Introduction to Object-Oriented Programming

Lesson 2

Objective Domain Matrix

Skills/Concepts	MTA Exam Objectives
Understanding Objects	Understand the fundamentals of classes (2.1)
Understanding Values and References	Understand computer storage and data types (1.1)
Understanding Encapsulation	Understand encapsulation (2.4)
Understanding Inheritance	Understand inheritance (2.2)
Understanding Polymorphism	Understand polymorphism (2.3)
Understanding Interfaces	Understand encapsulation (2.4)

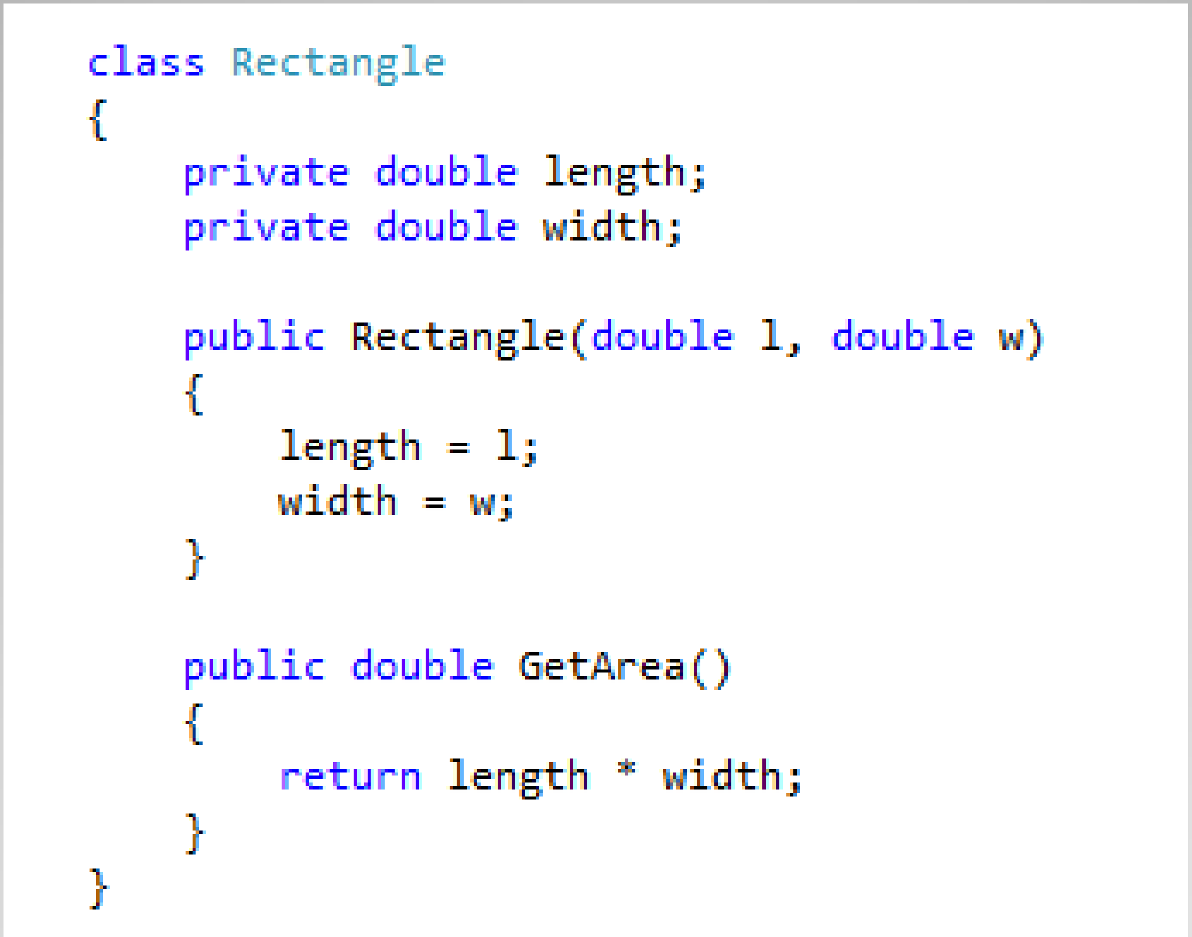
Objects

- Object-oriented programming is a programming technique that makes use of objects.
- Objects are self-contained data structures that consist of properties, methods, and events.
  - Properties specify the data represented by an object
  - Methods specify an object’s behavior
  - Events provide communication between objects

Classes

- A class defines a blueprint for an object.
- A class defines how the objects should be built and how they should behave.
- An object is also known as an instance of a class.

Defining a C# Class



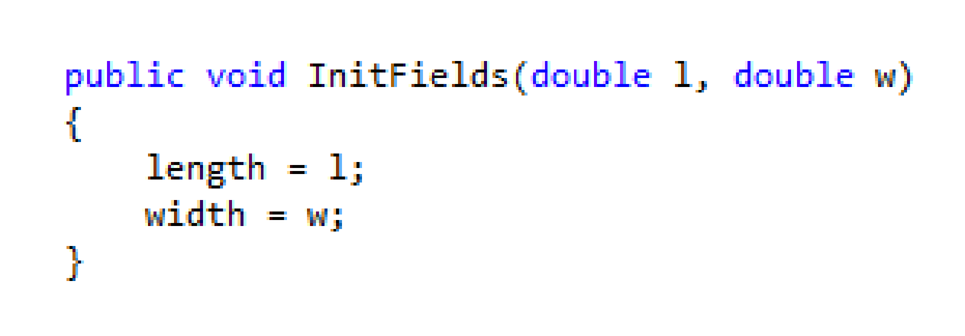
```
class Rectangle
{
    private double length;
    private double width;

    public Rectangle(double l, double w)
    {
        length = l;
        width = w;
    }

    public double GetArea()
    {
        return length * width;
    }
}
```

Methods

- A method is a block of code containing a series of statements.
- A method defines the actions or operations supported by a class.
- A method is defined by specifying the access level, the return type, the name of the method, and an optional list of parameters in parentheses followed by a block of code enclosed in braces.

Method Example

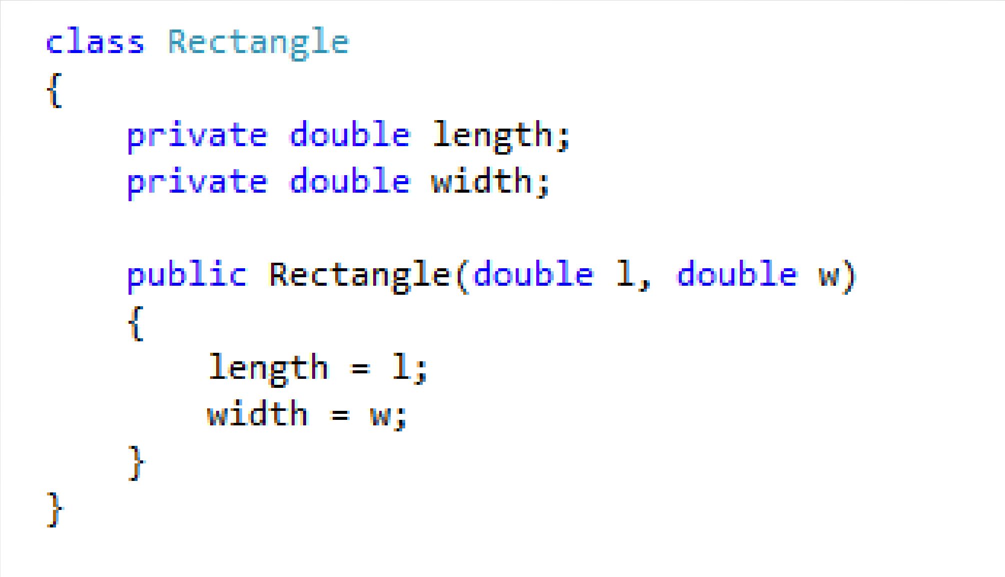
- The **InitFields** method takes two parameters and uses the parameter values to respectively assign the data field length and width.
- When a method’s return type is void, a return statement with no value can be used.
- If a return statement is not used, as in the **InitFields** method, the method will stop executing when it reaches the end of the code block.



```
public void InitFields(double l, double w)
{
    length = l;
    width = w;
}
```

