Lesson 2
Introduction to Object-Oriented Programming

Learning Objectives

Students will learn about:

- Objects
- Values and References
- Encapsulation
- Inheritance
- Polymorphism
- Interfaces

OBJECTIVE1: Understanding 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, and events provide communication between objects.

Thinking in an Object-Oriented Way

A software object is conceptually similar to a real-world object

A great way to start thinking in an object-oriented way is to look at real-world objects, such as cars, phones, music players, etc. You'll notice that these objects all have state and behavior. For example, cars have not only various states (e.g., model name, color, current speed, fuel level), but also various behaviors (e.g., accelerate, brake, change gear). Similarly, you'll notice that some objects are simple, whereas others are complex. Most complex objects (such as a car) are made up of smaller objects that in turn have their own state and behavior. You'll also notice that although a car is a complex object, you only need to know a few things in order to interact with it. As you drive a car, for example, you simply invoke a behavior such as accelerate or brake; you are spared from knowing the many thousands of internal details at work under the hood.

A software object is conceptually similar to a real-world object. Within the software environment, an object stores its state in fields and exposes its behavior through methods. When a method is invoked on an object, you get a well-defined functionality without the need to worry about the inner complexity of the object or the method itself. This concept of hiding complexity is called encapsulation, and it is one of many features of objected-oriented programming that you'll learn more about in this lesson

Understanding Classes

A class is the template from which individual objects are created.
In the real world, 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. For example, think about a particular make and model of car.

In the software world, a class is the template from which individual objects are created. An object is also known as an instance of a class.

A class is a construct that enables you to create your own custom types by grouping together variables of other types, methods and events. A class is like a blueprint. It defines the data and behavior of a type. If the class is not declared as static, client code can use it by creating objects or instances which are assigned to a variable. The variable remains in memory until all references to it go out of scope. At that time, the CLR marks it as eligible for garbage collection. If the class is declared as static, then only one copy exists in memory and client code can only access it through the class itself, not an instance variable.
Unlike structs, classes support inheritance, a fundamental characteristic of object-oriented programming.

Declaring Classes

Classes are declared by using the class keyword, as shown in the following example:

A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces.

Following is the general form of a class definition:

```
<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
   }
 }
```

The class keyword is preceded by the access level. If access level is public then anyone can create objects from this class. The name of the class follows the class keyword. The remainder of the definition is the class body, where the behavior and data are defined. Fields, properties, methods, and events on a class are collectively referred to as class members.

Please note that

1. Access specifiers specify the access rules for the members as well as the class itself, if not mentioned then the default access specifier for a class type is internal. Default access for the members is private.
2. Data type specifies the type of variable, and return type specifies the data type of the data, the method returns, if any.
3. To access the class members, you will use the dot (.) operator.
4. The dot operator links the name of an object with the name of a member.

Illustration1: CREATE A CLASS

GET READY. Before you begin these steps, be sure to launch Microsoft Visual Studio and open a new Console Application Project named Lesson02. Then, perform the following tasks:

1. Add a new Visual C# class named Rectangle at the end of Program class to the project
2. Replace the code for the Rectangle class with the following code:

```
class Rectangle
{
private double length;
private double width;
public Rectangle(double l, double w)
{
length = l;
width = w;
}
public double GetArea()
{
return length * width;
}
}
```

3. Select Build > Lesson02 to build the project. Ensure that there are no errors.

PAUSE. Leave the project open to use in the next exercise.

TakeNote

Each class is a definition of a new data type. Therefore, a class definition is sometimes also referred to as a type.

You have just created a new C# class named Rectangle. A new class is defined by using the keyword class. Here, the Rectangle class has two data fields, length and width. These fields are defined by using the access modifier private. An access modifier specifies what region of the code will have access to a field. For example, the access modifier public will not limit access, but the access modifier private will limit access within the class in which the field is defined.

This class also defines a method named GetArea. But what, exactly, is a method?

UNDERSTANDING METHODS

A method is a block of code containing a series of statements.

In the software world, 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. For instance, in the previous example, the class Rectangle defines a single method named GetArea. For GetArea, the access level is public, the return type is double, the method name is GetArea, the parameter list is empty, and the block of code is a single return statement.

A method can return a value to the calling code. If a method does not intend to return any value, its return type is specified by the keyword void. The method must use a return statement to return a value. The return statement terminates the execution of the method and returns the specified value to the calling code. The data type of the value returned from a method must match the return type specified on the method's declaration line.

To return to the earlier example, the return type of the method GetArea is double, which means that the GetArea method must return a value of the type double. The GetArea method satisfies this requirement by returning the expression length * width, which is a double value.

Takenote

A method's name, its parameter list, and the order of data types of the parameters are collectively recognized as the method's signature. A method signature must be unique within a class.

The following code defines an InitFields method that takes two parameters of type double and returns

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

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. The InitFields method can be used to properly initialize the value of the data fields, but as you'll learn in the following section, constructors already give you a way of initializing a class

UNDERSTANDING CONSTRUCTORS

Constructors are used to initialize the data members of the object.

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.

A constructor that takes no arguments is called the default constructor. If a class is defined without any constructor, an invisible default constructor that does absolutely nothing is automatically generated.

What do constructors look like?

A constructor looks very much like a method, but with no return type and a name which is the same as the name of the class. (The modifiers you can use with a constructor are slightly different, however - see the spec for more information about that.) Here's a short class to use as an example:

```
public class MySimpleClass
{
   public MySimpleClass (int x)
   {
      Console.WriteLine (x);
   }
}
```

This class has a single constructor, which takes an int as a parameter, and prints that int's value to the console.

