msdn training

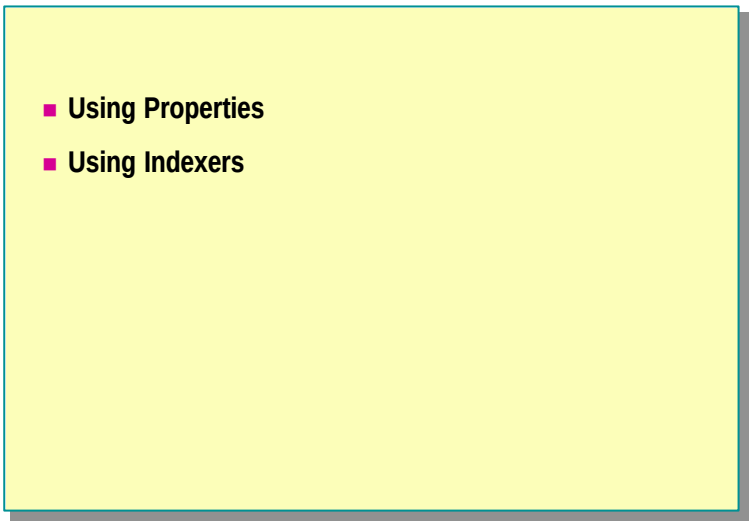
Module 13: Properties and Indexers

Contents

This course is based on the prerelease Beta 1 version of Microsoft® Visual Studio .NET. Content in the final release of the course may be different from the content included in this prerelease version. All labs in the course are to be completed with the Beta 1 version of Visual Studio .NET.

Microsoft

Overview



You can expose the named attributes for a class by using either fields or properties. Fields are implemented as member variables with public access. In C#, properties appear to be fields to the user of a class, but they use methods to get and set values.

C# provides an indexer feature that allows you to index the members of an object (with an instance indexer) or a class (with a static indexer) as if they were an array. As with properties, you use **get** or **set** methods to perform indexing operations.

In this module, you will learn how to use properties and indexers. You will learn how to use properties to enable field-like access and indexers to enable array-like access.

After completing this module, you will be able to:

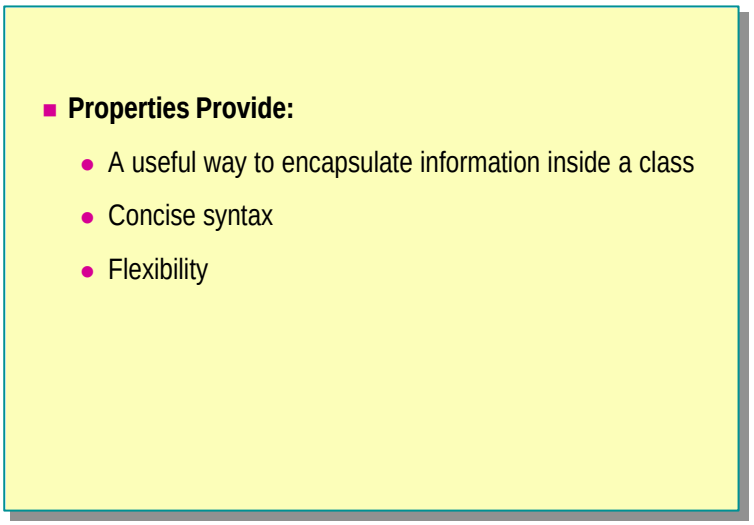
- Create properties to encapsulate data within a class.
- Define indexers to gain access to classes by using array-like notation.

Using Properties

- Why Use Properties?
- Using Accessors
- Comparing Properties to Fields
- Comparing Properties to Methods
- Property Types
- Property Example

In this section, you will learn how to use properties to encapsulate and access data in a class.

Why Use Properties?



Properties provide a useful way to encapsulate data within a class. Examples of properties include the length of a string, the size of a font, the caption of a window, the name of a customer, and so on.

Concise Syntax

C# adds properties as first-class elements of the language. Many existing languages, such as Microsoft® Visual Basic®, already have properties as first-class elements of the language. If you think of a property as a field, it can help you to focus on the application logic. Compare, for example, the following two statements. The first statement does not use properties, whereas and the second does use properties.

```
o.SetValue(o.GetValue( ) + 1);

o.Value++;
```

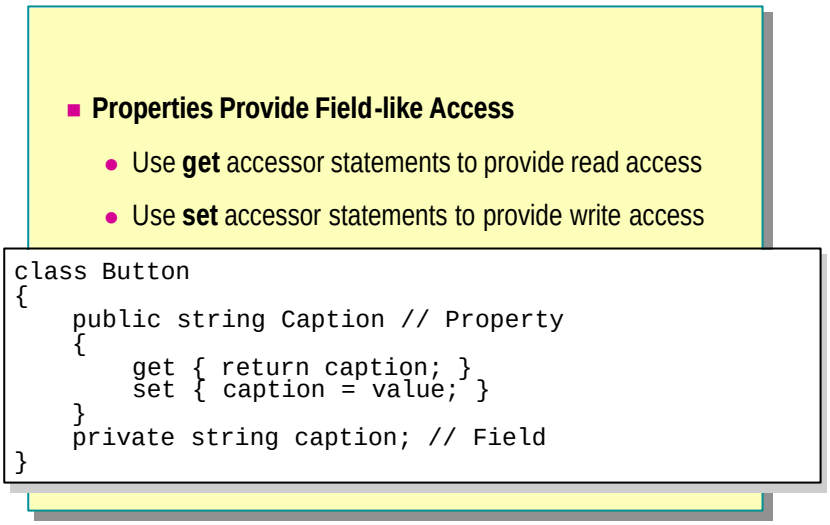
The statement that uses a property is certainly easier to understand and much less error prone.

