31927 32998:
Application Development with .NET

Week-6 Lecture

Programming in C#

Part-5

Outline

- Inheritance
- Method Overriding
- Method Hiding
- Abstract Class and methods
- Sealed Classes and Methods
- Interfaces
- .Net Standard Interfaces

Inheritance

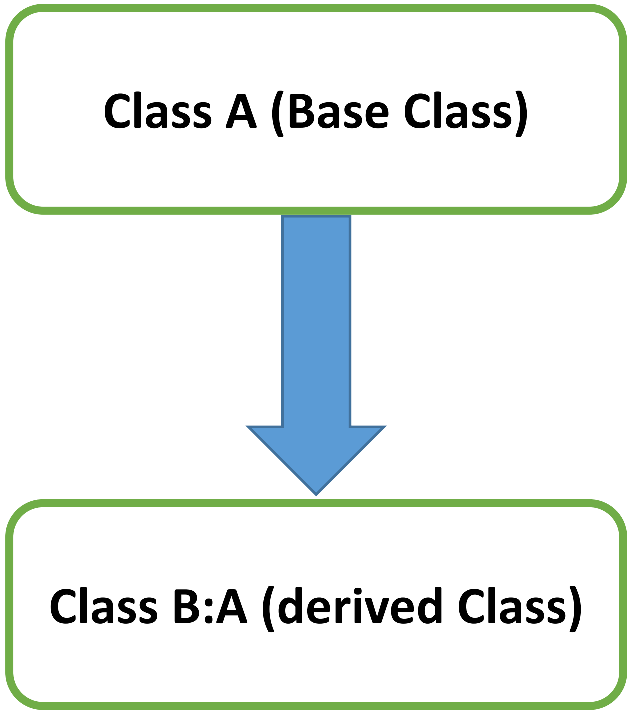
- Acquiring the properties of one class into another class is Inheritance.
- Provides reusability by allowing to reuse and extend existing class
- C# support single inheritance
- All classes ultimately derive from System.Object

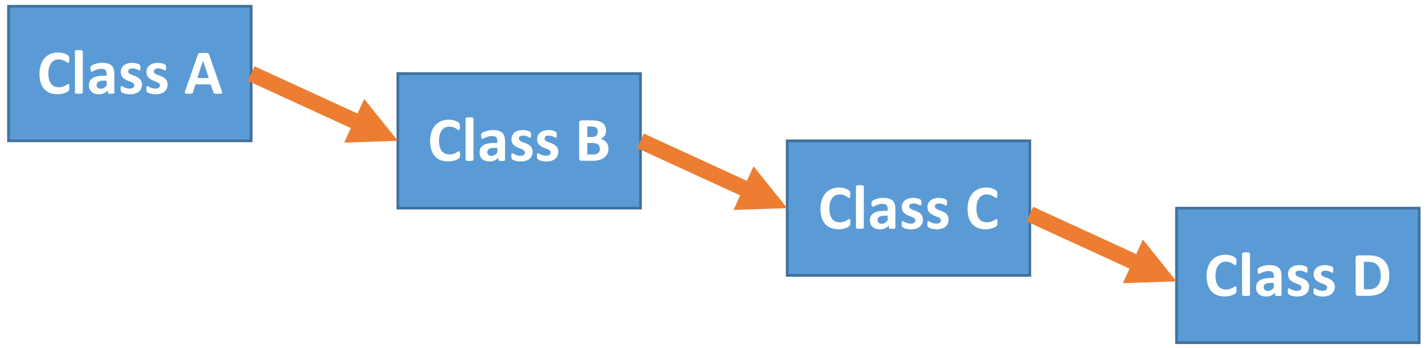
Inheritance

- Not all members are inherited by the derived class, the following members are not inherited:
  1. *Static constructors*: which initializes the static data of a class
  2. *Instance constructors*: Called to create a new instance of the class. Each class must define its own constructors.
  3. *Finalizers*
- Other members of the base class are inherited by the derived class based on their visibility.

Single Inheritance

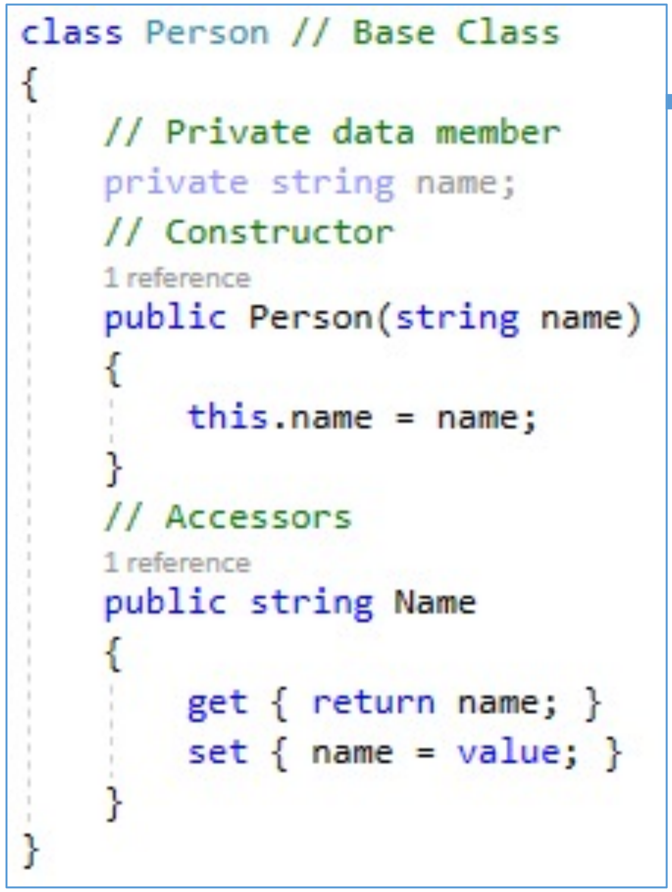
- One base class (parent class) and one derived class (child class)
- Inheritance is transitive, members of base class are available to child class



