым выделить для него память. Размер массива может быть любым целочисленным выражением.

int[] list;

list = new int[20];

Создание массива можно совместить с его объявлением:

int[] list = new int[20];

Кроме того можно инициализировать массив множеством значений, заданных в фигурных скобках {}. При этом можно не указывать количество элементов, а также полностью опустить указание на тип и ключевое слово new. В этом случае компилятор использует число элементов в множестве, чтобы определить размер массива.

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

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

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

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

Выше первые три примера инициализации допускают указание вместо типа переменной ключевого слова var, компилятор вычислит тип массива автоматически.

Если не инициализировать элементы массива, компилятор C# инициализирует их автоматически при его создании значениями по умолчанию для его базового типа с помощью ключевого слова new. Например, если массив содержит числовые данные, каждый элемент будет обнулен. Если массив содержит строки, каждый элемент будет инициализирован значением **null**.

**Многомерные (multiple) массивы.** Массив может иметь более одного измерения. Число измерений соответствует числу показателей, которые используются для идентификации отдельного элемента в массиве. Можно задать до 32 измерений массива, однако редко потребуется больше трех. Многомерный массив переменных можно объявить так же, как и одномерный, отделив измерения с помощью запятых.

int[,] table; // two-dimensional array

table = new int[10, 2];

int[,,] cube = new int[3, 2, 5]; // three-dimensional array

Как и для одномерного массива можно указать набор данных для каждого измерения, и компилятор использует число элементов в каждом множестве точно по размеру соответствующей размерности. При инициализации многомерных массивов множества вкладываются в фигурные скобки. В следующем примере кода показан синтаксис для объявления и инициализации многомерных массивов.

Type[ , , . . . ] arrayName1 = new Type[ Size1, Size2 , . . . ];

Type[ , , . . . ] arrayName2 = {{element1, element2, element3},

{element4, element5, element6},

...

{elementN-2, elementN-1, element}};

При добавлении размерности массиву общее хранилище для массива резко возрастает, поэтому, следует избегать объявления массива большей размерности, чем требуется.

**Массивы массивов (jagged array).** Многомерные массивы в C# могут быть прямоугольными, количество элементов в каждом измерении такого массива одинаково. Однако, C# также поддерживает непрямоугольные (jagged) массивы. Непрямоугольный массив это массив массивов, размеры которых могут различаться. Массивы массивов полезны для моделирования разреженных структур данных, когда не эффективно выделять память для каждого элемента, поскольку он не будет использоваться. Следует отметить, что IL содержит специальные инструкции для работы с одномерными массивами, индексированными с нуля. Поэтому массив массивов обрабатывается быстрее. В следующем примере кода показано, как объявить и инициализировать массив массивов. Следует обратить внимание, что необходимо указать размер первого массива, но не следует указывать размер массивов, которые содержатся в этом массиве. Память для каждого массива в пределах массива массивов выделяется отдельно, с помощью ключевого слова new.

Type [][] jaggedArray = new Type[10][];

jaggedArray[0] = new Type[5]; // Can specify different sizes

jaggedArray[1] = new Type[7];

...

JaggedArray[9] = new Type[21];

Массивы массивов можно смешивать с многомерными массивами. Ниже показано объявление и инициализация одномерного массива массивов, состоящего из двумерных элементов различного размера.

int[][,] jaggedArray = new int[3][,]

{

new int[,] {{1, 3}, {5, 7}},

new int[,] {{0, 2}, {4, 6}, {8, 10}},

new int[,] {{11, 22}, {99, 88}, {0, 9}}

};

**Неявно типизированные массивы.** Неявно типизированные массивы объявляются аналогично неявно типизированным переменным с помощью ключвого слова var. При использовании неявно типизированных массивов, тип элементов массива выводится из типа значений, указаных при его инициализации. Элементы, которые указаны при инициализации, все должны быть одного и того же типа, в противном случае компилятор сообщит об ошибке. Следующий код не скомпилируется, поскольку инициализатор содержит несколько элементов различных типов данных.

var mixed = new[] { 1, DateTime.Now, true, false, 1.2 };

В следующих примерах показано создание неявно типизированных массивов:

var a = new[] { 1, 10, 100, 1000 }; // int[]

var b = new[] { "hello", null, "world" }; // string[]

// single-dimension jagged array

var c = new[]

{

new[] {1, 2, 3, 4},

new[] {5, 6, 7, 8}

};

// jagged array of strings

var d = new[]

{

new[] {"Luca", "Mads", "Luke", "Dinesh"},

new[] {"Karen", "Suma", "Frances"}

};

Для неявно типизированных массивов квадратные скобки в левой части оператора инициализации не используются. Кроме того, инициализация массивов массивов, как и одномерных массивов, выполняется с помощью new []. Многомерные неявно типизированные массивы не поддерживаются.