Flexibility

To read or write the value of a property, you use field-like syntax. (In particular, you do not use parentheses.) However, the compiler translates this field-like syntax into encapsulated method-like **get** and **set** accessors. For example, **Value** could be a property of the object **o** in the expression **o.Value**, which will cause the statements inside the **get** accessor "method" for the **Value** property to be executed. This separation allows the statements inside the **get** and **set** accessors of a property to be modified without affecting the use of the property, which retains its field-like syntax. Because of this flexibility, you should use properties instead of fields whenever possible.

When you expose state through a property, your code is potentially less efficient than when you expose state directly through a field. However, when a property contains only a small amount of code and is non-virtual (which is frequently the case), the execution environment can replace calls to an accessor with the actual code of the accessor. This process is known as *inlining*, and it makes property access as efficient as field access, yet it preserves the increased flexibility of properties.

Using Accessors



A property is a class member that provides access to a field of an object. You use a property to associate actions with the reading and writing of an object's attribute. A property declaration consists of a type and a name and has either one or two pieces of code referred to as accessors. These accessors are as follows:

- **get** accessor
- **set** accessor

Accessors have no parameters. A property does not need to have both a **get** accessor and a **set** accessor. For example, a read-only property will provide only a **get** accessor. You will learn more about read-only properties later in this section.

Using the get Accessor

The **get** accessor of a property returns the value of the property. The following code provides an example:

```
public string Caption
{
    get { return caption; }
    ...
}
```

You implicitly call a property's **get** accessor when you use that property in a read context. The following code provides an example:

```
Button myButton;
...
string cap = myButton.Caption; // Calls "Caption.get"
```

Notice that you do not use parentheses after the property name. In this example, the statement `return caption;` returns a string. This string is returned whenever the value of the **Caption** property is read.

Reading a property should not change the object's data. When you invoke a **get** accessor, it is conceptually equivalent to reading the value of a field. A **get** accessor should not have observable side effects.

Using the set Accessor

The **set** accessor of a property modifies the value of a property.

```
public string Caption
{
    ...
    set { caption = value; }
}
```

You implicitly call a property's **set** accessor when you use that property in a write context—that is, when you use it in an assignment. The following code provides an example:

```
Button myButton;
...
myButton.Caption = "OK"; // Calls "Caption.set"
```

Notice again that you do not use parentheses. The variable *value* contains the value that you are assigning and is created automatically by the compiler. Inside the **set** accessor for the **Caption** property, *value* can be thought of as a string variable that contains the string "OK." A **set** accessor cannot return a value.

Invoking a **set** accessor is syntactically identical to a simple assignment, so you should limit its observable side effects. For example, it would be somewhat unexpected for the following statement to change both the speed and the color of the **thing** object.

```
thing.speed = 5;
```

However, sometimes **set** accessor side effects can be useful. For example, a shopping basket object could update its total whenever the item count in the basket is changed.

Comparing Properties to Fields

- **Properties Are "Logical Fields"**
  - The **get** accessor can return a computed value
- **Similarities**
  - Syntax for creation and use is the same
- **Differences**
  - Properties are not values; they have no address
  - Properties cannot be used as **ref** or **out** parameters to methods

As an experienced developer, you already know how to use fields. Because of the similarities between fields and properties, it is useful to compare these two programming elements.

Properties Are Logical Fields

You can use the **get** accessor of a property to calculate a value rather than return the value of a field directly. Think of properties as logical fields—that is, fields that do not necessarily have a direct physical implementation. For example, a **Person** class might contain a field for the person's date of birth and a property for the person's age that calculates the person's age:

```
class Person
{
    public Person(DateTime born)
    {
        this.born = born;
    }

    public int Age
    {
        // Simplified...
        get { return DateTime.Now.Year – born.Year; }
    }
    ...
    private readonly DateTime born;
}
```

Similarities with Fields

Properties are a natural extension of fields. Like fields, they:

- Specify a name with an associated non-void type, as shown:

```
class Example
{
    int field;
    int Property { ... }
}
```

- Can be declared with any access modifier, as shown:

```
class Example
{
    private int field;
    public int Property { ... }
}
```

- Can be static, as shown:

```
class Example
{
    static private int field;
    static public int Property { ... }
}
```

- Can hide base class members of the same name, as shown:

```
class Base
{
    public int field;
    public int Property { ... }
}
class Example: Base
{
    new public int field;
    new public int Property { ... }
}
```

- Are assigned to or read from by means of field syntax, as shown:

```
Example o = new Example( );
o.field = 42;
o.Property = 42;
```

Differences from Fields

Unlike fields, properties do not correspond directly to storage locations. Even though you use the same syntax to access a property that you would use to access a field, a property is not classified as a variable. So you cannot pass a property as a **ref** or **out** parameter without getting compile-time errors. The following code provides an example:

```
class Example
{
    public string Property
    {
        get { ... }
        set { ... }
    }
    public string Field;
}
class Test
{
    static void Main( )
    {
        Example eg = new Example( );

        ByRef(ref eg.Property); // Compile-time error
        ByOut(out eg.Property); // Compile-time error

        ByRef(ref eg.Field); // Okay
        ByOut(out eg.Field); // Okay
    }
    static void ByRef(ref string name) { ... }
    static void ByOut(out string name) { ... }
}
```

Comparing Properties to Methods

- **Similarities**
  - Both contain code to be executed
  - Both can be used to hide implementation details
  - Both can be virtual, abstract, or override
- **Differences**
  - Syntactic - properties do not use parentheses
  - Semantic - properties cannot be **void** or take arbitrary parameters

Similarities with Methods

With both properties and methods, you can:

- Specify statements to be executed.
- Specify a return type that must be at least as accessible as the property itself.
- Mark them as virtual, abstract, or override.
- Introduce them in an interface.
- Provide a separation between an object's internal state and its public interface (which you cannot do with a field).

