C# Fundamentals

Part 2

Classes
&
Objects

Classes:

- Classes are essentially templates from which you create objects.
- In C# .NET programming, everything you deal with involves classes and objects.
- The syntax of a class definition is:

    < access_modifiers > class Class_Name

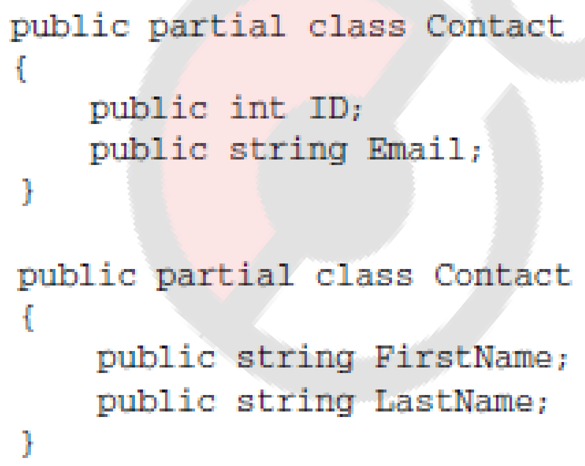
    {

    //--- Fields, properties, methods, and events ---

    }

Using Partial Classes

Instead of defining an entire class by using the class keyword, you can split the definition into multiple classes by using the partial keyword.



```
public partial class Contact
{
    public int ID;
    public string Email;
}

public partial class Contact
{
    public string FirstName;
    public string LastName;
}
```

Creating an instance of the class:

- To instantiate the Contact class defined earlier, you first create a variable of type Contact :

```
Contact contact1;
```

- Then instantiate it:

```
contact1 = new Contact();
```

- Once instantiated we can assign values to variables defined in it.

```
contact1.ID = 12;
contact1.FirstName = “Wei-Meng”;
contact1.LastName = “Lee”;
contact1.Email = “weimenglee@learn2develop.net”;
```

- You can also assign an object to an object, like the following:

```
Contact contact1 = new Contact();
Contact contact2 = contact1;
```

In these statements, contact2 and contact1 are now both pointing to the same object. Any changes made to one object will be reflected in the other object.

Anonymous types:

Anonymous types enable you to define data types without having to formally define a class.

```
var book1 = new
{
    ISBN = "978-0-470-17661-0",
    Title="Professional Windows Vista Gadgets Programming",
    Author = "Wei-Meng Lee",
    Publisher="Wrox"
};
```

C# anonymous types are immutable, which means all the properties are read-only — their values cannot be changed once they are initialized.

Class Members:

Instance Members

Static Members

Note**: Constants defined within a class are implicitly(by default) static

Access Modifiers:

- *C# has five access modifiers — private , public , protected , internal and protected internal .*

Inheritance

Inheritance is a property of OOPs in which the features and behavior of a class are derived from another class.

The class whose members are inherited is called the base class.

The class which inherits the members from the base class is called the derived class.

Types of inheritance:-

- Single Inheritance
- Multi-level inheritance
- Hierarchical inheritance
- Hybrid inheritance
- Multipath inheritance (not supported in C#)
- Multiple inheritance (interfaces)

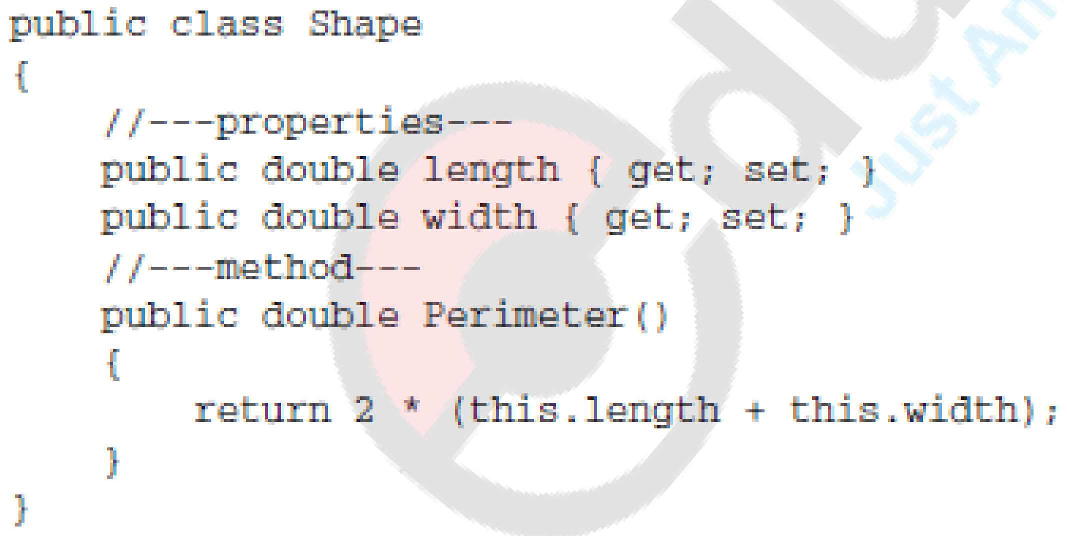
Inheritance

- Inheritance facilitates code reuse and allows you to extend the functionality of code that you have already written.

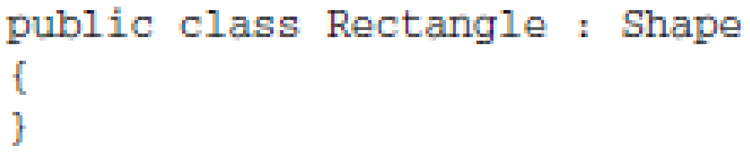
- In object - oriented programming, inheritance is classified into two types: implementation and interface.

- Implementation Inheritance:

  Implementation inheritance is when a class derives from another base class, inheriting all the base class ’ s members



```
public class Shape
{
    //---properties---
    public double length { get; set; }
    public double width { get; set; }
    //---method---
    public double Perimeter()
    {
        return 2 * (this.length + this.width);
    }
}
```

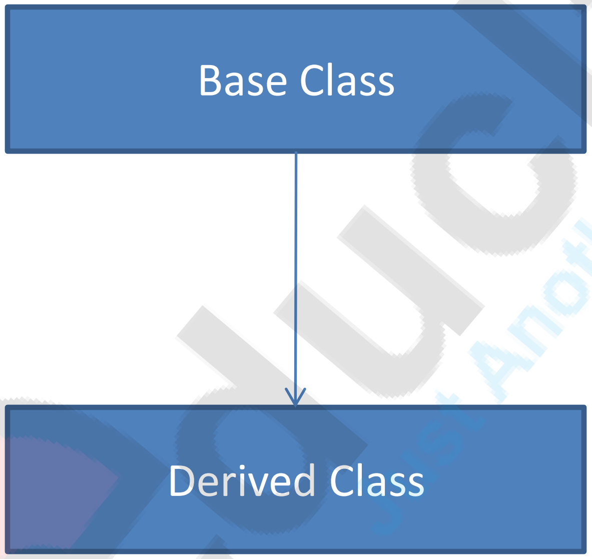


```
public class Rectangle : Shape
{
}
```

- C# supports only single - class inheritance
- If you do not specify the base class, the C# compiler assumes that it is inheriting from the System.Object class

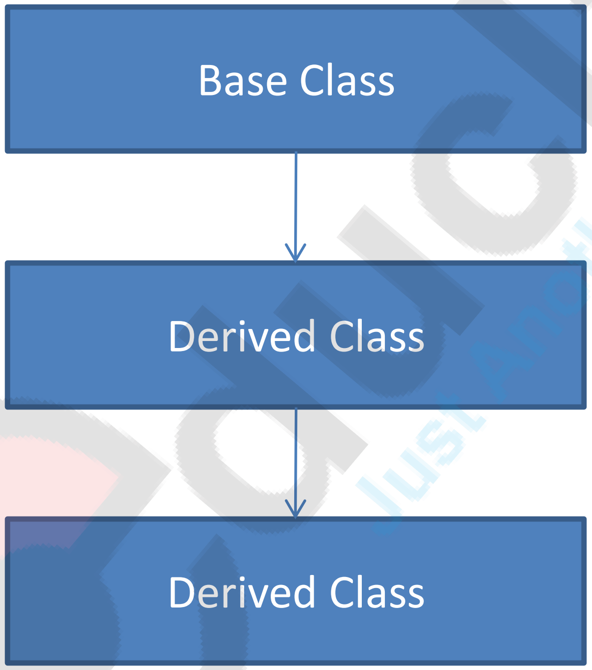
Inheritance

- Single Inheritance – when there is only one base class and one derived class.



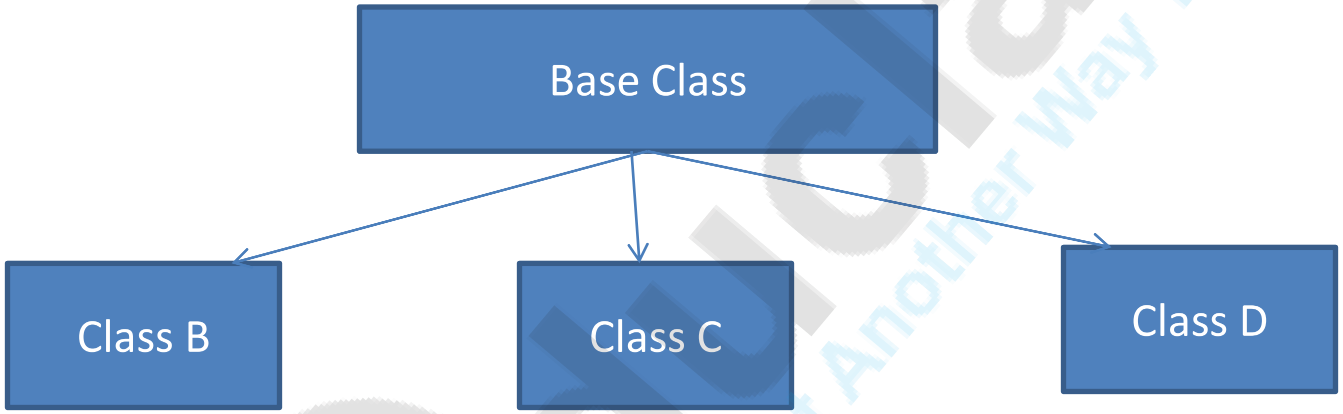
Inheritance

- Multilevel Inheritance – when a derived class becomes the base class to another derived class.



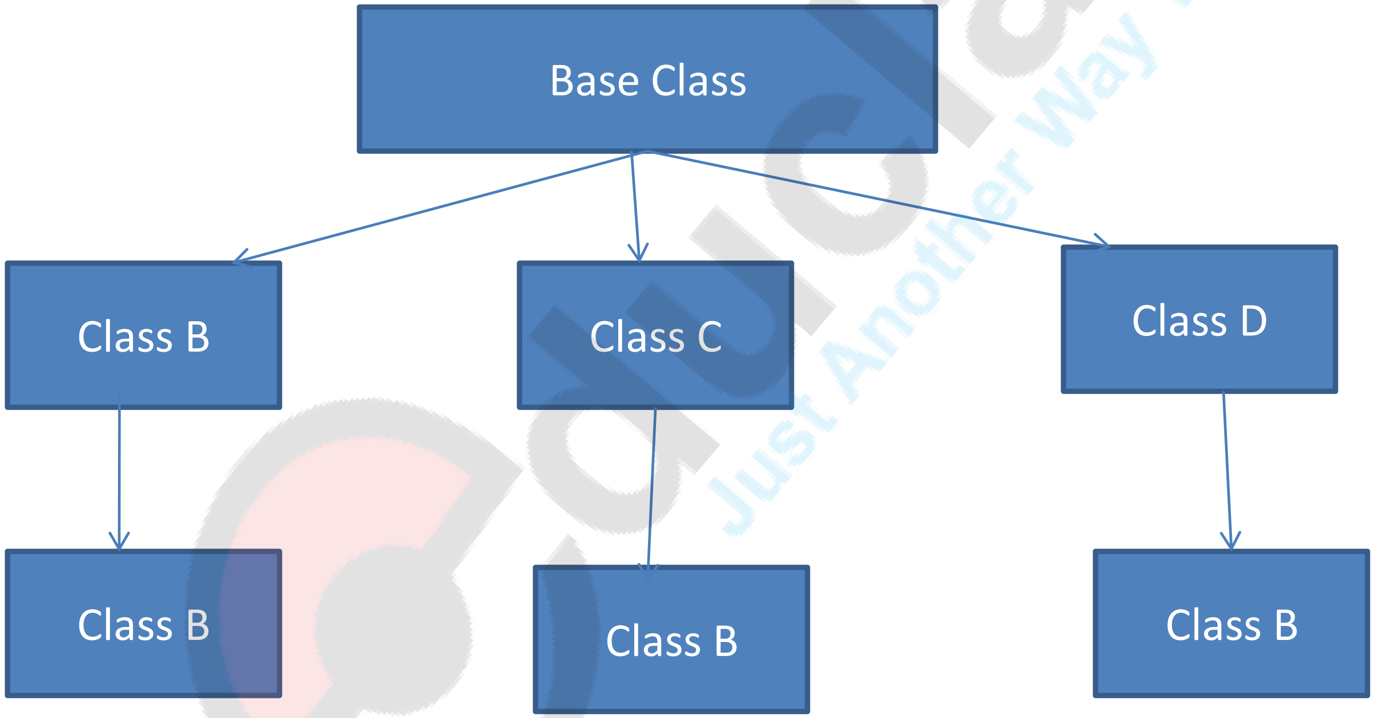
Inheritance

- Hierarchical Inheritance – when more than one derived class is inherited from a single base class.