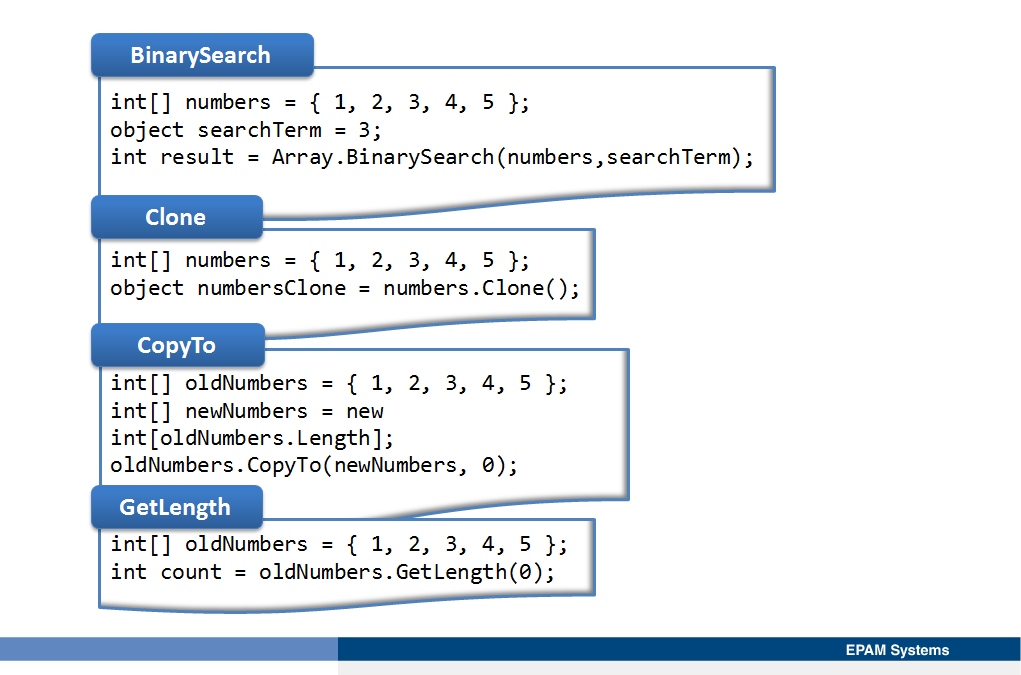
<http://go.microsoft.com/fwlink/?LinkId=192899>

<http://go.microsoft.com/fwlink/?LinkId=192900>

<http://go.microsoft.com/fwlink/?LinkId=192901>

<http://go.microsoft.com/fwlink/?LinkId=192902>

## Общие свойства и методы, предоставляемые массивами

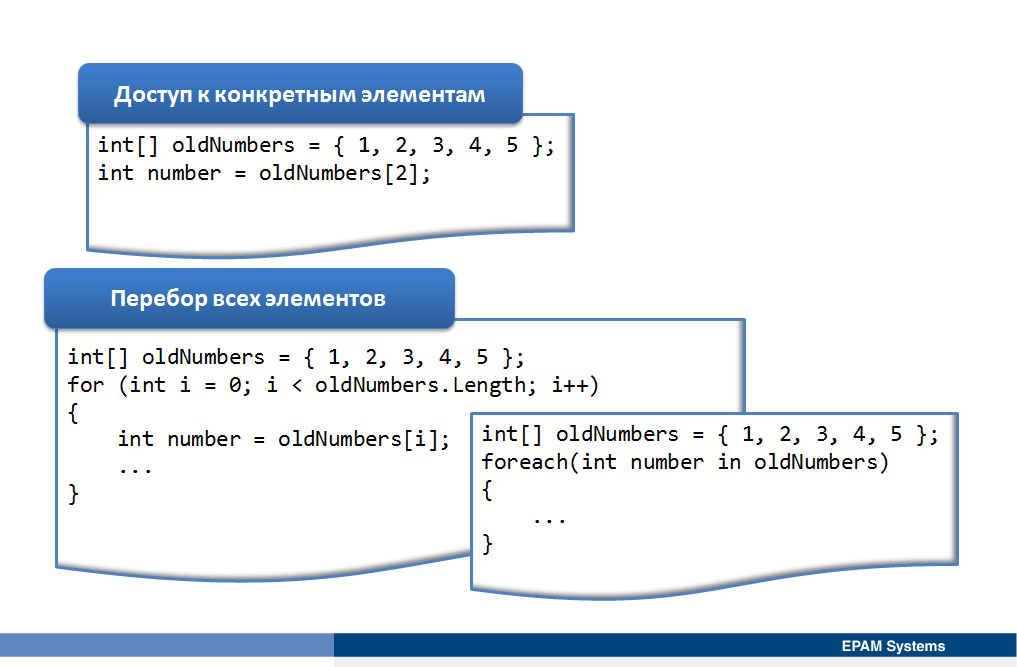


Массивы в C# являются очень полезными для хранения данных и предоставляют полезную функциональность для манипулирования данными. Все массивы в платформе .NET могут рассматриваться как классы, являющиеся потомками класса System.Array. Именно тип System.Array обеспечивает общую функциональность, которую может использовать любой пользовательский массив. В следующей таблице приведены некоторые из основных свойств и методов, которые предоставляют массивы.