This last point is perhaps the most important. You can change the implementation of a property without affecting the syntax of how you use the property. For example, in the following code, notice that the **TopLeft** property of the **Label** class is implemented directly with a **Point** field.

```
struct Point
{
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public int x, y;
}
class Label
{
    ...
    public Point TopLeft
    {
        get { return topLeft; }
        set { topLeft = value; }
    }
    private Point topLeft;
}
class Use
{
    static void Main( )
    {
        Label text = new Label(...);
        Point oldPosition = text.TopLeft;
        Point newPosition = new Point(10,10);
        text.TopLeft = newPosition;
    }
    ...
}
```

Because **TopLeft** is implemented as a property, you can also implement it without changing the syntax of how you use the property, as shown in this example, which uses two **int** fields named *x* and *y* instead of the **Point** field named *topLeft*.

```
class Label
{
    public Point TopLeft
    {
        get { return new Point(x,y); }
        set { x = value.x; y = value.y; }
    }
    private int x, y;
}
class Use
{
    static void Main( )
    {
        Label text = new Label(...);
        // Exactly the same
        Point oldPosition = text.TopLeft;
        Point newPosition = new Point(10,10);
        text.TopLeft = newPosition;
        ...
    }
}
```

Differences from Methods

Properties and methods differ in a few important ways, as summarized in the following table.

Feature	Properties	Methods
Use parentheses	No	Yes
Specify arbitrary parameters	No	Yes
Use void type	No	Yes

Consider the following examples:

- Properties do not use parentheses, although methods do.

```
class Example
{
    public int Property { ... }
    public int Method( ) { ... }
}
```

- Properties cannot specify arbitrary parameters, although methods can.

```
class Example
{
    public int Property { ... }
    public int Method(double d1, decimal d2) { ... }
}
```

- Properties cannot be of type **void**, although methods can.

```
class Example
{
    public void Property { ... }  // Compile-time error
    public void Method( ) { ... } // Okay
}
```

Property Types

- **Read/Write Properties**
  - Have both **get** and **set** accessors
- **Read-Only Properties**
  - Have **get** accessor only
  - Are not constants
- **Write-Only Properties - Very Limited Use**
  - Have **set** accessor only
- **Static Properties**
  - Apply to the class and can access only static data

When using properties, you can define which operations are allowed for each property. The operations are defined as follows:

- Read/write properties

  When you implement both **get** and **set**, you have both read and write access to the property.
- Read-only properties

  When you implement only **get**, you have read-only access to the property.
- Write-only properties

  When you implement only **set**, you have write-only access to the property.

Using Read-Only Properties

Properties that only have a **get** accessor are called read-only properties. In the example below, the **BankAccount** class has a **Balance** property with a **get** accessor but no **set** accessor. Therefore, **Balance** is a read-only property.

```
class BankAccount
{
    private decimal balance;
    public decimal Balance
    {
        get { return balance; } // But no set
    }
}
```

You cannot assign a value to a read-only property. For example, if you add the statements below to the previous example, you will get a compile-time error.

```
BankAccount acc = new BankAccount( );
acc.Balance = 1000000M;
```

A common mistake is to think that a read-only property specifies a constant value. This is not the case. In the following example, the **Balance** property is read-only, meaning you can only read the value of the balance. However, the value of the balance can change over time. For example, the balance will increase when a deposit is made.

```
class BankAccount
{
    private decimal balance;
    public decimal Balance
    {
        get { return balance; }
    }
    public void Deposit(decimal amount)
    {
        balance += amount;
    }
    ...
}
```

Using Write-Only Properties

Properties that only have a **set** accessor are called write-only properties. In general, you should avoid using write-only properties.

If a property does not have a **get** accessor, you cannot read its value; you can only assign a value to it. If you attempt to read from a property that does not have a **get** accessor, you will get a compile-time error.

Static Properties

A static property, like a static field and a static method, is associated with the class and not with an object. Because a static property is not associated with a specific instance, it can access only static data and cannot refer to **this** or instance data. Following is an example:

```
class MyClass
{
    private int MyData = 0;

    public static int ClassData
    {
        get {
            return this.MyData; // Error
        }
    }
}
```

You cannot include a **virtual**, **abstract**, or **override** modifier on a static property.

Property Example

```
public class Console
{
    public static TextReader In
    {
        get {
            if (reader == null) {
                reader = new StreamReader(...);
            }
            return reader;
        }
    }
    ...
    private static TextReader reader = null;
}
```

Just-in-Time Creation

You can use properties to delay the initialization of a resource until the moment it is first referenced. This technique is referred to as *lazy creation*, *lazy instantiation*, or *just-in-time creation*. The following code shows an example from the Microsoft .NET SDK Framework of just-in-time creation (simplified and not thread-safe):

```
public class Console
{
    public static TextReader In
    {
        get {
            if (reader == null) {
                reader = new StreamReader(...);
            }
            return reader;
        }
    }
    ...
    private static TextReader reader = null;
}
```

In the code, notice that:

- The underlying field called *reader* is initialized to **null**.
- Only the first read access will execute the body of the **if** statement inside the **get** accessor, thus creating the **new StreamReader** object. (**StreamReader** is derived from **TextReader**.)

Using Indexers

- What Is an Indexer?
- Comparing Indexers to Arrays
- Comparing Indexers to Properties
- Using Parameters to Define Indexers
- String Example
- BitArray Example

An *indexer* is a member that enables an object to be indexed in the same way as an array. Whereas you can use properties to enable field-like access to the data in your class, you can use indexers to enable array-like access to the members of your class.