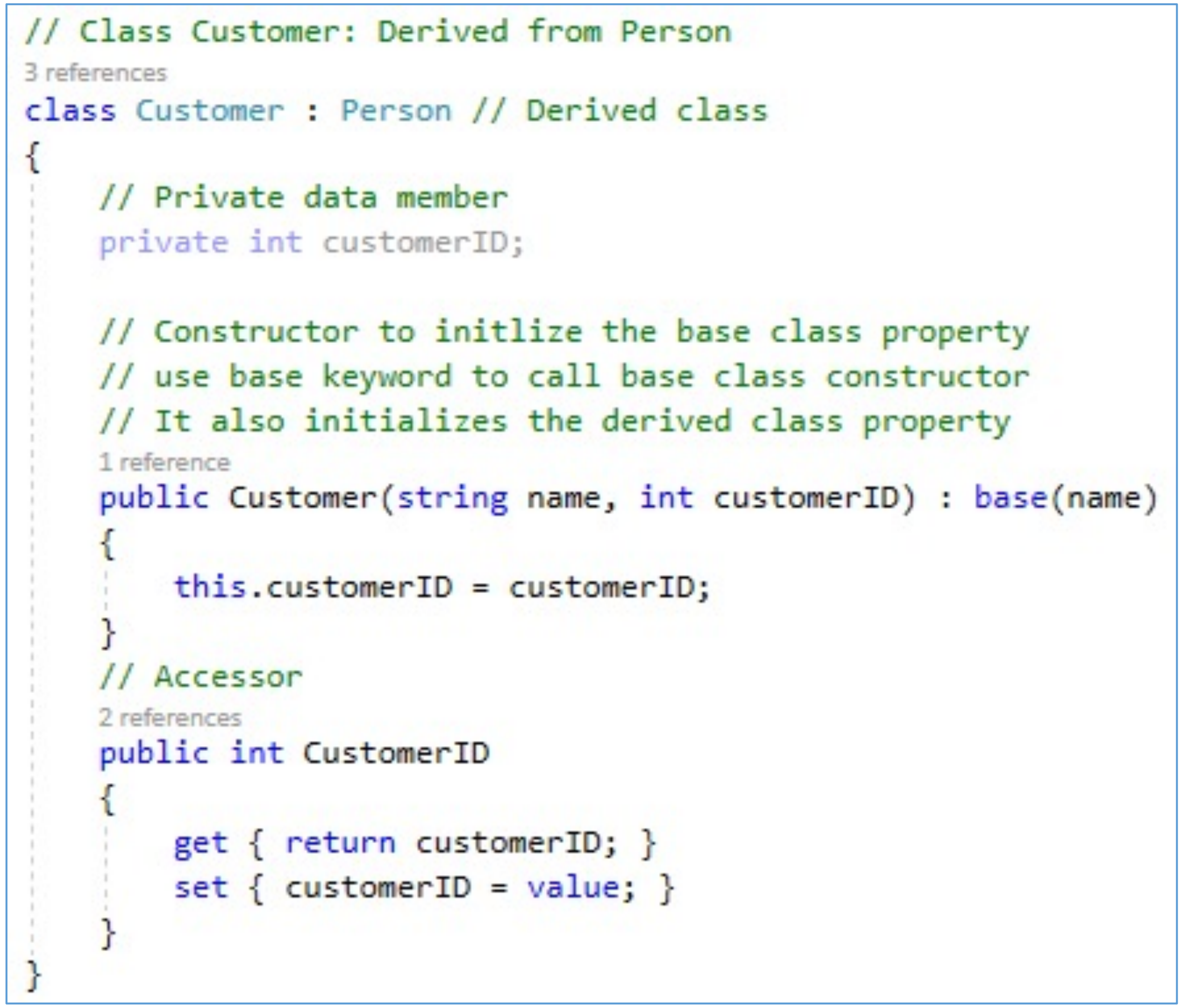


Single Inheritance

Example:



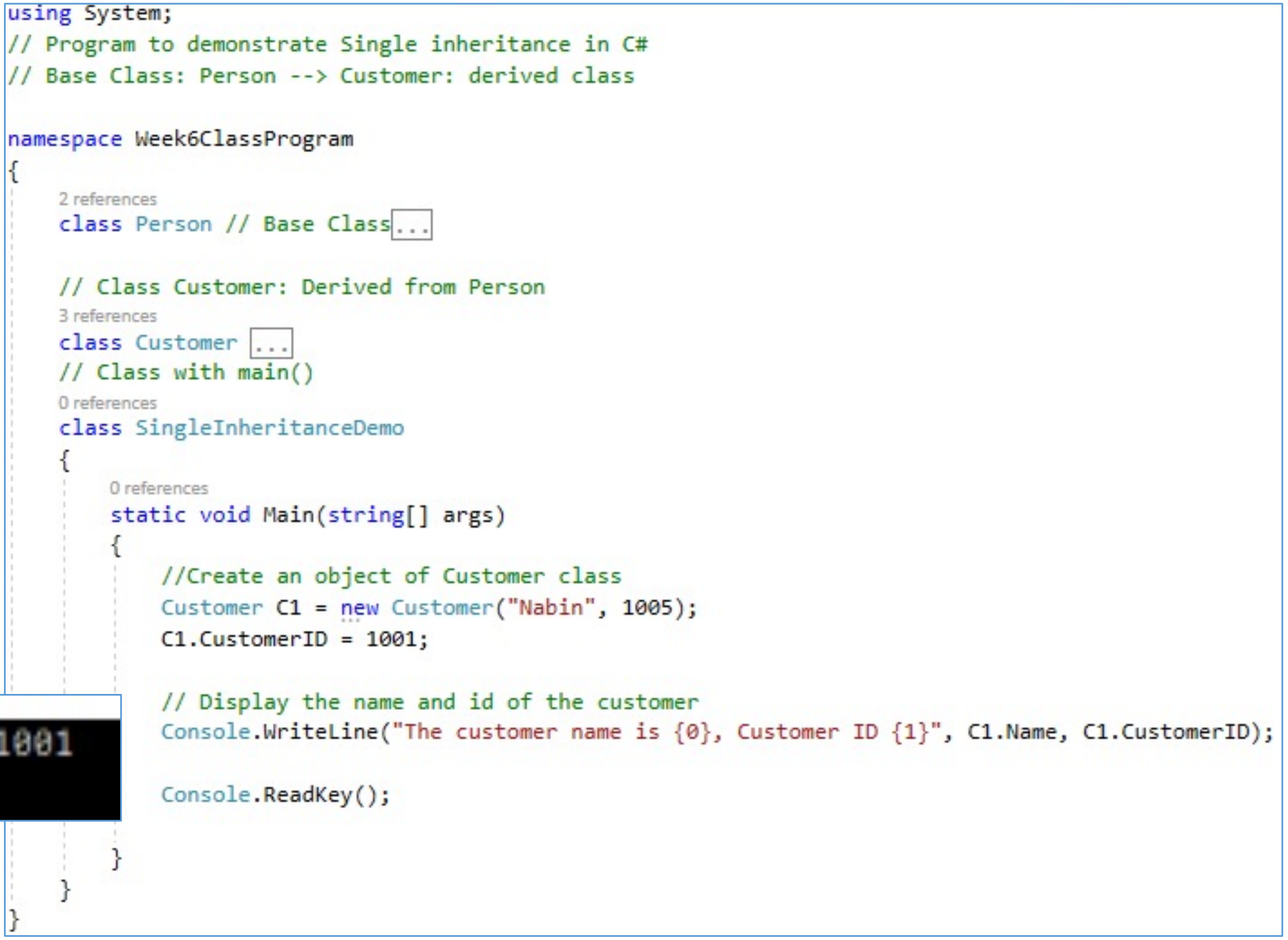
```
class Person // Base Class
{
    // Private data member
    private string name;
    // Constructor
    1 reference
    public Person(string name)
    {
        this.name = name;
    }
    // Accessors
    1 reference
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```



```
// Class Customer: Derived from Person
3 references
class Customer : Person // Derived class
{
    // Private data member
    private int customerID;

    // Constructor to initlize the base class property
    // use base keyword to call base class constructor
    // It also initializes the derived class property
    1 reference
    public Customer(string name, int customerID) : base(name)
    {
        this.customerID = customerID;
    }
    // Accessor
    2 references
    public int CustomerID
    {
        get { return customerID; }
        set { customerID = value; }
    }
}
```

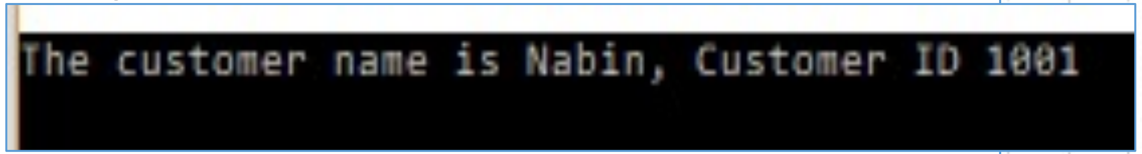
Single Inheritance

Example:



```
using System;
// Program to demonstrate Single inheritance in C#
// Base Class: Person --> Customer: derived class

namespace Week6ClassProgram
{
    2 references
    class Person // Base Class[...]