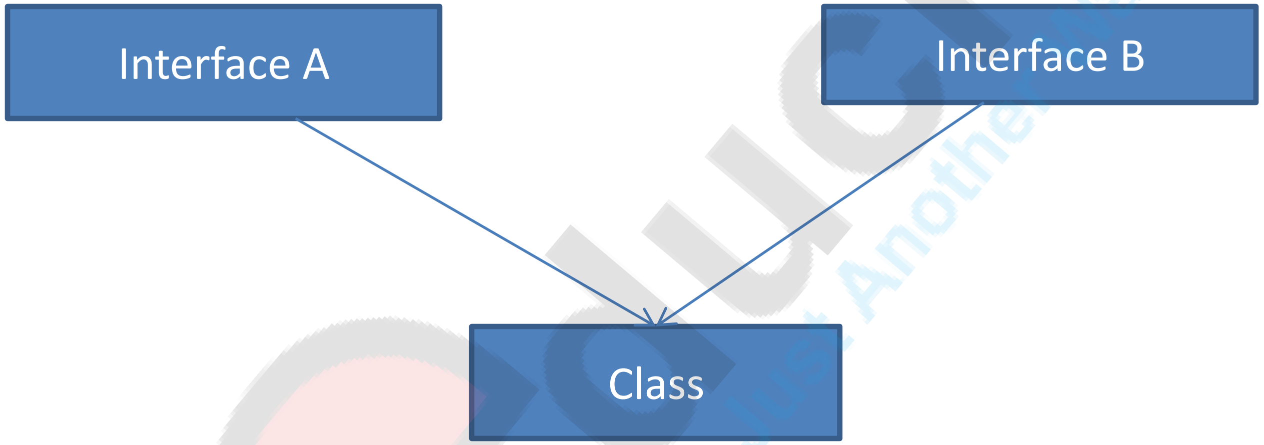
Inheritance

- Hybrid Inheritance – combination of two or more inheritance (single, multilevel, hierarchical)



Inheritance

- Multiple Inheritance – only multiple inheritance of interfaces are supported.



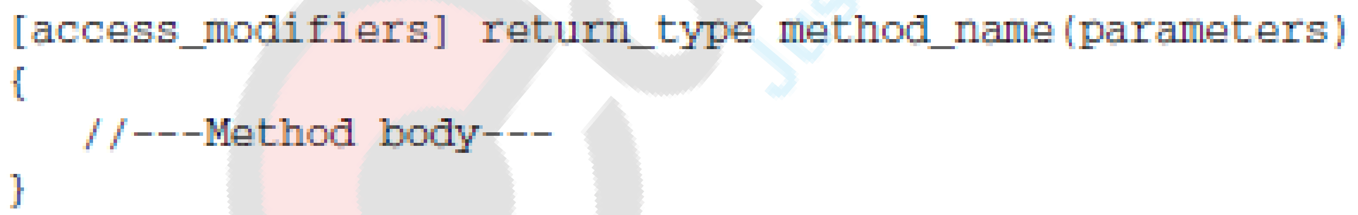
Access Modifiers

- private – highest restrictions. Access is limited only to the containing class.
- protected – access is limited to the containing class and its derived class.
- Internal – access is limited to the containing assembly.
- protected internal – access is limited to the containing assembly and derived classes in other assembly.
- Public – lowest restrictions. Access is not restricted.

Function Members:

- Methods
- Properties
- Events
- Indexers
- User - defined operators
- Constructors
- Destructors

- Method Syntax:



```
[access_modifiers] return_type method_name(parameters)
{
   //---Method body---
}
```

- A Method must be associated with a class.

Passing arguments to a method:

Call by value:

Let the function is:

```
public int AddNumbers(int x, int y)
{
    x++;
    y++;
    return x + y;
}
```

Then in the calling block

```
    int num1 = 4, num2 = 5;
    Console.WriteLine(AddNumbers(num1, num2));
    Console.WriteLine(num1,num2);
```

Call by reference: (either use ref or out keyword)

Because C# functions can only return single values, passing arguments by reference is useful when you need a method to return multiple values.

ref

In the definition of function:

```
public int AddNumbers(ref int x, ref int y)
{
    x++;
    y++;
    return x + y;
}
```

While calling:

```
int num1 = 4, num2 = 5;
Console.WriteLine(AddNumbers(ref num1, ref num2));
Console.WriteLine(num1,num2);
```

*Note**:* **The ref keyword requires that all the variables be initialized first.**

Out

- If your intention is to use the variables solely to obtain some return values from the method, you can use the out keyword.
- It is identical to the ref keyword except that it does not require the variables passed in to be initialized first.

For function definition:

```
public void GetDate(out int day, out int month, out int year)
{
        day = DateTime.Now.Day;
        month = DateTime.Now.Month;
        year = DateTime.Now.Year;
}
```

- in the calling method:

```
int day, month, year;
GetDate(out day, out month, out year);
```

- Note**: the out parameter in a function must be assigned a value before the function returns. If it isn ’ t, a compiler error results.

this keyword: The this keyword refers to the current instance of an object

(It is used to implement the Operator Overloading)

Properties: Properties are function members that provide an easy way to read or write the values of private data members.

Consider following example:

```
class  Student
{
        public string name;
        public int age;
}
```

Now you can create a student object and set its public data members as,

```
    Student s = new Student();
    s.name="Ajay Sawant";
    s.age=99;
```

Technically, the assignment to age variable is valid, but logically it should not be allowed.

A solution to this is to use properties.

The Student class can be re-written as:

```
class  Student
{
            string name;
            int age;
            public int Age
            {
                        get {
                                    return(age);
                        }
                        set
                                    age=value;
                        }
            }
            public string Name
            {
                        get{
                                    return(name);
                        }
                        set{
                                    name=value;
                        }
            }

}
```

The set accessor sets the value.

The get accessor returns the value.

Now to implement the student age constraint we can write:

```
public int Age
{
        get {
                return(age);
        }
        set{
                if (value<50 && value>6)
                        age=value;
                else
                        age=0;
        }
}
```

Read Only and Write Only properties:

- When a property has both get and set accessor → read- write property
- When a property has get but not set accessor → read- Only property
- When a property has set but not get accessor → write- Only property

(Write-Only properties are generally never used)

▪ Automatic Properties:

In C# you can shorten those properties that have no filtering (checking) rules by using a feature known as *automatic properties* .

```
public class Contact
{
    int _ID;
    public int ID
    {
        get
        {
            return _ID;
        }
        set
        {
            if (value > 0 && value <= 9999)
            {
                _ID = value;
            }
            else
            {
                _ID = 0;
            };
        }
    }
    public string FirstName {get; set;}
    public string LastName {get; set;}
    public string Email {get; set;}
}
```

▪Now there ’ s no need for you to define private members to store the values of the properties.

▪Compiler automatically generates the private variables in this case.

▪If you decide to add filtering rules to the properties later, you can simply implement the set and get accessor of each property.

▪To restrict the visibility of the get and set accessor when using the automatic properties feature, you simply prefix the get or set accessor with the private keyword, like this:

```
public string FirstName {get; private set;}
```

This statement sets the FirstName property as read - only.

Constructors

- A special method of the class that will be automatically invoked when an instance of the class is created is called a constructor.
- The main use of constructors is to initialize private fields of the class while creating an instance for the class.

Constructors

Types of constructors:-

- Default Constructor
- Parameterized Constructor
- Copy Constructor
- Static Constructor
- Private Constructor

Constructors

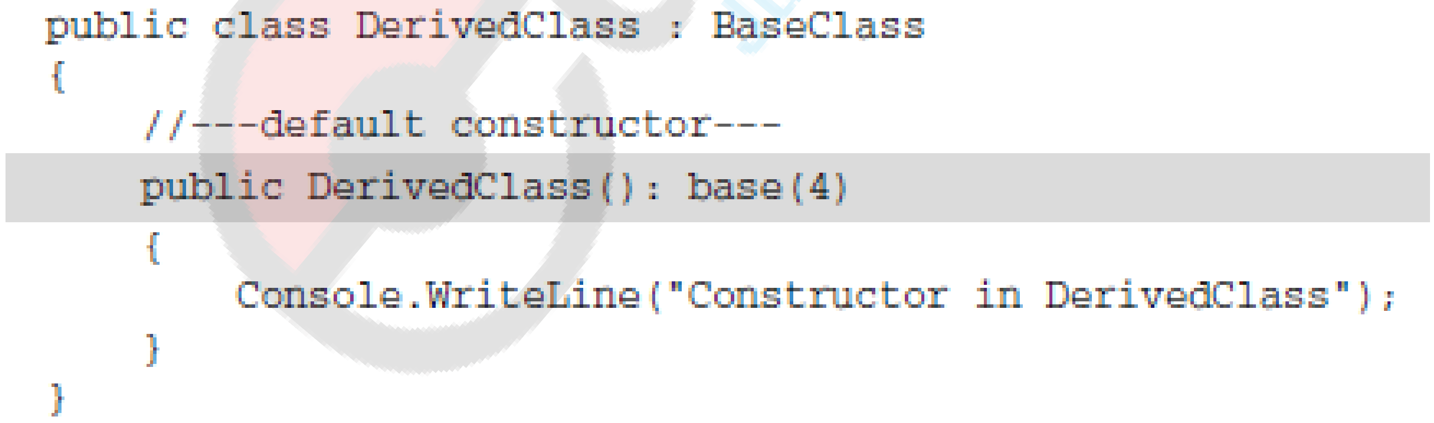
- **Default** Constructor – constructors with no parameters
- **Parameterized** Constructor - A constructor with at least one parameter
- **Copy** Constructor - The constructor which creates an object by copying variables from another object. Basically, it take the object of the same class as a parameter.
- **Static** Constructor - When a constructor is created as static, it will be invoked only once for all of instances of the class and it is invoked during the creation of the first instance of the class or the first reference to a static member in the class. A static constructor is used to initialize static fields of the class and to write the code that needs to be executed only once.
- **Private** Constructor - When a constructor is created with a private specifier, it is not possible for other classes to derive from this class, neither is it possible to create an instance of this class. They are usually used in classes that contain static members only.

Inheritance and Constructors

Remember that if a base class contains constructors, one of them must be a default constructor.

- Suppose BaseClass contains two constructors — one default and one parameterized: And DerivedClass contains one default constructor:

You can choose which constructor you want to invoke in BaseClass by using the base keyword in the default constructor in DerivedClass , like this:



```
public class DerivedClass : BaseClass
{
    //---default constructor---
    public DerivedClass(): base(4)
    {
        Console.WriteLine("Constructor in DerivedClass");
    }
}
```

Constructors:

Static Constructors

- 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.

