Object-Oriented Programming

PASSING PARAMETERS

EXCEPTIONS HANDLING

Defining Method Signatures

Methods, constructors, indexers, and operators are characterized by their signatures. A signature makes a method look unique to the C# compiler. The method name and the type and order of parameters all contribute to the uniqueness of signatures.

A method signature consists of the name of the method and the type, such as **value** or **reference**. A method signature does not include the return type.

Defining Method Signatures

Signatures Example class

```
void MyFunc(); // MyFunc ()

void MyFunc(int x); // MyFunc (int)

void MyFunc(ref int x); // MyFunc (ref int)

void MyFunc(out int x); // MyFunc (out int)

void MyFunc(int x, int y); // MyFunc (int, int)

int MyFunc(string s); // MyFunc (string)

int MyFunc(int x); // MyFunc (int)

void MyFunc(string[] a); // MyFunc (string[])
```

Writing Methods That Require a Single Argument

```
using System;
public class UseSevenPercentSalesTax
{
        public static void Main()
        {
                double myPurchase = 12.99;
                ComputeSevenPercentSalesTax(myPurchase);
                ComputeSevenPercentSalesTax(35.67);
        }
        public static void ComputeSevenPercentSalesTax(double saleAmount)
        {
                double tax;
                tax = saleAmount * 0.07;
                Console.WriteLine("The tax on {0} is {1}",
                      saleAmount, tax.ToString("f"));
        }
}
```

Complete program using the ComputeSevenPercentSalesTax() method

Writing Methods That Require Multiple Arguments

You can pass multiple arguments to a method by listing the arguments within the call to the method and separating them with commas

You can write a method to take any number of arguments in any order

Writing Methods That Return Values

The return type for a method can be any type used in the C# programming language

A method can also return nothing, in which case the return type is **void**

A method's return type is also known as a **method's type**

Writing Methods That Return Values

```
public static double CalcPay(double hours, double rate)
{
        double gross;
        gross = hours * rate;
        return gross;
}
```

The return type in the above example is **double**

The **return** statement causes the value stored in gross to be sent back to any method that calls the CalcPay() method

Passing Parameters Within Methods

In C#, you can write methods with three kinds of formal parameters listed within the parentheses in the method header

- Value parameters
- Reference parameters
- Output parameters

Passing Parameters by value

By default, parameters are passed by value. In this method a duplicate copy is made and sent to the called function. There are two copies of the variables. So if you change the value in the called method it won't be changed in the calling method.

We use this process when we want to use but don't want to change the values of the parameters passed.

Passing Parameters by value

```
using System;

namespace value_parameter
{
    class Program
    {
        class XX
        {
            public int sum(int a, int b)
            {
                a = a + 10;
                b = b + 20;
                return (a + b);
            }
        }

        static void Main(string[] args)
        {

// local data members have to initialized as they are not initiated with
class constructor
            int a = 10, b = 20;

            XX obj = new XX();
            Console.WriteLine("sum of a and b  is : " + obj.sum(a, b));
            Console.WriteLine("Value of a is : " + a);
            Console.WriteLine("Value of b is : " + b);
            Console.ReadLine();
        }
    }
}
```

Passing Parameters by out- ref

Both the **reference** and **output parameters** have memory addresses that are passed to a method, allowing it to alter the original variables

When you use a reference parameter to a method, the parameter must be assigned a value before you use it in the method call

When you use an output parameter, it does not contain an original value

Passing Parameters by out- ref

Both reference and output parameters act as **aliases**, or other names, for the same memory location

The keyword **ref** is used to indicate a reference parameter

The keyword **out** is used to indicate an output parameter

Passing Parameters by ref

```
using System;

namespace value_parameter
{
    class Program
    {
        class XX
        {
            public int sum( ref int a, ref int b)
            {
                a = a + 10;
                b = b + 20;
                return (a + b);
            }
        }

        static void Main(string[] args)
        {

// local data members have to initialized as they are not initiated with
class constructor
          int a = 10, b = 20;

          XX obj = new XX();
          Console.WriteLine("sum of a and b is : " + obj.sum(ref a, ref b));
          Console.WriteLine("Value of a is: " + a);
          Console.WriteLine("Value of b is: " + b);
          Console.ReadLine();
        }
    }
}
```

