# .NET FULLSTACK Foundation Course(Coursera)

27 October 2024 12:11

## .NET FRAMEWORK vs .NET CORE

| Based On | .NET Core | .NET Framework |
|---|---|---|
| Open source | .NET Core is an open source. | Certain components are open source. |
| Cross-Platform | Compatible with Windows, Linux, and Mac OS | compatible with the windows OS. |
| Application Models | does not support desktop applications | Supports desktop and web applications |
| Performance | offers high performance and scalability. | less effective in comparison to .NET |
| Security | Does not have features like Code Access Security. | Code access security feature is present |

.NET CLI and .NET CORE-

## INTRODUCTION TO .NET CLI

- .NET CLI is a new cross-platform tool.
- CLI tool is used for creating, restoring packages, building, and publishing .NET applications.
- It supports installation of packages.
- It supports various commands that can be used to create, build and run .NET Core projects.
- It can also be used to manage dependencies, including adding, removing and updating packages.
- It can be easily automated and integrated into build and deployment pipelines.

# .NET CLI COMMAND STRUCTURE

```
dotnet <command> <argument> <option>
```

- All the commands start with driver named dotnet.
- The driver starts the execution of the specified command.
- After dotnet, we can supply command (also known as verb) to perform a specific action.
- Each command can be followed by arguments and options.

# .NET CLI COMMANDS

Type ***dotnet –help*** will list all the commands the tool is offering:

| Command | Description |
|---|---|
| **new** | Initialize .NET projects. |
| **restore** | Restore dependencies specified in the .NET project. |
| **build** | Builds a .NET project. |
| **publish** | Publishes a .NET project for deployment (including the runtime). |
| **run** | Compiles and immediately executes a .NET project. |
| **test** | Runs unit tests using the test runner specified in the project. |
| **pack** | Creates a NuGet package. |
| **migrate** | Migrates a project.json based project to a msbuild based project |
| **clean** | Clean build output(s). |
| **sln** | Modify solution (SLN) files. |

# .NET CLI COMMANDS

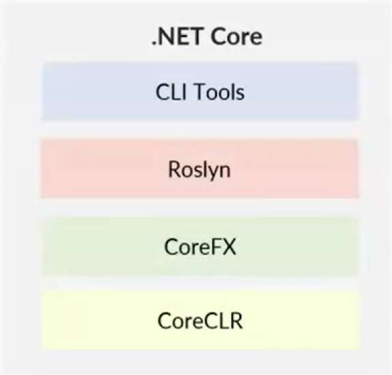
| Project Modification Commands | |
|---|---|
| **Command** | **Description** |
| add | Add items to the project |
| remove | Remove items from the project |
| list | List items in the project |

| Advanced Commands | |
|---|---|
| **Command** | **Description** |
| nuget | Provides additional NuGet commands. |
| msbuild | Runs Microsoft Build Engine (MSBuild). |
| vstest | Runs Microsoft Test Execution Command Line Tool. |

# .NET CORE PLATFORM COMPOSITION

**The .NET Core Framework composed of the following parts:**

- **CLI Tools:** A set of tooling for development and deployment.
- **Roslyn:** Language Compiler for C# and Visual Basic.
- **CoreFX:** Set of framework libraries.
- **CoreCLR:** A JIT based CLR (Command Language Runtime).




# .NET CORE LANGUAGE COMPILERS

- The Compilers included in .NET Core are responsible for translating code.
- Translated codes written in C#, F#, and Visual Basic into Intermediate Language (IL) code.
- IL is a low-level language that can be executed by the .NET runtime.
- Roslyn language compiler is used for C# and Visual Basic.
- F# Compiler (fsc.exe) is used to compile F# code.

# .NET CORE CLR

- CoreCLR is the .NET runtime (i.e., the execution engine) used in .NET Core.
- It is responsible for compiling and executing .NET Core applications.
- Provides features like garbage collection, just-in-time (JIT) compilation, and exception handling.
- One of the benefits of CoreCLR is its performance.
- It is optimized for modern hardware.
- It is lightweight and modular, it uses less resources than the full .NET Framework.

## .NET CoreFX

- CoreFX is foundational set of libraries that provides functionality to .NET Core applications.
- It provides a wide range of libraries.
- Those libraries provides functionality such as file I/O, networking, and collections.
- It is modular so that developers can choose the libraries that are needed for the application.

INTRODUCTION TO C#

## INTRODUCTION TO C#

- An object-oriented programming language developed by Microsoft.
- It basically runs on .NET Framework.
- C# is approved as a standard by ECMA and ISO.
- C# is designed for CLI (Common Language Infrastructure).
- It has a huge community support.
- C# is used to develop web apps, desktop apps, mobile apps, games and much more.

## FEATURES OF C#

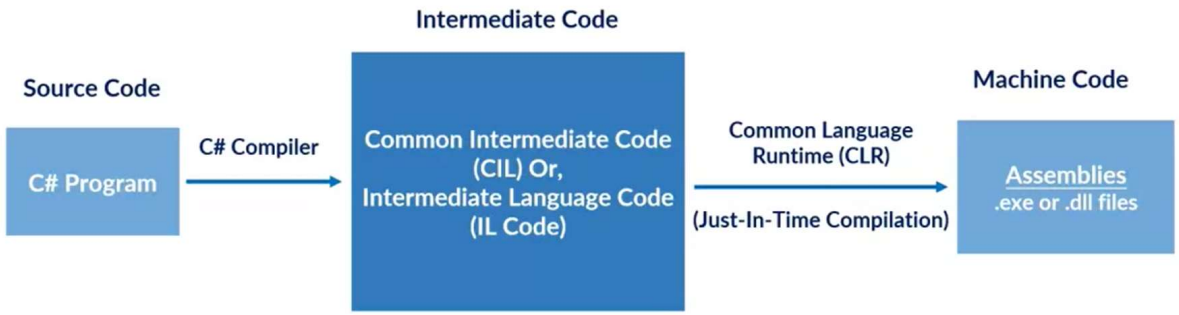
# HISTORY OF C# VERSIONS

- C# was first introduced with .NET Framework 1.0 in year 2002 and evolved much since then.
- Each new version of C# has built on the previous version,
- and added new features and improvements to the language.
- Developers can choose the version of C# to use based on the requirements of their application and the features that they need.

# HISTORY OF C# VERSIONS

| Version | Date | .NET | Visual Studio |
|---|---|---|---|
| C# 1.0 | January 2002 | .NET Framework 1.0 | Visual Studio .NET 2002 |
| C# 2.0 | November 2005 | .NET Framework 2.0 | Visual Studio 2005 |
| C# 3.0 | November 2007 | .NET Framework 3.0\3.5 | Visual Studio 2008 |
| C# 4.0 | April 2010 | .NET Framework 4.0 | Visual Studio 2008 |
| C# 5.0 | August 2012 | .NET Framework 4.5 | Visual Studio 2012/2013 |
| C# 6.0 | July 2015 | .NET Framework 4.6 | Visual Studio 2013/2015 |
| C# 7.0 | November 2017 | .NET Core 2.0 | Visual Studio 2017 |
| C# 8.0 | September 2019 | .NET Core 3.0 | Visual Studio 2019 |
| C# 9.0 | November 2020 | .NET 5.0 | Visual Studio 2019 |
| C# 10.0 | November 2021 | .NET 6.0 | Visual Studio 2022 |
| C# 11.0 | November 2022 | .NET 7 | Visual Studio 2022 |

