Unit 1

.NET Framework & Language Constructs

Introduction to .NET Framework

- .NET is a software framework which is designed and developed by Microsoft.
- first version of the .Net framework was 1.0 which came in the year 2002.
- and the current version is ~~4.7.1~~ 4.8.1
- In easy words, it is a virtual machine for compiling and executing programs written in different languages like C#, VB.Net etc.
- used to develop Windows Form-based applications, Web-based applications, and Web services.
- VB.Net and C# being the most common ones.
- It is used to build applications for Windows, phone, web, etc.
- .NET is not a language (Runtime and a library for writing and executing written programs in any compliant language)

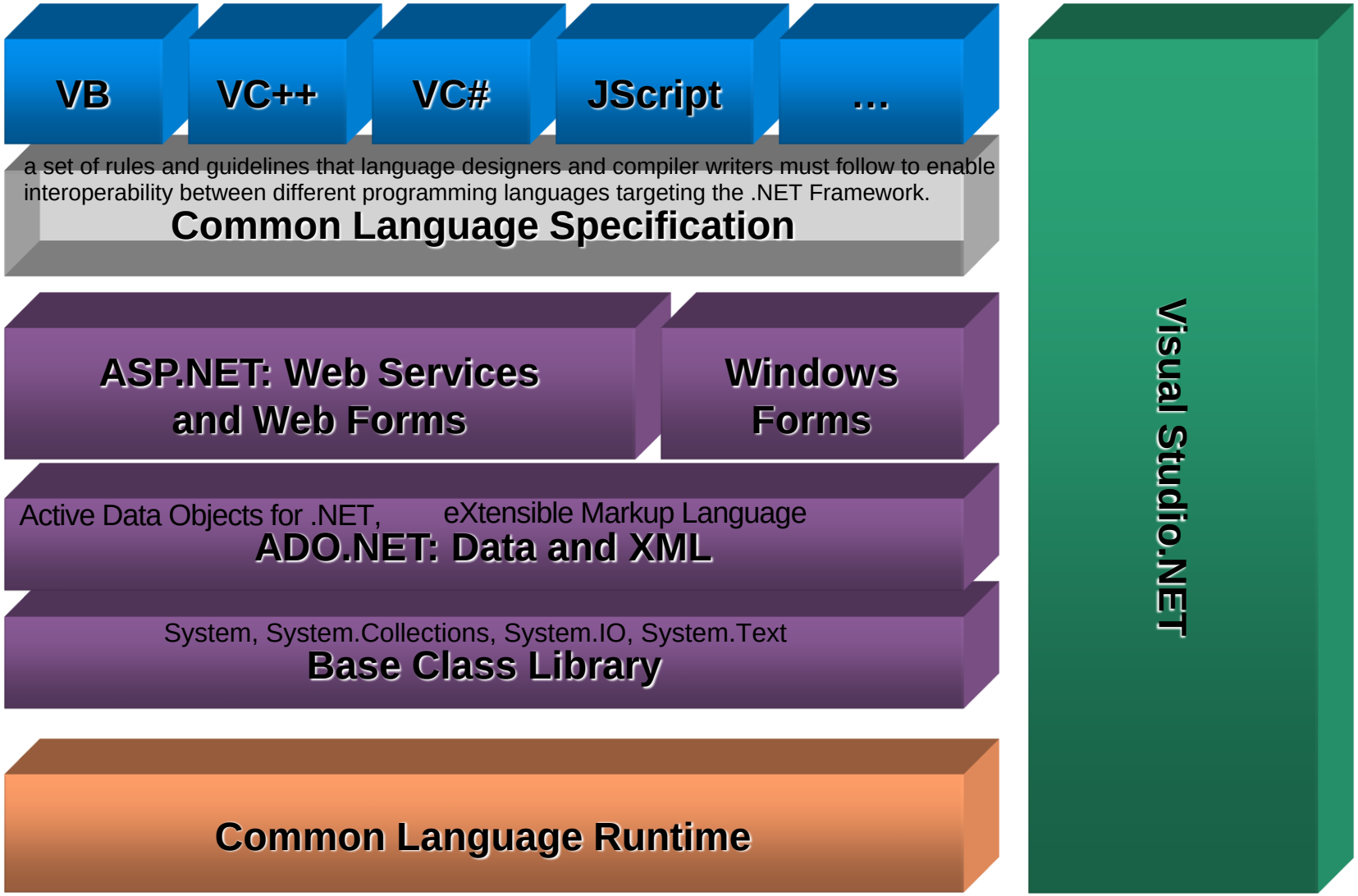
Introduction to .NET Framework

- .NET Framework supports more than 60 programming languages in which 11 are designed and developed by Microsoft,
- Some of them includes:
  - C#.NET
  - VB.NET
  - C++.NET
  - J#.NET
  - F#.NET
  - JSCRIPT.NET
  - WINDOWS POWERSHELL

Framework, Languages, And Tools

Microsoft Intermediate Language (MSIL)

