Introduction

- Encapsulation and abstraction are essential part of C# programming.

- Mostly used for hide complex code from unauthorized user and shows only relevant information.

- Encapsulation is the process of hiding irrelevant data from the user.

- Abstraction is just opposite of Encapsulation.

- Abstraction is mechanism to show only relevant data to the user.

Access Specifier

- It defines the scope of a class member.
- A class member can be variable or function.
- In C# there are **five types of access specifiers** are available:

  1. **Public.**
  2. **Private.**
  3. **Protected.**
  4. **Internal.**
  5. **Protected Internal.**

Public

- The class member, that is defined as public can be accessed by other class member that is initialized outside the class.

- A public member can be accessed from anywhere even outside the namespace.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Public_Access_Specifiers
{
   class access
   {
      // String Variable declared as public
      public string name;
      // Public method
      public void print()
      {
         Console.WriteLine("\nMy name is " + name);
      }
      }

      class Program
      {
         static void Main(string[] args)
         {
            access ac = new access();
            Console.Write("Enter your name:\t");
            // Accepting value in public variable that is outside the class
            ac.name = Console.ReadLine();
            ac.print();

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

Private

- The private access specifiers restrict the member variable or function to be called outside from the parent class.

- A private function or variable cannot be called outside from the same class. It hides its member variable and method from other class and methods.

- However, you can store or retrieve value from private access modifiers using get set property.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Private_Access_Specifiers
{
    class access
    {
        // String Variable declared as private
        private string name;
        public void print() // public method
        {
         Console.WriteLine("\nMy name is " + name);
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         access ac = new access();
         Console.Write("Enter your name:\t");
         // raise error because of its protection level
         ac.name = Console.ReadLine();
         ac.print();
         Console.ReadLine();
      }
   }
}
```

- Output will be:

Error 1: Private_Access_Specifiers.access.name' is inaccessible due to its protection level __

- In the example in the preivious, you cannot call name variable outside the class because it is declared as private.

Protected

- The protected access specifier hides its member variables and functions from other classes and objects.

- **Protected** variable or function can only be accessed in child class. It becomes very important while implementing inheritance.

```
namespace Protected_Specifier
{
    class access
    {
        // String Variable declared as protected
        protected string name;
        public void print()

        {
            Console.WriteLine("\nMy name is " + name);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            access ac = new access();
            Console.Write("Enter your name:\t");
            // raise error because of its protection level
            ac.name = Console.ReadLine();
            ac.print();
            Console.ReadLine();
        }
    }
}
```

- **Output**

‘Protected_Specifier.access.name’ is inaccessible due to its protection level

- This is because; the protected member can only be accessed within its **child class**.

- You can use protected access specifiers as follow:

```
namespace Protected_Specifier
{
    class access
    {
        // String Variable declared as protected
        protected string name;

            public void print()
            {
                Console.WriteLine("\nMy name is " + name);
            }
        }

        class Program : access // Inherit access class
        {
            static void Main(string[] args)
            {
                Program p = new Program();
                Console.Write("Enter your name:\t");
                p.name = Console.ReadLine(); // No Error!!
                p.print();
                Console.ReadLine();
            }
        }
    }
```

- The output will be:

Enter your name: Steven Clark

My name is Steven Clark

Internal

- The internal access specifier hides its member variables and methods from other classes and objects, that is resides in other namespace.
- The variable or classes that are declared with **internal** can be access by any member within application. It is the default access specifiers for a class in C# programming.

```
namespace Internal  Access  Specifier
    {
        class access
        {
            // String Variable declared as internal
            internal string name;
            public void print()
            {
                Console.WriteLine("\nMy name is " + name);
            }
        }

        class Program
        {
            static void Main(string[] args)
            {
                access ac = new access();
                Console.Write("Enter your name:\t");
                // Accepting value in internal variable
                ac.name = Console.ReadLine();
                ac.print();
                Console.ReadLine();
            }
        }
    }
```

Output :

Enter your name: Steven Clark

My name is Steven Clark

PROTECTED INTERNAL

- The protected internal access specifier allows its members to be accessed in derived class, containing class or classes within same application.

- However, this access specifier rarely used in C# programming but it becomes important while implementing inheritance.