|  |  |  |
| --- | --- | --- |
| **Член** | **Тип** | **Описание, пример использования** |
| **BinarySearch()** | Method | Позволяет осуществлять поиск отсортированного одномерного массива для определенного значения с помощью алгоритма бинарного поиска.  int[] numbers = { 1, 2, 3, 4, 5 };  object searchTerm = 3;  int result = Array.BinarySearch(numbers,searchTerm); |
| **Clone()** | Method | Позволяет создавать поверхностную (shallow) копию массива, которая создает только копии элементов в массиве, но не копирует объекты, на которые эти элементы могут ссылаться. (При полном (deep) копировании массива происходит копирование элементов, а также всех элементов, на которые они ссылаются явным или неявным образом)  int[] numbers = { 1, 2, 3, 4, 5 };  object numbersClone = numbers.Clone(); |
| **CopyTo()** | Method | Копирует все элементы текущего одномерного массива в заданный одномерный массив, начиная с указанного индекса в массиве назначения  int[] oldNumbers = { 1, 2, 3, 4, 5 };  int[] newNumbers = new int[oldNumbers.Length];  oldNumbers.CopyTo(newNumbers, 0); |
| **GetEnumerator()** | Method | Позволяет итерации по каждому из элементов в последовательности в массиве.  int[] oldNumbers = { 1, 2, 3, 4, 5 };  IEnumerator results = oldNumbers.GetEnumerator();  // OR  foreach (int number in oldNumbers)  {  } |
| **GetLength()** | Method | Позволяет получить длину конкретной размерности в массиве в заданном измерении.  int[] oldNumbers = { 1, 2, 3, 4, 5 };  int count = oldNumbers.GetLength(0); |
| **GetValue()** | Method | Позволяет получить значение по указанному индексу в массиве.  int[] oldNumbers = { 1, 2, 3, 4, 5 };  object number = oldNumbers.GetValue(2);  // returns the value 3 |
| **SetValue()** | Method | Позволяет установить значение по указанному индексу в массиве.  int[] oldNumbers = { 1, 2, 3, 4, 5 };  oldNumbers.SetValue(5000, 4);  // Changes the value 5 to 5000 |
| **Sort()** | Method | Позволяет сортировать элементы в одномерном массиве.  int[] oldNumbers = { 5, 2, 1, 3, 4 };  Array.Sort(oldNumbers);  // Sorted values: 1 2 3 4 5 |
| **Length** | Property | Позволяет получить количество элементов в массиве.  int[] oldNumbers = { 1, 2, 3, 4, 5 };  int numberCount = oldNumbers.Length;  // returns the value 5 |
| **Rank** | Property | Позволяет получить количество измерений в массиве.  int[] oldNumbers = { 1, 2, 3, 4, 5 };  int rank = oldNumbers.Rank;  // Returns the value 1 |

<http://go.microsoft.com/fwlink/?LinkId=192903>

## Доступ к данным в массиве



Получить доступ к данным в массиве можно несколькими способами – индексированием конкретного элемента или итерируясь по всей коллекции и возвращая каждый элемента в последовательности.

**Доступ к конкретным элементам.** Получить доступ к конкретным элементам массива можно с помощью индекса, указывающего на элемент, который нужно вернуть. Следует обратить внимание на то, что массивы индексируются с нуля, поэтому первый элемент в любой размерности массива имеет индекс нуль. Последний элемент в размерности имеет индекс N-1, где N является размером измерения. При попытке доступа к элементу вне этого диапазона, CLR генерирует исключение IndexOutOfRangeException. В следующем примере кода для доступа к элементу с индексом два используется индекс.

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

int number = oldNumbers[2];

**Перебор всех элементов.** Итерироваться через массив можно, используя цикл for. При этом для определения момена выхода из цикла используется свойство массива Length:

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

for (int i = 0; i < oldNumbers.Length; i++)

{

int number = oldNumbers[i];

...

}

Альтернативный подход состоит в использовании цикла foreach. Оператор foreach автоматически извлекает все элементы из массива в порядке индекса и присваивает их к переменной, указанной в конструкции foreach.

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

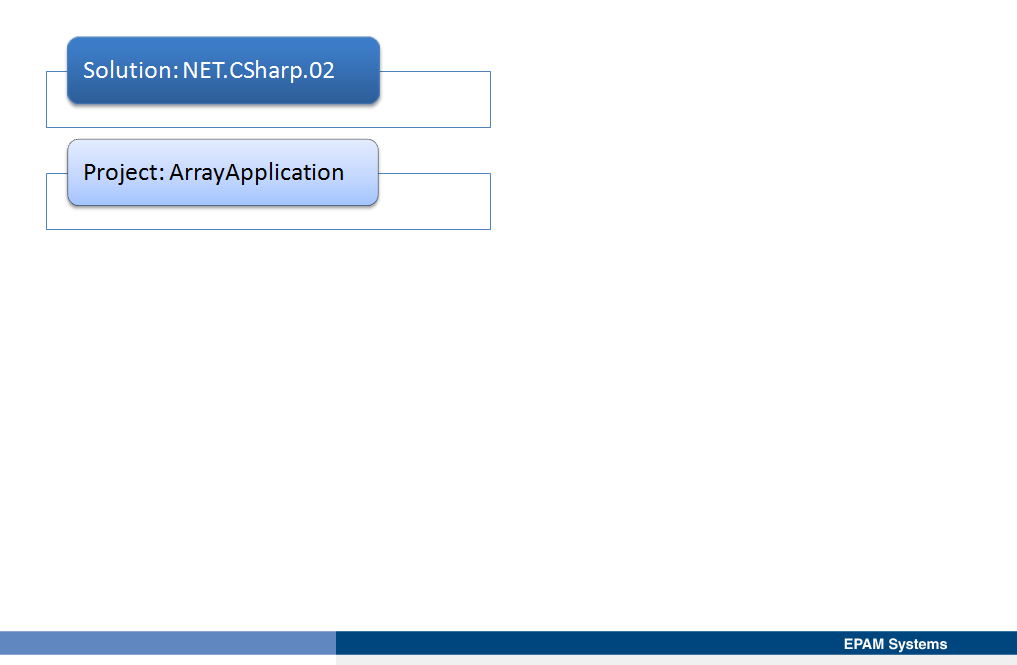
foreach(int number in oldNumbers)

{

...