Just In Time (JIT) compiler



Example of XML

```
<person>
  <name>John
Doe</name>
  <age>30</age>
  <city>New York</city>
</person>
```

The .NET Framework

.NET Framework Services

- Common Language Runtime
- Windows® Forms
- ASP.NET
  - Web Forms
  - Web Services
- ADO.NET, evolution of ADO
- Visual Studio.NET

Common Language Runtime (CLR)

- CLR works like a virtual machine in executing all languages.
- All .NET languages must obey the rules and standards imposed by CLR. Examples:
  - Object declaration, creation and use
  - Data Types, language libraries
  - Error and exception handling
  - Interactive Development Environment (IDE)

Common Language Runtime(CLR)

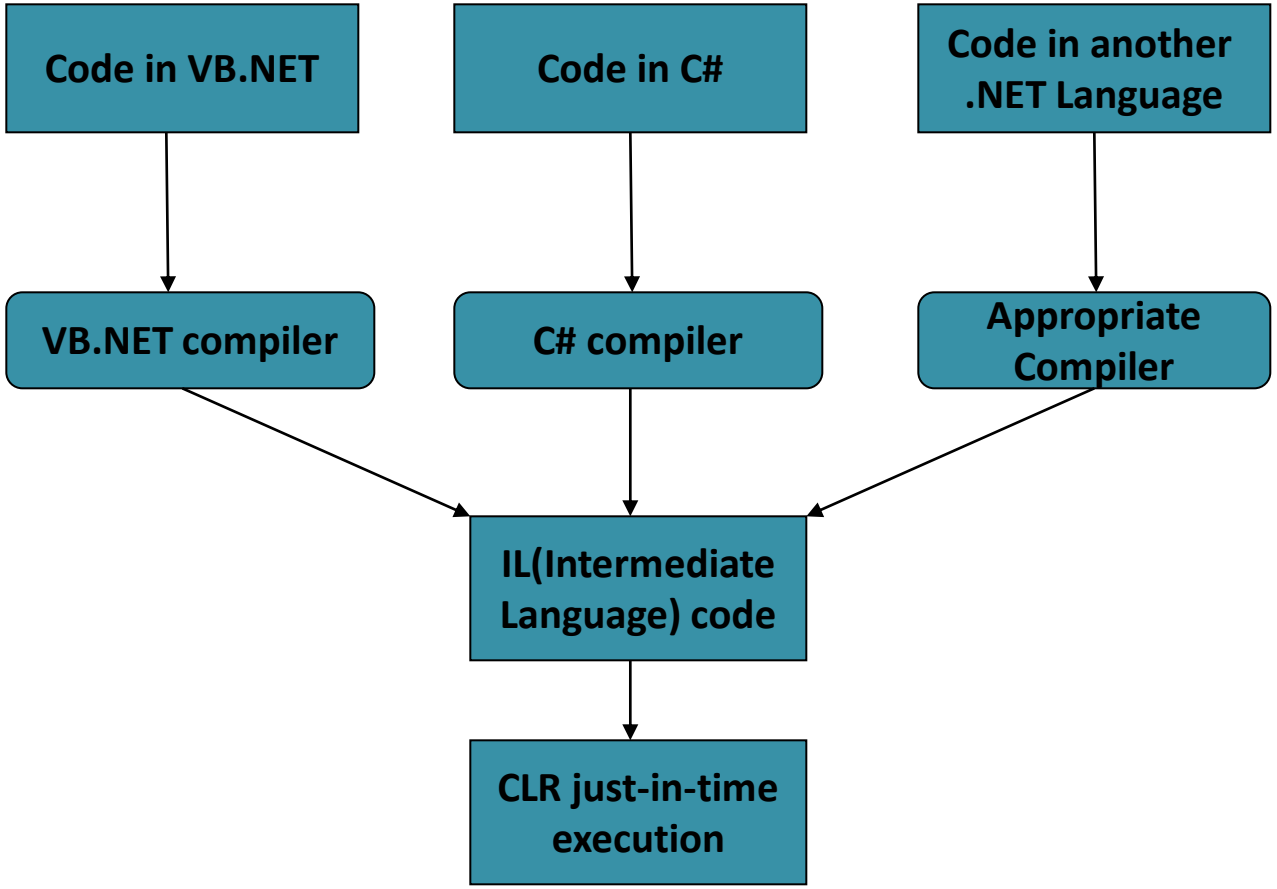
- Development
  - Mixed language applications
    - Common Language Specification (CLS)
    - Common Type System (CTS)
    - Standard class framework
    - Automatic memory management
  - Consistent error handling and safer execution
  - Potentially multi-platform
- Deployment
  - Removal of registration dependency
  - Safety – fewer versioning problems

Common Language Runtime

Multiple Language Support

- CTS is a rich type system built into the CLR
  - Implements various types (int, double, etc)
  - And operations on those types
- CLS is a set of specifications that language and library designers need to follow
  - This will ensure interoperability between languages

Compilation and Execution of .NET Application



Compilation and Execution of .NET Application

- Any code written in any .NET complaint languages when compiled, converts into MSIL (Microsoft Intermediate Language) code in form of an assembly through CLS, CTS.
- IL is the language that CLR can understand.
- On execution, this IL is converted into binary code(machine code) by CLR's just in time compiler (JIT) and these assemblies or DLL are loaded into the memory.
- Compilation can be done with Debug or Release configuration. The difference between these two is that in the debug configuration, only an assembly is generated without optimization. However, in release complete optimization is performed without debug symbols.