    // Class Customer: Derived from Person
    3 references
    class Customer [...]
    // Class with main()
    0 references
    class SingleInheritanceDemo
    {
        0 references
        static void Main(string[] args)
        {
            //Create an object of Customer class
            Customer C1 = new Customer("Nabin", 1005);
            C1.CustomerID = 1001;

            // Display the name and id of the customer
            Console.WriteLine("The customer name is {0}, Customer ID {1}", C1.Name, C1.CustomerID);

            Console.ReadKey();

        }
    }
}
```

Output:



```
The customer name is Nabin, Customer ID 1001
```

Member Visibility: Private/Protected/Public

1. *Private*:

   - member are only visible in the derived classes that are nested in their base class.

   - Else they are not visible.

2. *Protected*:

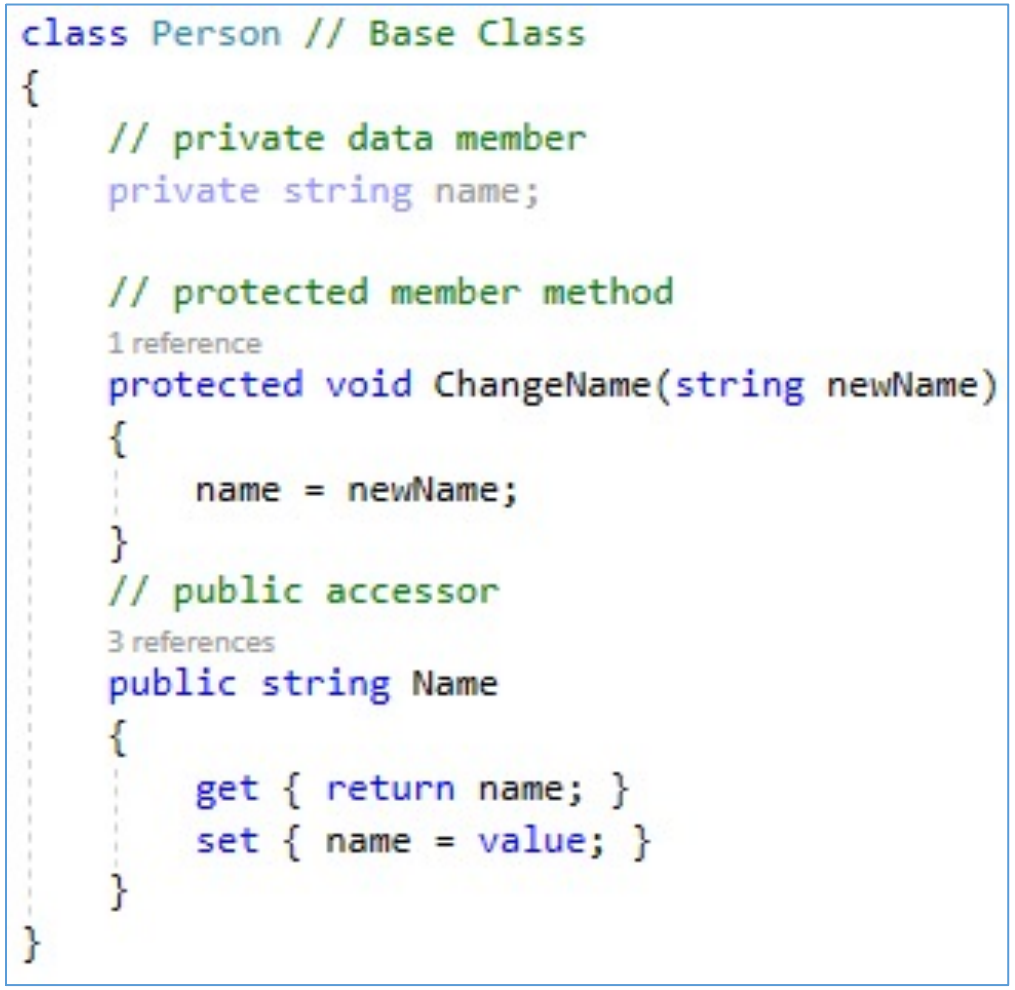
   - Members are only visible in the derived classes.

3. *Public*:

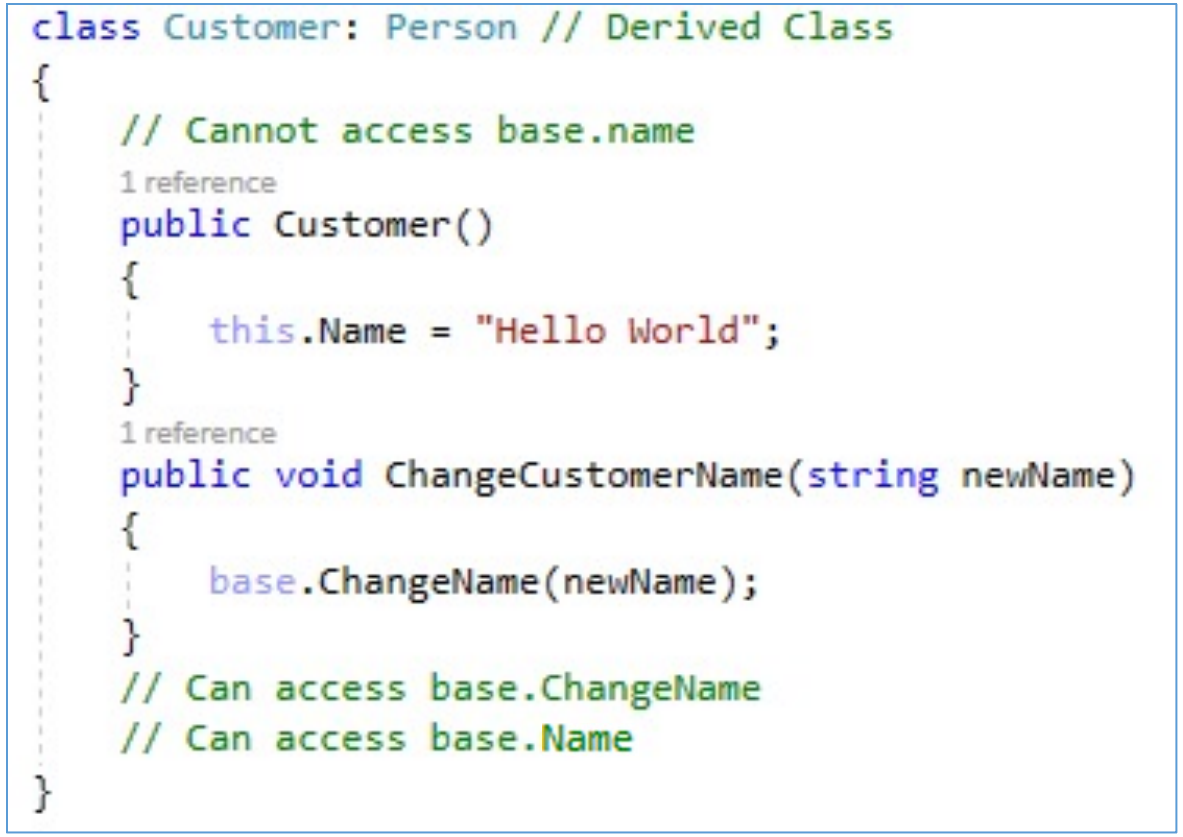
   - Members are visible in derived classes and are part of the derived class interface.

Member Visibility: Private/Protected/Public

Example:



```
class Person // Base Class
{
    // private data member
    private string name;

    // protected member method
    1 reference
    protected void ChangeName(string newName)
    {
        name = newName;
    }
    // public accessor
    3 references
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

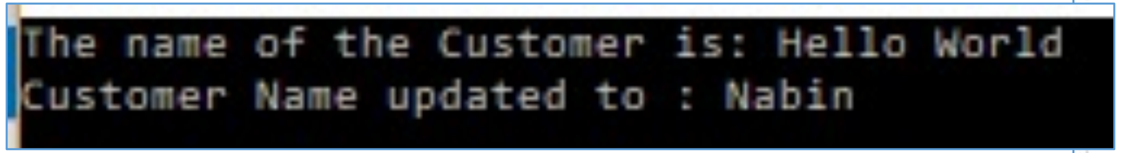


```
class Customer: Person // Derived Class
{
    // Cannot access base.name
    1 reference
    public Customer()
    {
        this.Name = "Hello World";
    }
    1 reference
    public void ChangeCustomerName(string newName)
    {
        base.ChangeName(newName);
    }
    // Can access base.ChangeName
    // Can access base.Name
}
```

