.NET Core Training

C# Fundamentals

Week 2 : 2nd Feb – 8th Feb

Action Items

- Classes
- Constructors
- Static key word
- Access modifiers
- Lists & Arrays

Classes and Objects

Class:

A Class is a "blueprint" for creating objects.

Syntax:
```
class className
{
datatype name = value;
}
```

Constructor :

Syntax:
```
public ConstructorName()
{

}
```

Classes and Objects

Arrays:

Arrays are used to store multiple values in a single variable.

Array Creation:

// Create an array of four elements, and add values later
String[ ] cars = new string[4];

// Create an array of four elements and add values right away
string[ ] cars = new string[4] {"Volvo", "BMW", "Ford", "Mazda"};

// Create an array of four elements without specifying the size
string[ ] cars = new string[ ] {"Volvo", "BMW", "Ford", "Mazda"};

// Create an array of four elements, omitting the new keyword, and without specifying the size
string[ ] cars = {"Volvo", "BMW", "Ford", "Mazda"};

Classes and Objects

ArrayList:

An ArrayList can be used to add unknown data where you don't know the types and the size of the data.

Example :

```
// adding elements using ArrayList.Add() method
var arlist1 = new ArrayList();
arlist1.Add(1);
arlist1.Add("Bill");
arlist1.Add(" ");
arlist1.Add(true);
arlist1.Add(4.5);
arlist1.Add(null); // adding elements using object initializer syntax
var arlist2 = new ArrayList() { 2, "Steve", " ", true, 4.5, null };
```

Classes and Objects

Lists:

The List<T> is a collection of strongly typed objects that can be accessed by index and having methods for sorting, searching, and modifying list.

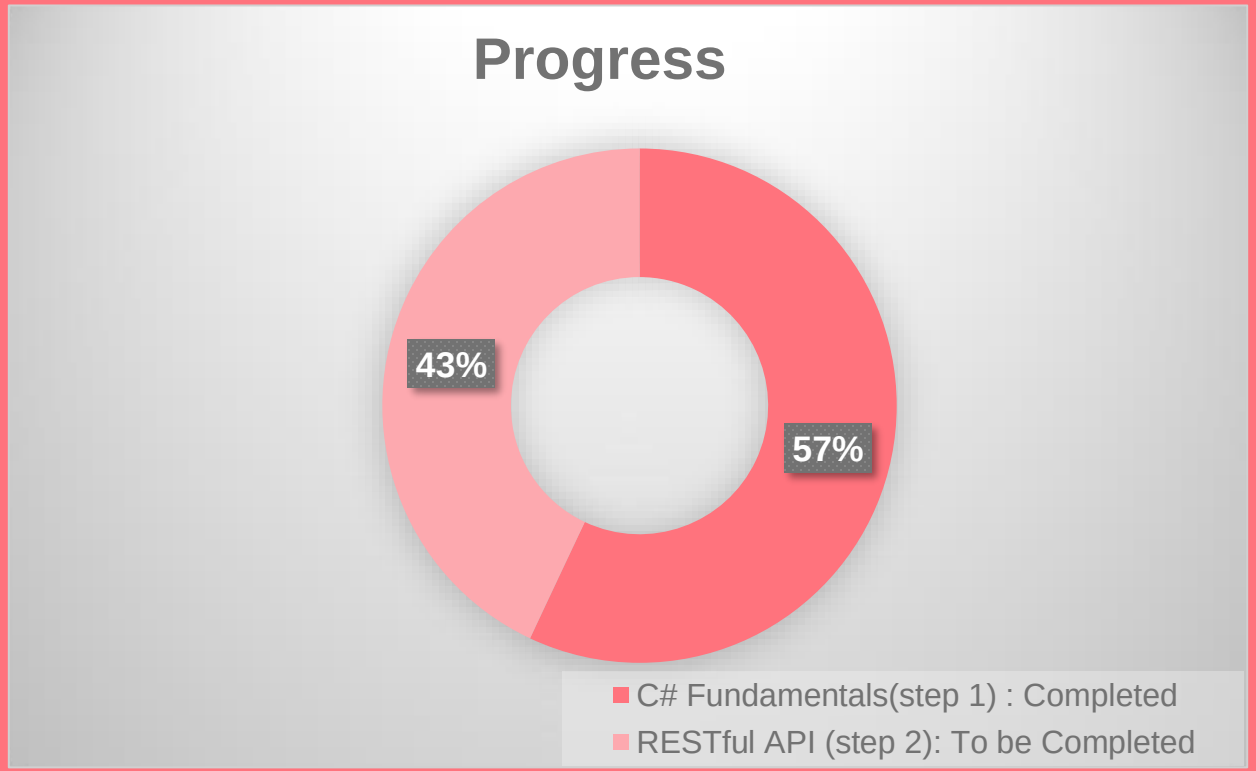
Syntax: *List<datatype> ListName = new List<datatype>();*

Static key word :

Static is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static members.

AGENDA

1. Progress
2. Learnings from the course
3. Demo


Progress
43%
57%
C# Fundamentals(step 1) : Completed
RESTful API (step 2): To be Completed

Learnings from the Course

Reference Types & Value Types

Control Flow

Building Types

OOP with C#

Reference Types and Value Types

Value Type :

A data type is a value type if it holds a data value within its own memory space.

