

C# Basics



Basics of C#

- C# is an object oriented programming language created by Microsoft.
- Build upon some of the best features of some major programming language.
- C# is one of the most popular programming language.
- C# is a case sensitive language.



Basic program in C#

```
using System;
```

```
class Program
```

```
{
```

```
    static void Main()
```

```
    {
```

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

```
        Console.ReadLine();
```

```
    }
```

```
}
```



Basic program in C#

- The Main method of a .Net program is the entry point of the program.
- The 'System' is a namespace provided by the .Net framework library.
- The static class 'Console' is contained inside the System namespace.
- The methods inside Console class can also be called by directly calling the System namespace.

```
System.Console.WriteLine("Hello");
```



Variables

- In C#, you declare variables using the following format:
datatype identifier;

```
class Program
{
    static void Main(string[] args)
    {
        //---declare the variables---
        int num1;
        int num2 = 5;
        float num3, num4;

        //---assign values to the variables---
        num1 = 4;
        num3 = num4 = 6.2f;

        //---print out the values of the variables---
        Console.WriteLine("{0} {1} {2} {3}", num1, num2, num3, num4);
        Console.ReadLine();
        return;
    }
}
```

Variables-contd.

- Note the following:
 - num1 is declared as an int (integer).
 - num2 is declared as an int and assigned a value at the same time.
 - num3 and num4 are declared as float (floating point number)
- You need to declare a variable before you can use it. If not, C# compiler will flag that as an error.
- You can assign multiple variables in the same statement, as is shown in the assignment of num3 and num4 .



Constants

Constants

- The **constants** refer to fixed values that the program may not alter during its execution. These fixed values are also called literals.
- To declare a constant in C#, you use the `const` keyword, like this:
- `//---declared the PI constant---`
- `const float PI=3.14f;`



- Find the output of the following:

```
static void Main(string[] args)
{
    const int m = 100;
    int n = 10;
    const int k = n * 5 * 100 / n;
    Console.WriteLine(m * k);
    Console.ReadLine();
}
```



Variable scope

- Two identically named variables in different scope would be legal

```
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    { //---i is visible within this loop only---
        Console.WriteLine(i);
    } //---i goes out of scope here---

    for (int i = 0; i < 3; i++)
    { //---i is visible within this loop only---
        Console.WriteLine(i);
    } //---i goes out of scope here---

    Console.ReadLine();
    return;
}
```

Variable scope

- Declaring another variable named `i` outside the loop or inside it will cause a compilation error as the following example shows:

```
static void Main(string[] args)
{
    int i = 4; //---error---

    for (int i = 0; i < 5; i++)
    {
        int i = 6; //---error---
        Console.WriteLine(i);
    }

    for (int i = 0; i < 3; i++)
    {
        Console.WriteLine(i);
    }

    Console.ReadLine();
    return;
}
```

Data Types

- C# is a **strongly typed language** and as such all variables and objects must have a declared data type. The data type can be one of the following:

- Value
- Reference
- User - defined
- Anonymous

- Value Types:

A value type variable contains the data that it is assigned.

And when you assign a value type variable to another, you make a copy of it.

Value Types

```
int num1 = 5; //---or---
```

```
System.Int32 num2 = 5;
```

- To get the type of a variable, use the GetType() method:

```
Console.WriteLine(num1.GetType()); //---System.Int32---
```

- In C#, all noninteger numbers are always treated as a double. And so if you want to assign a noninteger number like 3.99 to a float variable, you need to append it with the F (or f) suffix, like this:
- float price = 3.99F;
- You can also assign integer values using hexadecimal representation. Simply prefix the hexadecimal number with 0x , like this:

```
int num1 = 0xA;
```

```
Console.WriteLine(num1); //---10---
```

Value Types

- **Nullable Type**

- All value types in C# have a default value when they are declared. For example, the following declaration declares a Boolean and an int variable:

```
Boolean married; //---default value is false---  
int age; //--- default value is 0---
```

- However, C# prevents you from using a variable if you do not explicitly initialize it. The following statements, for instance, cause the compiler to complain:

```
Boolean married;  
//---error: Use of unassigned local variable 'married'---  
Console.WriteLine(married);
```

- To use the variable, you first need to initialize it with a value:

```
Boolean married = false;  
Console.WriteLine(married); //---now OK---
```