It is often useful to have additional constructors to provide more ways through which an object is initialized. The Rectangle class, defined earlier, is only one way to create and initialize its object: by calling the constructor that accepts two parameters, both of the default data type.

CREATING OBJECTS

Objects are created from the templates defined by classes

Illustration2: CREATE AN OBJECT

GET READY. For this activity, use the console application project (Lesson 02) that you created in the previous illustration. Then, perform these steps:

1. Modify the code of the Program class to the following:

```
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);
}
}
```

2. Select Debug > Start Without Debugging. A console window will pop up to display the area of the rectangle.
3. SAVE your project.

PAUSE. Leave the project open to use in the next exercise

The class Rectangle provides only one way to construct an instance of the class: by calling a constructor with two arguments of the double data type. Here, you create an object by using the new keyword followed by the call to the appropriate class constructor.

When the code executes, an object of Rectangle type is created in the heap memory. A reference to this memory is stored inside the variable rect, and the variable rect is stored on the stack. Later in this block of code, you can use rect to refer to and manipulate the object that was just created.

Using the object’s reference, you can access the class members. For example, the code calls the method GetArea on the object, and the value returned by the method is stored in the variable area. The data fields, length and width, of the object rect are not accessible here because they are marked as private in the class definition.

Take note

Classes and objects are different. A class defines the template for an object but is not an object itself. On the other hand, an object is a concrete instance of a class but is not a class itself

UNDERSTANDING PROPERTIES

Properties allow you to access class data in a safe and flexible way.

Properties are named members of classes, structures, and interfaces. Member variables or methods in a class or structures are called Fields. Properties are an extension of fields and are accessed using the same syntax. They use accessors through which the values of the private fields can be read, written or manipulated. Properties do not name the storage locations. Instead, they have accessors that read, write, or compute their values.

Properties are class members that can be accessed like data fields but contain code like a method.

Properties are often used to expose the data fields of a class in a more controlled manner. For example, a private field can be exposed by using a public property, but it is not necessary to use properties in this way.

For example, let us have a class named Student, with private fields for age, name and code. We cannot directly access these fields from outside the class scope, but we can have properties for accessing these private fields

A property has two accessors

Get: The get accessor is used to return the property value and the get { } implementation must include a return statement. It can access any member on the class.
Set: The set accessor is used to assign a new value to the property and the set { } implementation receives the implicit argument "value." This is the value to which the property is assigned.

A property is often defined as public and, by convention, always has a name that begins with a capital letter. In contrast, the convention for naming private data fields is to begin with a lower-case letter.

Illustration3: CREATE PROPERTIES

Get Ready: USE the project you saved in the previous exercise. Then, complete the following tasks:

1. Modify the code of class Rectangle as shown below. In this code, the constructor is removed and two properties are inserted:

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

public double Length                          // property to length field
get
{
```

```
return length;
}
Set
{
If (value>0.0)
length = value
}
}
public double Width                          // property to Width field

get
{
return length;
}
Set{
If (value>0.0)
width = value
}
}
public double GetArea()
{
return length * width;
}
}
```

2. Next, modify the code of the Program class to the following:

```
class Program
{
static void Main(string[] args)
{
Rectangle rect = new Rectangle();
rect.Length = 10.0;
rect.Width = 20.0;

double area = rect.GetArea();
Console.WriteLine(“Area of Rectangle: {0}”, area);
}
}
```

3. Select Debug > Start Without Debugging. A console window will pop up to display the area of the rectangle.
4. SAVE your project

PAUSE. Leave the project open to use in the next exercise.

In this exercise, you have modified the class Rectangle to introduce two properties, Length and Width.

Properties are often defined with a public access modifier. In the code for the property Length, the get accessor simply returns the value of the data field length. However, the set accessor checks for the value being assigned (using the value keyword) to the property and modifies the data field length only if the value is positive. The private fields length and width are also called backing fields for the properties that respectively expose them.

The class Rectangle also does not declare any explicit constructor. In this case, the users of the class (the Main method) need to use the default constructor and rely on properties to initialize the class data.

The Main method uses the properties Length and Width to set the data for the rect object. Trying to set either Length or Width to a negative value will be ignored, and in this case, the data fields will still have their original value of 0.

When defining properties, you can exclude either the get or the set accessor. If you don't include a set accessor, you don't provide a way to set the value of the property, and as a result, you have a read-only property. On the other hand, if you don't include the get accessor, you don't provide a way to get the value of the property, and as a result, you have a write-only property.

TakeNote

The usual programming pattern is that all the data fields of a class should be declared private, and that access to these private fields should be via public properties that check the data values for validity.

In C#, the characters // are used to add single- line comments to the code. The text following the // characters is ignored by the compiler. Multi-line comments start with the characters /* and end with the characters */.

UNDERSTANDING AUTO-IMPLEMENTED PROPERTIES

Auto-implemented properties simplify property declarations.

C# introduced auto-implemented properties beginning with version 3 to simplify property declaration when there is no additional logic specified in the get and set accessors. For example, without the validation checks, the Length and Width properties are defined like this:

```
private double length;
private double width;

public double Length                // property to length field
get
{
```

```
return length;
}
set
{
length = value
}
}
public double Width                    / / property to Width field
get
{
return length;
}
Set
{
width = value
}
}
```

In comparison, with C# auto-implemented properties, the simplified syntax for property declaration becomes:

public double Length { get; set; }
public double Width { get; set; }

In this case, the backing fields for the properties are defined behind the scenes and are not directly accessible by the code.

The auto-implemented properties used with default constructors can also simplify the creation and initialization of objects. For example, now an object can be created and initialized as follows:

```
static void Main(string[] args)
{
Rectangle rect = new Rectangle
{ Length = 10.0, Width = 20.0 };
Console.WriteLine("Area of Rectangle: {0}", rect.GetArea());
}
```

USING THE THIS KEYWORD

The *this* keyword can be used to access members from within constructors, instance methods, and accessors of instance properties and it also refers to the current instance of the class.

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. For example, earlier in this chapter, the Rectangle class was written as follows:

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

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

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

However, it could have been written like this:

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

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

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

As you can see, in the second example, the *this* keyword was used within the constructor and the GetArea method to refer to the data fields of the current object of the Rectangle class. Although it was not necessary to use the *this* keyword in this case, using it provides more flexibility in naming the method parameters. For example, you could define the constructor as follows:

```
public Rectangle(double length, double width)
{
// the parameter names length and width  shadow the class members length and
// width in this scope
this.length = length;
 this.width = width;
}
```

Two variables sometimes have the same identifier. A field can be named the same as a formal parameter. With the "**this**" keyword, we indicate the class member with that identifier .For example in the above code example within the scope of the definition of the Rectangle constructor, the names length and width will now refer to the parameter being passed. The names of the data fields have been shadowed by these block variables and can be only accessed by using the this keyword.

TakeNote

With "this" we eliminate naming conflicts. We indicate the current instance.

UNDERSTANDING DELEGATES

Delegates are special types that are used to encapsulate a method with a specific signature.

Delegates are special objects that can hold a reference to a method with a specific signature..A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

A delegate in C# is similar to a function pointer in C or C++. Using a delegate allows the programmer to encapsulate a reference to a method inside a delegate object. The delegate object can then be passed to code which can call the referenced method, without having to know at compile time which method will be invoked. Unlike function pointers in C or C++, delegates are object-oriented, type-safe, and secure.

Advantages

- Encapsulating the method's call from caller
- Effective use of delegate improves the performance of application
- Used to call a method asynchronously

Syntax for delegate declaration is:

delegate <return type> <delegate-name> <parameter list>

Instantiating Delegates

Once a delegate type has been declared, a delegate object must be created with the new keyword and be associated with a particular method. When creating a delegate, the argument passed to the newexpression is written like a method call, but without the arguments to the method.
For example:

```
public delegate void printString(String s);
...
printString ps1 = new printString(WriteToScreen);
printString ps2 = new printString(WriteToFile);
```

The preceding delegate can be used to reference any method that has a single string parameter and returns void type variable.

Another delegate example

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

The delegate definition specifies the signature of the method whose reference can be held by a delegate object. For example, in the above code, you define a RectangleHandler delegate that can hold references to a method that returns void and accepts a single parameter of the Rectangle type.

So, if you have a method with a similar signature as listed below, it is an ideal candidate for assignment to a delegate instance.

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

Then delegate type can be used to declare a variable that can refer to this method as it has the same signature as of the delegate. For example, you can say:

```
RectangleHandler handler;
```

And you can then assign the method to the delegate using the following syntax:

```
handler += new RectangleHandler(DisplayArea);
```

Alternatively, you can use the shortcut syntax shown below:

```
handler += DisplayArea;
```

Notice that the syntax uses the addition operation. This means that you can associate more than one method (of compatible signature), thereby creating an invocation list of one or more methods.

Finally, a call to a delegate can be made by a method-calling syntax, like this:

```
Rectangle rect = new Rectangle (10, 20);
handler(rect);
```

When the delegate is called in this way, it invokes all the methods in its invocation list. In this specific example, the handler object refers to only one method DisplayArea, and therefore, the DisplayArea method will be invoked with the rect object as a parameter.

Among many other applications, delegates form the basis for event declarations, as discussed in the next section.

TakeNote

You can use delegates without parameters or with parameter list

You should follow the same syntax as in the method
(If you are referring to the method with two int parameters and int return type, the delegate which you are declaring should be in the same format. This is why it is referred to as type safe function pointer.)

UNDERSTANDING 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.

Events are easy to understand in the context of a graphical user interface (GUI). For example, when a user clicks on a button, a Click event occurs. Multiple user interface elements can subscribe to this event and change their visual state accordingly (for example, some controls are enabled or disabled). In this type of event communication, the event publishers do not need to know which objects subscribe to the events that are being raised.

Events are not just limited to GUI programming. In fact, events play an important role in .NET Framework class libraries as a way for objects to signal any change in their state. You'll work with events in practically all programs.

When you define events, you generally need two pieces of information:

- A delegate that connects the event with its handler method(s)
- A class that contains the event data. This class is usually derived from the EventArgs class

To define an event, you can use a custom delegate. However, in most cases, if your event holds no event-specific data, using the predefined delegate EventHandler is sufficient. The EventHandler delegate is defined as follows:

public delegate void EventHandler(Object sender, EventArgs e);

Here, the sender parameter is a reference to the object that raises the event, and the e parameter is a reference to an event data object that contains no event data.

The EventArgs class is used by events that do not pass any event-related information to an event handler when an event is raised. If the event handler requires event-related information, the application must derive a class from the EventArgs class to hold the event-related data.

Illustration4 : PUBLISH AND SUBSCRIBE TO EVENTS

USE the project you saved in the previous exercise to carry out the following tasks:

1. Modify the code of class Rectangle as shown below:

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

2. Modify the code of the Program class to the following

