

Day 3 C#



Authorized & published by Summitworks Technologies Inc



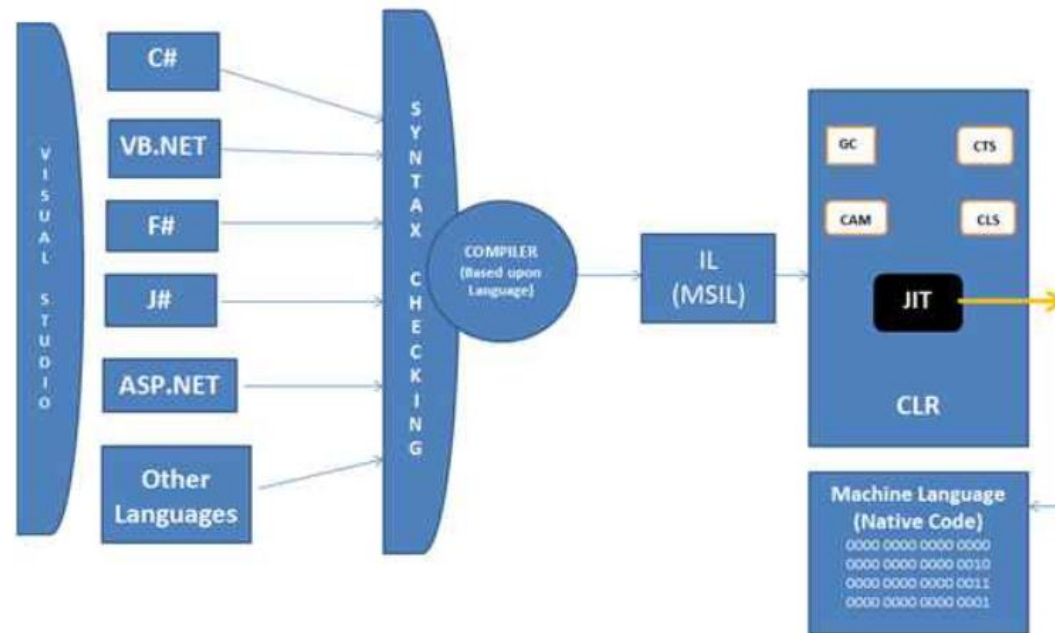
Agenda Day 3

- Introduction to .Net Framework
 - .Net Framework Architecture
 - .Net components
 - .Net Design principles
- C# and .Net version history
 - .Net Framework version history
 - C# version history
- Setting up the Environment
 - Download and Install Visual Studio
- Datatypes
 - Integer
 - Float
 - Double
 - Boolean
 - String
 - Nullable
- Implicit Conversion
- Namespace
- Variables and Operators
 - Arithmetic operator
 - Relational operator
 - Logical operator
 - Assignment operator
 - Expressions
- Null Operator
- Conditional Statements
 - IF
 - Loops (While and Do While)
 - Switch
 - Foreach
- Pass by Value, by reference and out parameters
- C# Enum
- Arrays
- StringBuilder
- Boxing and Unboxing

Introduction to .Net Framework

.Net Framework Architecture

- The .NET Framework is a class of reusable libraries (collection of classes) given by Microsoft to be used in other .Net applications and to develop, build and deploy many types of applications on the Windows platforms
- Using .NET Framework we can develop Console Applications, Windows Forms, Window Presentation Foundation(WPF), Web Applications, Windows Communication Foundation (WCF)
- Primarily it runs on the Microsoft windows operating system



Introduction to .Net Framework Cont

Compiling a .NET program

What really happens when we compile a .NET program?

- The exe file that is created doesn't contain executable code, rather it's MicroSoft Intermediate Language (MSIL) code
- When you run the EXE, a special runtime environment (the Common Language Runtime or CLR) is launched and the IL instructions are executed by the CLR to the machine language
- The CLR comes up with a Just In Time Compiler that translates the IL to native language the first it is encountered

So the process of programming goes through like:

- We write a program in C#, VB.Net and other languages
- We compile our code to IL code based on the language compiler (csc.exe, vbc.exe and so on)
- Run your IL program that launches the CLR to execute your IL, using its JIT to translate your program into native code as it executes

Introduction to .Net Framework Cont

.NET Framework Components

It mainly contains two components,

Common Language Runtime (CLR)

.Net Framework Class Library

Common Language Runtime(CLR)

- Common Language Runtime (CLR) provides an environment to run all the .Net Programs
- Role of CLR is to locate, load and manage .NET objects
- The code which runs under the CLR is called as Managed Code
- CLR also helps us with memory management
- CLR allocates the memory for scope and de-allocates the memory if the scope is completed
- Language Compilers (e.g. C#, VB.Net, J#) will convert the Code/Program to Microsoft Intermediate Language(MSIL) intern this will be converted to Native Code by CLR

