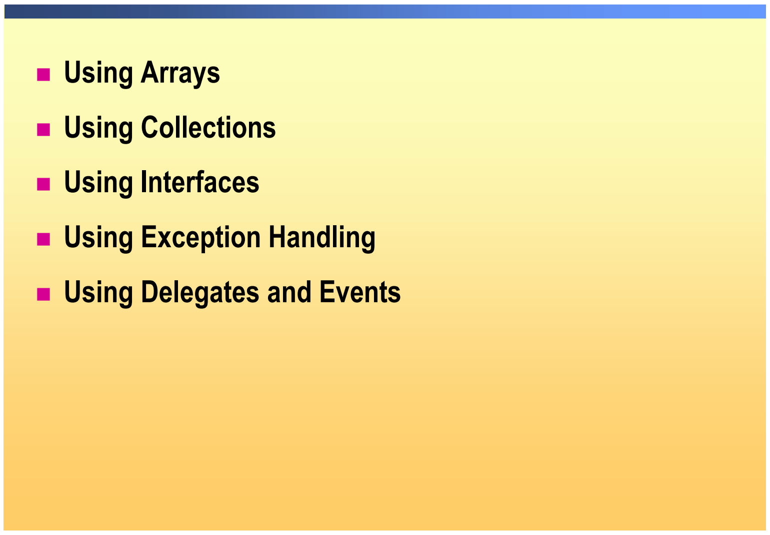
msdn training

Module 5: Programming with C#

Contents

Microsoft

Overview



- Using Arrays
- Using Collections
- Using Interfaces
- Using Exception Handling
- Using Delegates and Events

Introduction

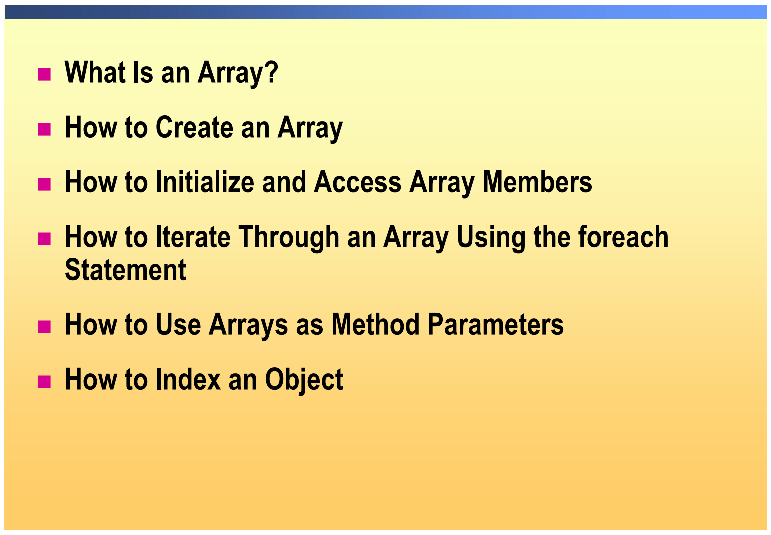
This module introduces various data structures including arrays (the **System.Array** class) and collections (classes in the **System.Collections** namespace), and explains when to use each of these data structures in an application. The module also introduces interfaces, describes the concepts and syntax of exception handling, and explains delegates and their use in event handling.

Objectives

After completing this module, you will be able to:

- Create and use arrays.
- Use classes in the **System.Collections** namespace.
- Use the **ArrayList** class.
- Use interfaces.
- Handle exceptions.
- Create and call a delegate.
- Use delegates to handle events.

Lesson: Using Arrays


- What Is an Array?
- How to Create an Array
- How to Initialize and Access Array Members
- How to Iterate Through an Array Using the foreach Statement
- How to Use Arrays as Method Parameters
- How to Index an Object

Introduction

This lesson introduces arrays and the **Array** class in the **System** namespace. It explains how to use arrays to hold a series of objects, and how to index the array.

Lesson objectives

After completing this lesson, students will be able to:

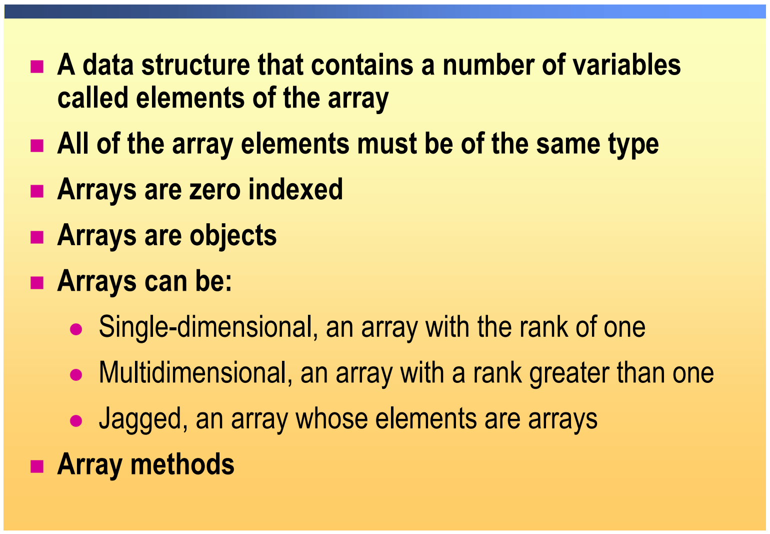
- Create an array.
- Index an array.
- Use the **foreach** statement to iterate through the items in the array.

Lesson agenda

This lesson includes the following topics and activity:

- What Is an Array?
- How to Create an Array
- How to Initialize and Access Array Members
- How to Iterate Through an Array Using the **foreach** Statement
- How to Use Arrays as Method Parameters
- How to Index an Object
- Practice: Using **foreach** with an Array
- Practice (optional): Using an Indexer

What Is an Array?



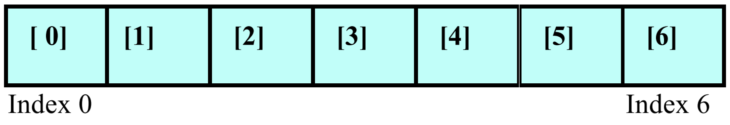
- A data structure that contains a number of variables called elements of the array
- All of the array elements must be of the same type
- Arrays are zero indexed
- Arrays are objects
- Arrays can be:
  - Single-dimensional, an array with the rank of one
  - Multidimensional, an array with a rank greater than one
  - Jagged, an array whose elements are arrays
- Array methods

Introduction

An array can be thought of as a list. By using arrays, you can store a group of elements that have the same data type under one variable name. You can also easily access, use, and store values of the same type. An array is a good choice when you want to maintain a list of items.

Definition

An *array* is a data structure that contains a number of variables called the *elements* of the array. To refer to a specific element in the series, you use a number, or *index.* C# arrays are *zero indexed*; that is, the array indexes start at zero. Arrays are objects.



Single dimensional array

An array that consists of a single list or sequence is called a *single-dimensional array*. An array has one or more dimensions.

Multidimensional array

A *multidimensional array* is indexed by more than one value. Multidimensional arrays of specific sizes are often referred to by size, such as two-dimensional arrays and three-dimensional arrays. You can think of a two-dimensional array as a grid. For example, you can store a set of graph coordinates, such as x and y, in a 2-dimensional array.

Analogy

Consider a shelf full of books as a single-dimensional array. The shelf is the array dimension and a book is an element in the array. A bookcase is more like a multidimensional array, with the shelves being one dimension on the array, and the books being another dimension. For example, you would refer to the third book on the second shelf.

Jagged array

The elements of an array can be any type, including an array type. An array of arrays is called a *jagged array*.

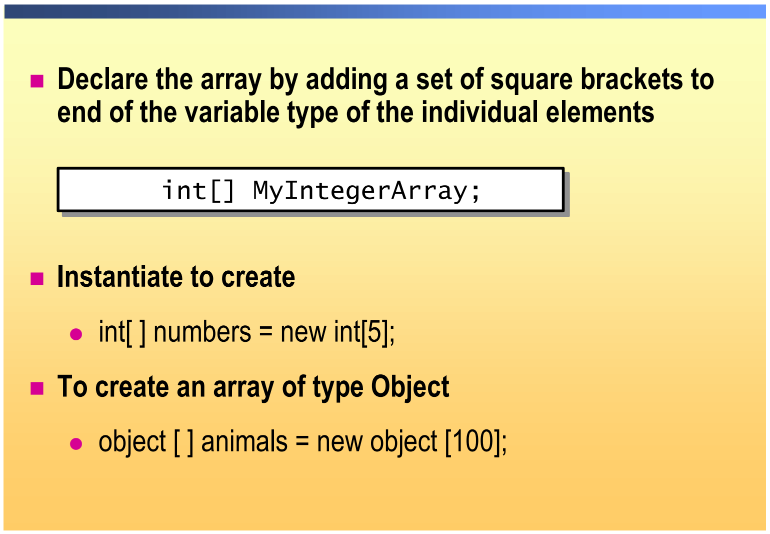
Array methods

C# natively supports arrays, based on the Microsoft® .NET Framework class **System.Array**. The **System.Array** class is an abstract class that provides many methods that you can use when working with arrays.

The following table includes some of the most commonly used methods.

Method	Description
Sort	Sorts the elements in an array
Clear	Sets a range of elements to zero or **null**
Clone	Creates a copy of the array
GetLength	Returns the length of a given dimension
IndexOf	Returns the index of the first occurrence of a value
Length	Gets the number of elements in the specified dimension of the array

How to Create an Array



- Declare the array by adding a set of square brackets to end of the variable type of the individual elements

```
int[] MyIntegerArray;
```

- Instantiate to create
  - int[ ] numbers = new int[5];
- To create an array of type Object
  - object [ ] animals = new object [100];

Introduction

Before you can use an array, you must create it by declaring it and then instantiating it.

Syntax

You create or declare arrays in code just as you declare other variables. You follow the same guidelines for naming, scoping, and choosing data types. When you declare an array, you place the brackets ([ ]) after the type. These square brackets are called the *index operators*.

```
int[] MyIntegerArray;

int[] table;
```

Declaring an array

To declare an array, use the following syntax:

```
type[] array-name;
```

For example:

```
int[] numbers;
```

Instantiating an array

Arrays in C# are objects and must be instantiated. When you instantiate the array, you set aside memory, on the heap, to hold the elements of the array.

The following code allocates space for 5 integers:

```
int[] numbers;
numbers = new int[5];
```

As with other variable declarations, you can combine these statements as follows:

```
int[] numbers = new int[5];
```

Initial values

When you create an array of value types, the contents of the array are initialized to the default value for that type. For an integer array, the default value is zero.

Examples

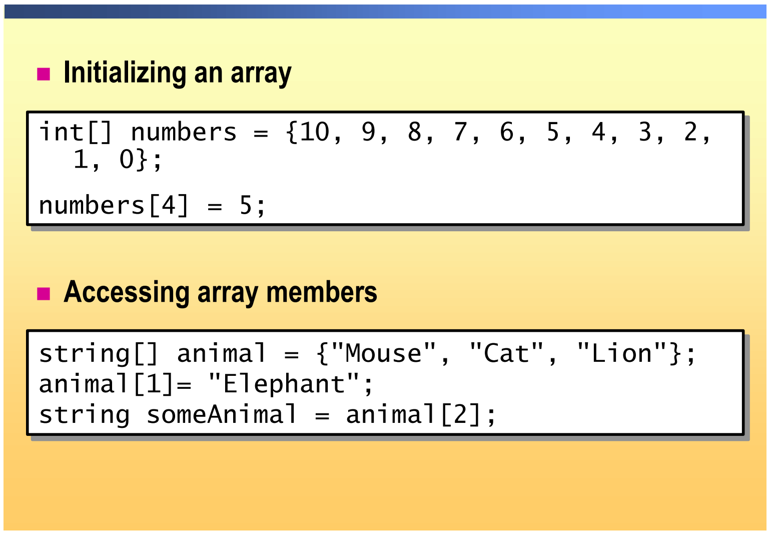
The array can be of any type. For example, you can maintain a list of names or bank account balances in an array.

```
string[] names = new names[7];
decimal[] balances = new balances[10];
```

You can also create an array of type **Object** as shown in the following example. Creating this type of array can be useful if you must manage a list of many different types of objects.

```
object[] animals = new object[100];
```

How to Initialize and Access Array Members



- **Initializing an array**

```
int[] numbers = {10, 9, 8, 7, 6, 5, 4, 3, 2,
  1, 0};
numbers[4] = 5;
```

- **Accessing array members**

```
string[] animal = {"Mouse", "Cat", "Lion"};
animal[1]= "Elephant";
string someAnimal = animal[2];
```

Introduction

C# provides simple ways to initialize arrays and access array members.

Initializing a single-dimensional array

To initialize a single-dimensional array when it is declared, enclose the initial values in curly braces { }.

Note If an array is not initialized when it is declared, array members are automatically initialized to the default initial value for the array type.

The following examples show various ways to initialize single-dimensional arrays:

```
int[] numbers = new int[5] {1, 2, 3, 4, 5};

string[] animals = new string[3] {"Elephant", "Cat", "Mouse"};
```

If an initializer is provided, you can omit the **new** statement, as shown in the following examples:

```
int[] numbers = {1, 2, 3, 4, 5};

string[] animals = {"Elephant", "Cat", "Mouse"};
```

Note that the size of the array is inferred from the number of elements that are specified.

Accessing array members

Accessing array members in C# is straightforward. For example, the following code creates an array called **numbers** and then assigns **5** to the fifth element of the array:

```
int[ ] numbers = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
numbers[4] = 5;
```

Example

The following code creates an array of 100 integers and fills them with the values **100** down to **1**:

```
int[] countdown = new int[100];

for ( int i = 0, j = 100; i < 100; i++, j-- ) {
  countdown[i] = j;
}
```

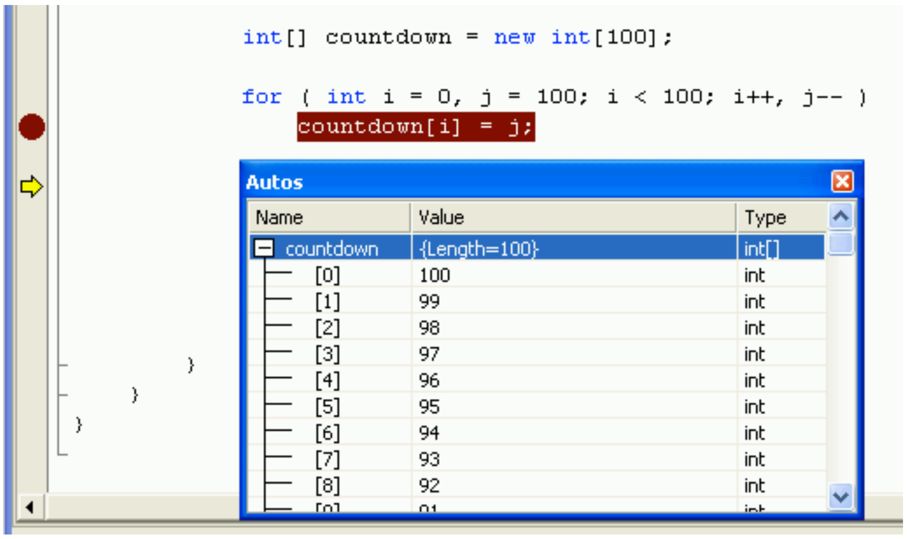
When you declare an array to include a number of elements, you must access only valid array elements. If you attempt to access an element that is out of range, you generate an **IndexOutOfRange** exception, a type of run-time error.

For example, the following code generates a run-time error because the array allocates 5 integers but attempts to access the sixth element in the array. Remember that arrays are zero-indexed.

```
int[] errorArray = new int[5];
errorArray[5] = 42;  // runtime error
```

Examining arrays using the debugging tool

It is very useful to check the contents of an array while your application is running. The debugging tool provides excellent access to array element values through the various debugging windows such as the **Autos** window and the **Watch** window:



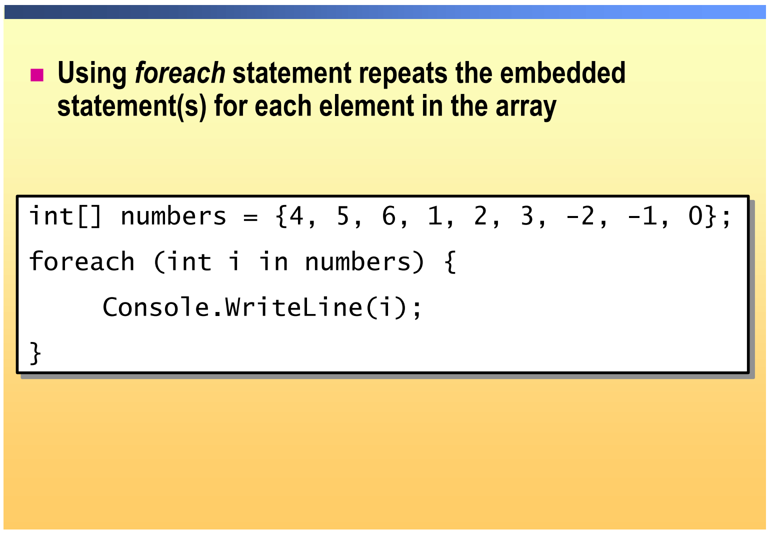
```
int[] countdown = new int[100];

for ( int i = 0, j = 100; i < 100; i++, j-- )
    countdown[i] = j;
```

Autos

Name	Value	Type
countdown	{Length=100}	int[]
[0]	100	int
[1]	99	int
[2]	98	int
[3]	97	int
[4]	96	int
[5]	95	int
[6]	94	int
[7]	93	int
[8]	92	int

To examine the contents of an array, set a breakpoint at the array, and then, in the Debug window, click the expand button next to the array name.