Passing Parameters by ref

In the preceding code:

- The modifier ref precedes the variable in both the method call and the method header
- The passed and received variables occupy the same memory location
- The passed variable was assigned a value

Passing Parameters by out

Unlike the reference parameter, the output parameter does not need a value assigned to it (before it is used in a method call)

The output parameter is convenient when the passed variable doesn't have a value yet

The output parameter gets its value from the method, and these values persist back to the calling program

Passing Parameters by out

```
using System;
namespace out_parameter
{
    class Program
    {
        class XX
        {
            public int sum(int a, int b, out int c, out int d)
            {
                c = a + b;
                d = c + 100;
                return (a + b);
            }
        }
        static void Main(string[] args)
        {
            int a = 10, b = 20;

            // the out data members doesn't need to assign initial value as
they are processed and brought back
            int c, d;

            XX obj = new XX();

            Console.WriteLine("sum of a and b  is : " + obj.sum(a,
b, out c, out d));
            Console.WriteLine("Value of a is : " + a);
            Console.WriteLine("Value of b is : " + b);
            Console.WriteLine("Value of c is : " + c);
            Console.WriteLine("Value of d is : " + d);

            Console.ReadLine();
        }
    }
}
```

Using ref and out Parameters Within Methods

What is the difference between ref and out parameter modifiers?

ref

- ref is a mechanism of parameter passing by reference
- A variable to be sent as ref parameter must be initialized.
- ref is bidirectional i.e. we have to supply value to the formal parameters and we get back processed value.

out

- Out is also a mechanism of parameter passing by reference
- A variable to be sent as out parameter don't need to be initialized
- Out is a unidirectional i.e. we don't supply any value but we get back processed value.

Exceptions Handling

The .NET Framework uses a structured exception handling mechanism. The following are some of the benefits of this structured exception handling:

- common support and structure across all .NET languages.
- support for the creation of protected blocks of code.
- the ability to filter exceptions to create efficient robust error handling.
- support of termination handlers to guarantee that cleanup tasks are completed.

Exceptions Handling

In other text, 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.

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**.

Exceptions 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. For example, if you open a file, it must be closed whether an exception is raised or not.

Exceptions Handling

```
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
 }
```

Example

```
using System;
namespace ErrorHandlingApplication
{
   class DivNumbers
   {
      int result;
      DivNumbers()
      {
         result = 0;
      }
      public void division(int num1, int num2)
      {
         try
         {
            result = num1 / num2;
         }
         catch (DivideByZeroException e)
         {
            Console.WriteLine("Exception caught: {0}", e);
         }
         finally
         {
            Console.WriteLine("Result: {0}", result);
         }
      }
      static void Main(string[] args)
      {
         DivNumbers d = new DivNumbers();
         d.division(25, 0);
         Console.ReadKey();
      }
   }
}
```

Syntax

- C# exceptions are represented by classes.
- The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class.
- Some of the exception classes derived from the **System**.
- Exception class are the ***System.ApplicationException*** and ***System.SystemException*** classes.

Syntax

- The ***System.ApplicationException*** class supports exceptions generated by application programs.

- So the exceptions defined by the programmers should derive from this class.

- The ***System.SystemException*** class is the base class for all predefined system exception.

Exception Classes in C#

Exception Class	Description
System.IO.IOException	Handles I/O errors.
System.IndexOutOfRangeException	Handles errors generated when a method refers to an array index out of range.
System.ArrayTypeMismatchException	Handles errors generated when type is mismatched with the array type.
System.NullReferenceException	Handles errors generated from deferencing a null object.
System.DivideByZeroException	Handles errors generated from dividing a dividend with zero.
System.InvalidCastException	Handles errors generated during typecasting.
System.OutOfMemoryException	Handles errors generated from insufficient free memory.
System.StackOverflowException	Handles errors generated from stack overflow.

