



C# PROGRAMMING

object-oriented programming

tutorialspoint
SIMPLY EASY LEARNING

About the Tutorial

C# is a simple, modern, general-purpose, object-oriented programming language developed by Microsoft within its .NET initiative led by Anders Hejlsberg. This tutorial covers basic C# programming and various advanced concepts related to C# programming language.

Audience

This tutorial has been prepared for the beginners to help them understand basics of c# Programming.

Prerequisites

C# programming is very much based on C and C++ programming languages, so if you have a basic understanding of C or C++ programming, then it will be fun to learn C#.

Disclaimer & Copyright

© Copyright 2014 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher. We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com.

Contents

About the Tutorial	i
Audience.....	i
Prerequisites.....	i
Disclaimer & Copyright	i
Contents	ii
 1. OVERVIEW.....	1
Strong Programming Features of C#	1
 2. ENVIRONMENT.....	3
The .Net Framework	3
Integrated Development Environment (IDE) for C#.....	4
Writing C# Programs on Linux or Mac OS.....	4
 3. PROGRAM STRUCTURE.....	5
Creating Hello World Program	5
Compiling and Executing the Program	6
C# Keywords	10
 4. BASIC SYNTAX.....	12
The <i>using</i> Keyword	13
The <i>class</i> Keyword	14
Comments in C#.....	14
Member Variables	14
Member Functions.....	14
Instantiating a Class	14
Identifiers	15
C# Keywords	15
 5. DATA TYPES.....	17

Value Type	17
Reference Type	18
Object Type	19
Dynamic Type	19
String Type.....	19
Pointer Type	20
6. TYPE CONVERSION	21
C# Type Conversion Methods	22
7. VARIABLES.....	24
Defining Variables.....	24
Initializing Variables.....	25
Accepting Values from User	26
Lvalue and Rvalue Expressions in C#:	26
8. CONSTANTS AND LITERALS.....	28
Integer Literals.....	28
Floating-point Literals	29
Character Constants.....	29
String Literals.....	30
Defining Constants.....	31
9. OPERATORS.....	33
Arithmetic Operators.....	33
Relational Operators.....	35
Logical Operators.....	38
Bitwise Operators	40
Assignment Operators	43
Miscillaneous Operators	46

Operator Precedence in C#	48
10. DECISION MAKING.....	51
if Statement.....	52
if...else Statement	54
The if...else if...else Statement.....	56
Nested if Statements	58
Switch Statement	60
The ? : Operator.....	65
11. LOOPS	66
While Loop	67
For Loop.....	69
Do...While Loop	72
Nested Loops	75
Loop Control Statements	78
Infinite Loop	83
12. ENCAPSULATION	84
Public Access Specifier	84
Private Access Specifier	86
Protected Access Specifier	88
Internal Access Specifier	88
13. METHODS.....	91
Defining Methods in C#.....	91
Calling Methods in C#	92
Recursive Method Call	95
Passing Parameters to a Method	96
Passing Parameters by Value	97

Passing Parameters by Reference	99
Passing Parameters by Output.....	100
14. NULLABLES.....	104
The Null Coalescing Operator (??).....	105
15. ARRAYS.....	107
Declaring Arrays	107
Initializing an Array.....	107
Assigning Values to an Array.....	108
Accessing Array Elements	108
Using the <i>foreach</i> Loop	110
C# Arrays	111
Multidimensional Arrays	112
Two-Dimensional Arrays.....	112
Jagged Arrays.....	115
Passing Arrays as Function Arguments.....	117
Param Arrays	118
Array Class	119
Properties of the Array Class.....	119
Methods of the Array Class.....	120
16. STRINGS.....	124
Creating a String Object	124
Properties of the String Class	126
Methods of the String Class	126
17. STRUCTURES	135
Defining a Structure	135
Features of C# Structures.....	137