}

## Демонстрация. Создание и использование массивов



# Строки

В уроке рассматривается тип данных string, являющийся псевдонимом класса String, и возможности работы с ним в приложениях.

## Использование строк

Основным типом для работы со строками является класс System.String (псевдоним в C# ‑ string). Объекты класса String объявляются, как и все прочие объекты простых типов – с явным или неявным вызовом конструктора класса. Чаще всего при объявлении строковой переменной конструктор явно не вызывается, а инициализация задается строковым литералом.

string s = "Hello, World!";

Но у класса String достаточно много конструкторов. Они позволяют сконструировать строку из массива символов или символа, повторенного заданное число раз.

Можно извлекать подстроки и объединять строки, как показано в следующем примере.

string s1 = "orange";

string s2 = "red";

s1 += s2;

Console.WriteLine(s1); // outputs "orangered"

s1 = s1.Substring(2, 5);

Console.WriteLine(s1); // outputs "anger"

Строковые объекты являются неизменяемыми: после создания их нельзя изменить. Методы, работающие со строками, возвращают новые строковые объекты. Если создать ссылку на строку, а затем «изменить» ее, то ссылка будет по-прежнему указывать на исходный объект, а не на новый объект, который был создан при изменении строки.

string s1 = "Hello";

string s2 = s1;

s1 += " and goodbye.";

Console.WriteLine(s2); //outputs "Hello"

Поскольку при изменениях строк создаются новые строковые объекты, то, в целях повышения производительности, большие объемы работы со строками (включая их объединение и другие операции) следует выполнять в классе StringBuilder, как показано в следующем примере.

StringBuilder sb = new StringBuilder();

sb.Append("one ");

sb.Append("two ");

sb.Append("three");

string str = sb.ToString();

Console.WriteLine(str);

// Outputs: one two three

## Работа со строками

Строки могут содержать escape-последовательности, такие как "\n" и "\t". Строка:

string hello = "Hello\nWorld!";

эквивалентна строке:

Hello

World!

Если требуется добавить в строку обратную косую черту, перед ней нужно поставить еще одну обратную косую черту. Следующая строка:

string filePath = "\\\\My Documents\\";

эквивалентна строке:

\\My Documents\

Для того, чтобы конструктор строк пропускал escape-последовательности и переносы строкииспользуется символ @. Следующие две строки являются идентичными:

string p1 = "\\\\My Documents\\My Files\\";

string p2 = @"\\My Documents\My Files\";

Чтобы поставить в точной строке знак двойных кавычек, нужно использовать по два таких знака, как показано в следующем примере:

string s = @"You say ""goodbye"" and I say ""hello""";

К отдельным символам строки можно получить доступ с помощью таких методов SubString и Replace.

string s3 = "Visual C# Express";

Console.WriteLine(s3.Substring(7, 2)); // outputs "C#"

Console.WriteLine(s3.Replace("C#", "Basic")); // outputs "Visual Basic Express"

Можно скопировать знаки в массив знаков.

string s4 = "Hello, World";

char[] arr = s4.ToCharArray(0, s4.Length);

foreach (char c in arr)

{

Console.Write(c); // outputs "Hello, World"

}

Возможность взятия индекса при работе со строками отражает тот факт, что строку можно рассматривать как массив и получать каждый её символ. Символ строки доступен только для чтения, но не для записи.

string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)

{

Console.Write(s5[s5.Length - i - 1]); // outputs "sdrawkcab gnitnirP"

}

Чтобы изменить регистр букв в строке (сделать их заглавными или строчными) следует использовать ToUpper или ToLower.

string s6 = "Battle of Hastings, 1066";

Console.WriteLine(s6.ToUpper()); // outputs "BATTLE OF HASTINGS 1066"

Console.WriteLine(s6.ToLower()); // outputs "battle of hastings 1066

Самый простой способ сравнения двух строк – использовать операции «**==**» и «**!=»**, осуществляющие сравнение с учетом регистра.

string color1 = "red";

string color2 = "green";

string color3 = "red";

if (color1 == color3)

{

Console.WriteLine("Equal");

}

if (color1 != color2)

{

Console.WriteLine("Not equal");

}

Для строковых объектов также существует метод CompareTo, возвращающий целочисленное значение, зависящее от того, что одна строка может быть меньше (<), равна (==) или больше другой (>). При сравнении строк используется значение Юникода, при этом значение строчных букв меньше, чем значение заглавных.

// Enter different values for string1 and string2 to

// experiement with behavior of CompareTo

string string1 = "ABC";

string string2 = "abc";

int result = string1.CompareTo(string2);

if (result > 0)

{

System.Console.WriteLine("{0} is greater than {1}", string1, string2);

}

else if (result == 0)

{

Console.WriteLine("{0} is equal to {1}", string1, string2);

}

else if (result < 0)

{

Console.WriteLine("{0} is less than {1}", string1, string2);

}

// Outputs: ABC is less than abc

Чтобы найти строку внутри другой строки следует использоватьметод IndexOf, который возвращает значение -1, если искомая строка не найдена; в противном случае возвращается индекс первого вхождения искомой строки (отсчет ведется с нуля).

string s9 = "Battle of Hastings, 1066";

Console.WriteLine(s9.IndexOf("Hastings")); // outputs 10

Console.WriteLine(s9.IndexOf("1967")); // outputs -1

Разделение строки на подстроки, аналогичное разделению предложения на отдельные слова, является распространенной задачей в программировании. Метод Split получает массив разделителей типа char (например, разделителем может быть пробел) и возвращает массив подстрок. Для доступа к этому массиву можно использовать foreach.

char[] delimit = new char[] { ' ' };

string s10 = "The cat sat on the mat.";

foreach (string substr in s10.Split(delimit))

{

System.Console.WriteLine(substr);

}

Этот код выводит каждое слово в отдельной строке.