In this section, you will learn how to define and use indexers.

What Is an Indexer?

- **An Indexer Provides Array-like Access to an Object**
  - Useful if a property can have multiple values
- **To Define an Indexer**
  - Create a property called *this*
  - Specify the index type
- **To Use an Indexer**
  - Use array notation to read or write the indexed property

An object is composed of a number of subitems. (For example, a list box is composed of a number of strings.) Indexers allow you to access the subitems by using array-like notation.

Defining Indexers

The following code shows how to implement an indexer that provides access to an internal array of strings called **list**:

```
class StringList
{
    public string[ ] list;
    public string this[int index]
    {
        get { return list[index]; }
        set { list[index] = value; }
    }
    ...
    // Other code and constructors to initialize list
}
```

The indexer is a property called **this** and is denoted by square brackets containing the type of index it uses. (Indexers must always be called **this**; they never have names of their own. They are accessed by means of the object they belong to.) In this case, the indexer requires that an **int** be supplied to identify the value to be returned or modified by the accessors.

Using Indexers

You can use the indexer of the **StringList** class to gain both read and write access to the members of **myList**, as shown in the following code:

```
...
StringList myList = new StringList( );
...
myList[3] = "Hello";              // Indexer write
...
string myString = myList[8];      // Indexer read
...
```

Notice that the syntax for reading or writing the indexer is very similar to the syntax for using an array. Referencing **myList** with an **int** in square brackets causes the indexer to be used. Either the **get** accessor or the **set** accessor will be invoked, depending upon whether you are reading or writing the indexer.

Comparing Indexers to Arrays

- **Similarities**
  - Both use array notation
- **Differences**
  - Indexers can use non-integer subscripts
  - Indexers can be overloaded- you can define several indexers, each using a different index type
  - Indexers are not variables, so they do not denote storage locations- you cannot pass an indexer as a **ref** or an **out** parameter

Although indexers use array notation, there are some important differences between indexers and arrays.

Defining Index Types

The type of the index used to access an array must be integer. You can define indexers to accept other types of indexes. For example, the following code shows how to use a string indexer:

```
class NickNames
{
    public Hashtable names = new Hashtable( );
    public string this[string realName]
    {
        get { return names[realName]; }
        set { names[realName] = value; }
    }
    ...
}
```

In the following example, the **NickNames** class stores real name and nickname pairs. You can store a nickname and associate it with a real name, and then later request the nickname for a given real name.

```
...
NickNames myNames = new NickNames( );
...
myNames["John"] = "Cuddles";
...
string myNickName = myNames["John"];
...
```

Overloading

A class can have multiple indexers, if they use different index types. You could extend the **NickNames** class to create an indexer that takes an integer index. The indexer could iterate through the Hashtable the specified number of times and return the value found there. Following is an example:

```
class NickNames
{
    public Hashtable names = new Hashtable( );
    public string this[string realName]
    {
        get { return names[realName]; }
        set { names[realName] = value; }
    }

    public string this[int nameNumber]
    {
        get
        {
          string nameFound;
          // Code that iterates through the Hashtable
          // and populates nameFound
          return nameFound;
        }
    }
    ...

}
```

Indexers Are Not Variables

Unlike arrays, indexers do not correspond directly to storage locations. Instead, indexers have **get** and **set** accessors that specify the statements to execute in order to read or write their values. This means that even though you use the same syntax for accessing an indexer that you use to access an array (you use square brackets in both cases), an indexer is not classified as a variable.

If you pass an indexer as a **ref** or **out** parameter, you will get compile-time errors, as the following example shows:

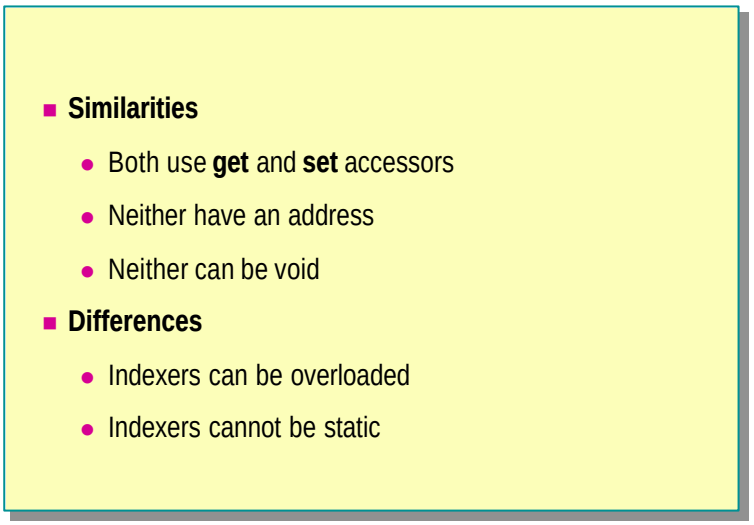
```
class Example
{
    public string[ ] array;
    public string this[int index]
    {
        get { ... }
        set { ... }
    }
}


class Test
{
    static void Main( )
    {
        Example eg = new Example( );

        ByRef(ref eg[0]); // Compile-time error
        ByOut(out eg[0]); // Compile-time error

        ByRef(ref eg.array[0]); // Okay
        ByOut(out eg.array[0]); // Okay
    }
    static void ByRef(ref string name) { ... }
    static void ByOut(out string name) { ... }
}
```

Comparing Indexers to Properties



Indexers are based on properties, and indexers share many of the features of properties. Indexers also differ from properties in certain ways. To understand indexers fully, it is helpful to compare them to properties.