# C# CODE EXECUTION

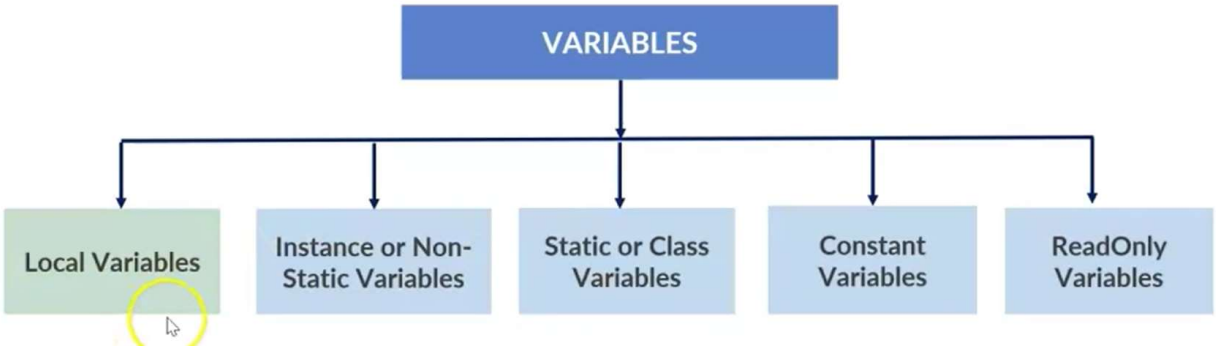


# VARIABLES

- A variable is a named storage location in computer memory that holds a value
- Variables are used to store and manipulate data in a program.
- In C#, all the variables must be declared before they can be used.
- It is the basic unit of storage in a program.
- The value stored in a variable can be changed during program execution.
- **Syntax:**

```
<data type> <variable name> = <value>;
```

# TYPES OF VARIABLES

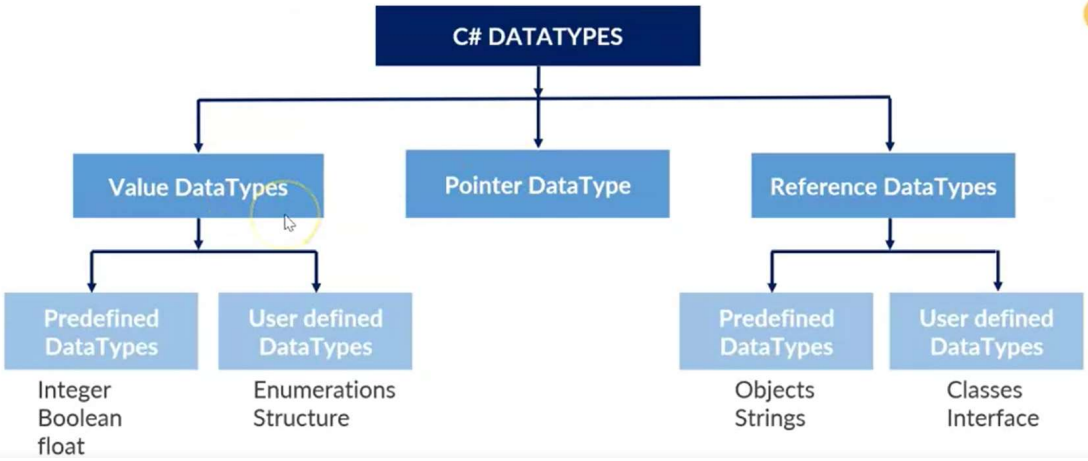



# DATA TYPES

- Data types specify the type of data that a valid C# variable can hold.
- C# is a strongly-typed language.
- It means we must declare the type of a variable that indicates the kind of values it is going to store, such as integer, float, decimal, text, etc.
- The following declares and initialized variables of different data types.

```
string stringVar = "Hello World!!";
int intVar = 100;
float floatVar = 10.2f;
char charVar = 'A';
bool boolVar = true;
```

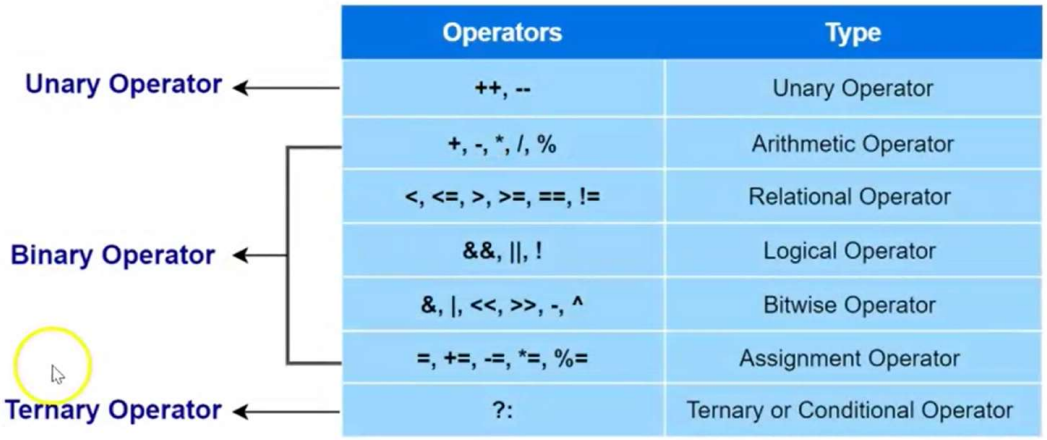
# DATA TYPES




# OPERATORS

- Operators are symbols that are used to perform operations on operands.
- Operands may be variables and/or constants.
- Operators are used to manipulate variables and values in a program.
- C# supports a number of operators that are classified based on type of operations they perform.

# TYPES OF OPERATORS

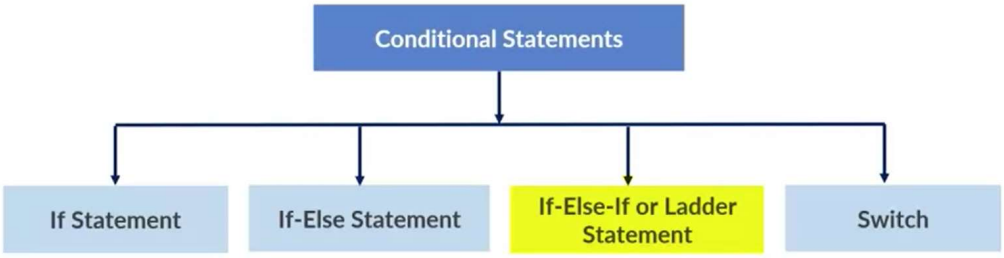



| Operators | Type |
|---|---|
| ++, -- | Unary Operator |
| +, -, *, /, % | Arithmetic Operator |
| <, <=, >, >=, ==, != | Relational Operator |
| &&, \|\|, ! | Logical Operator |
| &, \|, <<, >>, -, ^ | Bitwise Operator |
| =, +=, -=, *=, %= | Assignment Operator |
| ?: | Ternary or Conditional Operator |

# CONDITIONAL STATEMENTS

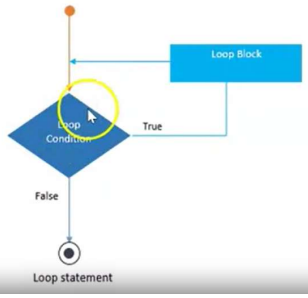
- Conditional statements are based on certain conditions and generate decisions accordingly.
- These statements are a bunch of codes that can be executed by "decisions statements".
- These conditions have some specific "boolean expressions".
- The boolean expression of these statements generates "Boolean Value" which could be either true or false.