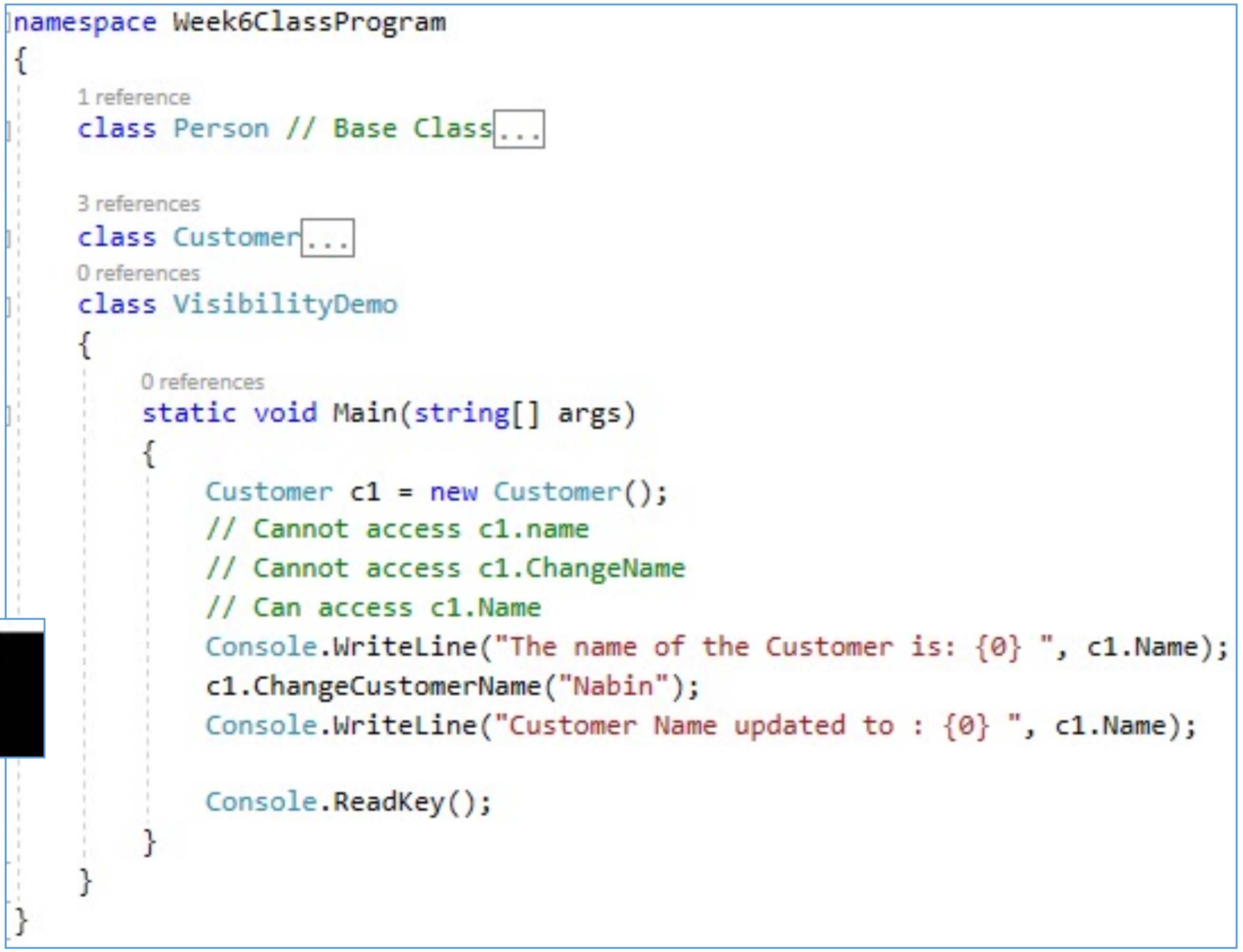
Member Visibility: Private/Protected/Public

Example:

Output:



```
The name of the Customer is: Hello World
Customer Name updated to : Nabin
```



```
namespace Week6ClassProgram
{
    1 reference
    class Person // Base Class...

    3 references
    class Customer...
    0 references
    class VisibilityDemo
    {
        0 references
        static void Main(string[] args)
        {
            Customer c1 = new Customer();
            // Cannot access c1.name
            // Cannot access c1.ChangeName
            // Can access c1.Name
            Console.WriteLine("The name of the Customer is: {0} ", c1.Name);
            c1.ChangeCustomerName("Nabin");
            Console.WriteLine("Customer Name updated to : {0} ", c1.Name);

            Console.ReadKey();
        }
    }
}
```

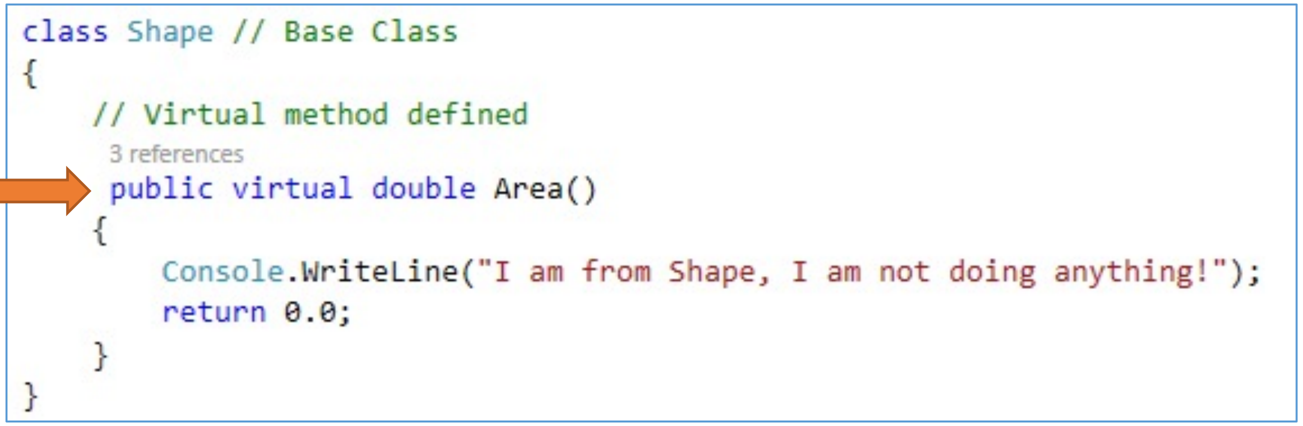
Method Overriding

- Method name and signature is same name in both base class and subclass.
- For method overriding:
  - Use virtual keyword in the ***base*** class
  - Use override keyword in the ***sub*** class, to override the implementation of a method inherited from base class.

