

## Introduction

C# (pronounced "C sharp") is a simple, modern, object-oriented, and type-safe programming language. It will immediately be familiar to C and C++ programmers. C# combines the high productivity of Rapid Application Development (RAD) languages and the raw power of C++.

Visual C# .NET is Microsoft's C# development tool. It includes an interactive development environment, visual designers for building Windows and Web applications, a compiler, and a debugger. Visual C# .NET is part of a suite of products, called Visual Studio .NET, that also includes Visual Basic .NET, Visual C++ .NET, and the JScript scripting language. All of these languages provide access to the Microsoft .NET Framework, which includes a common execution engine and a rich class library. The .NET Framework defines a "Common Language Specification" (CLS), a sort of lingua franca that ensures seamless interoperability between CLS-compliant languages and class libraries. For C# developers, this means that even though C# is a new language, it has complete access to the same rich class libraries that are used by seasoned tools such as Visual Basic .NET and Visual C++ .NET. C# itself does not include a class library.

### Getting started

The canonical "hello, world" program can be written as follows:

```
using System;

class Hello
{
    static void Main() {
        Console.WriteLine("hello, world");
    }
}
```

The source code for a C# program is typically stored in one or more text files with a file extension of .cs, as in hello.cs. Using the command-line compiler provided with Visual Studio .NET, such a program can be compiled with the command-line directive

```
csc hello.cs
```

which produces an application named hello.exe. The output produced by this application when it is run is:

hello, world

Close examination of this program is illuminating:

- The `using System;` directive references a namespace called `System` that is provided by the Microsoft .NET Framework class library. This namespace contains the `Console` class referred to in the `Main` method. Namespaces provide a hierarchical means of organizing the elements of one or more programs. A "using" directive enables unqualified use of the types that are members of the namespace. The "hello, world" program uses `Console.WriteLine` as shorthand for `System.Console.WriteLine`.
- The `Main` method is a member of the class `Hello`. It has the `static` modifier, and so it is a method on the class `Hello` rather than on instances of this class.
- The entry point for an application — the method that is called to begin execution — is always a static method named `Main`.
- The "hello, world" output is produced using a class library. The language does not itself provide a class library. Instead, it uses a class library that is also used by Visual Basic .NET and Visual C++ .NET.

For C and C++ developers, it is interesting to note a few things that do *not* appear in the "hello, world" program.

- The program does not use a global method for `Main`. Methods and variables are not supported at the global level; such elements are always contained within type declarations (e.g., class and struct declarations).
- The program does not use either `::` or `->` operators. The `::` is not an operator at all, and the `->` operator is used in only a small fraction of programs — those that employ unsafe code ([Section A](#)). The separator `.` is used in compound names such as `Console.WriteLine`.
- The program does not contain forward declarations. Forward declarations are never needed, as declaration order is not significant.
- The program does not use `#include` to import program text. Dependencies among programs are handled symbolically rather than textually. This approach eliminates barriers between applications written using multiple languages. For example, the `Console` class need not be written in C#.

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

sample code

```
using System;
```

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

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

## Literals and Constants

The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals. Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are also enumeration constants as well.

The constants are treated just like regular variables except that their values cannot be modified after their definition.

### Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and no prefix id for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals:

```
212      /* Legal */
215u     /* Legal */
0xFeeL   /* Legal */
078      /* Illegal: 8 is not an octal digit */
032UU    /* Illegal: cannot repeat a suffix */
```

Following are other examples of various types of Integer literals:

```
85        /* decimal */
0213      /* octal */
0x4b      /* hexadecimal */
30        /* int */
30u       /* unsigned int */
30l       /* long */
30ul      /* unsigned long */
```

### Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

Here are some examples of floating-point literals:

```

3.14159      /* Legal */
314159E-5L   /* Legal */
510E         /* Illegal: incomplete exponent */
210f         /* Illegal: no decimal or exponent */
.e55         /* Illegal: missing integer or fraction */

```

While representing in decimal form, you must include the decimal point, the exponent, or both; and while representing using exponential form you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

### Character Constants

Character literals are enclosed in single quotes. For example, 'x' and can be stored in a simple variable of char type. A character literal can be a plain character (such as 'x'), an escape sequence (such as '\t'), or a universal character (such as '\u02C0').

There are certain characters in C# when they are preceded by a backslash. They have special meaning and they are used to represent like newline (\n) or tab (\t). Here, is a list of some of such escape sequence codes:

Escape sequence	Meaning
\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

Following is the example to show few escape sequence characters:

```
using System;
namespace EscapeChar
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello\tWorld\n\n");
            Console.ReadLine();
        }
    }
}
```

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

```
Hello  World
```

## String Literals

String literals or constants are enclosed in double quotes "" or with @"". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating the parts using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"
"hello, \
dear"
"hello, " "d" "ear"
@"hello dear"
```

## Defining Constants

Constants are defined using the **const** keyword. Syntax for defining a constant is:

```
const <data_type> <constant_name> = value;
```

The following program demonstrates defining and using a constant in your program:

```
using System;
namespace DeclaringConstants
{
```

```
class Program
{
    static void Main(string[] args)
    {
        const double pi = 3.14159;

        // constant declaration
        double r;
        Console.WriteLine("Enter Radius: ");
        r = Convert.ToDouble(Console.ReadLine());
        double areaCircle = pi * r * r;
        Console.WriteLine("Radius: {0}, Area: {1}", r, areaCircle);
        Console.ReadLine();
    }
}
```

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

Enter Radius:

3

Radius: 3, Area: 28.27431



## C# - Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C# has a specific type, which determines the size and layout of the variable's memory the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

The basic value types provided in C# can be categorized as:

Type	Example
Integral types	sbyte, byte, short, ushort, int, uint, long, ulong, and char
Floating point types	float and double
Decimal types	decimal
Boolean types	true or false values, as assigned
Nullable types	Nullable data types

C# also allows defining other value types of variable such as **enum** and reference types of variables such as **class**, which we will cover in subsequent chapters.

### Defining Variables

Syntax for variable definition in C# is:

```
<data_type> <variable_list>;
```

Here, data\_type must be a valid C# data type including char, int, float, double, or any user-defined data type, and variable\_list may consist of one or more identifier names separated by commas.

Some valid variable definitions are shown here:

```
int i, j, k;

char c, ch;

float f, salary;

double d;
```

You can initialize a variable at the time of definition as:

```
int i = 100;
```

### Initializing Variables

Variables are initialized (assigned a value) with an equal sign followed by a constant expression. The general form of initialization is:

```
variable_name = value;
```

Variables can be initialized in their declaration. The initializer consists of an equal sign followed by a constant expression as:

```
<data_type> <variable_name> = value;
```

Some examples are:

```
int d = 3, f = 5;    /* initializing d and f. */
byte z = 22;        /* initializes z. */
double pi = 3.14159; /* declares an approximation of pi. */
char x = 'x';        /* the variable x has the value 'x'. */
```

It is a good programming practice to initialize variables properly, otherwise sometimes program may produce unexpected result.

The following example uses various types of variables:

```
using System;
namespace VariableDefinition
{
    class Program
    {
        static void Main(string[] args)
        {
            short a;
            int b;
            double c;

            /* actual initialization */
            a = 10;
            b = 20;
            c = a + b;
            Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);
            Console.ReadLine();
        }
    }
}
```

```
}  
}
```

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

```
a = 10, b = 20, c = 30
```

### Accepting Values from User

The **Console** class in the **System** namespace provides a function **ReadLine()** for accepting input from the user and store it into a variable.

For example,

```
int num;  
num = Convert.ToInt32(Console.ReadLine());
```

The function **Convert.ToInt32()** converts the data entered by the user to int data type, because **Console.ReadLine()** accepts the data in string format.

### Lvalue and Rvalue Expressions in C#:

There are two kinds of expressions in C#:

- **lvalue**: An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.
- **rvalue**: An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and hence they may appear on the left-hand side of an assignment. Numeric literals are rvalues and hence they may not be assigned and can not appear on the left-hand side. Following is a valid C# statement:

```
int g = 20;
```

But following is not a valid statement and would generate compile-time error:

```
10 = 20;
```

## C# - 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	$(-7.9 \times 10^{28} \text{ to } 7.9 \times 10^{28}) / 100 \text{ to } 28$	0.0M
double	64-bit double-precision floating point type	$(+/-)5.0 \times 10^{-324} \text{ to } (+/-)1.7 \times 10^{308}$	0.0D
float	32-bit single-precision floating point type	$-3.4 \times 10^{38} \text{ to } + 3.4 \times 10^{38}$	0.0F
int	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0
long	64-bit signed integer type	-9,223,372,036,854,775,808 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:

```
using System;
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;
```

## C# - Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C# has rich set of built-in operators and provides the following type of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

This tutorial explains the arithmetic, relational, logical, bitwise, assignment, and other operators one by one.

### Arithmetic Operators

Following table shows all the arithmetic operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20 then:

[Show Examples](#)

Operator	Description	Example
+	Adds two operands	A + B = 30
-	Subtracts second operand from the first	A - B = -10
*	Multiplies both operands	A * B = 200
/	Divides numerator by de-numerator	B / A = 2
%	Modulus Operator and remainder of after an integer division	B % A = 0
++	Increment operator increases integer value by one	A++ = 11
--	Decrement operator decreases integer value by one	A-- = 9

### Relational Operators

Following table shows all the relational operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20, then:

[Show Examples](#)



Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

### Logical Operators

Following table shows all the logical operators supported by C#. Assume variable **A** holds Boolean value true and variable **B** holds Boolean value false, then:

[Show Examples](#)

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A    B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

### Bitwise Operators

Bitwise operator works on bits and perform bit by bit operation. The truth tables for &, |, and ^ are as follows:

p	q	p & q	p   q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; then in the binary format they are as follows:

A = 0011 1100

B = 0000 1101

-----

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The Bitwise operators supported by C# are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

[Show Examples](#)

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A   B) = 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the	(~A ) = 61,

	effect of 'flipping' bits.	which is 1100 0011 in 2's complement due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15, which is 0000 1111

### Assignment Operators

There are following assignment operators supported by C#:

Show Examples

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B assigns value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A

<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
=	bitwise inclusive OR and assignment operator	C  = 2 is same as C = C   2

### Miscellaneous Operators

There are few other important operators including **sizeof**, **typeof** and **? :** supported by C#.

#### Show Examples

Operator	Description	Example
sizeof()	Returns the size of a data type.	sizeof(int), returns 4.
typeof()	Returns the type of a class.	typeof(StreamReader);
&	Returns the address of an variable.	&a; returns actual address of the variable.
*	Pointer to a variable.	*a; creates pointer named 'a' to a variable.
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y
is	Determines whether an object is of a certain type.	If( Ford is Car) // checks if Ford is an object of the Car class.
as	Cast without raising an exception if the cast fails.	Object obj = new StreamReader("Hello");  StreamReader r = obj as StreamReader;

## Operator Precedence in C#

Operator precedence determines the grouping of terms in an expression. This affects evaluation of an expression. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example  $x = 7 + 3 * 2$ ; here,  $x$  is assigned 13, not 20 because operator  $*$  has higher precedence than  $+$ , so the first evaluation takes place for  $3*2$  and then 7 is added into it.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators are evaluated first.

### Show Examples

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^=  =	Right to left
Comma	,	Left to right

## The checked and unchecked operators

The checked and unchecked operators are used to control the overflow checking context for integral-type arithmetic operations and conversions.

*checked-expression:*

checked ( *expression* )

*unchecked-expression:*

unchecked ( *expression* )

The checked operator evaluates the contained expression in a checked context, and the unchecked operator evaluates the contained expression in an unchecked context. A checked-expression or unchecked-expression corresponds exactly to a parenthesized-expression, except that the contained expression is evaluated in the given overflow checking context.

The overflow checking context can also be controlled through the checked and unchecked statements.

The following operations are affected by the overflow checking context established by the checked and unchecked operators and statements:

- The predefined ++ and -- unary operators and, when the operand is of an integral type.
- The predefined - unary operator, when the operand is of an integral type.
- The predefined +, -, \*, and / binary operators, when both operands are of integral types.
- Explicit numeric conversions from one integral type to another integral type, or from float or double to an integral type.

When one of the above operations produce a result that is too large to represent in the destination type, the context in which the operation is performed controls the resulting behavior:

- In a checked context, if the operation is a constant expression, a compile-time error occurs. Otherwise, when the operation is performed at run-time, a `System.OverflowException` is thrown.
- In an unchecked context, the result is truncated by discarding any high-order bits that do not fit in the destination type.

For non-constant expressions (expressions that are evaluated at run-time) that are not enclosed by any checked or unchecked operators or statements, the default overflow

checking context is unchecked unless external factors (such as compiler switches and execution environment configuration) call for checked evaluation.

For constant expressions (expressions that can be fully evaluated at compile-time), the default overflow checking context is always checked. Unless a constant expression is explicitly placed in an unchecked context, overflows that occur during the compile-time evaluation of the expression always cause compile-time errors.

In the example

```
class Test
{
    static readonly int x = 1000000;
    static readonly int y = 1000000;
    static int F() {
        return checked(x * y);    // Throws OverflowException
    }
    static int G() {
        return unchecked(x * y); // Returns -727379968
    }
    static int H() {
        return x * y;             // Depends on default
    }
}
```

no compile-time errors are reported since neither of the expressions can be evaluated at compile-time. At run-time, the F method throws a `System.OverflowException`, and the G method returns `-727379968` (the lower 32 bits of the out-of-range result). The behavior of the H method depends on the default overflow checking context for the compilation, but it is either the same as F or the same as G.

In the example

```
class Test
{
    const int x = 1000000;
    const int y = 1000000;
    static int F() {
        return checked(x * y);    // Compile error, overflow
    }
    static int G() {
        return unchecked(x * y); // Returns -727379968
    }
    static int H() {
        return x * y;             // Compile error, overflow
    }
}
```

the overflows that occur when evaluating the constant expressions in F and H cause compile-time errors to be reported because the expressions are evaluated in a checked context. An overflow also occurs when evaluating the constant expression in G, but since the evaluation takes place in an unchecked context, the overflow is not reported.

The checked and unchecked operators only affect the overflow checking context for those operations that are textually contained within the "(" and ")" tokens. The operators have no effect on function members that are invoked as a result of evaluating the contained expression. In the example

```
class Test
{
    static int Multiply(int x, int y) {
        return x * y;
    }
}
```



```
}  
  
static int F() {  
    return checked(Multiply(1000000, 1000000));  
}  
  
}
```

the use of checked in F does not affect the evaluation of  $x * y$  in Multiply, so  $x * y$  is evaluated in the default overflow checking context.

The unchecked operator is convenient when writing constants of the signed integral types in hexadecimal notation. For example:

```
class Test  
{  
    public const int AllBits = unchecked((int)0xFFFFFFFF);  
    public const int HighBit = unchecked((int)0x80000000);  
}
```

Both of the hexadecimal constants above are of type uint. Because the constants are outside the int range, without the unchecked operator, the casts to int would produce compile-time errors.

The checked and unchecked operators and statements allow programmers to control certain aspects of some numeric calculations. However, the behavior of some numeric operators depends on their operands' data types. For example, multiplying two decimals always results in an exception on overflow even within an explicitly unchecked construct. Similarly, multiplying two floats never results in an exception on overflow even within an explicitly checked construct. In addition, other operators are *never* affected by the mode of checking, whether default or explicit.

## Expressions (C# Programming Guide)

An *expression* is a sequence of one or more operands and zero or more operators that can be evaluated to a single value, object, method, or namespace. Expressions can consist of a literal value, a method invocation, an operator and its operands, or a *simple name*. Simple names can be the name of a variable, type member, method parameter, namespace or type.

Expressions can use operators that in turn use other expressions as parameters, or method calls whose parameters are in turn other method calls, so expressions can range from simple to very complex. Following are two examples of expressions:

```
((x < 10) && ( x > 5)) || ((x > 20) && (x < 25))
```

```
System.Convert.ToInt32("35")
```

### Expression Values

In most of the contexts in which expressions are used, for example in statements or method parameters, the expression is expected to evaluate to some value. If *x* and *y* are integers, the expression *x + y* evaluates to a numeric value. The expression *new MyClass()* evaluates to a reference to a new instance of a *MyClass* object. The expression *myClass.ToString()* evaluates to a string because that is the return type of the method. However, although a namespace name is classified as an expression, it does not evaluate to a value and therefore can never be the final result of any expression. You cannot pass a namespace name to a method parameter, or use it in a new expression, or assign it to a variable. You can only use it as a sub-expression in a larger expression. The same is true for types (as distinct from [System.Type](#) objects), method group names (as distinct from specific methods), and event [add](#) and [remove](#) accessors.

Every value has an associated type. For example, if *x* and *y* are both variables of type **int**, the value of the expression *x + y* is also typed as **int**. If the value is assigned to a variable of a different type, or if *x* and *y* are different types, the rules of type conversion are applied. For more information about how such conversions work, see [Casting and Type Conversions \(C# Programming Guide\)](#).

### Overflows

Numeric expressions may cause overflows if the value is larger than the maximum value of the value's type. For more information, see [Checked and Unchecked \(C# Reference\)](#) and [Explicit Numeric Conversions Table \(C# Reference\)](#).

## Branching and Looping in C#

Certainly you are familiar with if, switch, while, return and goto statements. These are famous Branching and Looping statements in many programming languages for selecting code paths conditionally, looping and jumping to another part of your application unconditionally. In this article we will discuss what C# offers to us from these statements. As you will see, the common programming errors that existed in C++ were eliminated with C# statements.

### The if Statement

The if statement gives you the ability to test an expression, then select a statement or a group of statements to be executed if the expression evaluates to true. If you have the else part of the statement, you can execute another statement (or statements) if the expression evaluates to false. The syntax of if statement is as follows:

```
if(bool expression)
statement 1;
```

As we will see shortly, the expression must evaluate to bool value (true or false). If you have more than one statement, you must put the statements in a block using curly braces:

```
if(bool expression)
{
    statement 1;
    statement 2;
    statement 3;
}
```

The same applies to the else statement, so if you have only one statement to execute with the else statement, you can write it like this:

```
if(bool expression)
    statement 1;
else
    statement 1;
```

But if you have multiple statements in your else part, you can write it as:

```
if(bool expression)
{
    statement 1;
    statement 2;
    statement 3;
}
```

```
else
{
    statement 1;
    statement 2;
    statement 3;
}
```

For consistency, use the curly braces even if you have only one statement with the if statement or with the else statement. Most programmers prefer this usage.

{mospagebreak title= No Implicit Conversion}

C# eliminates a common source of programming errors that existed in the world of C and C++, which is the implicit conversion of numeric values to Boolean values (0 to false and all the other numeric values to true). In C#, the if statement's expression must evaluate to a Boolean value, so the next block of code will not compile in C# but it will compile in C++:

```
int x = 10;
if(x)
{
    Console.WriteLine(x);
}
```

In C# you will get a compile time error that states "Can't implicitly convert type 'int' to 'bool'". You need to write this block as follows:

```
int x = 10;
if(x > 5)
{
    Console.WriteLine(x);
}
```

This will compile in C# because now it's a Boolean expression. Let's look at an example. The following example takes an input number from the user, tests to see if it's an even or odd number (using the module operator), and prints a line in each case.