The

cat

sat

on

the

mat.

Пустая строка – это экземпляр объекта **System..::.String**, содержащий 0 знаков. Пустые строки часто используются в различных сценариях программирования, представляя пустое текстовое поле. Для пустых строк можно вызывать методы, потому что такие строки являются допустимыми объектами **System..::.String.** Пустые строки инициализируются следующим образом:

string s = "";

Строки со значениями null (с нулевыми значениями), напротив, не ссылаются на экземпляр объекта System..::.String, любая попытка вызвать метод на строка со значением null приведет к ошибке NullReferenceException. Однако такие строки можно использовать в операциях объединения и сравнения с другими строками. В следующих примерах показаны некоторые случаи, в которых ссылка на строку со значением null вызывает либо не вызывает исключение:

string str = "hello";

string nullStr = null;

string emptyStr = "";

string tempStr = str + nullStr; // tempStr = "hello"

bool b = (emptyStr == nullStr);// b = false;

emptyStr + nullStr = ""; // creates a new empty string

int len = nullStr.Length; // throws NullReferenceException

## Использование класса StringBuilder

Класс StringBuilder создает строковый буфер, который позволяет повысить производительность, если в программе обрабатывается много строк. Класс StringBuilder также позволяет заново присваивать отдельные знаки, что не поддерживается встроенным строковым типом данных. Например, данный код заменяет содержимое строки без создания новой строки:

StringBuilder sb = new StringBuilder("Rat: the ideal pet");

sb[0] = 'C';

Console.WriteLine(sb.ToString());

Console.ReadLine();

//Outputs Cat: the ideal pet

В этом примере объект StringBuilder используется для создания строки из набора числовых типов:

class TestStringBuilder

{

static void Main()

{

StringBuilder sb = new StringBuilder();

// Create a string composed of numbers 0 - 9

for (int i = 0; i < 10; i++)

{

sb.Append(i.ToString());

}

Console.WriteLine(sb); // displays 0123456789

// Copy one character of the string (not possible with a System.String)

sb[0] = sb[9];

Console.WriteLine(sb); // displays 9123456789

}

}

## Разбор строк с помощью метода разделения

Следующий пример кода демонстрирует возможность разбора строки при помощи метода String..::.Split. Работа метода заключается в возврате массива строк, в котором каждый элемент представляет слово. В качестве ввода Split принимает массив символов, определяющих какие символы должны использоваться в качестве разделителей. В этом примере используются пробелы, запятые, точки, двоеточия и табуляция. Массив, содержащий эти разделители, передается в Split, и каждое слово в предложении выводится отдельно при помощи результирующего массива строк.

class TestStringSplit

{

static void Main()

{

char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo three:four,five six seven";

System.Console.WriteLine("Original text: '{0}'", text);

string[] words = text.Split(delimiterChars);

System.Console.WriteLine("{0} words in text:", words.Length);

foreach (string s in words)

{

System.Console.WriteLine(s);

}

}

}

Original text: 'one two three:four,five six seven'

7 words in text:

one

two

three

four

five

six

seven

**Поиск строк с помощью строковых методов**

Тип string, являющийся псевдонимом класса System..::.String, позволяет использовать несколько полезных методов для поиска содержимого строки. В следующем примере используются методы IndexOf, LastIndexOf, StartsWith и EndsWith.

class StringSearch

{

static void Main()

{

string str = "Extension methods have all the capabilities of regular static methods.";

// Write the string and include the quotation marks

System.Console.WriteLine("\"{0}\"",str);

bool test1 = str.StartsWith("extension");

System.Console.WriteLine("starts with \"extension\"? {0}", test1);

bool test2 = str.StartsWith("extension", System.StringComparison.OrdinalIgnoreCase);

System.Console.WriteLine("starts with \"extension\"? {0} (ignoring case)", test2);

bool test3 = str.EndsWith(".");

System.Console.WriteLine("ends with '.'? {0}", test3);

int first = str.IndexOf("method");

int last = str.LastIndexOf("method");

string str2 = str.Substring(first, last - first);

System.Console.WriteLine("between two \"method\" words: '{0}'", str2);

// Keep the console window open in debug mode

System.Console.WriteLine("Press any key to exit.");

System.Console.ReadKey();

}

}

'A silly sentence used for silly purposes.'

starts with 'a silly'? False

starts with 'a silly'? True (ignore case)

ends with '.'? True

between two 'silly' words: 'silly sentence used for '

**Поиск строк с помощью регулярных выражений**

Класс System.Text.RegularExpressions..::.Regex можно использовать для поиска строк. Поиск может отличаться по сложности и быть как простым, так и интенсивно использовать регулярные выражения. Ниже приведены два примера поиска строк с помощью класса Regex.

