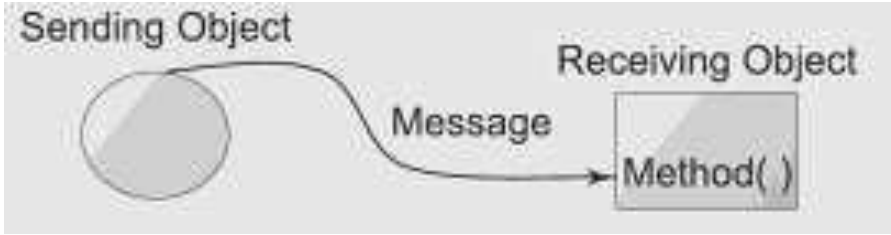
Classes & Objects in C#

INDEX

Introduction

- C# is a true object oriented language.
- C# program must be encapsulated in a class that defines the state and behaviour of the basic program components known as objects.
- Classes creates objects and objects use methods to communicate between them.
- Classes provide a convenient approach for packing together a group of logically related data items and functions that work on them.
- In C# data items are called fields and the functions are called methods.



Basic Principles of OOP

All object-oriented languages employ three core principles

Encapsulation: Provides the ability to hide the internal details of an object from its user. The outside user may not be able to change the state of an object directly. Encapsulation is implemented using the access modifier keywords **public, private and protected.** The concept of encapsulation is also known as data hiding or information hiding.

Inheritance: It is the concept we use to build new classes using the existing class definitions. Through inheritance we can modify a class the way we want to create new objects. The original class is known as *base* or *parent class* and the modified one is known as *derived class* or *subclass* or *child class.*

Polymorphism: it is the third concept of OOP. It is the ability to take more than one form.

Defining a class

- Class is user defined data type with a template that serves to define its properties.
  - One the class type has been defined we can create 'variables' of that type using declaration that are similar to the basic type declaration.
  - In C# these variables are termed as instances of classes, which are the actual *objects.*

```
class classname
{
    [variables declaration;]
    [methods declaration;]
}
```

class is a keyword and classname is any valid c# identifier. Everything inside the square brackets is optional

```
class Empty
{
}
```