How to Iterate Through an Array Using the *foreach* Statement



- Using *foreach* statement repeats the embedded statement(s) for each element in the array

```
int[] numbers = {4, 5, 6, 1, 2, 3, -2, -1, 0};
foreach (int i in numbers) {
     Console.WriteLine(i);
}
```

Introduction

Code that iterates through an array and executes the same statements on each element is very common. Although you can use any looping statement to iterate through an array, the **foreach** statement simplifies this common process by eliminating the need for you to check the size of the array.

Syntax

The syntax of the **foreach** statement is:

```
foreach ( type identifier in expression )
      statement-block
```

Using the foreach statement

The following code creates an array named **numbers**, iterates through the array with the **foreach** statement, and then writes the values out to the console, one per line:

```
int[] numbers = {4, 5, 6, 1, 2, 3, -2, -1, 0};
foreach (int i in numbers) {
      Console.WriteLine(i);
}
```

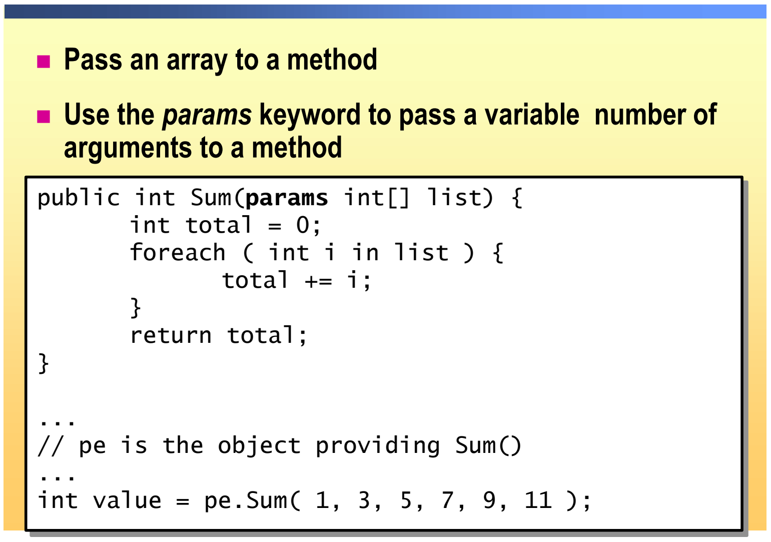
Important You must not modify the iteration variable in a **foreach** loop. In the preceding example, **i** is the iteration variable.

Example

This example shows how the **foreach** statement can be useful when you work with an array of objects. Two classes, **Lion** and **Elephant**, are derived from **Animal**. Both of the derived classes implement the **Eat** method. The classes are instantiated, and then a **Lion** and an **Elephant** object are placed in an array. The **foreach** statement is used to iterate through the array so that the polymorphic **Eat** method is called. Note that the base object type is used to specify the type of the identifier.

```
using System;

namespace ArraySample {
  public abstract class Animal {
      abstract public void Eat();
  }

  public class Lion : Animal {
      public override void Eat() {
          // eats meat
      }
  }

  public class Elephant : Animal {
      public override void Eat() {
          // eats vegetation
      }
  }

  class ClassZoo {
      static void Main(string[] args) {
          Lion aLion = new Lion();
          Elephant anElephant = new Elephant();

          Animal[] zoo = new Animal[2];
          zoo[0] = aLion;
          zoo[1] = anElephant;

          foreach ( Animal a in zoo ) {
              a.Eat();
          }
       }
   }
}
```

How to Use Arrays as Method Parameters



- **Pass an array to a method**
- **Use the *params* keyword to pass a variable number of arguments to a method**

```
public int Sum(params int[] list) {
      int total = 0;
      foreach ( int i in list ) {
            total += i;
      }
      return total;
}

...
// pe is the object providing Sum()
...
int value = pe.Sum( 1, 3, 5, 7, 9, 11 );
```

Introduction

You may want to write a method that can accept an unpredictable number of parameters, for example, a method that would return the sum of any set of integers.

Passing an array to a method

To write this type of method, you can place the integers in an array, pass the array to the method, and then use the **foreach** statement to iterate through the array.

Example

The following example shows how to do this with the **Sum** method. Note the declaration of the **Sum** method and note the declaration and initialization of the **tester** array.

```
using System;

namespace ParameterExample {
  public class ParamExample {
      public int Sum(int[] list) {
          int total = 0;
          foreach ( int i in list ) {
              total += i;
          }
          return total;
      }
  }

  class Tester {
      static void Main(string[] args) {
          ParamExample pe = new ParamExample();
          int[] tester = {1, 2, 3, 4, 5, 6, 7 };

          int total = pe.Sum( tester );

          Console.WriteLine( total );  // 28
      }
   }
}
```

Using params keyword

Although this approach works, C# provides a better solution by allowing you to use the **params** keyword rather than creating the array yourself. When you place the **params** keyword before the array declaration in the parameter list, you can use the method as shown in the following example:

```
using System;

namespace ParameterExample {
  public class ParamExample {
      public int Sum(params int[] list) {
          int total = 0;
          foreach ( int i in list ) {
              total += i;
          }
          return total;
      }
  }

  class Tester {
      static void Main(string[] args) {
          ParamExample pe = new ParamExample();

          int total = pe.Sum( 1, 2, 3, 4, 5, 6, 7 );

          Console.WriteLine( total );  // 28
      }
   }
}
```

The **params** keyword can modify any type of parameter. A **params** parameter need not be the only parameter. For example, you can add the following method to the **ParamExample** class:

```
class ParamExample {
    public string Combine(string s1, string s2,
                                  params object[] others) {
        string combination = s1 + " " + s2;
        foreach ( object o in others ) {
            combination += " " + o.ToString();
        }
        return combination;
    }
}
```

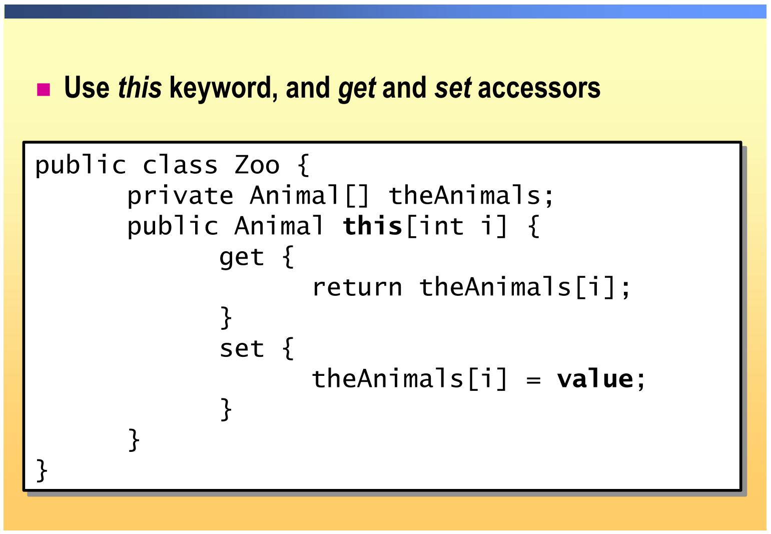
You can use this method as follows:

```
string combo = pe.Combine("One", "two", "three", "four" );
// combo has the value "One two three four"

combo = pe.Combine("alpha", "beta");
// combo has the value "alpha beta"
```

Notice how this method is implemented in the preceding example. The first call to **pe.Combine** matches the method that has the **params** parameter, and the compiler creates an array and then passes that array to your method. The second call to **pe.Combine** matches the method that takes two string parameters, and the compiler does not create an array. When using the **params** keyword, you must consider the overhead that is involved.

Tip If you expect that the users of your method are normally going to pass one, two, or three parameters, it is a good idea to create several overloads of the method that can handle those specific cases.

How to Index an Object



- Use *this* keyword, and *get* and *set* accessors

```
public class Zoo {
      private Animal[] theAnimals;
      public Animal this[int i] {
            get {
                  return theAnimals[i];
            }
            set {
                  theAnimals[i] = value;
            }
      }
}
```

Introduction

When a class contains an array, or a collection, it is useful to access the information as though the class itself were an array. An *indexer* is a property that allows you to index an object in the same way as an array.

Declaring an indexer

To declare an indexer in C#, you use the **this** keyword. Like properties, indexers can contain **get** and **set** clauses, as shown in the following example:

```
type this [ type index-argument ] { get-accessor; set-
accessor; }
```

The get accessor

The **get** accessor uses the same index-argument as the indexer, as shown in the following example:

```
public numbers this[int i] {
  get {
     return myIntegerArray[i];
  }
}
```

The set accessor

The **set** accessor uses the same index-argument as the indexer, in addition to the **value** implicit parameter, as shown in the following example:

```
set {
   myArray[i] = value;
}
```

Example

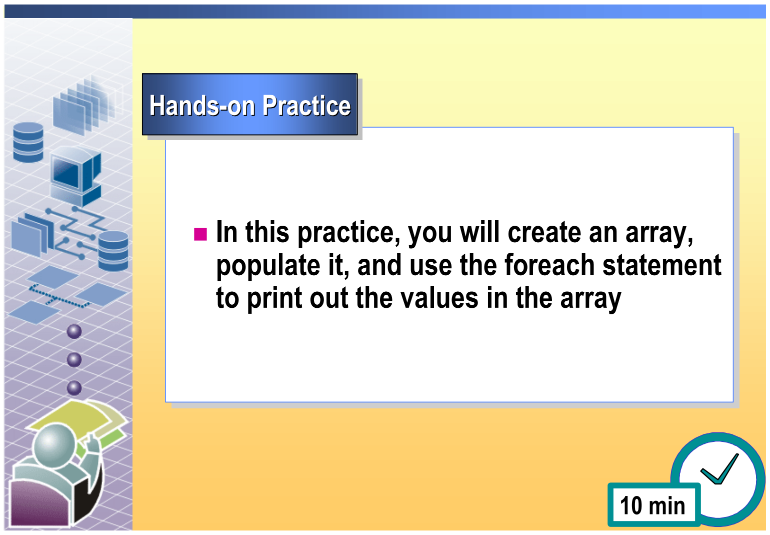
In the following complete example, the **Zoo** class maintains a private array of **Animal** objects named **theAnimals**. An indexer is provided so that users of the **Zoo** class can access the animals in **Zoo** just like an array, as shown in **Main** method.

```
using System;

namespace IndexExample {

  public class Zoo {
      private Animal[] theAnimals;
      public Animal this[int i] {
          get {
              return theAnimals[i];
          }
          set {
              theAnimals[i] = value;
          }
      }

      public Zoo() {
          // Our Zoo can hold 100 animals
          theAnimals = new Animal[100];
      }
  }

  class ZooKeeper {
      static void Main(string[] args) {
          Zoo myZoo = new Zoo();
          myZoo[0] = new Elephant();
          myZoo[1] = new Lion();
          myZoo[2] = new Lion();
          myZoo[3] = new Antelope();

          Animal oneAnimal = myZoo[3];
          // oneAnimal gets an antelope
      }
   }

  public abstract class Animal {
      abstract public void Eat();
  }
  public class Lion : Animal {
      public override void Eat() { }
  }
  public class Elephant : Animal {
      public override void Eat() { }
  }
  public class Antelope : Animal {
      public override void Eat() { }
  }
}
```

Instead of using the indexer, you can either make the array public, which violates the design principals of encapsulation, or write methods to add and remove animals from the array.

The preceding code is available on the Student Materials compact disc in the IndexObject.sln file in the Samples\Mod05\Indexers folder.

Practice: Using a *foreach* Statement with an Array


Hands-on Practice

- In this practice, you will create an array, populate it, and use the foreach statement to print out the values in the array

In this practice, you will use a **foreach** loop to list the contents of an array. Each **Animal** object in the array has a property named **Species** that returns the animal species as a string. You can display this property by using the **Output** method that is provided in the starter code as shown in the following code, assuming **anAnimal** is an **Animal** object:

```
Output ( anAnimal.Species );
```

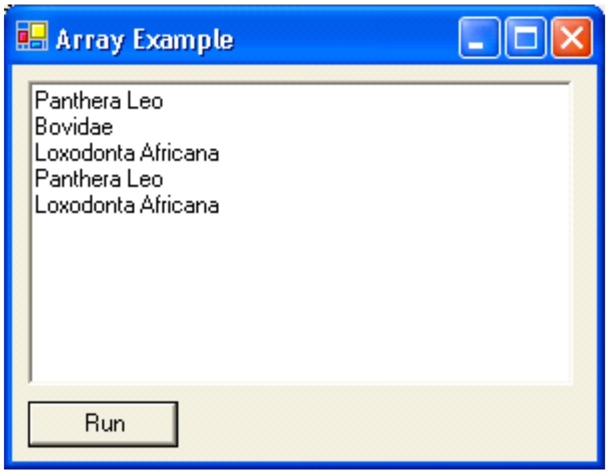
The array **animalArray** is defined as follows:

```
Animal animalArray[] = new Animal[5];
```

Animals are assigned to the array as follows:

```
animalArray[0] = new Lion();
```

The output is shown in the following illustration:

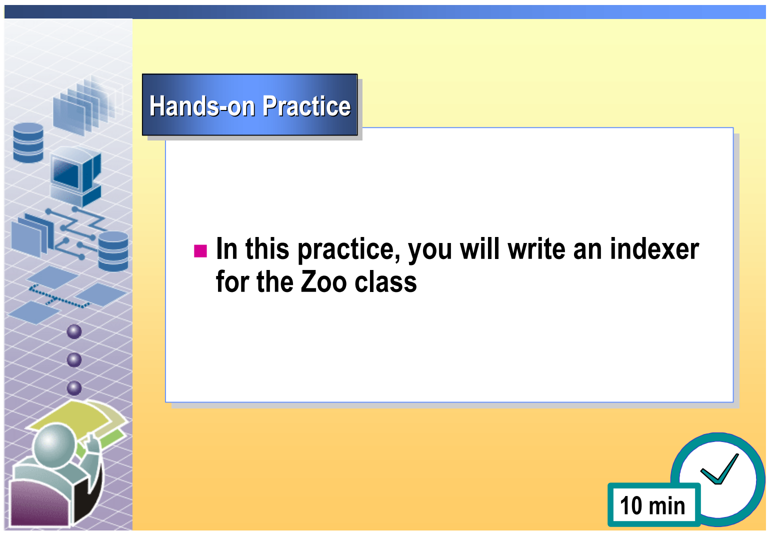


```
Panthera Leo
Bovidae
Loxodonta Africana
Panthera Leo
Loxodonta Africana
```

The solution for this practice is located in *install_folder*\Practices\Mod05\Arrays_Solution \ArrayExample.sln. Start a new instance of Microsoft Visual Studio® .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open *install_folder* \Practices\Mod05\Arrays \ArrayExample.sln.	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start Page**, click **Open Project**. **c.** In the **Open Project** dialog box, browse to *install_folder*\Practices \Mod05\Arrays, click **ArrayExample.sln**, and then click **Open**. **d.** In Solution Explorer, click **Form1.cs**, and then press F7 to open the Code Editor.
2. Review the Task List.	**a.** On the **View** menu, point to **Show Tasks**, and then click **All**. **b.** Review the tasks in the Task List.
3. Write a **foreach** loop that lists the contents of the **animalArray** array.	**a.** Double-click the task **TODO: Write a foreach loop that lists the animals**. Note that the **animalArray** has been declared and initialized with some animals. **b.** Write a **foreach** loop that displays the **Species** of every animal in the **animalArray**. **c.** Press F5 to compile and run your application. In your application window, click **Run**. Your application output should be the same as that shown in the introduction to this practice.
4. Save your solution, and the quit Visual Studio .NET.	**a.** On the **File** menu, click **Save All**. **b.** On the **File** menu, click **Exit**.

Practice (optional): Using an Indexer


Hands-on Practice

- In this practice, you will write an indexer for the Zoo class

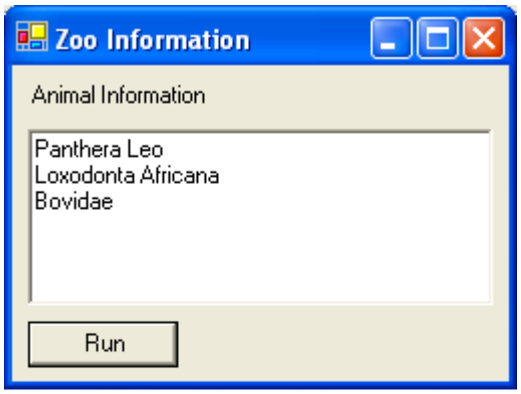
In this practice, you will write an indexer for the **Zoo** class. Currently, the **Zoo** class maintains **Animal** objects in a public array named **animalsArray**. When you click the **Run** button, a **Zoo** object named **myZoo** is created, and three **Animal** objects are created and placed in the **animalsArray** array in **myZoo**.

Next, a **for** loop is used to display the species of the animals in the **animalsArray** in the **Zoo** object.

Your tasks are to:

- Write an indexer to modify the array.
- Use the indexer to add the animals to the array (code is provided).
- Display the animals in the array, using the indexer (code is provided).