Class versus Structure	138
18. ENUMS	141
Declaring <i>enum</i> Variable	141
19. CLASSES.....	143
Defining a Class.....	143
Member Functions and Encapsulation	145
C# Constructors	148
C# Destructors	151
Static Members of a C# Class	152
20. INHERITANCE.....	156
Base and Derived Classes.....	156
Initializing Base Class	158
Multiple Inheritance in C#.....	160
21. POLYMORPHISM.....	163
Static Polymorphism.....	163
Dynamic Polymorphism	165
22. OPERATOR OVERLOADING	170
Implementing the Operator Overloading	170
Overloadable and Non-Overloadable Operators.....	173
23. INTERFACES.....	181
Declaring Interfaces	181
24. NAMESPACES	184
Defining a Namespace	184
The <i>using</i> Keyword.....	185
Nested Namespaces.....	187

25.	PREPROCESSOR DIRECTIVES	190
	Preprocessor Directives in C#.....	190
	The #define Preprocessor	191
	Conditional Directives.....	192
26.	REGULAR EXPRESSIONS	194
	Constructs for Defining Regular Expressions	194
	Character Escapes.....	194
	Character Classes.....	196
	Grouping Constructs	198
	Quantifier	199
	Backreference Constructs	200
	Alternation Constructs.....	201
	Substitution	202
	Miscellaneous Constructs	202
	The Regex Class	203
27.	EXCEPTION HANDLING	208
	Exception Classes in C#	209
	Handling Exceptions	210
	Creating User-Defined Exceptions.....	212
	Throwing Objects.....	213
28.	FILE I/O.....	214
	C# I/O Classes	214
	The FileStream Class	215
	Advanced File Operations in C#	217
	Reading from and Writing to Text Files	218
	The StreamReader Class	218
	The StreamWriter Class	220

Reading from and Writing into Binary files	222
The BinaryWriter Class.....	224
Windows File System	228
The DirectoryInfo Class	228
The FileInfo Class	230
29. ATTRIBUTES.....	234
Specifying an Attribute	234
Predefined Attributes	234
AttributeUsage	234
Conditional	235
Obsolete	237
Creating Custom Attributes	238
Constructing the Custom Attribute	239
Applying the Custom Attribute	241
30. REFLECTION.....	243
Applications of Reflection.....	243
Viewing Metadata	243
31. PROPERTIES.....	251
Accessors	251
Abstract Properties.....	255
32. INDEXERS	259
Use of Indexers	259
Overloaded Indexers.....	262
33. DELEGATES.....	266
Declaring Delegates	266
Instantiating Delegates	266

Multicasting of a Delegate	268
Using Delegates	270
34. EVENTS.....	272
Using Delegates with Events	272
Declaring Events	272
35. COLLECTIONS	279
ArrayList Class.....	280
Hashtable Class.....	284
SortedList Class.....	288
Stack Class	292
Queue Class	295
BitArray Class.....	297
36. GENERICS	302
Features of Generics	304
Generic Methods	304
Generic Delegates.....	306
37. ANONYMOUS METHODS	309
Writing an Anonymous Method.....	309
38. UNSAFE CODES.....	312
Pointers	312
Retrieving the Data Value Using a Pointer	313
Passing Pointers as Parameters to Methods	314
Accessing Array Elements Using a Pointer	315
Compiling Unsafe Code.....	316
39. MULTITHREADING.....	318
Thread Life Cycle.....	318

Properties and Methods of the Thread Class	319
Creating Threads.....	323
Managing Threads	324
Destroying Threads.....	326

1. OVERVIEW

C# is a modern, general-purpose, object-oriented programming language developed by Microsoft and approved by European Computer Manufacturers Association (ECMA) and International Standards Organization (ISO).

C# was developed by Anders Hejlsberg and his team during the development of .Net Framework.

C# is designed for Common Language Infrastructure (CLI), which consists of the executable code and runtime environment that allows use of various high-level languages on different computer platforms and architectures.

The following reasons make C# a widely used professional language:

- It is a modern, general-purpose programming language
- It is object oriented.
- It is component oriented.
- It is easy to learn.
- It is a structured language.
- It produces efficient programs.
- It can be compiled on a variety of computer platforms.
- It is a part of .Net Framework.