is also a valid class definition

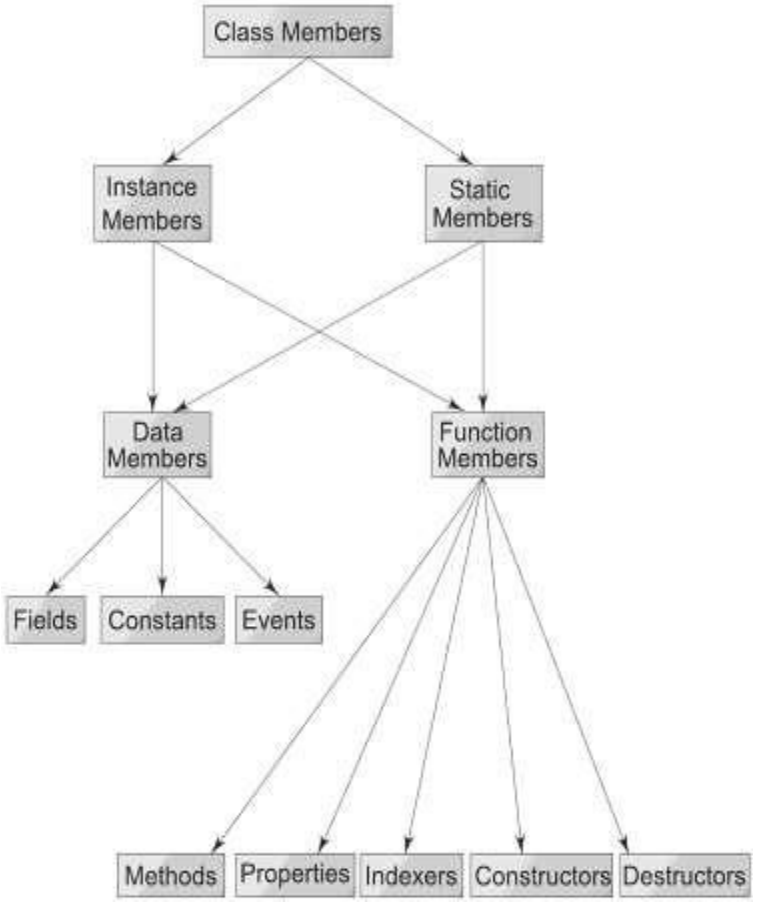
- *A class can be public or internal*

Adding Variables

Data is encapsulated in a class by placing data fields inside the body of the class definition .

```
class Rectangle
{
   int length;
   int width;
}
```

These instance variable is also known as member variable and no storage space has been created.



Adding Methods

- A class with only data fields and without methods that operates on that data has no life.
- The objects created by such class can not respond to any messages.
- Methods are declared inside the body of the class, usually after the declaration of instance variables.

```
type methodname (parameter-list)
{
    method-body;
}
```

```
class Rectangle
{
    int length;
    int width;

    public void GetData(int x, int y)
    {
        length = x;
        width = y;
    }

    public int RectArea()
    {
        int area = length * width;
        return area;
    }
}
```

Adding Methods (contd..)

- Instance variables and methods in classes are accessible by all the methods in the class, but a method can not access the variables declared in other methods.

```
class Access
{
    int x ;                          //instance variable
    public void Method1( )           //a method
    {
        int  y ;
        x = 10 ;          // legal
        y =  x  ;         // legal
    }

    public void Method2( )   //another method
    {
        int z   ;
        x = 5   ;         // legal
        z = 10  ;         // legal
        y = 1   ;         // illegal, y defined in Method1
    }
}
```

Member Access modifiers

- A class may be designed to hide its members from outside accessibility .
- C# provides a set of access modifiers that can be used with the members of a class to control their visibility to outside users.
- In C# all members have private access by default.

Modifier	*Accessibility Control*
private	Member is accessible only within the class containing the member.
public	Member is accessible from anywhere outside the class as well. It is also accessible in derived classes.
protected	Member is visible only to its own class and its derived classes.
internal	Member is available within the assembly or component that is being created but not to the clients of that component.
protected internal	Available in the containing program or assembly and in the derived classes.

Creating Objects

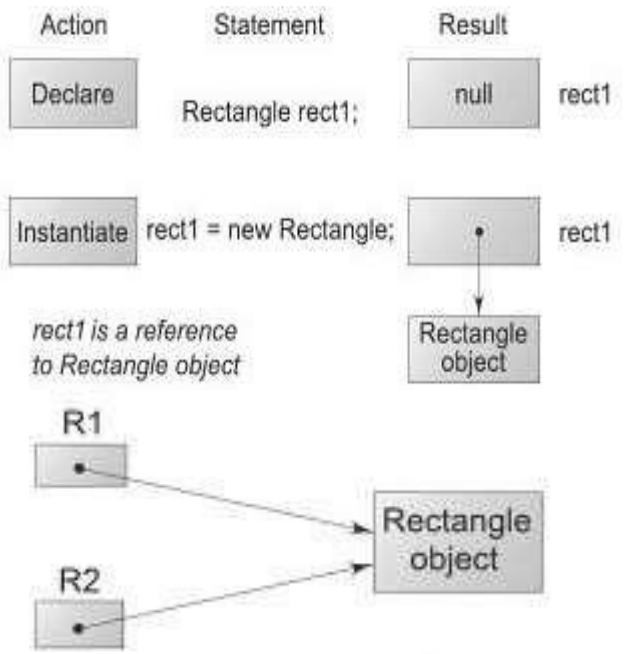
- Creating an objects is also referred to as instantiating an object.
- object in C# created using the **new** operator.
- The **new** operator creates an object of the specified class and returns a reference to that object.