```
using System;
namespace MyCompany
{
    public class IfStatement
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Please type a number");
            int x = Convert.ToInt32(Console.ReadLine());
```

```
    if(x % 2 == 0)
    {
        Console.WriteLine("this is an EVEN number");
    }
    else
    {
        Console.WriteLine("this is an ODD number");
    }
    Console.ReadLine();
}
}
```

{mospagebreak title=Nesting}

Of course you can nest if statements as much as you want. Let's take a look:

```
using System;
namespace MyCompany
{
    public class IfStatement
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Please type a number");
            int x = Convert.ToInt32(Console.ReadLine());
            if(x % 2 == 0)
            {
                if(x < 1000)
                {
                    Console.WriteLine("this is an EVEN number");
                }
                else
                {
                    Console.WriteLine("this is a big EVEN number");
                }
            }
            else
            {
                if(x < 1000)
                {
                    Console.WriteLine("this is an ODD number");
                }
                else
                {

```

```
        Console.WriteLine("this is a big ODD number");
    }
}
Console.ReadLine();
}
}
```

You can combine else with if in one statement to execute more than one test on the expression. As you already know, if tests the expression, and if it evaluates to false, the else block will be executed. Sometimes this is not what we need, however. We may need to be very specific with the else part. For example, take a look at the following code:

```
int x = 5;
if(x != 0)
{
    Console.WriteLine("x != 0");
}
else
{
    Console.WriteLine("x = 0");
}
```

There is nothing special about this code; but notice that with the if statement we test to see if the x is not equal to zero, and if so, it will execute the if block, and else it will execute the else block. Now let's look at the following code:

```
using System;
namespace MyCompany
{
    public class IfStatement
    {
        static void Main(string[] args)
        {
            int x = 5;
            if(x == 0)
            {
                Console.WriteLine("x != 0");
            }
            else if(x == 4)
            {
                Console.WriteLine("x == 4");
            }
            else if(x == 5)
            {
```

```
        Console.WriteLine("x == 5");
    }
    else
    {
        Console.WriteLine("x = 0");
    }
    Console.ReadLine();
}
}
```

if you compile and run the application, you will get the following in the console window:



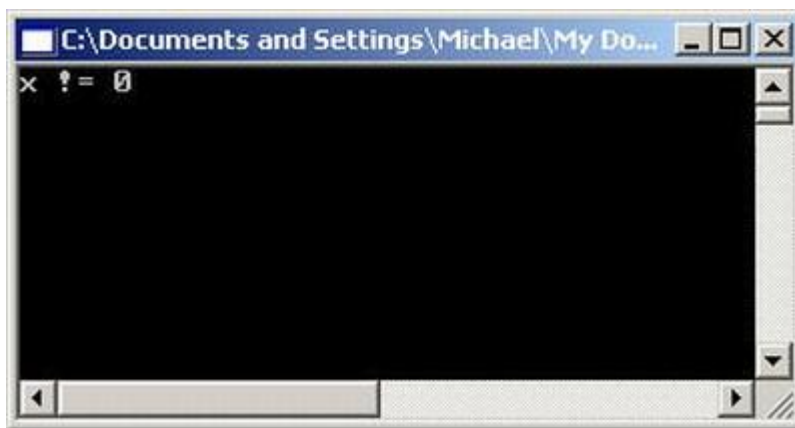
C# features the combination of else with if to further test the Boolean expression (only when the if statement evaluates to false), so if the if Boolean expression evaluated to true, the else if statements will never be tested. I will rewrite the above code as follows:

```
using System;
namespace MyCompany
{
    public class IfStatement
    {
        static void Main(string[] args)
        {
            int x = 5;
            if(x != 0)
            {
                Console.WriteLine("x != 0");
            }
            else if(x == 4)
            {
                Console.WriteLine("x == 4");
            }
            else if(x == 5)
            {

```

```
        Console.WriteLine("x == 5");
    }
    else
    {
        Console.WriteLine("x = 0");
    }
    Console.ReadLine();
}
}
```

You will get the following result:



This is an unexpected result, because x is assigned 5. It is true that x is assigned 5, but the code was not perfect; the if statement tested to check if x is not equal to 0, and it's not equal to zero, so it escaped the next else if statement, and that's why we got this value. With this in mind, never use an expression that is misleading while using else if statements.

{mospagebreak title=The switch Statement}

The switch statement is similar to multiple else if statements, but it accepts only expressions that will evaluate to constant values. The if/else structure is appropriate for small groups of Boolean expressions, but it would be very ugly to write 20 else if statements to test for some values: else if(x ==5) do something else if(x == 6) do another thing, and so on. This is the syntax of a switch statement:

```
switch(expression)
{
    case constant-value:
        statement 1;
        statement 2;
        jump statement
    case constant-value:
```



```
    statement 1;
    statement 2;
    jump statement
default:
    statement 1;
    statement 2;
    jump statement
}
```

There are more than three keywords you can use with the switch statement: first, the switch keyword which, followed by an expression that returns an integral value of type sbyte, byte, short, ushort, int, uint, long, ulong, char, string, or an Enumeration value. The switch expression's value can be returned from a method call, too.

After the switch expression is evaluated, the value will be compared to each case label, and when the value is found, control is transferred to the first statement in the this case statement. Each case statement must be ended with a jump statement (unlike C and C++), such as break or goto. The case keyword is used to define a constant value (called case label) that will be compared to the value returns from the switch expression. The constant value of the case label must have a compatible value with the switch block. This makes sense, because you can't define an int case label while the switch expression returns a string value. We will talk about jump statements later in the article.

The default keyword declares the default statement, which will be executed if none of the case labels match the switch expression. There can be only one default statement for each switch statement, and you can write your switch statement without a default statement, too. Let's take a look at an example of a switch statement with our famous Employee class example:

```
using System;
namespace MyCompany
{
    public enum Jobs
    {
        Programmer = 1,
        NetworkAdmin = 2,
        Designer = 3,
        COMDeveloper = 4
    }

    public class Employee
    {
        public Positions EmpJob;

        // class constructor
    }
}
```

```
public Employee(Jobs EJ)
{
    switch(EJ)
    {
        case Jobs.Programmer:
            Console.WriteLine("You are a programmer");
            break;
        case Jobs.NetworkAdmin:
            Console.WriteLine("You are a network administrator");
            break;
        case Jobs.Designer:
            Console.WriteLine("You are a designer");
            break;
        default:
            Console.WriteLine("You are an employee");
            break;
    }
}

public class EmployeeTest
{
    static void Main(string[] args)
    {
        Employee Michael = new Employee(Jobs.Designer);
        Console.ReadLine();
    }
}
```

This is a very simple example. The Employee class constructor accepts a value of type Positions, and defines a switch case block to print the Employee job title. One very important issue to notice is that, with the if/else structure you can write as many else if statements, each with different conditional expressions, as you want — but you can't do that with the switch/case structure. Here the case statements just test for the value of the switch expression, and if the comparison succeeds, the block get executed. If you have a statement that will be executed with more than one case label, you can combine case labels in the following way:

```
switch(EP)
{
    case Positions.Programmer:
    case Positions.NetworkAdmin:
    case Positions.Designer:
        Console.WriteLine("You are an IT Employee");
}
```

```
    break;
default:
    Console.WriteLine("You are an employee");
    break;
}
```

You simply write a case statement followed by the next, until you write them all; then write the statement you want to be executed, then a jump statement. In C# you must use jump statements between case statements, because the designers of C# eliminated the fall through that has been in C and C++. You can still explicitly state that you need the functionality of fall through by using the goto jump keyword:

```
switch(EP)
{
    case Positions.Programmer:
        goto case Positions.Designer;
    case Positions.NetworkAdmin:
        goto case Positions.Designer;
    case Positions.Designer:
        Console.WriteLine("You are an IT Employee");
        break;
    default:
        Console.WriteLine("You are an employee");
        break;
}
```

This is an explicit fall through that is clear for every programmer and in fact this technique eliminates the common problems associated with fall through.

{mospagebreak title=Looping Statements}

Executing a statement (or a group of statements) a number of times is called looping, and there are four statements for looping and iteration in C#, so let's begin with the for statement.

### The for Statement

This is the famous for loop structure that many of you have been using to write your own algorithms. The for loop is used when the number of loops is known (unlike the while loop, as we will see later). The syntax of the for loop is:

```
for (initialization; conditional expression; stepping)
{
    statement 1;
    statement 2;
}
```

As you can see, the initialization, the expression that controls the loop, and the stepping parts are all defined in the top of the statement and in one location. The parts are also separated by semicolons, and you begin the looped statements, or the statements that will be executed in the loop, using a block ( { } ). The first part of the for statement is the initialization part; use this part to initialize the variables that will be used in the algorithm of the for loop. Note that these variables are allocated on the stack, and this part will be executed only one time, because it doesn't make any sense to declare and initialize the same variable with the same value each time in the loop.

The next part is the conditional expression, which determines whether the loop will be executed again or not. If the condition (something like `i < myArray.Length`) evaluated to false, control passes to the first statement after the for statement block. If the condition evaluated to true, the body of the for block (the controlled statements) will be executed.

The last part is the stepping part. Usually it will be a counter that will increment the variable that the conditional expression uses, because at some point we need the conditional expression to evaluate to false and terminate the execution of the for loop. Note that the stepping part executes after the controlled statements. That is, first the initialization part executes (one time only) and the variables allocate space on the stack; second, the conditional expression is evaluated, and if true, the controlled statements execute, followed by the stepping part execution, and again the conditional expression is evaluated and the process iterates.

Note that any of the three parts that make the for statement can be empty; although it's not common, it can happen. Take a look:

```
static void Main(string[] args)
{
    int x = 10;
    for(; x < 100; x += 10)
    {
        Console.WriteLine(x);
    }
    Console.ReadLine();
}
```

Compile the method into a class and run the application; you will get the following result:

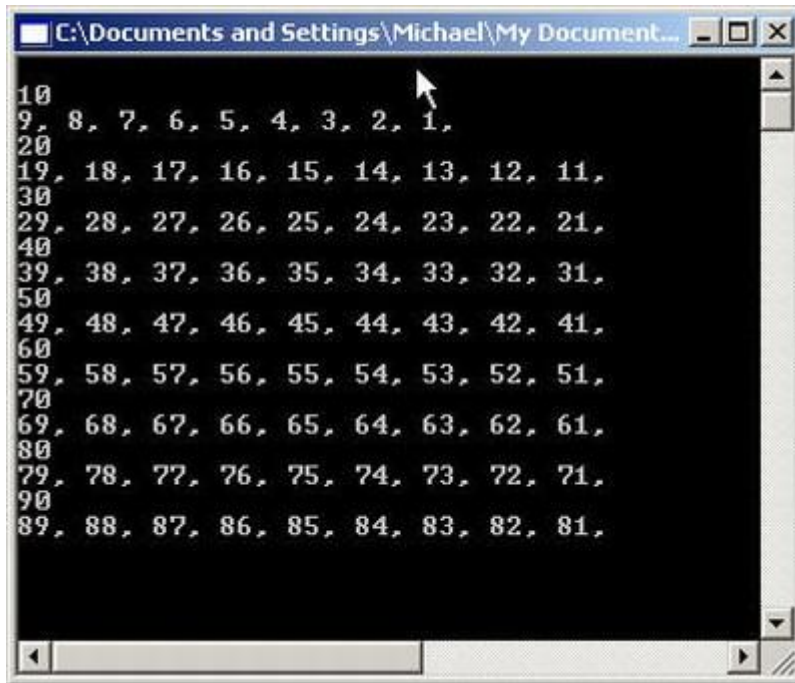


As you can see, we omit the initialization part and we just put in the semicolon. Also note that the counter can increment or decrement, and it's up to you to define the algorithm.

The for loop can be used to define very complex algorithms; this happens as a result of nesting the for loops. Let's take a very simple example which extends the above loop example. It will simply write the same numbers, but this time with a little difference.

```
public class Loops
{
    static void Main(string[] args)
    {
        for(int x = 10; x < 100; x += 10)
        {
            Console.WriteLine();
            Console.WriteLine(x);
            for(int y = x - 1, temp = x - 10; y > temp; y--)
            {
                Console.Write("{0}, ", y);
            }
            Console.ReadLine();
        }
    }
}
```

Compile the class and run it, and you will get the following result:



```
C:\Documents and Settings\Michael\My Document...
10
9, 8, 7, 6, 5, 4, 3, 2, 1,
20
19, 18, 17, 16, 15, 14, 13, 12, 11,
30
29, 28, 27, 26, 25, 24, 23, 22, 21,
40
39, 38, 37, 36, 35, 34, 33, 32, 31,
50
49, 48, 47, 46, 45, 44, 43, 42, 41,
60
59, 58, 57, 56, 55, 54, 53, 52, 51,
70
69, 68, 67, 66, 65, 64, 63, 62, 61,
80
79, 78, 77, 76, 75, 74, 73, 72, 71,
90
89, 88, 87, 86, 85, 84, 83, 82, 81,
```

I have used a nest for loop to print all the numbers between two iterations from the outer for loop. The ability to use for loops actually makes great programmers

## C# - Methods

A method is a group of statements that together perform a task. Every C# program has at least one class with a method named Main.

To use a method, you need to:

- Define the method
- Call the method

### Defining Methods in C#

When you define a method, you basically declare the elements of its structure. The syntax for defining a method in C# is as follows:

```
<Access Specifier> <Return Type> <Method Name>(Parameter List)

{
    Method Body
}
```

Following are the various elements of a method:

- **Access Specifier:** This determines the visibility of a variable or a method from another class.
- **Return type:** A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is **void**.
- **Method name:** Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.
- **Parameter list:** Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
- **Method body:** This contains the set of instructions needed to complete the required activity.

### Example

Following code snippet shows a function *FindMax* that takes two integer values and returns the larger of the two. It has public access specifier, so it can be accessed from outside the class using an instance of the class.

```
class NumberManipulator
{
    public int FindMax(int num1, int num2)
    {
        /* local variable declaration */
        int result;

        if (num1 > num2)
            result = num1;
        else
            result = num2;

        return result;
    }
    ...
}
```

#### Calling Methods in C#

You can call a method using the name of the method. The following example illustrates this:

```
using System;
namespace CalculatorApplication
{
    class NumberManipulator
    {
        public int FindMax(int num1, int num2)
        {
            /* local variable declaration */
            int result;

            if (num1 > num2)
                result = num1;
            else
                result = num2;
            return result;
        }
        static void Main(string[] args)
        {
            /* local variable definition */
            int a = 100;
            int b = 200;
            int ret;
            NumberManipulator n = new NumberManipulator();
        }
    }
}
```



```
//calling the FindMax method
ret = n.FindMax(a, b);
Console.WriteLine("Max value is : {0}", ret );
Console.ReadLine();
    }
}
}
```

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

```
Max value is : 200
```

You can also call public method from other classes by using the instance of the class. For example, the method *FindMax* belongs to the *NumberManipulator* class, you can call it from another class *Test*.

```
using System;
namespace CalculatorApplication
{
    class NumberManipulator
    {
        public int FindMax(int num1, int num2)
        {
            /* local variable declaration */
            int result;

            if(num1 > num2)
                result = num1;
            else
                result = num2;

            return result;
        }
    }

    class Test
    {
        static void Main(string[] args)
        {
            /* local variable definition */
            int a = 100;
            int b = 200;
            int ret;
```

```

    NumberManipulator n = new NumberManipulator();

    //calling the FindMax method
    ret = n.FindMax(a, b);
    Console.WriteLine("Max value is : {0}", ret );
    Console.ReadLine();
}
}
}

```

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

```
Max value is : 200
```

#### Recursive Method Call

A method can call itself. This is known as **recursion**. Following is an example that calculates factorial for a given number using a recursive function:

```

using System;
namespace CalculatorApplication
{
    class NumberManipulator
    {
        public int factorial(int num)
        {
            /* local variable declaration */
            int result;
            if (num == 1)
            {
                return 1;
            }
            else
            {
                result = factorial(num - 1) * num;
                return result;
            }
        }

        static void Main(string[] args)
        {
            NumberManipulator n = new NumberManipulator();
            //calling the factorial method

```

```
Console.WriteLine("Factorial of 6 is : {0}", n.factorial(6));  
Console.WriteLine("Factorial of 7 is : {0}", n.factorial(7));  
Console.WriteLine("Factorial of 8 is : {0}", n.factorial(8));  
Console.ReadLine();  
}  
}  
}
```

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

```
Factorial of 6 is: 720  
  
Factorial of 7 is: 5040  
  
Factorial of 8 is: 40320
```

#### Passing Parameters to a Method

When method with parameters is called, you need to pass the parameters to the method. There are three ways that parameters can be passed to a method:

Mechanism	Description
<b>Value parameters</b>	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
<b>Reference parameters</b>	This method copies the reference to the memory location of an argument into the formal parameter. This means that changes made to the parameter affect the argument.
<b>Output parameters</b>	This method helps in returning more than one value.

## Casting and Type Conversions

Because C# is statically-typed at compile time, after a variable is declared, it cannot be declared again or used to store values of another type unless that type is convertible to the variable's type. For example, there is no conversion from an integer to any arbitrary string. Therefore, after you declare `i` as an integer, you cannot assign the string "Hello" to it, as is shown in the following code.

C#

```
int i;
```

```
i = "Hello"; // Error: "Cannot implicitly convert type 'string' to 'int'"
```

However, you might sometimes need to copy a value into a variable or method parameter of another type. For example, you might have an integer variable that you need to pass to a method whose parameter is typed as **double**. Or you might need to assign a class variable to a variable of an interface type. These kinds of operations are called *type conversions*. In C#, you can perform the following kinds of conversions:

- **Implicit conversions:** No special syntax is required because the conversion is type safe and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.
- **Explicit conversions (casts):** Explicit conversions require a cast operator. Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range, and conversion of a base-class instance to a derived class.
- **User-defined conversions:** User-defined conversions are performed by special methods that you can define to enable explicit and implicit conversions between custom types that do not have a base class–derived class relationship. For more information, see [Conversion Operators \(C# Programming Guide\)](#).
- **Conversions with helper classes:** To convert between non-compatible types, such as integers and [System.DateTime](#) objects, or hexadecimal strings and byte arrays, you can use the [System.BitConverter](#) class, the [System.Convert](#) class, and the **Parse** methods of the built-in numeric types, such as [Int32.Parse](#). For more information, see [How to: Convert a byte Array to an int \(C# Programming Guide\)](#), [How to: Convert a String to a Number \(C# Programming Guide\)](#), and [How to: Convert Between Hexadecimal Strings and Numeric Types \(C# Programming Guide\)](#).

## Implicit Conversions

For built-in numeric types, an implicit conversion can be made when the value to be stored can fit into the variable without being truncated or rounded off. For example, a variable of type [long](#) (8 byte integer) can store any value that an [int](#) (4 bytes on a 32-bit

computer) can store. In the following example, the compiler implicitly converts the value on the right to a type **long** before assigning it to **bigNum**.

C#

```
// Implicit conversion. num long can
```

```
// hold any value an int can hold, and more!
```

```
int num = 2147483647;
```

```
long bigNum = num;
```

For a complete list of all implicit numeric conversions, see [Implicit Numeric Conversions Table \(C# Reference\)](#).

For reference types, an implicit conversion always exists from a class to any one of its direct or indirect base classes or interfaces. No special syntax is necessary because a derived class always contains all the members of a base class.

```
Derived d = new Derived();
```

```
Base b = d; // Always OK.
```

### Explicit Conversions

However, if a conversion cannot be made without a risk of losing information, the compiler requires that you perform an explicit conversion, which is called a *cast*. A cast is a way of explicitly informing the compiler that you intend to make the conversion and that you are aware that data loss might occur. To perform a cast, specify the type that you are casting to in parentheses in front of the value or variable to be converted. The following program casts a [double](#) to an [int](#). The program will not compile without the cast.

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
```

// Output: 1234

For a list of the explicit numeric conversions that are allowed, see [Explicit Numeric Conversions Table \(C# Reference\)](#).

For reference types, an explicit cast is required if you need to convert from a base type to a derived type:

C#

// Create a new derived type.

Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.