Constructors

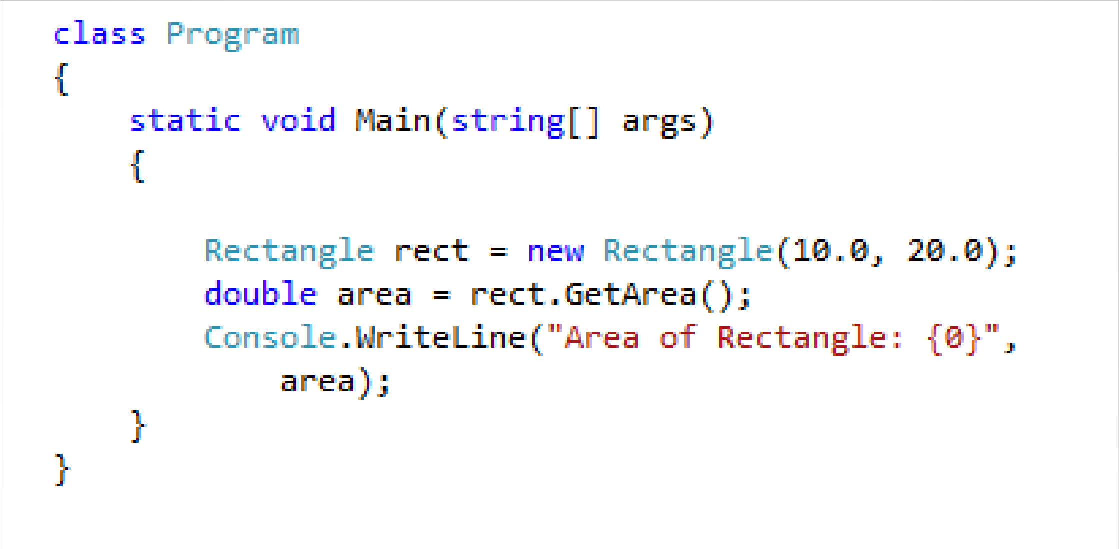
- Constructors are special class methods that are executed when a new instance of a class is created.
- Constructors are used to initialize the data members of the object.
- Constructors must have exactly the same name as the class and they do not have a return type.
- Multiple constructors, each with a unique signature, can be defined for a class.



```
class Rectangle
{
    private double length;
    private double width;

    public Rectangle(double l, double w)
    {
        length = l;
        width = w;
    }
}
```

Creating Objects

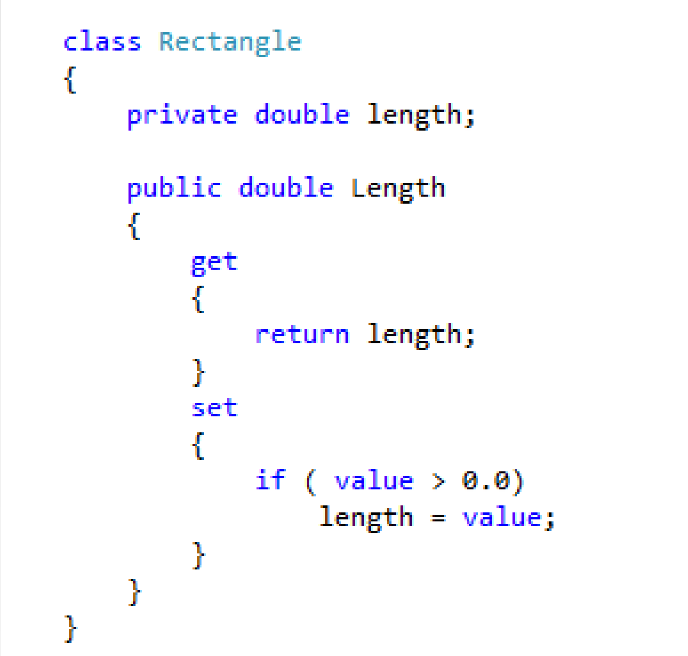
- Objects need a template that defines how they should be built.
- All objects created from the same template look and behave in a similar way.



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

        Rectangle rect = new Rectangle(10.0, 20.0);
        double area = rect.GetArea();
        Console.WriteLine("Area of Rectangle: {0}",
            area);
    }
}
```

Properties

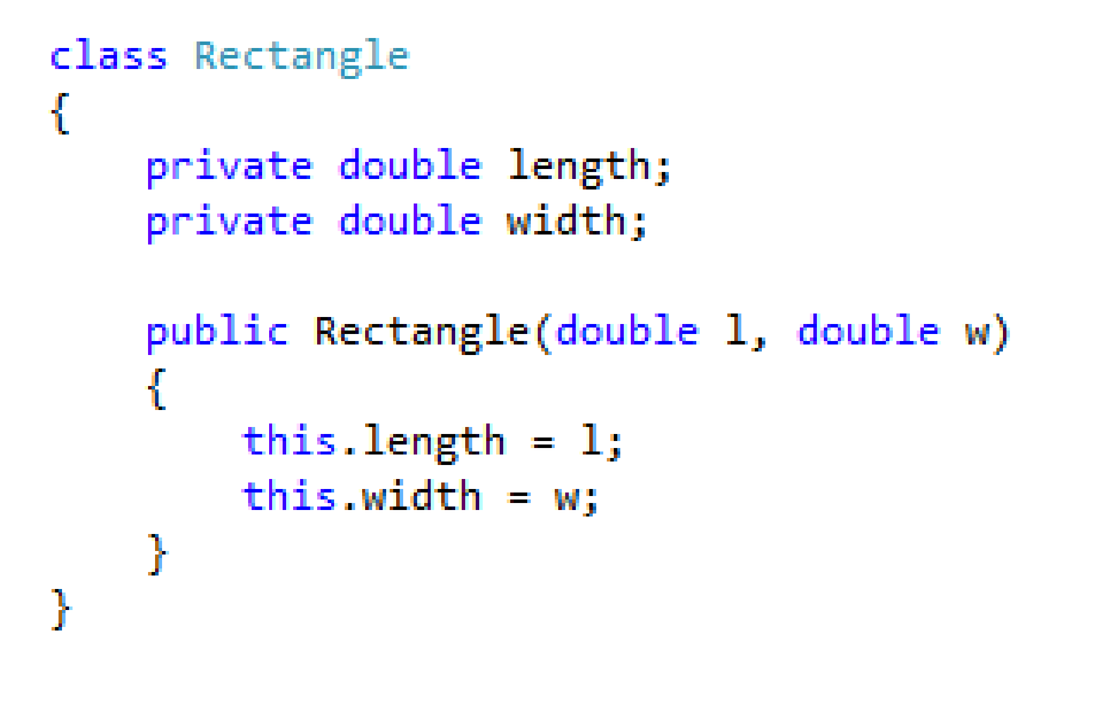
- Properties are class members that can be accessed like data fields but contain code like a method.
- A property has two accessors, **get** and **set**. The get accessor is used to return the property value, and the set accessor is used to assign a new value to the property.