**Пример**

Следующий пример кода является консольным приложением, которое выполняет простой поиск строк в массиве без учета регистра. Статический метод Regex..::.IsMatch выполняет поиск заданной строки и строки, содержащей шаблон поиска. В этом случае третий аргумент указывает, что регистр знаков следует игнорировать.

class TestRegularExpressions

{

static void Main()

{

string[] sentences =

{

"cow over the moon",

"Betsy the Cow",

"cowering in the corner",

"no match here"

};

string sPattern = "cow";

foreach (string s in sentences)

{

System.Console.Write("{0,24}", s);

if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern, System.Text.RegularExpressions.RegexOptions.IgnoreCase))

{

System.Console.WriteLine(" (match for '{0}' found)", sPattern);

}

else

{

System.Console.WriteLine();

}

}

}

}

cow over the moon (match for 'cow' found)

Betsy the Cow (match for 'cow' found)

cowering in the corner (match for 'cow' found)

no match here

Следующий пример кода является консольным приложением, которое использует регулярные выражения для проверки формата каждой строки массива. Для выполнения проверки требуется преобразование каждой строки в формат телефонного номера, в котором три группы цифр разделены дефисами, первые две группы содержат три цифры, а третья группа — четыре цифры. Для этого используется регулярное выражение ^\\d{3}-\\d{3}-\\d{4}$.

class TestRegularExpressionValidation

{

static void Main()

{

string[] numbers =

{

"123-456-7890",

"444-234-22450",

"690-203-6578",

"146-893-232",

"146-839-2322",

"4007-295-1111",

"407-295-1111",

"407-2-5555",

};

string sPattern = "^\\d{3}-\\d{3}-\\d{4}$";

foreach (string s in numbers)

{

System.Console.Write("{0,14}", s);

if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern))

{

System.Console.WriteLine(" - valid");

}

else

{

System.Console.WriteLine(" - invalid");

}

}

}

}

123-456-7890 - valid

444-234-22450 - invalid

690-203-6578 - valid

146-893-232 - invalid

146-839-2322 - valid

4007-295-1111 - invalid

407-295-1111 - valid

407-2-5555 – invalid

**Объединение нескольких строк**

Имеется несколько способов объединения нескольких строк: перегрузка оператора + для класса String и использование класса StringBuilder. Оператор + удобен в использовании и позволяет сформировать наглядный код, но он работает последовательно. При каждом использовании оператора создается новая строка, поэтому создание цепочки из нескольких операторов неэффективно. Пример.

string two = "two";

string str = "one " + two + " three";

System.Console.WriteLine(str);

Хотя в коде фигурируют четыре строки (три объединяемых строки и одна строка результата, содержащая все три), всего создается пять строк, поскольку сначала объединяются первые две строки, создавая одну результирующую строку. Третья строка присоединяется отдельно, формируя окончательную строку, хранящуюся в str.

Можно также использовать класс StringBuilder для добавления каждой строки в объект, который затем создает результирующую строку за одно действие. Этот подход показан в следующем примере.

Пример

В следующем коде используется метод "Append" класса StringBuilder для объединения трех строк без "эффекта цепочки" оператора +.

class StringBuilderTest

{

static void Main()

{

string two = "two";

System.Text.StringBuilder sb = new System.Text.StringBuilder();

sb.Append("one ");

sb.Append(two);

sb.Append(" three");

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

string str = sb.ToString();

System.Console.WriteLine(str);

}

}

**Изменение содержимого строки**

Строки являются неизменяемыми, поэтому их содержимое изменить невозможно. Однако содержимое строки можно извлечь в форму для редактирования, выполнить изменения, а затем передать в новый экземпляр строки.

Пример

В следующем примере для извлечения содержимого строки в массив типа char используется метод ToCharArray. Затем некоторые элементы массива изменяются. После этого массив char используется для создания нового экземпляра строки.

class ModifyStrings

{

static void Main()

{

string str = "The quick brown fox jumped over the fence";

System.Console.WriteLine(str);

char[] chars = str.ToCharArray();

int animalIndex = str.IndexOf("fox");

if (animalIndex != -1)

{

chars[animalIndex++] = 'c';

chars[animalIndex++] = 'a';

chars[animalIndex] = 't';

}

string str2 = new string(chars);

System.Console.WriteLine(str2);

}

}

The quick brown fox jumped over the fence

The quick brown cat jumped over the fence

**Определение наличия числового значения в строке**

Чтобы определить, является ли строка допустимым представлением указанного числового типа, воспользуйтесь статическим методом TryParse. Этот метод реализован всеми простыми числовыми типами и такими типами, как DateTime и IPAddress. В следующем примере показано, как определить, является ли число 108 допустимым типом int.

int i = 0;

string s = "108";

bool result = int.TryParse(s, out i); //i now = 108

Если строка содержит нечисловые знаки либо числовое значение слишком велико или мало для указанного типа, TryParse возвращает значение "false" и задает выходному параметру значение "0". В противном случае возвращается значение "true", а выходному параметру задается числовое значение строки.

Строка может содержать только числовые знаки и оставаться недопустимой для типа, где используется метод TryParse. Например, "256" не является допустимым значением для byte, однако оно допустимо для int. "98,6" не является допустимым значением для int, однако оно допустимо для decimal.

В следующем примере показано использование TryParse со строковыми представлениями значений long, byte и decimal.

static void Main(string[] args)

{

string s = "1287543"; //"1287543.0" will return false for a long

long number1 = 0;

bool result = long.TryParse(s, out number1);

if (result == true)

Console.WriteLine("number1 now = {0}", number1);

else

Console.WriteLine("s is not a valid long");

byte number2 = 0;

s = "255"; // A value of 256 will return false

result = byte.TryParse(s, out number2);

if (result == true)

Console.WriteLine("number2 now = {0}", number2);

else

Console.WriteLine("s is not a valid byte");

decimal number3 = 0;

s = "27.3"; //"27" is also a valid decimal

result = decimal.TryParse(s, out number3);

if (result == true)

Console.WriteLine("number3 now = {0}", number3);

else

Console.WriteLine("number3 is not a valid decimal");

}

Надежное программирование

Простые числовые типы также реализуют статический метод Parse, который вызывает исключение, если строка не является допустимым числом. TryParse обычно более эффективен, поскольку он просто возвращает значение "false", если число не является допустимым.