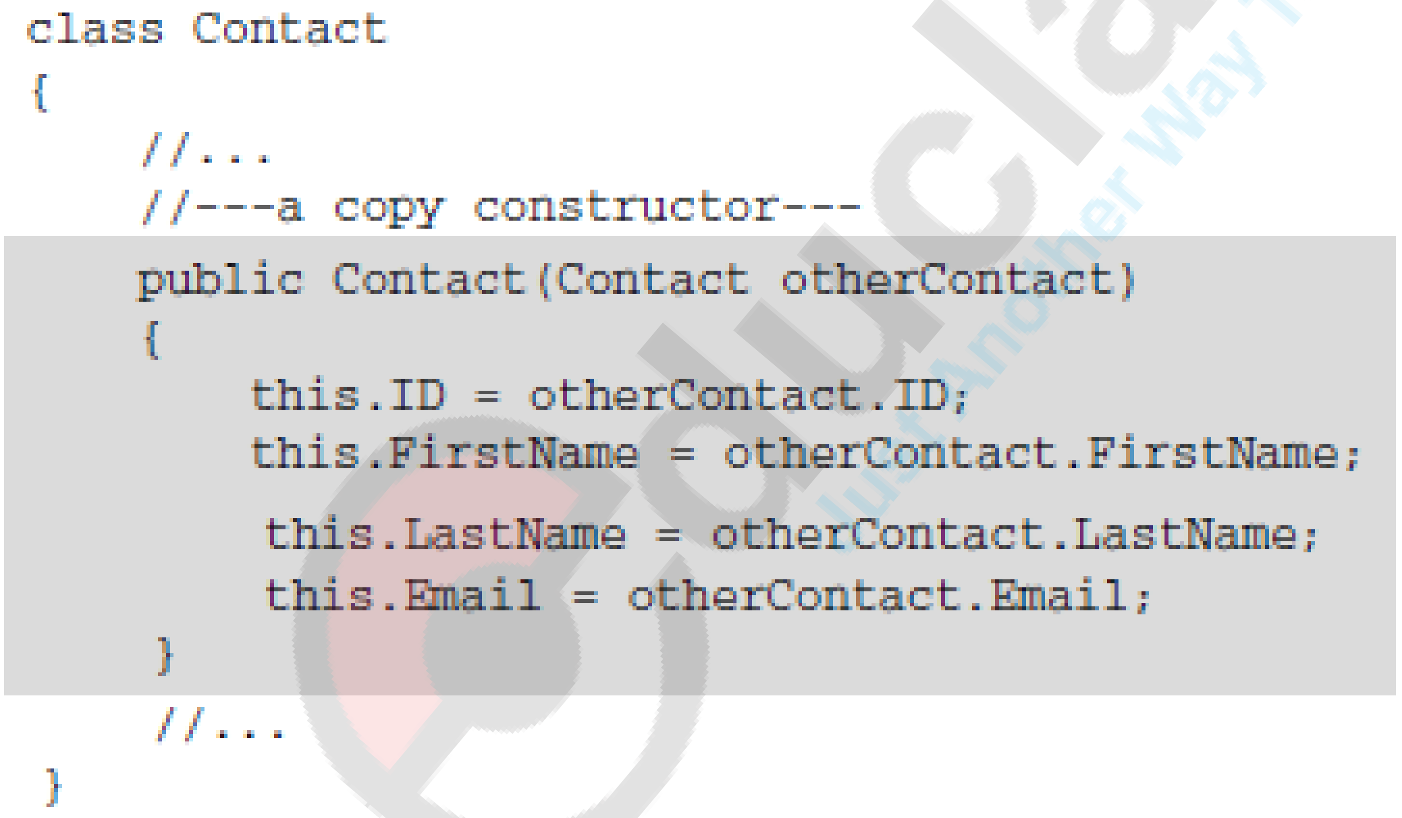
Static constructor to count no of Objects created for a class:

```
public class Contact
{
    //...
    public static int count;

    static Contact()
    {
        count = 0;
        Console.WriteLine("Static constructor");
    }

    //---first constructor---
    public Contact()
    {
        count++;
        Console.WriteLine("First constructor");
    }

    //...
}
```

Copy Constructor:

The C# language does not provide a copy constructor that allows you to copy the value of an existing object into a new object when it is created.

Instead, you have to write your own.



```
class Contact
{
     //...
     //---a copy constructor---
     public Contact(Contact otherContact)
     {
          this.ID = otherContact.ID;
          this.FirstName = otherContact.FirstName;
          this.LastName = otherContact.LastName;
          this.Email = otherContact.Email;
     }
     //...
}
```

To use the copy constructor, first create a `Contact` object:

```
Contact c1 = new Contact(1234, "Wei-Meng", "Lee",
                         "weimenglee@learn2develop.net");
```

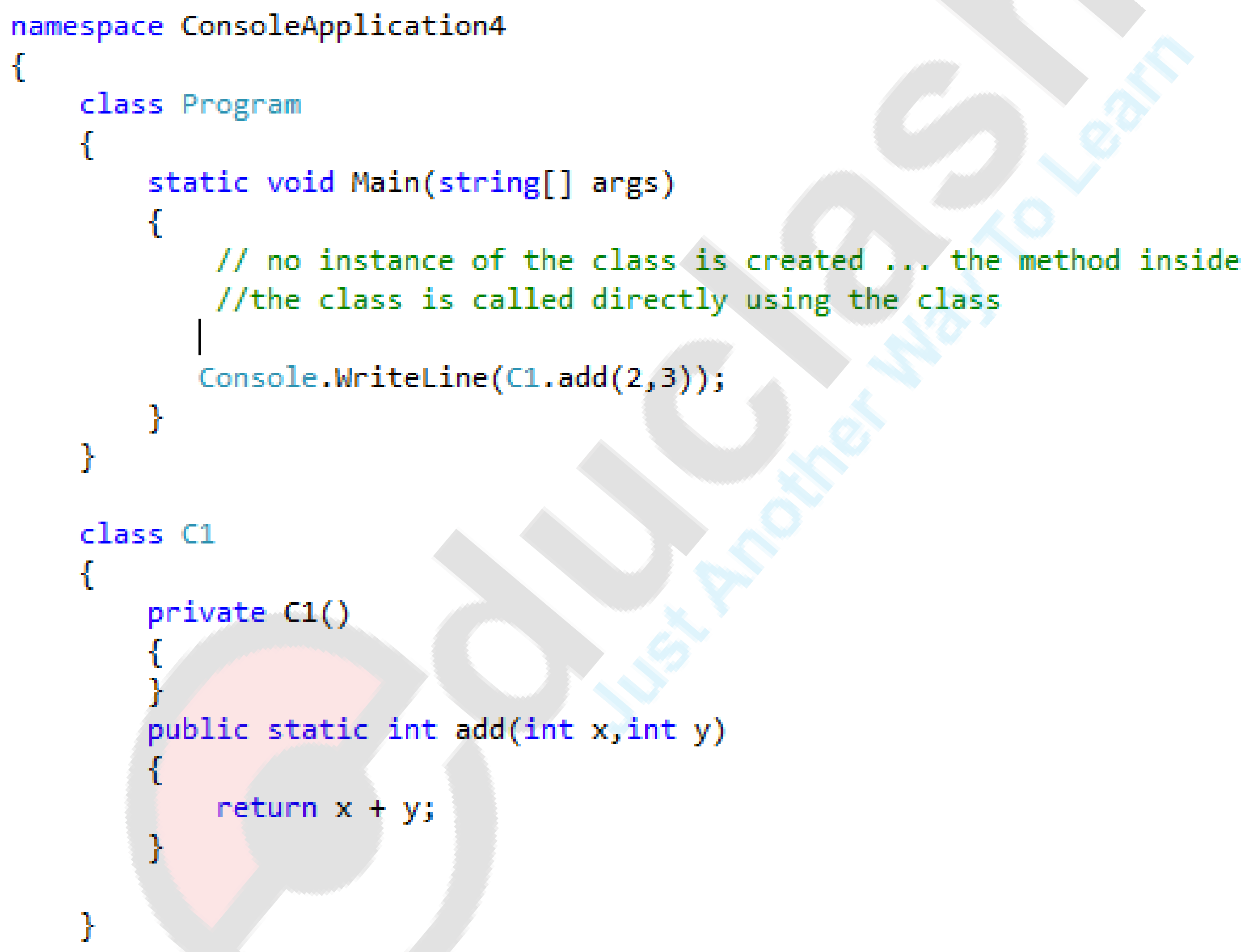
Then, instantiate another `Contact` object and pass in the first object as the argument:

```
Contact c2 = new Contact(c1);
Console.WriteLine(c2.ID);        //---1234---
Console.WriteLine(c2.FirstName); //----Wei-Meng---
Console.WriteLine(c2.LastName);  //---Lee---
Console.WriteLine(c2.Email);     //--- weimenglee@learn2develop.net---
```

Private Constructor

- A class with a private constructor cannot be inherited.
- An object of this class cannot be created until there is a parameterized constructor.
- Private constructors are used to put utility methods inside the class and to call those methods directly without creating instances.

Private Constructor



```
namespace ConsoleApplication4
{
    class Program
    {
        static void Main(string[] args)
        {
            // no instance of the class is created ... the method inside
            //the class is called directly using the class

           Console.WriteLine(C1.add(2,3));
        }
    }

    class C1
    {
        private C1()
        {
        }
        public static int add(int x,int y)
        {
            return x + y;
        }

    }
```

System.Object Class:

- In C#, all classes inherit from the System.Object base class.
- This means that all classes contain the methods defined in the System.Object class.
- All class definitions that do not inherit from other classes by default inherit directly from the System .Object class.

Methods of System.Object class:

Equals() —

- Checks whether the value of the current object is equal to that of another object.
- By default, the Equals() method checks for *reference equality (that is, if two objects are pointing to* the same object). You should override this method for your class.

GetHashCode() —

- Returns a hash code for the class. The GetHashCode() method is suitable for use in hashing algorithms and data structures, such as a hash table.

GetType() —

- Returns the type of the current object

ToString() —

- Returns the string representation of an object

Structures

Structures:

An alternative to using classes is to use a *struct.*

A struct is a lightweight user - defined type that is very similar to a class, but with some exceptions:

- Structs do not support inheritance or destructors.
- A struct is a value type (class is a reference type).
- A struct cannot declare a default constructor.
- structs are allocated memory on stack.

- Like classes, structs support constructor, properties, and methods.

Abstract Class:

- The Shape class does not specify a particular shape, and thus it really does not make sense for you to instantiate it directly.
- Instead, all other shapes should inherit from this base class.
- To ensure that you cannot instantiate the Shape class directly, you can make it an *abstract class by using the abstract keyword.*

```
public abstract class Shape
{
    //---properties---
    public double length { get; set; }
    public double width { get; set; }

    //---method---
    public double Perimeter()
    {
        return 2 * (this.length + this.width);
    }
}
```

Abstract Methods:

An abstract method has no implementation, and its implementation is left to the classes that inherit from the class that defines it.

- An abstract method is defined just like a normal method without the normal method block ({}).

- Classes that inherit from a class containing abstract methods must provide the implementation for those methods.

- Note**: An abstract class can contain both abstract as well as non abstract methods.

```
public abstract class Shape
{
    //---properties---
    public double length { get; set; }

    public double width { get; set; }

    //---method---
    public double Perimeter()
    {
        return 2 * (this.length + this.width);
    }

    //---abstract method---
    public abstract double Area();
}
```

```
public class Rectangle : Shape
{
    //---provide the implementation for the abstract method---
    public override double Area()
    {
        return this.length * this.width;
    }
}
```

Interfaces

Interface definition

The interface defines the composition of a class, such as methods properties, and so on. However, the interface does not provide any implementation for any of these members.