Introduction to .Net Framework Cont

.NET Framework Class Library (FCL)

- Framework Class Library also known as Base Class Library BCL
- BCL defines types that can be used to build any type of software application
- Developers can use BCL for database interactions, reading and writing to file

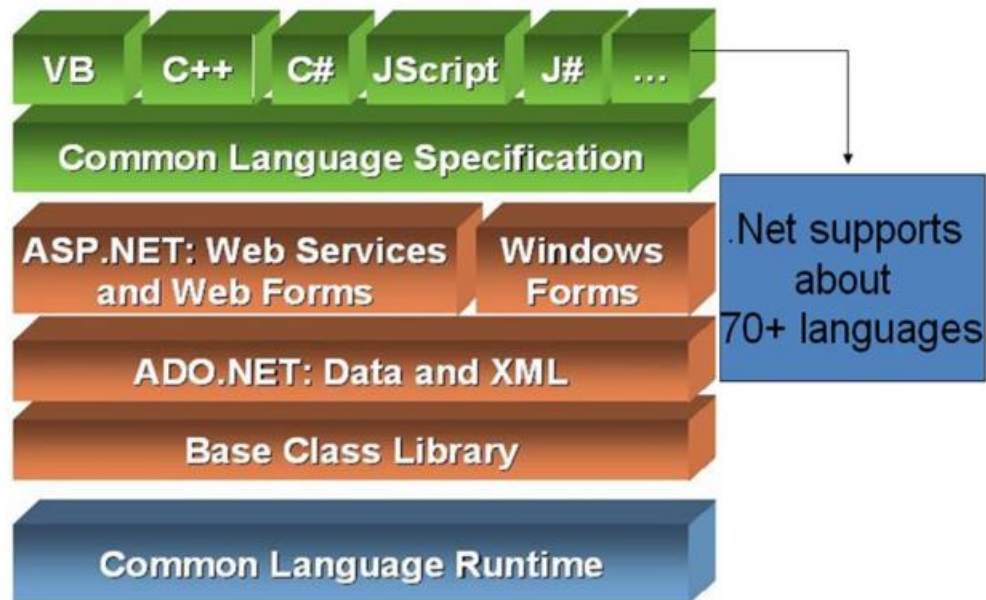
Common Type System (CTS)

- CTS describes all possible data types and all programming rules and conventions supported by the runtime
- The communication between programs written in any .NET compliant language, the types have to be compatible on the basic level

Introduction to .Net Framework Cont

Common Language Specification (CLS)

- It is a sub set of CTS and it specifies a set of rules that needs to be adhered or satisfied by all language compilers targeting CLR
- It helps in cross language inheritance and cross language debugging



C# Version History

C# Version	.Net Version	Visual Studio	Features	Year
C# 1.0	.Net Framework 1.0/1.1	2002	<ul style="list-style-type: none">• Automatic Memory management using garbage collection• Attribute based programming	2002
C# 2.0	.Net Framework 2.0	2005	<ul style="list-style-type: none">• Generic Types and members• Anonymous methods• Partial types• Nullable types	2005

C# Version History

C# Version	.Net Version	Visual Studio	Features	Year
C# 3.0	.Net Framework 3.0/3.5	2008	<ul style="list-style-type: none">• Strongly-typed queries using LINQ• Anonymous types• Extension methods• Lambda expressions• Object initialization	2008
C# 4.0	.Net Framework 4.0	2010	<ul style="list-style-type: none">• Dynamic keyword• Optional parameter/named method argument	2010
C# 5.0	.Net Framework 4.5	2012/2013	<ul style="list-style-type: none">• Asynchronous programming	2012

C# Version History

C# Version	.Net Version	Visual Studio	Features	Year
C# 6.0	.Net Framework 4.6	2013/2015	<ul style="list-style-type: none">• Static imports• Exception filters• Null propagator• String Interpolation• Auto-property Initializer	
C# 7.0	.Net Core	2017	<ul style="list-style-type: none">• out variables• Tuples• Discards• Pattern Matching• Local functions	