Basic Languages constructs

- Data Types
- Variables
- Conditional Statements
- Looping Statements
- Array
- Functions
- Class, Object, Methods, Properties
- Inheritance, Polymorphism

Lets Get Started

- Before we begin, download visual studio 2019
- Here is the download link
- https://visualstudio.microsoft.com/downloads/

C# Overview

- C# is general-purpose, object-oriented programming language developed by Microsoft.
- 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.
- Reasons - C# a widely used professional language:
  - It is object oriented & structured language
  - It is component oriented.
  - It is easy to learn & produces efficient programs.
  - It can be compiled on a variety of computer platforms.
  - a part of .Net Framework.

Strong Programming 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

C# Program Structure

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {



        }
    }
}
```

C# - Program.cs (First Program in C#)

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {



        }
    }
}
```

Data Types

- int
- short
- long
- char
- string
- bool
- float
- decimal
- double
- object

```
float f1 = 10.31f ;
decimal d1 = 23.34m ;
string name = “Your Name” ;

int x = 10;
object obj = x; // boxing, implicit
int y = (int) obj ;  //unboxing, explicit
```

C# - Program.cs (First Program in C#)

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("My First Program in C#");
            Console.ReadKey();
         }
    }
}
```

Example

```
static void Main(string[] args)
{
    Console.WriteLine("This is my First Program in C#");
    int n1, n2;
    Console.Write("Enter n1 : ");
    n1 = Convert.ToInt32(Console.ReadLine());
    Console.Write("Enter n2 : ");
    n2 = int.Parse(Console.ReadLine());
    int sum = n1 + n2;