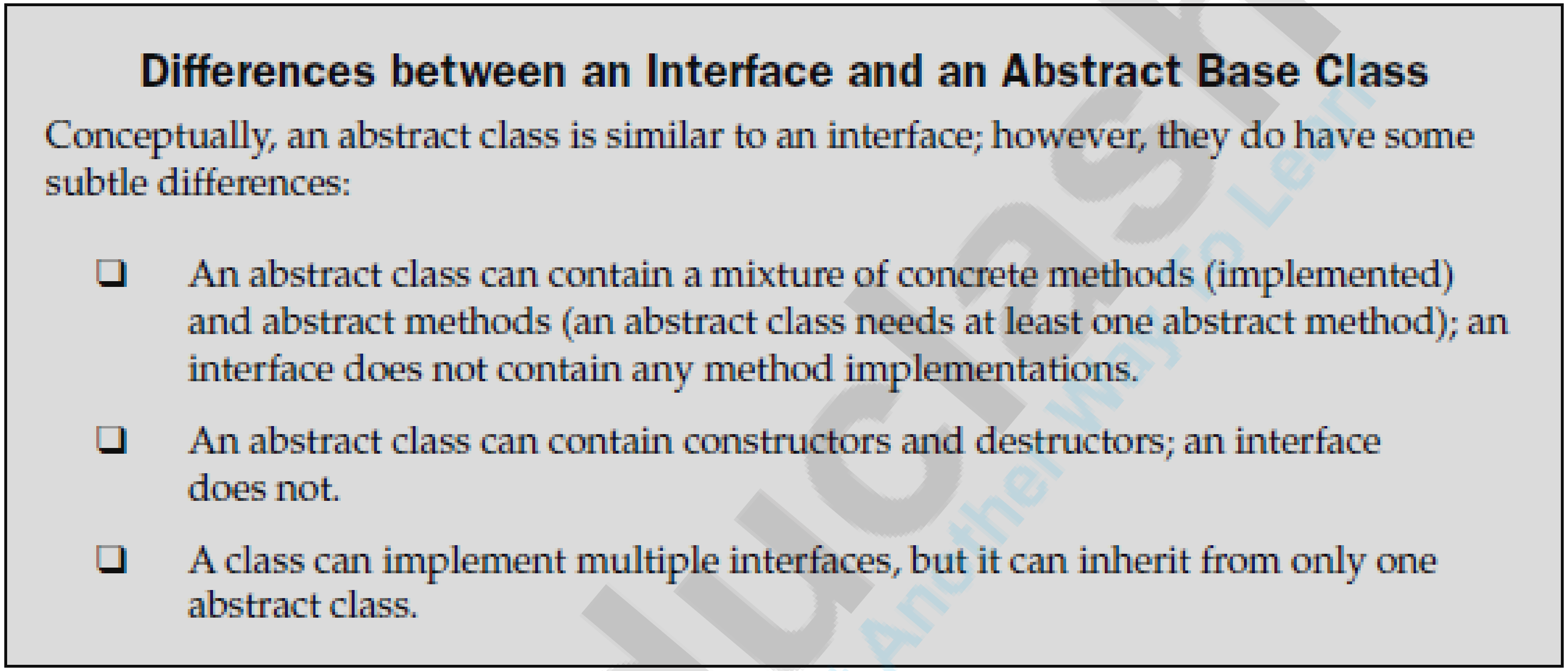
Implementing class

The class that implements a particular interface provides the implementation for all the members defined in that interface.

Clients

Objects that instantiate from the implementing classes are known as the *client* .

The client invokes the methods defined in the interface, whose implementation is provided by the implementing class.



Differences between an Interface and an Abstract Base Class

Conceptually, an abstract class is similar to an interface; however, they do have some subtle differences:

- ❑ An abstract class can contain a mixture of concrete methods (implemented) and abstract methods (an abstract class needs at least one abstract method); an interface does not contain any method implementations.
- ❑ An abstract class can contain constructors and destructors; an interface does not.
- ❑ A class can implement multiple interfaces, but it can inherit from only one abstract class.

Defining Interface:

Defining an interface is similar to defining a class — you use the interface keyword followed by an identifier (the name of the interface) and then specify the interface body.

Eg.

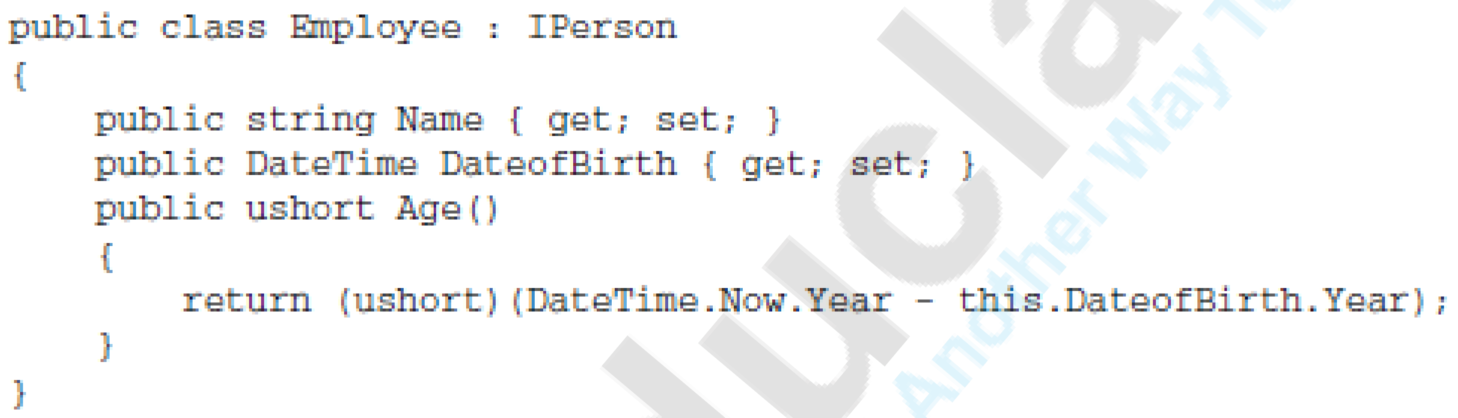
```
interface IPerson
{
    string Name { get; set; }
    DateTime DateofBirth { get; set; }
    ushort Age();
}
```

You do not use any access modifiers on interface members — they are implicitly public.

It ’ s important to note that you cannot create an instance of the interface directly; you can only instantiate a class that implements that interface.

Implementing an Interface:

Once an interface is defined, you can create a new class to implement it. The class that implements that particular interface must provide all the implementation for the members defined in that interface.



```
public class Employee : IPerson
{
    public string Name { get; set; }
    public DateTime DateofBirth { get; set; }
    public ushort Age()
    {
        return (ushort)(DateTime.Now.Year - this.DateofBirth.Year);
    }
}
```

All implemented members must have the public access modifiers.

You can now us e the class as you would a normal class:

```
Employee e1 = new Employee();
e1.DateofBirth = new DateTime(1980, 7, 28);
e1.Name = "Janet";
Console.WriteLine(e1.Age());  //---prints out 28---
```

Implementing Multiple Interfaces:

A class can implement any number of interfaces.

```
public class Employee : IPerson, IAddress
{
   //---implementation here---
}
```

Extending Interfaces:

You can extend interfaces if you need to add new members to an existing interface.

For example, you might want to define another interface named IManager to store information about managers.

Basically, a manager uses the same members defined in the IPerson interface, with perhaps just one more additional property — Dept.

In this case, you can define the IManager interface by extending the IPerson interface, like this:

```
interface IPerson
{
    string Name { get; set; }
    DateTime DateofBirth { get; set; }
    ushort Age();
}
```

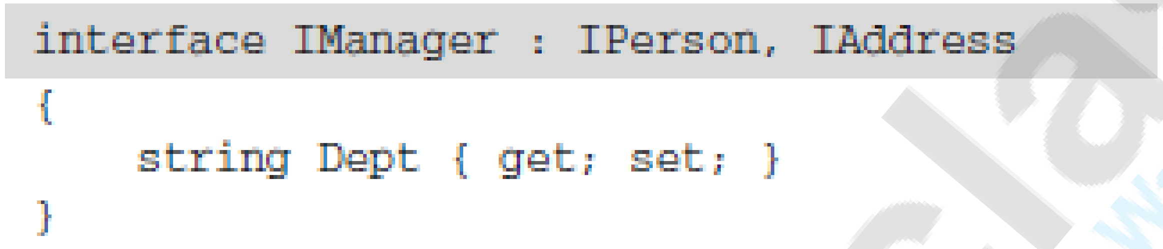
```
interface IManager : IPerson
{
    string Dept { get; set; }
}
```

To use the IManager interface, you define a Manager class that implements the IManager interface, like this:

```
public class Manager : IManager
{
    //---IPerson---
    public string Name { get; set; }
    public DateTime DateofBirth { get; set; }
    public ushort Age()
    {
        return (ushort)(DateTime.Now.Year - this.DateofBirth.Year);
    }

    //---IManager---
    public string Dept { get; set; }
}
```

You can also extend multiple interfaces at the same time.

The following example shows the Imanager interface extending both the IPerson and the IAddress interfaces:



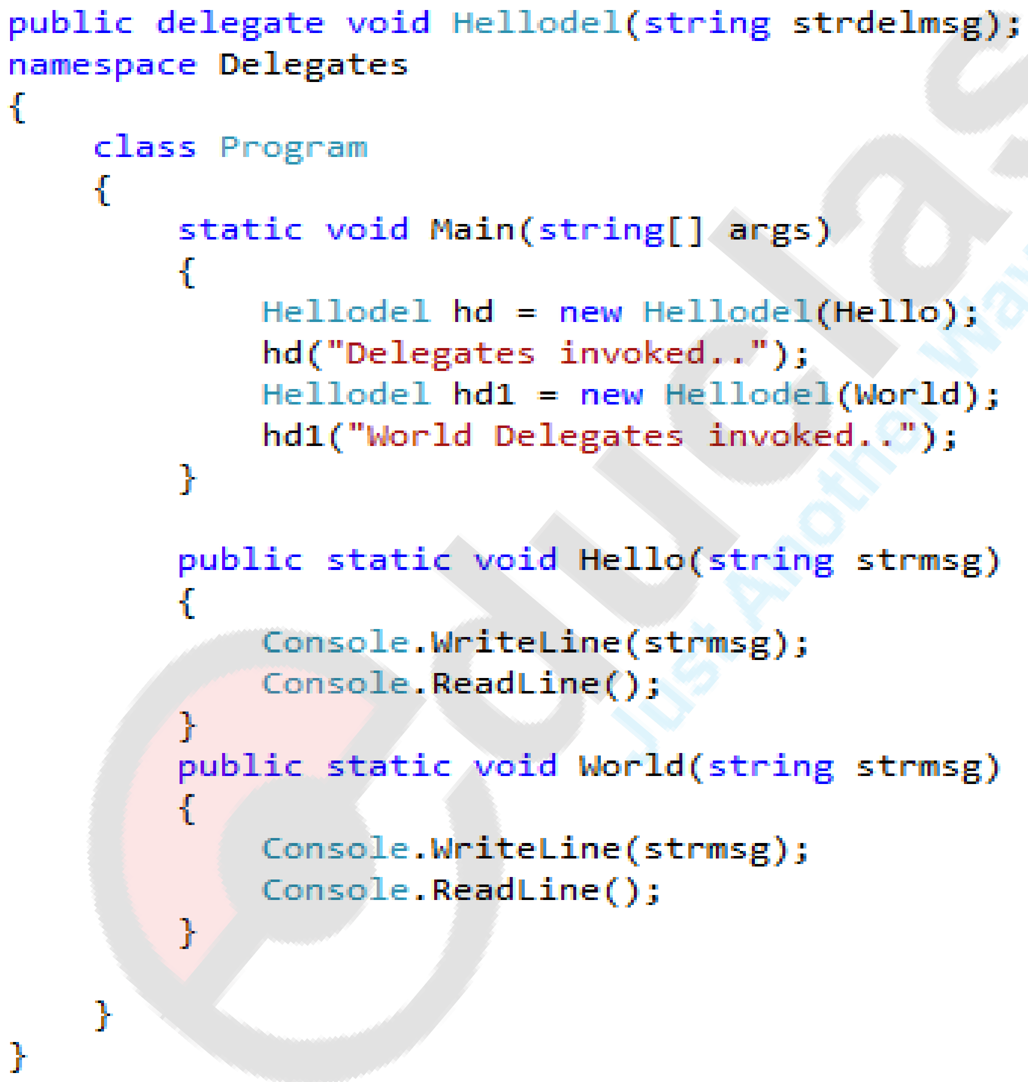
```
interface IManager : IPerson, IAddress
{
    string Dept { get; set; }
}
```

The Manager class now needs to implement the additional members defined in the IAddress interface