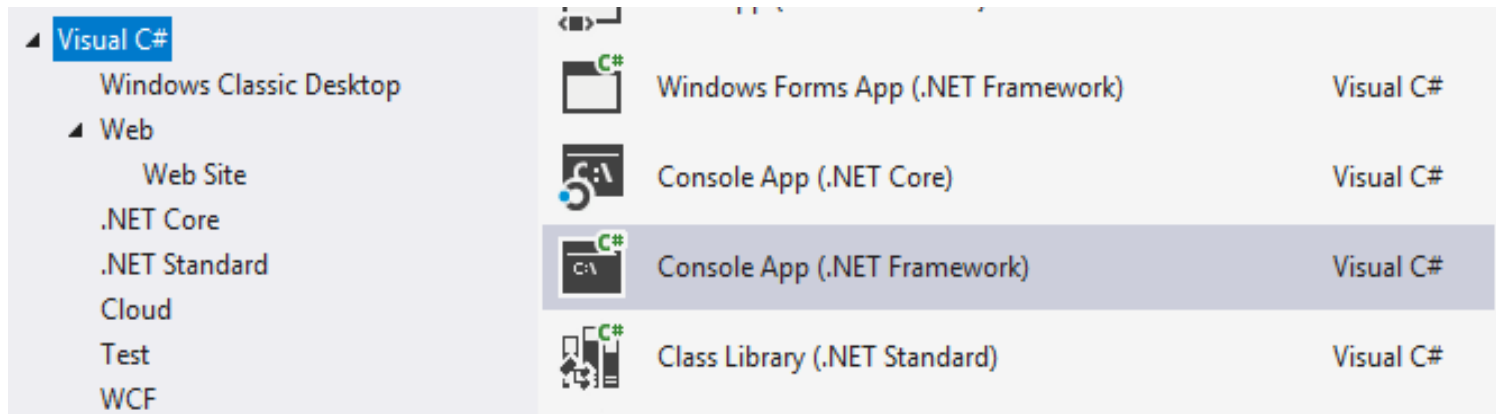
First Console Application

Download and Install the latest version of Visual Studio community from visualstudio.com

C# can be used in a window-based, web-based, or console application. To start with, we will create a console application to work with C#.

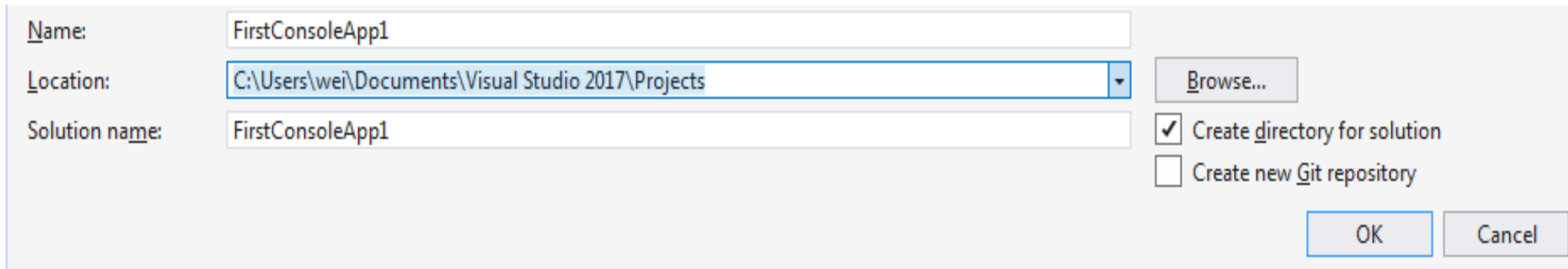
Step 1: Open Visual Studio 2017 installed on your local machine. Click on File -> New Project... from the top menu.

Step 2: From the **New Project** popup select Visual C# in the left side panel and select Console App in the right side panel.



First Console Application

Give an appropriate project name and location where you want to create all your project files and solution name. Click Ok to create the console project.