Value Types

- Now married has a default value of false . There are times, though, when you do not know the marital status of a person, and the variable should be neither true nor false .
- In C#, you can declare value types to be *nullable* , meaning that they do not yet have a value.
- To make the married variable nullable, the above declaration can be rewritten in two different ways (all are equivalent):

```
Boolean? married = null;
```

```
// ---or---
```

```
Nullable < Boolean > married = null;
```

- To check the value of a nullable variable, use the HasValue property, like this:

```
if (married.HasValue) {
```

```
// ---this line will be executed only
```

```
// if married is either true or false---
```

```
Console.WriteLine(married.Value);    }
```

Value Types

- When dealing with nullable types, you may want to assign a nullable variable to another variable, like this:

```
int? num1 = null;
```

```
int num2 = num1;    // Error
```

- To resolve this, you can use the null *coalescing operator* (??). Consider the following example:

```
int? num1 = null;
```

```
int num2 = num1 ?? 0;
```

```
Console.WriteLine(num2); // ---0---
```

Reference Types

- For reference types, the variable stores a reference to the data rather than the actual data. Consider the following:

```
Button btn1, btn2;  
btn1 = new Button();  
btn1.Text = "OK";  
btn2 = btn1;  
Console.WriteLine("{0} {1}", btn1.Text, btn2.Text);  
btn2.Text = "Cancel";  
Console.WriteLine("{0} {1}", btn1.Text, btn2.Text);
```

- // First Output: OK OK
- When you change btn2 ' s Text property to " Cancel ", you invariably change btn1 ' s Text property, as the second output shows:
- //Cancel Cancel
- That ' s because btn1 and btn2 are both pointing to the same Button object
- To remove the reference to an object in a reference type, simply use the null keyword:
 btn2 = null;

Reference Types

- C# supports two predefined reference types — object and string — which are described in the following table.

C# Type	.NET Framework Type	Descriptions
object	System.Object	Root type from which all types in the CTS (Common Type System) derive
string	System.String	Unicode character string



Value Types vs Reference Types

Value Types versus Reference Types

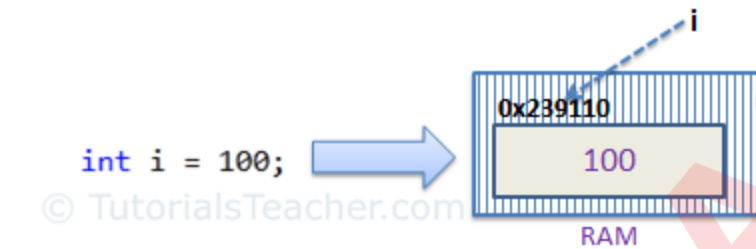
For any discussion about value types and reference types, it is important to understand how the .NET Framework manages the data in memory.

Basically, the memory is divided into two parts — the stack and the heap. The stack is a data structure used to store value-type variables. When you create an `int` variable, the value is stored on the stack. In addition, any call you make to a function (method) is added to the top of the stack and removed when the function returns.

In contrast, the heap is used to store reference-type variables. When you create an instance of a class, the object is allocated on the heap and its address is returned and stored in a variable located on the stack.

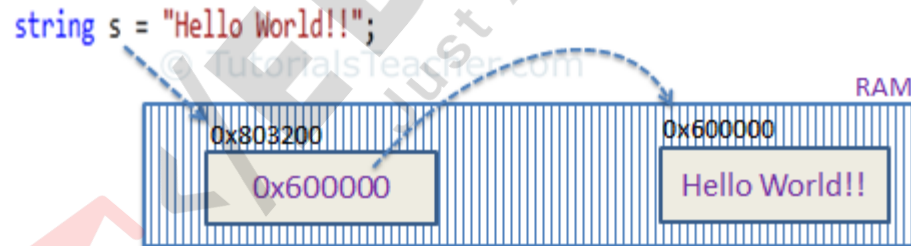
Memory allocation and deallocation on the stack is much faster than on the heap, so if the size of the data to be stored is small, it's better to use a value-type variable than reference-type variable. Conversely, if the size of data is large, it is better to use a reference-type variable.