    Console.WriteLine("The sum is : " + sum);
    Console.WriteLine("The sum of " + n1 + " and " + n2 + " " + sum);
    Console.WriteLine("The sum of {0} and {1} is {2}", n1, n2, sum);
    Console.ReadKey();
}
```

Operator

- Arithmetic :
  - +, -, *, /, %, ++, --
- Comparison :
  - >, <, >=, <=, ==, ===, !=
- Logical :
  - && (AND), || (OR), ! (NOT)
- Assignment :
  - =, +=, -=, *=, /=, %=
- Conditional or Ternary - **? :**

Arithmetic Operator

Operator	Description	Example	Result
+	Addition	x=2 y=2 x+y	4
-	Subtraction	x=5 y=2 x-y	3
*	Multiplication	x=5 y=4 x*y	20
/	Division	15/5 5/2	3 2,5
%	Modulus (division remainder)	5%2 10%8 10%2	1 2 0
++	Increment	x=5 x++	x=6
--	Decrement	x=5 x--	x=4

Assignment Operator

Operator	Example	Is The Same As
=	x=y	x=y
+=	x+=y	x=x+y
-=	x-=y	x=x-y
*=	x*=y	x=x*y
/=	x/=y	x=x/y
%=	x%=y	x=x%y

Comparison Operator

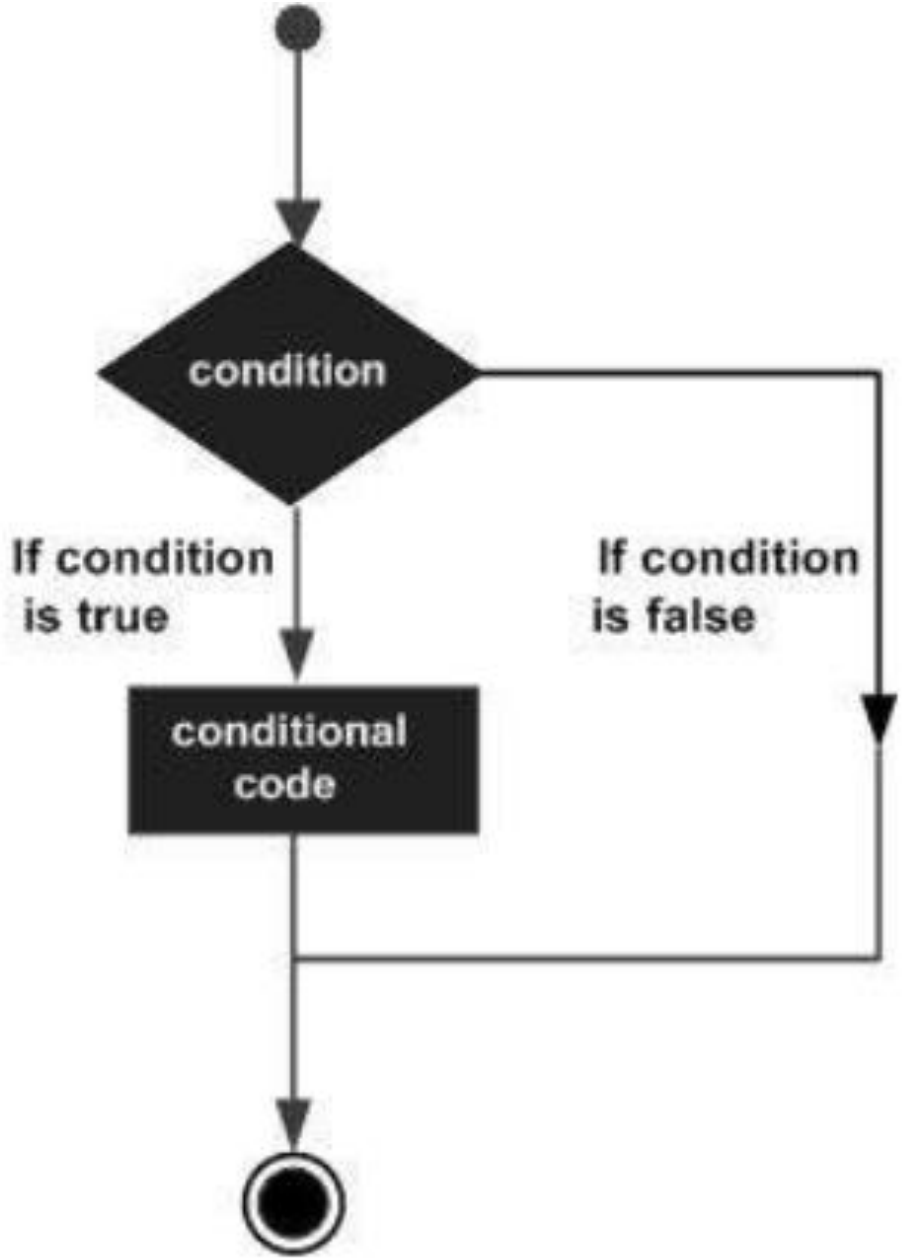
Operator	Description	Example
==	is equal to	5==8 returns false
===	is equal to (checks for both value and type)	x=5 y="5" x==y returns true x===y returns false
!=	is not equal	5!=8 returns true
>	is greater than	5>8 returns false
<	is less than	5<8 returns true
>=	is greater than or equal to	5>=8 returns false
<=	is less than or equal to	5<=8 returns true

Logical Operator

Operator	Description	Example
&&	and	x=6 y=3 (x < 10 && y > 1) returns true
||	or	x=6 y=3 (x==5 || y==5) returns false
!	not	x=6 y=3 !(x==y) returns true

Controlling Program Flow

- Conditions: Making Decisions – 2 Ways
  - if … else statement
  - Switch ....case



Controlling Program Flow

- forms of **if..else** statement
  - if statement
  - if...else statement
  - if...else if... statement.

- Syntax
  - if(expression) {

    Statement(s) to be executed if true

    }

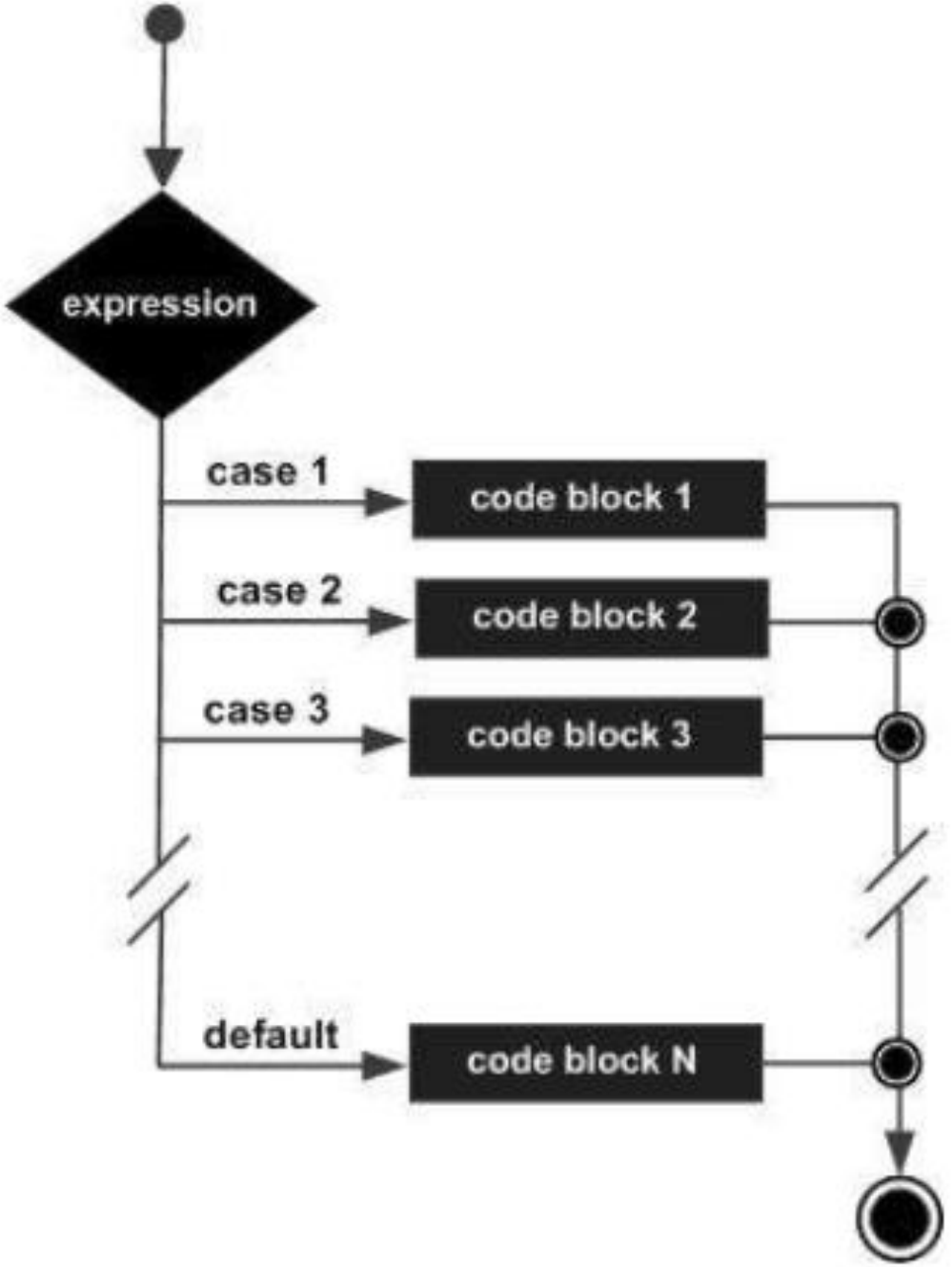
Example
If - else