```
namespace Protected_Internal
{
    class access
    {
        // String Variable declared as protected internal
        protected internal string name;
        public void print()
        {
            Console.WriteLine("\nMy name is " + name);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            access ac = new access();
            Console.Write("Enter your name:\t");
            // Accepting value in protected internal variable
            ac.name = Console.ReadLine();
            ac.print();
            Console.ReadLine();
        }
    }
}
```

Output:

Enter your name: Steven Clark

My name is Steven Clark

GET & SET Modifier

- The ***get set*** accessor or modifier mostly used for **storing and retrieving value from the private field**.

- The **get** accessor must **return a value** of property type, where **set** accessor **returns void.**

- The set accessor uses an **implicit parameter** called **value**.

GET & SET Modifier

- In simple words:

- The **get** method is used for **retrieving** value from ***private*** field

- The **set** method is used for **storing** value in ***private*** variables.

```
namespace Get_Set
{
    class access
    {
        // String Variable declared as private
        private static string name;
        public void print()
        {
            Console.WriteLine("\nMy name is " + name);
        }

        public string Name      //Creating Name property
        {
            get                 //get method for returning value
            {
                return name;
            }
            set         // set method for storing value in name field.
            {
                name = value;
            }
        }
    }
```

```
class Program
{
    static void Main(string[] args)
    {
        access ac = new access();
        Console.Write("Enter your name:\t");
        // Accepting value via Name property


            ac.Name = Console.ReadLine();
            ac.print();
            Console.ReadLine();
        }
    }
}
```

Output:

Enter your name: Steven Clark

My name is Steven Clark **Access**

Access Modifier	Accessibility
Public	Anywhere. No restrictions.
Private	Only in the containing class.
Protected	Within the containing class and to the classes that derive from the containing class.
Internal	Anywhere within the containing assembly.
Protected Internal	Anywhere within the containing assembly and from within a derived class in any another assembly.

1) Check Exercises
2) Solve Home Work

The File is in Profile

INHERITANCE

- Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.

- This also **provides an opportunity to reuse the code functionality and speeds up implementation time.**

- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class.

- This ***existing class*** is called the **base class**, and the ***new class*** is referred to as the **derived class**.

Base and Derived Classes

- A **class** can be derived from more than one class or interface, which means that it can **inherit data and functions** from **multiple base** classes or interfaces.
- To create a derived class in C#:
  - you enter the name of the class,
  - followed by a colon **:**
  - and the name of the base class.

Base and Derived Classes

- The syntax used in C# for creating **derived classes** is as follows:

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

Example:

Consider a base class Shape and its derived class Rectangle:

```
using System ;
namespace InheritanceApplication
{
        class Shape
        {
                public void setWidth(int w)
                {
                width = w;
                }
                public void setHeight(int h)
                {
                height = h;
                }
                protected int width;
                protected int height;
        }
```

```
// Derived class
class Rectangle: Shape
{
        public int getArea()
        {
        return (width * height);
        }
}
class RectangleTester
{
        static void Main(string[] args)
        {
                Rectangle Rect = new Rectangle();
                Rect.setWidth(5);
                Rect.setHeight(7);
                // Print the area of the object.
                Console.WriteLine("Total area: {0}", Rect.getArea());
                Console.ReadKey();
        }
}
}
```

When the code in previous slide is compiled and executed, it produces the following result:

Total area: 35

Multiple Inheritance

- C# **does not support** multiple inheritance.

- However, you can use **interfaces** to implement **multiple inheritance**.

The following program demonstrates this:

```
using System ;
namespace InheritanceApplication
{
        class Shape
        {
                public void setWidth(int w)
                {
                        width = w;
                }
                public void setHeight(int h)
                {
                        height = h;
                }
                protected int width;
                protected int height;
        }
```

```
// Base class PaintCost
public interface PaintCost
{
        int getCost(int area);
}
// Derived class
class Rectangle : Shape, PaintCost
{
        public int getArea()
        {
                return (width * height);
        }
        public int getCost(int area)
        {
                return area * 70;
        }
}
```

```
class RectangleTester
{
        static void Main(string[] args)
        {
                Rectangle Rect = new Rectangle();
                int area;
                Rect.setWidth(5);
                Rect.setHeight(7);
                area = Rect.getArea();
                // Print the area of the object.
                Console.WriteLine("Total area: {0}", Rect.getArea());
                Console.WriteLine("Total paint cost: $ {0}" ,
                Rect.getCost(area));
                Console.ReadKey();
        }
}
}
```

- When the code in previous slides is compiled and executed, it produces the following result:

Total area: 35

Total paint cost: $ 2450