Безопасность

Для проверки данных, введенных пользователем в такие элементы управления, как текстовые поля и поля со списком, всегда следует использовать метод TryParse или Parse.

# Регулярные выражения в .NET Framework

Регулярные выражения предоставляют мощный, гибкий и эффективный метод обработки текста. Обширная система обозначений для поиска соответствий позволяет выполнять быстрый разбор большого текста и поиск определенных групп символов для извлечения, изменения, замены или удаления текстовых подстрок, а также для добавления извлеченных строк в коллекцию для создания отчета. Регулярные выражения являются необходимым инструментом для многих приложений, которые выполняют строковые операции (например, обработка HTML, разбор файла журнала и разбор заголовка HTTP).

В регулярные выражения Microsoft .NET Framework включены наиболее распространенные методы, встречающиеся в других реализациях регулярных выражений, например в языках Perl и awk. Регулярные выражения .NET Framework совместимы с регулярными выражениями Perl 5 и содержат такие функции, как обратное (справа налево) сопоставление и компиляция в процессе обработки, которые еще не реализованы в других продуктах.

Классы регулярных выражений .NET Framework являются частью библиотеки базовых классов, их можно использовать с другим языком или инструментом, работающим с общеязыковой средой выполнения, например ASP.NET и Visual Studio 2005.

# Регулярные выражения в .NET Framework

Регулярные выражения предоставляют мощный, гибкий и эффективный метод обработки текста. Обширные возможности сопоставления шаблонов, предоставляемые регулярными выражениями, позволяют быстро анализировать большие объемы текста, отыскивая в них определенные символьные шаблоны, проверять текст на соответствие определенным заранее шаблонам (например, формату адреса электронной почты), извлекать, изменять, заменять или удалять подстроки текста, а также добавлять извлеченные строки в коллекцию для формирования отчетов. Регулярные выражения — это незаменимый инструмент для многих приложений, в которых ведется работа со строками или анализ объемных блоков текста.

## [Как работают регулярные выражения](javascript:void(0))

Основа обработки текста с помощью регулярных выражений – это подсистема обработки регулярных выражений, представленная в платформе .NET Framework объектом [System.Text.RegularExpressions.Regex](http://msdn.microsoft.com/ru-ru/library/system.text.regularexpressions.regex.aspx). Минимальный набор сведений, который требуется предоставить подсистеме обработки регулярных выражений для обработки текста с помощью регулярных выражений, сводится к двум вещам.

* Шаблон регулярного выражения, который требуется найти в тексте. В платформе .NET Framework шаблоны регулярных выражений определяются с использованием особого синтаксиса или языка, совместимого с регулярными выражениями Perl 5 и обладающего некоторыми дополнительными возможностями (например, он поддерживает сопоставление справа налево).
* Текст, который требуется проанализировать с помощью шаблона регулярного выражения.

Методы класса [Regex](http://msdn.microsoft.com/ru-ru/library/system.text.regularexpressions.regex.aspx) позволяют выполнять следующие действия:

* Определить, встречается ли во входном тексте шаблон регулярного выражения, можно путем вызова метода [IsMatch](http://msdn.microsoft.com/ru-ru/library/system.text.regularexpressions.regex.ismatch.aspx). Пример использования метода [IsMatch](http://msdn.microsoft.com/ru-ru/library/system.text.regularexpressions.regex.ismatch.aspx) для проверки текста. В примере задается метод IsValidEmail, который возвращает значение true, если строка содержит допустимый адрес электронной почты, и значение false, если нет, но не выполняет никаких других действий. Чтобы проверить правильность адреса электронной почты, метод вызывает метод [Regex.IsMatch(String, String)](http://msdn.microsoft.com/ru-ru/library/sdx2bds0.aspx), который проверяет, соответствует ли адрес шаблону регулярного выражения. Метод IsValidEmail можно использовать для отсеивания адресов электронной почты, содержащих недопустимые символы, перед сохранением адресов в базе данных или их отображением на странице ASP.NET.

using System;

using System.Text.RegularExpressions;

public class RegexUtilities

{

public static bool IsValidEmail(string strIn)

{

// Return true if strIn is in valid e-mail format.

return Regex.IsMatch(strIn,

@"^(?("")("".+?""@)|(([0-9a-zA-Z]((\.(?!\.))|[-!#\$%&'\\*\+/=\?\^`\{\}\|~\w])\*)(?<=[0-9a-zA-Z])@))" +

@"(?(\[)(\[(\d{1,3}\.){3}\d{1,3}\])|(([0-9a-zA-Z][-\w]\*[0-9a-zA-Z]\.)+[a-zA-Z]{2,6}))$");

}

}

. . .

public class Application

{

public static void Main()

{

string[] emailAddresses = { "david.jones@proseware.com", "d.j@server1.proseware.com",

"jones@ms1.proseware.com", "j.@server1.proseware.com",

"j@proseware.com9", "js#internal@proseware.com",

"j\_9@[129.126.118.1]", "j..s@proseware.com",

"js\*@proseware.com", "js@proseware..com",

"js@proseware.com9", "j.s@server1.proseware.com" };

foreach (string emailAddress in emailAddresses)