```
namespace ConsoleApp1
{
    class CondtionalStmt
    {
        static void Main(string[] args)
        {
            Console.Write("Enter for n: ");
            int n = Convert.ToInt32(Console.ReadLine());
            string res = "";
            if (n > 0)
                res = "Greater than 0";
            else if (n < 0)
                res = "Less than 0";
            else
                res = "Equals 0";

            Console.WriteLine(res);
            Console.ReadKey();

        }
    }
}
```

Controlling Program Flow

- switch ....case

```
switch (expression) {
    case condition 1:
     statement(s);
     break;
    case condition 2:
     statement(s);
     break;
    ...
    case condition n:
     statement(s);
     break;
    default: statement(s)
}
```

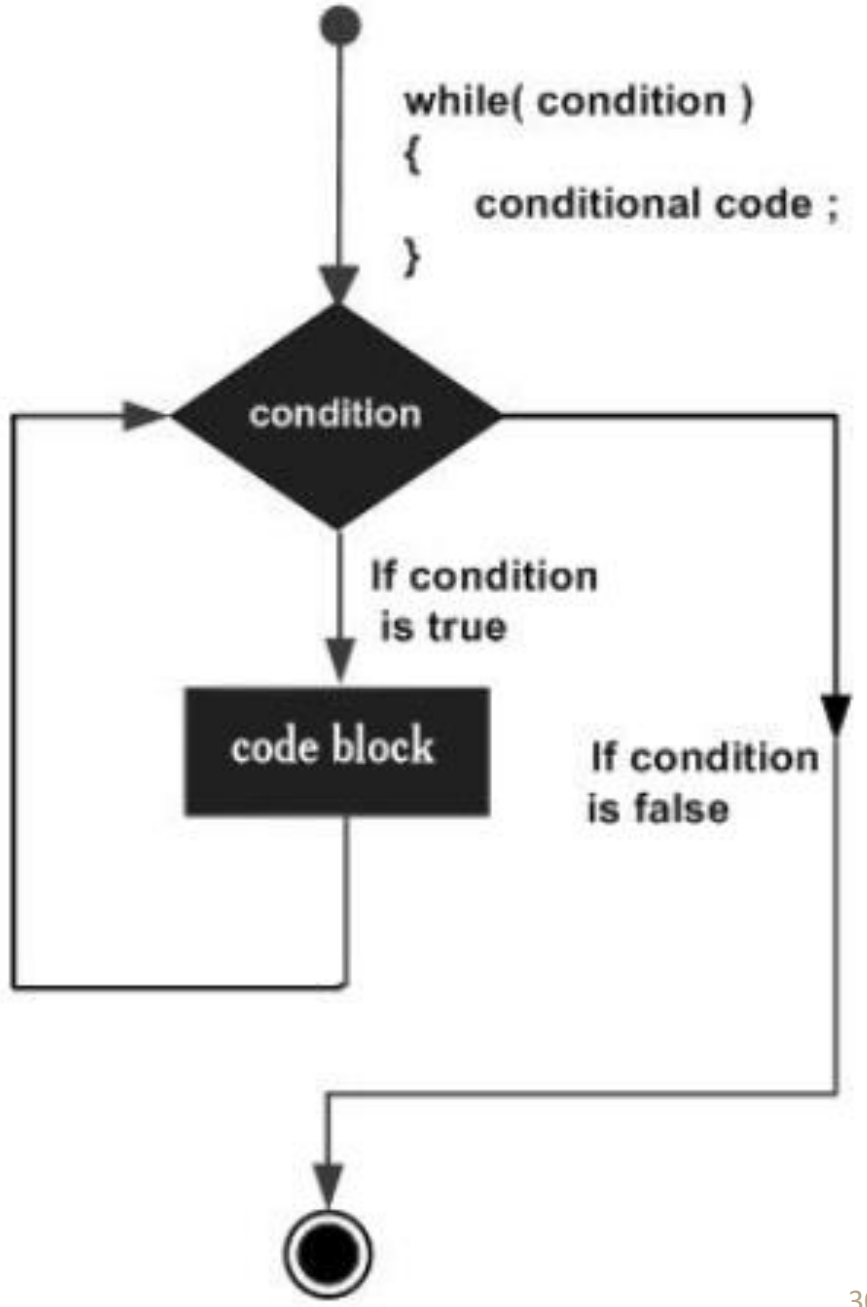


Example switch case

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

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

Loop

- Used to perform an action repeatedly till satisfied condition meets.
- 3 Types of Loops
  - While loop
  - Do While loop
  - For loop
- These loops have
  - Initialization statement
  - Condition statement
  - Update (increment or decrement) statement



While Loop