Ex.

```
Rectangle rect1;    //declare
Rect1 = new Rectangle( );
```

OR

```
Rectangle r1 = new Reactangle();
Rectangle r2 = r1;
```



Accessing Class Members

objectname.variable_name;

objectname.methodname(parameter-list);

Example:

```
Rectangle rect1 = new Rectangle( );
Rectangle rect2 = new Rectangle();
rect1.length = 15; rect1.width = 10;
rect2.length =20; rect2.width = 12;
```

Application of Classes and Objects

```
using System;
class Rectangle
{
    public int length, width;              //  Declaration of variables

    public void GetData(int x, int y)      //  Definition of method
    {
        length = x;
        width  = y;
    }

    public int RectArea()
    {
        int area = length * width;
        return (area);
    }
}
```

```
class RectArea                                  //  class with main method
{
    public static void Main( )
    {
        int area1,area2;                        //      Local variables
        Rectangle rect1 = new Rectangle();      //  Creating objects
        Rectangle rect2 = new Rectangle();

        rect1.length = 15;                      //  Accessing variables
        rect1.width  = 10;
        area1 = rect1.length * rect1.width;

        rect2.GetData(20,12);                   //  Accessing methods
        area2 = rect2.RectArea();

        Console.WriteLine("Area1 = " + area1);
        Console.WriteLine("Area2 = " + area2);
    }
}
```

Constructors

- Initialization of all objects can be done by two approaches.
- **First Approach** uses the dot operator to access the instance variables and then assigns value to them individually.
- **Second approach** takes the help of a function like **LoadData** to initialize each object.
- Both the approaches are tedious to initialize objects.

Constructors (contd..)

- C# supports a special type of method, called constructor. That enables an object to initialize itself when it is created.
- Constructor have same name as the class itself.
- They do not specify a return type, not even void.
- By default constructors are public but they can also be declared as private or protected. In such cases, the objects of that class can not be created and also the class can not be used as a base class for inheritance.

```
class Rectangle
{
  public int length;
  public int width;

  public Rectangle(int x, int )
  {
    length = x;
    width = y;
  }

  public int RectArea()
  {
    return length * width;
  }
}
```

Overloaded Constructors

- In C# , it is possible to create methods that have the same name, but different parameter lists and different definitions. **(Method overloading).**
- Method overloading can be used when objects are required to perform similar tasks but using different parameters.
- Similarly we can create overloaded constructor method by providing several different constructor definitions with different parameter lists.

```
class Rectangle
{
  public int length; public
  int width;  public

   Rectangle(int x, int y)
   {
     length = x;
     width = y;
   }

   public Rectangle(int x)
   {
     length = width = x;
   }

   public int RectArea( )
   {
     return length * width;
   }
}
```

Static Members

- Class contains two sections.
  - one declares variables (instance variables)
  - other declares methods (instance methods).
- Every time the class is instantiated, a new copy of each is created and accessed using dot operator.
- static members and methods are declared by using keyword ***static*.**
- static variable are associated with the class itself rather than with individual objects.
- static variables and static methods are referred as class variables and class methods.
- static variables are used when we want to have a variable common to all instances of a class.
- static method can be called without using the objects.
- They are also for use by other classes.

Static Members (contd..)

```
using system;
class Xyz
{
  public static int Mul(int x, int y)
  {
    return x * y;
  }
  public static float Div(float x, float y)
  {
    return x / y;
  }
}
class Calculation
{
  public void static Main()
  {
    int a = Xyz.Mul(4,5);
    float b = Xyz.Div(3.4F, 1.7F);
    Console.WriteLine("a = {0} and b = {1}", a, b);
  }
}
```

Note: Static methods are called using class names. static methods have following restrictions

1. **They can only call other static methods.**
2. **they can only access static data.**
3. **they can not refer to this or base in any ways.**

Static constructors

- static constructor is called before any object of the class is created.
- static constructor is declared by prefixing a **static** keyword to the constructor definition.
- it can not have any parameters.

Example:

```
class abc
{
        static abc( )     //no parameters
        {
        ……..                  //set value for static members
        ……..
        }
}
```

- A static constructor does not take access modifiers or have parameters.
- A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.
- A static constructor cannot be called directly.
- The user has no control on when the static constructor is executed in the program.
- A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.