Value Type assignment



Memory allocation for Value Type

Reference Type assignment



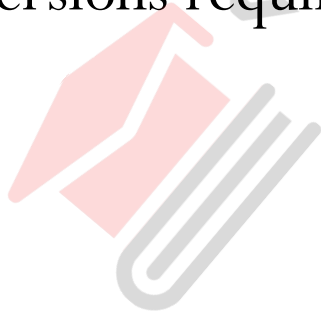
Reference type variable
contains address where the
value is stored

Actual value

Memory allocation for Reference type

Type Conversion (Type Casting)

- Type conversion is converting one type of data to another type. It is also known as Type Casting. In C#, type casting has two forms:
- **Implicit type conversion** - These conversions are performed by C# in a type-safe manner. For example, are conversions from smaller to larger integral types and conversions from derived classes to base classes.
- **Explicit type conversion** - These conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.



Type Conversion (Type Casting)

- C# is a strongly typed language, so when you are assigning values of variables from one type to another, you must take extra care to ensure that the assignment is compatible.

```
int num;  
short sNum = 20;  
num = sNum; //---OK---
```

- *Converting a value from a smaller range to a bigger range is known as widening*

```
num = 5;  
sNum = num; //---not allowed---
```

- The preceding statement could be made valid when you perform a type casting operation

```
num = 5;  
sNum = (short) num; //---sNum is now 5---
```

- ****There may be overflow while typecasting, to avoid this we can use 'check' keyword.**

Correct Set of Code for given data 'a' and 'b' to print output for 'c' as 74 ?

a) `int a = 12;`
`float b = 6.2f;`
`int c;`
`c = a / b + a * b;`
`Console.WriteLine(c);`

b) `int a = 12;`
`float b = 6.2f;`
`int c;`
`c = a / Convert.ToInt32(b) + a * b;`
`Console.WriteLine(c);`

c) `int a = 12;`
`float b = 6.2f;`
`int c;`
`c = a / Convert.ToInt32(b) + a * Convert.ToInt32(b);`
`Console.WriteLine(c);`

d) `int a = 12;`
`float b = 6.2f;`
`int c;`
`c = Convert.ToInt32(a / b + a * b);`

`Console.WriteLine(c);`

Does the output remain same or different for both cases?

- `char l = 'k';`

`float b = 19.0f;`

`int c;`

`c = (l / Convert.ToInt32(b));`

`Console.WriteLine(c);`

- `char l = 'k';`

`float b = 19.0f;`

`int c;`

`c = Convert.ToInt32(l / b);`

`Console.WriteLine(c);`

Correct output for code is?

```
static void Main(string[] args)
{
    float a = 10.553f;
    long b = 12L;
    int c;
    c = Convert.ToInt32(a + b);
    Console.WriteLine(c);
}
```

- a) 23.453
- b) 22
- c) 23
- d) 22.453

Select correct set of code to display the value of given variable 'c' as '25.302'.

a) float a = (double) 12.502f;
float b = 12.80f;
float c;
c = (float) a + b;
Console.WriteLine(c);
Console.ReadLine();

b) float a = 12.502D;
float b = 12.80f;
float c;
c = a + b;
Console.WriteLine(c);
Console.ReadLine();

c) double a = 12.502;
float b = 12.802f;
float c;
c = (float)a + b;
Console.WriteLine(c);
Console.ReadLine();

d) double a = (float) 12.502f;
float b = 12.80f;
float c;
c = a + b;
Console.WriteLine(c);
Console.ReadLine();

++/-- operator

The increment operator (++) or the decrement operator(--) increments or decrements its operand by 1. The increment/decrement operator can appear before or after its operand: ++variable and variable++.

Pre - The result of the operation is the value of the operand after it has been incremented or decremented.

Post - The result of the operation is the value of the operand before it has been incremented or decremented.



Select output for the following set of code.

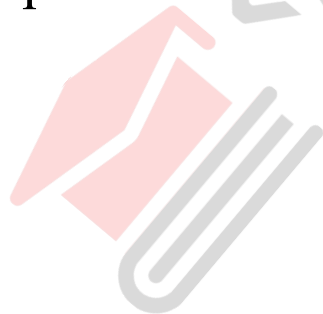
```
static void Main(string[] args)
{
    int a = 5;
    int b = 10;
    int c;
    Console.WriteLine(c = ++ a + b ++);
    Console.WriteLine(b);
    Console.ReadLine();
}
```