```
class Rectangle
{
    private double length;

    public double Length
    {
        get
        {
            return length;
        }
        set
        {
            if ( value > 0.0)
                length = value;
        }
    }
}
```

The this Keyword

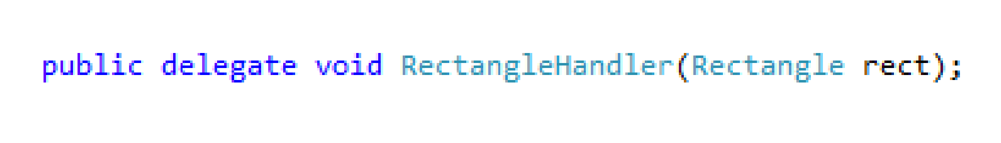
- The **this** keyword is a reference to the current instance of the class.
- You can use the this keyword to refer to any member of the current object.



```
class Rectangle
{
    private double length;
    private double width;

    public Rectangle(double l, double w)
    {
        this.length = l;
        this.width = w;
    }
}
```

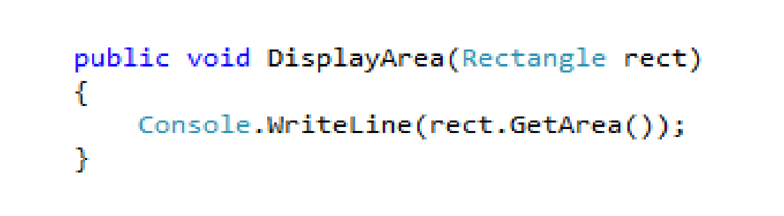
Delegates

- Delegates are special objects that can hold a reference to a method with a specific signature.



```
public delegate void RectangleHandler(Rectangle rect);
```

- Here, you define a **RectangleHandler** delegate that can hold references to a method that returns void and accepts a single parameter of the Rectangle type.

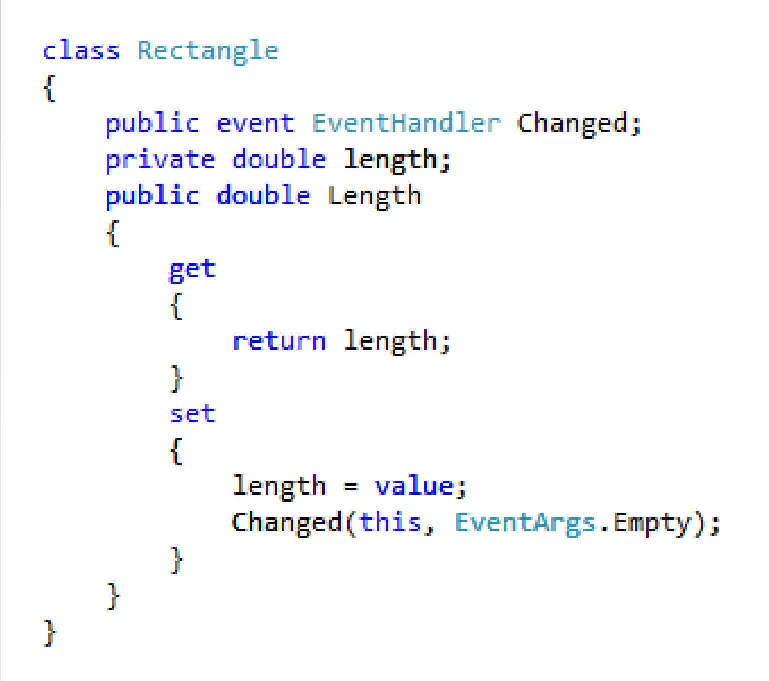


```
public void DisplayArea(Rectangle rect)
{
    Console.WriteLine(rect.GetArea());
}
```

- The signature of **DisplayArea** method matches the **RectangleHandler** delegate and therefore can be assigned to one of its instance.

Events

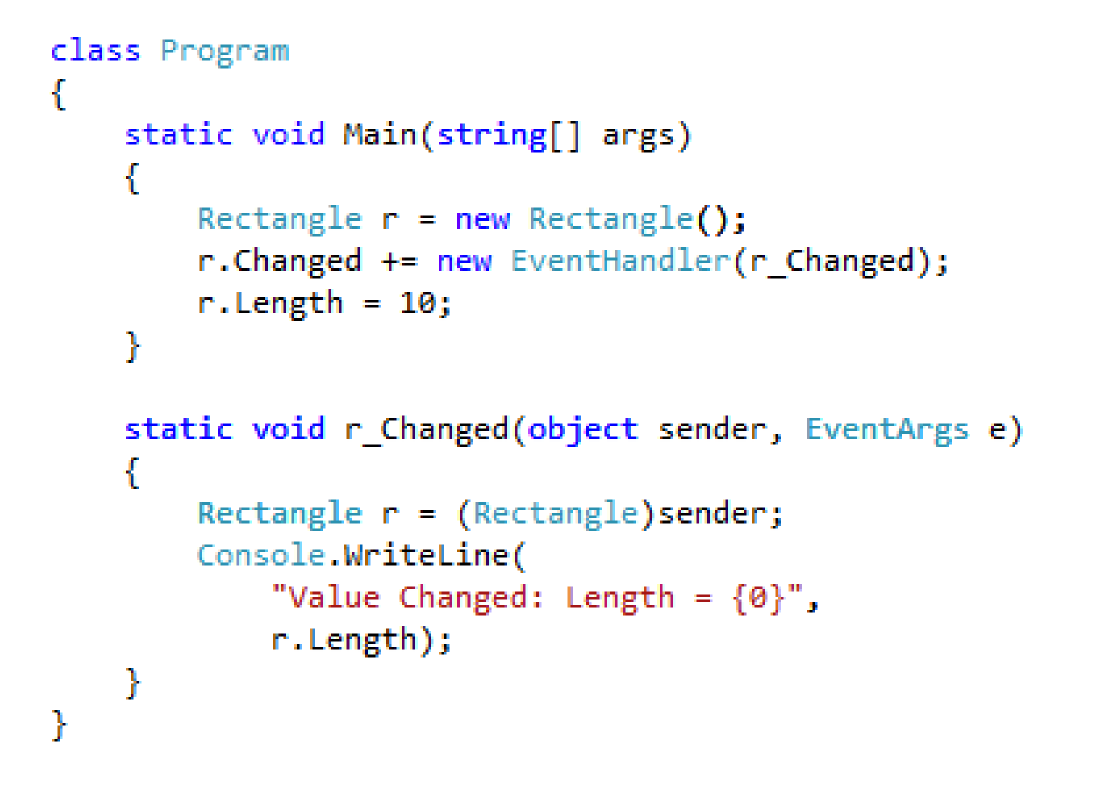
- Events are a way for a class to notify other classes or objects when something of interest happens.
- The class that sends the notification is called a publisher of the event.
- The class that receives the notification is called the subscriber of the event.