Example

```
using System;
class TryCatch1
{
    static void Main()
    {
        int x = 10,y=0,z =0;
        int[] p=new int[3];
        try{
            z = x/y;
            Console.WriteLine("Result of dividing {0} by {1} is {2}", x,y,z);
        }
        catch(DivideByZeroException ex)
        {
            Console.WriteLine("Exception has occurred - divide by zero", ex);
        }
        try
        {
            p[3]=23;
            Console.WriteLine("Array element is {0}", p[0]);
        }
        catch(System.IndexOutOfRangeException ex)
        {
            Console.WriteLine("Exception has occurred - index out of range", ex);
        }
        finally
        {
            Console.WriteLine("This is the finally block");
        }
    }
}
```

Creating Objects in C#

Overview

Defining a Class

Declaring Methods

Using Constructors

Using Static Class Members

Defining a Class

What Are Classes and Objects?

How to Define a Class and Create an Object

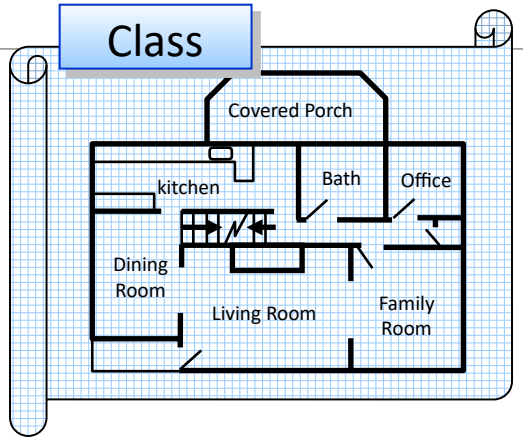
How to Organize Classes Using Namespaces

How to Define Accessibility and Scope

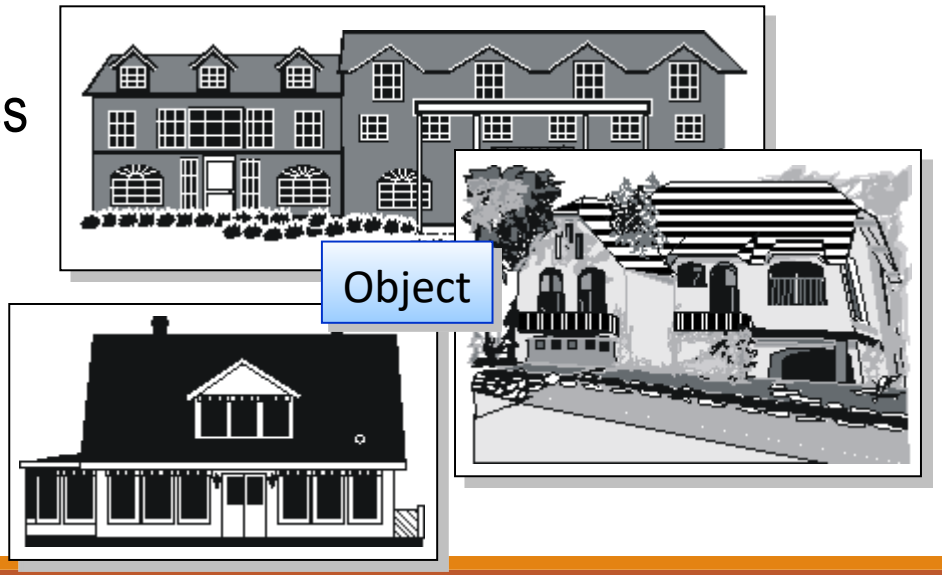
What Are Classes and Objects?