```
Console.WriteLine("Starting Loop");
int i = 1;
while(i <=10)
{
    Console.WriteLine("Count is : " + i);
    i++;
}
Console.WriteLine("Loop Stopped!");
```

```
Starting Loop
Count is : 1
Count is : 2
Count is : 3
Count is : 4
Count is : 5
Count is : 6
Count is : 7
Count is : 8
Count is : 9
Count is : 10
Loop stopped!
```

do while loop

```
Console.WriteLine("Starting Loop");
int i = 1;
do
{
    Console.WriteLine("Count is : " + i);
    i++;
} while (i <= 10);
Console.WriteLine("Loop Stopped!");
```

Loop – For loop

- Syntax

```
for (initialize; condition; iteration) {
        Statement(s) to be executed if test condition is true
}
```

- Ex

```
for (int i=1; i<=10; i++) {
        Console.WriteLine(”Count is : ”  + i);
}
```

Array & for each loop

```
class SecondProgram
{
    0 references
    public static void Main(string[] args)
    {
        Console.WriteLine("Second Program");
        int[] arr = { 10, 20, 30 };
        for (int i = 0; i < arr.Length; i++)
            Console.WriteLine("arr[{0}] = {1}", i, arr[i]);

        int sum = 0;
        foreach(int j in arr)
        {
            sum += j;
        }
        Console.WriteLine("The sum is : " + sum);
        Console.ReadKey();
    }
}
```

Strings

- Used for storing and manipulating text
- A string variable contains zero or more characters within double quotes.

```
// J a v a s c r i p t
// 0 1 2 3 4 5 6 7 8 9
string s = "Javascript";
Console.WriteLine("Length: " + s.Length);
Console.WriteLine("Index Of: " + s.IndexOf("va"));
Console.WriteLine("Upper Case: " + s.ToUpper());
Console.WriteLine("Lower Case: " +    s.ToLower());
Console.WriteLine("Substring: " + s.Substring(0,4));
Console.ReadKey();
```

Functions

```
class FunctionTest
{
    static void Main(string[] args)
    {
        F1();
        int sq = Square(10);
        Console.WriteLine(sq);
    }

    static void F1()
    {
        Console.WriteLine("This is F1 function");
    }
    static int Square(int n)
    {
        return n * n;
    }
}
```

Function Overloading

```
static void Main(string[] args)
{
    Console.WriteLine("Min : " + Min(5,10));
    Console.WriteLine("Min : " + Min("cow", "hen"));
}
static int Min(int n1, int n2)
{
    if (n1 < n2)
        return n1;
    else
        return n2;
}
static string Min(string s1, string s2)
{
    if (s1.CompareTo(s2) < 0)
        return s1;
    else
        return s2;
}
```

Class & Object

- A *class* consists of data declarations plus functions that act on the data.
  - Normally the data is private
  - The public functions (or methods) determine what clients can do with the data.

- An instance of a class is called an *object*.
  - Objects have identity and lifetime.
  - Like variables of built-in types.

37

Encapsulation

- By default the class definition *encapsulates*, or hides, the data inside it.
- Key concept of object oriented programming.
- The outside world can see and use the data only by calling the build-in functions.
  - Called “methods”

38

Class Members

- Methods and variables declared inside a class are called *members* of that class.
  - Member variables are called *fields*.
  - Member functions are called *methods*.

- In order to be visible outside the class definition, a member must be declared *public*.

39

Objects

- An instance of a class is called an *object*.
- You can create any number of instances of a given class.
  - Each has its own identity and lifetime.
  - Each has its own copy of the fields associated with the class.
- When you call a class method, you call it through a particular object.
  - The method sees the data associated with *that object*.

40

Creating a Class