Name: FirstConsoleApp1

Location: C:\Users\wei\Documents\Visual Studio 2017\Projects

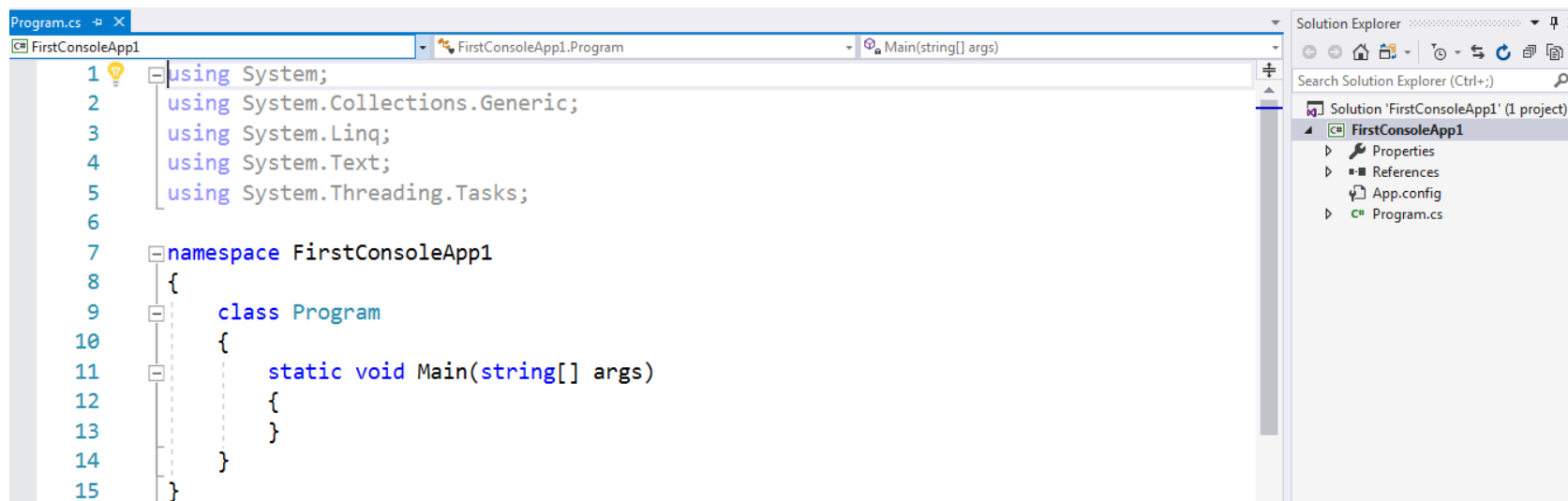
Solution name: FirstConsoleApp1

☒ Create directory for solution

☐ Create new Git repository

OK Cancel

program.cs will be created by default where you can write your c# code



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace FirstConsoleApp1
8 {
9     class Program
10     {
11         static void Main(string[] args)
12         {
13         }
14     }
15 }
```

Solution Explorer: FirstConsoleApp1 (1 project)

- Properties
- References
- App.config
- Program.cs

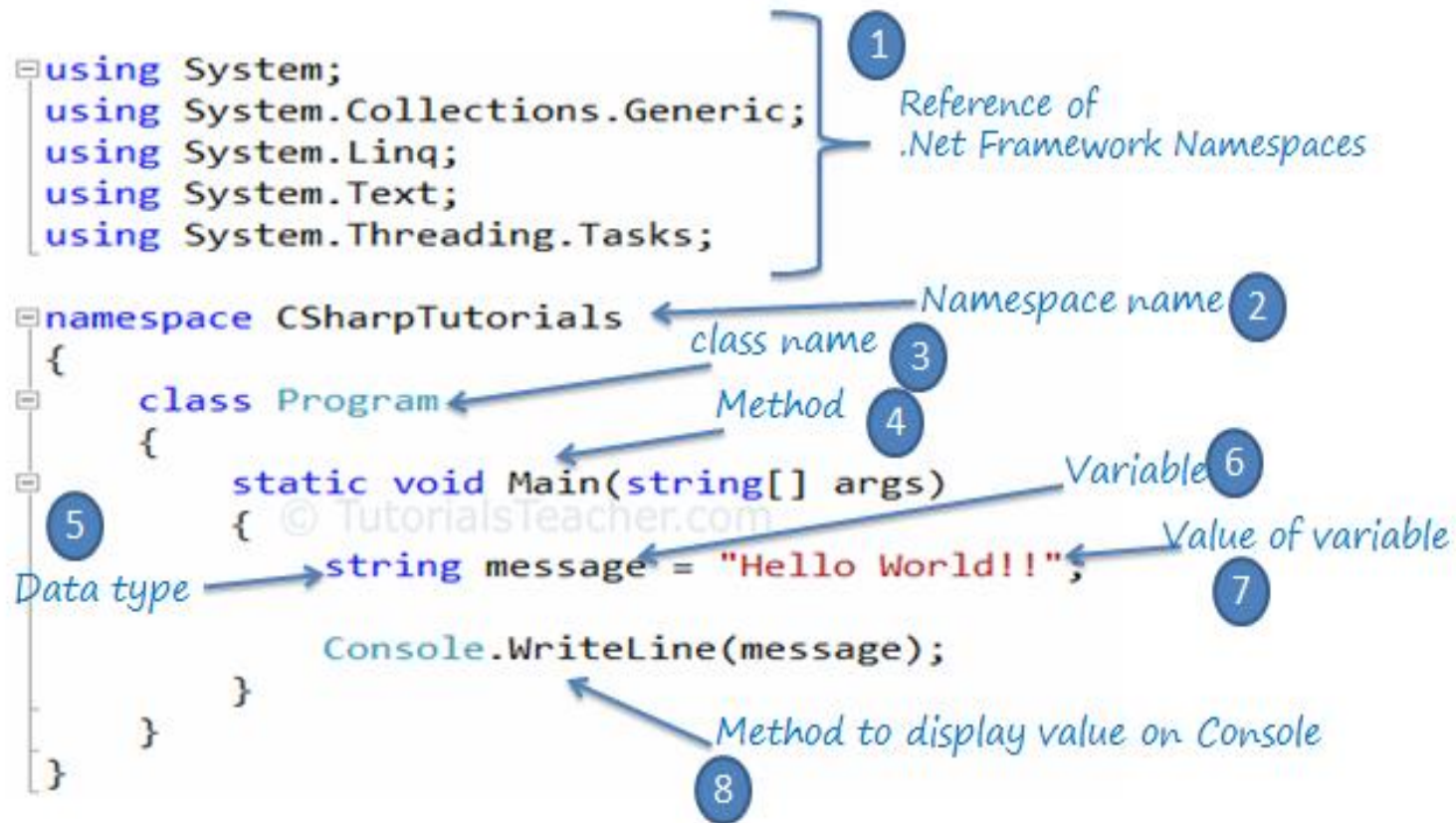
First Console Application

Lets us see a simple example that displays “Hello World!!” on the console.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FirstConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            string message = "Hello World!!";
            Console.WriteLine(message);
        }
    }
}
```