{

if (RegexUtilities.IsValidEmail(emailAddress))

Console.WriteLine("Valid: {0}", emailAddress);

else

Console.WriteLine("Invalid: {0}", emailAddress);

}

}

}

// The example displays the following output:

// Valid: david.jones@proseware.com

// Valid: d.j@server1.proseware.com

// Valid: jones@ms1.proseware.com

// Invalid: j.@server1.proseware.com

// Invalid: j@proseware.com9

// Valid: js#internal@proseware.com

// Valid: j\_9@[129.126.118.1]

// Invalid: j..s@proseware.com

// Invalid: js\*@proseware.com

// Invalid: js@proseware..com

// Invalid: js@proseware.com9

// Valid: j.s@server1.proseware.com

* Извлечь из текста одно или все вхождения, соответствующие шаблону регулярного выражения, можно путем вызова метода [Match](http://msdn.microsoft.com/ru-ru/library/system.text.regularexpressions.regex.match.aspx) или [Matches](http://msdn.microsoft.com/ru-ru/library/system.text.regularexpressions.regex.matches.aspx). Первый метод возвращает объект[Match](http://msdn.microsoft.com/ru-ru/library/system.text.regularexpressions.match.aspx), предоставляющий сведения о совпадении в тексте.Второй метод возвращает коллекцию [MatchCollection](http://msdn.microsoft.com/ru-ru/library/system.text.regularexpressions.matchcollection.aspx), в которую входят объекты [Match](http://msdn.microsoft.com/ru-ru/library/system.text.regularexpressions.match.aspx) для всех совпадений, найденных в проанализированном тексте.
* Заменить текст, соответствующий шаблону регулярного выражения, можно путем вызова метода [Replace](http://msdn.microsoft.com/ru-ru/library/system.text.regularexpressions.regex.replace.aspx). Примеры использования метода [Replace](http://msdn.microsoft.com/ru-ru/library/system.text.regularexpressions.regex.replace.aspx) для изменения формата даты и удаления недопустимых символов из строки.

static string MDYToDMY(string input)

{

return Regex.Replace(input,

"\\b(?<month>\\d{1,2})/(?<day>\\d{1,2})/(?<year>\\d{2,4})\\b",

"${day}-${month}-${year}");

}

. . .

using System;

using System.Globalization;

using System.Text.RegularExpressions;

public class Class1

{

public static void Main()

{

string dateString = DateTime.Today.ToString("d",

DateTimeFormatInfo.InvariantInfo);

string resultString = MDYToDMY(dateString);

Console.WriteLine("Converted {0} to {1}.", dateString, resultString);

}

static string MDYToDMY(string input)

{

return Regex.Replace(input,

"\\b(?<month>\\d{1,2})/(?<day>\\d{1,2})/(?<year>\\d{2,4})\\b",

"${day}-${month}-${year}");

}

}

// The example displays the following output to the console if run on 8/21/2007:

// Converted 08/21/2007 to 21-08-2007.

<http://msdn.microsoft.com/ru-ru/library/30wbz966.aspx>

Пример 1. Замена подстрок. Предположим, список рассылки содержит записи, в которых, помимо имени и фамилии, может указываться обращение ("Mr.", "Mrs.", "Miss" или "Ms."). Если при создании меток для конвертов по такому списку указывать обращение не требуется, можно использовать для удаления обращений регулярное выражение, как показано в следующем примере.

using System;

using System.Text.RegularExpressions;

public class Example

{

public static void Main()

{

string pattern = "(Mr\\.? |Mrs\\.? |Miss |Ms\\.? )";

string[] names = { "Mr. Henry Hunt", "Ms. Sara Samuels",

"Abraham Adams", "Ms. Nicole Norris" };

foreach (string name in names)

Console.WriteLine(Regex.Replace(name, pattern, String.Empty));

}

}

// The example displays the following output:

// Henry Hunt

// Sara Samuels

// Abraham Adams

// Nicole Norris

Шаблон регулярного выражения (Mr\.? |Mrs\.? |Miss |Ms\.? ) совпадает с любым вхождением "Mr ", "Mr. ", "Mrs ", "Mrs. ", "Miss ", "Ms или "Ms. ". Вызов метода [Regex.Replace](http://msdn.microsoft.com/ru-ru/library/system.text.regularexpressions.regex.replace.aspx) приведет к замене найденных при сопоставлении подстрок на [String.Empty](http://msdn.microsoft.com/ru-ru/library/system.string.empty.aspx); другими словами, найденная подстрока удаляется из исходной строки.

**Разбор строк**

Метод Parse преобразует строку, представляющую базовый тип .NET Framework, в реальный базовый тип .NET Framework. Поскольку разбор — это операция, обратная форматированию (преобразования базового типа в строковое представление), то применимы многие схожие правила и условия. Подобно тому, как при форматировании используется объект, реализующий интерфейс IFormatProvider для форматирования данных в строку, точно так же и при разборе используется объект, реализующий интерфейс IFormatProvider для преобразования в строковое представление. <http://msdn.microsoft.com/ru-ru/library/ms123401>

1. В .NET Framework 4 появилась возможность динамической типизации в С# с использованием нового ключевого слова dynamic. [↑](#footnote-ref-1)
2. В соответствии с терминологией, принятой при переводе англоязычной литературы по программированиию, термин operator в данном курсе переводится как «операция», а statement – как «оператор». [↑](#footnote-ref-2)