The solution will look like this:

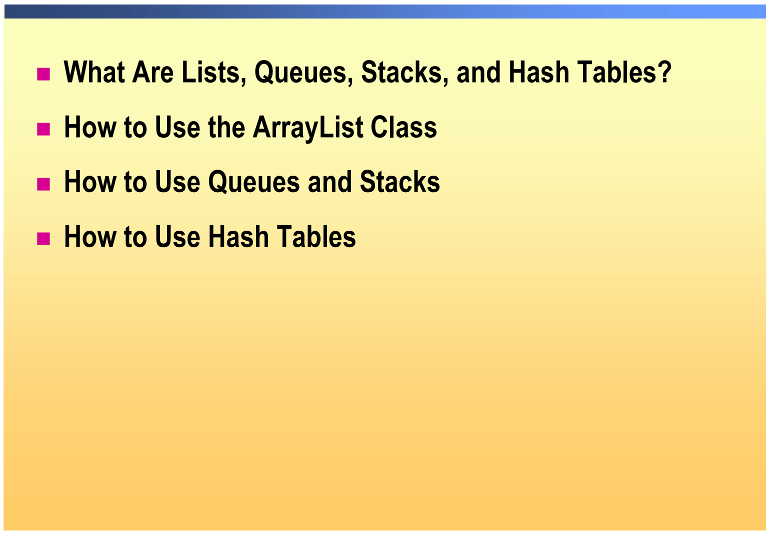


```
Panthera Leo
Loxodonta Africana
Bovidae
```

The solution for this practice is located in *install_folder*\Practices\Mod05\Indexers_Solution\Indexers.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open *install_folder* \Practices\Mod05\Indexers \Indexers.sln.	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start Page**, click **Open Project**. **c.** In the **Open Project** dialog box, browse to *install_folder*\Practices\Mod05\Indexers, click **Indexers.sln**, and then click **Open**. **d.** In Solution Explorer, click **Form1.cs**, and then press F7 to open the Code Editor.
2. Review the Task List.	**a.** On the **View** menu, point to **Show Tasks**, and then click **All**. **b.** Review the tasks in the Task List.
3. Locate and complete the task **Write an Indexer for this class, based on animalArray**.	**a.** Double-click the task **TODO: Write an Indexer for this class, based on animalArray**. **b.** Write an indexer for the **Zoo** class. In the **get** accessor, return an **Animal** object from the **animalArray**. In the **set** accessor, add an **Animal** object to the **animalArray**.
4. Comment out the lines in the **Zoo** class **runExample_Click** method that directly use **animalArray**.	To test your indexer, you will comment out the code in the **runExample_click** method that uses **animalArray** directly, and uncomment the code that uses the indexer. **a.** Double-click the task **TODO 2 : Comment out the following:** **b.** Comment out the lines that assign animals to the **animalArray**, and comment out the for loop immediately following these assignments. Note that you can use the **Comment Selection** function to comment out selected lines of code. This function is available on the **Edit** menu under **Advanced**, and also as a toolbar button, as shown on the left.
5. Uncomment the code that uses the Zoo indexer to read assign the animals and display them.	**a.** Locate the task **TODO 3: Uncomment the following lines**. **b.** Remove the comments from the lines following this task description.
6. Compile and run your application, and test that the animal names display correctly when you click **Run**.	▪ Press F5 to compile and run your application, and then click **Run** on your application window. The output should list the three animal species.
7. Quit Visual Studio .NET, saving your solution.	**a.** On the **File** menu, click **Save All**. **b.** On the **File** menu, click **Exit**.

Lesson: Using Collections



- What Are Lists, Queues, Stacks, and Hash Tables?
- How to Use the ArrayList Class
- How to Use Queues and Stacks
- How to Use Hash Tables

Introduction

This lesson introduces the various data structures in the **Collections** namespace, with special attention given to the **ArrayList** class. The lesson explains how to create an **ArrayList** class, add items to a collection, iterate through the items in a collection, and use hash tables to access collection elements.

Lesson objectives

After completing this lesson, you will be able to:

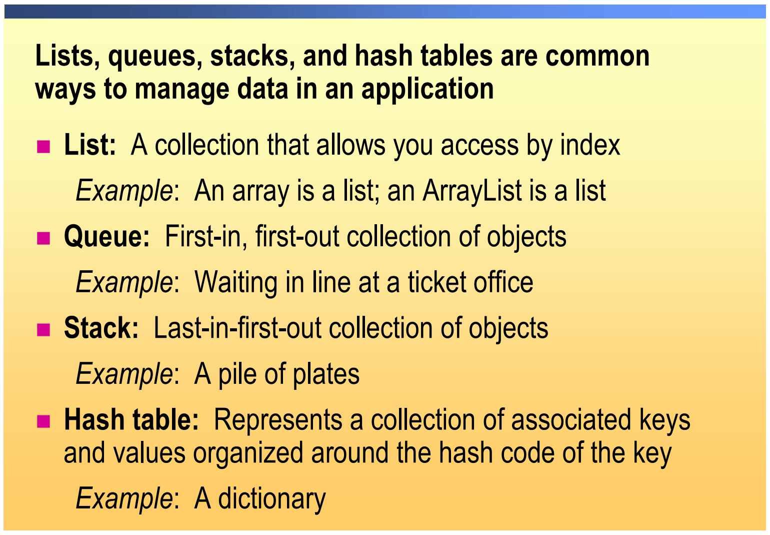
- Create and use collections.
- Use a hash table to access collection elements.

Lesson agenda

This lesson includes the following topics and activity:

- What Are Lists, Queues, Stacks, and Hash tables?
- How to Use the **ArrayList** Class
- How to Use Queues and Stacks
- How to Use Hash Tables
- Practice: Creating and Using Collections

What Are Lists, Queues, Stacks, and Hash Tables?



Lists, queues, stacks, and hash tables are common ways to manage data in an application

- **List:** A collection that allows you access by index

  Example: An array is a list; an ArrayList is a list
- **Queue:** First-in, first-out collection of objects

  Example: Waiting in line at a ticket office
- **Stack:** Last-in-first-out collection of objects

  Example: A pile of plates
- **Hash table:** Represents a collection of associated keys and values organized around the hash code of the key

  Example: A dictionary

Introduction

An array is a useful data structure, but it has some limitations. For example, when you create an array, you must know how many elements you will need, and accessing the element by a sequential index may not be the most convenient method for your application. Lists, queues, stacks, and hash tables are other common ways to manage data in an application.

Definition

The **System.Collections** namespace contains interfaces and classes that define various collections of objects, such as lists, queues, stacks, and hash tables, that provide a useful variety of data structures.

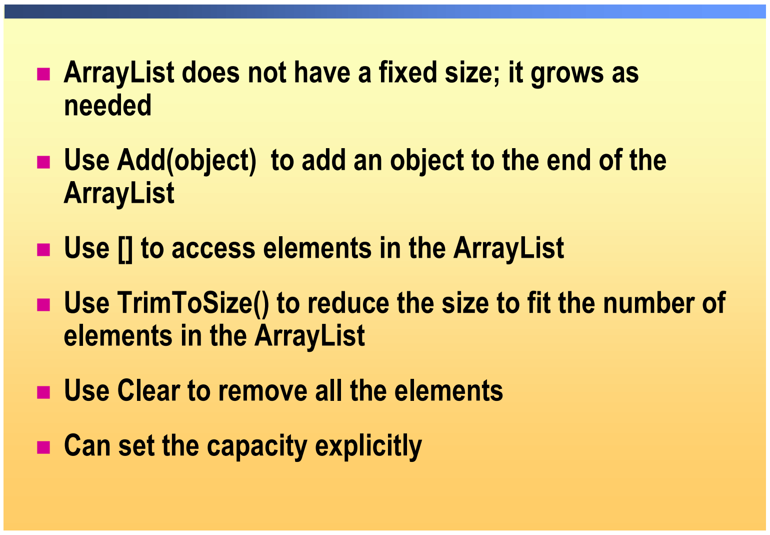
Many of the objects in the .NET Framework classes use collections to manage their data, so understanding these data structures is critical to your ability to successfully create C# applications.

Description

The following table shows some of the classes in the **System.Collections** namespace and illustrates their best uses through examples.

Class	Description	Use	Example
ArrayList	Represents an ordered collection of objects that can be individually indexed.	Use an ArrayList when you want to access elements by using an index. In almost every situation, an ArrayList is a good alternative to an array.	Mailboxes: items can be inserted or removed at any position.
Queue	Represents a first-in, first-out collection of objects.	Use a queue when you need first-in, first-out access. A queue is often used to hold elements in that are discarded immediately thereafter, such as information in a buffer.	Waiting in line at a ticket office, where you join at the back and leave from the front. Requests coming over a network are queued and then discarded after they are processed.
Stack	Represents a simple last-in, first-out collection of objects.	Use a stack when you need last-in, first-out access. A stack is often used to hold items during calculations.	A pile of plates, in a cupboard, where you place them on top, and remove them from the top.
Hashtable	Uses a key to access the elements in the collection.	Use a hash table when you must access elements by using an index and you can identify a useful index value.	You can access book titles by their ISBN numbers.

How to Use the ArrayList Class



- ArrayList does not have a fixed size; it grows as needed
- Use Add(object) to add an object to the end of the ArrayList
- Use [] to access elements in the ArrayList
- Use TrimToSize() to reduce the size to fit the number of elements in the ArrayList
- Use Clear to remove all the elements
- Can set the capacity explicitly

Introduction

The **ArrayList** class solves the main disadvantage of an array, which is that you must know the capacity of the data structure when you instantiate the array, by providing a data structure that behaves like an array but can grow as required. As elements are added to an **ArrayList** object, the capacity is automatically increased. An **ArrayList** object initially allocates 16 elements. When you add a seventeenth element, the **ArrayList** expands to 32 elements.

Accessing elements in an ArrayList

You can access elements in an **ArrayList** object in the same way that you access arrays. You can also use **ArrayList** methods to add elements to or remove elements from an **ArrayList**. To decrease the capacity of an **ArrayList**, you can call the **TrimToSize** method or explicitly set the **Capacity** property.

Use the **Add** method to add items to an **ArrayList**.

You can also use **foreach** to iterate over items in an **ArrayList**.

Note that **ArrayList** elements are objects, such as **System.Object**, so when you retrieve the elements from the list, you most likely must perform type conversion.

Methods

Method	Use
Add	Adds an object to the end of the ArrayList.
Remove	Removes the first occurrence of a specific object from the ArrayList.
Clear	Removes all elements from the ArrayList.
Insert	Inserts an element into the ArrayList at the specified index.
TrimToSize	Sets the capacity to the actual number of elements in the ArrayList.
Sort	Sorts the elements in the ArrayList.
Reverse	Reverses the elements in the ArrayList.

Example

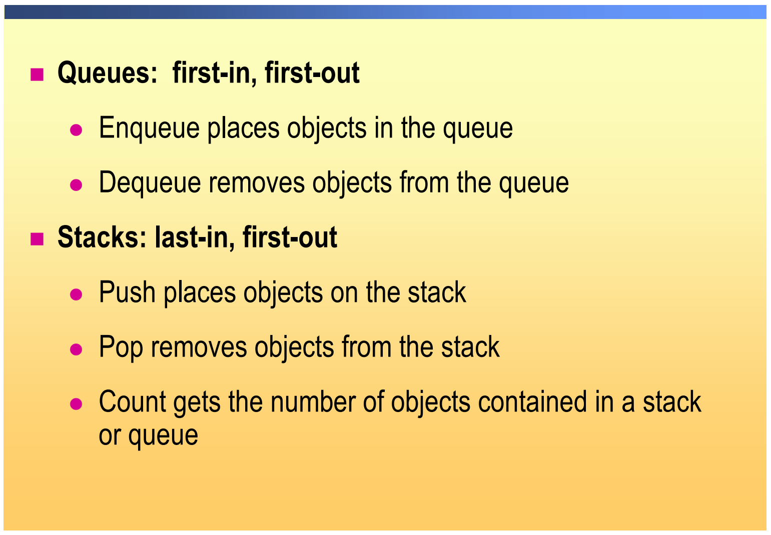
The following code shows how to use an **ArrayList**. Note the following points:

- You must include the **System.Collections** namespace.
- The ArrayList (**theAnimals**) is initialized without specifying its size, because it will grow as needed.
- The **Add** and **Insert** methods are used to add elements to the array. This is the difference between arrays and ArrayLists.
- In both of the places where elements are retrieved from the ArrayList, they must be converted to the type of the variable to which they are being assigned.
- You can access the elements of the ArrayList by using the index operator [].

```
using System;
using System.Collections;

namespace ArrayListExample {
  public class Zoo {
      private ArrayList  theAnimals;
      public ArrayList ZooAnimals {
          get {
              return theAnimals;
          }
      }
      public Animal this[int i] {
          get {
              return (Animal) theAnimals[i];
          }
          set {
              theAnimals[i] = value;
          }
      }
      public Zoo() {
          theAnimals = new ArrayList();
      }
  }

  public class ZooKeeper {
      static void Main(string[] args) {
          Zoo myZoo = new Zoo();
          myZoo.ZooAnimals.Add( new Lion() );
          myZoo.ZooAnimals.Add( new Elephant() );
          myZoo.ZooAnimals.Insert( 1, new Lion() );

          Animal a = myZoo[0];
          myZoo[1] = new Antelope();
      }
  }

  public abstract class Animal {
      abstract public void Eat();
  }

  public class Lion : Animal {
      public override void Eat() { }
  }

  public class Elephant : Animal {
      public override void Eat() { }
  }

  public class Antelope : Animal {
      public override void Eat() { }
  }
}
```

The preceding code is available on the Student Materials compact disc in the ArrayListExample.sln file in the Samples\Mod05\ArrayList folder.

How to Use Queues and Stacks



- **Queues: first-in, first-out**
  - Enqueue places objects in the queue
  - Dequeue removes objects from the queue
- **Stacks: last-in, first-out**
  - Push places objects on the stack
  - Pop removes objects from the stack
  - Count gets the number of objects contained in a stack or queue

Introduction

Queue objects and **Stack** objects are collection objects in the **System.Collections** namespace. The **Queue** class tracks objects on a first-in, first-out basis. The **Stack** class tracks objects on a first-in, last-out basis. By using the public methods of both **Queue** and **Stack** classes, you can move objects to different locations.

Using queues

The table below shows some of the public methods of the **Queue** class and a description of how they move objects:

Public methods	Description
Enqueue-Queue	Adds an object to the end of the **Queue**.
Dequeue-Queue	Removes and returns the object at the beginning of the **Queue**.

Example

The following example shows how to use a **Queue** object to handle messages. The **Messenger** class in this code calls the **SendMessage** method to send messages in the sequence “One”, “Two”, “Three”, and “Four”. The **Queue** object places the messages in the buffer by using the **Enqueue** method. When it is ready to receive the messages, it writes them to the console; in this case, by calling the **Dequeue** method.

```
using System;
using System.Collections;

namespace QueueExample {
  class Message {
      private string messageText;
      public Message (string s) {
          messageText = s;
      }
      public override string ToString() {
          return messageText;
      }
  }

  class Buffer {
      private Queue  messageBuffer;
      public void SendMessage( Message m ) {
          messageBuffer.Enqueue( m );
      }
      public void ReceiveMessage( ) {
          Message m = (Message) messageBuffer.Dequeue();
          Console.WriteLine( m.ToString() );
      }
      public Buffer() {
          messageBuffer = new Queue();
      }
  }

  class Messenger {
      static void Main(string[] args) {
          Buffer buf = new Buffer();
          buf.SendMessage( new Message("One") );
          buf.SendMessage( new Message("Two") );
          buf.ReceiveMessage ();
          buf.SendMessage( new Message("Three") );
          buf.ReceiveMessage ();
          buf.SendMessage( new Message("Four") );
          buf.ReceiveMessage ();
          buf.ReceiveMessage ();
      }
   }
}
```

The preceding code produces the following output:

```
One
Two
Three
Four
```

The preceding code is available on the Student Materials compact disc in the Queues.sln file in the Samples\Mod05\Queue folder.

Using stacks

The following table shows some of the public methods of the **Stack** class and a description of how they move objects to different locations.

Public methods	Description
Count	Gets the number of objects contained in a stack.
Push	Inserts an object at the top of the stack.
Pop	Removes and returns the object at the top of the stack.

Example

The following code uses a **Stack** object instead of a **Queue** object. Note that the messages are added by using the **Push** method and removed by using the **Pop** method.