Example : int i = 100;

Reference Type :

A reference type contains a pointer to another memory location that holds the data.

Example : string s = "Hello World!!";

Reference Types and Value Types

Passing Value Type Variables :

Example :

```
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 Types and Value Types

Passing Reference Type Variables :

Example :

```
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);
 }
```

Control Flow

C# IF Statement:

Syntax:

```
if(condition)
{
//code to be executed
}
```

C# IF-else Statement:

Syntax:

```
if(condition){
//code if condition is true
}
else{
//code if condition is false
}
```

Control Flow

C# IF-else-if ladder Statement :

Syntax:

```
if(condition1){
//code to be executed if condition1 is true
}
else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

Control Flow

C# switch :

Syntax:

```
switch(expression){
case value1:
//code to be executed;
break;
case value2:
 //code to be executed;
 break;
.....

default:
//code to be executed if all cases are not matched;
break;
}
```

Control Flow

C# For Loop:

Syntax:
```
for(initialization; condition; incr/decr){
//code to be executed
}
```

C# Do-While Loop :

Syntax:
```
do{
//code to be executed
}while(condition);
```

C# While Loop :

Syntax:
```
while(condition){
//code to be executed
}
```

Control Flow

C# Break Statement :

Syntax: *jump-statement;*
break;

C# Continue Statement :

Syntax: *jump-statement;*
continue;

Control Flow

Exception Handling in C#:

- **Try/Catch and Finally**

Syntax:

```
try
{ // statements causing exception
}
catch( ExceptionName e1 )
{ // error handling code
}
finally { Console.WriteLine(“ "); }
 Console.WriteLine("Rest of the code");
```

Building Types

Method Overloading :

With method overloading, multiple methods can have the same name with different parameters.

Example : *int MyMethod(int x)*

float MyMethod(float x)

double MyMethod(double x, double y)

Building Types

Properties in C# :

A property is like a combination of a variable and a method, and it has two methods: a **get** and a **set** method.

Example:

```
public string Name
{
  get
 {
  return name;
 }
  set
{
  name = value;
 }
```

Building Types

Delegates :
A delegate is a reference type variable that holds the reference to a method. The reference can be changed at runtime.

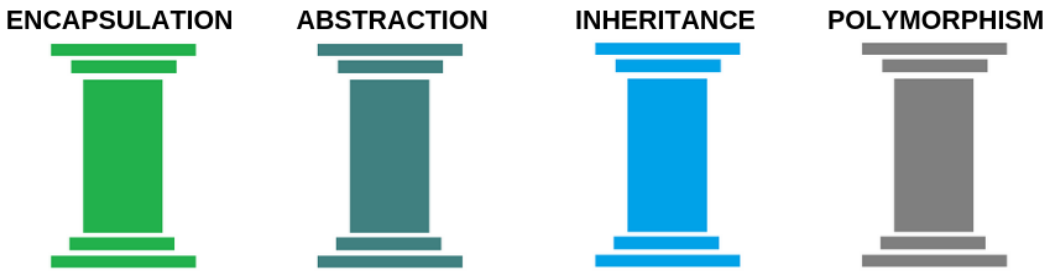
Delegate Declaration :
Syntax: ***delegate*** *<return type> <delegate-name> <parameter list> ;*

Events :
Events are user actions such as clicks, mouse movements etc.,. Events are used for inter-process communication.

Event Declaration :
Syntax: ***event*** *<delegate-name> <event-name> ;*

OOPs with C#

Four Pillars of OOP :



OOPs with C#

Encapsulation :

Encapsulation, in object-oriented programming methodology, prevents access to implementation details. Encapsulation is implemented by using **access specifiers**.

C# supports the following access specifiers :

- Public
- Private
- Protected
- Internal
- Protected internal

OOPs with C#

Inheritance :

In C#, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which is defined in other class.

In C#, the class which inherits the members of another class is called **derived class** and the class whose members are inherited is called **base class**.

Syntax: *<acess-specifier> class <base_class> { ... } class <derived_class> : <base_class> { ... }*

OOPs with C#

Polymorphism :

Overriding Methods

Overriding allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its parent classes.

virtual keyword: This modifier or keyword use within base class method. It is used to modify a method in base class for overridden that particular method in the derived class.

override: This modifier or keyword use with derived class method. It is used to modify a virtual or abstract method into derived class which presents in base class.

OOPs with C#

Abstraction :

Data abstraction is the process of hiding certain details and showing only essential information to the user.
Abstraction can be achieved with either **abstract classes** or **interfaces.**

•**Abstract class:** is a restricted class that cannot be used to create objects.

•**Abstract method:** can only be used in an abstract class, and it does not have a body.

OOPs with C#

Abstraction :

Example :

```
//Abstract Class
abstract class Animal
{
//Abstract method
public abstract void animalSound();
public void sleep()
{
Console.WriteLine(“ ");
}
}
```

OOPs with C#

Abstraction :

Interface :
An interface is a completely "abstract class", which can only contain abstract methods and properties (with empty bodies):

Example:
```
// interface
interface Animal
{
void animalSound(); // interface method (does not have a body)
void run(); // interface method (does not have a body)
}
```

DEMO

THANKYOU!