Animal a = g;

// Explicit conversion is required to cast back

// to derived type. Note: This will compile but will

// throw an exception at run time if the right-side

// object is not in fact a Giraffe.

Giraffe g2 = (Giraffe) a;

A cast operation between reference types does not change the run-time type of the underlying object; it only changes the type of the value that is being used as a reference to that object. For more information, see [Polymorphism \(C# Programming Guide\)](#).

### Type Conversion Exceptions at Run Time

In some reference type conversions, the compiler cannot determine whether a cast will be valid. It is possible for a cast operation that compiles correctly to fail at run time. As shown in the following example, a type cast that fails at run time will cause an [InvalidCastException](#) to be thrown.

using System;

class Animal

```
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Reptile : Animal { }
class Mammal : Animal { }

class UnSafeCast
{
    static void Main()
    {
        Test(new Mammal());

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

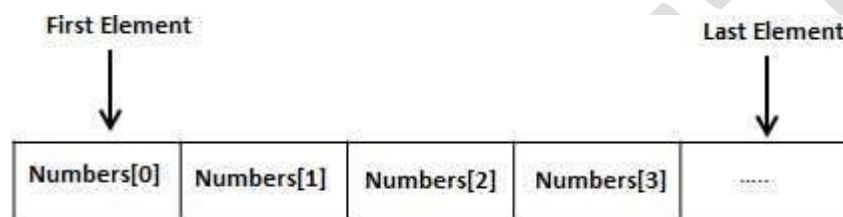
    static void Test(Animal a)
    {
        // Cause InvalidCastException at run time
        // because Mammal is not convertible to Reptile.
        Reptile r = (Reptile)a;
    }
}
```

## C# - Arrays

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type stored at contiguous memory locations.

Instead of declaring individual variables, such as `number0`, `number1`, ..., and `number99`, you declare one array variable such as `numbers` and use `numbers[0]`, `numbers[1]`, and ..., `numbers[99]` to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



### Declaring Arrays

To declare an array in C#, you can use the following syntax:

```
datatype[] arrayName;
```

where,

- *datatype* is used to specify the type of elements in the array.
- `[ ]` specifies the rank of the array. The rank specifies the size of the array.
- *arrayName* specifies the name of the array.

For example,

```
double[] balance;
```

### Initializing an Array

Declaring an array does not initialize the array in the memory. When the array variable is initialized, you can assign values to the array.



Array is a reference type, so you need to use the **new** keyword to create an instance of the array. For example,

```
double[] balance = new double[10];
```

### Assigning Values to an Array

You can assign values to individual array elements, by using the index number, like:

```
double[] balance = new double[10];  
balance[0] = 4500.0;
```

You can assign values to the array at the time of declaration, as shown:

```
double[] balance = { 2340.0, 4523.69, 3421.0};
```

You can also create and initialize an array, as shown:

```
int [] marks = new int[5] { 99, 98, 92, 97, 95};
```

You may also omit the size of the array, as shown:

```
int [] marks = new int[] { 99, 98, 92, 97, 95};
```

You can copy an array variable into another target array variable. In such case, both the target and source point to the same memory location:

```
int [] marks = new int[] { 99, 98, 92, 97, 95};  
int[] score = marks;
```

When you create an array, C# compiler implicitly initializes each array element to a default value depending on the array type. For example, for an int array all elements are initialized to 0.

## Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example,

```
double salary = balance[9];
```

The following example, demonstrates the above-mentioned concepts declaration, assignment, and accessing arrays:

```
using System;
namespace ArrayApplication
{
    class MyArray
    {
        static void Main(string[] args)
        {
            int [] n = new int[10]; /* n is an array of 10 integers */
            int i,j;

            /* initialize elements of array n */
            for ( i = 0; i < 10; i++ )
            {
                n[ i ] = i + 100;
            }

            /* output each array element's value */
            for ( j = 0; j < 10; j++ )
            {
                Console.WriteLine("Element[{0}] = {1}", j, n[j]);
            }
            Console.ReadKey();
        }
    }
}
```

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

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

### Using foreach Loop

In the previous example, we used a for loop for accessing each array element. You can also use a **foreach** statement to iterate through an array.

```
using System;
namespace ArrayApplication
{
    class MyArray
    {
        static void Main(string[] args)
        {
            int [] n = new int[10]; /* n is an array of 10 integers */

            /* initialize elements of array n */
            for ( int i = 0; i < 10; i++ )
            {
                n[i] = i + 100;
            }

            /* output each array element's value */
            foreach (int j in n )
            {
                int i = j-100;
                Console.WriteLine("Element[{0}] = {1}", i, j);
                i++;
            }
            Console.ReadKey();
        }
    }
}
```

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

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

### C# Arrays

There are following few important concepts related to array which should be clear to a C# programmer:

Concept	Description
<b><u>Multi-dimensional arrays</u></b>	C# supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
<b><u>Jagged arrays</u></b>	C# supports multidimensional arrays, which are arrays of arrays.
<b><u>Passing arrays to functions</u></b>	You can pass to the function a pointer to an array by specifying the array's name without an index.
<b><u>Param arrays</u></b>	This is used for passing unknown number of

	parameters to a function.
<b><u>The Array Class</u></b>	Defined in System namespace, it is the base class to all arrays, and provides various properties and methods for working with arrays.

## C# - Array Class

The Array class is the base class for all the arrays in C#. It is defined in the System namespace. The Array class provides various properties and methods to work with arrays.

### Properties of the Array Class

The following table describes some of the most commonly used properties of the Array class:

Sr.No	Property
1	<b>IsFixedSize</b> Gets a value indicating whether the Array has a fixed size.
2	<b>IsReadOnly</b> Gets a value indicating whether the Array is read-only.
3	<b>Length</b> Gets a 32-bit integer that represents the total number of elements in all the dimensions of the Array.
4	<b>LongLength</b> Gets a 64-bit integer that represents the total number of elements in all the dimensions of the Array.
5	<b>Rank</b> Gets the rank (number of dimensions) of the Array.

### Methods of the Array Class

The following table describes some of the most commonly used methods of the Array class:

Sr.No	Methods
-------	---------

1	<b>Clear</b> Sets a range of elements in the Array to zero, to false, or to null, depending on the element type.
2	<b>Copy(Array, Array, Int32)</b> Copies a range of elements from an Array starting at the first element and pastes them into another Array starting at the first element. The length is specified as a 32-bit integer.
3	<b>CopyTo(Array, Int32)</b> Copies all the elements of the current one-dimensional Array to the specified one-dimensional Array starting at the specified destination Array index. The index is specified as a 32-bit integer.
4	<b>GetLength</b> Gets a 32-bit integer that represents the number of elements in the specified dimension of the Array.
5	<b>GetLongLength</b> Gets a 64-bit integer that represents the number of elements in the specified dimension of the Array.
6	<b>GetLowerBound</b> Gets the lower bound of the specified dimension in the Array.
7	<b>GetType</b> Gets the Type of the current instance. (Inherited from Object.)
8	<b>GetUpperBound</b> Gets the upper bound of the specified dimension in the Array.
9	<b>GetValue(Int32)</b> Gets the value at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer.
10	<b>IndexOf(Array, Object)</b> Searches for the specified object and returns the index of the first

	occurrence within the entire one-dimensional Array.
11	<b>Reverse(Array)</b> Reverses the sequence of the elements in the entire one-dimensional Array.
12	<b>SetValue(Object, Int32)</b> Sets a value to the element at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer.
13	<b>Sort(Array)</b> Sorts the elements in an entire one-dimensional Array using the IComparable implementation of each element of the Array.
14	<b>ToStringk</b> Returns a string that represents the current object. (Inherited from Object.)

For complete list of Array class properties and methods, please consult Microsoft documentation on C#.

#### Example

The following program demonstrates use of some of the methods of the Array class:

```
using System;
namespace ArrayApplication
{
    class MyArray
    {
        static void Main(string[] args)
        {
            int[] list = { 34, 72, 13, 44, 25, 30, 10 };
            int[] temp = list;
            Console.WriteLine("Original Array: ");

            foreach (int i in list)
            {
                Console.Write(i + " ");
            }
            Console.WriteLine();
        }
    }
}
```



```
// reverse the array
Array.Reverse(temp);
Console.Write("Reversed Array: ");

foreach (int i in temp)
{
    Console.Write(i + " ");
}
Console.WriteLine();

//sort the array
Array.Sort(list);
Console.Write("Sorted Array: ");

foreach (int i in list)
{
    Console.Write(i + " ");
}
Console.WriteLine();
Console.ReadKey();
}
```

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

```
Original Array: 34 72 13 44 25 30 10
Reversed Array: 10 30 25 44 13 72 34
Sorted Array: 10 13 25 30 34 44 72
```

## C# - ArrayList Class

It represents an ordered collection of an object that can be indexed individually. It is basically an alternative to an array. However, unlike array you can add and remove items from a list at a specified position using an **index** and the array resizes itself automatically. It also allows dynamic memory allocation, adding, searching and sorting items in the list.

### Methods and Properties of ArrayList Class

The following table lists some of the commonly used **properties** of the **ArrayList** class:

Property	Description
Capacity	Gets or sets the number of elements that the ArrayList can contain.
Count	Gets the number of elements actually contained in the ArrayList.
IsFixedSize	Gets a value indicating whether the ArrayList has a fixed size.
IsReadOnly	Gets a value indicating whether the ArrayList is read-only.
Item	Gets or sets the element at the specified index.