```
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);
}
}
```

3. Select Debug > Start Without Debugging. A console window will pop up to display that the value of the Length property is changed.
4. SAVE your project.

PAUSE. Leave the project open to use in the next exercise

TakeNote

The EventArgs.Empty field represents an event with no event data. This field is equivalent to having a read- only instance of the EventArgs class

The code in the r_ Changed method uses a cast operator to convert an object data type to the Rectangle data type. Casting is explained later in this lesson, in the section entitled “Casting Between Types.”

In the example you just completed, the Rectangle class defines a Changed event that is invoked when the Length property of the Rectangle object is changed. The delegate of the Changed event is of EventHandler type. In the Rectangle class, the event Changed is invoked when the set accessor of the Length property is called.

You subscribe to the Changed event inside the Main method by attaching the r_Changed method as an event handler for the event by using the following code:

```
r.Changed += new EventHandler(r_Changed);
```

The signature of the r_Changed method matches the requirements of the EventHandler delegate. The r_Changed method is invoked as soon as you set the value of Length property in the Main method.

The above code uses the += operator rather than the simple assignment operator (=) to attach the event handler. By using the += operator, you make sure that this event handler will be added to a list of event handlers already attached with the event. This technique allows you have multiple event handlers that may respond to an event. If you use the assignment operator (=) to assign the new event handler, it will override any existing event handler that is attached to the event, and as a result, the newly attached event handler will be only one that is fired when the event is invoked.

UNDERSTANDING NAMESPACES

A namespace allows you to organize code and create unique class names.

A namespace is designed for providing a way to keep one set of names separate from another. The class names declared in one namespace will not conflict with the same class names declared in another.

A namespace is a language element that allows you to organize code and create globally unique class names. Let's say you create a class of the name Widget. Chances are that some other company will also ship code that contains a class of the name Widget. In that case, how do you handle the ambiguity in names? The solution is to organize the code within a namespace. A common convention is to use the company name in the namespace

Defining a Namespace

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

```
namespace namespace_name
{
  // code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:

```
namespace_name.item_name;
```

The following program demonstrates use of namespaces:

```
using System;
namespace first_space
{
  class namespace_cl
  {
    public void func()
    {
      Console.WriteLine("Inside first_space");
    }
  }
}
namespace second_space
{
  class namespace_cl
  {
    public void func()
    {
      Console.WriteLine("Inside second_space");
    }
  }
}
class TestClass
{
  static void Main(string[] args)
  {
    first_space.namespace_cl fc = new first_space.namespace_cl();
    second_space.namespace_cl sc = new second_space.namespace_cl();
    fc.func();
    sc.func();
    Console.ReadKey();
  }
}
```

When the above code is compiled and executed, it produces the following result:

```
Inside first_space
Inside second_space
```

The .NET Framework uses namespaces liberally to organize all its classes. For example, the System namespace groups all the fundamental classes. The System.Data namespace organizes classes for data access. Similarly, the System.Web namespace is used for Web-related classes.

Of course, with the use of namespaces, you might end up getting really long fully qualified class names that may result in verbose programs and a lot of typing. C# solves this inconvenience via the using directive. You can use the using directive at the top of the class file like this:

using System.Text;

Once you have included the using directive for a namespace, you don't need to fully qualify classes from that namespace in the file.

The *using* Keyword

The **using** keyword states that the program is using the names in the given namespace. For example, we are using the **System** namespace in our programs. The class Console is defined there. We just write:

```
Console.WriteLine ("Hello there");
```

We could have written the fully qualified name as:

```
System.Console.WriteLine("Hello there");
```

You can also avoid prepending of namespaces with the **using** namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code:

Let us rewrite our preceding example, with using directive:

```
using System;
using first_space;
using second_space;

namespace first_space
{
  class abc
  {
    public void func()
    {
      Console.WriteLine("Inside first_space");
    }
  }
}
namespace second_space
{
  class efg
  {
    public void func()
    {
      Console.WriteLine("Inside second_space");
    }
  }
}
class TestClass
{
  static void Main(string[] args)
  {
    abc fc = new abc();
    efg sc = new efg();
    fc.func();
    sc.func();
    Console.ReadKey();
  }
}
```

When the above code is compiled and executed, it produces the following result:

```
Inside first_space
Inside second_space
```

UNDERSTANDING STATIC MEMBERS

Whenever you write a function or declare a variable, it doesn't create instance in a memory until you create object of class. But if you declare any function or variable with static modifier, it directly create instance in a memory and acts globally. The static modifier doesn't reference with any object.

Members of a class are either static members or instance members. Generally speaking, it is useful to think of static members as belonging to classes and instance members as belonging to objects (instances of classes).

When a field, method, property, event, operator, or constructor declaration includes a static modifier, it declares a static member. In addition, a constant or type declaration implicitly declares a static member. The class members discussed so far in this section (e.g., data fields, methods, and properties) all operate on individual objects. Such members are called as instance members because they can be used only after an instance of a class is created. In contrast, the static keyword is used to declare members that do not belong to individual objects but to a class itself. Such class members are called as static members. One common example of a static member is the familiar Main method that serves as the entry point for your program

Illustration5: CREATE STATIC MEMBERS

USE the project you saved in the previous exercise. Then, perform the following steps:

1. Modify the code of class Rectangle as shown below:

```
class Rectangle
{
public static string ShapeName
{
get { return "Rectangle"; }
}
public double Length { get; set; }
public double Width { get; set; }

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

2. Modify the code of the Program class to the following:

```
class Program
{
static void Main(string[] args)
{
Rectangle rect = new Rectangle
{ Length = 10.0, Width = 20.0 };

