Unit -2

C# Languages Basics

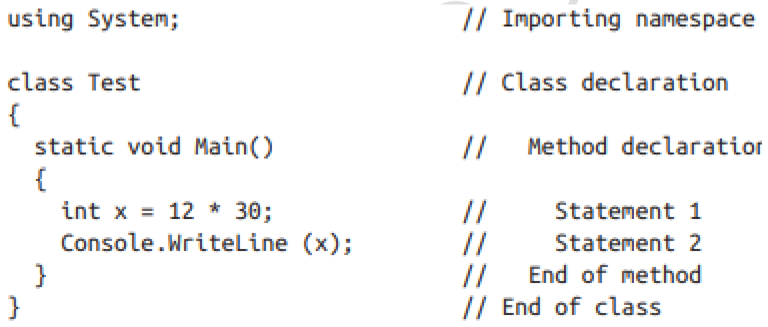
Course Contents [12 Hrs.]

Writing Console and GUI Applications; Identifiers and keywords; Writing comments; Data Types; Expressions and Operators; Strings and Characters; Arrays; Variables and Parameters; Statements (Declaration, Expression, Selection, Iteration and Jump Statements); Namespaces

---

Introduction to C# Language

Here is a program that multiplies 12 by 30 and prints the result, 360, to the screen. The double forward slash indicates that the remainder of a line is a comment.



```
using System;                  // Importing namespace

class Test                     // Class declaration
{
  static void Main()           //   Method declaratior
  {
    int x = 12 * 30;           //     Statement 1
    Console.WriteLine (x);     //     Statement 2
  }                            //   End of method
}                              // End of class
```

Writing higher-level functions that call upon lower-level functions simplifies a pro- gram. We can refactor our program with a reusable method that multiplies an integer by 12 as follows:

saralnotes.com

```
using System;

class Test
{
  static void Main()
  {
    Console.WriteLine (FeetToInches (30));      // 360
    Console.WriteLine (FeetToInches (100));     // 1200
  }

  static int FeetToInches (int feet)
  {
    int inches = feet * 12;
    return inches;
  }
}
```

Compilation

The C# compiler compiles source code, specified as a set of files with the .cs extension, into an assembly. An assembly is the unit of packaging and deployment in .NET. An assembly can be either an application or a library. A normal console or Windows application has a Main method and is an .exe file.

A library is a .dll and is equivalent to an .exe without an entry point. Its purpose is to be called upon (referenced) by an application or by other libraries. The .NET Framework is a set of libraries.

The name of the C# compiler is csc.exe. You can either use an IDE such as Visual Studio to compile, or call csc manually from the command line.

To compile manually, first save a program to a file such as *MyFirstProgram.cs*, and then go to the command line and invoke csc

(located in C:\Windows\Microsoft.NET\Framework\v4.0.30319) as follows:

csc MyFirstProgram.cs

This produces an application named **MyFirstProgram.exe**

To produce a library (.dll), do the following:

csc /target:library MyFirstProgram.cs

Identifiers and Keywords

Identifiers are names that programmers choose for their classes, methods, variables, and so on. These are the identifiers in our example program, in the order they appear:

```
System   Test   Main   x   Console   WriteLine
```

saralnotes.com

An identifier must be a whole word, essentially made up of Unicode characters starting with a letter or underscore. C# identifiers are case-sensitive. By convention, parameters, local variables, and private fields should be in camel case (e.g., myVariable), and all other identifiers should be in Pascal case (e.g., MyMethod).

Keywords are names that mean something special to the compiler. These are the keywords in our example program:

```
using   class   static   void   int
```

Most keywords are reserved, which means that you can't use them as identifiers. Here is the full list of C# reserved keywords (Total 77)

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

Avoiding conflicts

If you really want to use an identifier that clashes with a reserved keyword, you can do so by qualifying it with the @ prefix. For instance:

```
class class  {...}      // Illegal
class @class {...}      // Legal
```

The @ symbol doesn't form part of the identifier itself. So @myVariable is the same as myVariable.

Contextual keywords

saral.com

Some keywords are contextual, meaning they can also be used as identifiers— without an @ symbol. These are:

```
add         dynamic  in      orderby  var
ascending   equals   into    partial  when
async       from     join    remove   where
await       get      let     select   yield
by          global   nameof  set
descending  group    on      value
```

Literals, Punctuators, and Operators

Literals are primitive pieces of data lexically embedded into the program. The literals we used in our example program are 12 and 30.