Similarities with Properties

Indexers are similar to properties in many ways:

- Both use **get** and **set** accessors.
- Neither denote physical storage locations; therefore neither can be used as **ref** or **out** parameters.

```
class Dictionary
{
    public string this[string index]
    {
        get { ... }
        set { ... }
    }
}
Dictionary oed = new Dictionary( );
...
Method(ref oed["life"]); // Compile-time error
Method(out oed["life"]); // Compile-time error
```

- Neither can specify a **void** type.

  For example, in the code above, `oed["life"]` is an expression of type **string** and could not be an expression of type **void**.

Differences from Properties

It is also important to understand how indexers and properties differ:

- Identification

  A property is identified only by its name. An indexer is identified by its signature; that is, by the square brackets and the type of the indexing parameters.

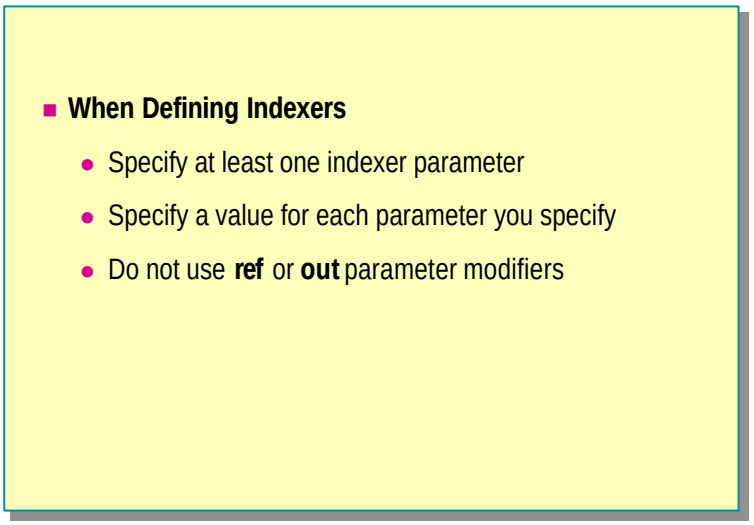
- Overloading

  Since a property is identified only by its name, it cannot be overloaded. However, since an indexer's signature includes the types of its parameters, an indexer can be overloaded.

- Static or dynamic

  A property can be a static member, whereas an indexer is always an instance member.

Using Parameters to Define Indexers



There are three rules that you must follow to define indexers:

- Specify at least one indexer parameter.
- Specify a value for each parameter.
- Do not use **ref** or **out** as parameter modifiers.

Syntax Rules for Indexer Parameters

When defining an indexer, you must specify at least one parameter (index) for the indexer. You have seen examples of this already. There are some restrictions on the storage class of the parameter. For example, you cannot use **ref** and **out** parameter modifiers:

```
class BadParameter
{
    // Compile-time error
    public string this[ref int index] { ... }
    public string this[out string index] { ... }
}
```

Multiple Parameters

You can specify more than one parameter in an indexer. The following code provides an example:

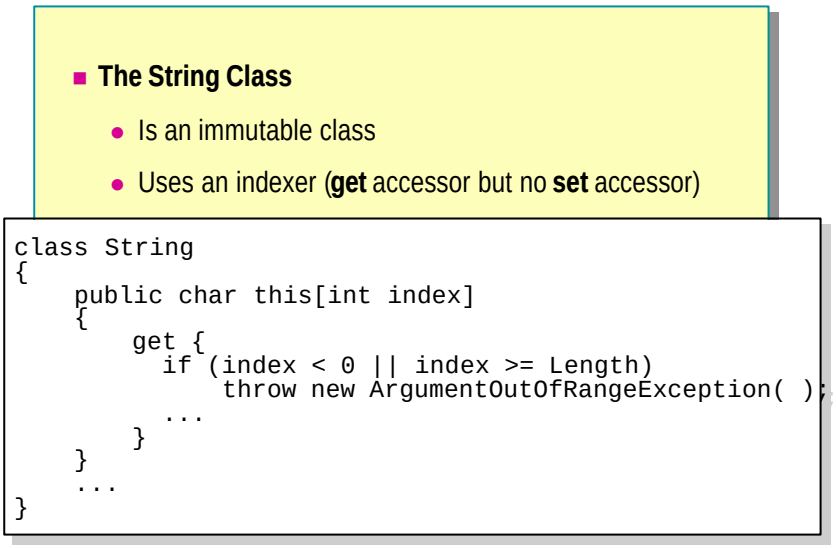
```
class MultipleParameters
{
    public string this[int one, int two]
    {
        get { ... }
        set { ... }
    }
    ...
}
```

To use the indexer of the **MultipleParameters** class, you must specify two values, as shown in the following code:

```
...
MultipleParameters mp = new MultipleParameters( );
string s = mp[2,3];
...
```

This is the indexer equivalent of a multidimensional array.

String Example



The **string** type is a fundamental type in C#. It is a keyword that is an alias for the **System.String** class in the same way that **int** is an alias for the **System.Int32** struct.

The String Class

The **String** class is an immutable, sealed class. This means that when you call a method on a **string** object, you are guaranteed that the method will not change that **string** object. If a **string** method returns a string, it will be a new string.

The Trim Method

To remove trailing white space from a string, use the **Trim** method:

```
public sealed class String {
    ...
    public String Trim( ) { ... }
    ...
}
```

The **Trim** method returns a new trimmed string, but the string used to call **Trim** remains untrimmed. The following code provides an example:

```
string s = "Trim me    ";
string t = s.Trim( );
Console.WriteLine(s); // Writes "Trim me   "
Console.WriteLine(t); // Writes "Trim me"
```

The String Class Indexer

No method of the **String** class ever changes the string used to call the method. You define the value of a string when it is created, and the value never changes.