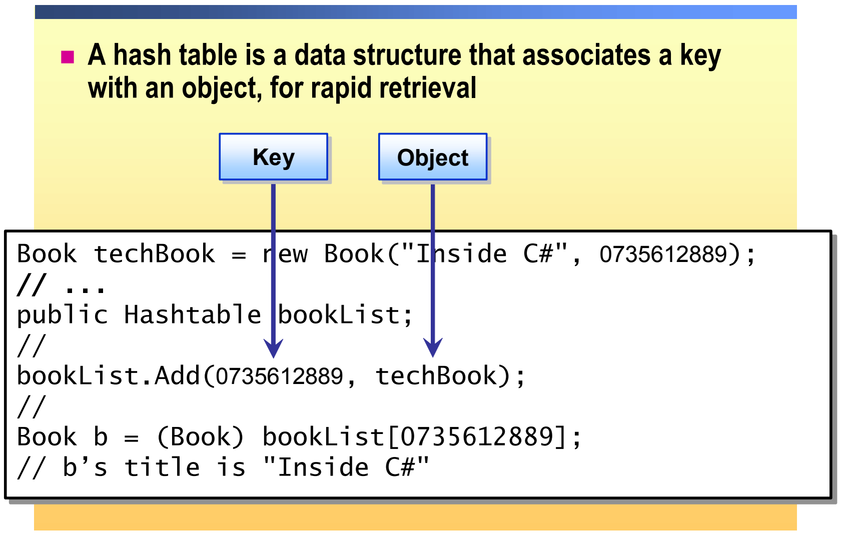
```
using System;
using System.Collections;

namespace StacksExample {
  class Message {
      private string messageText;
      public Message (string s) {
          messageText = s;
      }
      public override string ToString() {
          return messageText;
      }
  }

  class Buffer {
      private Stack  messageBuffer;
      public void SendMessage( Message m ) {
          messageBuffer.Push( m );
      }
      public void ReceiveMessage( ) {
          Message m = (Message) messageBuffer.Pop();
          Console.WriteLine( m.ToString() );
      }
      public Buffer() {
          messageBuffer = new Stack();
      }
  }

  class Messenger {
      static void Main(string[] args) {
          Buffer buf = new Buffer();
          buf.SendMessage( new Message("One") );
          buf.SendMessage( new Message("Two") );
          buf.ReceiveMessage ();
          buf.SendMessage( new Message("Three") );
          buf.ReceiveMessage ();
          buf.SendMessage( new Message("Four") );
          buf.ReceiveMessage ();
          buf.ReceiveMessage ();
      }
   }
}
```

The preceding code produces the following output:

```
Two
Three
Four
One
```

This code is available on the Student Materials compact disc in the Stacks.sln file in the Samples\Mod05\Stack folder.

How to Use Hash Tables



- A hash table is a data structure that associates a key with an object, for rapid retrieval

```
Book techBook = new Book("Inside C#", 0735612889);
// ...
public Hashtable bookList;
//
bookList.Add(0735612889, techBook);
//
Book b = (Book) bookList[0735612889];
// b's title is "Inside C#"
```

Introduction

A hash table is a data structure that is designed for fast retrieval. It does this by associating a key with each object that you store in the table. When you use this key to retrieve a value, the hash table can quickly locate the value.

Methods

Use a hash table when the data that you want to manage has some attribute that can act as the key. For example, if you are representing customers at a bank, you can use their taxpayer ID or their name. Or, a car rental company can use the rental agreement number as the key to the customer record.

In the .NET Framework, you can create hash tables by using the **Hashtable** class. After you have created a **Hashtable** object, you can use the **Add** method to add entries to it. The method takes two parameters, the key and the value, as shown in the following example:

```
myHashTable.Add( rentalAgreementNumber, someCustomerRecord );
```

You can retrieve objects from a **Hashtable** object by using the index operator and the key value, as shown in the following code.

```
Customer cr = (Customer) myHashTable[rentalAgreementNumber];
```

When you select the key value, choose one that is as short as possible and that will not change over the life of the object.

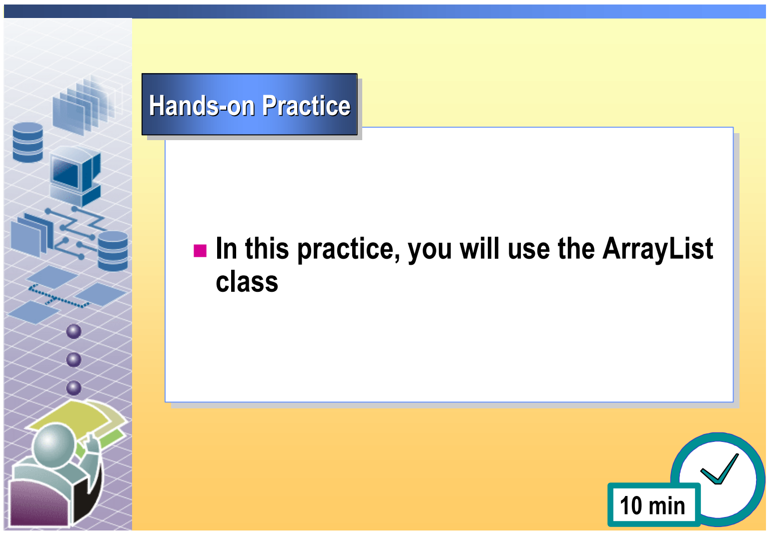
Example

The following example maintains a list of books by using the ISBN number as the key. Remember that the **Hashtable** class stores the data as type **Object**, so it is necessary to convert the retrieved object to the correct type before use.

```
using System;
using System.Collections;

namespace HashExample {

   // A library contains a list of books.
   class Library {
       public Hashtable bookList;
       public Library() {
           bookList = new Hashtable();
       }
   }

   // Books are placed in the library
   class Book {
       public Book( string t, int n) {
           Title = t; ISBN = n;
       }
       public string Title;
       public int ISBN;
   }

   class ClassMain {
       static void Main(string[] args) {
           Book b1 = new Book("Programming Microsoft Windows with C#", 0735613702 );
           Book b2 = new Book("Inside C#", 0735612889 );
           Book b3 = new Book("Microsoft C# Language Specifications", 0735614482 );
           Book b4 = new Book("Microsoft Visual C# .NET Lang. Ref.", 0735615543 );

           Library myReferences = new Library();
           myReferences.bookList.Add(b1.ISBN, b1);
           myReferences.bookList.Add(b2.ISBN, b2);
           myReferences.bookList.Add(b3.ISBN, b3);
           myReferences.bookList.Add(b4.ISBN, b4);

           Book b = (Book) myReferences.bookList[0735612889];
           Console.WriteLine( b.Title );
       }
   }
}
```

The preceding code is available on the Student Materials compact disc in the HashExample.sln file in the Samples\Mod05\Hashtable folder.

Practice: Creating and Using Collections


Hands-on Practice

- In this practice, you will use the ArrayList class

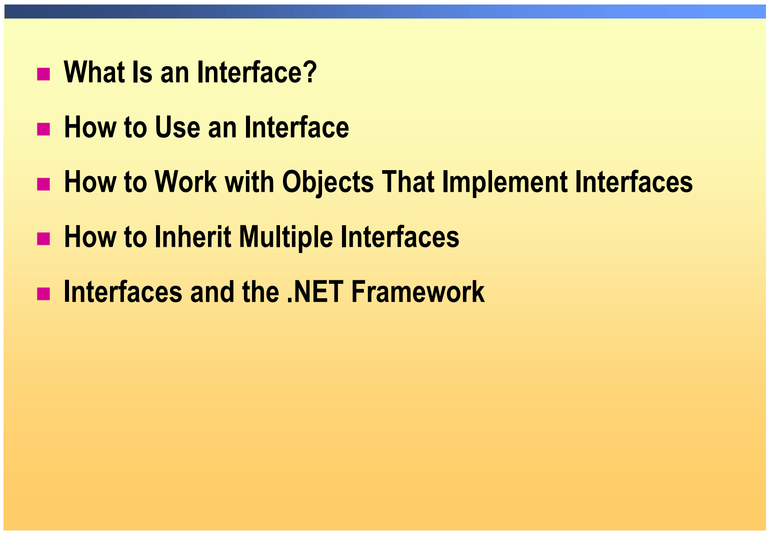
The **Zoo** class uses an object of type ArrayList to maintain a list of **Animal** objects. You will add missing code to initialize the list and to add animals to the list.

The solution for this practice is located in *install_folder*\Practices\Mod05\Collections_Solution \Collections.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio. NET, and then open *install_folder* \Practices\Mod05\Collections \Collections.sln.	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start Page**, click **Open Project**. **c.** In the **Open Project** dialog box, browse to *install_folder*\Practices \Mod05\Collections, click **Collections.sln**, and then click **Open**. **d.** In Solution Explorer, click **Form1.cs**, and then press F7 to open the Code Editor.
2. Review the Task List.	**a.** On the **View** menu, point to **Show Tasks**, and then click **All**. **b.** Review the tasks in the Task List.
3. Create the ArrayList.	**a.** Double-click the task **TODO: Instantiate the ArrayList**. **b.** Use the **new** keyword to create the ArrayList.

Tasks	Detailed steps
4. Ensure that the indexer returns an **Animal** type.	▪ Double-click the task **TODO: Fix the return type**, and return an **Animal** object from the **get** block in the indexer.
5. Implement the **Add** method in the **Zoo** class.	**a.** Double-click the task **TODO: Implement the Add method**. **b.** Write an **Add** method that adds an animal to the ArrayList.
6. Test your application.	▪ Press F5 to test your application. Fix any problems.
7. Quit Visual Studio .NET, saving your solution.	**a.** On the **File** menu, click **Save All**. **b.** On the **File** menu, click **Exit**.

Lesson: Using Interfaces



- What Is an Interface?
- How to Use an Interface
- How to Work with Objects That Implement Interfaces
- How to Inherit Multiple Interfaces
- Interfaces and the .NET Framework

Introduction

This lesson describes why interfaces are an important part of C# programming and explains how to implement interfaces in a C# application. This lesson explains how to work with objects that implement interfaces and how to implement multiple inheritances. How interfaces work in the .NET Framework is also covered in this lesson.

Lesson objectives

After completing this lesson, you will be able to:

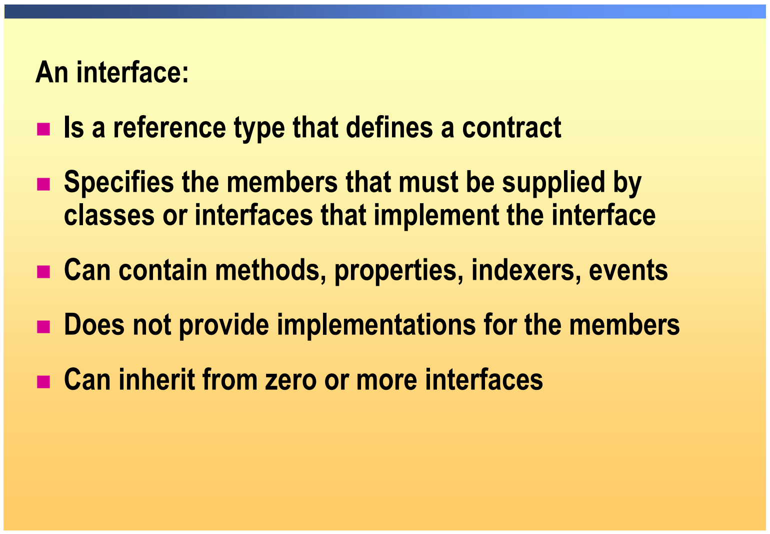
- Describe an interface.
- Work with objects that implement interfaces.
- Use interfaces to implement multiple inheritance.

Lesson agenda

This lesson includes the following topics and activity:

- What Is an Interface?
- How to Use an Interface
- How to Work with Objects That Implement Interfaces
- How to Inherit Multiple Interfaces
- Interfaces and the .NET Framework
- Practice: Using Interfaces

What Is an Interface?



An interface:

- Is a reference type that defines a contract
- Specifies the members that must be supplied by classes or interfaces that implement the interface
- Can contain methods, properties, indexers, events
- Does not provide implementations for the members
- Can inherit from zero or more interfaces

Introduction

An interface is in some ways like an abstract class. It defines a set of methods, properties, indexers, and events, just like any other class. However, it provides no implementation. The class that inherits the interface must provide the implementation.

Definition

Interface is a reference type that defines a contract in that a class that implements an interface must implement every aspect of that interface exactly as it is defined. Like classes, interfaces can contain methods, properties, indexers, and events as members. Other types implement an interface to guarantee that they support certain operations. The interface specifies the members that must be supplied by classes or other interfaces that implement it. Providing an implementation of the methods, properties, indexers, and events that are declared by the interface is called *implementing* the interface.

Although C# imposes a single inheritance rule for classes, the language is designed so that a class can inherit multiple interfaces.

Design considerations

A well-designed interface combines a set of closely related features that define a specific behavior. When an object uses this interface, it implements that behavior.

You can develop enhanced implementations for your interfaces without jeopardizing existing code, thus minimizing compatibility problems. You can also add new features at any time by developing additional interfaces and implementations.

Interface invariance

Although interface implementations can evolve, interfaces themselves cannot be changed after they are published. This is called *interface invariance*. Changes to a published interface may break existing code. When an interface needs enhancement, you must create a new interface.

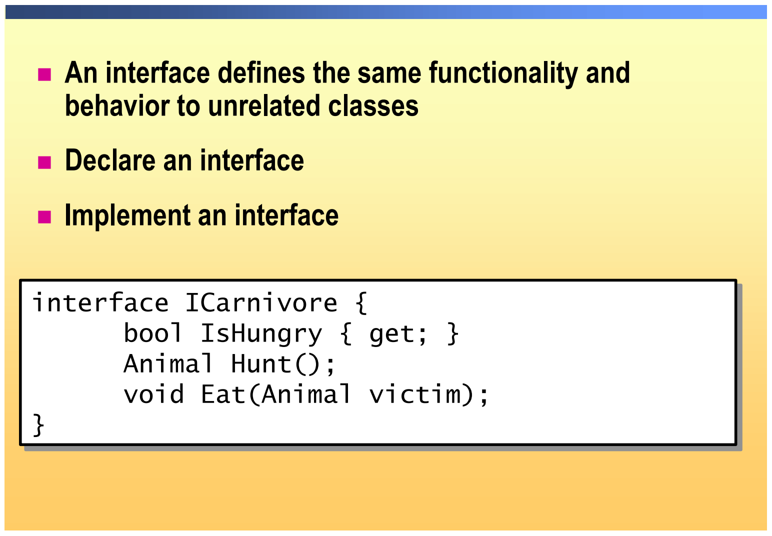
You are less likely to make errors when you design an interface than when you create a large inheritance tree. If you start with a small number of interfaces, you can have parts of a system running relatively quickly. The ability to evolve the system by adding interfaces allows you to gain the advantages that object-oriented programming is intended to provide.

Purpose

There are several reasons that you may want to use interfaces instead of class inheritance:

- Interfaces are better suited to situations in which your applications require many possibly unrelated object types to provide certain functionality.
- Interfaces permit polymorphism between classes with different base classes.
- Interfaces are more flexible than base classes because you can define a single implementation that can implement multiple interfaces.
- Interfaces are better in situations in which you do not need to inherit implementation from a base class.
- Interfaces are useful in cases where you cannot use class inheritance.

How to Use an Interface



- An interface defines the same functionality and behavior to unrelated classes
- Declare an interface
- Implement an interface

```
interface ICarnivore {
      bool IsHungry { get; }
      Animal Hunt();
      void Eat(Animal victim);
}
```

Introduction

Interfaces define a standard set of properties, methods, indexers, and events that are found in any class that implements the interface. As such, interfaces are useful when you want to maintain the same functionality and behavior in unrelated classes.

Declaring an interface

You use the **interface** keyword to declare an interface. The syntax is:

```
[attributes] [access-modifier] interface interface-name
[:base-list] { interface-body }
```

Example

The following example defines an interface named **ICarnivore** with one method and one property. The class that implements this interface must implement the **EatMeat** method and the **IsHungry** property. The compiler enforces this implementation.

```
interface ICarnivore {
  bool    IsHungry { get; }
  Animal  Hunt();
  void    Eat(Animal victim);
}
```

If you want the user of the interface to be able to set the **IsHungry** property, define it as follows:

```
bool IsHungry { get; set; }
```

Note Interface names normally start with an upper case **I**. The language does not enforce this, but it is a good practice.

Implementing an interface

To implement a specific interface, a class must inherit that interface. In the following example, the **Lion** class implements the **ICarnivore** interface.

```
public class Lion: ICarnivore {
  private bool hungry;
  public bool IsHungry {
      get {
          return hungry;
      }
  }

  public Animal Hunt() {
      // hunt and capture implementation
      // return animal object
  }

  public void Eat( Animal victim ) {
      // implementation
  }
}
```

The preceding code defines the **Lion** class as having the behaviors of the **ICarnivore** interface. The primary benefit of this approach is that any other object that can work with objects that implement the **ICarnivore** interface can now work with your object.

The class that implements an interface can be a derived class that includes some unique class members. The following example shows a **Lion** class that inherits the **Animal** class and implements the **ICarnivore** interface.