# TYPES OF CONDITIONAL STATEMENTS

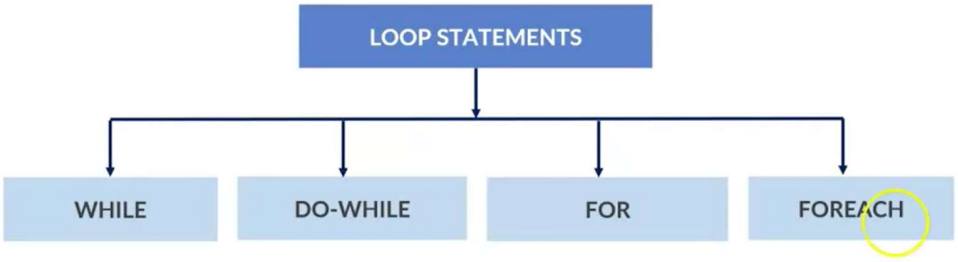



# LOOP STATEMENTS

- Loops are used to execute a block of code repeatedly until a certain condition is met.
- Used to repeat a block of statements for certain times.
- A loop statement continue its execution until the specified expression evaluates to false.




# TYPES OF LOOP STATEMENTS




FOR

```
string[] names = new string[5]{"King Kochhar", "Sarah Bowling", "John Smith", "Roger Lee", "James Lee"};

for (int i = 0; i < 5 ; i++)
   Console.WriteLine(names[i]);
```

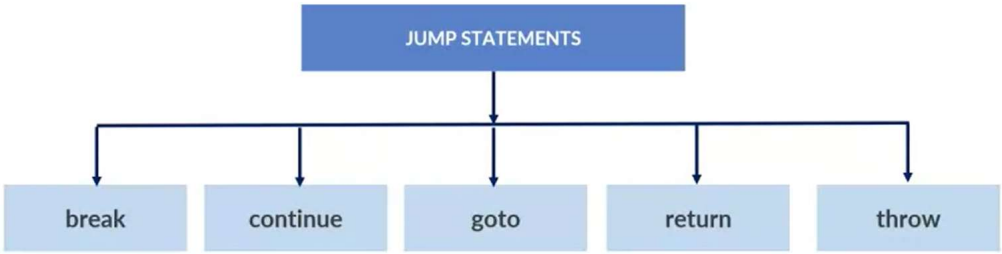
WHILE

WHILE

```
i = 0;
while (i < names.Length)
{
    Console.WriteLine(names[i]);
    i++;
}
```

## JUMP STATEMENTS

- Jump Statements are keywords that allows you to control the flow of execution in a program.
- These are used to transfer program control from one point to another point in the program.
- **There are five keywords in Jump Statements:**

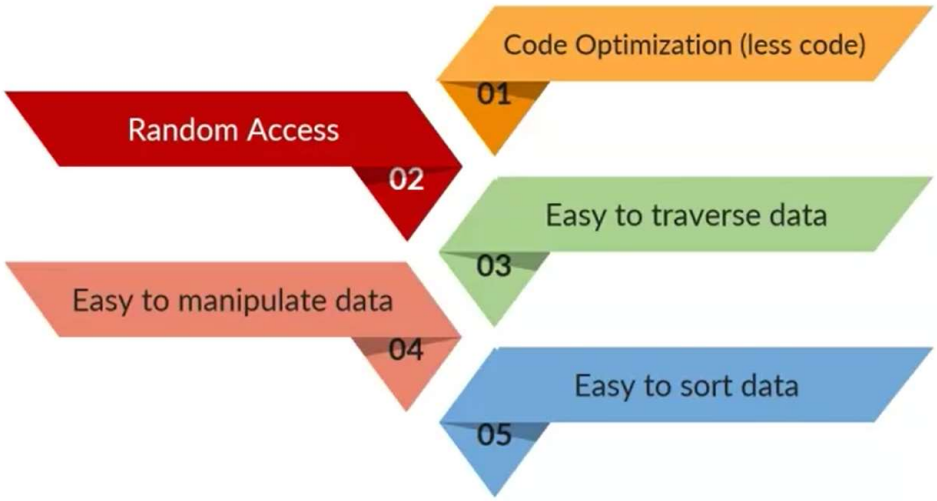



## ARRAYS

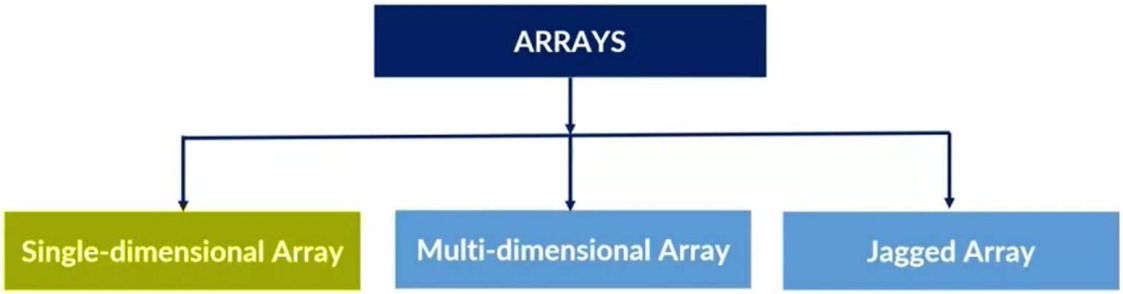
- A collection of elements of same data type that are stored in contiguous memory locations.
- Each element in an array is identified by its index or position within the array, starting from 0.
- Arrays are declared using square brackets "[]" after the type name, followed by the array name.
- **For example,** to declare and initialize an array named myArray that holds 5 integers, you can use the following code:

```
int[] myArray  = new int[5];
```

# ADVANTAGES OF ARRAYS




# ARRAYS




# ONE-DIMENSIONAL ARRAY

- It is the simplest type of array that contains only one row for storing data.
- It has single set of square bracket ("[]").
- To declare single dimensional array in C#, you can write the following code.
- **For example**, to declare an array of integers named "age" that can hold 5 elements in a single row, you would write:

```
// declare an array
int[] age;

// allocate memory for array
age = new int[5];
```

# MULTI-DIMENSIONAL ARRAY

- This Array contains more than one row to store data on it.
- Also known as rectangular array because it has the same length of each row.
- It can be two-dimensional array or three-dimensional array or more.
- Contains more than one comma (,) within single rectangular brackets ("[ , , ,]").
- To storing and accessing the elements from a multidimensional array, you need to use a nested loop in the program.
- ***Example:***

```
int[,] s = new int [3, 3];
```

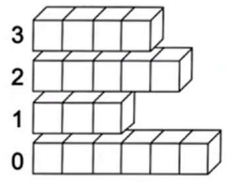
int s[ ] [ ]= new int [3][3];

| S[0][0] | S[0][1] | S[0][2] |
|---|---|---|
| S[1][0] | S[1][1] | S[1][2] |
| S[2][0] | S[2][1] | S[2][2] |

# JAGGED ARRAY

- A jagged array is an array of arrays, where each sub-array can have a different length.
- Jagged arrays are useful when you need to store a collection of arrays of different sizes.
- **For example**, to declare and initialize a jagged array named myArray that contains 3 sub-arrays of integers with different lengths, you can use the following code:

```
int[][] myArray = new int[3][];
myArray[0] = new int[2] { 1, 2 };
myArray[1] = new int[3] { 3, 4, 5 };
myArray[2] = new int[4] { 6, 7, 8, 9 };
```




Jagged Array
int [4] []

SINGLE DIMENTIONAL ARRAY-

```
int[] marks = new int[5] { 45, 56, 65, 67, 78 };
/*marks[0] = 25;
marks[1] = 50;
marks[2] = 55;
marks[3] = 60;
marks[4] = 40;*/

foreach (int mark in marks)
    Console.WriteLine(mark);
```

```
for (int i = 0; i < marks.Length; i++)
    Console.WriteLine(marks[0]);
```

Multi DIMENTIONAL ARRAY-

```
int[,] multiArray = new int[3, 4]
{ { 1, 2, 3, 4 }, { 1, 2, 3, 4 }, { 1, 2, 3, 4 } };

for (int i = 0;i <= 2;i++)
{
  for (int j = 0; j <= 3; j++)
  {
    Console.Write(multiArray[i,j] + "\t");
  }
  Console.WriteLine()
}
```

(local variable) int[,] multiArray
'multiArray' is not null here.