```
class Rectangle
{
    public event EventHandler Changed;
    private double length;
    public double Length
    {
        get
        {
            return length;
        }
        set
        {
            length = value;
            Changed(this, EventArgs.Empty);
        }
    }
}
```

Subscribing to Events

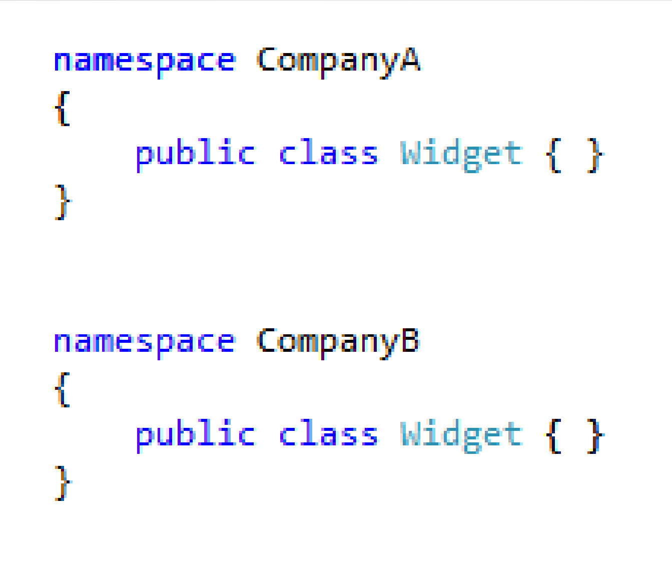
- The signature of the event handler method matches the requirements of the event's delegate.



```
class Program
{
    static void Main(string[] args)
    {
        Rectangle r = new Rectangle();
        r.Changed += new EventHandler(r_Changed);
        r.Length = 10;
    }

    static void r_Changed(object sender, EventArgs e)
    {
        Rectangle r = (Rectangle)sender;
        Console.WriteLine(
            "Value Changed: Length = {0}",
            r.Length);
    }
}
```

Namespaces

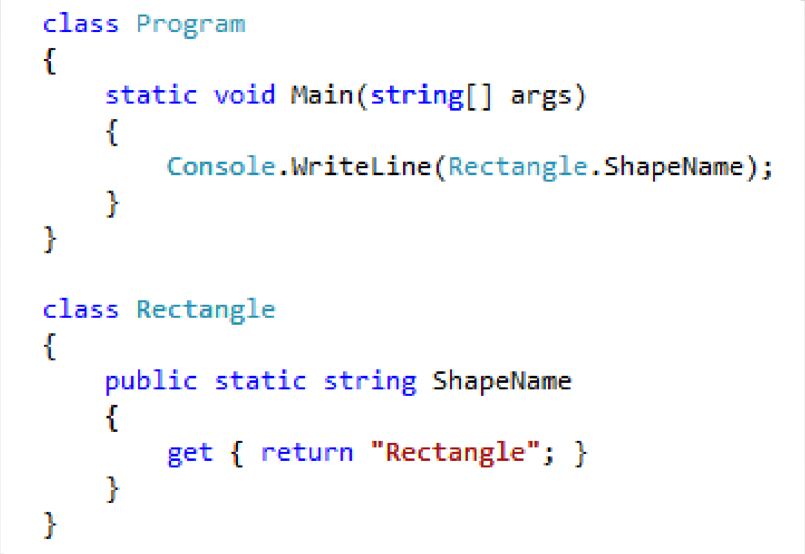
- A namespace is a language element that allows you to organize code and create globally unique class names.
- The .NET Framework uses namespaces to organize all its classes.
  - The System namespace groups all the fundamental classes.
  - The System.Data namespace organizes classes for data access.
  - The System.Web namespace is used for Web-related classes.



```
namespace CompanyA
{
    public class Widget { }
}


namespace CompanyB
{
    public class Widget { }
}
```

Static Members

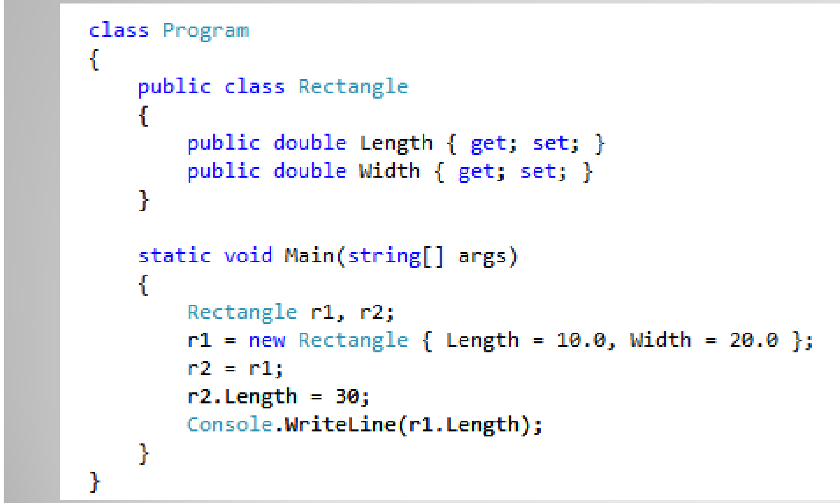
- The **static** keyword is used to declare members that do not belong to individual objects but to a class itself.
- When an instance of a class is created, a separate copy is created for each instance field, but only one copy of a static field is shared by all instances.
- A static member cannot be referenced through an instance object. Instead, a static member is referenced through the class name.



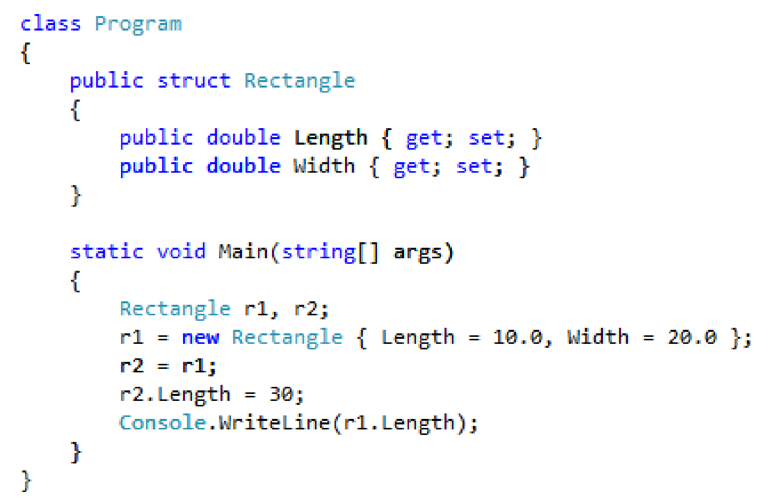
```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(Rectangle.ShapeName);
    }
}

class Rectangle
{
    public static string ShapeName
    {
        get { return "Rectangle"; }
    }
}
```

Values and References

- A value type directly stores data within its memory.
- Reference types store only a reference to a memory location. The actual data is stored at the memory location being referred to.
- When you copy a reference type variable to another variable of the same type, only the references are copied. As a result, after the copy, both variables will point to the same object.