Because of this design decision, the **String** class has an indexer that is declared with a **get** accessor but no **set** accessor, as shown in the following example:

```
class String
{
    public char this[int index]
    {
        get {
           if (index < 0 || index >= Length)
               throw new ArgumentOutOfRangeException( );
           ...
        }
    }
    ...
}
```

If you attempt to use a string indexer to write to the string, you will get a compile-time error:

```
string s = "Sharp";
Console.WriteLine(s[0]); // Okay
s[0] = 'S'; // Compile-time error
s[4] = 'k'; // Compile-time error
```

The **String** class has a companion class called **StringBuilder** that has a read-write indexer.

BitArray Example

```
class BitArray
{
   public bool this[int index]
   {
     get {
        BoundsCheck(index);
        return (bits[index >> 5] & (1 << index)) != 0;
     }
     set {
        BoundsCheck(index);
        if (value) {
            bits[index >> 5] |= (1 << index);
        } else {
            bits[index >> 5] &= ~(1 << index);
        }
     }
   }
   private int[ ] bits;
}
```

This is a more complex example of indexers, based on the **BitArray** class from the .NET Framework SDK. By implementing indexers, the **BitArray** class uses less memory than the corresponding Boolean array.

Comparing the BitArray Class to a Boolean Array

The following example shows how to create an array of Boolean flags:

```
bool[ ] flags = new bool[32];
flags[12] = false;
```

This code works, but unfortunately it uses a single byte to store each **bool**. The state of a Boolean flag (**true** or **false**) can be stored in a single bit, but a byte is eight bits wide. Therefore, an array of bools uses eight times more memory than it needs.

To address this memory issue, the .NET SDK provides the **BitArray** class, which implements indexers and also uses less memory than the corresponding bool array. Following is an example:

```
class BitArray
{
   public bool this[int index]
   {
     get {
        BoundsCheck(index);
        return (bits[index >> 5] & (1 << index)) != 0;
     }
     set {
        BoundsCheck(index);
        if (value) {
            bits[index >> 5] |= (1 << index);
        } else {
            bits[index >> 5] &= ~(1 << index);
        }
     }
   }
   private int[ ] bits;
}
```

How BitArray Works

To learn how the **BitArray** class works, consider step-by-step what the code is doing:

1. Store 32 **bool**s in one **int**.

   BitArray uses substantially less memory than a corresponding bool array by storing the state for 32 **bool**s in one **int**. (Remember that **int** is an alias for **Int32**.)

2. Implement an indexer:

   ```
   public bool this[int index]
   ```

   The **BitArray** class contains an indexer to allow a **BitArray** object to be used in an array-like manner. In fact, a **BitArray** can be used exactly like a **bool** [ ].

   ```
   BitArray flags = new BitArray(32);
   flags[12] = false;
   ```

3. Extract the individual bits.

   To extract the individual bits, you must shift the bits. For example, the following expression appears frequently because shifting right by 5 bits is equivalent to dividing by 32, because 2*2*2*2*2 == 2^5 == 32. Therefore, the following shift expression locates the **int** that holds the bit at position index:

   ```
   index >> 5
   ```

4. Determine the value of the correct bit.

   After the correct **int** is found, the individual bit (out of all 32) still needs to be determined. You can do this by using the following expression:

   ```
   1 << index
   ```

   To understand how this works, you need to know that when you shift an **int** left only the lowest 5 bits of the second argument are used. (Again, only 5 bits are used because the **int** being shifted has 32 bits.) In other words, the above shift-left expression is semantically the same as the following:

   ```
   1 << (index % 32)
   ```

More Details About BitArray

Following is the **BitArray** class in more detail:

```
class BitArray
{
    public BitArray(int length)
    {
        if (length < 0) throw new ArgumentException( );
        bits = new int[((length - 1) >> 5) + 1];
        this.length = length;
    }

    public int Length
    {
        get { return length; }
    }

    public bool this[int index]
    {
        get {
            BoundsCheck(index);
            return (bits[index >> 5] & (1 << index)) != 0;
        }
        set {
            BoundsCheck(index);
            if (value) {
                bits[index >> 5] |= (1 << index);
            } else {
                bits[index >> 5] &= ~(1 << index);
            }
        }
    }

    private void BoundsCheck(int index)
    {
        if (index < 0 || index >= length) {
            throw new IndexOutOfRangeException( );
        }
    }

    private int[ ] bits;
    private int length;
}
```

Lab 13: Using Properties and Indexers

Objectives

After completing this lab, you will be able to:

- Create properties to encapsulate data within a class.
- Define indexers for accessing classes by using array-like notation.

Prerequisites

Before working on this lab, you must be able to:

- Create and use classes.
- Use arrays and collections.

Estimated time to complete this lab: 30 minutes

Exercise 1
Enhancing the Account Class

In this exercise, you will remove the bank account number and bank account type methods from the **BankAccount** class (which you created in previous labs and is provided here) and replace them with read-only properties. You will also add to the **BankAccount** class a read/write string property for the account holder's name.

↙ To change the account number and type methods into read-only properties

1. Open the Bank.sln project in the *install folder*\Labs\Lab13\Exercise 1\Starter\Bank folder.
2. In the **BankAccount** class, replace the method called **Number** with a read-only property (a property that has a **get** accessor but no **set** accessor). This is shown in the following code:

```
public long Number
{
  get { return accNo; }
}
```

3. Compile the project.

   You will receive error messages. This is because **BankAccount.Number** is still being used as a method in the four overloaded **Bank.CreateAccount** methods.

4. Change these four **Bank.CreateAccount** methods to access the bank account number as a property.

   For example, change

```
long accNo = newAcc.Number( );
```

   to

```
long accNo = newAcc.Number;
```

5. Save and compile the project.
6. In the **BankAccount** class, replace the method called **Type** with a read-only property whose **get** accessor returns **accType.Format.**
7. Save and compile the project.