Copy Constructor

- A copy constructor creates an object by copying variables from another object
- C# does not provide a copy constructor .
- if we wish to add this feature to the class . A copy constructor is defined as follows

```
public Item (Item item)
{
code = item.code;
price = item.price;
}
```

- copy constructor is invoked by instantiating an object of type **Item** and passing it the object to be copied.

Item item2 **= new Item** (item1);

now item1 is copy to item2.

Destructor: As C# manages the memory dynamically and uses a garbage collector, running on a separate thread to execute all destructor on exit.

Private Constructor

- C# does not have global variables or constants.
- All declarations must be contained in a class.
- In many situations we define some utility classes that contains only static members.
- object of a class having private constructor cannot be instantiated from outside of the class. However, we can create object of the class inside class methods itself.
- Private constructor is constructor that is preceded by **private** access specifier.

Private Constructor

```
using System;

namespace Xyz
{
   class User
   {
      // private Constructor
      private User()
      {
        Console.WriteLine("Private Constructor");
      }
      public static string name, location;
      // Default Constructor
      public User(string a, string b)
      {
         name = a;
         location = b;
      }
   }
```

```
class Program
   {
      static void Main(string[] args)
      {
         // The following comment line will throw an error
        // because constructor is inaccessible
         //User user = new User();

         // Only Default constructor with parameters will invoke
         User user1 = new User("JITIN KUMAR", "NEW DELHI");
         Console.WriteLine(User.name + ", " + User.location);
         Console.WriteLine("\nPress Enter Key to Exit..");
         Console.ReadLine();
      }
   }
}
```

Destructors

A destructor is opposite to a constructor. It is a method called when an object is no more required. The name of the destructor is the same as the class name and is preceded by a tilde (–). Like constructors, a destructor has no return type. *Example:*

```
class Fun
{
    . . . .
    . . . .
    ~ Fun ( ) //No arguments
    {
    . . . .
    }
    }
```

Note that the destructor takes no arguments.

C# manages the memory dynamically and uses a *garbage collector,* running on a separate thread, to execute all destructors on exit. The process of calling a destructor when an object is reclaimed by the garbage collector is called *finalization.*

The **this** Reference

C# supports the keyword **this** which is a reference to the object that called the method. The **this** reference is available within all the member methods and always refers to the current instance. It is normally used to distinguish between local and instance variables that have the same name. Consider the code segment shown below:

```
class Integers
{
    int x;
    int y;
    public void SetXY(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    ....
    ....
}
```

In the assignment statements, **this.x** and **this.y** refer to the class members named **x** and **y** whereas simple **x** and **y** refer to the parameters of the **SetXY( )** method.

Indexers

Indexers are location indicators and are used to access class objects, just like accessing elements in an array. They are useful in cases where a class is a container for other objects.

An indexer looks like a property and is written the same way a property is written, but with two differences:

- The indexer takes an *index* argument and looks like an array.
- The indexer is declared using the name **this**.

- Indexer Concept is object act as an array.
- Indexer an object to be indexed in the same way as an array.
- Indexer modifier can be private, public, protected or internal.
- The return type can be any valid C# types.
- Indexers in C# must have at least one parameter. Else the compiler will generate a compilation error.

```
this [Parameter]
{
   get
   {
      // Get codes goes here
   }
   set
   {
      // Set codes goes here
   }
}
```

```
using System;
using System.Collections;
class List
{
    ArrayList array = new ArrayList ( );
    public object this[ int index ]
    {
        get
        {
            if(index < 0 || index >= array.Count)
            {
                return null;
            }
            else
            {
                return (array [index] );
            }
        }
        set
        {
            array[index] = value;
        }
    }
}
class IndexerTest
{
    public static void Main ( )
    {
        List list = new List ( );
        list [0] = "123";
        list [1] = "abc";
        list [2] = "xyz";
        for (int i = 0, i < list.Count; i++)
            Console.WriteLine( list[i]);
    }
}
```