C# Code Structure



C# Code Structure

Data Types - Boolean

- Boolean:
 - The *bool* (boolean) data type is one of the simplest found in the .NET framework, because it only has two possible values: false or true.
 - By default, the value of a bool is false, but you can of course change that - either when you declare the variable or later on
 - Working with a boolean value usually means checking its current state and then reacting to it

0 references

```
static void Main(string[] args)
{
    bool isAdult = true;
    if (isAdult == true)
        Console.WriteLine("An adult");
    else
        Console.WriteLine("A child");
}
```

```
static void Main(string[] args)
{
    bool isAdult = true;
    if (isAdult)
        Console.WriteLine("An adult");
    else
        Console.WriteLine("A child");
}
```

```
bool isAdult = true;
if (!isAdult)
    Console.WriteLine("NOT an adult");
else
    Console.WriteLine("An adult");
```


Data Types - Integer

- **byte** - an unsigned integer which can hold a number between 0 and 255.
- **short** - a 16-bit signed integer, which can hold a value between -32,768 and 32,767. Also known under its formal name *Int16*.
- **long** - a 64-bit integer, which can hold a value between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807. Also known under its formal name *Int64*.

Math with integers

```
int a = 42;  
int b = 8;  
  
Console.WriteLine(a + b);
```

```
int a = 42;  
int b = 8;  
  
Console.WriteLine(200 - ((a + b) * 2));
```

Type conversion/casting

Implicit conversion

```
int a = 10;  
long b = a;
```

Explicit conversion

```
long a = 10;  
int b = (int)a;
```

Variables

- A variable is a name given to the data value the content of the variable can vary at anytime as long as it is accessible
- Variable is always defined with a data type
- A variable can be declared and initialized later, or it can be declared and initialized at the same time
- Multiple variables can be defined separated by comma (,) in a single or multiple line till semicolon(;
- A value must be assigned to a variable before using it otherwise it will give compile time error
- Variables allow you to store data of various types, e.g. text strings, numbers or custom objects. There are local variables, which are accessible inside of the method in which it has been defined, and then there are class fields, which can be accessed from all the methods of the class and even outside of the class, if the *visibility* permits it.

String and Number Variables

String Variables

0 references

```
static void Main(string[] args)
{
    string firstName = "John";
    string lastName = "Doe";

    Console.WriteLine("Name: " + firstName + " " + lastName);

    Console.WriteLine("Please enter a new first name:");
    firstName = Console.ReadLine();

    Console.WriteLine("New name: " + firstName + " " + lastName);

    Console.ReadLine();
}
```

Number Variables

```
static void Main(string[] args)
{
    int number1, number2;

    Console.WriteLine("Please enter a number:");
    number1 = int.Parse(Console.ReadLine());

    Console.WriteLine("Thank you. One more:");
    number2 = int.Parse(Console.ReadLine());

    Console.WriteLine("Adding the two numbers: " + (number1 + number2));

    Console.ReadLine();
}
```

Variable Scope

- In C# the variables defined inside a method can't be accessed outside by other methods these variables are local variables
- C# doesn't support global variables instead we can declare a field on a class which can be accessed from all the methods in that class

```
2 references
class VariableScope
{
    private static string helloClass = "Hello, class!";
    0 references
    static void Main(string[] args)
    {
        string helloLocal = "Hello, local!";
        Console.WriteLine(helloLocal);
        Console.WriteLine(VariableScope.helloClass);
        DoStuff();
    }
    1 reference
    static void DoStuff()
    {
        Console.WriteLine("A message from DoStuff: " + VariableScope.helloClass)
    }
}
```