Delegates

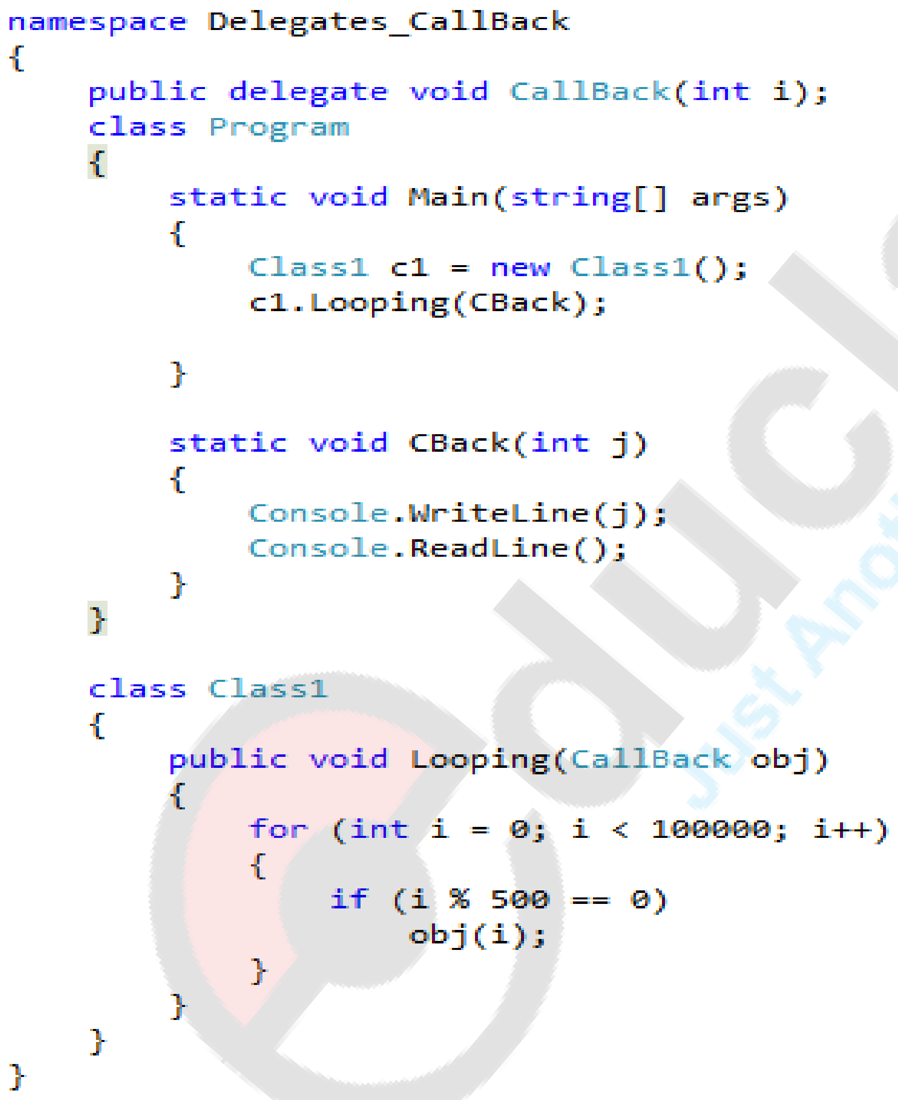
- A delegate is a type-safe function pointer i.e. when you invoke the delegate, the function gets invoked.
- The signature of the delegate must match the signature of the function, the delegate points to (type-safe).
- To use a delegate an instance has to be created similar to class and invoked similar to a method.
- The syntax of a delegate is also similar to the syntax of a method.
- Delegates can be used to define callback methods.
- Delegates allow methods to be passed as parameters.

Delegates



```
public delegate void Hellodel(string strdelmsg);
namespace Delegates
{
    class Program
    {
        static void Main(string[] args)
        {
            Hellodel hd = new Hellodel(Hello);
            hd("Delegates invoked..");
            Hellodel hd1 = new Hellodel(World);
            hd1("World Delegates invoked..");
        }

        public static void Hello(string strmsg)
        {
            Console.WriteLine(strmsg);
            Console.ReadLine();
        }
        public static void World(string strmsg)
        {
            Console.WriteLine(strmsg);
            Console.ReadLine();
        }

    }
}
```

Delegates - CallBack

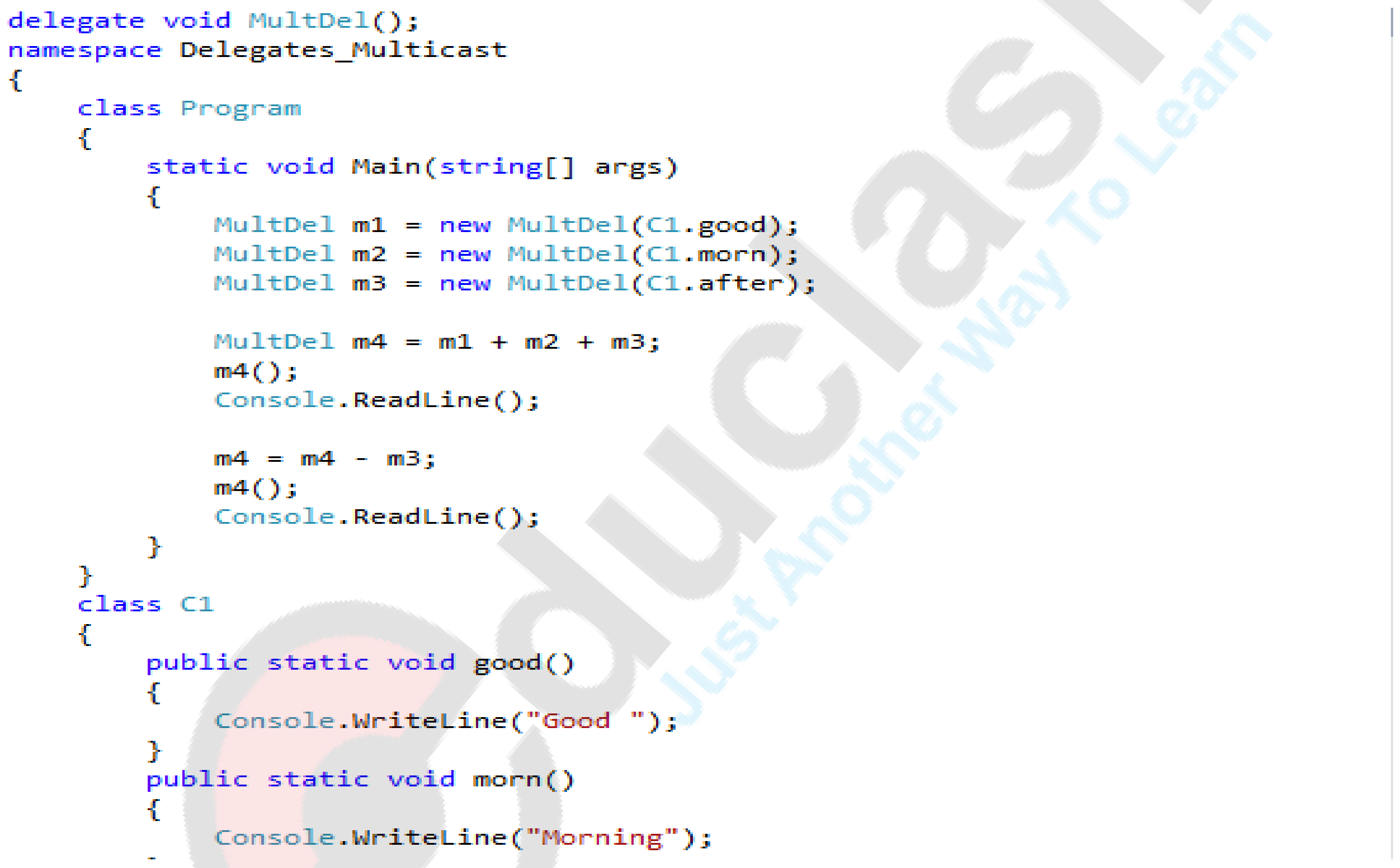


```
namespace Delegates_CallBack
{
    public delegate void CallBack(int i);
    class Program
    {
        static void Main(string[] args)
        {
            Class1 c1 = new Class1();
            c1.Looping(CBack);

        }

        static void CBack(int j)
        {
            Console.WriteLine(j);
            Console.ReadLine();
        }
    }

    class Class1
    {
        public void Looping(CallBack obj)
        {
            for (int i = 0; i < 100000; i++)
            {
                if (i % 500 == 0)
                    obj(i);
            }
        }
    }
}
```

Delegates - MultiCast

- Delegate which hold and invoke multiple methods such Delegates are called Multicast Delegates.
- Multicast Delegates are also known as Combinable Delegates.
- The methods must satisfy the conditions like the return type of the Delegate.
- Multicast Delegate instance is created by combining two Delegates.

Delegates - MultiCast



```
delegate void MultDel();
namespace Delegates_Multicast
{
    class Program
    {
        static void Main(string[] args)
        {
            MultDel m1 = new MultDel(C1.good);
            MultDel m2 = new MultDel(C1.morn);
            MultDel m3 = new MultDel(C1.after);

            MultDel m4 = m1 + m2 + m3;
            m4();
            Console.ReadLine();

            m4 = m4 - m3;
            m4();
            Console.ReadLine();
        }
    }
    class C1
    {
        public static void good()
        {
            Console.WriteLine("Good ");
        }
        public static void morn()
        {
            Console.WriteLine("Morning");
```

Virtual Methods

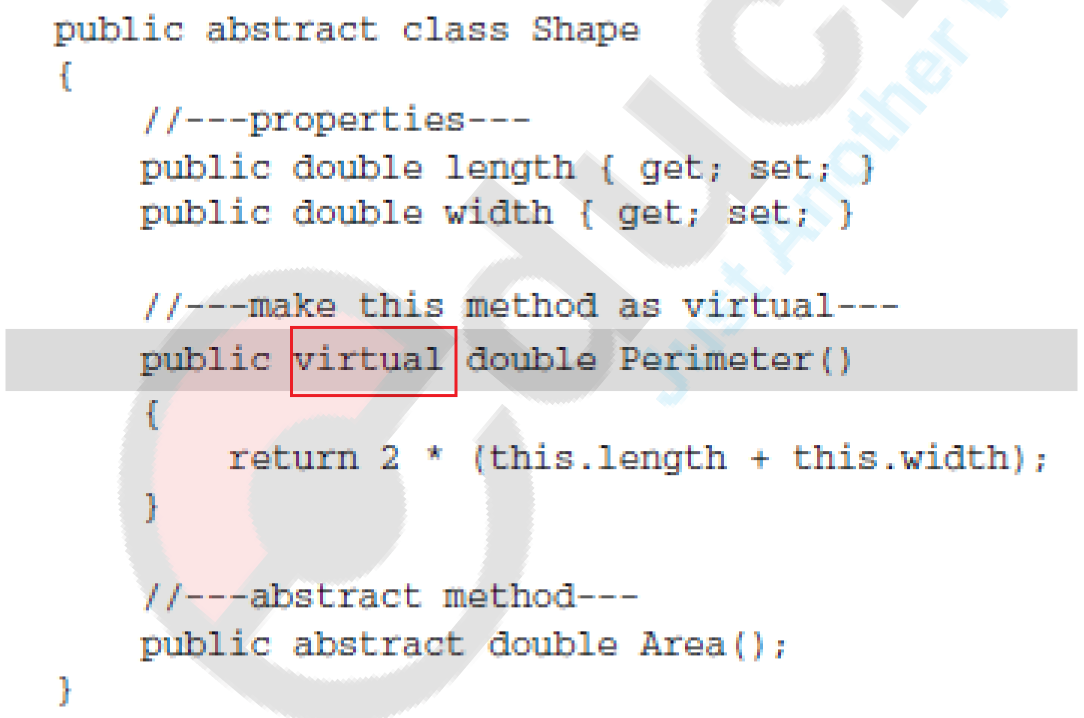
(Function Overriding, Runtime polymorphism)

If we want to create a Circle class, we will derive it from shape class.

Then the perimeter function which is defined in the shape class can not be used, as Circle don't have length and breadth.