- a) 11, 10
- b) 16, 10
- c) 16, 11
- d) 15, 11



Operators

- Arithmetic Operator
- Relational Operator
- Logical Operator
- Assignment Operator
- Increment and Decrement Operator
- Conditional Operator
- Bitwise Operator
- Special Operator



Arithmetic Operator

- $+$: Addition or Unary Plus
- $-$: Subtraction or unary minus
- $*$: Multiplication
- $/$: Division
- $\%$: Modulo Division
- Statements using Arithmetic operator follows BODMAS rule



Output:

```
static void Main(string[] args)
```

```
{
```

```
int a, b, c, x;
```

```
a = 90;
```

```
b = 15;
```

```
c = 3;
```

```
x = a - b / 3 + c * 2 - 1;
```

```
Console.WriteLine(x);
```

```
Console.ReadLine();
```

```
}
```

a) 92

b) 89

c) 90

d) 88



- Output:

```
static void Main(string[] args)
```

```
{
```

```
int a, b, c, x;
```

```
a = 90;
```

```
b = 15;
```

```
c = 3;
```

```
x = a - b / 3 + c * 2 - 1;
```

```
Console.WriteLine(x);
```

```
Console.ReadLine();
```

```
}
```

a) 92

b) 89

c) 90

d) 88



```
static void Main(string[] args)
```

```
{
```

```
int a, b, c, x;
```

```
a = 80;
```

```
b = 15;
```

```
c = 2;
```

```
x = a - b / (3 * c) * (a + c);
```

```
Console.WriteLine(x);
```

```
Console.ReadLine();
```

```
}
```

a) 78

b) -84

c) 80

d) 98



Relational Operator

- <: less than
- <=: less than or equal to
- >: greater than
- >=: greater than or equal to
- ==: equal to
- !=: not equal to

The value of a relational expression is either true or false.

Example:-

a=10;

b=20;

Console.WriteLine(a>b); // will return false

Logical Operator

- &&: Logical AND
- ||: Logical OR
- !: Logical NOT
- &: Bitwise logical AND
- |: Bitwise logical OR
- ^: Bitwise logical exclusive OR

Op 1	Op2	Op1 && Op2	Op1 Op2
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Assignment Operator

- Assignment operators are used to assign the value of an expression to a variable.

Eg: $a = a + 1$

- Shorthand operators can also be used like $a += 1$

Assignment operator	Shorthand operator
$a = a + 1$	$a += 1$
$a = a - 1$	$a -= 1$
$a = a * (n + 1)$	$a *= (n + 1)$
$a = a / (n - 1)$	$a /= (n - 1)$
$a = a \% b$	$a \% = b$

Conditional Operator

- The character pair `?:` is a ternary operator available in C#. This operator is used to construct conditional expressions of the form:
- `exp1?exp2:exp3`
- Eg: `x=(a>b)?a:b`
- It can also be written in the following format:

`If(a>b)`

`x=a;`

`else`

`x=b;`

- Output:-

```
public static void Main(string[] args)
```

```
{
```

```
int a = 4;
```

```
int c = 2;
```

```
bool b = (a % c == 0 ? true : false);
```

```
Console.WriteLine(b.ToString());
```

```
if (a/c == 2)
```

```
{
```

```
Console.WriteLine("true");
```

```
}
```

```
else
```

```
{
```

```
Console.WriteLine("false");
```

```
}
```

```
Console.ReadLine();
```

a) True

False

b) False

True

c) True

True

d) False

False

Bitwise Operator

A=60 // 0011 1100 B=13 // 0000 1101

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 effect of 'flipping' bits.	(~A) = 61, 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

```
static void Main(string[] args)
{
byte varA = 10;
byte varB = 20;
long result = varA & varB;
Console.WriteLine("{0} AND {1} Result : {2}", varA, varB, result);
varA = 10;
varB = 10;
result = varA & varB;
Console.WriteLine("{0} AND {1} Result : {2}", varA, varB, result);
Console.ReadLine();
}
```

- a) 0, 20
- b) 10, 10
- c) 0, 10
- d) 0, 0

```
static void Main(string[] args)
{
    byte varA = 10;
    byte varB = 20;
    long result = varA | varB;
    Console.WriteLine("{0} AND {1} Result : {2}", varA, varB,
    result);
    varA = 10;
    varB = 10;
    result = varA & varB;
    Console.WriteLine("{0} AND {1} Result : {2}", varA, varB,
    result);
    Console.ReadLine();
}
```

- a) 0, 20
- b) 10, 10
- c) 0, 10
- d) 0, 0

Control Structures

- Decision making Structures or conditional statements
- Loops

Decision making structures or Conditional Statements

- if
- If-else
- Nested if
- switch



Control Structures

- if Statement

The if statement consists of a boolean expression followed by one or more statements.