```
class Program
{
    public class Rectangle
    {
        public double Length { get; set; }
        public double Width { get; set; }
    }

    static void Main(string[] args)
    {
        Rectangle r1, r2;
        r1 = new Rectangle { Length = 10.0, Width = 20.0 };
        r2 = r1;
        r2.Length = 30;
        Console.WriteLine(r1.Length);
    }
}
```



```
class Program
{
    public struct Rectangle
    {
        public double Length { get; set; }
        public double Width { get; set; }
    }

    static void Main(string[] args)
    {
        Rectangle r1, r2;
        r1 = new Rectangle { Length = 10.0, Width = 20.0 };
        r2 = r1;
        r2.Length = 30;
        Console.WriteLine(r1.Length);
    }
}
```

Encapsulation

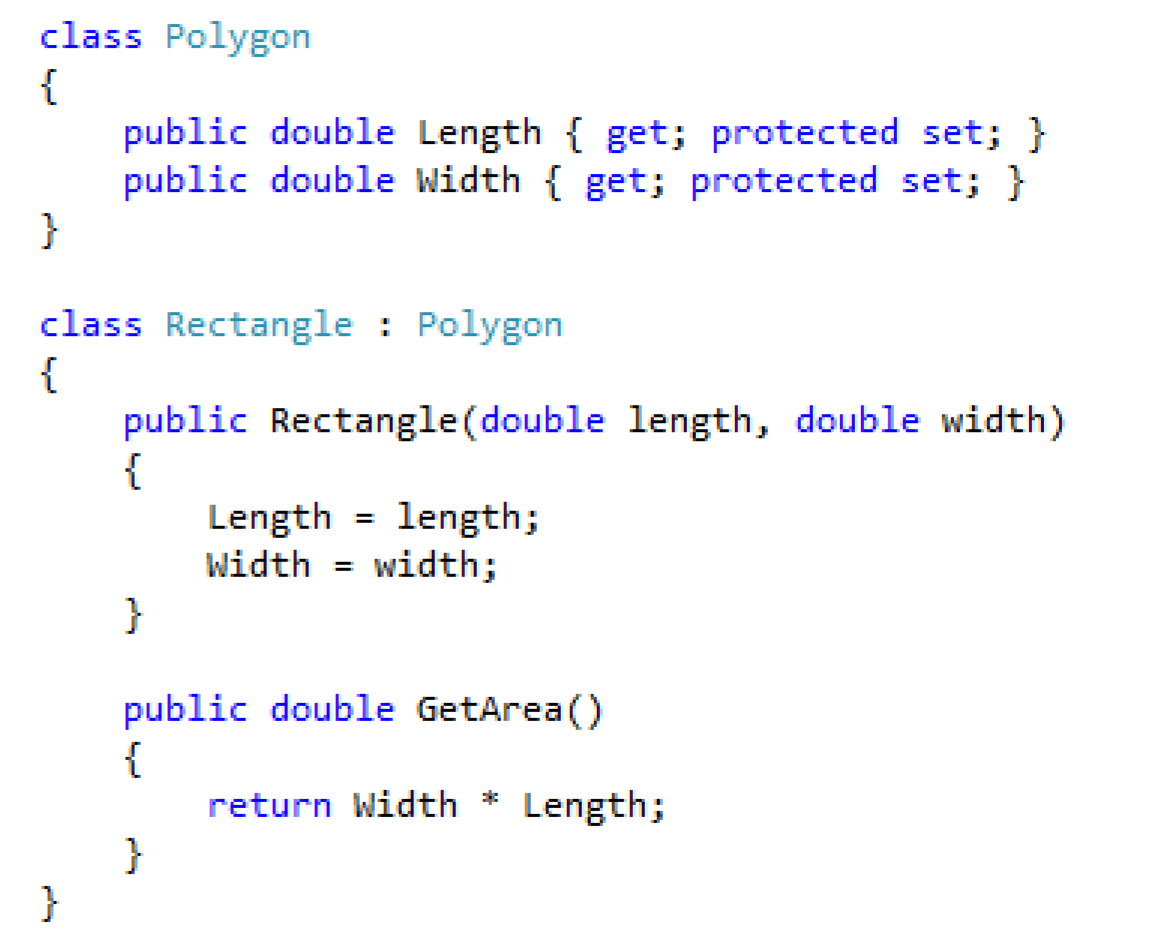
- Encapsulation is a mechanism to restrict access to a class or class members in order to hide design decisions that are likely to change.
- Access modifiers control where a type or type member can be used.

Access modifier	Description
public	Access is not restricted.
private	Access is restricted to the containing class.
protected	Access is restricted to the containing class and to any class that is derived directly or indirectly from the containing class.
internal	Access is restricted to the code in the same assembly.
protected internal	A combination of protected and internal—that is, access is restricted to any code in the same assembly and only to derived classes in another assembly.

Inheritance

- Inheritance is an OOP feature that allows you to develop a class once, and then reuse that code over and over as the basis of new classes.
- The class whose functionality is inherited is called a base class.
- The class that inherits the functionality is called a derived class
- A derived class can also define additional features that make it different from the base class.
- Unlike classes, the structs do not support inheritance.

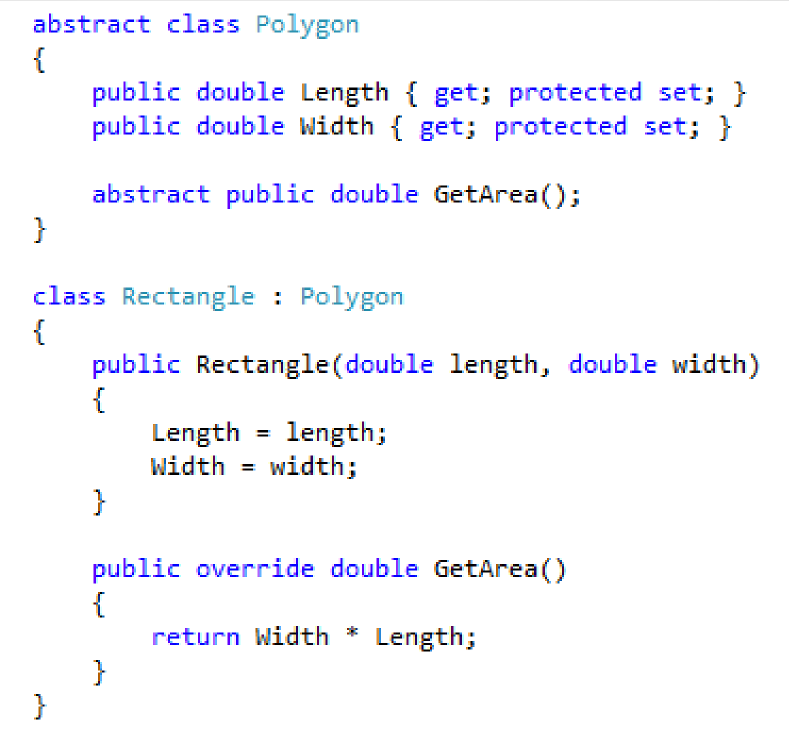
Inheritance - Example