Variables and Parameters

- A variable represents a storage location that has a modifiable value
- A variable can be local variable, parameter, field, or array element
- The Stack and the Heap are the places where variables and constants reside, each has very different lifetime semantics
- Stack is the block of memory for storing local variables and parameters
- Stack logically grows and shrinks as a function is entered and exited

- Static int Factorial (int x)
 {
 if(x==0) return 1;
 return x * Factorial(x-1);
 }

Operators

- Arithmetic Operators are used to perform mathematic operations on numbers
- The list of operators available in C#

Operator	Description
+	Adds two operands
-	Subtracts the second operand from the first
*	Multiplies both operands
/	Divides the numerator by de-numerator
%	Modulus Operator and a remainder of after an integer division
++	Increment operator increases integer value by one
--	Decrement operator decreases integer value by one

Operators

- Relational operators are used for performing relational operation on numbers
- The list of Relational operators available in C#

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

Operators

- Logical operators are used for performing logical operations on values
- List of Logical operators available in C#

Operator	Description
&&	This is the Logical AND operator. If both the operands are true, then condition becomes true.
	This is the Logical OR operator. If any of the operands are true, then condition becomes true.
!	This is the Logical NOT operator.

Null Operator

- The ?? Operator is called the null coalescing operator that simplifies checking for null values
- It can be used with both nullable types and reference types
- It returns the left hand operand if the operand is not null, otherwise it returns the right hand operand
- Let's see an example

```
int? x = null;

// Set y to the value of x if x is NOT null; otherwise,
// if x == null, set y to -1.
int y = x ?? -1;

// Assign i to return value of the method if the method's result
// is NOT null; otherwise, if the result is null, set i to the
// default value of int.
int i = GetNullableInt() ?? default(int);

string s = GetStringValue();
// Display the value of s if s is NOT null; otherwise,
// display the string "Unspecified".
Console.WriteLine(s ?? "Unspecified");
```

Selection Statements

- A selection statement causes the program control to be transferred to a specific flow based upon whether a certain condition is true or false
- The following keywords are used in selection statements
 - If
 - Else
 - switch
 - case
 - default
- The if statement is used to evaluate a boolean expression before executing a set of statements. If an expression evaluates to true, then it will run one set of statements else it will run another set of statements.

Selection Statements

```
{  
    int m = 12;  
    int n = 18;  
  
    if (m > 10)  
        if (n > 20)  
        {  
            Console.WriteLine("Result1");  
        }  
        else  
        {  
            Console.WriteLine("Result2");  
        }  
}
```

Switch Statement

- The switch statement is used to evaluate an expression and run different statements based on the result of the expression. If one condition does not evaluate to true, the switch statement will then move to the next condition and so forth

Selection Statements

```
public static void Main()
{
    int caseSwitch = 1;

    switch (caseSwitch)
    {
        case 1:
            Console.WriteLine("Case 1");
            break;
        case 2:
            Console.WriteLine("Case 2");
            break;
        default:
            Console.WriteLine("Default case");
            break;
    }
}
```

Iteration Statements

- Iteration statements are used to create loops. Iteration statements cause embedded statements to be executed a number of times, subject to the loop-termination criteria
- The following keywords are used in the Iteration Statements:
 - while
 - do
 - for
 - foreach
- The while loop executes a statement or block of statements while a specified boolean expression evaluates to true. The expression is evaluated before each execution of the loop
- We can break out of the while loop at any point using the break statement.

```
int n = 0;
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
```

Iteration Statements

- The do statement executes a statement or block of statements while a specified boolean expression evaluates to true. The expression is evaluated after each execution of the loop it is executed one or more times
- We can break out of the while loop at any point using the break statement

```
int n = 0;  
do  
{  
    Console.WriteLine(n);  
    n++;  
} while (n < 5);
```

- The foreach statement allows the iteration of processing over the elements in arrays and collections.
- Within the foreach loop parentheses, the expression consists of two parts separated by the keyword in. To the right of in is the collection, and to the left is the variable with the type identifier matching whatever type the collection returns.

Iteration Statements

Each iteration queries the collection for a new value for *i*. As long as the collection `intNumbers` returns a value, the value is put into the variable *i* and the loop will continue. When the collection is fully traversed, the loop will terminate.

```
Int16[] intNumbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };  
foreach (Int16 i in intNumbers)  
{  
    System.Console.WriteLine(i);  
}
```

For loop is used when we want to repeat a certain set of statements for a particular number of times. The for statements defines initializer, condition, and Iterator sections

```
static void Main(string[] args)  
{  
    for (Int32 i = 0; i < 3; i++)  
    {  
        Console.WriteLine(i);  
        Console.ReadKey();  
    }  
}
```

Value and Reference Types

- The data type is a value type if it holds a data value within its own memory space.
- The value types are all stored in the stack memory
- The value types cannot be a null value
- Example: `int i = 100;`

This value 100 is stored in some memory location that is allocated for i

The following data types are all value types:

- Bool
- byte
- int
- double
- decimal
- float
- enum

Passing by value

- When you pass a value type variable from one method to another method, the system creates a separate copy of a variable in another method, so that if value got changed in the one method won't affect on the variable in another method.