JAGGED ARRAY

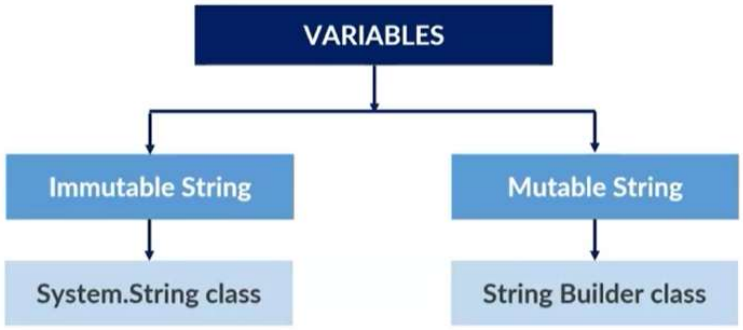
```
int[][] jaggedArray = new int[2][];
jaggedArray[0] = new int[2];
jaggedArray[1] = new int[3];
```

# STRINGS

- String is an object of System.String class that represent sequence of characters.
- For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and ‘o’.
- We can perform many operations on strings such as concatenation, comparison, getting substring, search, trim, replacement etc.
- In C#, string is keyword which is an alias for System.String class.
- That is why string and String are equivalent. We are free to use any naming convention.

```
string s1 = "hello";//creating string using string keyword
String s2 = "welcome";//creating string using String class
```

# TYPES OF STRINGS




Mutable Strings are Modifiable while Immutable Strings can't be modified.

# STRING METHODS

| Functions | Description |
|---|---|
| Clone() | Make clone of strings. |
| CompareTo() | Compare two strings and returns integer value as output. |
| Contains() | It checks whether specified character exists or not in the string value. |
| EndsWith() | Checks if the string ends with the given string |
| Equals() | Compares two strings and returns boolean value as output. |
| ToUpper() | Converts the string to uppercase |
| ToLower() | Converts the string to lowercase |
| Insert() | Insert a string or character in the string at the specified position. |
| IndexOf() | Returns the index position of first occurrence of specified character. |

```
string str1 = "Hello World";
string str2 = "C# Programming";
Console.WriteLine(str1);
Console.WriteLine(str1.Length);
string str3 = string.Concat(str1, str2);
Console.WriteLine(str3);
Console.WriteLine(str1.Equals(str2));
```

```
// Immutable String

string s1 = "C# Programming";
string s2 = "Java Programming";

// Mutable String

StringBuilder stringBuilder = new StringBuilder();
stringBuilder.Append("C# Programming");
stringBuilder.Append("Java Programming");
```

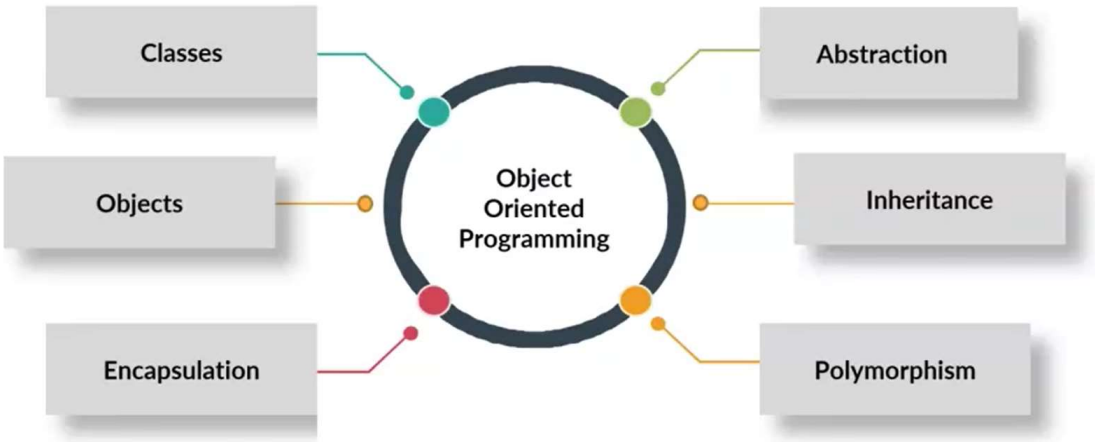
```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.Append("C# Programming");
stringBuilder.Append("Java Programming");
```

We can perform many operations on strings such as concatenation, comparision, getting substring, search, trim, replacement etc.

## WHAT IS OBJECT-ORIENTED PROGRAMMING?

- OOPs stands for Object-Oriented Programming (OOP) concepts.
- C# is an object-oriented programming language that supports the OOP paradigm.
- Object Oriented Concepts provides a clear modular structure of programs.
- This makes easy to maintain the existing code.
- Codes can be reused without any redundancy.
- The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

## OBJECT-ORIENTED CONCEPTS



# CLASSES

- A class is a blueprint or a template for creating objects.
- It defines the properties and behavior of an object.
- A class can have fields, properties, methods, and events,
- They collectively define the data and behavior of an object.
- In object creating, class gets its own set of data and behavior based on properties and methods defined in the class.
- **Syntax:**

```
AccessSpecifier class NameOfClass
{
    // Member variables
    // Member functions
}
```

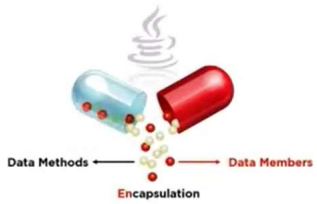
# OBJECTS

- An object is a dynamically created instance of the class.
- It is created at runtime so it can also be called a runtime entity.
- All the members of the class can be accessed using the object.
- The object definition starts with the class name followed by the object name.
- Then the new operator is used to create the object.
- **Syntax:**

```
NameOfClass NameOfObject = new NameOfClass();
```

# ENCAPSULATION

- Encapsulation is defined as the wrapping up of data under a single unit.
- It is the mechanism that binds together code and the data it manipulates.
- In a different way, encapsulation is a protective shield that prevents the data from being accessed by the code outside this shield.
- In encapsulation, data in a class is hidden from other classes, so it is also known as data-hiding.
- **Encapsulation can be achieved by:** Declaring all the variables in the class as private.



# ABSTRACTION

- Data Abstraction is the property by virtue of which only essential details are exhibited to user.
- Abstraction can be achieved with either abstract classes or interfaces.
- **The abstract keyword is used for classes and methods:**
  - **Abstract class:**

    - It is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
  - **Abstract method:**

    - It can only be used in an abstract class, and it does not have a body.

    - The body is provided by the derived class (inherited from).