Classes:

- Are like blueprints for objects
- Contain methods and data



- **Objects:**
  - Are instances of a class
  - Created using the new keyword
  - Have actions



How to Define a Class and Create an Object

- **How to define a class**

```
public class Customer {
 public string  name;
 public decimal creditLimit;
 public uint customerID;
}
```

- **How to instantiate a class as an object**

```
Customer nextCustomer = new Customer();
```

- **How to access class variables**

```
nextCustomer.name = "Suzan Fine";
```

How to Organize Classes Using Namespaces

Declaring a namespace

```
namespace CompanyName {
  public class Customer () { }
}
```

- **Nested namespaces**

```
namespace CompanyName {
  namespace Sales {
     public class Customer () { }
  }
}
// Or
namespace CompanyName.Sales { ... }
```

- **The using statement**

```
using System;
using CompanyName.Sales;
```

Class Creation

```
<access specifier> class  class_name
{
    // member variables
    <access specifier> <data type> variable1;
    <access specifier> <data type> variable2;
    ...
    <access specifier> <data type> variableN;
    // member methods
    <access specifier> <return type> method1(parameter_list)
    {
        // method body
    }
    <access specifier> <return type> method2(parameter_list)
    {
        // method body
    }
    ...
    <access specifier> <return type> methodN(parameter_list)
    {
        // method body
    }
}
```

```
using System;
namespace BoxApplication
{
    class Box
    {
        public double length;   // Length of a box
        public double breadth;  // Breadth of a box
        public double height;   // Height of a box
    }
    class Boxtester
    {
        static void Main(string[] args)
        {
            Box Box1 = new Box();    // Declare Box1 of type Box
            Box Box2 = new Box();    // Declare Box2 of type Box
            double volume = 0.0;     // Store the volume of a box here

            // box 1 specification
            Box1.height = 5.0;
            Box1.length = 6.0;
            Box1.breadth = 7.0;

            // box 2 specification
            Box2.height = 10.0;
            Box2.length = 12.0;
            Box2.breadth = 13.0;

            // volume of box 1
            volume = Box1.height * Box1.length * Box1.breadth;
            Console.WriteLine("Volume of Box1 : {0}",  volume);

            // volume of box 2
            volume = Box2.height * Box2.length * Box2.breadth;
            Console.WriteLine("Volume of Box2 : {0}", volume);
            Console.ReadKey();
        }
    }
}
```

How to Define Accessibility and Scope

- **Access modifiers are used to define the accessibility level of class members**

Declaration	Definition
public	Access not limited.
private	Access limited to the containing class.
internal	Access limited to this program.
protected	Access limited to the containing class and to types derived from the containing class
protected internal	Access limited to the containing class, derived classes, or to members of this program

Access Modifiers , Access Specifiers

Access Modifiers (Access Specifiers) describes as the scope of accessibility of an Object and its members. All C# types and type members have an accessibility level .

We can control the scope of the member object of a class using access specifiers.

We are using access modifiers for providing security of our applications.

When we specify the accessibility of a type or member we have to declare it by using any of the access modifiers provided by C# language.

Access Modifiers , Access Specifiers

public :

public is the most common access specifier in C# . It can be access from anywhere, that means there is no restriction on accessibility.

The scope of the accessibility is inside class as well as outside. The type or member can be accessed by any other code in the same assembly or another assembly that references it.

Access Modifiers , Access Specifiers

private :

The scope of the accessibility is limited only inside the classes or struct in which they are declared. The private members cannot be accessed outside the class and it is the least permissive access level.

protected :

The scope of accessibility is limited within the class and the class derived (Inherited)from this class.

Property for private variables

Properties combine aspects of both fields and methods. To the user of an object, a property appears to be a field, accessing the property requires the same syntax. To the implementer of a class, a property is one or two code blocks, representing a get accessor and/or a set accessor.