Strong Programming Features of C#

Although C# constructs closely follow traditional high-level languages, C and C++ and being an object-oriented programming language. It has strong resemblance with Java, it has numerous strong programming features that make it endearing to a number of programmers worldwide.

Following is the list of few important features of C#:

- Boolean Conditions
- Automatic Garbage Collection
- Standard Library
- Assembly Versioning
- Properties and Events

- Delegates and Events Management
- Easy-to-use Generics
- Indexers Conditional Compilation
- Simple Multithreading
- LINQ and Lambda Expressions
- Integration with Windows

2. ENVIRONMENT

In this chapter, we will discuss the tools required for creating C# programming. We have already mentioned that C# is part of .Net framework and is used for writing .Net applications. Therefore, before discussing the available tools for running a C# program, let us understand how C# relates to the .Net framework.

The .Net Framework

The .Net framework is a revolutionary platform that helps you to write the following types of applications:

- Windows applications
- Web applications
- Web services

The .Net framework applications are multi-platform applications. The framework has been designed in such a way that it can be used from any of the following languages: C#, C++, Visual Basic, Jscript, COBOL, etc. All these languages can access the framework as well as communicate with each other.

The .Net framework consists of an enormous library of codes used by the client languages such as C#. Following are some of the components of the .Net framework:

- Common Language Runtime (CLR)
- The .Net Framework Class Library
- Common Language Specification
- Common Type System
- Metadata and Assemblies
- Windows Forms
- ASP.Net and ASP.Net AJAX
- ADO.Net
- Windows Workflow Foundation (WF)
- Windows Presentation Foundation
- Windows Communication Foundation (WCF)
- LINQ

For the jobs each of these components perform, please see [ASP.Net - Introduction](#), and for details of each component, please consult Microsoft's documentation.

Integrated Development Environment (IDE) for C#

Microsoft provides the following development tools for C# programming:

- Visual Studio 2010 (VS)
- Visual C# 2010 Express (VCE)
- Visual Web Developer

The last two are freely available from Microsoft official website. Using these tools, you can write all kinds of C# programs from simple command-line applications to more complex applications. You can also write C# source code files using a basic text editor like Notepad, and compile the code into assemblies using the command-line compiler, which is again a part of the .NET Framework.

Visual C# Express and Visual Web Developer Express edition are trimmed down versions of Visual Studio and has the same appearance. They retain most features of Visual Studio. In this tutorial, we have used Visual C# 2010 Express.

You can download it from [Microsoft Visual Studio](#). It gets installed automatically on your machine.

Note: You need an active internet connection for installing the express edition.

Writing C# Programs on Linux or Mac OS

Although the .NET Framework runs on the Windows operating system, there are some alternative versions that work on other operating systems. **Mono** is an open-source version of the .NET Framework which includes a C# compiler and runs on several operating systems, including various flavors of Linux and Mac OS. Kindly check [Go Mono](#).

The stated purpose of Mono is not only to be able to run Microsoft .NET applications cross-platform, but also to bring better development tools for Linux developers. Mono can be run on many operating systems including Android, BSD, iOS, Linux, OS X, Windows, Solaris, and UNIX.

3. PROGRAM STRUCTURE

Before we study basic building blocks of the C# programming language, let us look at a bare minimum C# program structure so that we can take it as a reference in upcoming chapters.

Creating Hello World Program

A C# program consists of the following parts:

- Namespace declaration
- A class
- Class methods
- Class attributes
- A Main method
- Statements and Expressions
- Comments

Let us look at a simple code that prints the words "Hello World":

```
using System;

namespace HelloWorldApplication
{
    class HelloWorld
    {
        static void Main(string[] args)
        {
            /* my first program in C# */
            Console.WriteLine("Hello World");
            Console.ReadKey();
        }
    }
}
```

When this code is compiled and executed, it produces the following result:

```
Hello World
```

Let us look at the various parts of the given program:

- The first line of the program **using System;** - the **using** keyword is used to include the **System** namespace in the program. A program generally has multiple **using** statements.
- The next line has the **namespace** declaration. A **namespace** is a collection of classes. The *HelloWorldApplication* namespace contains the class *HelloWorld*.
- The next line has a **class** declaration, the class *HelloWorld* contains the data and method definitions that your program uses. Classes generally contain multiple methods. Methods define the behavior of the class. However, the *HelloWorld* class has only one method **Main**.
- The next line defines the **Main** method, which is the **entry point** for all C# programs. The **Main** method states what the class does when executed.
- The next line */*...*/* is ignored by the compiler and it is put to add **comments** in the program.
- The Main method specifies its behavior with the statement **Console.WriteLine("Hello World");**
- *WriteLine* is a method of the *Console* class defined in the *System* namespace. This statement causes the message "Hello, World!" to be displayed on the screen.
- The last line **Console.ReadKey();** is for the VS.NET Users. This makes the program wait for a key press and it prevents the screen from running and closing quickly when the program is launched from Visual Studio .NET.

It is worth to note the following points:

- C# is case sensitive.
- All statements and expression must end with a semicolon (;).
- The program execution starts at the Main method.
- Unlike Java, program file name could be different from the class name.

Compiling and Executing the Program

If you are using Visual Studio.Net for compiling and executing C# programs, take the following steps:

- Start Visual Studio.
- On the menu bar, choose File -> New -> Project.
- Choose Visual C# from templates, and then choose Windows.
- Choose Console Application.

- Specify a name for your project and click OK button. This creates a new project in Solution Explorer.
- Write code in the Code Editor.
- Click the Run button or press F5 key to execute the project. A Command Prompt window appears that contains the line Hello World.

You can compile a C# program by using the command-line instead of the Visual Studio IDE:

- Open a text editor and add the above-mentioned code.
- Save the file as **helloworld.cs**
- Open the command prompt tool and go to the directory where you saved the file.
- Type **csc helloworld.cs** and press enter to compile your code.
- If there are no errors in your code, the command prompt takes you to the next line and generates **helloworld.exe** executable file.
- Type **helloworld** to execute your program.
- You can see the output Hello World printed on the screen.

C# is an object-oriented programming language. In Object-Oriented Programming methodology, a program consists of various objects that interact with each other by means of actions. The actions that an object may take are called methods. Objects of the same kind are said to have the same type or are said to be in the same class.

For example, let us consider a Rectangle object. It has attributes such as length and width. Depending upon the design, it may need ways for accepting the values of these attributes, calculating the area, and displaying details.

Let us look at implementation of a Rectangle class and discuss C# basic syntax:

```
using System;

namespace RectangleApplication
{
    class Rectangle
    {
        // member variables
        double length;
        double width;
        public void Acceptdetails()
        {
            length = 4.5;
        }
    }
}
```

```
    width = 3.5;

}

public double GetArea()

{

    return length * width;

}

public void Display()

{

    Console.WriteLine("Length: {0}", length);

    Console.WriteLine("Width: {0}", width);

    Console.WriteLine("Area: {0}", GetArea());

}

}

class ExecuteRectangle

{

    static void Main(string[] args)

    {

        Rectangle r = new Rectangle();

        r.Acceptdetails();

        r.Display();

        Console.ReadLine();

    }

}

}
```

When the above code is compiled and executed, it produces the following result:

```
Length: 4.5
```

Width: 3.5

Area: 15.75

The **using** Keyword

The first statement in any C# program is

```
using System;
```

The **using** keyword is used for including the namespaces in the program. A program can include multiple using statements.

The **class** Keyword

The **class** keyword is used for declaring a class.

Comments in C#

Comments are used for explaining code. Compilers ignore the comment entries. The multiline comments in C# programs start with /* and terminates with the characters */ as shown below:

```
/* This program demonstrates
   The basic syntax of C# programming
   Language */
```

Single-line comments are indicated by the // symbol. For example,

```
//end class Rectangle
```

Member Variables

Variables are attributes or data members of a class, used for storing data. In the preceding program, the *Rectangle* class has two member variables named *length* and *width*.

Member Functions

Functions are set of statements that perform a specific task. The member functions of a class are declared within the class. Our sample class *Rectangle* contains three member functions: *AcceptDetails*, *GetArea* and *Display*.