Punctuators help demarcate the structure of the program. We used these punctuators in our example program:

```
{  }  ;
```

The **braces** group multiple statements into a statement block. The **semicolon** terminates a statement. (Statement blocks, however, do not require a semicolon.) Statements can wrap multiple lines:

```
Console.WriteLine
  (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

An **operator** transforms and combines expressions. Most operators in C# are denoted with a symbol, such as the multiplication operator, *. We will discuss operators in more detail later in this chapter. We used these operators in our example program:

```
.  ()  *  =
```

A **period** denotes a member of something (or a decimal point with numeric literals). **Parentheses** are used when declaring or calling a method; empty parentheses are used when the method accepts no arguments. An **equals sign** performs assignment. (The double equals sign, ==, performs equality comparison).

Comments

C# offers two different styles of source-code documentation: single-line comments and multiline comments. A single-line comment begins with a double forward slash and continues until the end of the line. For example:

saralnotes.com

```
int x = 3;    // Comment about assigning 3 to x
```

A multiline comment begins with /* and ends with */. For example:

```
int x = 3;    /* This is a comment that
                 spans two lines */
```

Type Basics

A type defines the blueprint for a value. In our example, we used two literals of type int with values 12 and 30. We also declared a variable of type int whose name was x:

```
static void Main()
{
  int x = 12 * 30;
  Console.WriteLine (x);
}
```

A variable denotes a storage location that can contain different values over time. In contrast, a constant always represents the same value.

```
const int y = 360;
```

All values in C# are instances of a type. The meaning of a value, and the set of possible values a variable can have, is determined by its type.

Predefined Type Examples

Predefined types are types that are specially supported by the compiler. The **int type** is a predefined type for representing the set of integers that fit into 32 bits of memory and is the default type for numeric literals within this range.

We can perform functions such as arithmetic with instances of the int type as follows:

```
int x = 12 * 30;
```

Another predefined C# type is **string**. The string type represents a sequence of characters, such as ".NET" or "http://oreilly.com". We can work with strings by calling functions on them as follows:

```
string message = "Hello world";
string upperMessage = message.ToUpper();
Console.WriteLine (upperMessage);              // HELLO WORLD

int x = 2015;
message = message + x.ToString();
Console.WriteLine (message);                   // Hello world2015
```

The predefined bool type has exactly two possible values: true and false. The bool type is commonly used to conditionally branch execution flow based with an if statement. For example:

Saralnotes.com

```
bool simpleVar = false;
if (simpleVar)
  Console.WriteLine ("This will not print");

int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
  Console.WriteLine ("This will print");
```

In C#, predefined types (also referred to as built-in types) are recognized with a C# keyword. The System namespace in the .NET Framework contains many important types that are not predefined by C# (e.g., DateTime).

Custom Type Examples

Just as we can build complex functions from simple functions, we can build complex types from primitive types. In this example, we will define a custom type named UnitConverter — a class that serves as a blueprint for unit conversions:

A type contains **data members and function members**. The data member of UnitConverter is the field called ratio. The function members of UnitConverter are the Convert method and the UnitConverter's constructor.

```
using System;

public class UnitConverter
{
  int ratio;                                                // Field
  public UnitConverter (int unitRatio) {ratio = unitRatio; } // Constructor
  public int Convert   (int unit)    {return unit * ratio; } // Method
}

class Test
{
  static void Main()
  {
    UnitConverter feetToInchesConverter = new UnitConverter (12);
    UnitConverter milesToFeetConverter  = new UnitConverter (5280);

    Console.WriteLine (feetToInchesConverter.Convert(30));    // 360
    Console.WriteLine (feetToInchesConverter.Convert(100));   // 1200
    Console.WriteLine (feetToInchesConverter.Convert(
                         milesToFeetConverter.Convert(1)));   // 63360
  }
}
```

Downloaded from Saral Notes Learn something new today. Join Group

Conversions

C# can convert between instances of compatible types. A conversion always creates a new value from an existing one. Conversions can be either implicit or explicit: implicit conversions happen automatically, and explicit conversions require a cast.

In the following example, we implicitly convert an int to a long type (which has twice the bitwise capacity of an int) and explicitly cast an int to a short type (which has half the capacity of an int):