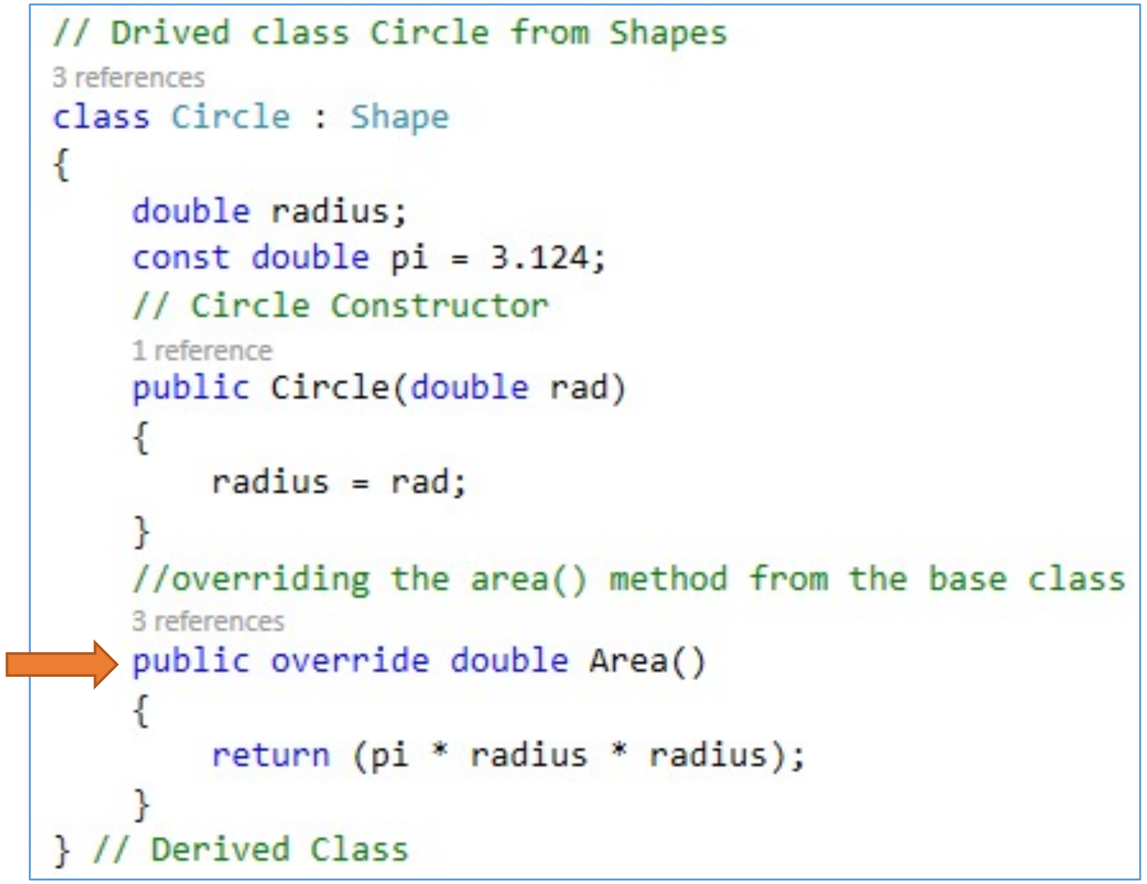
  - By default, methods are non-virtual. You cannot override a non-virtual method.

Method Overriding

Example:



```
class Shape // Base Class
{
    // Virtual method defined
     3 references
     public virtual double Area()
    {
        Console.WriteLine("I am from Shape, I am not doing anything!");
        return 0.0;
    }
}
```

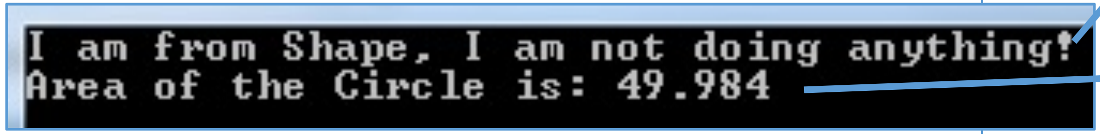


```
// Drived class Circle from Shapes
3 references
class Circle : Shape
{
    double radius;
    const double pi = 3.124;
    // Circle Constructor
    1 reference
    public Circle(double rad)
    {
        radius = rad;
    }
    //overriding the area() method from the base class
    3 references
    public override double Area()
    {
        return (pi * radius * radius);
    }
} // Derived Class
```

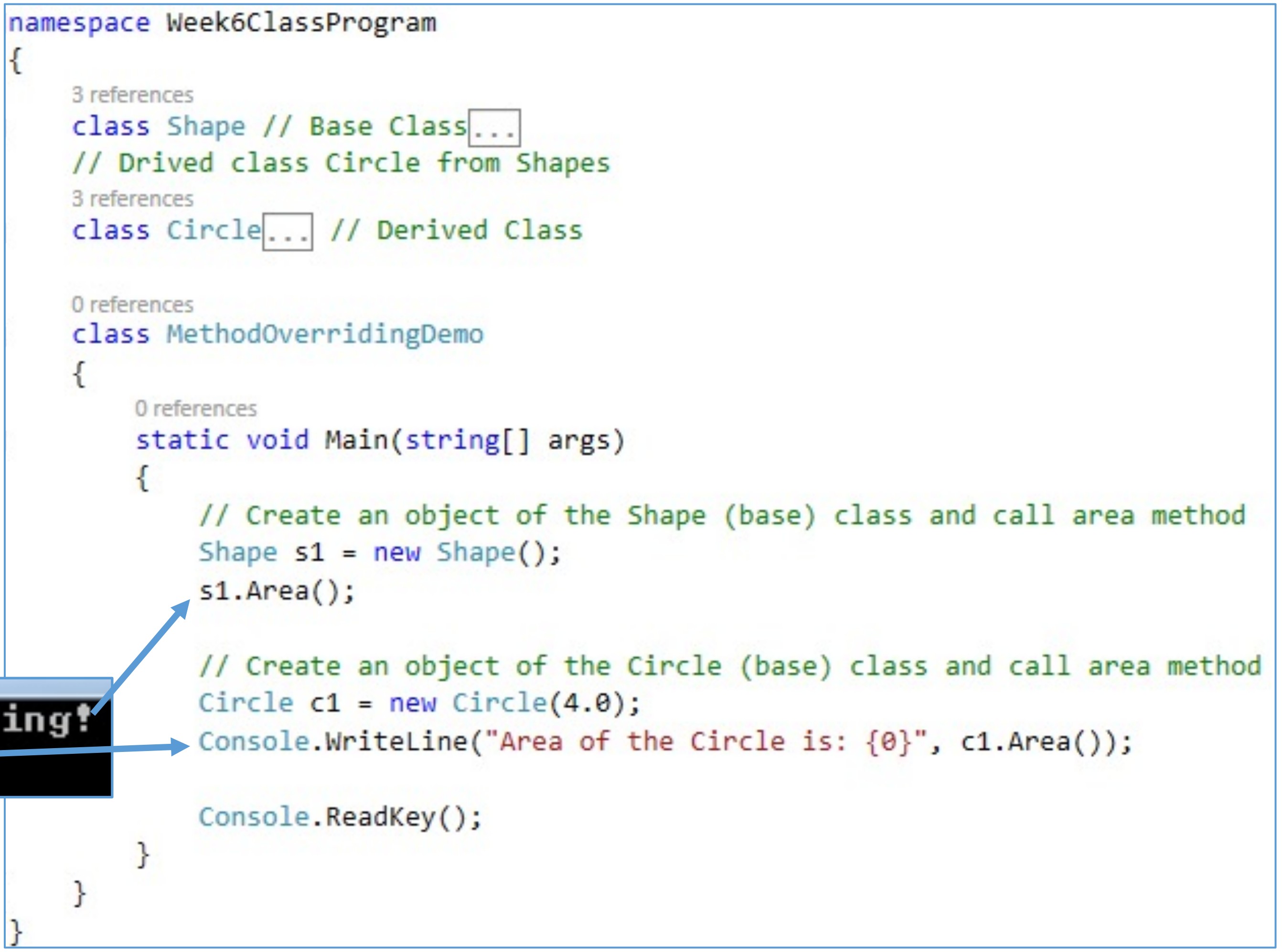
Method Overriding

Example:

Output:



```
I am from Shape, I am not doing anything!
Area of the Circle is: 49.984
```