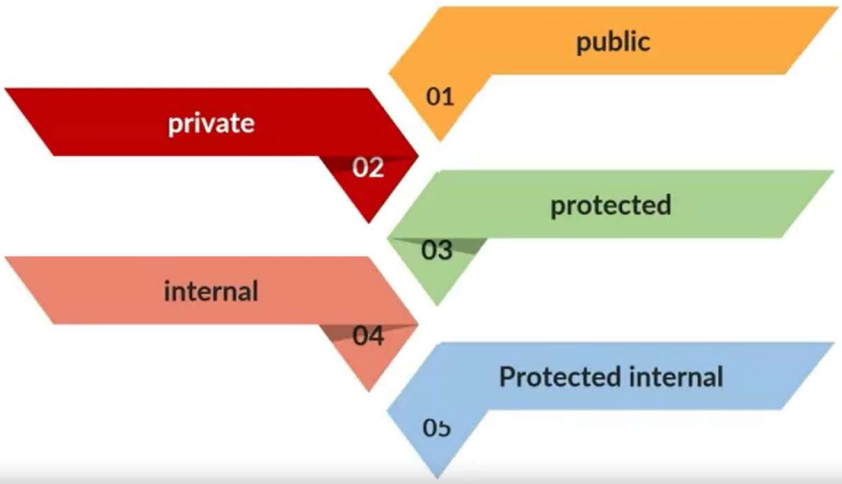
# ABSTRACTION

- **For Example:** Consider a real-life scenario of withdrawing money from ATM.
- The user only knows that in ATM machine first enter ATM card,
- then enter the pin code of ATM card,
- and then enter amount which he/she wants to withdraw and at last, he/she gets their money.
- The user does not know about the inner mechanism of the ATM of withdrawing money etc.

# ACCESS MODIFIERS

- Access modifiers or specifiers are the keywords.
- They are used to specify accessibility or scope of variables and functions in the C# application.
- We can choose any of these to protect our data.
- Public is not restricted and Private is most restricted.

# TYPES OF ACCESS MODIFIERS




# ACCESS MODIFIERS

| Access Specifier | Description |
|---|---|
| **Public** | It specifies that access is not restricted. |
| **Protected** | It specifies that access is limited to the containing class or in derived class. |
| **Internal** | It specifies that access is limited to the current assembly. |
| **Protected internal** | It specifies that access is limited to the current assembly or types derived from the containing class. |
| **Private** | It specifies that access is limited to the containing type. |

# CONSTRUCTORS

- A constructor is a special method that is used to initialize an object of a class
- It is similar to a method that is invoked when an object of the class is created.
- **However, unlike methods, a constructor:**
  - has the same name as that of the class
  - does not have any return type

# TYPES OF CONSTRUCTORS

| Constructor | Description |
|---|---|
| **Default** | A constructor with no parameters is called a default constructor. |
| **Parameterized** | This constructor can also accept parameters. |
| **Copy** | We use a copy constructor to create an object by copying data from another object. |
| **Private** | Once constructor is private, we cannot create objects of class in other classes. |
| **Static** | This constructor is initialized static fields or data of the class and to be executed only once. |

```
// Default Constructor
1 reference
public Student()
{
    studentId = 101;
    studentName = "Annonymous";
}

// Parameterized Constructor
1 reference
public Student(int sId, string sName)
{
    studentId = sId;
    studentName = sName;
}
```

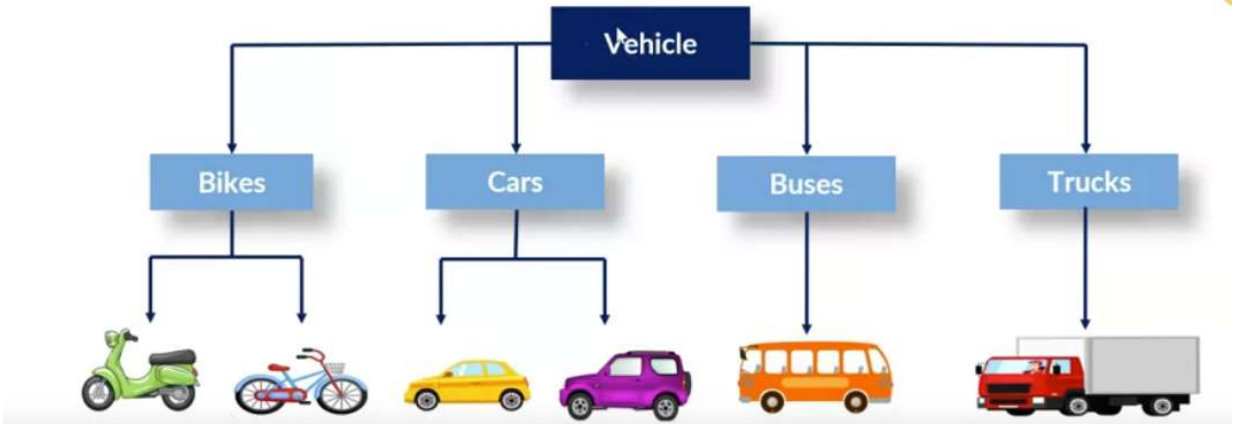
# CONSTRUCTOR OVERLOADING

- It allows you to define multiple constructors for a class, each with a different set of parameters.
- This allows you to create objects of the class with different initial states, depending on the arguments passed to the constructor.
- It also allows you to make your code more flexible and reusable.