```
int x = 12345;       // int is a 32-bit integer
long y = x;          // Implicit conversion to 64-bit integer
short z = (short)x;  // Explicit conversion to 16-bit integer
```

Implicit conversions are allowed when both of the following are true:

- The compiler can guarantee they will always succeed.
- No information is lost in conversion.

Conversely, explicit conversions are required when one of the following is true:

- The compiler cannot guarantee they will always succeed.
- Information may be lost during conversion.

Value Types Versus Reference Types

All C# types fall into the following categories:

- Value types
- Reference types
- Generic type parameters
- Pointer types

Value types comprise most built-in types (specifically, all numeric types, the char type, and the bool type) as well as custom struct and enum types.

Reference types comprise all class, array, delegate, and interface types. (This includes the predefined string type.) The fundamental difference between value types and reference types is how they are handled in memory.

Value types

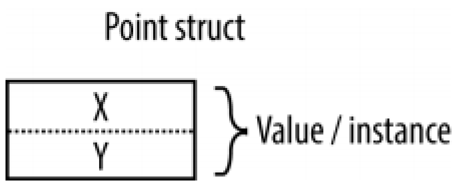
The content of a value type variable or constant is simply a value. For example, the content of the built-in value type, int, is 32 bits of data. You can define a custom value type with the **struct** keyword:

saralnotes.com

```
public struct Point { public int X; public int Y; }
```

or more tersely:

```
public struct Point { public int X, Y; }
```



The assignment of a value-type instance always copies the instance. For example:

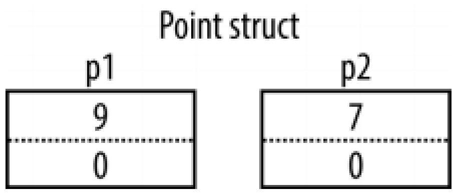
```
static void Main()
{
  Point p1 = new Point();
  p1.X = 7;