Example: In this code the value of i in Main method remains unchanged even after passing it to the ChangeValue() methods and change it's value there

```
static void ChangeValue(int x)
{
    x = 200;

    Console.WriteLine(x);
}

static void Main(string[] args)
{
    int i = 100;

    Console.WriteLine(i);

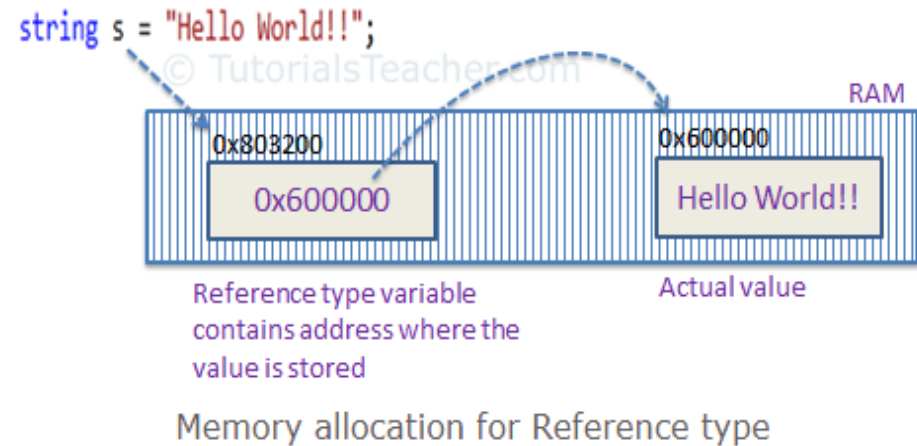
    ChangeValue(i);

    Console.WriteLine(i);
}
```

Reference Type

- Reference types contains a pointer to another memory location that holds the data
- Reference type are by default has null value

Example: String s = "Hello World";



The following data types are of reference types:

- String
- All arrays, even if their elements are value types
- Class
- Delegates

Passing by reference

- When you pass a reference type variable from one method to another, it doesn't create a new copy; instead, it passes the address of the variable. If we now change the value of the variable in a method, it will also be reflected in the calling method.

In the code when we pass the Student object std1 to the ChangeReferenceType() method we are sending the memory address of the std1. When the method changes the StudentName it actually changes the StudentName of std1.

```
static void ChangeReferenceType(Student std2)
{
    std2.StudentName = "Steve";
}

static void Main(string[] args)
{
    Student std1 = new Student();
    std1.StudentName = "Bill";

    ChangeReferenceType(std1);

    Console.WriteLine(std1.StudentName);
}
```

Enum

- The enum keyword is used to declare an enumeration, a distinct type that consists of a set of named constants called the enumerator list
- Enum can be defined directly within the namespace so that all the classes in the namespace can access it equally
- By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1
- we will define an enumeration called days, which will be used to store the days of the week

```
public class EnumTest
{
    enum Day { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

    static void Main()
    {
        int x = (int)Day.Sun;
        int y = (int)Day.Fri;
        Console.WriteLine("Sun = {0}", x);
        Console.WriteLine("Fri = {0}", y);
    }
}
```

Arrays

- An array is a special type of data type which can store fixed number of values sequentially
- Array can be declared using a type name followed by square brackets[]

Example: Array declaration in C#

```
int[] intArray; // can store int values

bool[] boolArray; // can store boolean values

string[] stringArray; // can store string values

double[] doubleArray; // can store double values

byte[] byteArray; // can store byte values

Student[] customClassArray; // can store instances of Student class
```


Arrays

- Array can be declared and initialized at the same time using the new keyword

Example: Array Declaration & Initialization

```
// defining array with size 5. add values later on
int[] intArray1 = new int[5];

// defining array with size 5 and adding values at the same time
int[] intArray2 = new int[5]{1, 2, 3, 4, 5};

// defining array with 5 elements which indicates the size of an array
int[] intArray3 = {1, 2, 3, 4, 5};
```

- Arrays can be initialized after declaration
- String [] strArray1;
- strArray1 = new string[4]{"1st Element", "2nd Element, 3rd Element, 4th Element};

We can assign values to array index like int[] intArray = new int[3];

intArray[0] = 10;