```
class Point
{
    public int x, y;
    public void Show()
    {
        Console.WriteLine("x : " + x);
        Console.WriteLine("y : " + y);
    }
}
class OOPClassTest
{
    static void Main(string[] args)
    {
        Point p = new Point();
        p.x = 10;
        p.y = 20;
        p.Show();
        Console.ReadKey();
    }
}
```

Class - Properties

```
class Point1
{
    private int x;
    private int y;
    public int X
    {
        get { return x; }
        set { x = value;  }
    }
    public int Y
    {
        get { return y; }
        set { y = value;  }
    }
    public void Show()
    {
        Console.WriteLine("x : " + x);
        Console.WriteLine("y : " + y);
    }
}
```

```
static void Main(string[] args)
{
    Point1 p = new Point1();
    p.X = 10;
    p.Y = 20;
    Console.WriteLine("X : " + p.X);
    Console.WriteLine("Y : " + p.Y);
    Console.ReadKey();
}
```

Class - Constructor

```
class Point2
{
    public int x;
    public int y;
    public Point2() { }  // Default Constructor
    public Point2(int x, int y)  //Parameterized C.
    {
        this.x = x;
        this.y = y;
    }
}
```

```
class OOPConstructorTest
{
    static void Main(string[] args)
    {
        Point2 p1 = new Point2();
        p1.x = 11;
        p1.y = 12;
        Point2 p2 = new Point2(2,5);
    }
}
```

Task

Create a class “Employee” with following specs:

- Field Members : firstName, lastName, salary
- Properties : FirstName, LastName, Salary
- Methods : ShowFullName, IncrementSalary(double s)
- Constructor : Employee(__ , ___ , ___)

Now, create object of Employee(“Ram”, “Bahadur”, 20000)

- Show Employee Fullname & Salary
- Change FirstName to “Hari” & increment salary by 5000
- Show full name & salary

Inheritance

- New Classes called derived classes are created from existing classes called base classes

```
public class Class A
 {


 }
public class Class B : A
 {

 }
```

Inheritance Example

```
public class ParentClass
{
        public ParentClass() {
                Console.WriteLine("Parent Constructor");
        }
        public void Print() {
                Console.WriteLine("I'm a Parent Class.");
        }
}
public class ChildClass : ParentClass {
        public ChildClass (){
                Console.WriteLine("Parent Constructor");
        }
}
```

Inheritance Example

```
class Program
{
    static void Main(string[] args)
    {
                        ChildClass cc= new ChildClass();
                cc.Print();
     }
}
```

Use base key word

```
public class ParentClass
{
        public int x = 10;
        public ParentClass()
        {
                Console.WriteLine(“Parent Constructor”);
        }
        public void Print() {
                Console.WriteLine(“I’m a Parent Class.”);
        }
}
```

Use base key word

```
public class ChildClass : ParentClass
{
        public ChildClass() : base()
    {
        Console.WriteLine(“Child Constructor”);
        base.Print();
        Console.WriteLine(base.x);
    }
}
```

Inheritance Example

```
class Program
{
    static void Main(string[] args)
    {
                ChildClass cc= new ChildClass()
                cc.Print();
                Console.ReadKey();
     }
}
```

Indexer

- An **indexer** allows an object to be indexed such as an array.
- When you define an indexer for a class, this class behaves similar to a **virtual array**.
- You can then access the instance of this class using the array access operator ([ ]).

```
element-type this[int index]
{
   get    {   // return the value specified by index   }
   set    {   // set the value specified by index   }
}
```

Indexer Example

```
class Student
{
    private int roll;
    public int Roll
    {
        get { return roll; }
        set { roll = value; }
    }
    private int[] marks = new int[3];
    public int this[int index]
    {
        get { return marks[index]; }
        set { marks[index] = value; }
    }
    public double GetPercent()
    {
        double total = 0.0;
        foreach (int m in marks)
        {
            total = total + m;
        }
        return total / marks.Length;
    }
}
```

Indexer Example

```
static void Main()
{
    Student s1 = new Student();
    s1.Roll = 1;
    s1[0] = 50;
    s1[1] = 25;
    s1[2] = 30;
    Console.WriteLine(s1.GetPercent());
    Student s2 = new Student();
    s2.Roll = 2;
    s2[0] = 20;
    s2[1] = 30;
    s2[2] = 40;
    Console.WriteLine(s2.GetPercent());
}
```

The sealed class

- Sealed classes are used to restrict the inheritance feature of object oriented programming. Once a class is defined as a sealed class, the class cannot be inherited.
- In C#, the sealed modifier is used to define a class as sealed

```
sealed class SealedClass
{

}
```

Abstract Class

- Classes can be declared as abstract by using keyword abstract.
- Abstract classes are one of the essential behaviors provided by .NET.
- If you like to make classes that only represent base classes, and don’t want anyone to create objects of these class types, use abstract class to implement such functionality.
- Object of this class can be instantiated, but can make derivations of this.
- The derived class should implement the abstract class members.

Abstract Class