↙ To add to the BankAccount class a read/write property for the account holder

1. Add a private field called *holder* of type **string** to the **BankAccount** class.
2. Add a public read/write property called **Holder** (note the capital "H") of type **string** to the **BankAccount** class.

   The **get** and **set** accessors of this property will use the holder string you have just created:

```
public string Holder
{
  get { return this.holder; }
  set { holder = value; }
}
```

3. Save your work, compile the project, and correct any errors.
4. Modify the **BankAccount.ToString** method so that the string it returns contains the account holder's name in addition to the account number, type, and balance.

↙ To test the properties

1. Open the TestHarness.sln test harness in the *install folder*\Labs\Lab13\Exercise 1\Starter\TestHarness folder.
2. Add a reference to the Bank library (the DLL that contains the components that you worked on in the previous two procedures) by performing the following steps:
   a. Expand the project in Solution Explorer.
   b. Right-click **References**, and then click **Add Reference**.
   c. Click **Browse**.
   d. Navigate to the *install folder*\Labs\Lab13\Exercise 1\Starter\Bank\Bin\Debug folder.
   e. Click **Bank.dll**, click **Open**, and then click **OK**.
3. Add two statements to the **Main** method of the **CreateAccount** class, as follows:
   - Set the name of the holder of *acc1* to "Sid."
   - Set the name of the holder of *acc2* to "Ted."
4. Add statements that retrieve and print the number and type of each account.
5. Save your work, compile the project, and correct any errors.
6. Run the project and verify that the account numbers, the account types, and the names "Sid" and "Ted" appear.

Exercise 2
Modifying the Transaction Class

In this exercise, you will modify the **BankTransaction** class (which you developed in previous labs and which is provided here). As you may recall, the **BankTransaction** class was created for holding information about a financial transaction pertaining to a **BankAccount** object.

You will replace the methods **When** and **Amount** with a pair of read-only properties. (**When** returns the date of the transaction, **Amount** returns the transaction amount.)

↙ To change the When method into a read-only property

1. Open the Bank.sln project in the *install folder*\Labs\Lab13\Exercise 2\Starter\Bank folder.
2. In the **BankTransaction** class, replace the method called **When** with a read-only property of the same name.
3. Compile the project.

   You will receive an error message. This is because **BankTransaction.When** is still being used as a method in **Audit.RecordTransaction**. (The **Audit** class records an audit trail of transaction information, so it uses the **When** and **Amount** methods to find the date and amount of each transaction.)
4. Change the **Audit.RecordTransaction** method so that it accesses the **When** member as a property.
5. Save your work, compile the project, and correct any errors.

↙ To change Amount into a read-only property

1. In the **BankTransaction** class, replace the method called **Amount** with a read-only property.
2. Compile the project.

   You will receive error messages. This is because **BankTransaction.Amount** is still being used as a method in **Audit.RecordTransaction**.
3. Change the **Audit.RecordTransaction** method so that it accesses the **Amount** member as a property.
4. Save your work, compile the project, and correct any errors.

↙ To test the properties

1. Open the TestHarness.sln test harness in the *install folder*\Labs\Lab13\Exercise 2\Starter\TestHarness folder.
2. Add a reference to the Bank library (the DLL that contains the components that you worked on in the previous procedures) by performing the following steps:
   a. Expand the project in Solution Explorer.
   b. Right-click **References**, and then click **Add Reference**.
   c. Click **Browse**.
   d. Navigate to the *install folder*\Labs\Lab13\Exercise 2\Starter\Bank\Bin\Debug folder.
   e. Click **Bank.dll**, click **Open**, and then click **OK**.
3. Add statements to the **Main** method of the **CreateAccount** class that will:
   a. Deposit money into accounts *acc1* and *acc2*. (Use the **Deposit** method, and make up your own numbers.)
   b. Withdraw money from accounts *acc1* and *acc2*. (Use the **Withdraw** method.)
   c. Print the transaction history for each account. A method called **Write** has been supplied at the end of the test harness. You pass it an account whose transaction history you want to display. It uses and tests the **When** and **Amount** properties of the **BankTransaction** class. Following is an example:

      ```
      Write(acc1);
      ```

4. Save your work, compile the project, and correct any errors.
5. Run the project, and verify that the transaction details appear as expected.

Exercise 3
Creating and Using an Indexer

In this exercise, you will add an indexer to the **BankAccount** class to provide access to any of the **BankTransaction** objects cached in the internal array.

The transactions that belong to an account are accessible by means of a queue (**System.Collections.Queue**) that is in the **BankAccount** object itself.

You will define an indexer on the **BankAccount** class that retrieves the transaction at the specified point in the queue or returns **null** if no transaction exists at that point. For example,

```
myAcc.AccountTransactions[2]
```

will return transaction number 2, the third one in the queue.

The **GetEnumerator** method of **System.Collections.Queue** will be useful in this exercise.

↙ To declare a read-only BankAccount indexer

1. Open the Bank.sln project in the *install folder*\Labs\Lab13\Exercise 3\Starter\Bank folder.
2. In the **BankAccount** class, declare a public indexer that returns a **BankTransaction** and takes a single **int** parameter called **index**, as follows:

```
public BankTransaction this[int index]
{
  ...
}
```

3. Add a **get** accessor to the body of the indexer, and implement it with a single `return new BankTransaction(99);` statement, as follows.

```
public BankTransaction this[int index]
{
  get { return new BankTransaction(99); }
}
```

   The purpose of this step is only to test the syntax of the indexer. Later, you will implement the indexer properly.