```
class Polygon
{
    public double Length { get; protected set; }
    public double Width { get; protected set; }
}

class Rectangle : Polygon
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double GetArea()
    {
        return Width * Length;
    }
}
```

Abstract Classes

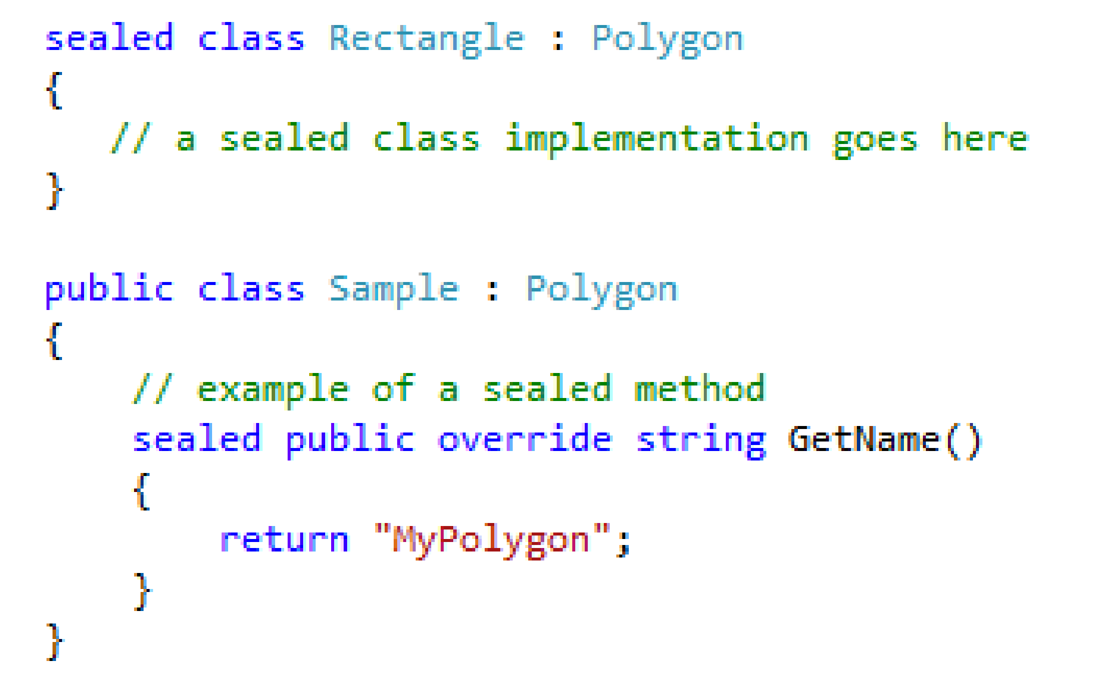
- The **abstract** classes provide a common definition of a base class that can be shared by multiple derived classes.
- The **abstract** class often provides incomplete implementation.
- To instantiate an abstract class you must inherit from it and complete its implementation.



```
abstract class Polygon
{
    public double Length { get; protected set; }
    public double Width { get; protected set; }

    abstract public double GetArea();
}

class Rectangle : Polygon
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public override double GetArea()
    {
        return Width * Length;
    }
}
```

Sealed Classes

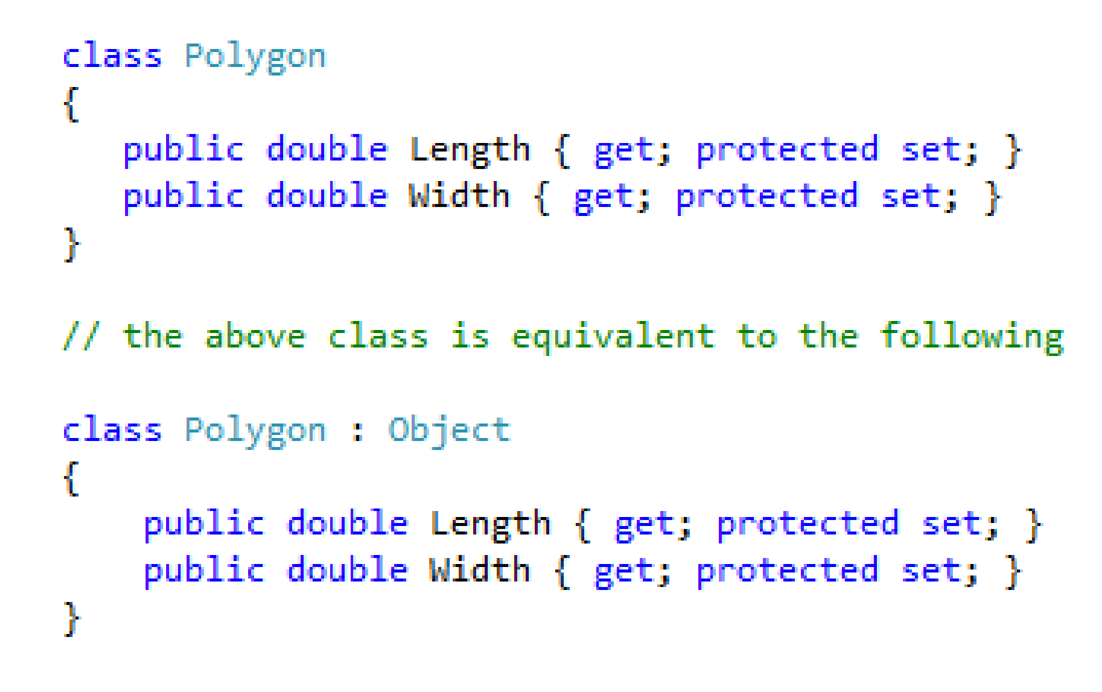
- The **sealed** classes provide complete functionality but cannot be used as base classes.
- Use the **sealed** keyword when your implementation is complete and you do not want a class or its member to be inherited.



```
sealed class Rectangle : Polygon
{
   // a sealed class implementation goes here
}

public class Sample : Polygon
{
    // example of a sealed method
    sealed public override string GetName()
    {
        return "MyPolygon";
    }
}
```

Inheriting from Object

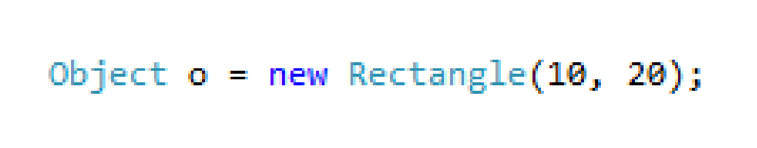
- The **Object** class is the ultimate base class of all the classes in the .NET Framework.
- All classes in the .NET Framework inherit either directly or indirectly from the **Object** class.