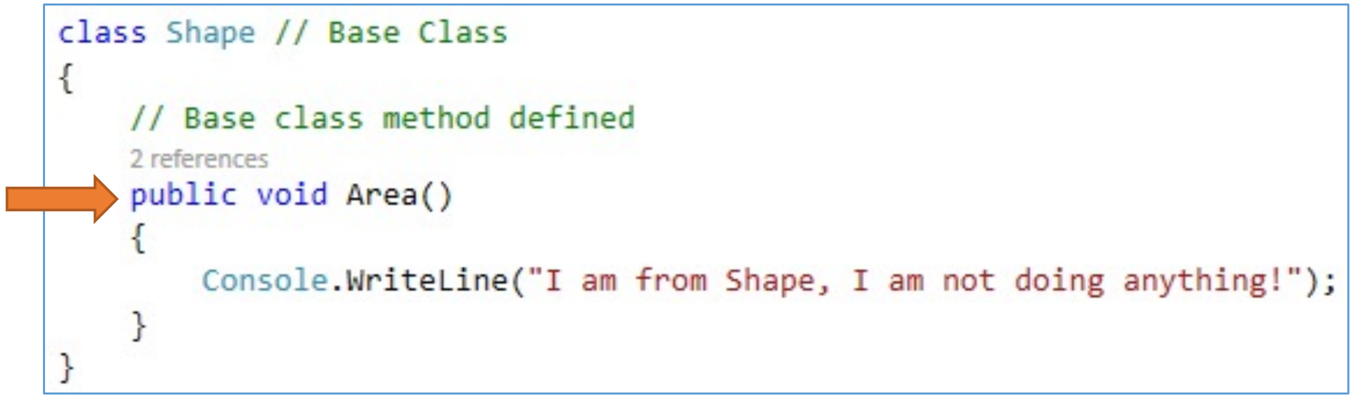
```
namespace Week6ClassProgram
{
    3 references
    class Shape // Base Class...
    // Drived class Circle from Shapes
    3 references
    class Circle... // Derived Class

    0 references
    class MethodOverridingDemo
    {
        0 references
        static void Main(string[] args)
        {
            // Create an object of the Shape (base) class and call area method
            Shape s1 = new Shape();
            s1.Area();

            // Create an object of the Circle (base) class and call area method
            Circle c1 = new Circle(4.0);
            Console.WriteLine("Area of the Circle is: {0}", c1.Area());

            Console.ReadKey();
        }
    }
}
```

Method Hiding

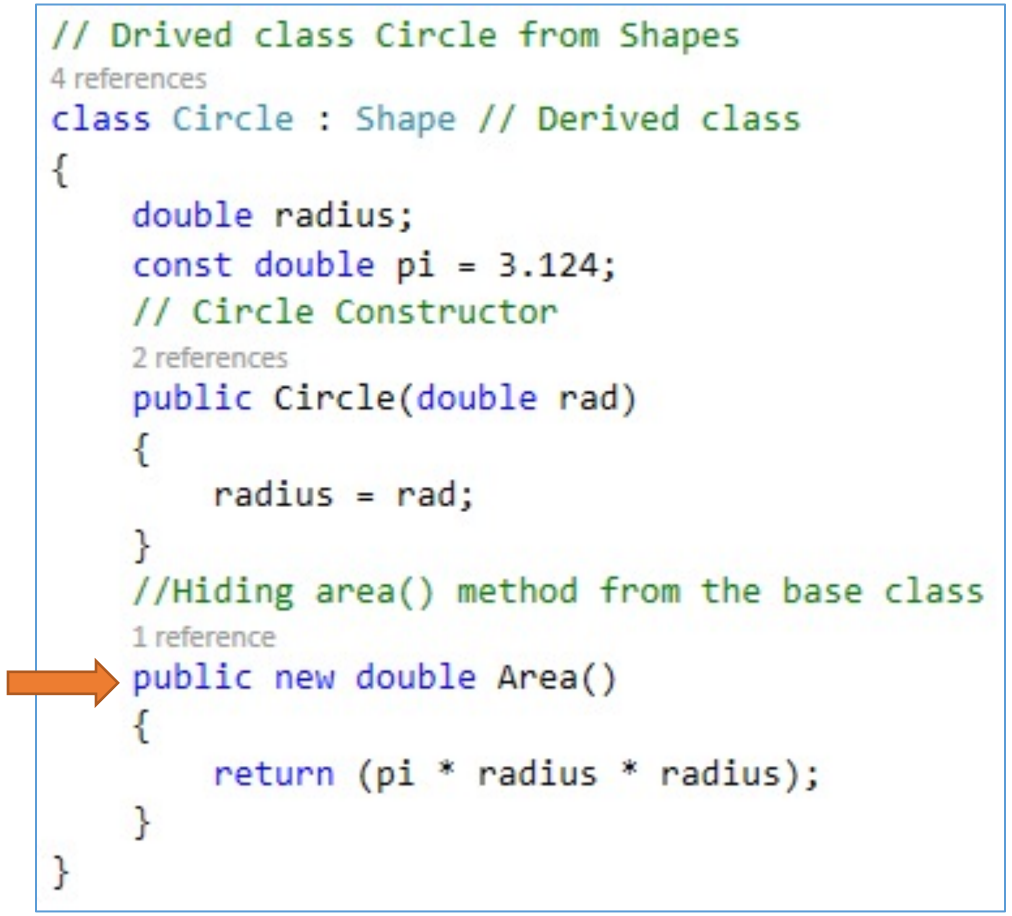
- Subclass method “hides” base class method.
- Method called depends on type of variable used to refer to the instance rather than the type of the instance itself
- No polymorphism
- To explicitly hide base class method use new

Method Hiding

Example:



```
class Shape // Base Class
{
    // Base class method defined
    2 references
    public void Area()
    {
        Console.WriteLine("I am from Shape, I am not doing anything!");
    }
}
```

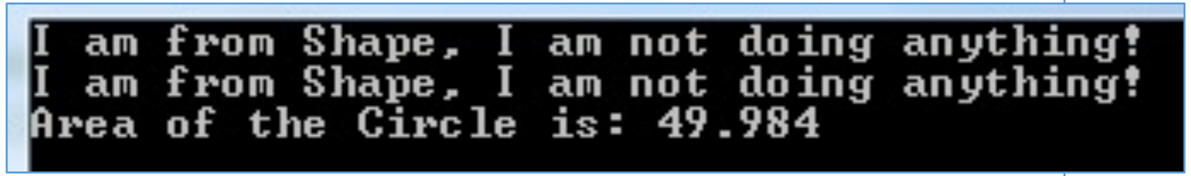


```
// Drived class Circle from Shapes
4 references
class Circle : Shape // Derived class
{
    double radius;
    const double pi = 3.124;
    // Circle Constructor
    2 references
    public Circle(double rad)
    {
        radius = rad;
    }
    //Hiding area() method from the base class
    1 reference
    public new double Area()
    {
        return (pi * radius * radius);
    }
}
```