Instantiating a Class

In the preceding program, the class *ExecuteRectangle* contains the *Main()* method and instantiates the *Rectangle* class.

Identifiers

An identifier is a name used to identify a class, variable, function, or any other user-defined item. The basic rules for naming classes in C# are as follows:

- A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9) or underscore. The first character in an identifier cannot be a digit.
- It must not contain any embedded space or symbol such as ? - +! @ # % ^ & * () [] { } . ; : " ' / and \. However, an underscore (_) can be used.
- It should not be a C# keyword.

C# Keywords

Keywords are reserved words predefined to the C# compiler. These keywords cannot be used as identifiers. However, if you want to use these keywords as identifiers, you may prefix the keyword with the @ character.

In C#, some identifiers have special meaning in context of code, such as get and set are called contextual keywords.

The following table lists the reserved keywords and contextual keywords in C#:

Reserved Keywords						
abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	In	in (generic modifier)	int
interface	internal	is	lock	long	namespace	new
null	object	operator	out	out (generic modifier)	override	params
private	protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string	struct
switch	this	throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using	virtual	void
volatile	while					

Contextual Keywords						
add	alias	ascending	descending	dynamic	from	get
global	group	into	join	let	orderby	partial (type)
partial (method)	remove	select	set			

4. BASIC SYNTAX

C# is an object-oriented programming language. In Object-Oriented Programming methodology, a program consists of various objects that interact with each other by means of actions. The actions that an object may take are called methods. Objects of the same kind are said to have the same type or, more often, are said to be in the same class.

For example, let us consider an object Rectangle. It has attributes such as length and width. Depending upon the design, it may need ways for accepting the values of these attributes, calculating area, and display details.

Let us look at an implementation of a Rectangle class and discuss C# basic syntax:

```
using System;  
  
namespace RectangleApplication  
{  
  
    class Rectangle  
{  
  
        // member variables  
  
        double length;  
  
        double width;  
  
        public void Acceptdetails()  
{  
  
            length = 4.5;  
  
            width = 3.5;  
  
        }  
  
        public double GetArea()  
{  
  
            return length * width;  
  
        }  
  
        public void Display()
```

```

    {
        Console.WriteLine("Length: {0}", length);
        Console.WriteLine("Width: {0}", width);
        Console.WriteLine("Area: {0}", GetArea());
    }

}

class ExecuteRectangle
{
    static void Main(string[] args)
    {
        Rectangle r = new Rectangle();
        r.Acceptdetails();
        r.Display();
        Console.ReadLine();
    }
}

```

When the above code is compiled and executed, it produces the following result:

```

Length: 4.5
Width: 3.5
Area: 15.75

```

The **using** Keyword

The first statement in any C# program is -

```
using System;
```

The **using** keyword is used for including the namespaces in the program. A program can include multiple **using** statements.

The **class** Keyword

The **class** keyword is used for declaring a class.

Comments in C#

Comments are used for explaining code. Compiler ignores the comment entries. The multiline comments in C# programs start with /* and terminates with the characters */ as shown below:

```
/* This program demonstrates
   The basic syntax of C# programming
   Language */
```

Single-line comments are indicated by the '//' symbol. For example,

```
//end class Rectangle
```

Member Variables

Variables are attributes or data members of a class. They are used for storing data. In the preceding program, the *Rectangle* class has two member variables named *length* and *width*.

Member Functions

Functions are set of statements that perform a specific task. The member functions of a class are declared within the class. Our sample class *Rectangle* contains three member functions: *AcceptDetails*, *GetArea*, and *Display*.

Instantiating a Class

In the preceding program, the class *ExecuteRectangle* is used as a class, which contains the *Main()* method and instantiates the *Rectangle* class.

Identifiers

An identifier is a name used to identify a class, variable, function, or any other user-defined item. The basic rules for naming classes in C# are as follows:

- A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9), or underscore. The first character in an identifier cannot be a digit.
- It must not contain any embedded space or symbol like ? - +! @ # % ^ & * () [] { } . ; " ' / and \. However, an underscore (_) can be used.
- It should not be a C# keyword.