Console.WriteLine(“Shape Name: {0}, Area: {1}”,
Rectangle.ShapeName,
 rect.GetArea());
}
}
```

3. Select Debug > Start Without Debugging. A console window will pop up to display the name and area of the shape.
4. SAVE your project.

PAUSE. Leave the project open to use in the next exercise

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 (such as Rectangle.ShapeName in the above exercise).

TakeNote

Note that it is not possible to use the this keyword reference with a static method or property because the this keyword can only be used to access instance objects

OBJECTIVE 2: Understanding Values and References

A value type directly stores a value, whereas a reference type only stores a reference to an actual value

A value type directly stores data within its memory. Reference types, on the other hand, store only a reference to a memory location; here, the actual data is stored at the memory location being referred to.

Most built-in elementary data types (such as bool, int, char, double, etc.) are value types. User-defined data types created by using the keyword struct are value types as well. Reference types include the types created by using the keywords object, string, interface, delegate, and class.

Understanding Structs

The keyword struct is used to create user-defined types that consist of small groups of related fields. Structs are value types—as opposed to classes, which are reference types.

Structs are defined by using the keyword struct, as shown below:

```
public struct Point
{
public double X, Y;
}
```

Structs can contain most of the elements that classes can contain, such as constructors, methods, properties, etc. However, as you'll learn in the next section, structs are value types, whereas classes are reference types. Unlike a class, a struct cannot inherit from another class or struct.

TakeNote

Structs are mostly used to create simple types. If you find yourself creating a very complex struct, you should consider using a class instead.

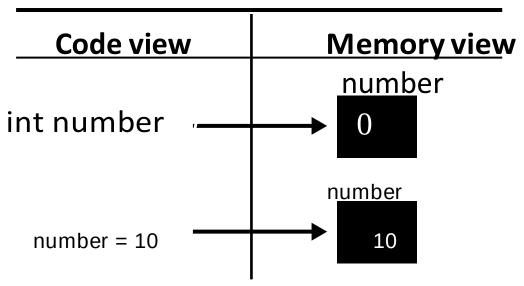
Understanding Memory Allocation

After you enter a value or text into a cell, you can modify it in a number of ways. In particular, you can remove the contents completely, enter a different value to replace what was there, or alter what you have entered

A good way to understand how value types differ from reference types is to visualize how each of them is represented in memory. Figure 2-1 shows how value types are created in memory. When you create a variable of type int, for instance, a named memory location is created that you can use to store a value of type int. Initially, when you don't explicitly assign a value, the default value of the data type (for int, the default value is 0) is stored in the memory location. Then, when an assignment is made, the memory address identified by the variable name is updated with the new value (10 in the case of the assignment in Figure 2-1).

Figure 2-1

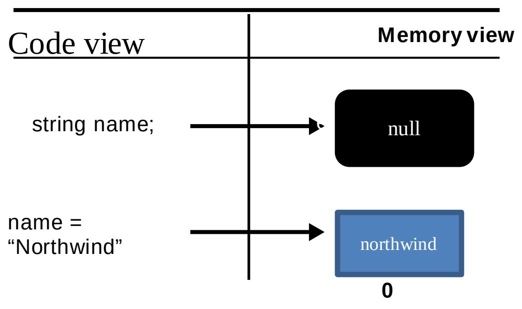
Visualizing a value type in memory



Now, take a look at Figure 2-2, which shows a reference type—specifically, the string data type. When you create a variable of type string, a memory location is created that will be identified by this name. However, this memory location will not contain the content of the string. Rather, this variable will store the memory address (a reference) of the location where the string is actually stored.

Figure 2-2

Visualizing a reference type in memory



Initially, when no value is assigned, the variable will have the value of null (a null reference; in other words, this this variable does not refer to a valid memory address). Then, in the next statement, when you say:

name = “Northwind”;

the string “Northwind” is created at a particular memory location (to keep things simple, let’s say the memory address is m100), and that memory address is stored in the variable name. Later, when it is time to retrieve the value of the variable name, the runtime will know that its contents are not stored in the variable itself, but rather at the memory location pointed to by the variable.

Illustration6: COPY VALUE AND REFERENCE TYPE

USE the project you saved in the previous exercise to complete the steps below

1. Add the following code after the Rectangle class definition to create a Point struct:

```
struct Point
{
public double X, Y;
}
```

2. Modify the code of the Main method as shown below:

```
static void Main(string[] args)
{
Point p1 = new Point();
p1.X = 10;
p1.Y = 20;
Point p2 = p1;
p2.X = 100;
Console.WriteLine(“p1.X = {0}”, p1.X);
Rectangle rect1 = new Rectangle
{ Length = 10.0, Width = 20.0 };
Rectangle rect2 = rect1;
rect2.Length = 100.0;
Console.WriteLine(“rect1.Length = {0}”, rect1.Length);
}
```

3. Select **Debug > Start Without Debugging**. A console window will pop up to display the values for p1.X and rect1.Length.
4. SAVE your project

PAUSE. Leave the project open to use in the next exercise

TakeNote

It is possible to create a struct without using the new operator. You can simply say
Point p1;
to create a variable of the struct type.

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

Here, the first part of the program creates a copy of the value type Point, and the second half creates a copy of the reference type Rectangle.

Let's start by analyzing how the copy of a value type is made. To begin, when the following statement is executed, a new variable p2 is created in memory, and its contents are copied from the variable p1:

```
Point p2 = p1;
```

After this statement is executed, the variable p2 is created, and the content of variable p1 is copied to variable p2. Both p1 and p2 have their own set of values available in their respective memory locations. So, when the following statement is executed:

```
p2.X = 100;
```
it only affects the value of X corresponding to the memory location of variable p2. The value of X for variable p1 remains unaffected.

Now, let's analyze how the copy works between reference types. In this case, when the following statement is executed, a new variable rect2 is created, and just as before, the contents of rect1 are copied into the memory location of rect2:

```
Rectangle rect2 = rect1;
```

However, because the class Rectangle is a reference type, the content of variable rect1 is actually a reference to a memory location that holds a Rectangle object. So, after the above initialization, both rect1 and rect2 point to the same memory location and in turn the same Rectangle object. In other words, there is only one rectangle object in memory, and both rect1 and rect2 are referring to it. The next statement modifies the Length of that rectangle object:

```
rect2.Length = 100.0;
```

This statement references the memory location pointed to by rect2 (which happens to be the same memory location pointed to by rect1) and modifies the Length of the Rectangle object. Now, if you attempt to reference the same memory location via the rect1 object, you get the modified object and the following code displays the value "rect1.Length = 100":

```
Console.WriteLine("rect1.Length={0}",
    rect1.Length);
```

TakeNote

Objects are always allocated memory on the heap. The heap is the memory available to a program at runtime for dynamic memory allocation. In contrast, some data items can be created on the execution stack or the call stack. Items created on the stack are the method parameters and the local variables declared within a method. The stack memory is reclaimed when the stack unwinds (when a method returns, for example). The memory allocated in the heap is automatically reclaimed by the garbage collector when the objects are not in use any more (i.e., no other objects are holding a reference to them)

OBJECTIVE 3: Understanding Encapsulation

Encapsulation is an information-hiding mechanism that makes code easy to maintain and understand
Encapsulation is defined 'as the process of enclosing one or more items within a physical or logical package'.

Encapsulation, in object oriented programming methodology, prevents access to implementation details.
Encapsulation is a mechanism to restrict access to a class or class members in order to hide design decisions that are likely to change. Encapsulation gives class designers the flexibility to change a section of code when needed without changing all the other code that makes use of that code. Also, when you hide information, you hide the complexity associated with it. As a result, with the help of encapsulation, you can write code that is easier to understand and maintain.

In the previous exercises, you saw encapsulation at work when you declared the data members as private and enforced data-field initialization via a constructor. Because the data members are hidden from the users of the class, the developer of the Rectangle class can change the names of the data fields without requiring any changes in the calling code.

Properties offer a great way to encapsulate data fields along with any accompanying logic. Also, access modifiers such as private and public allow you to control the level of access for a class member or for the class itself.

Understanding Access Modifiers

Access modifiers control where a type or type member can be used.

All types and type members have an access level that specifies where that class or its members can be used in your code. The access level can be set using one of the access modifiers specified in following table

Table 2-1

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. (You'll learn more about derived classes later in this lesson, in the section entitled "Understanding Inheritance.")
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.

Access modifiers are not allowed in namespace declarations; public access is implied for namespaces. The top-level classes (declared directly under a namespace) can be only public or internal. The internal access modifier is the default for a class if no access modifier is specified. (For instance, the class Rectangle defined in the previous exercise defaults to having an internal access.) The accessibility of a nested class may not be less restrictive than the accessibility of the containing class.

TakeNote

When C# code is compiled, the output executable code contained within a .dll or an .exe file is also called as an *assembly*. An assembly is a unit of executable code that can be independently versioned and installed.
You should use the most restrictive access level that makes sense for a type member

OBJECTIVE4: Understanding Inheritance

Inheritance is a feature of object-oriented programming that allows you to develop a class once, and then reuse that code over and over as the basis of new classes. Inheritance enables you to create new classes that reuse, extend, and modify the functionality defined in existing classes. The class that inherits the functionality is called a derived class, and the class whose functionality is inherited is called a base class. A derived class inherits all the functionality of the base class and can also define additional features that make it different from the base class

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.

The idea of inheritance implements the IS-A relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

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 class or interface.

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

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

Say that we want to create a set of classes that describes polygons such as rectangles or triangles. These classes will have some common properties, such as width and length. For this case, you can create a base class Polygon with the Width and Length properties, and the derived classes Rectangle and Triangle will inherit these properties while providing their own functionality. The following exercise explains this concept in more detail.

TakeNote

Unlike classes, the structs do not support inheritance.

Illustration7: CREATE DERIVED CLASSES

USE the project you saved in the previous exercise to perform the following actions:

1. Add a new class named Polygon as shown below:

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

    public double Width { get; protected set; }
}
```

2. Modify the Rectangle class as shown below:

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

3. Now, modify the code of the Main method as shown below:

```
static  void  Main(string[]  args)
{
Rectangle rect = new Rectangle(10, 20);
 Console.WriteLine("Width={0}, Length={1}, Area = {2}",
rect.Width, rect.Length, rect.GetArea());
}
```

4. Select Debug > Start Without Debugging. A console window will pop up to display the width, length, and the area of the rectangle.
5. SAVE your project.

PAUSE. Leave the project open to use in the next exercise

To define a derived class, you put a colon after the derived class name, followed by the name of the base class. Here, the Polygon class is the base class for the Rectangle class.

The properties Length and Width in the Polygon class are declared with a protected access modifier for the set accessor. This means that access to the set accessor is available only inside the Polygon class and its derived classes. You can still get the value of the Length and Width properties in the Main method, but you'll get an error if you attempt to assign a value to these properties.

The Rectangle class inherits all the non-private data and behavior of the Polygon class. In addition, the Rectangle class defines additional functionality (GetArea method) that is not available in the base class.