The following table lists some of the commonly used **methods** of the **ArrayList** class:

Sr.No.	Methods
1	<b>public virtual int Add(object value);</b> Adds an object to the end of the ArrayList.
2	<b>public virtual void AddRange(ICollection c);</b> Adds the elements of an ICollection to the end of the ArrayList.
3	<b>public virtual void Clear();</b>

	Removes all elements from the ArrayList.
4	<b>public virtual bool Contains(object item);</b> Determines whether an element is in the ArrayList.
5	<b>public virtual ArrayList GetRange(int index, int count);</b> Returns an ArrayList which represents a subset of the elements in the source ArrayList.
6	<b>public virtual int IndexOf(object);</b> Returns the zero-based index of the first occurrence of a value in the ArrayList or in a portion of it.
7	<b>public virtual void Insert(int index, object value);</b> Inserts an element into the ArrayList at the specified index.
8	<b>public virtual void InsertRange(int index, ICollection c);</b> Inserts the elements of a collection into the ArrayList at the specified index.
9	<b>public virtual void Remove(object obj);</b> Removes the first occurrence of a specific object from the ArrayList.
10	<b>public virtual void RemoveAt(int index);</b> Removes the element at the specified index of the ArrayList.
11	<b>public virtual void RemoveRange(int index, int count);</b> Removes a range of elements from the ArrayList.
12	<b>public virtual void Reverse();</b> Reverses the order of the elements in the ArrayList.
13	<b>public virtual void SetRange(int index, ICollection c);</b> Copies the elements of a collection over a range of elements in the ArrayList.

14	<b>public virtual void Sort();</b> Sorts the elements in the ArrayList.
15	<b>public virtual void TrimToSize();</b> Sets the capacity to the actual number of elements in the ArrayList.

### Example

The following example demonstrates the concept:

```
using System;
using System.Collections;

namespace CollectionApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList al = new ArrayList();

            Console.WriteLine("Adding some numbers:");
            al.Add(45);
            al.Add(78);
            al.Add(33);
            al.Add(56);
            al.Add(12);
            al.Add(23);
            al.Add(9);

            Console.WriteLine("Capacity: {0} ", al.Capacity);
            Console.WriteLine("Count: {0}", al.Count);

            Console.Write("Content: ");
            foreach (int i in al)
            {
                Console.Write(i + " ");
            }

            Console.WriteLine();
            Console.Write("Sorted Content: ");
            al.Sort();
            foreach (int i in al)
```

```
{  
    Console.Write(i + " ");  
}  
Console.WriteLine();  
Console.ReadKey();  
}  
}  
}
```

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

```
Adding some numbers:  
Capacity: 8  
Count: 7  
Content: 45 78 33 56 12 23 9  
Content: 9 12 23 33 45 56 78
```

## C# - Strings

In C#, you can use strings as array of characters, However, more common practice is to use the **string** keyword to declare a string variable. The string keyword is an alias for the **System.String** class.

### Creating a String Object

You can create string object using one of the following methods:

- By assigning a string literal to a String variable
- By using a String class constructor
- By using the string concatenation operator (+)
- By retrieving a property or calling a method that returns a string
- By calling a formatting method to convert a value or an object to its string representation

The following example demonstrates this:

```
using System;
namespace StringApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            //from string literal and string concatenation
            string fname, lname;
            fname = "Rowan";
            lname = "Atkinson";

            string fullname = fname + lname;
            Console.WriteLine("Full Name: {0}", fullname);

            //by using string constructor
            char[] letters = { 'H', 'e', 'l', 'l', 'o' };
            string greetings = new string(letters);
            Console.WriteLine("Greetings: {0}", greetings);

            //methods returning string
            string[] sarray = { "Hello", "From", "Tutorials", "Point" };
            string message = String.Join(" ", sarray);
            Console.WriteLine("Message: {0}", message);

            //formatting method to convert a value
            DateTime waiting = new DateTime(2012, 10, 10, 17, 58, 1);
```

```

string chat = String.Format("Message sent at {0:t} on {0:D}", waiting);
Console.WriteLine("Message: {0}", chat);
}
}
}

```

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

```

Full Name: Rowan Atkinson
Greetings: Hello
Message: Hello From Tutorials Point
Message: Message sent at 5:58 PM on Wednesday, October 10, 2012

```

### Properties of the String Class

The String class has the following two properties:

Sr.No	Property
1	<b>Chars</b> Gets the <i>Char</i> object at a specified position in the current <i>String</i> object.
2	<b>Length</b> Gets the number of characters in the current String object.

### Methods of the String Class

The String class has numerous methods that help you in working with the string objects. The following table provides some of the most commonly used methods:

Sr.No	Methods
1	<b>public static int Compare(string strA, string strB)</b> Compares two specified string objects and returns an integer that indicates their relative position in the sort order.

2	<b>public static int Compare(string strA, string strB, bool ignoreCase )</b> Compares two specified string objects and returns an integer that indicates their relative position in the sort order. However, it ignores case if the Boolean parameter is true.
3	<b>public static string Concat(string str0, string str1)</b> Concatenates two string objects.
4	<b>public static string Concat(string str0, string str1, string str2)</b> Concatenates three string objects.
5	<b>public static string Concat(string str0, string str1, string str2, string str3)</b> Concatenates four string objects.
6	<b>public bool Contains(string value)</b> Returns a value indicating whether the specified String object occurs within this string.
7	<b>public static string Copy(string str)</b> Creates a new String object with the same value as the specified string.
8	<b>public void CopyTo(int sourceIndex, char[] destination, int destinationIndex, int count)</b> Copies a specified number of characters from a specified position of the String object to a specified position in an array of Unicode characters.
9	<b>public bool EndsWith(string value)</b> Determines whether the end of the string object matches the specified string.
10	<b>public bool Equals(string value)</b> Determines whether the current String object and the specified String object have the same value.
11	<b>public static bool Equals(string a, string b)</b> Determines whether two specified String objects have the same value.



12	<b>public static string Format(string format, Object arg0)</b> Replaces one or more format items in a specified string with the string representation of a specified object.
13	<b>public int IndexOf(char value)</b> Returns the zero-based index of the first occurrence of the specified Unicode character in the current string.
14	<b>public int IndexOf(string value)</b> Returns the zero-based index of the first occurrence of the specified string in this instance.
15	<b>public int IndexOf(char value, int startIndex)</b> Returns the zero-based index of the first occurrence of the specified Unicode character in this string, starting search at the specified character position.
16	<b>public int IndexOf(string value, int startIndex)</b> Returns the zero-based index of the first occurrence of the specified string in this instance, starting search at the specified character position.
17	<b>public int IndexOfAny(char[] anyOf)</b> Returns the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters.
18	<b>public int IndexOfAny(char[] anyOf, int startIndex)</b> Returns the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters, starting search at the specified character position.
19	<b>public string Insert(int startIndex, string value)</b> Returns a new string in which a specified string is inserted at a specified index position in the current string object.
20	<b>public static bool IsNullOrEmpty(string value)</b> Indicates whether the specified string is null or an Empty string.

21	<b>public static string Join(string separator, params string[] value)</b> Concatenates all the elements of a string array, using the specified separator between each element.
22	<b>public static string Join(string separator, string[] value, int startIndex, int count)</b> Concatenates the specified elements of a string array, using the specified separator between each element.
23	<b>public int LastIndexOf(char value)</b> Returns the zero-based index position of the last occurrence of the specified Unicode character within the current string object.
24	<b>public int LastIndexOf(string value)</b> Returns the zero-based index position of the last occurrence of a specified string within the current string object.
25	<b>public string Remove(int startIndex)</b> Removes all the characters in the current instance, beginning at a specified position and continuing through the last position, and returns the string.
26	<b>public string Remove(int startIndex, int count)</b> Removes the specified number of characters in the current string beginning at a specified position and returns the string.
27	<b>public string Replace(char oldChar, char newChar)</b> Replaces all occurrences of a specified Unicode character in the current string object with the specified Unicode character and returns the new string.
28	<b>public string Replace(string oldValue, string newValue)</b> Replaces all occurrences of a specified string in the current string object with the specified string and returns the new string.
29	<b>public string[] Split(params char[] separator)</b> Returns a string array that contains the substrings in the current string object, delimited by elements of a specified Unicode character array.

30	<b>public string[] Split(char[] separator, int count)</b> Returns a string array that contains the substrings in the current string object, delimited by elements of a specified Unicode character array. The int parameter specifies the maximum number of substrings to return.
31	<b>public bool StartsWith(string value)</b> Determines whether the beginning of this string instance matches the specified string.
32	<b>public char[] ToCharArray()</b> Returns a Unicode character array with all the characters in the current string object.
33	<b>public char[] ToCharArray(int startIndex, int length)</b> Returns a Unicode character array with all the characters in the current string object, starting from the specified index and up to the specified length.
34	<b>public string ToLower()</b> Returns a copy of this string converted to lowercase.
35	<b>public string ToUpper()</b> Returns a copy of this string converted to uppercase.
36	<b>public string Trim()</b> Removes all leading and trailing white-space characters from the current String object.

You can visit MSDN library for the complete list of methods and String class constructors.

#### Examples

The following example demonstrates some of the methods mentioned above:

#### Comparing Strings:

```
using System;
```

```
namespace StringApplication
{
    class StringProg
    {
        static void Main(string[] args)
        {
            string str1 = "This is test";
            string str2 = "This is text";

            if (String.Compare(str1, str2) == 0)
            {
                Console.WriteLine(str1 + " and " + str2 + " are equal.");
            }
            else
            {
                Console.WriteLine(str1 + " and " + str2 + " are not equal.");
            }
            Console.ReadKey();
        }
    }
}
```

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

This is test and This is text are not equal.