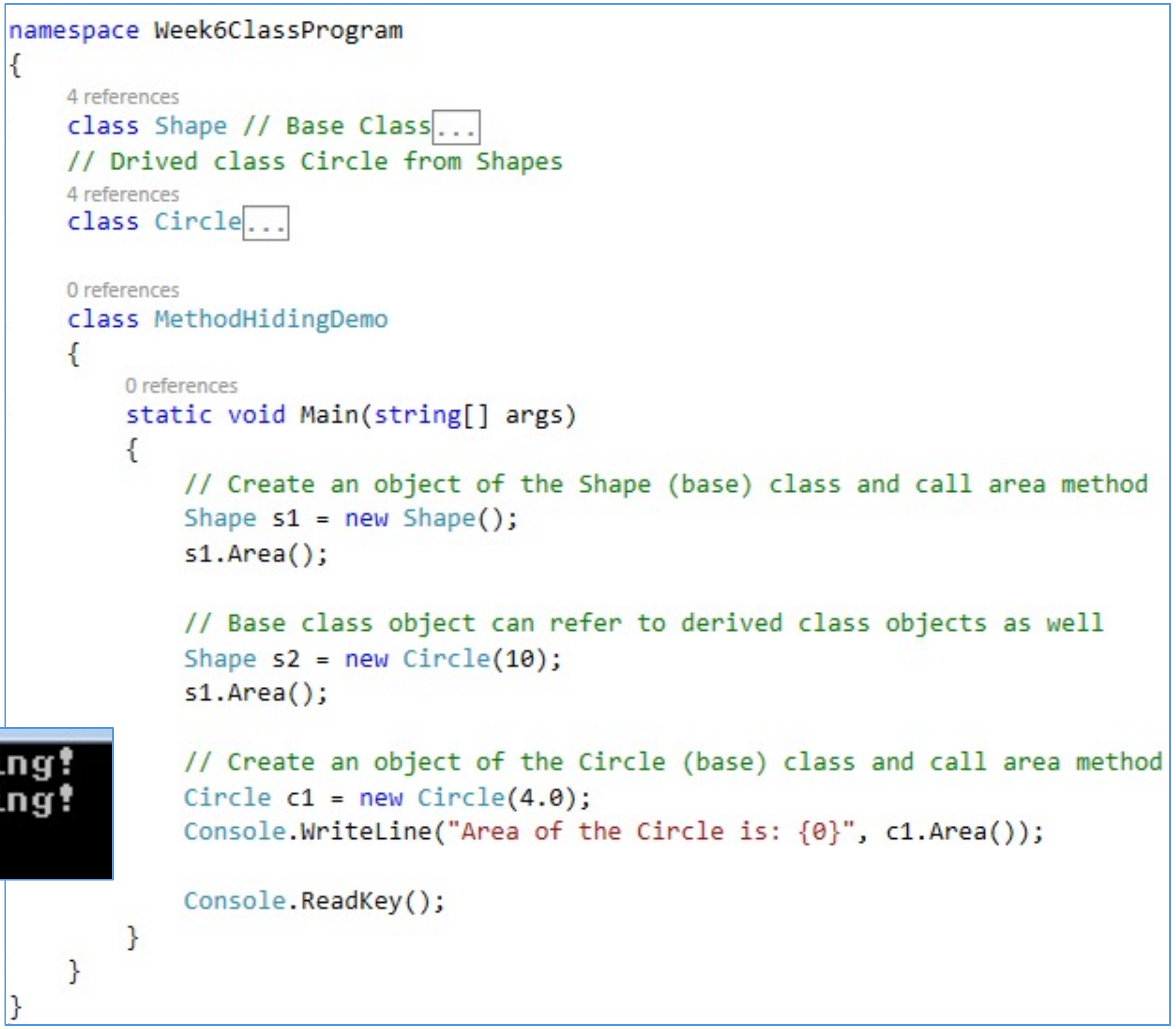
Method Hiding

Example:

Output:



```
I am from Shape, I am not doing anything!
I am from Shape, I am not doing anything!
Area of the Circle is: 49.984
```



```
namespace Week6ClassProgram
{
    4 references
    class Shape // Base Class...
    // Drived class Circle from Shapes
    4 references
    class Circle...

    0 references
    class MethodHidingDemo
    {
        0 references
        static void Main(string[] args)
        {
            // Create an object of the Shape (base) class and call area method
            Shape s1 = new Shape();
            s1.Area();

            // Base class object can refer to derived class objects as well
            Shape s2 = new Circle(10);
            s1.Area();

            // Create an object of the Circle (base) class and call area method
            Circle c1 = new Circle(4.0);
            Console.WriteLine("Area of the Circle is: {0}", c1.Area());

            Console.ReadKey();
        }
    }
}
```

Abstract Classes

- The abstract modifier indicates that the thing being modified has incomplete/missing implementation.
- Cannot be instantiated
- Contain abstract methods for implementation in derived classes
- Abstract methods are implicitly virtual
- Abstract method are only permitted in Abstract class
- A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

Abstract Classes

```
abstract class Person
{
   private string name;

   abstract string TransformName(string name);
}
```

No implementation provided

```
class Customer : Person
{
   public string TransformName(string foo)
   { ... }
}
```

Child Class must provide implementation for all inherited abstract methods

Sealed Classes

- Sealed modifier prevents other classes from inheriting from it.
- May not be inherited from
- Not common, but may be used in class libraries or commercial situations
- Methods may be declared sealed to prevent overriding

```
sealed class MyClass
{
   // ...
}
```

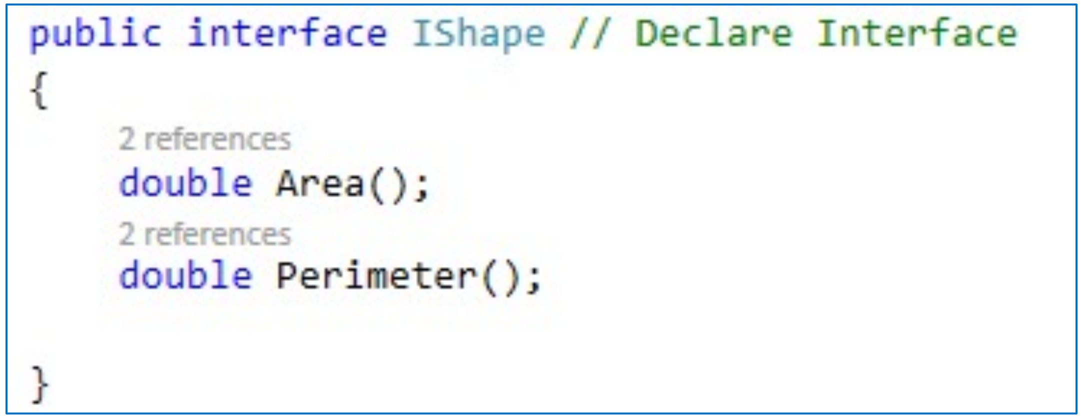
```
public sealed override void SomeMethod()
{
   // ...
}
```

Interfaces

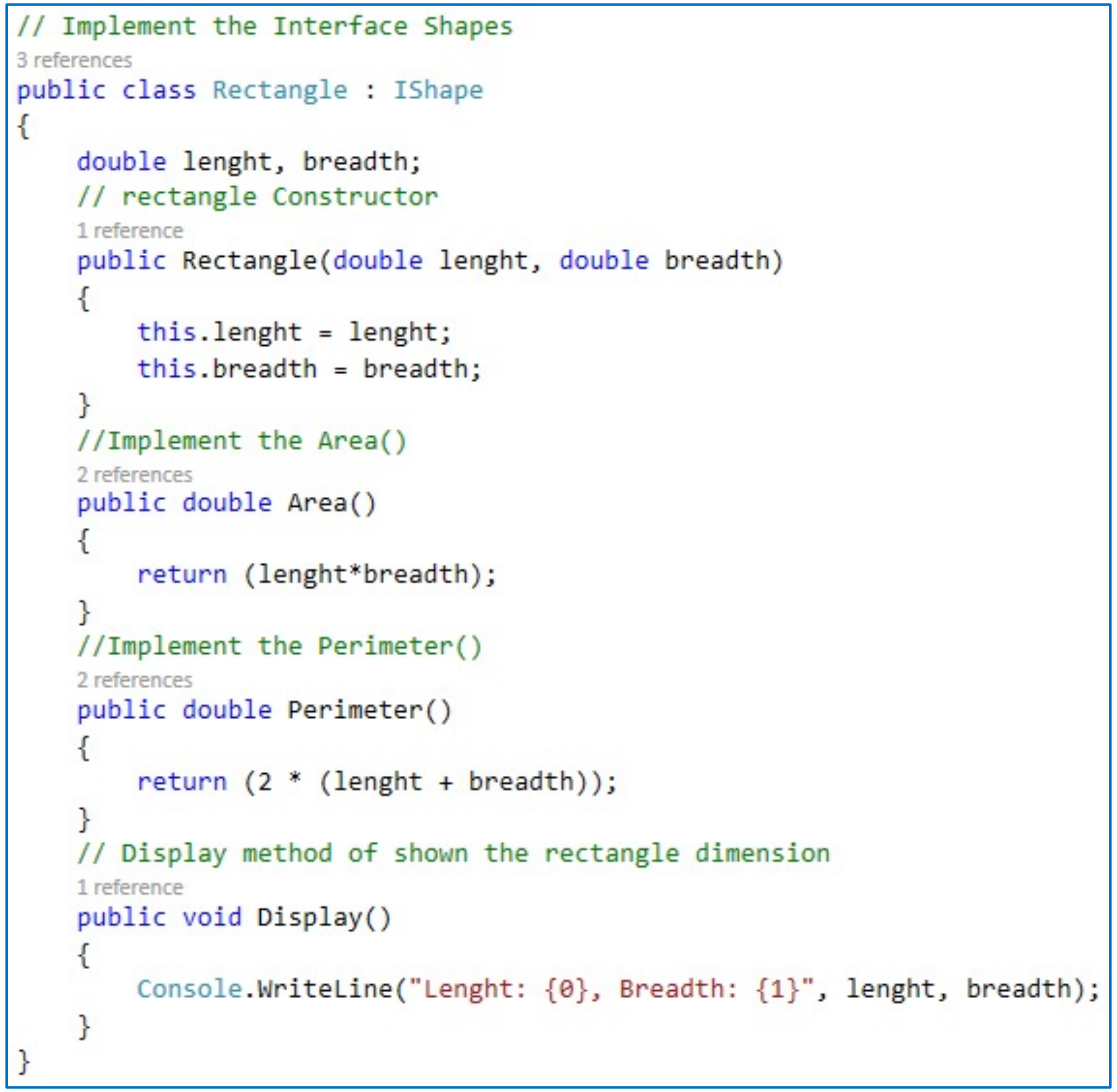
- An interface defines a set of methods and properties but none of them are implemented.
- An interface contains only the signatures of methods, properties, events or indexers
- In an interface, every method is implicitly `public` and no explicit access specifier is allowed.
- Classes which are derived from the interface must implement every method.
- Classes may implement multiple interfaces.