Understanding Abstract and Sealed Classes

The abstract classes provide a common definition of a base class that can be shared by multiple derived classes. The sealed classes, on the other hand, provide complete functionality but cannot be used as base classes

Abstract classes, marked by the keyword abstract in the class definition, are typically used to define a base class in the hierarchy. What's special about them, is that you can't create an instance of them - if you try, you will get a compile error.

In the previous exercise, you defined a GetArea method on the Rectangle class. Suppose you want to create another class, Triangle, that is of the Polygon type. Here, you'll need a GetArea method in the Triangle class that will calculate a triangle's area.

Often, base classes act as the repository of common functionality. In the case of Polygon, the polygon itself won't know how to calculate the area without knowledge of the shape type. But in general, we can expect all classes of the Polygon type to be able to calculate their area. Such expectations can be rolled over to the base class with the help of an abstract keyword

Illustration8: **CREATE ABSTRACT CLASSES**

USE the project you saved in the previous exercise, and perform the following steps.

1. Modify the Polygon class as shown below:

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

2. Modify the Rectangle class as shown below:

```
class Rectangle: Polygon
{
public Rectangle(double length, double width)
{
Length = length;
 Width = width;
}
public override double GetArea()
{
return Width * Length;
}
}
```

3. Note that no modification to the Main method is needed.
4. Select Debug > Start Without Debugging. A console window will pop up to display the width, length, and the area of the rectangle.
5. SAVE your project

PAUSE. Leave the project open to use in the next exercise.

This version of the Polygon class defines a method named GetArea. The main reason for including this method in the base class is that now the base class can provide a common template of functionality for the derived classes. But, as we discussed, the Polygon base class doesn't know enough to calculate the area of the shape. This situation can be handled by marking the method as abstract. An abstract method provides a definition but does not offer any implementation (the method body). If any of the members of a class are abstract, the class itself needs to be marked as abstract. An abstract class cannot be instantiated.

Derived classes can provide an implementation of an abstract class to create a concrete class (a non-abstract class). The derived classes can offer an implementation of an abstract method by overriding it in a derived class. For example, the Rectangle class overrides the abstract GetArea method of the base class and provides a full implementation. As a result, the Rectangle class is no longer an abstract class and can be instantiated directly.

TakeNote

You cannot create instances of an abstract class.

Sealed classes, on the other hand, are defined when your implementation is complete and you do not want a class to be inherited. A sealed class can be created by using the keyword sealed, as in the following example:

```
sealed class Rectangle: Polygon
{
//class members
//
}
```

Because Rectangle is a sealed class, it cannot be a used as a base class. It is also possible to mark selected class members as sealed to avoid them being overridden in a derived class. For example, you could say:

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

This declaration ensures that the method GetArea cannot be overridden in a derived class.

TakeNote

C# does not support inheriting from more than one base class, often referred to as multiple inheritance

Inheriting from the Object Class

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. For example, when you declared the following class earlier in this lesson:

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

it was functionally equivalent to the following declaration:

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

However, you are not required to declare the Polygon class in the latter way because inheritance from the Object class is implicitly assumed. As part of this inheritance, a derived class can override the methods of the Object class. Two of the most common methods for doing this are as follows:

- Equals: Supports comparison between two objects, and returns true if the two objects have the same value.
- ToString: Returns a string representation of the class. By default, it returns the full name of the class. It is often useful to override this method so that it returns a string representation of the current state of the object.

The following example shows how you can override the ToString method in the Rectangle class:

```
class  Rectangle:  Polygon
{
public Rectangle(double length, double width)
{
Length = length; Width = width;
}
public override double GetArea()
{
return Width * Length;
}
public override string ToString()
{
return String.Format(
“Width = {0}, Length = {1}”, Width, Length);
}
}
```

Casting between Types

In C#, the runtime allows you to cast an object to any of its base types.

Derived classes have an “is-a” relationship with their base class. For example, we can say that the Rectangle is-a Polygon. Thus, an object of the Rectangle class has effectively two data types in this case: the object is a Rectangle, and the object is also a Polygon.

In C#, the runtime allows you to cast an object to its class or to any of its base classes. For example, you can say:

Polygon p = new Rectangle(10, 20);

Here, a new Rectangle object is created and is cast to its base type Polygon. C# doesn’t require any special syntax to do this, because cast to base type is considered a safe conversion.

Casting is also possible the other way round. For example, you can say:

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

Here, a Rectangle object is first assigned to an Object (the ultimate base class), and the resultant object is then cast back as a Rectangle. When the latter assignment happens, an explicit cast is required because you are converting a more general object to a less general object. The runtime checks whether the value of the variable o is compatible with the Rectangle class.

If, at execution time, the value of o is not compatible with the Rectangle class, the runtime throws a System.InvalidCastException

USING THE IS OPERATOR

To avoid runtime errors such as the InvalidCastException, the IS operator can be used to check whether the cast is allowed before actually performing the cast, as in this example:

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

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

USING THE AS OPERATOR

Another useful cast operator is 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. For example, consider the following code:

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

If, at runtime, 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.

TakeNote

If you are using the as operator to convert a type, the is operator check is not necessary. You can simply check the return value from as against null.

OBJECTIVE5: UNDERSTANDING POLYMORPHISM

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

You are developing an application that allows users to work with different kind of polygons. You have a collection that contains several types of polygons, such as a rectangle, a triangle, and a square. Each polygon provides you with its own implementation of the Draw method. When you work with this collection, you don’t necessarily know exactly which shape you are working with, but you would like the correct Draw method to be invoked each time. Polymorphism enables you to do exactly this.

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 appro- priate method is invoked from the derived class.