4. Save your work, compile the project, and correct any errors.

↙ To create transactions

1. Open the TestHarness.sln test harness in the *install folder*\Labs\Lab13\Exercise 3\Starter\TestHarness folder.
2. Add a reference to the Bank library (the DLL that contains the components that you worked on in the previous stage) by performing the following steps:
   a. Expand the project in Solution Explorer.
   b. Right-click **References**, and then click **Add Reference**.
   c. Click **Browse**.
   d. Navigate to the *install folder*\Labs\Lab13\Exercise 3\Starter\Bank\Bin\Debug folder.
   e. Click **Bank.dll**, click **Open**, and then click **OK**.
3. Create some transactions by adding the following statements to the end of the **CreateAccount.Main** method:

```
for (int i = 0; i < 5; i++) {
    acc1.Deposit(100);
    acc1.Withdraw(50);
}
Write(acc1);
```

   The calls to **Deposit** and **Withdraw** create transactions.
4. Save your work, compile the project, and correct any errors.

   Run the project, and verify that the **Deposit** and **Withdraw** transactions are correctly displayed.

↙ To call the BankAccount indexer

1. The last few statements of the **CreateAccount.Write** method currently display the transactions using by a **foreach** statement, as follows:

```
Queue tranQueue = acc.Transactions( );
foreach (BankTransaction tran in tranQueue) {
    Console.WriteLine("Date: {0}\tAmount: {1}", tran.When,
➥tran.Amount);
}
```

2. Change the way transactions are displayed as follows:
   a. Replace this **foreach** statement with a **for** statement that increments an **int** variable called *counter* from zero to the value returned from **tranQueue.Count**.
   b. Inside the **for** statement, call the **BankAccount** indexer that you declared in the previous procedure. Use *counter* as the subscript parameter, and save the returned **BankTransaction** in a local variable called *tran.*
   c. Print the details from *tran*:

```
for (int counter = 0; counter < tranQueue.Count;
➥counter++) {
  BankTransaction tran = acc[counter];
  Console.WriteLine("Date: {0}\tAmount: {1}", tran.When,
➥tran.Amount);
}
```

3. Save your work, compile the project, and correct any errors.
4. Run the project.

   It will display a series of transactions with a value of 99 (the temporary test value that you used earlier) because the indexer has not yet been fully implemented.

↙ To complete the BankAccount indexer

1. Return to the Bank project (Bank.sln in the *install folder*\Labs\Lab13\Exercise 3\Starter\Bank folder).
2. In the **BankAccount** class, delete the
   `return new BankTransaction(99);`
   statement from the body of the indexer.
3. The **BankAccount** transactions are held in a private field called *tranQueue* of type **System.Collections.Queue**. This **Queue** class does not have an indexer, so to access a given element you will need to manually iterate through the class. The process for doing this is as follows:
   a. Declare a variable of type *IEnumerator* and initialize it by using the **GetEnumerator** method of *tranQueue.* (All queues provide an enumerator to allow you to step through them.)
   b. Iterate through the queue *n* times, using the **MoveNext** method of the *IEnumerator* variable to move to the next item in the queue.
   c. Return the **BankTransaction** found at the *n*th location.

Your code should look as follows:

```
IEnumerator ie = tranQueue.GetEnumerator( );
for (int i = 0; i <= index; i++) {
    ie.MoveNext( );
}
BankTransaction tran = (BankTransaction)ie.Current;
return tran;
```

4. Check that the **int** parameter *index* is neither greater than **tranQueue.Count** nor less than zero.

   Check for this before iterating through **tranQueue**.

5. The complete code for the indexer should look as follows:

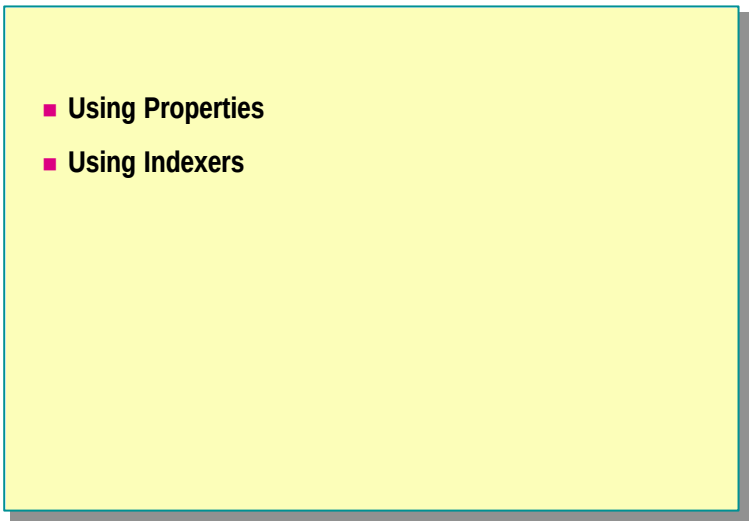
```
public BankTransaction this[int index]
{
  get
  {
     if (index < 0 || index >= tranQueue.Count)
          return null;

     IEnumerator ie = tranQueue.GetEnumerator( );
     for (int i = 0; i <= index; i++)  {
         ie.MoveNext( );
     }
     BankTransaction tran = (BankTransaction)ie.Current;
     return tran;
   }
}
```

6. Save your work, compile the project, and correct any errors.
7. Return to TestHarness and execute it.

   Verify that all ten transactions appear correctly.

Review



1. Declare a **Font** class that contains a read-only property called **Name** of type **string**.

2. Declare a **DialogBox** class that contains a read/write property called **Caption** of type **string**.

3. Declare a **MutableString** class that contains a read/write indexer of type **char** that expects a single **int** parameter.

4. Declare a **Graph** class that contains a read-only indexer of type **double** that expects a single parameter of type **Point**.