Interfaces

Example:



```
public interface IShape // Declare Interface
{
    2 references
    double Area();
    2 references
    double Perimeter();

}
```

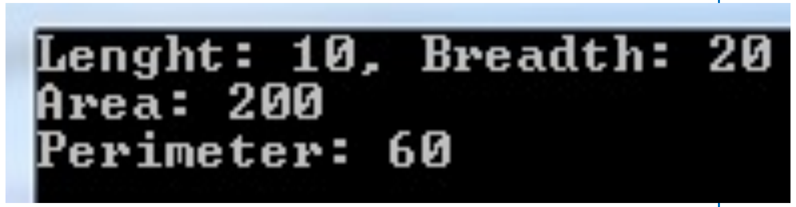


```
// Implement the Interface Shapes
3 references
public class Rectangle : IShape
{
    double lenght, breadth;
    // rectangle Constructor
    1 reference
    public Rectangle(double lenght, double breadth)
    {
        this.lenght = lenght;
        this.breadth = breadth;
    }
    //Implement the Area()
    2 references
    public double Area()
    {
        return (lenght*breadth);
    }
    //Implement the Perimeter()
    2 references
    public double Perimeter()
    {
        return (2 * (lenght + breadth));
    }
    // Display method of shown the rectangle dimension
    1 reference
    public void Display()
    {
        Console.WriteLine("Lenght: {0}, Breadth: {1}", lenght, breadth);
    }
}
```

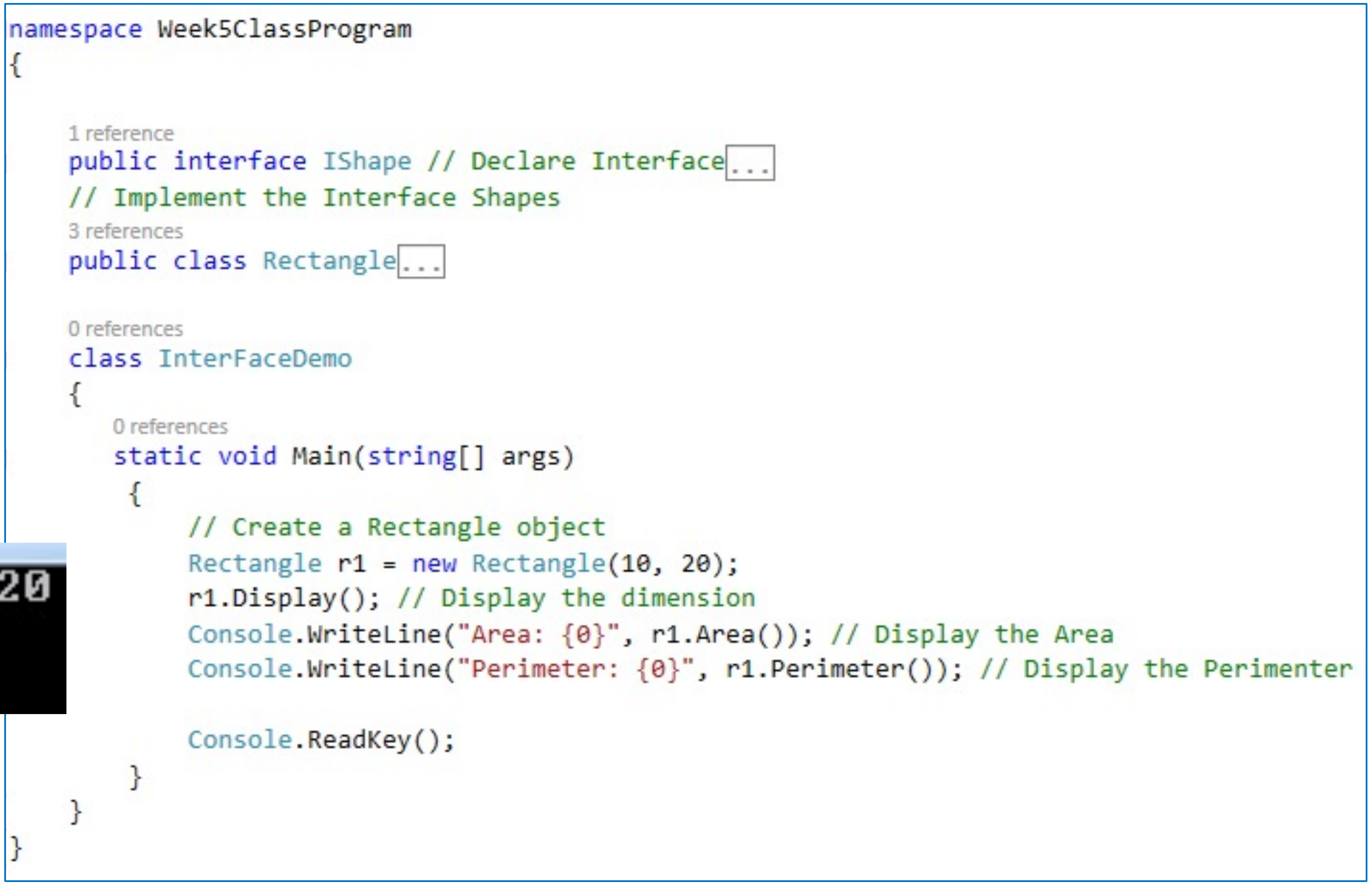
Interfaces

Example:

Output:



```
Lenght: 10, Breadth: 20
Area: 200
Perimeter: 60
```



```
namespace Week5ClassProgram
{

    1 reference
    public interface IShape // Declare Interface...
    // Implement the Interface Shapes
    3 references
    public class Rectangle...

    0 references
    class InterFaceDemo
    {
        0 references
        static void Main(string[] args)
         {
            // Create a Rectangle object
            Rectangle r1 = new Rectangle(10, 20);
            r1.Display(); // Display the dimension
            Console.WriteLine("Area: {0}", r1.Area()); // Display the Area
            Console.WriteLine("Perimeter: {0}", r1.Perimeter()); // Display the Perimenter

            Console.ReadKey();
         }
    }
}
```

.Net Standard Interfaces

- The .NET Framework defines a large number of interfaces.
- The three that you will most commonly come across are
  - System.IComparable
  - System.IEnumerable
  - System.ICollections
- Any object that is IComparable requires that the object has declared the following public method
  - int CompareTo(object obj)
  - ComparableEx.cs
- The foreach command can be used on any object that implements the IEnumerable interface. It must implement the following public method
  - IEnumerator GetEnumerator()
- We will examine collections later.