### String Contains String:

```
using System;
namespace StringApplication
{
    class StringProg
    {
        static void Main(string[] args)
        {
            string str = "This is test";
            if (str.Contains("test"))
            {
                Console.WriteLine("The sequence 'test' was found.");
            }
            Console.ReadKey();
        }
    }
}
```

The sequence 'test' was found.

```
using System;
namespace StringApplication
{
    class StringProg
    {
        static void Main(string[] args)
        {
            string str = "Last night I dreamt of San Pedro";
            Console.WriteLine(str);
            string substr = str.Substring(23);
            Console.WriteLine(substr);
        }
    }
}
```

## San Pedro

```
using System;
namespace StringApplication
{
    class StringProg
    {
        static void Main(string[] args)
        {
            string[] starray = new string[]{"Down the way nights are dark",
            "And the sun shines daily on the mountain top",
            "I took a trip on a sailing ship",
            "And when I reached Jamaica",
```

```
"I made a stop";  
  
string str = String.Join("\n", starray);  
Console.WriteLine(str);  
    }  
}  
}
```

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

```
Down the way nights are dark  
And the sun shines daily on the mountain top  
I took a trip on a sailing ship  
And when I reached Jamaica  
I made a stop
```

## Using the StringBuilder Class in the .NET Framework

The [String](#) object is immutable. Every time you use one of the methods in the [System.String](#) class, you create a new string object in memory, which requires a new allocation of space for that new object. In situations where you need to perform repeated modifications to a string, the overhead associated with creating a new [String](#) object can be costly. The [System.Text.StringBuilder](#) class can be used when you want to modify a string without creating a new object. For example, using the [StringBuilder](#) class can boost performance when concatenating many strings together in a loop.

### Instantiating a StringBuilder Object

You can create a new instance of the [StringBuilder](#) class by initializing your variable with one of the overloaded constructor methods, as illustrated in the following example.

```
StringBuilder MyStringBuilder = new StringBuilder("Hello World!");
```

### Setting the Capacity and Length

Although the [StringBuilder](#) is a dynamic object that allows you to expand the number of characters in the string that it encapsulates, you can specify a value for the maximum number of characters that it can hold. This value is called the capacity of the object and should not be confused with the length of the string that the current [StringBuilder](#) holds. For example, you might create a new instance of the [StringBuilder](#) class with the string "Hello", which has a length of 5, and you might specify that the object has a maximum capacity of 25. When you modify the [StringBuilder](#), it does not reallocate size for itself until the capacity is reached. When this occurs, the new space is allocated automatically and the capacity is doubled. You can specify the capacity of the [StringBuilder](#) class using one of the overloaded constructors. The following example specifies that the [MyStringBuilder](#) object can be expanded to a maximum of 25 spaces.

```
StringBuilder MyStringBuilder = new StringBuilder("Hello World!", 25);
```

Additionally, you can use the read/write [Capacity](#) property to set the maximum length of your object. The following example uses the **Capacity** property to define the maximum object length.

```
MyStringBuilder.Capacity = 25;
```

The [EnsureCapacity](#) method can be used to check the capacity of the current **StringBuilder**. If the capacity is greater than the passed value, no change is made; however, if the capacity is smaller than the passed value, the current capacity is changed to match the passed value.

The [Length](#) property can also be viewed or set. If you set the **Length** property to a value that is greater than the **Capacity** property, the **Capacity** property is automatically changed to the same value as the **Length** property. Setting the **Length** property to a value that is less than the length of the string within the current **StringBuilder** shortens the string.

### Modifying the StringBuilder String

The following table lists the methods you can use to modify the contents of a **StringBuilder**.

Method name	Use
<a href="#">StringBuilder.Append</a>	Appends information to the end of the current <b>StringBuilder</b> .
<a href="#">StringBuilder.AppendFormat</a>	Replaces a format specifier passed in a string with formatted text.
<a href="#">StringBuilder.Insert</a>	Inserts a string or object into the specified index of the current <b>StringBuilder</b> .
<a href="#">StringBuilder.Remove</a>	Removes a specified number of characters from the current <b>StringBuilder</b> .
<a href="#">StringBuilder.Replace</a>	Replaces a specified character at a specified index.

### Append

The **Append** method can be used to add text or a string representation of an object to the end of a string represented by the current **StringBuilder**. The following example initializes a **StringBuilder** to "Hello World" and then appends some text to the end of the object. Space is allocated automatically as needed.

```
StringBuilder MyStringBuilder = new StringBuilder("Hello World!");
```

```
MyStringBuilder.Append(" What a beautiful day.");
```



```
Console.WriteLine(MyStringBuilder);
```

```
// The example displays the following output:
```

```
//    Hello World! What a beautiful day.
```

## AppendFormat

The [StringBuilder.AppendFormat](#) method adds text to the end of the [StringBuilder](#) object. It supports the composite formatting feature (for more information, see [Composite Formatting](#)) by calling the [IFormattable](#) implementation of the object or objects to be formatted. Therefore, it accepts the standard format strings for numeric, date and time, and enumeration values, the custom format strings for numeric and date and time values, and the format strings defined for custom types. (For information about formatting, see [Formatting Types in the .NET Framework](#).) You can use this method to customize the format of variables and append those values to a [StringBuilder](#). The following example uses the [AppendFormat](#) method to place an integer value formatted as a currency value at the end of a [StringBuilder](#) object.

```
int MyInt = 25;
```

```
StringBuilder MyStringBuilder = new StringBuilder("Your total is ");
```

```
MyStringBuilder.AppendFormat("{0:C} ", MyInt);
```

```
Console.WriteLine(MyStringBuilder);
```

```
// The example displays the following output:
```

```
//    Your total is $25.00
```

## Insert

The [Insert](#) method adds a string or object to a specified position in the current [StringBuilder](#) object. The following example uses this method to insert a word into the sixth position of a [StringBuilder](#) object.

```
StringBuilder MyStringBuilder = new StringBuilder("Hello World!");
```

```
MyStringBuilder.Insert(6, "Beautiful ");
```

```
Console.WriteLine(MyStringBuilder);
```

```
// The example displays the following output:
```

```
//    Hello Beautiful World!
```

## Remove

You can use the **Remove** method to remove a specified number of characters from the current [StringBuilder](#) object, beginning at a specified zero-based index. The following example uses the **Remove** method to shorten a [StringBuilder](#) object.

```
StringBuilder MyStringBuilder = new StringBuilder("Hello World!");
```

```
MyStringBuilder.Remove(5,7);
```

```
Console.WriteLine(MyStringBuilder);
```

```
// The example displays the following output:
```

```
// Hello
```

## Replace

The **Replace** method can be used to replace characters within the [StringBuilder](#) object with another specified character. The following example uses the **Replace** method to search a [StringBuilder](#) object for all instances of the exclamation point character (!) and replace them with the question mark character (?).

```
StringBuilder MyStringBuilder = new StringBuilder("Hello World!");
```

```
MyStringBuilder.Replace('!', '?');
```

```
Console.WriteLine(MyStringBuilder);
```

```
// The example displays the following output:
```

```
// Hello World?
```

## Converting a StringBuilder Object to a String

You must convert the [StringBuilder](#) object to a [String](#) object before you can pass the string represented by the [StringBuilder](#) object to a method that has a [String](#) parameter or display it in the user interface. You do this conversion by calling the [StringBuilder.ToString](#) method. The following example calls a number of [StringBuilder](#) methods and then calls the [StringBuilder.ToString\(\)](#) method to display the string.

```
using System;  
using System.Text;
```

```
public class Example
```

```
{
    public static void Main()
    {
        StringBuilder sb = new StringBuilder();
        bool flag = true;
        string[] spellings = { "recieve", "receeve", "receive" };
        sb.AppendFormat("Which of the following spellings is {0}:", flag);
        sb.AppendLine();
        for (int ctr = 0; ctr <= spellings.GetUpperBound(0); ctr++) {
            sb.AppendFormat(" {0}. {1}", ctr, spellings[ctr]);
            sb.AppendLine();
        }
        sb.AppendLine();
        Console.WriteLine(sb.ToString());
    }
}

// The example displays the following output:
//      Which of the following spellings is True:
//      0. recieve
//      1. receeve
//      2. receive
```

## C# - Structures

In C#, a structure is a value type data type. It helps you to make a single variable hold related data of various data types. The **struct** keyword is used for creating a structure.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

### Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member for your program.

For example, here is the way you can declare the Book structure:

```
struct Books
{
    public string title;
    public string author;
    public string subject;
    public int book_id;
};
```

The following program shows the use of the structure:

```
using System;
struct Books
{
    public string title;
    public string author;
    public string subject;
    public int book_id;
};

public class testStructure
{
    public static void Main(string[] args)
    {
        Books Book1; /* Declare Book1 of type Book */
    }
}
```

```
Books Book2; /* Declare Book2 of type Book */

/* book 1 specification */
Book1.title = "C Programming";
Book1.author = "Nuha Ali";
Book1.subject = "C Programming Tutorial";
Book1.book_id = 6495407;

/* book 2 specification */
Book2.title = "Telecom Billing";
Book2.author = "Zara Ali";
Book2.subject = "Telecom Billing Tutorial";
Book2.book_id = 6495700;

/* print Book1 info */
Console.WriteLine("Book 1 title : {0}", Book1.title);
Console.WriteLine("Book 1 author : {0}", Book1.author);
Console.WriteLine("Book 1 subject : {0}", Book1.subject);
Console.WriteLine("Book 1 book_id :{0}", Book1.book_id);

/* print Book2 info */
Console.WriteLine("Book 2 title : {0}", Book2.title);
Console.WriteLine("Book 2 author : {0}", Book2.author);
Console.WriteLine("Book 2 subject : {0}", Book2.subject);
Console.WriteLine("Book 2 book_id : {0}", Book2.book_id);

Console.ReadKey();

}
```

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

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

## Features of C# Structures

You have already used a simple structure named Books. Structures in C# are quite different from that in traditional C or C++. The C# structures have the following features:

- Structures can have methods, fields, indexers, properties, operator methods, and events.
- Structures can have defined constructors, but not destructors. However, you cannot define a default constructor for a structure. The default constructor is automatically defined and cannot be changed.
- Unlike classes, structures cannot inherit other structures or classes.
- Structures cannot be used as a base for other structures or classes.
- A structure can implement one or more interfaces.
- Structure members cannot be specified as abstract, virtual, or protected.
- When you create a struct object using the **New** operator, it gets created and the appropriate constructor is called. Unlike classes, structs can be instantiated without using the New operator.
- If the New operator is not used, the fields remain unassigned and the object cannot be used until all the fields are initialized.