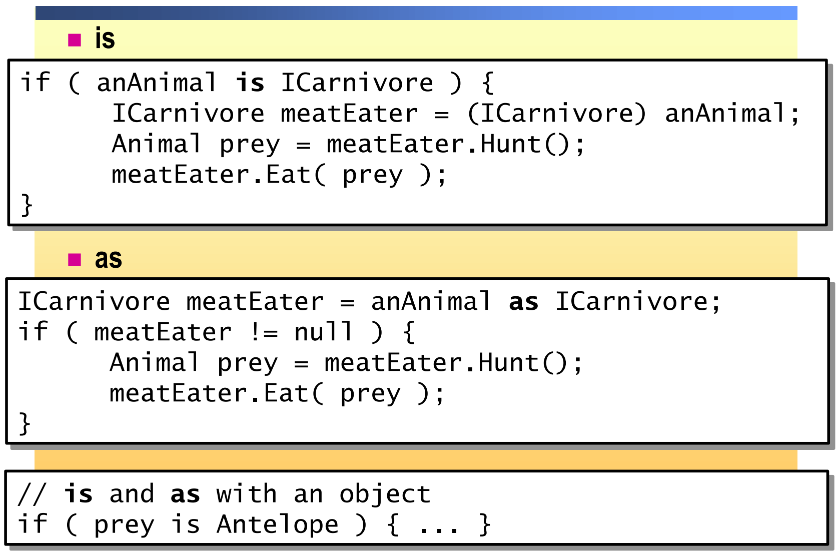
The following example is also provided on the Student Materials compact disc in the Carnivore.sln file in the Samples\Mod05\Interfaces folder.

```
Using System;

namespace LearningCSharp {

  interface ICarnivore {
      bool IsHungry { get; }
      Animal Hunt();
      void Eat(Animal victim);
  }

  public abstract class Animal {
      public abstract void Sleep();
  }

  public class Antelope: Animal {
      public override void Sleep() { }
  }

  public class Lion: Animal, ICarnivore {

      public Lion() {
          hungry = true;
      }

      // ICarnivore implementation
      private bool hungry;
      public bool IsHungry {
          get {
              return hungry;
          }
      }

      public Animal Hunt( ) {
          // hunt and capture implementation
          return new Antelope();
      }

      public void Eat( Animal prey ) {
          // implementation
          Console.WriteLine("Lion is no longer hungry");
      }

      // Inherited from base class
      public override void Sleep() {
          // sleeping
      }

      public void JoinPride() {
          // Join with a Pride of other Lions
      }
  }
```

Code continued on the following page.

```
    class Tester {
        static void Main(string[] args) {
            Lion aLion = new Lion();
            Antelope a = new Antelope();

            // carnivore-like behavior
            if ( aLion.IsHungry ) {
                Animal victim = aLion.Hunt();
                if ( victim != null ) {
                    aLion.Eat( victim );
                }
            }

            // Lion specific
            aLion.JoinPride();

            // Animal behavior
            aLion.Sleep();
        }
    }
}
```

How to Work with Objects That Implement Interfaces



- **is**

```
if ( anAnimal is ICarnivore ) {
      ICarnivore meatEater = (ICarnivore) anAnimal;
      Animal prey = meatEater.Hunt();
      meatEater.Eat( prey );
}
```

- **as**

```
ICarnivore meatEater = anAnimal as ICarnivore;
if ( meatEater != null ) {
      Animal prey = meatEater.Hunt();
      meatEater.Eat( prey );
}
```

```
// is and as with an object
if ( prey is Antelope ) { ... }
```

Introduction

At run time, your application may not know if an object implements a certain interface. You can use the **is** and **as** keywords to determine whether an object implements a specific interface.

Example

For example, you may want to know whether the objects in the preceding examples implement the **ICarnivore** or **IHerbivore** interface, so that you can call only the **GatherFood** method for the animals that are herbivores:

```
interface ICarnivore {
    bool IsHungry { get; }
    Animal Hunt();
    void Eat(Animal victim);
}
interface IHerbivore {
    bool Hungry { get; }
    void GatherFood();
}

public class Chimpanzee: Animal, IHerbivore, ICarnivore {
    //todo: implement members of IHerbivore and ICarnivore
}
public class Lion: Animal, ICarnivore {
    //todo: implement members of ICarnivore
}
public class Antelope: Animal, IHerbivore {
    //todo: implement members of IHerbivore
}
public class elephant: Animal, IHerbivore {
    //todo: implement members of IHerbivore
}
```

Suppose that you have an ArrayList **zoo** that contains objects that are derived from the **Animal** class, some of which are **Lion**, which implements **ICarnivore**, **Antelope**, which implements **IHerbivore** and others such as **Elephant**. To discover if the animal implements **IHerbivore**, use the **is** keyword.

After you determine that the object implements the interface, you must obtain a reference to the interface. To obtain a reference to the interface, you can cast to the interface, as shown in the following example, where **someAnimal** is cast to type **IHerbivore** as it is assigned to **veggie**.

```
foreach ( Animal someAnimal in zoo ) {
  if ( someAnimal is IHerbivore ) {
      IHerbivore veggie = (IHerbivore) someAnimal;
      veggie.GatherFood();
  }
}
```

Note that when the application tries to perform a type cast, it checks to make sure that it will succeed. In the preceding example, checking is performed twice, because the **is** operator also checks the type of **someAnimal**. Because this is a fairly common situation, C# provides a way to avoid the double check, by using the **as** operator.

The **as** operator combines the check with the type cast, allowing you to rewrite the preceding code as follows:

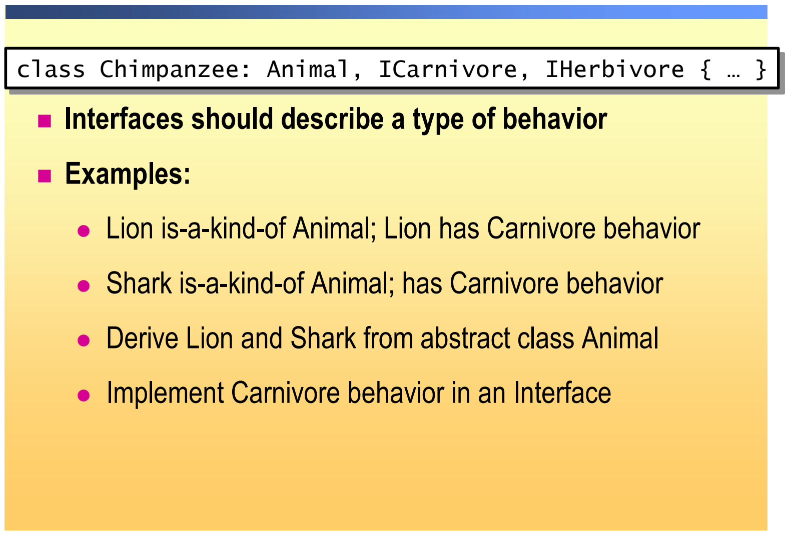
```
foreach ( Animal someAnimal in zoo ) {
  IHerbivore veggie = someAnimal as IHerbivore;
  if ( veggie != null ) {
      veggie.EatPlant();
  }
}
```

Using is and as with other types

Note that the **is** and **as** operators work with other types. You can use them to determine the type of a class at run time. For example, the **Eat** method can be rewritten as follows:

```
public void Eat( Animal prey ) {
  // implementation
  if ( prey is Antelope ) {
      Console.WriteLine("Favorite meal");
  }
  Console.WriteLine("Lion is no longer hungry");
}
```

How to Inherit Multiple Interfaces



```
class Chimpanzee: Animal, ICarnivore, IHerbivore { … }
```

- Interfaces should describe a type of behavior
- Examples:
  - Lion is-a-kind-of Animal; Lion has Carnivore behavior
  - Shark is-a-kind-of Animal; has Carnivore behavior
  - Derive Lion and Shark from abstract class Animal
  - Implement Carnivore behavior in an Interface

Introduction

A class can inherit multiple interfaces. Interfaces can also inherit from one or more interfaces.

Inheriting multiple interfaces

To implement multiple interface inheritance, you list the interfaces in a comma-separated list, as shown in the following example:

```
class Chimpanzee: Animal, ICarnivore, IHerbivore { … }
```

The **Chimpanzee** class must provide the implementation of all of the members of **ICarnivore** and of **IHerbivore**, as shown in the following code:

```
interface ICarnivore {
  bool IsHungry { get; }
  Animal Hunt();
  void Eat(Animal victim);
}

interface IHerbivore {
    bool IsHungry { get; }
    void GatherFood();
}
```

Interfaces inheriting interfaces

An interface can inherit from other interfaces. Unlike classes, interfaces can inherit more than one other interface. To inherit more than one interface, the interface identifier is followed by a colon and a comma-separated list of base interface identifiers. An interface inherits all members of its base interfaces and the user of the interface must implement all the members of all the inherited interfaces.

For example, if an **IOmnivore** interface inherits both **ICarnivore** and **IHerbivore**, any class that implements **IOmnivore** must write an implementation of **IsHungry**, **Hunt()**, **Eat(Animal)**, and **GatherFood()**, as shown in the following code:

```
interface ICarnivore {
  bool IsHungry { get; }
  Animal Hunt();
  void Eat(Animal victim);
}
interface IHerbivore {
  bool IsHungry { get; }
  void GatherFood();
}

interface IOmnivore: IHerbivore, ICarnivore {
}
```

You can extend the **IOmnivore** interface by adding another member, as shown in the following code:

```
interface IOmnivore: IHerbivore, ICarnivore {
  void DecideWhereToGoForDinner();
}
```

Explicit interface implementation

In the preceding example, it is not possible to determine whether the **IsHungry** property in the **Chimpanzee** class implements **IHerbivore.IsHungry** or **ICarnivore.IsHungry**. To make this implementation explicit, you must declare the interface in the declaration, as shown in the following example:

```
public class Chimpanzee: Animal, IHerbivore, ICarnivore {
...// implement other interface members
  bool ICarnivore.IsHungry {
      get {
          return false;
      }
  }

  bool IHerbivore.IsHungry {
      get {
          return false;
      }
  }
}
```

Access modifiers are not permitted on explicit interface implementations, so to access these members you must convert the object to the interface type, as shown in the following code:

```
Chimpanzee chimp = new Chimpanzee();
IHerbivore vchimp = (IHerbivore) chimp;
bool hungry = vchimp.IsHungry;
```

Interfaces vs. abstract classes

Deciding whether to design your functionality as an interface or an abstract class is sometimes difficult. An abstract class is a class that cannot be instantiated, but must be inherited from. An abstract class may be fully implemented, but is more usually partially implemented or not implemented at all, thereby encapsulating common functionality for inherited classes.

An interface, by contrast, is a totally abstract set of members that can be thought of as defining a contract for conduct. The implementation of an interface is left completely to the developer.

Abstract classes provide a simple and easy way to manage versions of your components. By updating the base class, all inheriting classes are automatically updated with the change. Interfaces, on the other hand, cannot be changed after they are created. If an interface needs revisions, you must create a new interface.

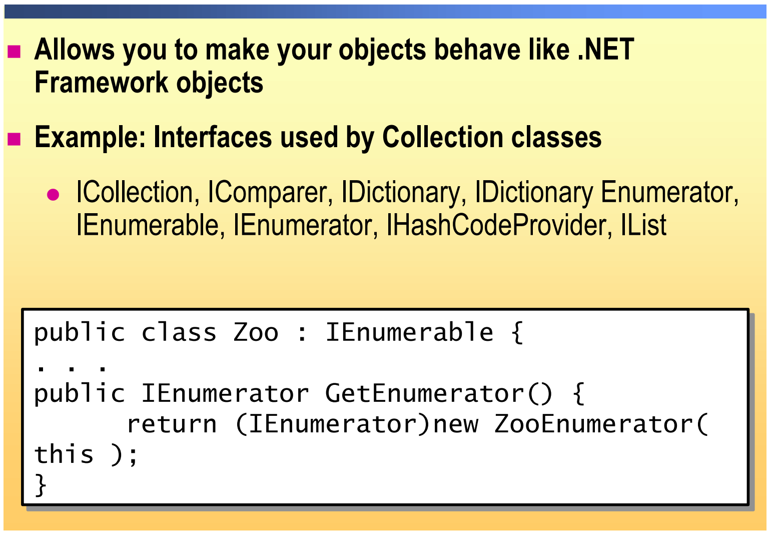
Recommendations

The following recommendations suggest when to use an interface or an abstract class to provide polymorphism for your components:

When you:	Use:
Create multiple versions of your component	Abstract class
Create a function that is useful across a wide range of disparate objects	Interface
Design small, concise bits of functionality	Interface
Design large functional units	Abstract class

Note Abstract classes should be used primarily for objects that are closely related, whereas interfaces are best suited for providing common functionality to unrelated classes.

Interfaces and the .NET Framework



- Allows you to make your objects behave like .NET Framework objects
- Example: Interfaces used by Collection classes
  - ICollection, IComparer, IDictionary, IDictionary Enumerator, IEnumerable, IEnumerator, IHashCodeProvider, IList

```
public class Zoo : IEnumerable {
. . .
public IEnumerator GetEnumerator() {
      return (IEnumerator)new ZooEnumerator(
this );
}
```

Introduction

Interfaces are used in many places in the .NET Framework. You can enhance the usefulness of classes that you develop by implementing appropriate interfaces.

One area of the .NET Framework that makes extensive use of interfaces is the set of classes in the **System.Collections** namespace.

Definitions

Collection classes use the following interfaces:

ICollection	Defines size, enumerators, and synchronization methods for all collections.
IComparer	Exposes a method that compares two objects.
IDictionary	Represents key-value pairs, as used by hash tables.
IDictionaryEnumerator	Enumerates the elements in a dictionary (a hashtable is a dictionary).
IEnumerable	Exposes the enumerator, for iteration over a collection.
IEnumerator	Supports iteration over a collection. **IEnumerator** is the base interface for all enumerators. Enumerators only allow reading the data in the collection. Enumerators cannot be used to modify the underlying collection.
IHashCodeProvider	Defines a method for getting a hash code.
IList	Supports array-like indexing.
ICloneable	Supports cloning, which creates a new instance of a class with the same value as an existing instance.

Collection requirements

By implementing specific interfaces, you can make your objects behave like collection objects, as shown in the following two examples.

Example 1

Suppose that you have a class that maintains a set of objects as follows:

```
public class Zoo {
  private int insertPosition = 0;
  private Animal[] animals;

  public Zoo() {
      animals = new Animal[100];
  }

  public void Add(Animal a) {
      if ( insertPosition >= 100 ) {
          return;
      }
      animals[insertPosition++] = a;
  }
}
```

The user of the class may find it useful to use the **foreach** statement to iterate through the elements of your class.

To iterate through a collection, a class (or struct or interface) must implement the **IEnumerable** interface. The **IEnumerator** interface contains one instance method named **GetEnumerator** that returns an object that implements **IEnumerator**.

A class that implements **IEnumerator** must contain:

- A property named **Current** that returns the current element of the collection.
- A **bool** method named **MoveNext** that increments an item counter and returns **true** if there are more items in the collection.
- A void method named **Reset** that resets the item counter.

Example 2

In this example, animals are added to a **Zoo** object. The **Zoo** object implements the **IEnumerable** interface:

```
public class Zoo : IEnumerable {
. . .
public IEnumerator GetEnumerator() {
  return (IEnumerator) new ZooEnumerator( this );
}
```

The **GetEnumerator** method returns an instance of **ZooEnumerator**. The **ZooEnumerator** class provides the specific **IEnumerator** implementation for moving through the data structure that contains the animals in the zoo. Although this data structure is an array, the programmer can use any data structure to maintain the **Animal** objects.

Because the **ZooEnumerator** class is so closely linked to the **Zoo** class, it is declared in the **Zoo** class. The full code sample is included at the end of this topic.

```
private class ZooEnumerator : IEnumerator {
  private Zoo z;
  private int currentPosition = -1;

  public ZooEnumerator(Zoo aZoo) {
      z = aZoo;
  }
. . .
}
```

The constructor simply saves the reference to the **Zoo** object **aZoo** that is passed in as a parameter. Note that **currentPosition** is set to **-1**. This is because the **MoveNext** method will increment this to zero and then check to see if this is a valid position in the data.

```
public bool MoveNext() {
  ++currentPosition;
  if ( currentPosition < z.insertPosition ) {
      return true;
  }
  return false;
}
```

z.insertPosition is the first unassigned position. The code returns **false** if the current position moves beyond this point, and this will cause the **foreach** loop that is using your **Zoo** class collection to exit.

The **Current** property is used to retrieve the current element:

```
public object Current {
  get {
      return z.animals[currentPosition];
  }
}
```

And the **Reset** method simply resets **currentPosition**:

```
public void Reset() {
  currentPosition = -1;
}
```

The preceding code implements the **IEnumerable** and **IEnumerator** interfaces, and it is now possible to iterate through **Zoo** with the **foreach** statement:

```
Zoo z = new Zoo();
z.Add( new Antelope() );
z.Add( new Elephant() );
z.Add( new Antelope() );

foreach ( Animal a in z ) {
  a.Sleep();
  Console.WriteLine( a.ToString() );
}
```

The complete code follows:

```
using System;
using System.Collections;

namespace LearningCSharp {
  public abstract class Animal {
      public abstract void Sleep();
  }

  public class Elephant : Animal {
      public override void Sleep() { }
  }

  public class Antelope : Animal {
      public override void Sleep() { }
  }

  public class Zoo : IEnumerable {
      private int insertPosition = 0;
      private Animal[] animals;

      // Constructor
      public Zoo() {
          animals = new Animal[100];
      }

      // public methods
      public void Add(Animal a) {
          if ( insertPosition >= 100 ) {
              return;
          }
          animals[insertPosition++] = a;
      }

      // IEnumerable
      public IEnumerator GetEnumerator() {
          return (IEnumerator)new ZooEnumerator( this );
      }

      // ZooEnumerator as private class
      private class ZooEnumerator : IEnumerator {
          private Zoo z;
          private int currentPosition = -1;
```

Code continued on the following page.