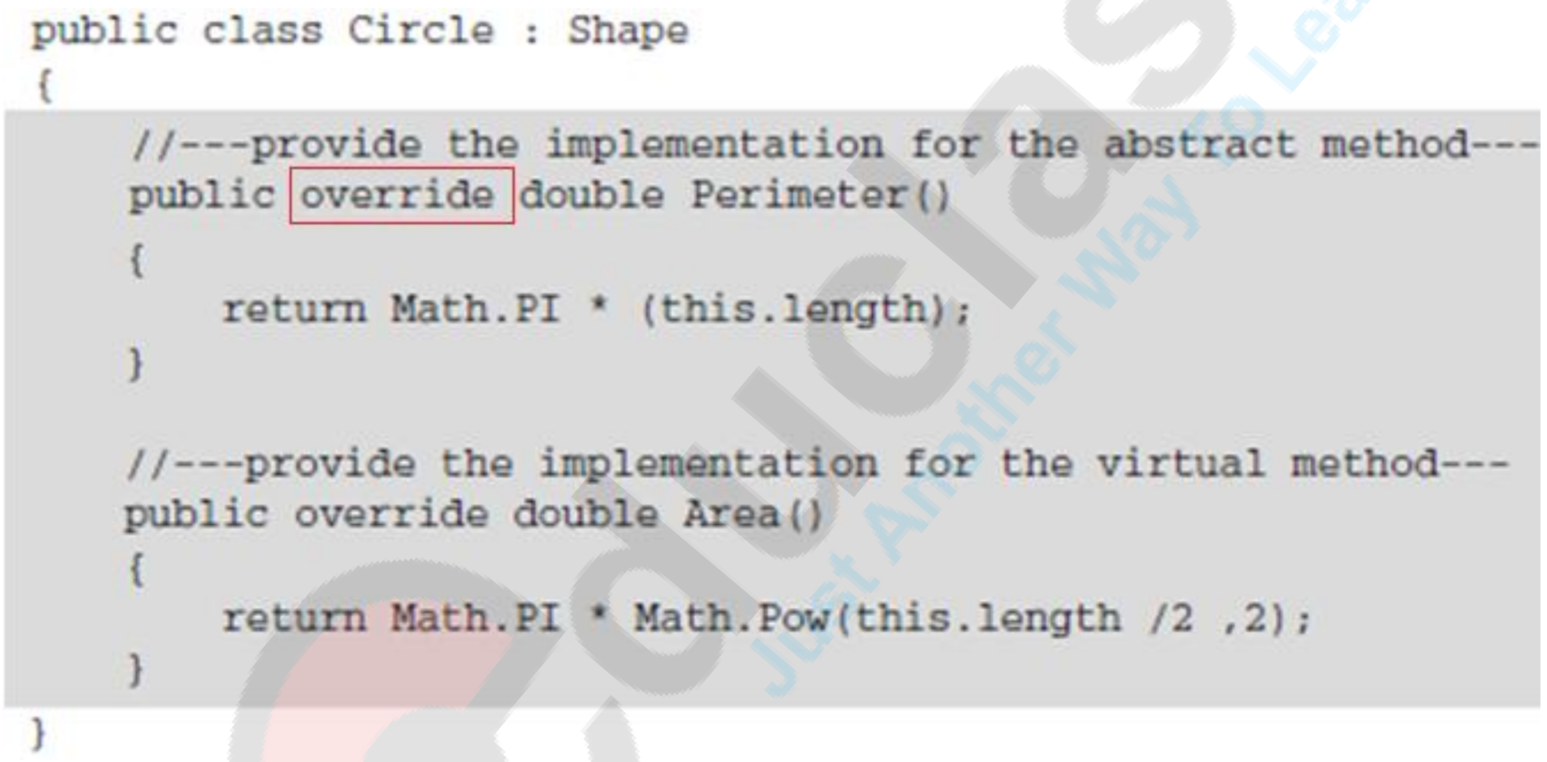
Hence we need to create a new perimeter function in Circle class.

To do so, we need to prefix the Perimeter() method with the virtual keyword to indicate that all derived classes have the option to change its implementation:



```
public abstract class Shape
{
    //---properties---
    public double length { get; set; }
    public double width { get; set; }

    //---make this method as virtual---
    public virtual double Perimeter()
    {
        return 2 * (this.length + this.width);
    }

    //---abstract method---
    public abstract double Area();
}
```

The Circle class now has to provide implementation for both the Perimeter() and Area() methods

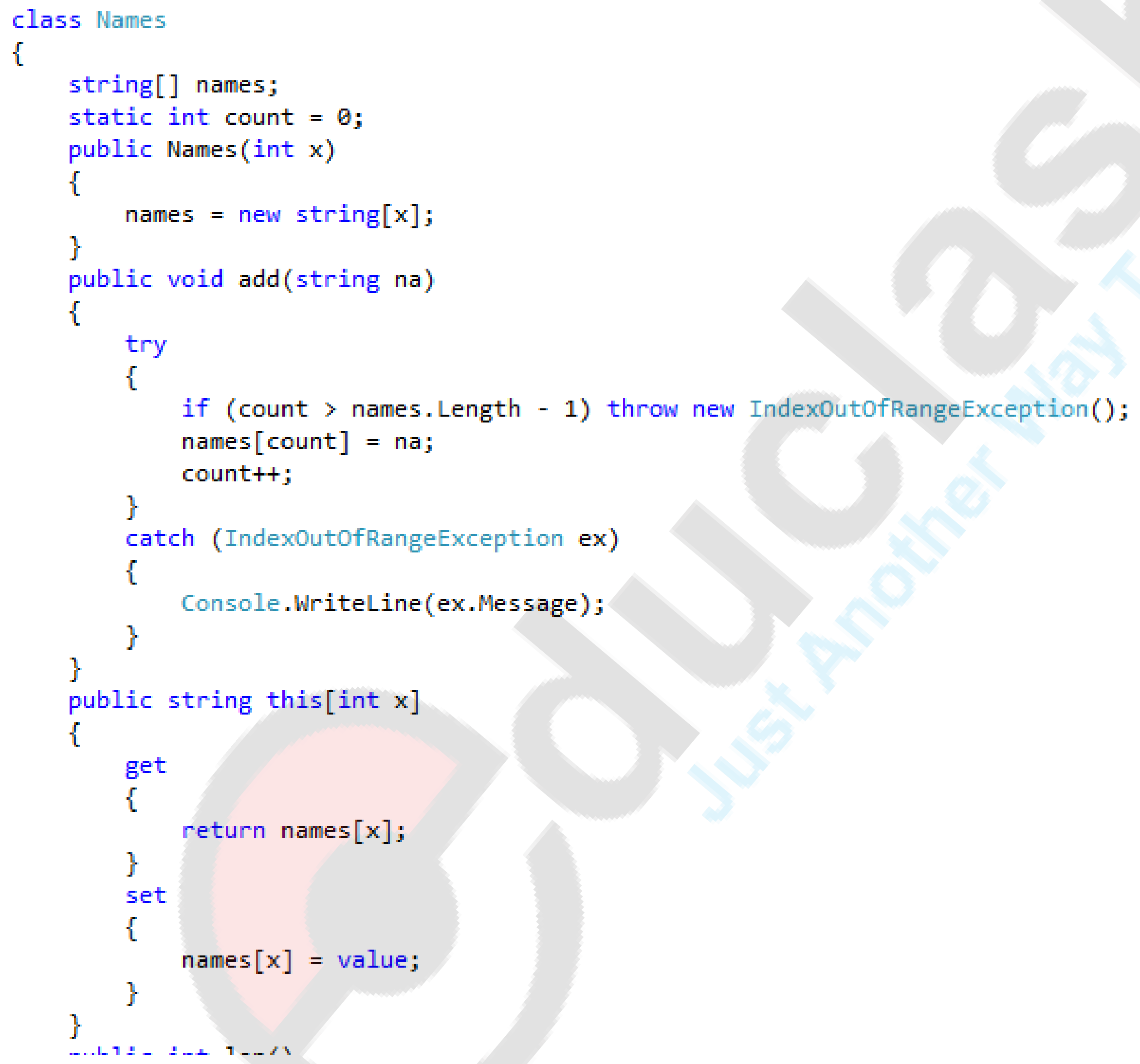


```
public class Circle : Shape
{
    //---provide the implementation for the abstract method---
    public override double Perimeter()
    {
        return Math.PI * (this.length);
    }

    //---provide the implementation for the virtual method---
    public override double Area()
    {
        return Math.PI * Math.Pow(this.length /2 ,2);
    }
}
```

Indexers

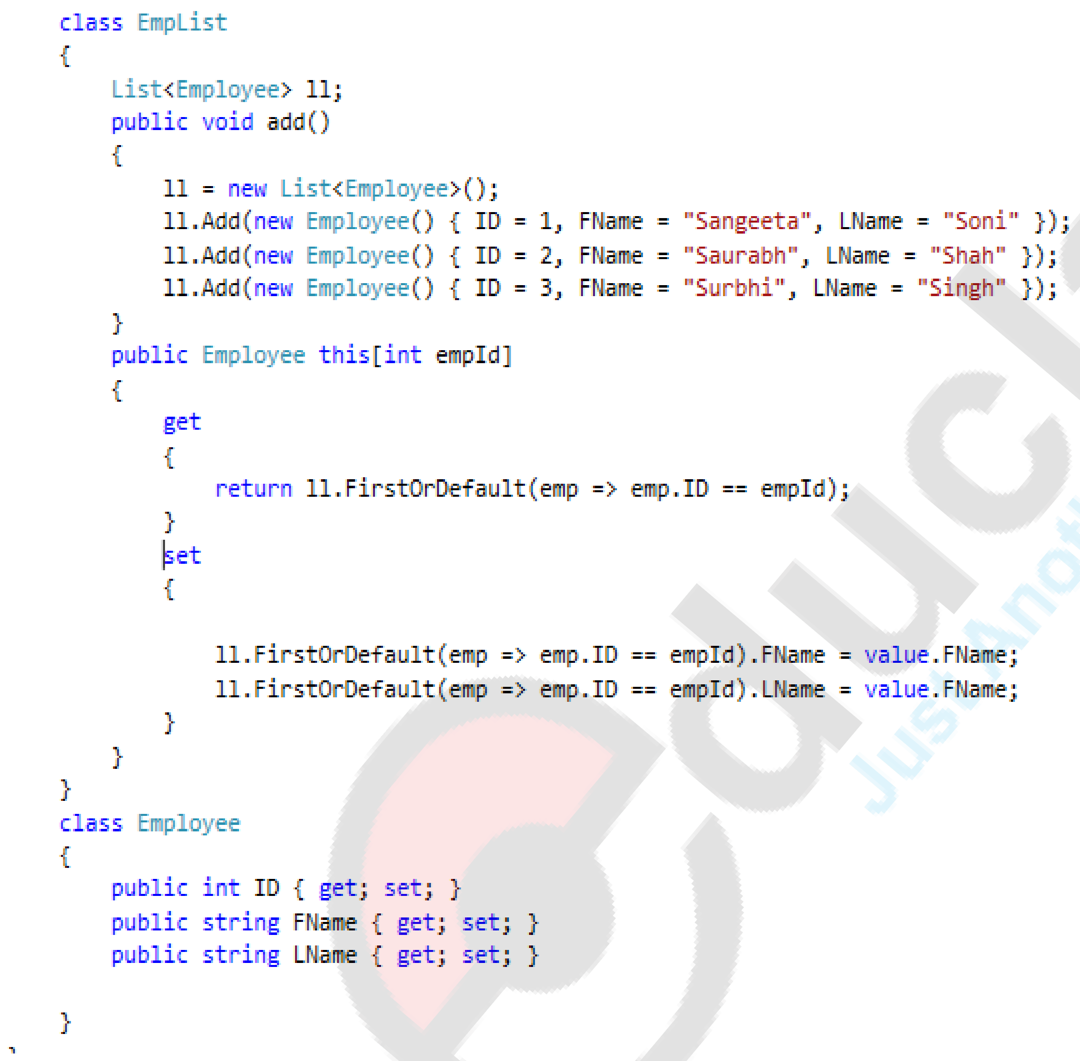
- Allows elements of an internal collection of an object to be accessed by using array notation on the object itself.
- Indexers allow instances of a class or struct to be indexed just like arrays.
- Accessed through an index.
- Indexers use simple get and set accessor methods to assign and retrieve values.