```
class Polygon
{
   public double Length { get; protected set; }
   public double Width { get; protected set; }
}

// the above class is equivalent to the following

class Polygon : Object
{
    public double Length { get; protected set; }
    public double Width { get; protected set; }
}
```

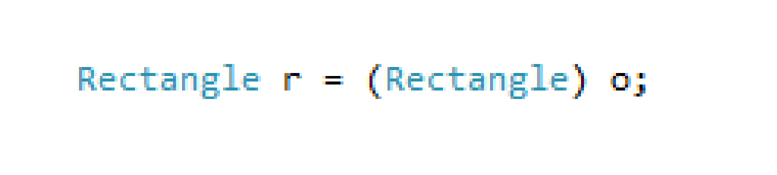
Casting

- In C#, you can cast an object to any of its base types.
- All classes in the .NET Framework inherit either directly or indirectly from the Object class.
- Assigning a derived class object to a base class object doesn’t require any special syntax:



```
Object o = new Rectangle(10, 20);
```

- Assigning a base class object to a derived class object must be explicitly cast:

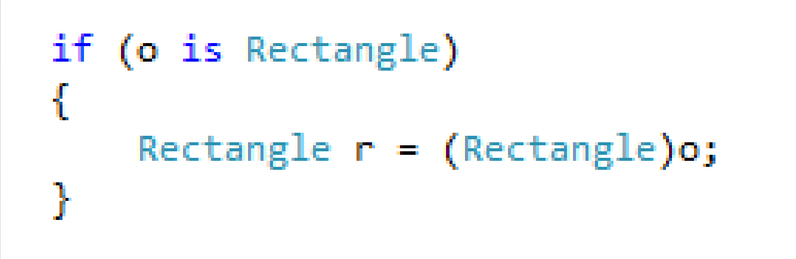


```
Rectangle r = (Rectangle) o;
```

- At execution time, if the value of o is not compatible with the **Rectangle** class, the runtime throws a System.InvalidCastException.

The is Operator

- To avoid runtime errors such as **InvalidCastException**, the **is** operator can be used to check whether the cast is allowed before actually performing the cast.

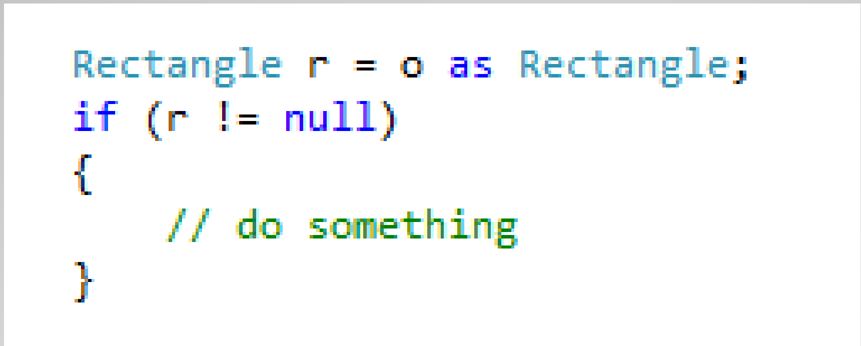


```
if (o is Rectangle)
{
    Rectangle r = (Rectangle)o;
}
```

- Here, the runtime checks the value of the object **o.** The **cast** statement is only executed if **o** contains a **Rectangle** object.

The as Operator

- The **as** operator is similar to the cast operation but, in the case of as, if the type conversion is not possible, null is returned instead of raising an exception.



```
Rectangle r = o as Rectangle;
if (r != null)
{
    // do something
}
```

- At runtime, if it is not possible to cast the value of variable **o** to a rectangle, a value of null is assigned to the variable **r**. No exceptions will be raised.

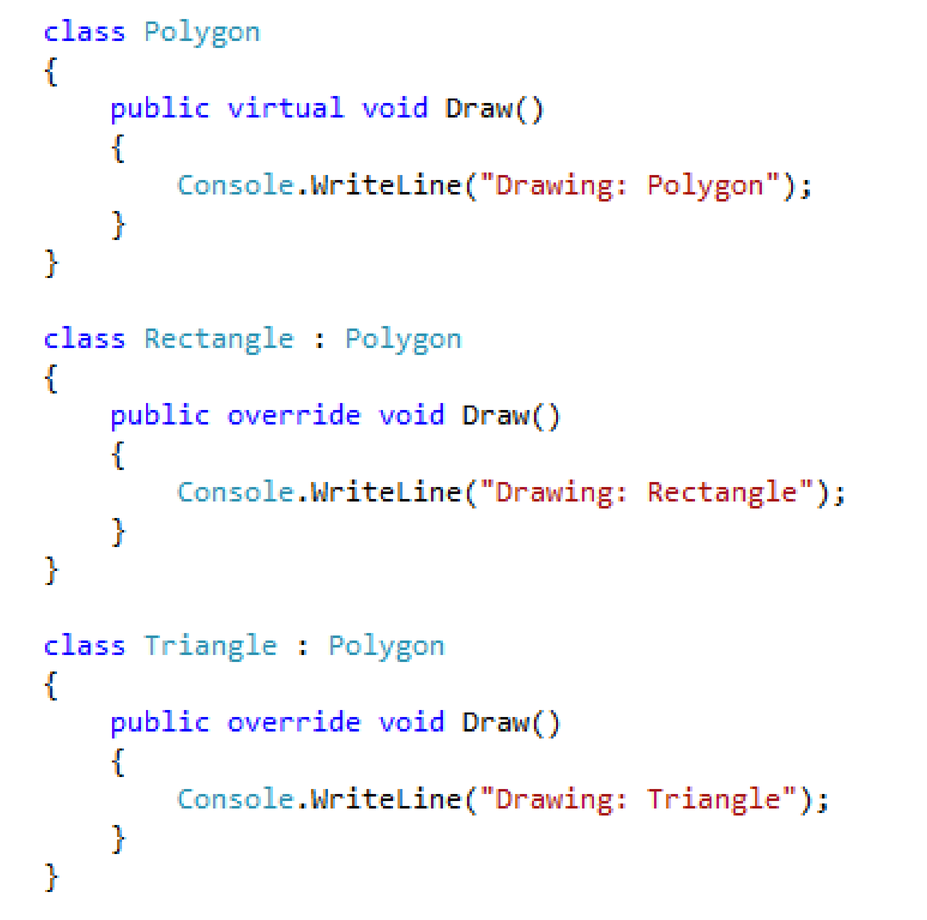
Polymorphism

- Polymorphism is the ability of derived classes to share common functionality with base classes but still define their own unique behavior.

- Polymorphism allows the objects of a derived class to be treated at runtime as objects of the base class. When a method is invoked at runtime, its exact type is identified, and the appropriate method is invoked from the derived class.

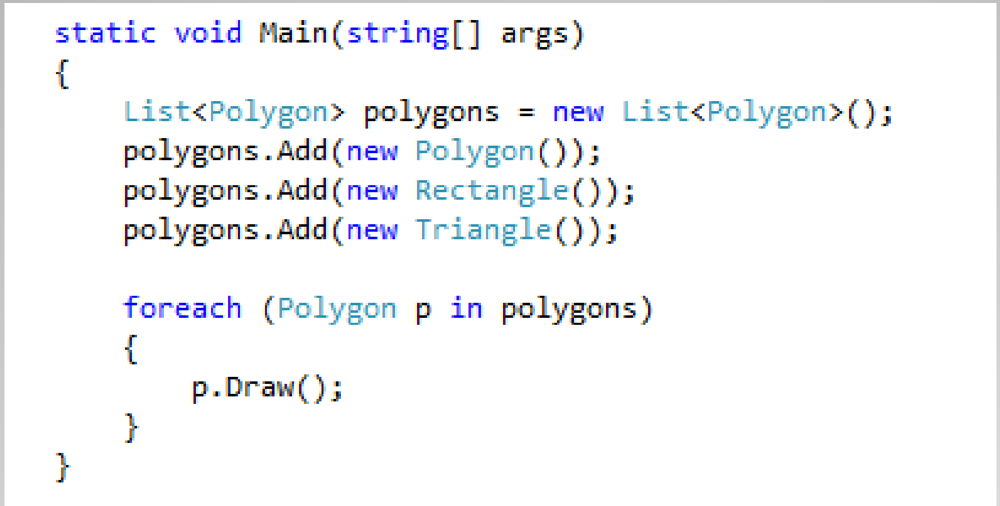
Polymorphism - Example

- Consider the following set of classes:

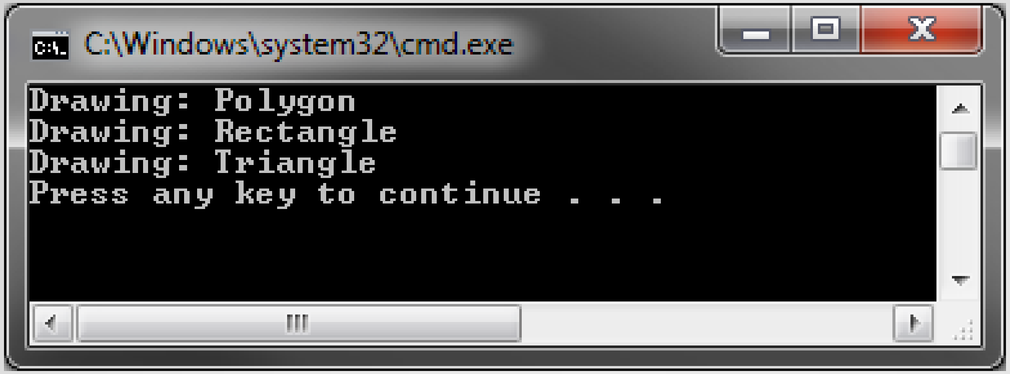