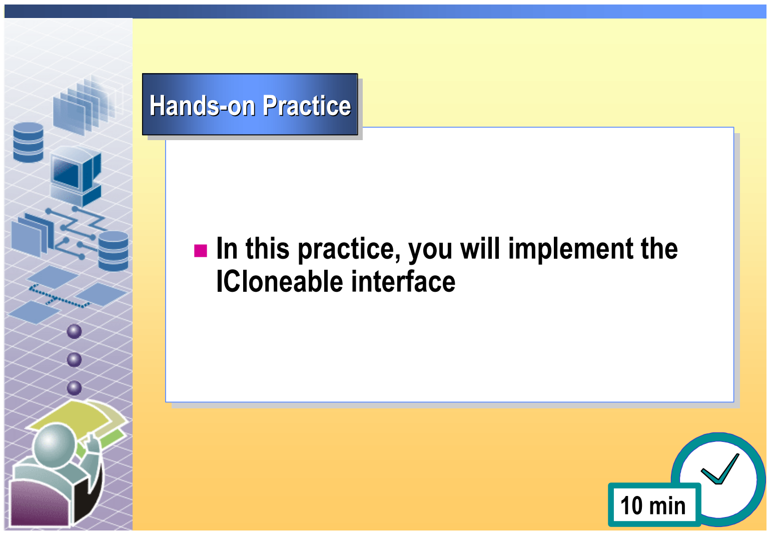
```
            public ZooEnumerator(Zoo aZoo) {
                z = aZoo;
            }

            // IEnumerator
            public object Current {
                get {
                    return z.animals[currentPosition];
                }
            }
            public bool MoveNext() {
                ++currentPosition;
                if ( currentPosition < z.insertPosition ) {
                    return true;
                }
                return false;
            }
            public void Reset() {
                currentPosition = -1;
            }
        }
    }

    class Tester {
        static void Main(string[] args) {
            Zoo z = new Zoo();
            z.Add( new Antelope() );
            z.Add( new Elephant() );
            z.Add( new Antelope() );

            foreach ( Animal a in z ) {
                a.Sleep();
                Console.WriteLine(a.ToString());
            }
        }
    }
}
```

This preceding code produces the following output:

```
LearningCSharp.Antelope
LearningCSharp.Elephant
LearningCSharp.Antelope
```

This code is available on the Student Materials compact disc in the Interfaces2.sln file in the Samples\Mod05\Interfaces2 folder.

Practice: Using Interfaces


Hands-on Practice

- In this practice, you will implement the ICloneable interface

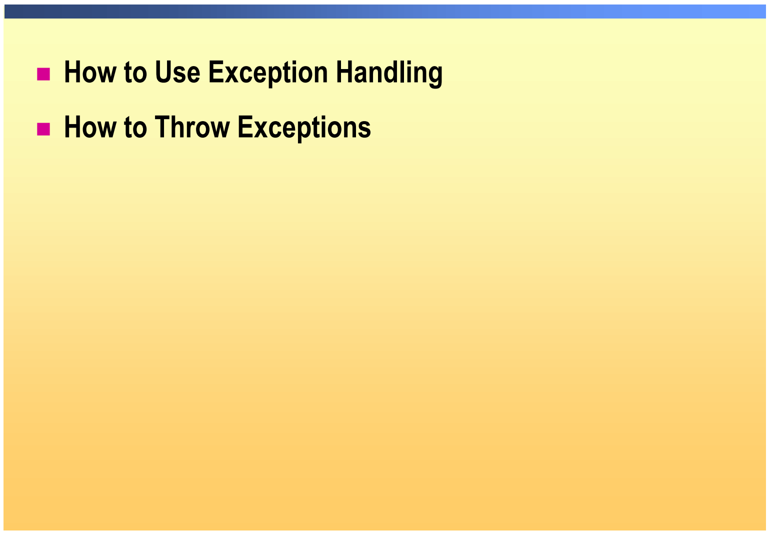
In this practice, you will add the **ICloneable** interface to the **Zoo** class. The **ICloneable** interface consists of one method: **object Clone()**. The Clone method creates and returns a copy of an object.

The solution for this practice is located in *install_folder*\Practices\Mod05\Interfaces_Solution \Interfaces.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open *install_folder*\Practices \Mod05\Interfaces\Interfaces.sln.	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start Page**, click **Open Project**. **c.** In the **Open Project** dialog box, browse to *install_folder* \Practices\Mod05\Interfaces, click **Interfaces.sln**, and then click **Open**. **d.** In Solution Explorer, click **Form1.cs**, and then press F7 to open the Code Editor.
2. Review the Task List.	**a.** On the **View** menu, point to **Show Tasks**, and then click **All**. **b.** Review the tasks in the Task List.
3. Add the **ICloneable** interface reference to the class. Compile your application and read the error message.	**a.** Double-click **TODO: State that this class implements the ICloneable Interface**. **b.** Add the **ICloneable** interface to the class definition. **c.** Press F5 to test your application. Read the error message. **d.** On the **View** menu, point to **Show Tasks**, and then click **All**.

Tasks	Detailed steps
4. Add the **Clone** method to the **Zoo** class.	**a.** Double-click **TODO: Implement ICloneable**. **b.** Remove the comments from the **Clone** method. **c.** Note that the **Clone** method creates and returns a new **Zoo** object.
5. Test your implementation of **ICloneable** by declaring a new **Zoo** object **zooTwo**, and cloning **myZoo.**	**a.** Double-click **TODO: Test ICloneable**. **b.** Uncomment the call to **Clone**, and to **DisplayZooInformation**. **c.** Press F5 to test your application. Fix any problems.
6. Quit Visual Studio .NET, saving your solution.	**a.** On the **File** menu, click **Save All**. **b.** On the **File** menu, click **Exit**.

Lesson: Using Exception Handling



- How to Use Exception Handling
- How to Throw Exceptions

Introduction

This lesson describes error handling using exception handlers, including user exceptions and basic exception handling syntax, such as **try**, **catch**, and **finally**. It also explains how to use exception types. The lesson also covers throwing exceptions by using the **throw** keyword.

Lesson objectives

After completing this lesson, you will be able to:

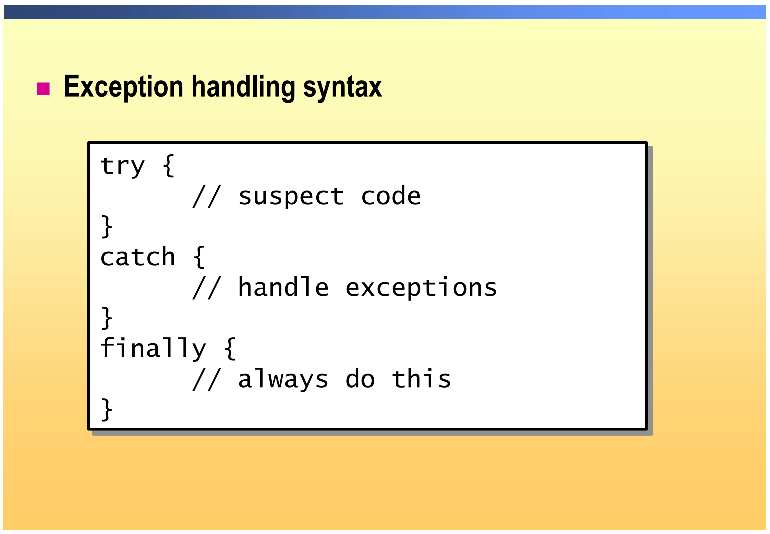
- Explain exception handling.
- Use the **throw** keyword.
- Use the **try**, **catch**, and **finally** keywords.
- Catch specific exception types.

Lesson agenda

This lesson includes the following topics and activity:

- How to Use Exception Handling
- How to Throw Exceptions
- Practice: Using Exception Handling

How to Use Exception Handling



- **Exception handling syntax**

```
try {
      // suspect code
}
catch {
      // handle exceptions
}
finally {
      // always do this
}
```

Introduction

An *exception* is any error condition or unexpected behavior that is encountered by an executing program. Exceptions can be raised because of a fault in your code or in code that you call, resources not being available, such as running out of memory or not being able to locate a particular file, or other unexpected conditions.

You should, of course, attempt to eliminate all bugs in your code. Exceptions are not designed to handle programming errors; they are designed to provide control in situations where there is a true exception to the expected behavior of the application. When possible, make every effort to avoid a condition that could throw an exception. Under normal circumstances, your application should not encounter any exceptions.

Note The code samples in this topic are intended to illustrate exception handling. As such, these samples do not try to anticipate and avoid error conditions.

Using try/catch to catch exceptions

The following examples represent situations that might cause exceptions:

Scenario	Solution
Opening a file	Before opening a file, check that the file exists and that you can open it.
Reading an XML document	You will normally read well-formed XML documents, but you should deal with the exceptional case where the document is not valid XML. This is a good example of where to rely on exception handling.
Accessing an invalid member of an array	If you are the user of the array, this is an application bug that you should eliminate. Therefore, you should not use exception handling to catch this exception.
Dividing a number by zero	This can normally be checked and avoided.
Converting between types using the **System.Convert** classes	With some checking, these can be avoided.

You can handle these error conditions by using **try**, **catch**, and **finally** keywords.

To write an exception handler, place the sections of code that may throw exceptions in a **try** block, and then place code that handles exceptions in a **catch** block. The **catch** block is a series of statements that begin with the **catch** keyword, followed by an exception type and an action to be taken.

Example

The following code shows a Microsoft Windows® text box named **numberOfTickets** into which the user types the number of tickets that they want to purchase. Text boxes provide access to their contents through the **Text** property, which is a **string** type, so the user input must be converted to an integer. The following code uses a byte to hold the number of tickets:

```
numberOfTickets = new TextBox();
...
byte tickets = Convert.ToByte(numberOfTickets.Text);
```

The **Convert.ToByte** method throws an exception, **System.FormatException** if the user enters a character string. The following code demonstrates how to handle this situation:

```
try {
  byte tickets = Convert.ToByte(numberOfTickets.Text);
}
catch {
  MessageBox.Show("Please enter a number");
}
```

The **try** block encloses the code that may throw an exception. The **catch** block catches all exceptions that are thrown in the **try** block, because **catch** does not specify the exception that it will handle.

Handling specific exceptions

You can also specify the type of the exception that you want to catch, as shown in the following code:

```
try {
  byte tickets = Convert.ToByte(numberOfTickets.Text);
}
catch (FormatException) {
  MessageBox.Show("Format Exception: please enter a number");
}
```

When the user enters text instead of numbers, a message box appears containing an appropriate message.

Multiple catch blocks

Suppose that the user wants to buy 400 tickets. This quantity exceeds the capacity of the byte, so a **System.OverflowException** is thrown. To handle the **OverflowException** and the **FormatException**, you must specify more than one **catch** block, as shown in the following example:

```
try {
  byte tickets = Convert.ToByte(numberOfTickets.Text);
}
catch (FormatException e) {
  MessageBox.Show("Format Exception: please enter a number");
}
catch (OverflowException e) {
  MessageBox.Show("Overflow: too many tickets");
}
```

When the user tries to purchase 400 tickets, the overflow message is displayed. When the user enters text, the format exception message is displayed.

Planning the catch sequence

The order in which the exceptions are listed is significant. For example, **System.DivideByZeroException** is derived from **System.ArithmeticException**. If you try to catch an ArithmeticException before a DivideByZeroException, the DivideByZeroException is not caught because the DivideByZeroException is a type of ArithmeticException—that is, it derived from ArithmeticException. Fortunately, the C# compiler checks for this situation and provides a warning message if you try to catch exceptions in an order that does not work.

Although these examples simply inform the user that the exception has occurred, in a real application you should make some attempt to correct the situation that caused the error. However, you should also keep the code in the **catch** block as small as possible to avoid the possibility of throwing another exception while handling the first.

Using the finally keyword

When an exception occurs, execution stops and control is given to the closest exception handler. This often means that lines of code that you expect to always be called are not executed. However, some resource cleanup, such as closing a file, must always be executed even if an exception is thrown. To accomplish this, you can use a **finally** block. A **finally** block is always executed, regardless of whether an exception is thrown.

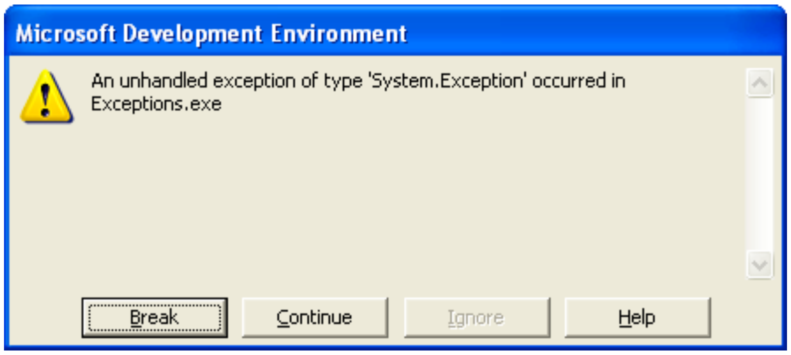
```
FileStream xmlFile = null;
try {
  xmlFile = new FileStream("XmlFile.xml", FileMode.Open);
}
catch( System.IO.IOException e ) {
  return;
}
finally {
  if ( xmlFile != null ) {
      xmlFile.Close();
  }
}
```

In this example, the **finally** block is used to close the file, if it was open.

Scope of exception handlers

When an exception occurs, as shown in the following code, where **ReadSetupFile** does not have a **try/catch** block in the method, the application looks up through the stack for the first exception handler that can handle the current exception type. If you do not provide any exception handler, the runtime provides a handler as in the following code:

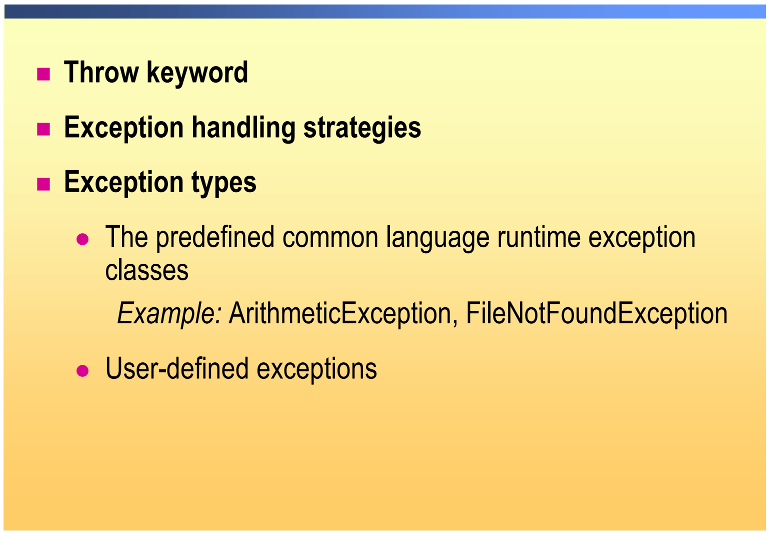
```
try {
      zoo.ReadSetupFile();
}
catch {
      // error handling
}
```


Microsoft Development Environment

An unhandled exception of type 'System.Exception' occurred in Exceptions.exe

You may have seen this handler if any of your programs have ever thrown an exception.

How to Throw Exceptions



- Throw keyword
- Exception handling strategies
- Exception types
  - The predefined common language runtime exception classes

    Example: ArithmeticException, FileNotFoundException
  - User-defined exceptions

Introduction

Sometimes you may want to catch an exception, do some work to handle the exception, and then pass an exception on to the calling code. This is called *throwing an exception*. It is good coding practice to add information to an exception that is rethrown to provide more information when debugging.

You use the **throw** keyword to throw an exception, as shown in the following code:

```
Exception e = new Exception();
throw e;
```

The preceding code is equivalent to:

```
throw new Exception();
```

Exception handling strategies

There are several strategies for handling exceptions:

- You can decide to ignore an exception, relying instead on the caller to handle the exception. This strategy can leave your object in an incorrect state, but it is sometimes necessary.
- You can catch the exception, try to fix the error, ensuring that at least your object is in a known state, and then rethrow the exception. This strategy has the advantage of leaving your object in a usable state, but it does not give the caller much useful information.
- You can catch the exception and add information to it by throwing a new exception that wraps around the old exception. This strategy is preferable because your object can then provide additional information about the error.

System.Exception

Exceptions inherit from the **System.Exception** base class. There are two important classes that are derived from **System.Exception**:

- **System.SystemException** is the base class for exceptions that are defined by the system.
- **System.ApplicationException** is the base class for exceptions that are defined by applications.

Example 1

This example throws an exception if the contents of a text box cannot be converted to a number, and catches **FormatException**. If the problem occurred because the user did not type a value, it creates a new exception that wraps up the **FormatException**, and adds information.

```
private int ReadData() {
  byte tickets = 0;
  try {
      tickets = Convert.ToByte(textBox1.Text);
  }
  catch ( FormatException e ) {
      if ( textBox1.Text.Length == 0 ) {
          throw (new FormatException("No user input ", e));
      }
      else {
          throw e;
      }
  }
  return tickets;
}
```

In a real application, you check the user input rather than throwing an exception.

The user of the **ReadData** method can catch the new **FormatException** and retrieve the additional information provided by the object that threw the exception, as shown in the following code:

```
private void run_Click(object sender, System.EventArgs ea) {
  int tickets = 0;
  try {
      tickets = ReadData();
  }
  catch ( FormatException e ) {
      MessageBox.Show( e.Message + "\n" );
      MessageBox.Show( e.InnerException.Message );
  }
}
```

When the exception is caught by the **run_Click** method, **e** references the new **FormatException** object thrown in **ReadData**, so the value of **e.Message** is **No user input**. The property **e.InnerException** refers to the original exception, so the value of **e.InnerException.Message** is **Input string was not in a correct format**, which is the default message for **FormatException**.

This code sample is provided on the Student Materials compact disc in the Exception.sln file in the Samples\Mod05\Exception folder.

Throwing user-defined exceptions

You can create your own exception classes by deriving from the **Application.Exception** class. When creating your own exceptions, it is good coding practice to end the class name of the user-defined exception with the word “Exception”.