## Class versus Structure

Classes and Structures have the following basic differences:

- classes are reference types and structs are value types
- structures do not support inheritance
- structures cannot have default constructor

In the light of the above discussions, let us rewrite the previous example:

```
using System;
struct Books
{
    private string title;
    private string author;
    private string subject;
    private int book_id;
    public void getValues(string t, string a, string s, int id)
    {
        title = t;
        author = a;
        subject = s;
        book_id = id;
    }
    public void display()
```

```
{
    Console.WriteLine("Title : {0}", title);
    Console.WriteLine("Author : {0}", author);
    Console.WriteLine("Subject : {0}", subject);
    Console.WriteLine("Book_id :{0}", book_id);
}
};

public class testStructure
{
    public static void Main(string[] args)
    {

        Books Book1 = new Books(); /* Declare Book1 of type Book */
        Books Book2 = new Books(); /* Declare Book2 of type Book */

        /* book 1 specification */
        Book1.getValues("C Programming",
            "Nuha Ali", "C Programming Tutorial",6495407);
        /* book 2 specification */
        Book2.getValues("Telecom Billing",
            "Zara Ali", "Telecom Billing Tutorial", 6495700);
        /* print Book1 info */
        Book1.display();
        /* print Book2 info */
        Book2.display();

        Console.ReadKey();
    }
}
```

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

```
Title : C Programming
Author : Nuha Ali
Subject : C Programming Tutorial
Book_id : 6495407
Title : Telecom Billing
Author : Zara Ali
Subject : Telecom Billing Tutorial
Book_id : 6495700
```

## C# - Enums

An enumeration is a set of named integer constants. An enumerated type is declared using the **enum** keyword.

C# enumerations are value data type. In other words, enumeration contains its own values and cannot inherit or cannot pass inheritance.

### Declaring enum Variable

The general syntax for declaring an enumeration is:

```
enum <enum_name>
{
    enumeration list
};
```

Where,

- The *enum\_name* specifies the enumeration type name.
- The *enumeration list* is a comma-separated list of identifiers.

Each of the symbols in the enumeration list stands for an integer value, one greater than the symbol that precedes it. By default, the value of the first enumeration symbol is 0. For example:

```
enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };
```

Example

The following example demonstrates use of enum variable:

```
using System;
namespace EnumApplication
{
    class EnumProgram
    {
        enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

        static void Main(string[] args)
        {

```



```
int WeekdayStart = (int)Days.Mon;  
int WeekdayEnd = (int)Days.Fri;  
Console.WriteLine("Monday: {0}", WeekdayStart);  
Console.WriteLine("Friday: {0}", WeekdayEnd);  
Console.ReadKey();  
}  
}  
}
```

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

Monday: 1

Friday: 5

## Boxing and Unboxing

Boxing is the process of converting a value type to the type **object** or to any interface type implemented by this value type. When the CLR boxes a value type, it wraps the value inside a System.Object and stores it on the managed heap. Unboxing extracts the value type from the object. Boxing is implicit; unboxing is explicit. The concept of boxing and unboxing underlies the C# unified view of the type system in which a value of any type can be treated as an object.

In the following example, the integer variable *i* is *boxed* and assigned to object *o*.

```
int i = 123;
```

```
// The following line boxes i.
```

```
object o = i;
```

The object *o* can then be unboxed and assigned to integer variable *i*:

```
o = 123;
```

```
i = (int)o; // unboxing
```

The following examples illustrate how boxing is used in C#.

```
// String.Concat example.
```

```
// String.Concat has many versions. Rest the mouse pointer on  
// Concat in the following statement to verify that the version  
// that is used here takes three object arguments. Both 42 and  
// true must be boxed.
```

```
Console.WriteLine(String.Concat("Answer", 42, true));
```

```
// List example.
```

```
// Create a list of objects to hold a heterogeneous collection  
// of elements.
```

```
List<object> mixedList = new List<object>();
```

```
// Add a string element to the list.
```

```
mixedList.Add("First Group:");
```

```
// Add some integers to the list.
```

```
for (int j = 1; j < 5; j++)
```

```
{
```

```
    // Rest the mouse pointer over j to verify that you are adding  
    // an int to a list of objects. Each element j is boxed when  
    // you add j to mixedList.
```

```
    mixedList.Add(j);
```

```
}
```

```
// Add another string and more integers.
mixedList.Add("Second Group:");
for (int j = 5; j < 10; j++)
{
    mixedList.Add(j);
}
// Display the elements in the list. Declare the loop variable by
// using var, so that the compiler assigns its type.
foreach (var item in mixedList)
{
    // Rest the mouse pointer over item to verify that the elements
    // of mixedList are objects.
    Console.WriteLine(item);
}

// The following loop sums the squares of the first group of boxed
// integers in mixedList. The list elements are objects, and cannot
// be multiplied or added to the sum until they are unboxed. The
// unboxing must be done explicitly.
var sum = 0;
for (var j = 1; j < 5; j++)
{
    // The following statement causes a compiler error: Operator
    // '*' cannot be applied to operands of type 'object' and
    // 'object'.
    //sum += mixedList[j] * mixedList[j];

    // After the list elements are unboxed, the computation does
    // not cause a compiler error.
    sum += (int)mixedList[j] * (int)mixedList[j];
}

// The sum displayed is 30, the sum of 1 + 4 + 9 + 16.
Console.WriteLine("Sum: " + sum);

// Output:
// Answer42True
// First Group:
// 1
// 2
// 3
// 4
// Second Group:
// 5
// 6
// 7
```

```
// 8
// 9
// Sum: 30
```

## Performance

In relation to simple assignments, boxing and unboxing are computationally expensive processes. When a value type is boxed, a new object must be allocated and constructed. To a lesser degree, the cast required for unboxing is also expensive computationally. For more information, see [Performance](#).

## Boxing

Boxing is used to store value types in the garbage-collected heap. Boxing is an implicit conversion of a [value type](#) to the type **object** or to any interface type implemented by this value type. Boxing a value type allocates an object instance on the heap and copies the value into the new object.

Consider the following declaration of a value-type variable:

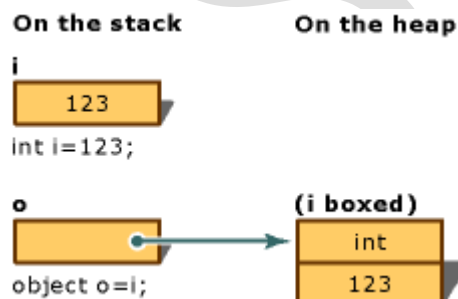
```
int i = 123;
```

The following statement implicitly applies the boxing operation on the variable **i**:

```
// Boxing copies the value of i into object o.
```

```
object o = i;
```

The result of this statement is creating an object reference **o**, on the stack, that references a value of the type **int**, on the heap. This value is a copy of the value-type value assigned to the variable **i**. The difference between the two variables, **i** and **o**, is illustrated in the following figure.



## Boxing Conversion

It is also possible to perform the boxing explicitly as in the following example, but explicit boxing is never required:

```
int i = 123;
```

```
object o = (object)i; // explicit boxing
```

## Description

This example converts an integer variable **i** to an object **o** by using boxing. Then, the value stored in the variable **i** is changed from **123** to **456**. The example shows that the original value type and the boxed object use separate memory locations, and therefore can store different values.

```
class TestBoxing
{
    static void Main()
    {
        int i = 123;

        // Boxing copies the value of i into object o.
        object o = i;

        // Change the value of i.
        i = 456;

        // The change in i does not effect the value stored in o.
        System.Console.WriteLine("The value-type value = {0}", i);
        System.Console.WriteLine("The object-type value = {0}", o);
    }
}
/* Output:
    The value-type value = 456
    The object-type value = 123
*/
```

## Unboxing

Unboxing is an explicit conversion from the type **object** to a **value type** or from an interface type to a value type that implements the interface. An unboxing operation consists of:

- Checking the object instance to make sure that it is a boxed value of the given value type.
- Copying the value from the instance into the value-type variable.

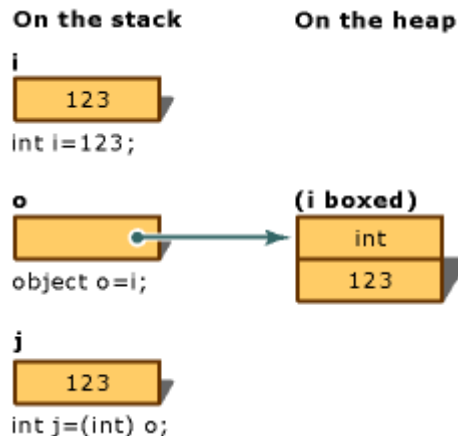
The following statements demonstrate both boxing and unboxing operations:

```
int i = 123;    // a value type
```

```
object o = i;    // boxing
```

```
int j = (int)o;  // unboxing
```

The following figure demonstrates the result of the previous statements.



## Unboxing Conversion

For the unboxing of value types to succeed at run time, the item being unboxed must be a reference to an object that was previously created by boxing an instance of that value type. Attempting to unbox **null** causes a [NullReferenceException](#). Attempting to unbox a reference to an incompatible value type causes an [InvalidCastException](#).

The following example demonstrates a case of invalid unboxing and the resulting **InvalidCastException**. Using **try** and **catch**, an error message is displayed when the error occurs.

```
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i; // implicit boxing

        try
        {
            int j = (short)o; // attempt to unbox

            System.Console.WriteLine("Unboxing OK.");
        }
        catch (System.InvalidCastException e)
        {
            System.Console.WriteLine("{0} Error: Incorrect unboxing.", e.Message);
        }
    }
}
```

```
}  
}
```

This program outputs:

Specified cast is not valid. Error: Incorrect unboxing.

If you change the statement:

```
int j = (short) o;
```

to:

```
int j = (int) o;
```

the conversion will be performed, and you will get the output:

Unboxing OK.

Ref: <http://www.tutorialspoint.com/>

<https://msdn.microsoft.com>

<http://www.aspfree.com/c/a/c-sharp/branching-and-looping-in-c-part-1>