Illustration9: USE POLYMORPHISM

USE the project you saved in the previous exercise to carry out the following steps:

1. Modify the Polygon class as shown below:

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

2. Modify the Rectangle class as shown below:

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

3. Add a new class called Triangle, as shown below:

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

4. Modify the Main method as follows:

```
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();
}
}
```

5. Select Debug > Start Without Debugging. A console window will pop up to display the drawing message for each polygon.
6. SAVE your project.

PAUSE. Leave the project open to use in the next exercise.

In this exercise, the definitions of the Polygon and the Rectangle class are simplified to emphasize the concept of polymorphism. The base class provides a single Draw method. The important thing to note here is the keyword virtual. This keyword allows the derived classes to override the method.

Both the Rectangle and Triangle classes override the base class Draw method with their own definition by using the override keyword. When executed, the Main method generates the following output:

```
Drawing: Polygon
Drawing: Rectangle
Drawing: Triangle
```

The List<Polygon> data type is capable of storing a collection of objects that are of type Polygon or types that derive from Polygon. The foreach loop is iterating over a collection of Polygon objects. The underlying type of the first object is Polygon, but the second and third objects in the collection are actually Rectangle and Triangle objects that just happen to be cast as Polygons. The runtime will look at the actual underlying type and invoke the overridden method from the derived class. That’s the reason why the derived class version of the Draw method is called for both Rectangle and Triangle objects.

Understanding 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.

When a base class defines a virtual member, the derived class has two options for handling it—specifically, the derived class can use either the override keyword or the new keyword. The override keyword takes priority over the base-class definition of the member. Here, the object of the derived class will call the overridden member instead of the base-class member.

In comparison, if the new keyword is used, a new definition of the member is created and the base-class member is hidden. However, if the derived class is cast to an instance of the base class, the hidden members of the class can still be called.

To better understand these concepts, modify the Triangle method from the previous exercise to the following:

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

Then, modify the code in the Main class to the following:

```
Triangle t = new Triangle();
 t.Draw();
Polygon p = t;
 p.Draw();
```

The program will produce the following output:

```
Drawing: Triangle
Drawing: Polygon
```

Here, when the Draw method is directly invoked on the object of the derived class, the new version of the method is used. However, if the method is executed when the derived class is cast as a base class, the hidden base-class version of the Draw method is executed.

TakeNote

If the method in the derived class is not preceded by the new keyword or the override keyword, the compiler will issue a warning, and the method will behave as if the new keyword were present

The System.Object class provides a ToString method. By convention, you should use this method to return the human-readable representation for a class. When you create your types, it is good practice to override this method to return readable information about the objects.

OBJECTIVE6: UNDERSTANDING INTERFACES

Interfaces are used to establish contracts through which objects can interact with each other without knowing the implementation details.

Interfaces are defined by using the interface keyword. An interface definition consists of a set of signatures for methods, properties, delegates, events, or indexers. 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);
}
```

TakeNote

By convention, all interfaces defined in the .NET Framework begin with a capital I. Although you are free to name your interfaces as you wish, it is best to follow the Framework convention

The IComparable interface has a single method (CompareTo) that accepts an object and returns an int. The return value of this method indicates the result of comparing the given parameter with the current object. According to the documentation of the CompareTo method:

- If the instance is equal to the parameter, CompareTo returns 0.
- If the parameter value is less than the instance or if the parameter is null, then a positive value is returned.
- If the parameter value is greater than the instance, then a negative value is returned.
- If the parameter is not of the compatible type, then an ArgumentException is thrown.

How does IComparable decide how to compare two Rectangle objects or two Employee objects? It doesn't. The classes that are interested in such comparisons must implement the IComparable interface by providing a method body for the CompareTo method. Each class that implements IComparable is free to provide its own custom comparison logic inside the CompareTo method.

Illustration10: USE THE ICOMPARABLE INTERFACE

USE the project you saved in the previous exercise to carry out the following steps:

1. Modify the Rectangle class as shown below:

```
class Rectangle: Polygon, IComparable
{
public double Length { get; set; }
 public double Width { get; set; }
public override void Draw()
{
Console.WriteLine("Drawing: Rectangle");
}
public double GetArea()
{
return Length * Width;
}
public int CompareTo(object obj)
{
if (obj == null)
return 1;

if (!(obj is Rectangle))
    throw new ArgumentException();

Rectangle target = (Rectangle)obj;
double diff = this.GetArea() - target.GetArea();

if (diff == 0)
return 0;
 else if (diff > 0)
return 1;

else
 return -1;
}
}
```

2. Then, modify the Main method as shown below:

```
static void Main(string[] args)
{
Rectangle rect1 = new Rectangle
{ Length = 10, Width = 20 };
Rectangle rect2 = new Rectangle
{ Length = 100, Width = 200 };
 Console.WriteLine(rect1.CompareTo(rect2));
}
```

3. Select **Debug > Start Without Debugging**. A console window will pop up and display the value –1 because the area of rect1 is less than the area of rect2.
4. SAVE your project

Here, the class Rectangle both derives from the Polygon class and implements the IComparable interface. A class that implements an interface must implement all the methods declared in that interface.

An interface is similar to an abstract class, but there are some noticeable differences. For one, an abstract class provides incomplete implementation, whereas an interface provides no implementation at all. A class can also implement multiple interfaces but is limited to inheriting from only a single base class.

So, how do you decide whether to use an abstract class or an interface? One way is to check whether an “is-a” relationship exists between the two concepts. For example, if an inheritance relationship exists between a SalariedEmployee and an Employee, then you can use an abstract class to standardize common functionality among derived classes. On the other hand, there is no “is-a” relationship between an Employee and the IComparable. Therefore, the comparison functionality is best implemented as an interface