The code block for the **get** accessor is executed when the property is read; the code block for the **set** accessor is executed when the property is assigned a new value. A property without a **set** accessor is considered read-only. A property without a **get** accessor is considered write-only. A property that has both accessors is read-write.

Property for private variables

```
class person
{
    private string name;
    private int age;
 //property
    2 references
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
}
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {

        person p1 = new person();
        p1.Name = "ahmed";    //set
        Console.WriteLine(p1.Name); // get


};
```

Declaring Methods

How to Write a Method

How to Pass Parameters to a Method

How to Pass Parameters by Reference

How to Pass a Reference Type

How to Overload a Method

How to Use XML Code Comment Features

How to Write a Method

- **A method is a command for action**

```
class Lion {
  private int weight;
  public bool IsNormalWeight () {
     if ((weight < 100)||(weight > 250)) {
        return false;
     }
     return true;
  }
  public void Eat() { /* some action */ }
  public int GetWeight() {return this.weight;}
}
. . .
Lion bigLion = new Lion();
bool weightNormal = bigLion.IsNormalWeight();
bigLion.Eat();
int weight = bigLion.GetWeight();
```

How to Pass Parameters to a Method

- **Passing by value**

```
class Lion {
  private int weight;
  public void SetWeight(int newWeight) {
     weight = newWeight;
  }
}
. . .

Lion bigLion = new Lion();

int bigLionWeight = 250;
bigLion.SetWeight( bigLionWeight );
```

How to Pass Parameters by Reference

- **Using the** ref **keyword**

```
public void GetAddress(ref int number,
      ref string street) {
      number = this.number;
      street = this.street;
}
. . .
int sNumber = 0; string streetName = null;
zoo.GetAddress( ref sNumber, ref
streetName );
// sNumber and streetName have new values
```

- **Using the** out **parameter keyword**
  - Allows you to initialize a variable in a method

How to Pass a Reference Type

When you pass a reference type to a method, the method can alter the actual object

```
class Zoo {
  public void AddLion( Lion newLion ) {
     newLion.location = "Exhibit 3";
     . . .
  }
}

. . .

Zoo myZoo = new Zoo();
Lion babyLion = new Lion();
myZoo.AddLion( babyLion );
// babyLion.location is "Exhibit 3"
```

How to Overload a Method

Overloading enables you to create multiple methods within a class that have the same name but different signatures

```
class Zoo {
  public void AddLion(Lion newLion)
  { ...
  }
  public void AddLion(Lion newLion,
            int exhibitNumber) {
  ...
  }
}
```

Using Constructors

How to Initialize an Object

How to Overload a Constructor

How to Initialize an Object

- **Instance constructors are special methods that implement the actions required to initialize an object**
  - Have the same name as the name of the class
  - Default constructor takes no parameters

```
public class Lion {
  public Lion() {
     Console.WriteLine("Constructing Lion");
  }
}
```

- **Readonly**
  - Used to assign a value to a variable in the constructor

How to Overload a Constructor

Create multiple constructors that have the same name but different signatures

- Specify an initializer with **this**

```
public class Lion {
  private string   name;
  private int  age;

  public Lion() : this( "unknown", 0 ) {
      Console.WriteLine("Default: {0}", name);
  }
  public Lion( string theName, int theAge ) {
      name = theName;
      age = theAge;
      Console.WriteLine("Specified: {0}", name);
  }
}
```

Using Static Class Members

How to Use Static Class Members

How to Initialize a Class

How to Use Static Class Members

Static Members

- Belong to the class
- Initialize before an instance of the class is created
- Shared by all instances of the class

```
class Lion {
  public static string family = "felidae";
}
...
// A Lion object is not created in this code
Console.WriteLine( "Family: {0}", Lion.family );
```

How to Initialize a Class

Static Constructors

- Will only ever be executed once
- Run before the first object of that type is created
- Have no parameters
- Do not take an access modifier
- May co-exist with a class constructor
- Used to initialize a class