```
abstract class AbsClass
{
    public abstract void AbstractMethod();
    public void NonAbstractMethod()
    {
        Console.WriteLine("NonAbstract Method");
    }
}

class Derived : AbsClass
{
    public override void AbstractMethod()
    {
        Console.WriteLine("Overriding AbstractMethod in Derived Class");
    }
}
```

Abstract Class

```
class OOPAbstractClass
{
    static void Main(string[] args)
    {
        Derived d = new Derived();
        d.NonAbstractMethod();
        d.AbstractMethod();
        Console.ReadKey();
    }
}
```

Interface

- An interface is not a class. It is an entity that is defined by the keyword Interface.
- By Convention, Interface Name starts with letter ‘I’
- has no implementation; just the declaration of the methods without the body.
- a class can implement more than one interface but can only inherit from one class.
- interfaces are used to implement multiple inheritance.

```
interface IFace
{

}
```

Partial Classes

- In C#, a class definition can be divided over multiple files.
  - Helpful for large classes with many methods.
  - Used by Microsoft in some cases to separate automatically generated code from user written code.

- If class defintion is divided over multiple files, each part is declared as a *partial* class.

Partial Classes

In file circ1.cs

```
partial class Circle
{
  // Part of class defintion
  ...
}
```

In file circ2.cs

```
partial class Circle
{
  // Another part of class definition
  ...
}
```

Exception Handling

- An exception is a problem that arises during the execution of a program.
- A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero, Array Index Out of Bounds, etc
- Exceptions provide a way to transfer control from one part of a program to another.
- C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.

Exception Handling

- **try**: A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.
- **catch**: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **finally**: The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown.
- **throw**: A program throws an exception when a problem shows up. This is done using a throw keyword.

Exception Handling

Syntax

```
try
{
    // statements causing exception
}
catch( ExceptionName e1 )
{
    // error handling code
}
catch( ExceptionName e2 )
{
    // error handling code
}
catch( ExceptionName eN )
{
    // error handling code
}
finally
{
    // statements to be executed
}
```

Exception Handling

- What will happen to this program?
- In which line, we encounter the error?
- Will this execute all statements?
- Can this program display the last 2 lines?

1. int a = 10;
2. int b = 0;
3. int c = a / b;
4. Console.WriteLine(c);
5. Console.WriteLine(“This is last line”);

Exception Handling

```
try
{
    int a = 10;
    int b = 0;       // assign b = 2
    int c = a / b;
    Console.WriteLine(c);
    int[] arr = {10, 20, 12};
    Console.WriteLine(arr[5]);
}
catch (DivideByZeroException e1)
{
    Console.WriteLine(e1.ToString());
}
catch (IndexOutOfRangeException e2)
{
    Console.WriteLine("Array index problem");
}
```

Delegate

- C# delegates are similar to pointers to functions, in C or C++.
- A **delegate** is a reference type variable that holds the reference to a method. The reference can be changed at runtime.
- Delegates are especially used for implementing events and the call-back methods.
- Syntax – Delegate Declaration :

delegate <return-type> <delegate_name> <params>

Delegate

- Delegate Declaration :

delegate <return-type> DelegateName> <arg_list>

- Object Creation :

DelegateName d = new DelegateName<function to which the delegate points>

Invoking :

d<list of args that are to be passed to the functions>

Delegate - Ex

```
// 1. Declaration
public delegate void SimpleDelegate();
class DelegateTest
{
    static void Main(string[] args)
    {
        // 2. Instantiation
        SimpleDelegate d = new SimpleDelegate(MyFunc);
        d();          // 3. Invocation
    }
    public static void MyFunc()
    {
        Console.WriteLine("I was called by delegate");
    }
}
```

Collection Types

- Collection Types are specialized classes for data storage and retrieval.
- These classes provide support for stacks, queues, lists, and hash tables.
- Collection classes serve various purposes, such as allocating memory dynamically to elements and accessing a list of items on the basis of an index etc.
- Namespaces:
  - System.Collection
  - System.Collection.Generic

Collection Types

- System.Collection
  - ArrayList, Hashtable, SortedList, Stack, Queue
- System.Collection.Generic
  - generic collection is strongly typed (type safe), that you can only put one type of object into it.
  - This eliminates type mismatches at runtime.
  - Another benefit of type safety is that performance is better
  - Ex: List, Dictionary

Array List – System.Collections

```
ArrayList al = new ArrayList();
al.Add(1);
al.Add("Hari");
al.Add(3.4);
Console.WriteLine(al.Count);

al.Remove(3.4);
al.RemoveAt(0);
Console.WriteLine(al.Count);
```

List – System.Collection.Generic

```
List<string> names = new List<string>();
names.Add("Ram");
names.Add("Hari");
names.Add("Sam");
Console.WriteLine(names.IndexOf("Ram"));

Console.WriteLine(names.Count);
names.RemoveAt(2);
names.Remove("Ram");
foreach(string n in names)
{
    Console.WriteLine(n);
}
```