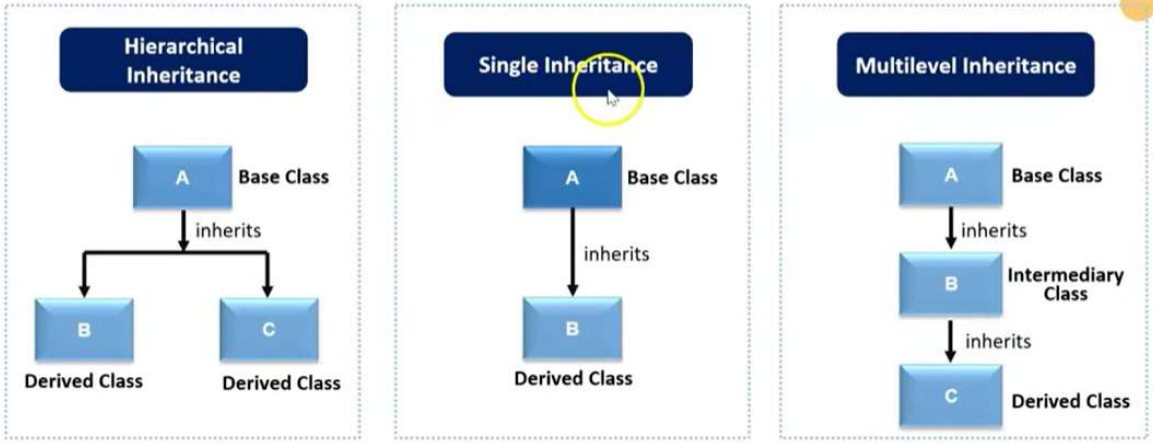
# INHERITANCE

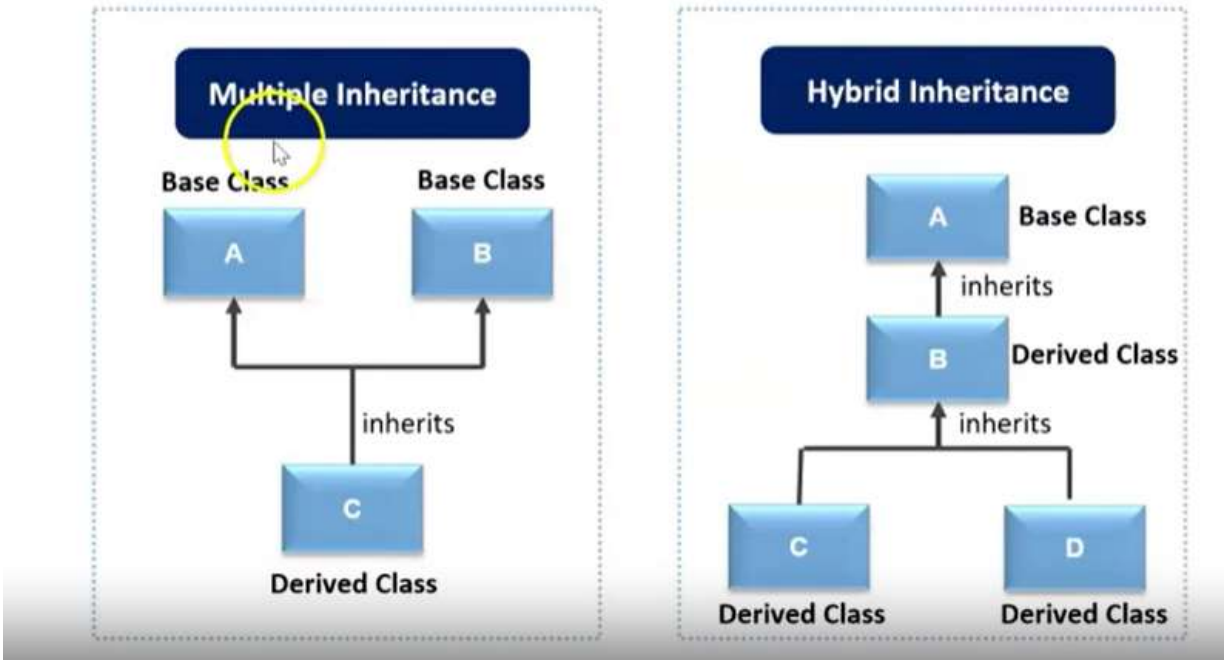
- In C#, it is possible to inherit fields and methods from one class to another.
- It allows us to define a new class based on an existing class.
- The new class inherits the properties and methods of the existing class and can also add new properties and methods of its own.
- It promotes code reuse, simplifies code maintenance, and improves code organization.

# INHERITANCE




# TYPES OF INHERITANCE





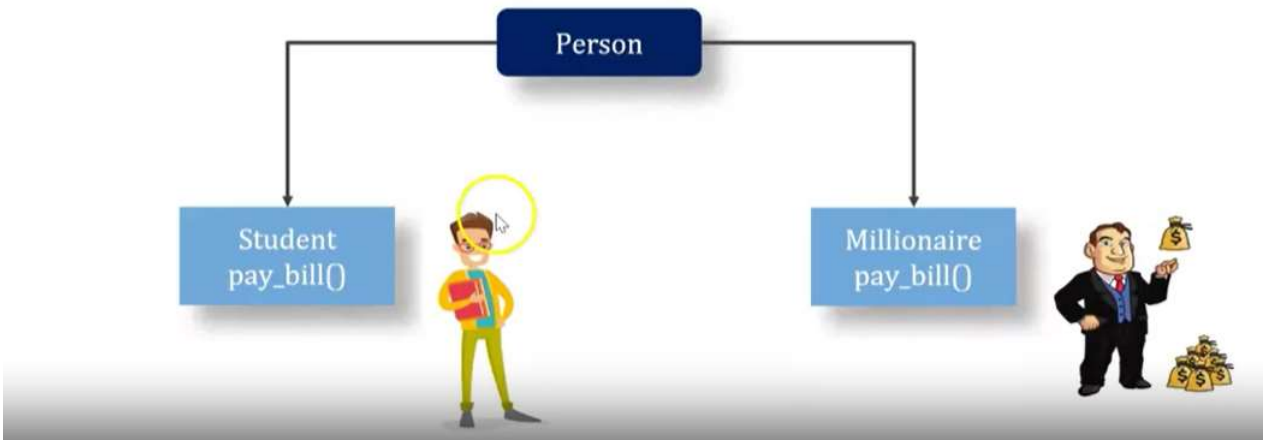

ADVANCESD CONSEPTS IN C#

## POLYMORPHISM

- Polymorphism is a Greek word that means multiple forms or shapes.
- You can use polymorphism if you want to have multiple forms of one or more methods of a class with the same name.
- **In C#, Polymorphism can be achieved in two ways:**
  - Compile-time Polymorphism / Static Polymorphism
  - Runtime Polymorphism / Dynamic Polymorphism

## POLYMORPHISM

**When one task is performed by different ways, then it is called Polymorphism.**

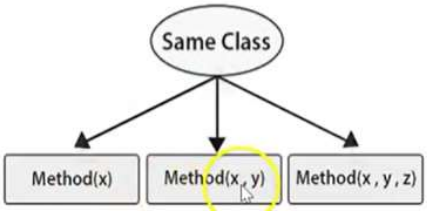


# COMPILE-TIME POLYMORPHISM

- In this, the compiler identifies which method is being called at the compile time.
- **In C#, Compile-time Polymorphism can be achieved in two ways:**
  - Method Overloading
  - Constructor Overloading

# METHOD OVERLOADING

- **In a C# class, we can create methods with the same name in a class if they have:**
  - different numbers of parameter
  - types of parameter
- Method overloading is also known as early binding or static binding,
- because which method to call is decided at compile time, early than the runtime.
- In C#, we can overload method, constructors and indexed properties.
- It is because these members have parameters only.




- The following example demonstrates the method overloading by defining multiple ***Print()*** methods with a different number of parameters of the same type.

```
class ConsolePrinter
{
    public void Print(string str){
        Console.WriteLine(str);
    }
    public void Print(string str1, string str2){
        Console.WriteLine($"{str1}, {str2}");
    }
    public void Print(string str1, string str2, string str3){
        Console.WriteLine($"{str1}, {str2}, {str3}");
    }
}
```

# METHOD OVERRIDING

- In Method Overriding, Derived class defines same method as defined in its base class.
- It is used to achieve runtime polymorphism.
- Enables you to provide implementation of method which is already provided by its base class.
- You need to use ***virtual*** keyword with base class method and ***override*** keyword with derived class method.

- The ***Animal*** class has a method called ***MakeSound***, which is marked as ***virtual***.
- This means that subclasses are allowed to override this method.
- The Dog class overrides the MakeSound method and provides a different implementation.
- When MakeSound method is called on a Dog object, it will print "The dog barks".

```
class Animal
{
    public virtual void MakeSound()
    {
        Console.WriteLine("The animal makes a sound");
    }
}
class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("The dog barks");
    }
}
```

# ABSTRACT CLASS

- An abstract class is a class that cannot be instantiated.
- Instead, it serves as a base class for other classes to inherit from.
- They are used to define a common set of properties that derived classes should have.
- "***abstract***" keyword is used to create an abstract class.

```
// create an abstract class
abstract class Test {
  // fields and methods
}
...
// try to create an object Language
// throws an error
Test obj = new Test();
```

- An abstract class can have both abstract methods (method without body) and non-abstract methods (method with the body).
- **For Example:**

```
abstract class Test {

  // abstract method
  public abstract void display1();
  // non-abstract method
  public void display2() {
    Console.WriteLine("Non abstract method");
  }
}
```

## ABSTRACT METHOD

- A method that does not have a body is known as an abstract method.
- The abstract keyword is used to indicate that a method is abstract.
- An abstract method is a method that is declared,
- but not defined in a base class, and its implementation is left to the derived classes.
- An abstract method must be declared in an abstract class.
- **For Example:**

```
public abstract class Shape
{
    public abstract double GetArea();
}
```

## INTERFACE

- An interface is similar to abstract class.
- However, unlike abstract classes, all methods of an interface are fully abstract (method without body).
- We use the interface keyword to create an interface.
- **For Example:**

```
interface Rectangle {

  // method without body
  void calculateArea();
}
```

Here,
- Rectangle is the name of the interface.
- By convention, interface starts with I so that we can identify it just by seeing its name.
- We cannot use access modifiers inside an interface.
- All members of an interface are public by default.
- An interface doesn't allow fields.