```
class Polygon
{
    public virtual void Draw()
    {
        Console.WriteLine("Drawing: Polygon");
    }
}

class Rectangle : Polygon
{
    public override void Draw()
    {
        Console.WriteLine("Drawing: Rectangle");
    }
}

class Triangle : Polygon
{
    public override void Draw()
    {
        Console.WriteLine("Drawing: Triangle");
    }
}
```

Polymorphism - Example



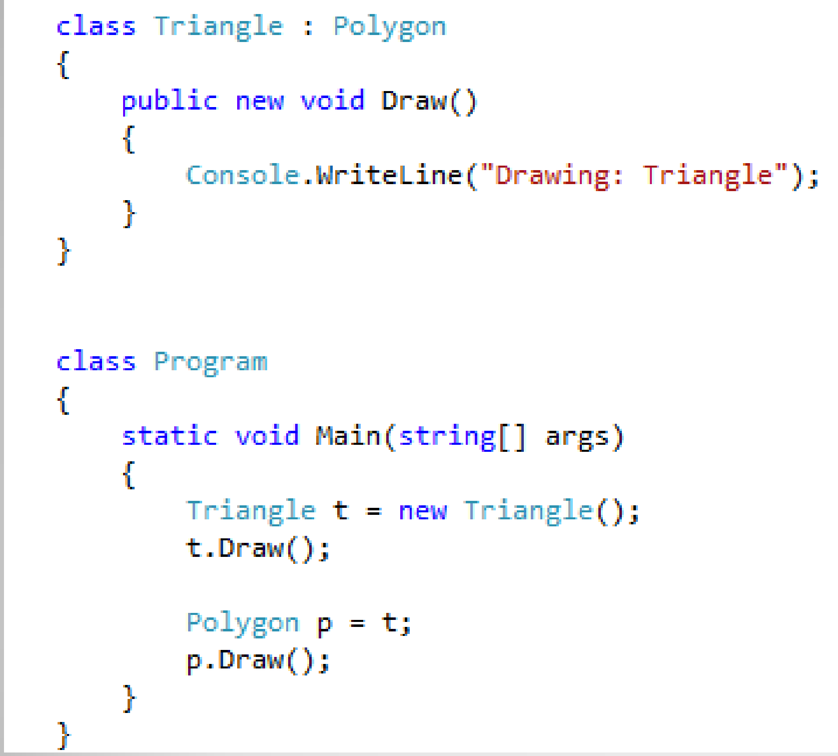
```
static void Main(string[] args)
{
    List<Polygon> polygons = new List<Polygon>();
    polygons.Add(new Polygon());
    polygons.Add(new Rectangle());
    polygons.Add(new Triangle());

    foreach (Polygon p in polygons)
    {
        p.Draw();
    }
}
```



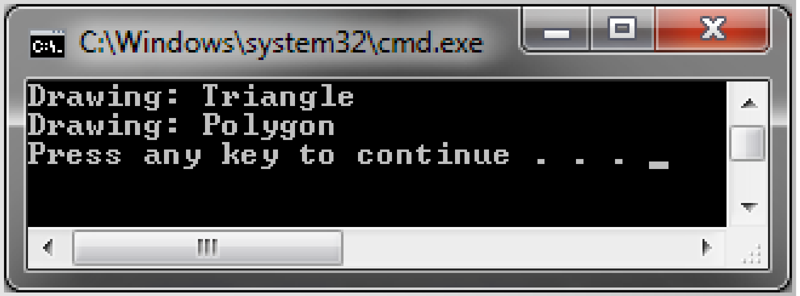
```
Drawing: Polygon
Drawing: Rectangle
Drawing: Triangle
Press any key to continue . . .
```

The override and new Keywords

- The **override** keyword replaces a base class member in a derived class.
- The **new** keyword creates a new member of the same name in the derived class and hides the base class implementation.



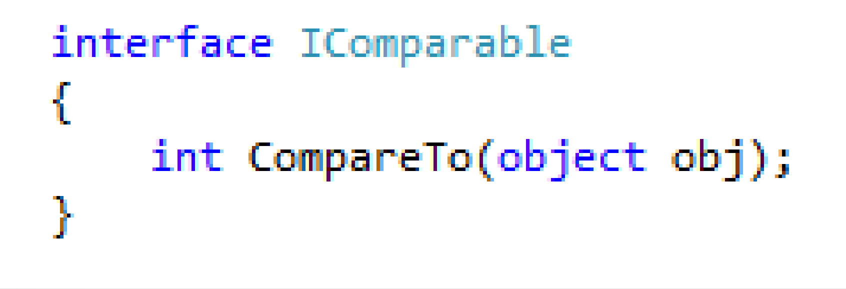
```
class Triangle : Polygon
{
    public new void Draw()
    {
        Console.WriteLine("Drawing: Triangle");
    }
}


class Program
{
    static void Main(string[] args)
    {
        Triangle t = new Triangle();
        t.Draw();

        Polygon p = t;
        p.Draw();
    }
}
```



```
Drawing: Triangle
Drawing: Polygon
Press any key to continue . . . _
```

Interfaces

- Interfaces are used to establish contracts through which objects can interact with each other without knowing the implementation details.
- An interface definition cannot consist of any data fields or any implementation details such as method bodies.
- A common interface defined in the System namespace is the **IComparable** namespace. This is a simple interface defined as follows:



```
interface IComparable
{
    int CompareTo(object obj);
}
```

- Each class that implements **IComparable** is free to provide its own custom comparison logic inside the **CompareTo** method.

Recap

- Objects
  - Classes, methods, properties, delegates, events
  - Namespaces
  - Static members
- Values and References
- Encapsulation
  - Access Modifiers
- Inheritance
  - Abstract and sealed classes
  - Casting, is and as operators
- Polymorphism
  - Override and new keywords
- Interfaces