Example 2

This example defines a new class named **TicketException** that is thrown when there is an error in the ticket ordering process. **TicketException** is derived from **ApplicationException**.

```
class TicketException: ApplicationException {
  private bool purchasedCompleted = false;
  public bool PurchaseWasCompleted {
      get {
          return purchasedCompleted;
      }
  }
  public TicketException( bool completed, Exception e )
                         : base ("Ticket Purchase Error", e ){
      purchasedCompleted = completed;
  }
}
```

The **ReadData** and run_**Click** methods can use **TicketException**:

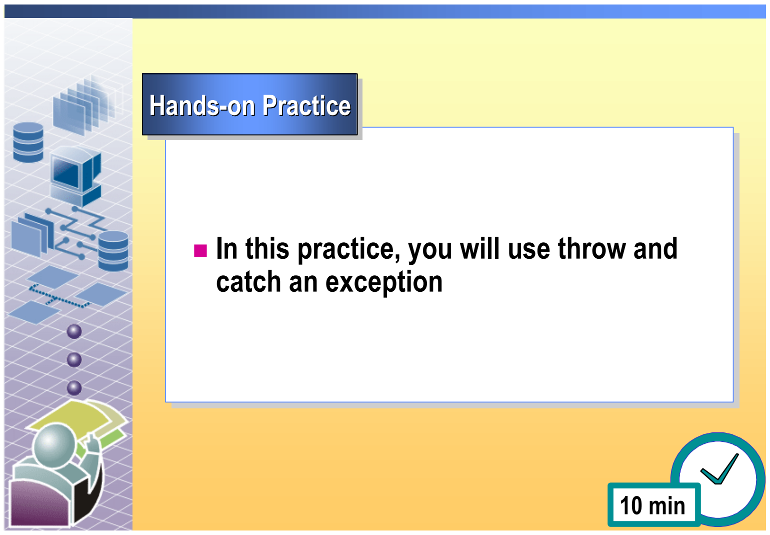
```
private int ReadData() {
  byte tickets = 0;
  try {
      tickets = Convert.ToByte(textBox1.Text);
  }
  catch ( Exception e ) {
      // check if purchase was complete
      throw ( new TicketException( true, e ) );
  }
  return tickets;
}

private void run_Click(object sender, System.EventArgs ea) {
  int tickets = 0;
  try {
      tickets = ReadData();
  }
  catch ( TicketException e ) {
      MessageBox.Show( e.Message );
      MessageBox.Show( e.InnerException.Message );
  }
}
```

Typically, you should overload the constructor so that the user can provide more detailed information about the error.

When an error occurs, this sample produces the **Ticket Purchase Error** message, and also provides access to the original exception through the **InnerException** property.

This code sample is provided on the Student Materials compact disc in the UserException.sln file in the Samples\Mod05\UserException folder.

Practice: Using Exception Handling


Hands-on Practice

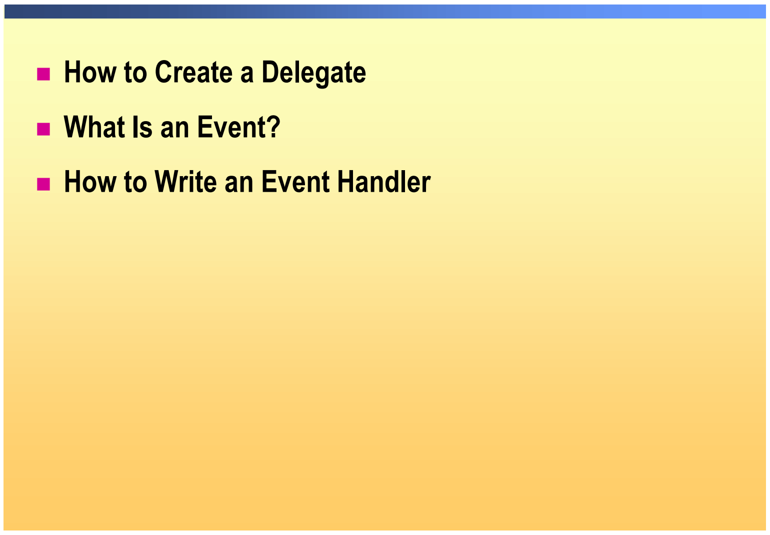
- In this practice, you will use throw and catch an exception

In this practice, you will use exception handling to prevent a run-time error from occurring when you attempt to access an invalid index of the **Zoo** class.

The solution for this practice is located in *install_folder*\Practices\Mod05\Exceptions_Solution \Exceptions.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open *install_folder* \Practices\Mod05\Exceptions \Exceptions.sln.	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start Page**, click **Open Project**. **c.** In the **Open Project** dialog box, browse to *install_folder*\Practices \Mod05\Exceptions, click **Exceptions.sln**, and then click **Open**. **d.** In Solution Explorer, click **Form1.cs**, and then press F7 to open the Code Editor.
2. Cause a run-time error by attempting to access an animal in the zoo that does not exist.	**a.** Double-click the task **TODO: catch the error thrown in this assignment.** The assignment **from myZoo[10]** throws an exception because the **myZoo** only has 3 elements. **b.** Press F5 to compile and run your application, and note the error message that results. **c.** You will receive a **System.ArgumentOutOfRangeException** exception, along with some explanatory text. *In the **Microsoft Development Environment** dialog box, you can click **Break** to debug your application. When you click **Break**, the line that threw the exception is displayed in the code window (the Zoo Indexer). To see the code that called the Indexer, in the **Call Stack** window, double-click the line immediately below the line that is highlighted with the green arrow. To access the **Call Stack** window, on the **Debug** menu, point to **Windows** and then click **Call Stack**, or press CTRL+ALT+C.*
3. Use a **try...catch** block to catch the ArgumentOutOfRangeException and use the following code to display your own message: `MessageBox.Show("Zoo access error");`	**a.** Enclose the **myZoo[10]** statement in a **try** block. **b.** Write a **catch** block that catches an ArgumentOutOfRangeException exception. **c.** In the **catch** block, use the following code to display a message box: `MessageBox.Show("Zoo access error");`
4. Test the application.	▪ Press F5 to test your application. Fix any problems.
5. Quit Visual Studio .NET, saving your solution.	**a.** On the **File** menu, click **Save All**. **b.** On the **File** menu, click **Exit**.

Lesson: Using Delegates and Events


- How to Create a Delegate
- What Is an Event?
- How to Write an Event Handler

Introduction

This lesson introduces delegates, describes the purpose and syntax of delegates, and explains how to create and use a delegate function. This lesson also introduces event handling and explains how events are handled in C#.

Lesson objectives

After completing this lesson, you will be able to:

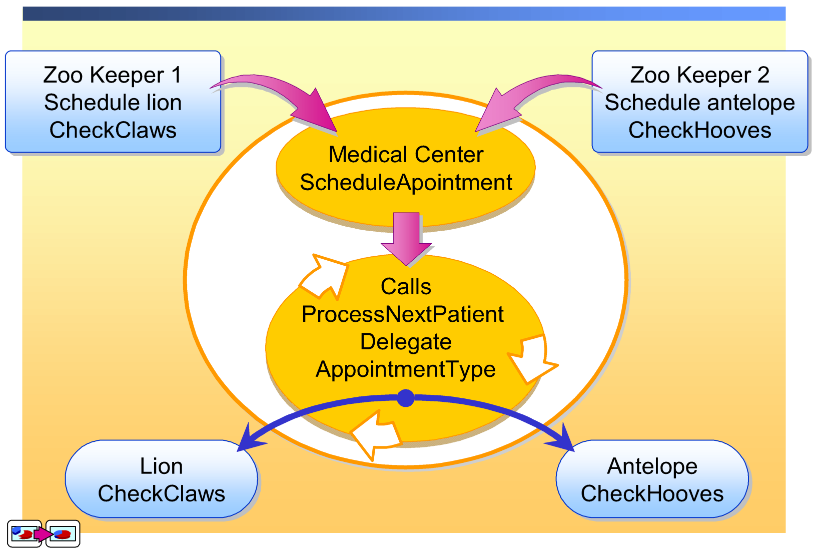
- Describe a delegate.
- Create a delegate.
- Use a delegate.
- Describe an event.
- Write an event handler.

Lesson agenda

This lesson includes the following topics and activity:

- How to Create a Delegate
- What Is an Event?
- How to Write an Event Handler
- Practice: Declaring and Calling a Delegate

How to Create a Delegate



Introduction

A delegate is an object that contains a reference to a method.

Definition

A *delegate* is a variable, a reference type that can contain a reference to a method. Using delegates is useful when you know that your application must perform an action by calling a method; but, at compile time, you do not know what that action will be.

Delegates allow you to specify at runtime the method to be invoked. Delegates are object-oriented, type-safe, and secure.

Declaring a delegate

A delegate specifies the return type and parameters that each method must provide.

Example

The following example shows how to declare a delegate type for a method that takes a single argument of type **Image** and has a **void** return type:

```
delegate void ImageFunctionsDelegateType(Image i);
```

Instantiating a delegate

After you declare a delegate type, you can create a delegate object and associate it with a particular method. A new delegate object is created with the **new** operator.

For example, the **Invert** method is in the **ImageUtility** class, as shown in the following code:

```
class ImageUtility {
  void Invert( Image i ) {
      // inverts an image
  }
}
```

The class is instantiated, as shown in the following code:

```
ImageUtility utilities = new ImageUtility();
```

You create an instance of the delegate as shown in the following code:

```
ImageFunctionsDelegateType someUtility;
someUtility = new ImageFunctionsDelegateType(
utilities.Invert );
```

The preceding code initialized **someUtility**, which is an instance of the delegate type **ImageFunctionsDelegateType**. The **someUtility** variable is then initialized, like any other reference type. The parameter that is passed to the delegate constructor is the name of a method that can be called through the delegate. Note that this must have the same signature as the delegate.

Calling a delegate

When the **someUtility** method delegate instance is subsequently called, the actual method that is invoked is **utilities.Invert**.

```
this.someUtility( imageToBeProcessed );
```

A delegate does not need to know the class of the object that it references. A delegate can reference any object as long as the signature of the method matches the signature of the delegate.

Example

For example, The Zoo Medical Center creates a class that zookeepers use when they check the health of the animals. The medical checkup to be performed, such as a tooth check or a claw check, is specific to the animal and therefore is listed as part of the animal definition. The following example shows how the **MedicalCenter** class uses a delegate instance called **animalCheckup** to reference and call the method instance passed as a parameter to **ScheduleAppointment**.

1. Define the animals and include the information about the types of checkup they need:

   ```
   public abstract class Animal { }

   public class Antelope : Animal {
      public void CheckHooves() {
          Console.WriteLine("Schedule Hoof Checkup");
      }
   }
   public class Lion: Animal {
      public void CheckClaws() {
          Console.WriteLine("Schedule Claw Checkup");
      }
   }
   ```

2. Define the delegate:

   ```
   public delegate void AppointmentType();
   ```

   The signature of the delegate matches the **CheckHooves** and **CheckClaws** methods.

3. Define the medical center:

   ```
   public class MedicalCenter {
      private AppointmentType animalCheckup;

      public void ScheduleAppointment(AppointmentType a) {
         animalCheckup = a;
      }
      public void ProcessNextPatient() {
         animalCheckup();
      }
   }
   ```

   The **MedicalCenter** class has a private member **animalCheckup**, which is the delegate instance. The **ScheduleAppointment** method is called by the user of the **MedicalCenter** class to schedule an animal for a check-up or a medical procedure. It assigns the instance method passed in the **AppointmentType** parameter to **animalCheckup**.

   To use this system, create or obtain a **MedicalCenter** object and some animals:

   ```
   MedicalCenter animalHospital = new MedicalCenter();
   Antelope bigMaleAntelope = new Antelope();
   Lion notVeryBraveLion = new Lion();
   ```

4. Schedule a medical check-up.

   ```
   animalHospital.ScheduleAppointment( new AppointmentType(
   bigMaleAntelope.CheckHooves ) );
   ```

   This code creates a new **AppointmentType** delegate instance that will call the **CheckHooves** method and passes it to the **MedicalCenter** object.

   In this implementation, only one appointment is stored, and it can be processed by invoking:

   ```
   animalHospital.ProcessNextPatient();
   ```

   This code in turn calls **animalCheckup()**, which invokes the **CheckHooves** method.

   In a more realistic implementation, the medical center may maintain a queue of animals that are expecting procedures, and the **ProcessNextPatient** method would dequeue animals when facilities became available, always calling the **animalCheckup** method to invoke the specific medical procedure for that animal.

The complete code sample follows:

```
using System;

namespace DelegateExample {
  class Zoo {
      public abstract class Animal { }

      public class Antelope : Animal {
          public void CheckHooves() {
              Console.WriteLine("Schedule Hoof Checkup");
          }
      }
      public class Lion: Animal {
          public void CheckClaws() {
              Console.WriteLine("Schedule Claw Checkup");
          }
      }

      public class MedicalCenter {
          public delegate void AppointmentType();
          public AppointmentType animalCheckup;

          public void ScheduleAppointment(AppointmentType a) {
              animalCheckup = a;
          }
          public void ProcessNextPatient() {
              if ( animalCheckup != null ) {
                  animalCheckup();
              }
          }
      }

      static void Main(string[] args) {
          MedicalCenter animalHospital = new MedicalCenter();
          Lion notVeryBraveLion = new Lion();
          Antelope bigMaleAntelope = new Antelope();

          animalHospital.ScheduleAppointment( new
MedicalCenter.AppointmentType( bigMaleAntelope.CheckHooves )
);
          animalHospital.ProcessNextPatient();

          animalHospital.ScheduleAppointment ( new
MedicalCenter.AppointmentType( notVeryBraveLion.CheckClaws )
);

          animalHospital.ProcessNextPatient();
      }
  }
}
```

This code sample is provided on the Student Materials compact disc in the DelegateSample.sln file in the Samples\Mod05\DelegateSample folder.

Multicasting

In the preceding example, it is only possible to schedule one appointment at a time. To schedule two or more procedures for one animal, you could maintain a queue, or array, of the methods and call them in sequence. However, a delegate can call more that one method. This is called multicasting. The delegate maintains a list of methods that it calls in order. This list is called the delegate's *invocation list*.

Multicasting is used frequently in the .NET Framework to allow a user interface object to have multiple event handlers.

Syntax

To combine delegates, use the following syntax:

```
DelegateType d1 = d2 + d3;
```

When d2 and d3 are combined, the methods that they encapsulate are added to the d1's invocation list. Invoking d1 invokes both methods that are referenced in d2 and d3.

A more common way to add delegates is to use the += operator to add delegates. This syntax is used to add event handlers to Windows controls, as shown in the following example:

```
Button myButton.Click += new System.EventHandler(myAction);
```

You can also use – and **-=** to remove delegates as follows:

```
d1 -= d3;
```

This removes the last delegate from d1 that contains a method that matches the method *named* in d3. This means that you can also remove a delegate as follows:

```
myClass.d1 += new DelegateType ( myMethod );
//. . .
d1 -= new DelegateType ( myMethod );
```

The removal statement removes a **myMethod** method from the invocation list.

Example

For example, the code in the Zoo Medical Center application can be modified so that one animal can be scheduled for multiple procedures, as shown in the following example. The following code invokes both **CheckHooves** and **TakeBloodSample** from one delegate. The methods simply write a string to the console.

```
public class Antelope : Animal {
  public void CheckHooves() {
      Console.WriteLine("Schedule Hoof Checkup");
  }
  public void TakeBloodSample() {
      Console.WriteLine("Schedule Blood Check");
  }
}
```

Then instantiate the **Antelope** object, just as before, but this time the **ScheduleAppointment** method is called twice.

```
Antelope bigMaleAntelope = new Antelope();

animalHospital.ScheduleAppointment( new
MedicalCenter.AppointmentType( bigMaleAntelope.CheckHooves )
);
animalHospital.ScheduleAppointment( new
MedicalCenter.AppointmentType( bigMaleAntelope.TakeBloodSample
) );
```

In the original **ScheduleAppointment** method, the **TakeBloodSample** method replaces the **CheckHooves** method in the delegate, so the **ScheduleAppointment** method must be modified slightly, as shown in the following code:

```
public void ScheduleAppointment(AppointmentType a) {
  animalCheckup += a
}
```

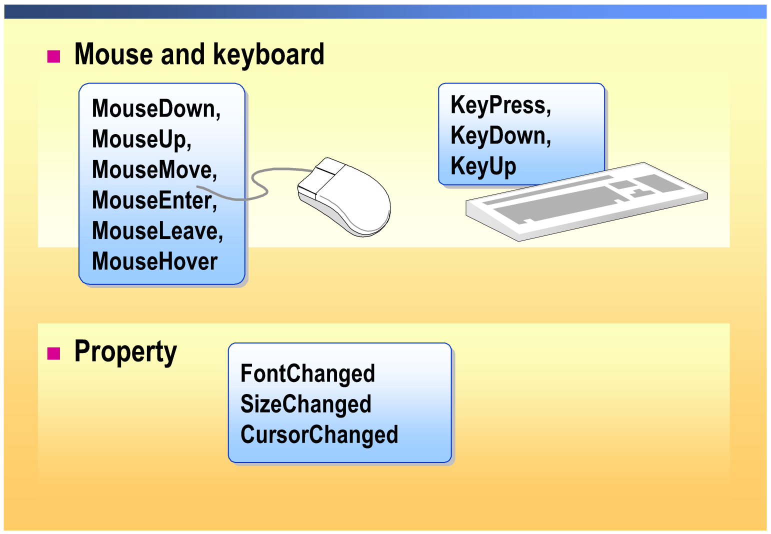
The **animalCheckup** delegate now adds the second delegate to itself. The **ProcessNextPatient** code is unchanged:

```
public void ProcessNextPatient() {
  if ( animalCheckup != null ) {
      animalCheckup();
  }
}
```

A single call to **ProcessNextPatient** produces the following output:

```
Schedule Hoof Checkup
Schedule Blood Check
```

What Is an Event?