- if-else Statement

The **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false.

- Nested if

An if or if-else statement inside another if or if-else statement.

- Switch

A **switch** statement allows a variable to be tested for equality against a list of values.

Switch

- Example

```
switch (caseSwitch)
```

```
{
```

```
    case 1:
```

```
        Console.WriteLine("Case 1");
```

```
        break;
```

```
    case 2:
```

```
        Console.WriteLine("Case 2");
```

```
        break;
```

```
    default:
```

```
        Console.WriteLine("Default case");
```

```
        break;
```

```
}
```

Switch

- In C#, fall - throughs are not allowed;
- That is, each case block of code must include the break keyword so that execution can be transferred out of the switch block (and not “fall through” the rest of the case statements).
- However, there is one exception to this rule — when a case block is empty. Here ’ s an example:

```
string symbol = "INTC";  
switch (symbol)  
{  
    case "MSFT": Console.WriteLine(27.96);  
        break;  
    case "GOOG": Console.WriteLine(437.55);  
        break;  
    case "INTC":  
    case "YHOO": Console.WriteLine(27.15);  
        break;  
    default: Console.WriteLine("Stock symbol not recognized");  
        break;  
}
```

- The case for “INTC” has no execution block/statement and hence the execution will fall through into the case for “YHOO”, which will incorrectly print the output “27.15”. In this case, you need to insert a break statement after the “INTC” case to prevent the fall - through

Loops

- A loop is a statement, or set of statements, repeated for a specified number of times or until some condition is met.
- C# supports the following looping constructs:
 - for
 - foreach
 - while and do – while
- For:
- Normal for loop..... `for (int i=0; i < 9; i++) { //code statements ... }`
- Foreach

```
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
foreach (int i in nums)
{
    i += 4; //---error: cannot change the value of i---
    Console.WriteLine(i);
}
```

While, do while loops

- The while statement checks the condition before executing the block of code.
- To execute the code at least once before evaluating the condition, use the do - while statement.
- do-while loop can be used to write Menu Driven program

```
string reply;  
do  
{  
    Console.WriteLine("Are you sure you want to quit? [y/n]");  
    reply = Console.ReadLine();  
} while (reply != "y");
```

Exiting from a loop

- To break out of a loop prematurely (before the exit condition is met), you can use one of the following keywords:
 - break
 - return
 - throw
 - goto
- Break:
 - The break keyword allows you to break out of a loop prematurely.
- Return:
 - The return keyword allows you to terminate the execution of a method and return control to the calling method. When you use it within a loop, it will also exit from the loop.

Return

class Program

```
{  
    static string FindWord(string[] arr, string word)  
    {  
        foreach(string w in arr)  
        {  
            if(w.StartsWith(word))  
                return w;  
        }  
        return string.Empty;  
    }  
    static void Main(string[] args)  
    {  
        string[] words = {"abcd","efgh","ijkl"};  
        Console.WriteLine(FindWord(words,"efgh"));  
        Console.ReadLine();  
    }  
}
```


Skipping an iteration.... Use of 'continue'

- To skip to the next iteration in the loop, you can use the continue keyword. Consider the following block of code:

```
for (int i = 0; i < 9; i++)
{
    if (i % 2 == 0)
    {
        //---print i if it is even---
        Console.WriteLine(i);
        continue;
    }
    //---print this when i is odd---
    Console.WriteLine("*****");
}
```



Pre-processor Directives

- The pre-processor Directives tells the compiler to process information before compilation starts.
- All pre-processor directives start with a #.
- Pre-processor directives are not statements and thus does not end with a semi-colon.



Pre-processor Directives

Pre-processor Directives	
<code>#define</code> <code>#undef</code>	Define and undefine symbols
<code>#if</code> <code>#else</code> <code>#elif</code> <code>#endif</code>	Use logic to see if symbol is defined or not
<code>#region</code> <code>#endregion</code>	Used for documentation of code.

C# Fundamentals

Part 1

Ends