## INTERFACE

# INTERFACE

- Interfaces specify what a class must do and not how.
- Interfaces can't have private members.
- By default all the members of Interface are public and abstract.
- Interface cannot contain fields because they represent a particular implementation of data.
- Multiple inheritance is possible with the help of Interfaces but not with classes.

# ADVANTAGES OF INTERFACE


**Interface vs. Abstract Class**

Interfaces and abstract classes are similar. The following describes some important differences:

- An abstract class may have member variables as well as non-abstract methods or properties. An interface cannot.
- A class or abstract class can only inherit from one class or abstract class.
- A class or abstract class may implement one or more interfaces.
- An interface can only extend other interfaces.
- An abstract class may have non-public methods and properties (also abstract ones). An interface can only have public members.
- An abstract class may have constants, static methods and static members. An interface cannot.
- An abstract class may have constructors. An interface cannot.

# STATIC CLASS

- Static means something which cannot be instantiated.
- You cannot create an object of a static class,
- and cannot access static members using an object.
- C# static class cannot contain instance constructors.
- Apply ***static*** modifier before the class name and after access modifier to make a class static.
- **Syntax:**

```
static class classname
{
    //static data members
    //static methods
}
```

# STATIC CLASS AND STATIC METHOD

- Below, the ***Calculator*** class is a static. All the members of it are also static.

```
public static class Calculator
{
    private static int _resultStorage = 0;

    public static string Type = "Arithmetic";

    public static int Sum(int num1, int num2)
    {
        return num1 + num2;
    }
    public static void Store(int result)
    {
        _resultStorage = result;
    }
}
```

## ADVANTAGES OF STATIC CLASS

- You will get an error if you declare any member as a non-static member.
- When you try to create an instance to the static class, it again generates a compile time error
- because the static members can be accessed directly with their class name.
- *Static* keyword is used before the class keyword in a class definition to declare a static class.
- Static class members are accessed by the class name followed by the member name.

## EXTENSION METHODS

- Extension methods, as the name suggests, are additional methods.
- These methods create and add new methods to existing class without creating new child class.
- They are the special type of the static methods that can be called as instance methods.
- We can add extension methods in both predefined classes and user created custom classes.

```
int i = 10;

bool result = i.IsGreaterThan(100); //returns false
```

- In the following example, ***IsGreaterThan()*** is an extension method for int type,
- which returns true if value of the int variable is greater than the supplied integer parameter.

# EXTENSION METHODS

**We need to consider the following points to define an extension method.**

- An extension method should be a static method.
- It must have this keyword associate with class name.
- The class name should be the first parameter in the parameter list.

# PARTIAL CLASS

- Partial Class is a unique feature of C#.
- You can split the implementation of a class, a struct, a method, or an interface in multiple .cs files
- The compiler will combine all the implementation from multiple .cs files when the program is compiled.
- The ***partial*** keyword is used to build a partial class.
- **Syntax:**

```
public partial Class_name
{
    // code
}
```

# ADVANTAGES OF PARTIAL CLASS

- Multiple developers can work simultaneously in the same class in different files.
- You can split the UI of design code to read and understand the code.
- When you were working with automatically generated code,
- the code can be added to class without having to recreate the source file like in Visual studio.
- You can also maintain your application in an efficient manner by compressing large classes into small ones.

# PARTIAL METHODS

- A partial class may contain a partial method.
- One part of the class contains the signature of the method.
- An optional implementation may be defined in the same part or another part.
- If the implementation is not supplied, then method and all calls are removed at compile time.
- Both declaration and implementation of a method must have the ***partial*** keyword.
- **Syntax:**

```
partial void method_name
{
    // Code
}
```

# PROPERTY

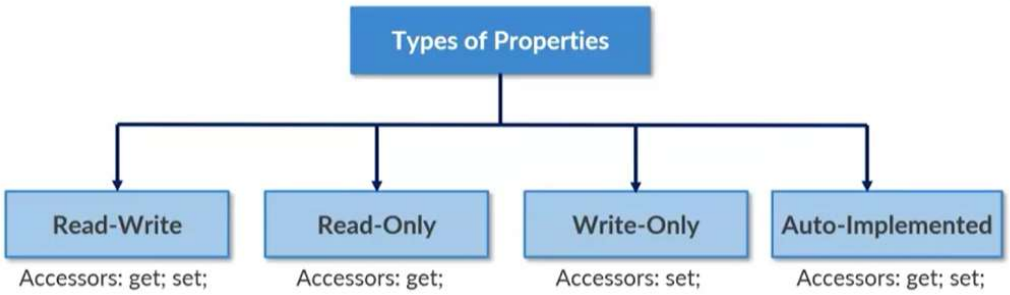
- Property is a class member that exposes the class' private fields.
- Internally, C# properties are special methods called accessors.
- It has two accessors, a get property accessor or a getter and a set property accessor or a setter.
- A get accessor returns a property value, and a set accessor assigns a new value.
- The value keyword represents the value of a property.
- **The general form of declaring a property is as follows:**

```
{
    get{ }
    set{ }
}
```

# USAGE OF PROPERTIES

- C# Properties can be read-only or write-only.
- We can have logic while setting values in the C# Properties.
- We make fields of class private, so that fields can't be accessed from outside the class directly
- Now we are forced to use C# properties for setting or getting values.

# USAGE OF PROPERTIES




# 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 ([ ]).
- **A one dimensional indexer has the following syntax:**

```
element-type this[int index] {

   // The get accessor.
   get {
      // return the value specified by index
   }
   // The set accessor.
   set {
      // set the value specified by index
   }
}
```

# INDEXER vs. PROPERTIES

| Indexer | Properties |
|---|---|
| Indexers are created with this keyword. | Properties don't require this keyword. |
| Indexers are identified by signature. | Properties are identified by their names. |
| Indexers are accessed using indexes. | Properties are accessed by their names. |
| Indexer are instance member, so can't be static. | Properties can be static as well as instance members. |
| A get accessor has the same formal parameter list as the indexer. | A get accessor of a property has no parameters. |
| A set accessor has the same formal parameter list as the indexer, in addition to the value parameter. | A set accessor of a property contains implicit value parameter. |

# ENUM

- An enum is a user-defined data type that has a fixed set of related values.
- We use the enum keyword to create an enum.
- **For Example:**

```
enum Months
{
    May,
    June,
    July
}
```

# EXCEPTION HANDLING

- An exception is an unexpected event that occurs during program execution.
- They abnormally terminate flow of program instructions, we need to handle those exceptions.
- The actions to be performed in case of occurrence of an exception is not known to program.
- In such a case, we create an exception object and call the exception handler code.
- Responding or handling exceptions is called Exception Handling.

# EXCEPTION HANDLER KEYWORDS

| Keyword | Definition |
|---|---|
| try | Used to define a try block. This block holds the code that may throw an exception. |
| catch | Used to define a catch block. This block catches exception thrown by the try block. |
| finally | Used to define the finally block. This block holds the default code. |
| throw | Used to throw an exception manually. |

# TRY-CATCH BLOCK

- The ***try..catch*** block is used to handle exceptions in C#.
- **Syntax:**

```
try
{
    // code that may raise an exception
}
catch (Exception e)
{
    // code that handles the exception
}
```

- Here, we place the code that might generate an exception inside the try block.
- The try block then throws the exception to the catch block which handles the raised exception.