C# Keywords

Keywords are reserved words predefined to the C# compiler. These keywords cannot be used as identifiers. However, if you want to use these keywords as identifiers, you may prefix them with the @ character.

In C#, some identifiers have special meaning in context of code, such as get and set, these are called contextual keywords.

The following table lists the reserved keywords and contextual keywords in C#:

Reserved Keywords						
abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	in (generic modifier)	int
interface	internal	is	lock	long	namespace	new
null	object	operator	out	out (generic modifier)	override	params
private	protected	public	readonly	ref	return	sbyte

sealed	short	sizeof	stackalloc	static	string	struct
switch	this	throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using	virtual	void
volatile	while					

Contextual Keywords

add	alias	ascending	descending	dynamic	from	get
global	group	into	join	let	orderby	partial (type)
partial (method)	remove	select	set			

5. DATA TYPES

The variables in C#, are categorized into the following types:

- Value types
- Reference types
- Pointer types

Value Type

Value type variables can be assigned a value directly. They are derived from the class **System.ValueType**.

The value types directly contain data. Some examples are **int**, **char**, and **float**, which stores numbers, alphabets, and floating point numbers, respectively. When you declare an **int** type, the system allocates memory to store the value.

The following table lists the available value types in C# 2010:

Type	Represents	Range	Default Value
bool	Boolean value	True or False	False
byte	8-bit unsigned integer	0 to 255	0
char	16-bit Unicode character	U +0000 to U +ffff	'\0'
decimal	128-bit precise decimal values with 28-29 significant digits	(-7.9 x 10 ²⁸ to 7.9 x 10 ²⁸) / 10 ⁰ to 28	0.0M
double	64-bit double-precision floating point type	(+/-)5.0 x 10 ⁻³²⁴ to (+/-)1.7 x 10 ³⁰⁸	0.0D
float	32-bit single-precision floating point type	-3.4 x 10 ³⁸ to + 3.4 x 10 ³⁸	0.0F
Int	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0
long	64-bit signed integer type	-923,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
sbyte	8-bit signed integer type	-128 to 127	0

short	16-bit signed integer type	-32,768 to 32,767	0
uint	32-bit unsigned integer type	0 to 4,294,967,295	0
ulong	64-bit unsigned integer type	0 to 18,446,744,073,709,551,615	0
ushort	16-bit unsigned integer type	0 to 65,535	0

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** method. The expression `sizeof(type)` yields the storage size of the object or type in bytes. Following is an example to get the size of `int` type on any machine:

```
namespace DataTypeApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Size of int: {0}", sizeof(int));
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Size of int: 4
```

Reference Type

The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables.

In other words, they refer to a memory location. Using multiple variables, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this

change in value. Example of **built-in** reference types are: **object**, **dynamic**, and **string**.

Object Type

The **Object Type** is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for System.Object class. The object types can be assigned values of any other types, value types, reference types, predefined or user-defined types. However, before assigning values, it needs type conversion.

When a value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing**.

```
object obj;
obj = 100; // this is boxing
```

Dynamic Type

You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at run-time.

Syntax for declaring a dynamic type is:

```
dynamic <variable_name> = value;
```

For example,

```
dynamic d = 20;
```

Dynamic types are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run time.

String Type

The **String Type** allows you to assign any string values to a variable. The string type is an alias for the System.String class. It is derived from object type. The value for a string type can be assigned using string literals in two forms: quoted and @quoted.

For example,

```
String str = "Tutorials Point";
```

A @quoted string literal looks as follows:

```
@"Tutorials Point";
```

The user-defined reference types are: class, interface, or delegate. We will discuss these types in later chapter.

Pointer Type

Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as the pointers in C or C++.

Syntax for declaring a pointer type is:

```
type* identifier;
```

For example,

```
char* cptr;  
int* iptr;
```

We will discuss pointer types in the chapter 'Unsafe Codes'.

End of ebook preview
If you liked what you saw...
Buy it from our store @ <https://store.tutorialspoint.com>