Indexers



```
class Names
{
    string[] names;
    static int count = 0;
    public Names(int x)
    {
        names = new string[x];
    }
    public void add(string na)
    {
        try
        {
            if (count > names.Length - 1) throw new IndexOutOfRangeException();
            names[count] = na;
            count++;
        }
        catch (IndexOutOfRangeException ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
    public string this[int x]
    {
        get
        {
            return names[x];
        }
        set
        {
            names[x] = value;
        }
    }
    public int len()
```

Indexers



```
class EmpList
{
    List<Employee> ll;
    public void add()
    {
        ll = new List<Employee>();
        ll.Add(new Employee() { ID = 1, FName = "Sangeeta", LName = "Soni" });
        ll.Add(new Employee() { ID = 2, FName = "Saurabh", LName = "Shah" });
        ll.Add(new Employee() { ID = 3, FName = "Surbhi", LName = "Singh" });
    }
    public Employee this[int empId]
    {
        get
        {
            return ll.FirstOrDefault(emp => emp.ID == empId);
        }
        set
        {

            ll.FirstOrDefault(emp => emp.ID == empId).FName = value.FName;
            ll.FirstOrDefault(emp => emp.ID == empId).LName = value.FName;
        }
    }
}
class Employee
{
    public int ID { get; set; }
    public string FName { get; set; }
    public string LName { get; set; }

}
```

Sealed classes and Methods

Sealed Class:

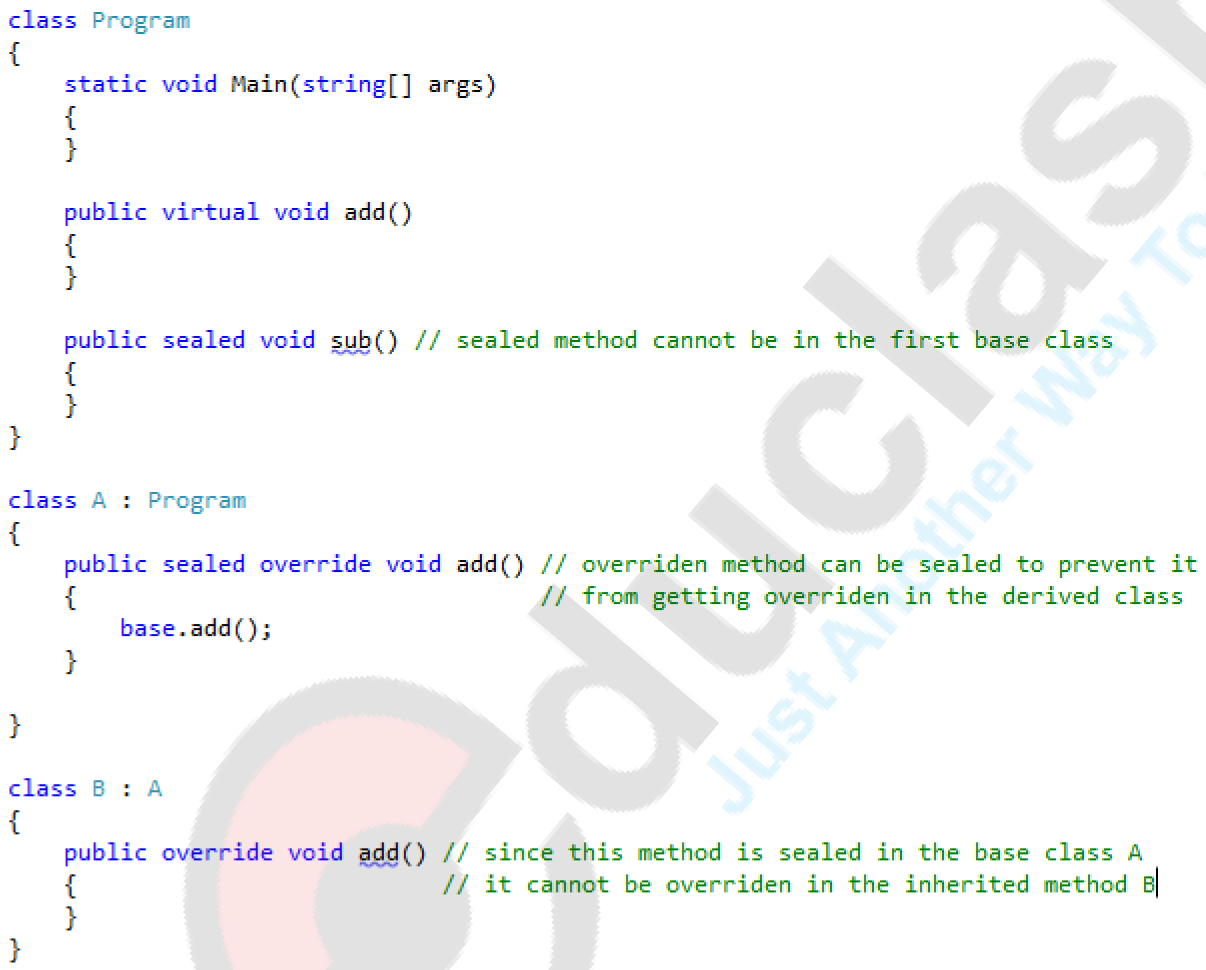
- A class prefixed with the sealed keyword prevents other classes inheriting from it.
- A sealed class cannot contain virtual methods.

Sealed Methods:

- You can also seal methods so that other derived classes cannot override the implementation that you have provided in the current class.
- However, sealed methods cannot be in the first base class.

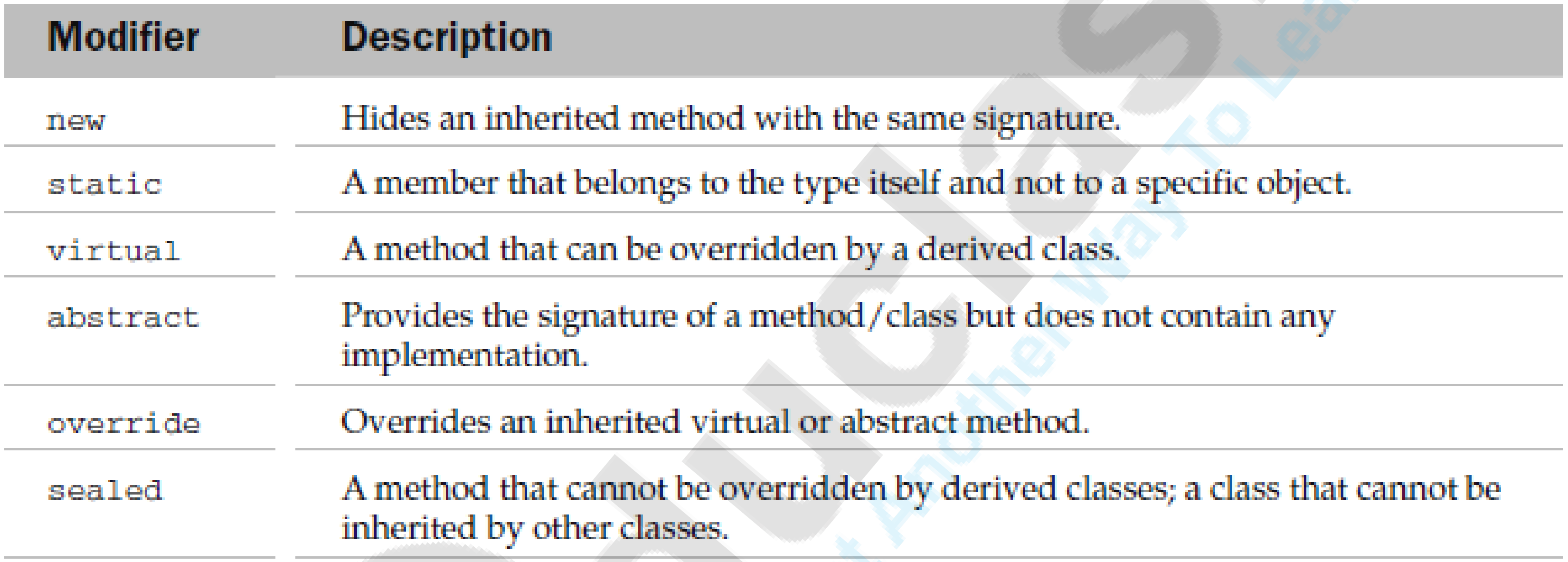
```
public class Rectangle : Shape
{
    public override sealed double Area()
    {
            return this.length * this.width;
    }
}
```

Sealed classes and Methods



```
class Program
{
    static void Main(string[] args)
    {
    }

    public virtual void add()
    {
    }

    public sealed void sub() // sealed method cannot be in the first base class
    {
    }
}

class A : Program
{
    public sealed override void add() // overriden method can be sealed to prevent it
    {                                 // from getting overriden in the derived class
        base.add();
    }

}

class B : A
{
    public override void add() // since this method is sealed in the base class A
    {                          // it cannot be overriden in the inherited method B
    }
}
```

The following table summarizes the different keywords used for inheritance.



Modifier	Description
`new`	Hides an inherited method with the same signature.
`static`	A member that belongs to the type itself and not to a specific object.
`virtual`	A method that can be overridden by a derived class.
`abstract`	Provides the signature of a method/class but does not contain any implementation.
`override`	Overrides an inherited virtual or abstract method.
`sealed`	A method that cannot be overridden by derived classes; a class that cannot be inherited by other classes.

Polymorphism

- Runtime Polymorphism or late binding or dynamic binding
  - method overriding
  - Same method, same signature, different class
- Compile time Polymorphism or early binding or static binding - overloading

Overloading Methods

(Function Overloading, Compile time polymorphism)

Method Overloading:

When you have multiple methods in a class having the same name but different signatures (parameters), they are known as *overloaded methods* .

```
class Class1
 {
     void print(int i)
     {
        Console.WriteLine("Printing int: {0}", i);
     }
     void print(double f)
     {
        Console.WriteLine("Printing float: {0}", f);
     }
     void print(string s)
     {
        Console.WriteLine("Printing string: {0}", s);
     }
 }
```

Operator Overloading
(Compile time Polymorphism)

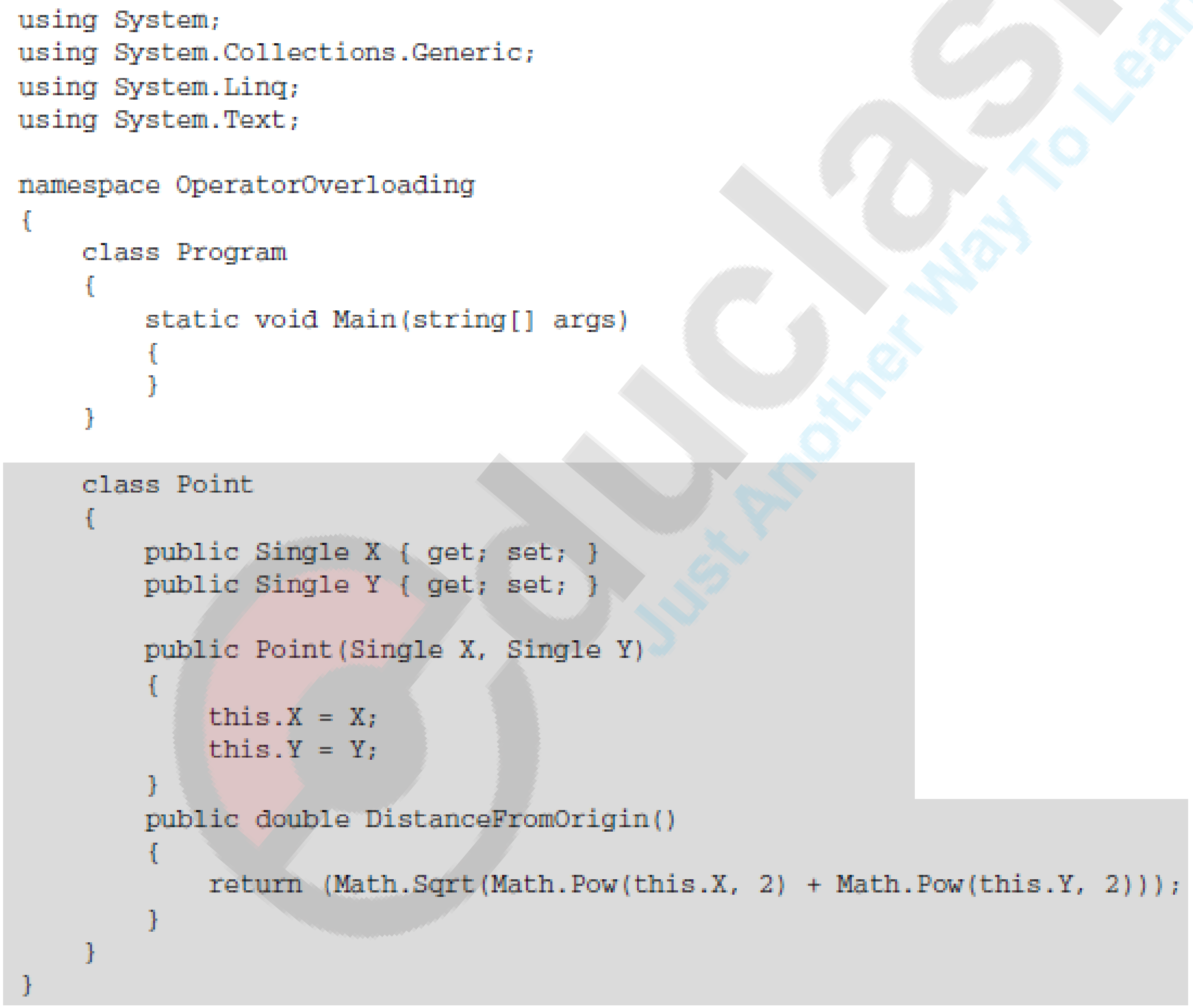
- Overloaded operators are functions with special names the keyword **operator** followed by the symbol for the operator being defined.
- An overloaded operator has a return type and a parameter list.
- Overloaded operators are always static.

```
public static B operator+(B b1, B b2)
{ return b1.Add(b2); }
```