# TRY-CATCH-FINALLY BLOCK

- You can also use finally block with try and catch block.
- The finally block is always executed whether there is an exception or not.
- **Syntax:**

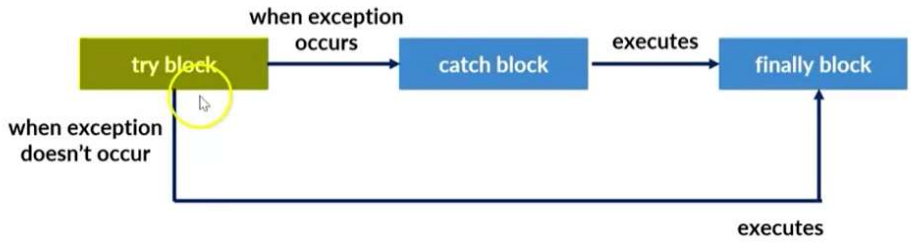
```
try
{
    // code that may raise an exception
}
catch (Exception e)
{
    // code that handles the exception
}
finally
{
    // this code is always executed
}
```

# TRY-CATCH-FINALLY BLOCK

**We can see in below image that finally block is executed in both cases.**

**The finally block is executed:**

- **after try and catch block -** when exception has occurred
- **after try block -** when exception doesn't occur




# ANONYMOUS TYPE

- It is introduced in C# 3.0.
- Anonymous types allow us to create an object that has read only properties.
- Anonymous object is an object that has no explicit type.
- C# compiler generates type name and is accessible only for the current block of code.
- These are best for the "use and throw" types.
- To create anonymous types, we must use new operator with an object initializer.

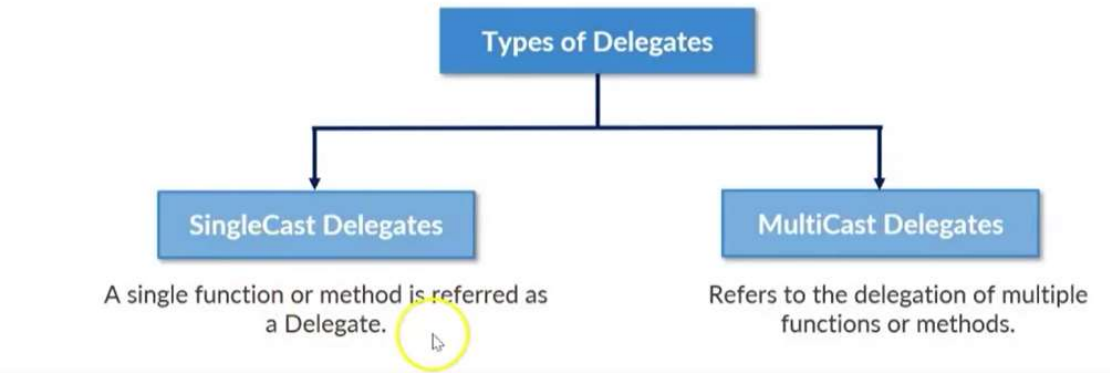
- In the below example, we are creating anonymous types by using “new” keyword with the object initializer.

```
var anonyInfo = new
{
    Fname = "abc",
    Lname = "xyz"
};
Console.WriteLine("Fname:"+anonyInfo.Lname);
```

## DELEGATES

- A delegate is a pointer to a method.
- But it is objected-oriented, secured and type-safe than function pointer.
- That means, a delegate holds the address of a method which can be called using that delegate.
- For static method, delegate encapsulates method only.
- But for instance method, it encapsulates method and instance both.
- **There are three steps involved while working with delegates:**
  - Declare a delegate
  - Set a target method
  - Invoke a delegate

## TYPES OF DELEGATES



# EVENTS

- Events in C#, being a subset of delegates are defined by using... delegates.
- An event is an encapsulated delegate.
- To raise an event in C# you need a publisher,
- and to receive and handle an event you need a subscriber or multiple subscribers.
- These are usually implemented as publisher and subscriber classes.
- **Syntax:**

```
event delegate_name event_name;
```

# ANONYMOUS METHOD

- As the name suggests, an anonymous method is a method without a name.
- Anonymous methods can be defined using the delegate keyword.
- They can be assigned to a variable of delegate type.
- Anonymous methods can access variables defined in an outer function.
- **Example:**

```
public delegate void Print(int value);

static void Main(string[] args)
{
    Print print = delegate(int val) {
        Console.WriteLine("Inside Anonymous method. Value: {0}",
    };

    print(500);
}
```

- **Example:**

```
public delegate void Print(int value);

static void Main(string[] args)
{
    Print print = delegate(int val) {
        Console.WriteLine("Inside Anonymous method. Value: {0}",
    };

    print(500);
}
```

# LAMBDA EXPRESSION

- C# Lambda Expression is a short block of code that accepts parameters and returns a value.
- It is defined as an anonymous function (function without a name).
- Lambda expressions in C# are used like anonymous functions,
- with the difference that in Lambda expressions you don't need to specify the type of the value that you input thus making it more flexible to use.
- The '=>' is the lambda operator which is used in all lambda expressions.
- **The Lambda expression is divided into two parts,**
  - the left side is the input and the right is the expression.

# TYPES OF LAMBDA EXPRESSION

**The Lambda Expressions can be of two types:**

- **Expression Lambda:** Consists of the input and the expression.
- **Syntax:**

```
input => expression;
```

- **Statement Lambda:** Consists of the input and a set of statements to be executed.

```
// Expression Lambda:

var numbers = new int[] { 2, 5, 6, 5, 1, 3, 5, 7 };
var count = numbers.Count(x => x == 5);
Console.WriteLine(count);

// Staement Lambda

List<int> numbers2 = new List<int> { 2, 5, 6, 5, 1, 3, 5, 7 };
count = numbers.Count(x => { return x == 5; });
Console.WriteLine(count);
```

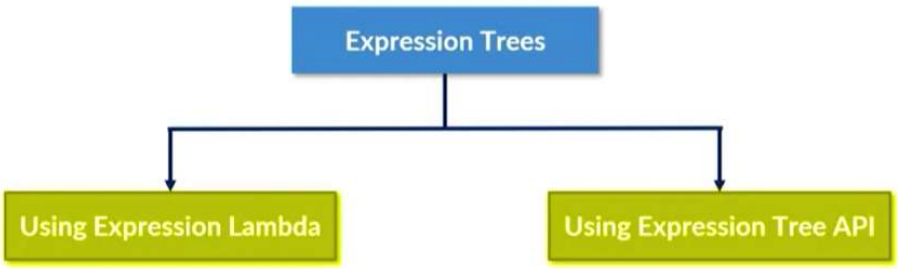
# EXPRESSION TREE

- Expression tree is nothing but expressions arranged in a tree-like data structure.
- Each node in an expression tree is an expression.
- Expression tree is an in-memory representation of a lambda expression.
- It holds the actual elements of the query, not the result of the query.
- The expression tree makes the structure of the lambda expression transparent and explicit.
- You can interact with the data in expression tree just as you can with any other data structure.
- **Syntax:**

```
Expression<TDelegate> name = lambdaExpression;
```

# EXPRESSION TREE

**Expression trees can be created by using following two ways:**




# EXPRESSION TREE STRUCTURE

- The simple structure of an ***Expression<TDelegate>*** has four properties as given below:
  - **Body:** The body of the expression.
  - **Parameters:** The parameters of the lambda expression.
  - **NodeType:** The type of node in the tree
  - **Type:** The type of the expression

```
// Expression Tree:

Func<string, string, string> stringJoins = (str1, str2) => string.Concat(str1, str2);

Expression< Func<string, string, string>> stringJoinExpr = (str1, str2) => string.Concat(str1, str2);

var func = stringJoinExpr.Compile();
var result = func("Hello", "World");
Console.WriteLine(result);

// OR

result = stringJoinExpr.Compile()("Hello", "Everyone");
Console.WriteLine(result);
```