StringBuilder

- StringBuilder is a dynamic object that allows you to expand the number of characters in the string. It doesn't create a new object in the memory but dynamically expands memory to accommodate the modified string
- StringBuilder is found in the System.Text namespace we have to import this namespace in our code to use it
- We have to create a new instance of the stringBuilder class by initializing to a variable

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
```
- We can set the capacity and the length of stringBuilder by specifying the size

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!", 25);
```
- We can modify the contents of the stringBuilder using the build-in methods
- StringBuilder performs faster than string when concatenating multiple strings

StringBuilder

- List of methods we can use to modify the content of the stringbuilder

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

StringBuilder

- Use Append method of StringBuilder to add or append a string to StringBuilder. AppendLine() method appends the string with a newline at the end

Example:

```
StringBuilder sb = new StringBuilder("Hello ",50);

sb.Append("World!!");
sb.AppendLine("Hello C#!");
sb.AppendLine("This is new line.");

Console.WriteLine(sb);
```

AppendFormat() is used to format input string into specified format and then append it

```
StringBuilder amountMsg = new StringBuilder("Your total amount is ");
amountMsg.AppendFormat("{0:C} ", 25);

Console.WriteLine(amountMsg);
```

StringBuilder

- Insert() method inserts the string at a specified index in stringbuilder

```
StringBuilder sb = new StringBuilder("Hello World!!",50);  
sb.Insert(5," C#");
```

```
Console.WriteLine(sb);
```

- Remove() method removes the string at a specified index with specified length

```
StringBuilder sb = new StringBuilder("Hello World!!",50);  
sb.Remove(6, 7);
```

```
Console.WriteLine(sb);
```

StringBuilder

- Replace() method replaces all occurrence of a specified string with a specified replacement string

```
StringBuilder sb = new StringBuilder("Hello World!!",50);  
sb.Replace("World", "C#");  
  
Console.WriteLine(sb);
```

- Indexer is used to set or get a character at specified index

```
StringBuilder sb = new StringBuilder("Hello World!!");  
  
for(int i=0; i< sb.Length; i++)  
    Console.Write(sb[i]);
```

StringBuilder

- ToString() method is used to read a string from the stringbuilder

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

```
string str = sb.ToString(); // "Hello World!!"
```

Stack and Heap Memory

- The Heap is the block of memory in which objects (i.e reference type instances) resides
- Whenever a new object is created, it is allocated on the heap and a reference to that object is returned
- An object can't be deleted explicitly, an unreferenced object is eventually collected by the garbage collector
- C# enforces definite assignment, so it is impossible to access uninitialized memory
- Local variables must be assigned a value before they can be read
- Function arguments must be supplied when a method is called(unless marked as optional parameter)
- All other variables (such as fields and array elements) are automatically initialized by the runtime

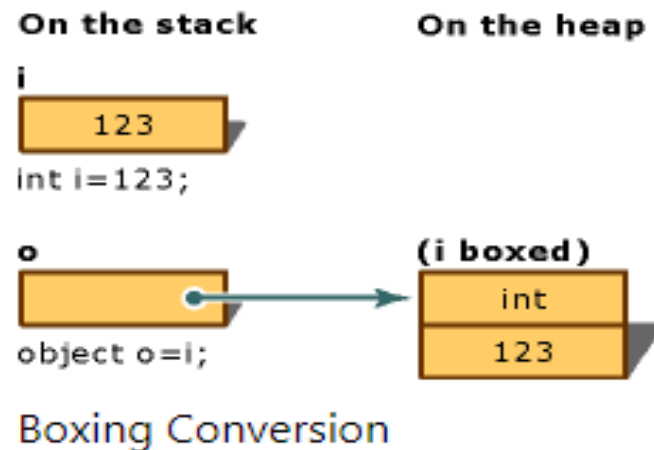
Boxing and Unboxing

- Boxing is an implicit conversion of a value type to the type object
- Boxing a value type allocates an object instance on the heap and copies the value into the new object
- Example `int i = 123;`

Implicitly applies the boxing operation on the variable i.

`object o = i;`

A reference of object o is created on the stack that references a value of the type int on the heap.

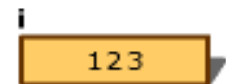


Unboxing

- Unboxing is an explicit conversion from the type object to a value type

```
int i = 123;    // a value type
object o = i;   // boxing
int j = (int)o; // unboxing
```

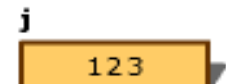
On the stack



int i=123;



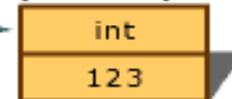
object o=i;



int j=(int) o;

On the heap

(i boxed)



Unboxing Conversion