  Point p2 = p1;             // Assignment causes copy

  Console.WriteLine (p1.X);  // 7
  Console.WriteLine (p2.X);  // 7

  p1.X = 9;                  // Change p1.X

  Console.WriteLine (p1.X);  // 9
  Console.WriteLine (p2.X);  // 7
}
```

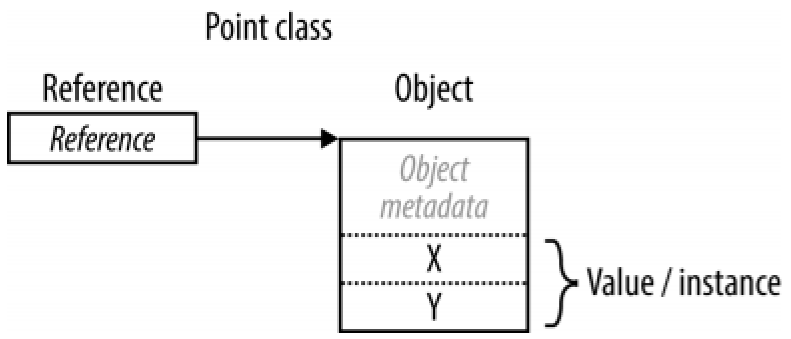


Reference types

A reference type is more complex than a value type, having two parts: an object and the reference to that object. The content of a reference-type variable or constant is a reference to an object that contains the value. Here is the Point type from our previous example rewritten as a class, rather than a struct.

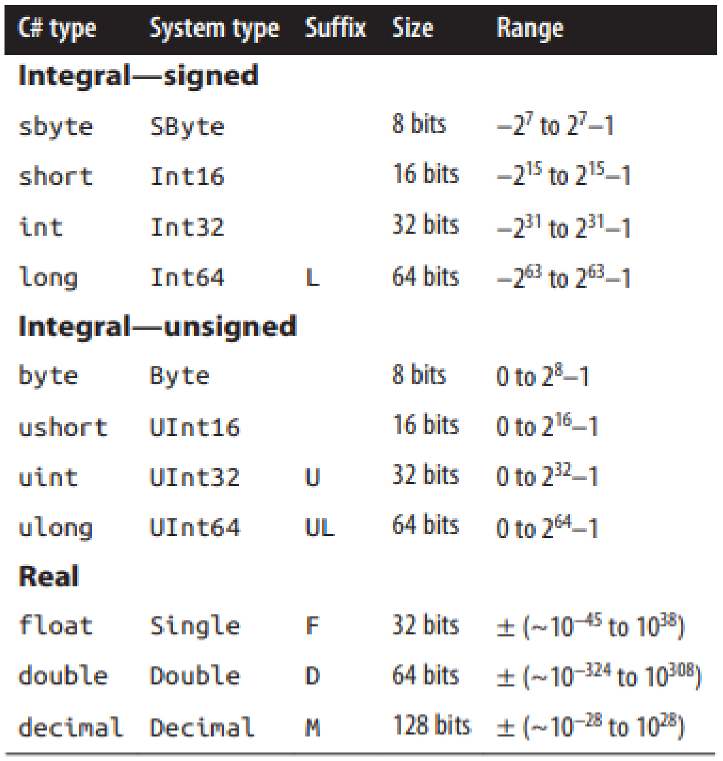
```
public class Point { public int X, Y; }
```

Downloaded from Saral Notes Learn something new today. Join Group



Numeric Types

C# has the predefined numeric types:



C# type	System type	Suffix	Size	Range
Integral—signed				
sbyte	SByte		8 bits	-2^{7} to $2^{7}-1$
short	Int16		16 bits	-2^{15} to $2^{15}-1$
int	Int32		32 bits	-2^{31} to $2^{31}-1$
long	Int64	L	64 bits	-2^{63} to $2^{63}-1$
Integral—unsigned				
byte	Byte		8 bits	0 to $2^{8}-1$
ushort	UInt16		16 bits	0 to $2^{16}-1$
uint	UInt32	U	32 bits	0 to $2^{32}-1$
ulong	UInt64	UL	64 bits	0 to $2^{64}-1$
Real				
float	Single	F	32 bits	$\pm(\sim10^{-45}$ to $10^{38})$
double	Double	D	64 bits	$\pm(\sim10^{-324}$ to $10^{308})$
decimal	Decimal	M	128 bits	$\pm(\sim10^{-28}$ to $10^{28})$

Numeric Conversions

Converting between integral types

Integral type conversions are implicit when the destination type can represent every possible value of the source type. Otherwise, an explicit conversion is required. For example:

```
int x = 12345;       // int is a 32-bit integer
long y = x;          // Implicit conversion to 64-bit integral type
short z = (short)x;  // Explicit conversion to 16-bit integral type
```

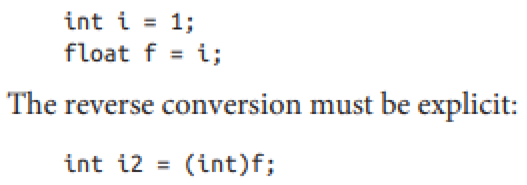
saral.com

Converting between floating-point types

A float can be implicitly converted to a double, since a double can represent every possible value of a float. The reverse conversion must be explicit.

Converting between floating-point and integral types

All integral types may be implicitly converted to all floating-point types:

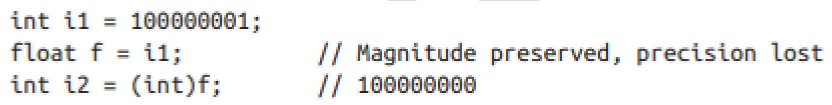


```
int i = 1;
float f = i;
```

The reverse conversion must be explicit:

```
int i2 = (int)f;
```

When you cast from a floating-point number to an integral type, any fractional portion is truncated; no rounding is per- formed. The static class System.Convert provides methods that round while converting between various numeric types. Implicitly converting a large integral type to a floating-point type preserves magnitude but may occasionally lose precision. This is because floating-point types always have more magnitude than integral types, but may have less precision. Rewriting our example with a larger number demonstrates this:



```
int i1 = 100000001;
float f = i1;          // Magnitude preserved, precision lost
int i2 = (int)f;       // 100000000
```

Decimal conversions

All integral types can be implicitly converted to the decimal type, since a decimal can represent every possible C# integral-type value. All other numeric conversions to and from a decimal type must be explicit.

saralnotes.com