Operator Overloading

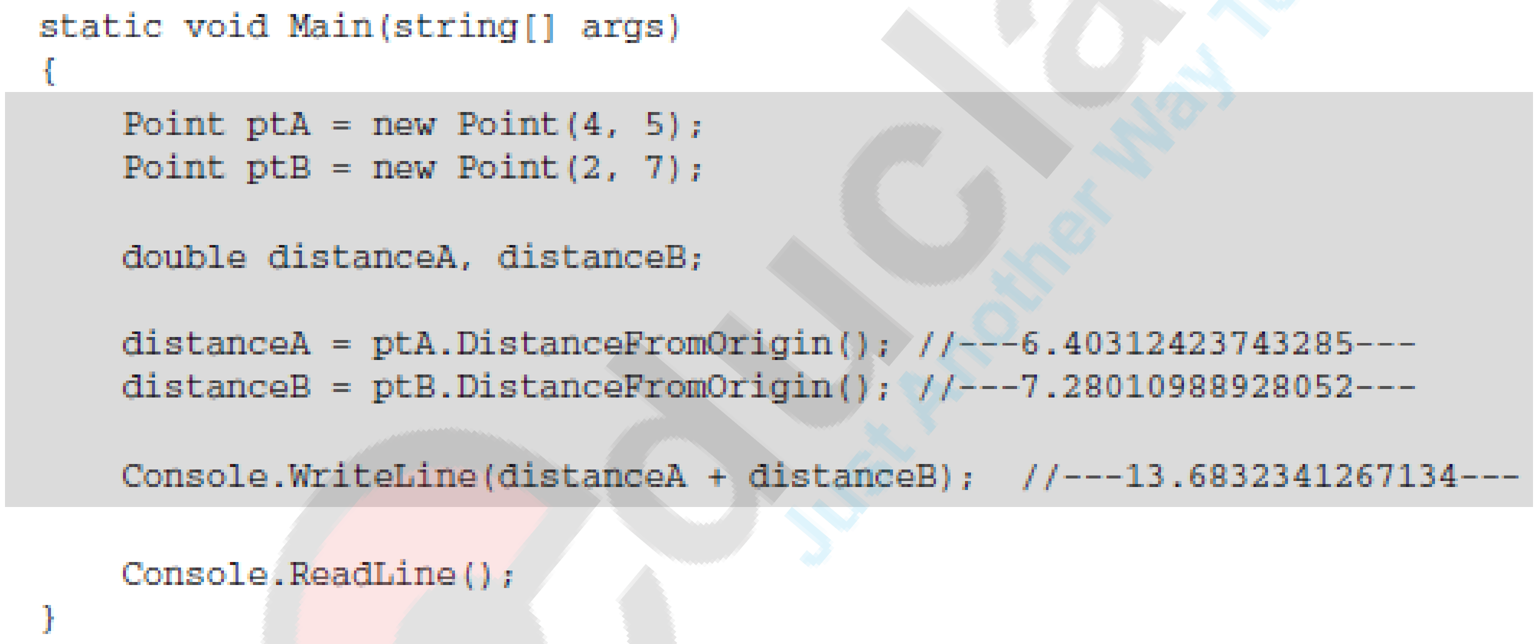
(Compile time Polymorphism)

To see how operator overloading works, consider the following program containing the Point class representing a point in a coordinate system:



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

namespace OperatorOverloading
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }

    class Point
    {
        public Single X { get; set; }
        public Single Y { get; set; }

        public Point(Single X, Single Y)
        {
            this.X = X;
            this.Y = Y;
        }
        public double DistanceFromOrigin()
        {
            return (Math.Sqrt(Math.Pow(this.X, 2) + Math.Pow(this.Y, 2)));
        }
    }
}
```

- The Point class contains two public properties (X and Y), a constructor, and a method — DistanceFromOrigin() .
- If you constantly perform calculations where you need to add the distances of two points (from the origin), your code may look like this:



```
static void Main(string[] args)
{
    Point ptA = new Point(4, 5);
    Point ptB = new Point(2, 7);

    double distanceA, distanceB;

    distanceA = ptA.DistanceFromOrigin(); //---6.40312423743285---
    distanceB = ptB.DistanceFromOrigin(); //---7.28010988928052---

    Console.WriteLine(distanceA + distanceB);  //---13.6832341267134---

    Console.ReadLine();
}
```

- A much better implementation is to overload the + operator for use with the Point class.

To overload the + operator, define a public static operator within the Point class as follows:

```
class Point
{
    public Single X { get; set; }
    public Single Y { get; set; }

    public Point(Single X, Single Y)
    {
        this.X = X;
        this.Y = Y;
    }
    public double DistanceFromOrigin()
    {
        return (Math.Sqrt(Math.Pow(this.X, 2) + Math.Pow(this.Y, 2)));
    }

    public static double operator +(Point A, Point B)
    {
        return (A.DistanceFromOrigin() + B.DistanceFromOrigin());
    }
}
```

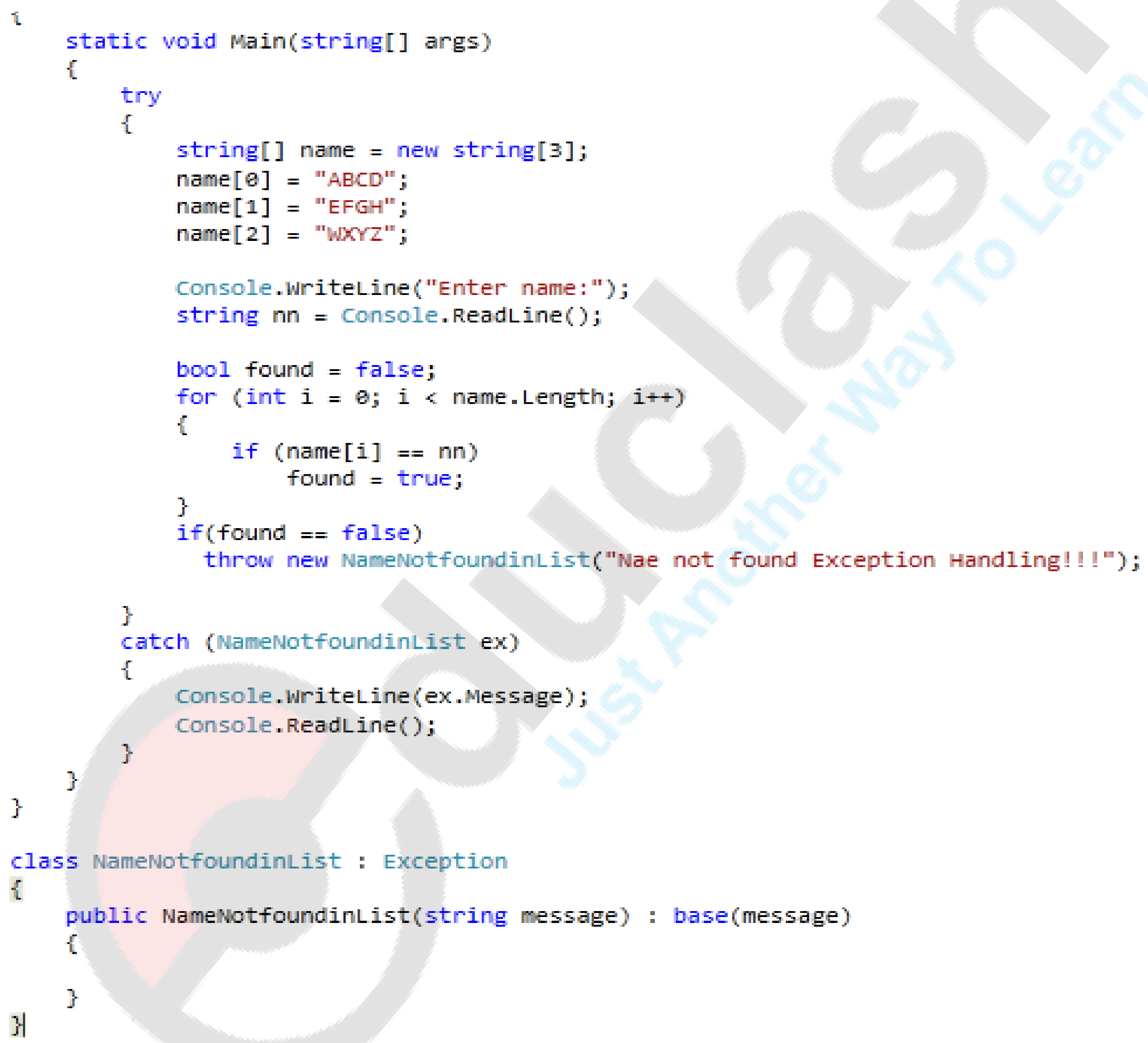
The operator keyword overloads a built - in operator.

```
static void Main(string[] args)
{
    Point ptA = new Point(4, 5);
    Point ptB = new Point(2, 7);

    Console.WriteLine(ptA + ptB);  //---13.6832341267134---
    Console.ReadLine();
}
```

Exception Handling

- Exceptions are a type of error that occurs during the execution of an application.
- **try** – A try block is used to encapsulate a region of code. If any code throws an exception within that try block, the exception will be handled by the corresponding catch.
- **catch** – When an exception occurs, the Catch block of code is executed. This is where you are able to handle the exception, log it, or ignore it. Multiple catch blocks are allowed.
- **finally** – The finally block allows you to execute certain code if an exception is thrown or not. For example, disposing of an object that must be disposed of.
- **throw** – The throw keyword is used to actually create a new exception that is the bubbled up to a try catch finally block.

Exception Handling



```
{
    static void Main(string[] args)
    {
        try
        {
            string[] name = new string[3];
            name[0] = "ABCD";
            name[1] = "EFGH";
            name[2] = "WXYZ";

            Console.WriteLine("Enter name:");
            string nn = Console.ReadLine();

            bool found = false;
            for (int i = 0; i < name.Length; i++)
            {
                if (name[i] == nn)
                    found = true;
            }
            if(found == false)
              throw new NameNotfoundinList("Nae not found Exception Handling!!!");

        }
        catch (NameNotfoundinList ex)
        {
            Console.WriteLine(ex.Message);
            Console.ReadLine();
        }
    }
}

class NameNotfoundinList : Exception
{
    public NameNotfoundinList(string message) : base(message)
    {

    }
}
```

C#

Fundamentals

Part 2

Ends