Introduction

When you create a Windows-based application or a Web application, you create an event-driven application. Event-driven applications execute code in response to an event. Each form that you create and control that you use exposes a predefined set of events that you can program against. When one of these events occurs and there is code in the associated event handler, that code is invoked.

Definition

An *event* is an action that you can respond to, or handle, in code. Events can be generated by a user action, such as clicking a button with the mouse or pressing a key. Events can also be programmatically generated. The primary use of multicasting is to handle events by using program code or the operating system.

Types of events

The types of events that are raised by an object vary, but many types of events are common to most controls. For example, most objects handle a **Click** event—when a user clicks a form, code in the **Click** event handler of the form is executed. The most commonly used events are keyboard, mouse, and property events. If your application supports drag-and-drop operations, it will handle drag-and-drop events.

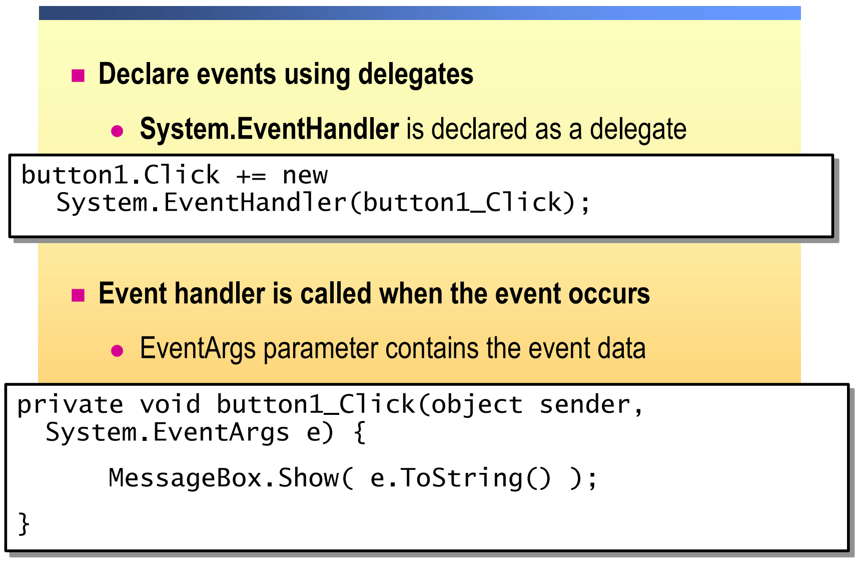
Mouse and keyboard events

Several events are related to the user's use of the mouse and keyboard. Each of these events has an event handler for which you can write code in your Windows-based applications. These events include **MouseDown**, **MouseUp**, **MouseMove**, **MouseEnter**, **MouseLeave**, **MouseHover**, **KeyPress**, **KeyDown**, and **KeyUp**. The mouse-related event handlers receive an argument of type **EventArgs**, which contain data related to their events. The key-related event handlers receive an argument of type **KeyEventArgs**, which contains data related to their events.

Property events

Property events occur when a property changes. For example, a control can register to receive a **SizeChanged** event when its **Size** property changes.

How to Write an Event Handler



- **Declare events using delegates**
  - **System.EventHandler** is declared as a delegate

```
button1.Click += new
  System.EventHandler(button1_Click);
```

- **Event handler is called when the event occurs**
  - EventArgs parameter contains the event data

```
private void button1_Click(object sender,
  System.EventArgs e) {

      MessageBox.Show( e.ToString() );

}
```

Introduction

The most familiar use for events is in graphical user interfaces. Typically, the classes that represent controls in the graphical user interface include events that are notified when a user manipulates the control, such as when a user clicks a button.

Events also allow an object to signal state changes that may be useful to clients of that object. Delegates are particularly suited for event handling and are used to implement event handling in C#.

Delegates as event handlers

The .NET framework event model uses delegates to bind events to the methods that are used to handle them. The delegate allows other classes to register for event notification by specifying a handler method. When the event occurs, the delegate calls the bound method or methods. The method that is called when the event occurs is referred as the *event handler*.

Tip Remember that multicasting allows a delegate to call several methods.

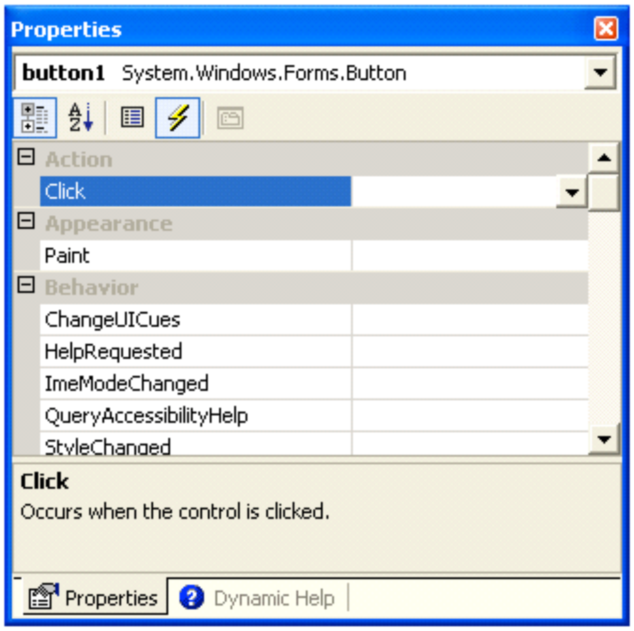
Receiving an event

The most common use of events in the .NET Framework is to handle activity in the user interface.

The easiest way to register to receive events is to use the Visual Studio .NET development environment.

The Properties window in the development environment contains an **Events** icon, as shown below:

Clicking this icon displays a list of events that can be sent from the selected object.



Click
Occurs when the control is clicked.

When you double-click the name of an event, the development environment creates an event handler for that event. The following code, which is generated in the initialization method, adds the **button1_Click** method to the delegate:

```
this.button1.Click += new
System.EventHandler(this.button1_Click);
```

Visual Studio .NET inserts the **button1_Click** method into the code, and you place your event-handling code in this method:

```
private void button1_Click(object sender, System.EventArgs e)
{

}
```

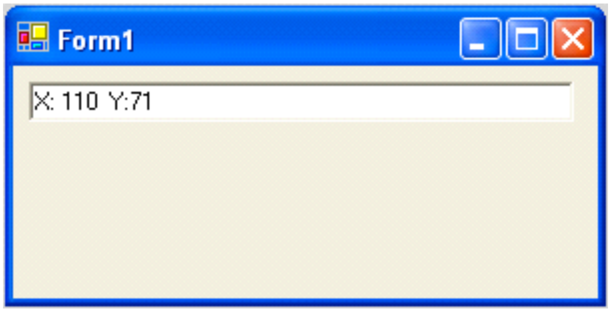
The **sender** parameter passes a reference to the object that caused the event. In the preceding example, this is a reference to the **button1** object.

The **System.EventArgs** class contains useful information about the event. Certain events pass a specific **EventArgs** class. For example, the **MouseUp**, **MouseDown**, and **MouseMove** events pass a **MouseEventArgs** object to the subscribed event handler. The **MouseEventArgs** class defines the X and Y position of the mouse, in addition to button click and mouse wheel information.

Example

The following example shows how to handle mouse events.

This Windows form displays the current coordinates of the mouse when the left mouse button is held down. As the mouse moves, the coordinates are updated. When the button is released, the display is cleared.



X: 110 Y:71

The text box is named **MouseCoordinateDisplay**. Three event handlers are registered for the form (**Form1)**, one each for **MouseDown**, **MouseUp**, and **MouseMove** event.

When the mouse button is pressed, a **MouseDown** event is sent to the handler, which sets a Boolean variable **tracking** to **true**.

When the mouse is moved, the **MouseMove** event handler is called. If tracking is set to **true**, this handler updates the display with the current mouse position.

When the button is released, the **MouseUp** event handler is called, which sets **tracking** to **false** and clears the display.

The event-handling code follows:

```
private bool tracking = false;
string coordinateDisplay = null;

private void Form1_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e) {
  tracking = true;
}

private void Form1_MouseUp(object sender,
System.Windows.Forms.MouseEventArgs e) {
  tracking = false;
  MouseCoordinateDisplay.Clear();
}

private void Form1_MouseMove(object sender,
System.Windows.Forms.MouseEventArgs e) {
  if ( tracking ) {
      coordinateDisplay = "X: " + e.X + "  Y:" + e.Y;
      MouseCoordinateDisplay.Text = coordinateDisplay;
  }
}
```

The following code registers the event handlers (**this** is Form1):

```
this.MouseDown += new
System.Windows.Forms.MouseEventHandler(this.Form1_MouseDown);
this.MouseUp += new
System.Windows.Forms.MouseEventHandler(this.Form1_MouseUp);
this.MouseMove += new
System.Windows.Forms.MouseEventHandler(this.Form1_MouseMove);
```

MouseEventHandler is the delegate type that is declared in the **Forms** class, and **MouseDown**, **MouseUp**, and **MouseMove** are the **Forms** class delegates that provide the event notification.

This code sample is available on the Student Materials compact disc, in the WinEventExample.sln file in the Samples\Mod05\WinEventSample folder.

Sending (declaring) events

When you declare an event in a class, you state that your class will notify other objects that have registered an event handler with your object.

To declare an event inside a class, you must first declare a delegate type for the event. The delegate type defines the set of arguments that are passed to the method that handles the event.

Example

Suppose that the Zoo Medical Center wants to allow users to receive an event when an animal is released. This example uses two classes: the **MedicalCenter** class, which implements an event, and the **ZooKeeper** class, which must be notified when animals are released. The example uses **Lion** and **Antelope** objects, the implementation of which can be seen in the full code example at the end of this topic.

The delegate type for the event is declared in the **MedicalCenter** class:

```
public delegate void AnimalCollectionHandler(Animal a);
```

The event itself is also declared in the **MedicalCenter** class. You declare an event as you would declare a field of **delegate** type, except that the **event** keyword follows the modifiers and precedes the **delegate** type. Events usually are declared public, but any accessibility modifier is allowed.

```
public event AnimalCollectionHandler
OnAnimalReadyForCollection;
```

It is normal to prefix your event with **On**.

The **ZooKeeper** class must have a method that will be invoked by the event. This method is as follows:

```
public void ReleasedNotification (Animal a) { . . . }
```

The purpose of this method is to notify the keeper when an animal is released from the medical center. Because it is invoked by the event delegate, it must have a matching signature.

The **Keeper** class registers this method as an event handler for **OnAnimalReadyForCollection** events:

```
public void GetReleaseNotifications(MedicalCenter hospital) {
  hospital.OnAnimalReadyForCollection += new
MedicalCenter.AnimalCollectionHandler(this.ReleasedNotificatio
n);
}
```

The **MedicalCenter** object will continue to process animals until none are left, sending an event as each animal is released. Pending work is stored on the **queuedProcedures** queue, and the event handlers are invoked by the **OnAnimalReadyForCollection** call. The **if** statement that checks against **null** is to test if any event handlers are registered.

```
public void ProcessPatients() {
  Animal a;
  for(;;) {
      if ( queuedProcedures.Count == 0 )
          break;
      a = (Animal) queuedProcedures.Dequeue();
      a.Checkup();
      if ( OnAnimalReadyForCollection != null ) {
          OnAnimalReadyForCollection( a );
      }
   }
}
```

Note the use of **for (; ;)** to loop forever.

The complete code example follows:

```
using System;
using System.Collections;

namespace EventExample {
   class Zoo {
       // represents a Zoo medical center
       public class MedicalCenter {
           private Queue queuedProcedures;

           // User of this class can register for the following event
           public delegate void AnimalCollectionHandler(Animal a);

           public event AnimalCollectionHandler OnAnimalReadyForCollection;

           public MedicalCenter() {
               // procedures are stored in a queue
               queuedProcedures = new Queue();
           }

           public void Add(Animal sickAnimal) {
               // Add animals to the work queue
               queuedProcedures.Enqueue(sickAnimal);
           }

           public void ProcessPatients() {
               Animal a;
               for(;;) {
                   if ( queuedProcedures.Count == 0 )
                       break;
                   a = (Animal) queuedProcedures.Dequeue();
                   a.Checkup(); // Do the medical procedure
                   if ( OnAnimalReadyForCollection != null ) {
                       // Call the event handler
                       OnAnimalReadyForCollection( a );
                   }
               }
           }
       }

       // User of the MedicalCenter
       public class ZooKeeper {
           // This is called when an animal is released
           public void ReleasedNotification (Animal a) {
               Console.WriteLine("Keeper: " + a + " was just released from the
medical center");
           }

           // This method registers the event handler
           public void GetReleaseNotifications(MedicalCenter hospital) {
               hospital.OnAnimalReadyForCollection += new
MedicalCenter.AnimalCollectionHandler(this.ReleasedNotification);
           }
       }
```

Code continued on the following page.

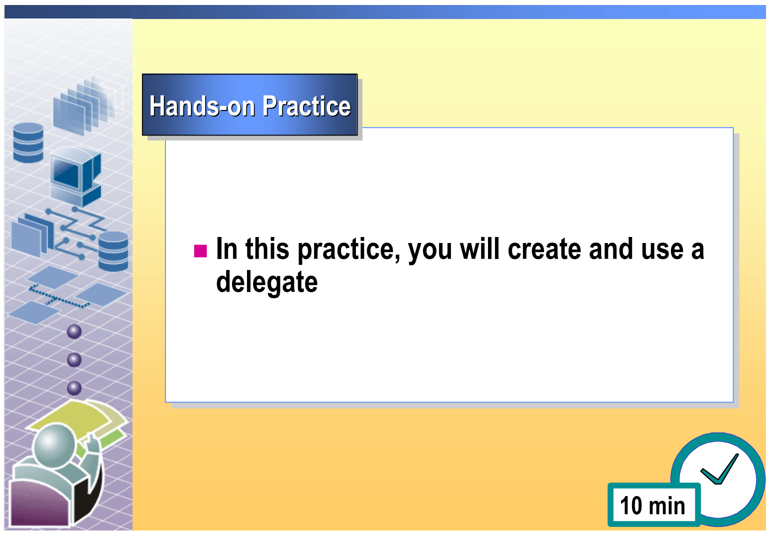
```
        static void Main(string[] args) {
            MedicalCenter animalHospital = new MedicalCenter();

            Lion notVeryBraveLion = new Lion();
            Antelope bigMaleAntelope = new Antelope();

            ZooKeeper manager = new ZooKeeper();
            manager.GetReleaseNotifications(animalHospital);

            animalHospital.Add( notVeryBraveLion );
            animalHospital.Add( bigMaleAntelope );

            animalHospital.ProcessPatients();

            Console.ReadLine();
        }
    }

    public abstract class Animal {
        public abstract void Checkup();
    }
    public class Antelope : Animal {
        public override void Checkup() {
            Console.WriteLine("Antelope: Checkup ");
        }
        public override string ToString() {
            return "Antelope";
        }
    }
    public class Lion: Animal {
        public override void Checkup() {
            Console.WriteLine("Lion: Checkup");
        }
        public override string ToString() {
            return "Lion";
        }
    }
}
```

The preceding code produces the following output:

```
Lion: Checkup
Keeper: Lion was just released from the medical center
Antelope: Checkup
Keeper: Antelope was just released from the medical center.
```

This code sample is available on the Student Materials compact disc, in the EventExample.sln file in the Samples\Mod05\EventSample folder.

Practice: Declaring and Calling a Delegate

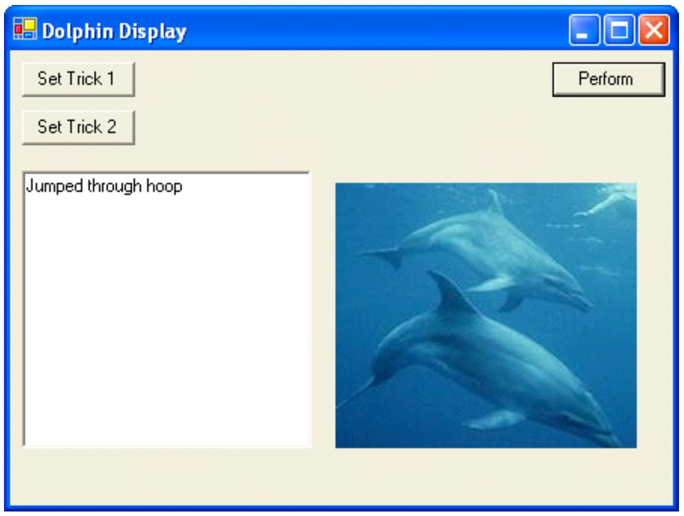

Hands-on Practice

- In this practice, you will create and use a delegate

In this practice, you will create and use a delegate.

The zoo has some interactive displays, where visitors can locate information about the animals in the zoo. One of the displays is oriented to children and displays information about dolphin behaviors and tricks that they can perform.

Because these tricks may change from time to time as old dolphins are taught new tricks; the trick implementation is referenced in the **Dolphin** class by a delegate.



Jumped through hoop

In this application, clicking a **Set Trick** button attaches a method to the dolphin's **Trick** delegate. Clicking **Perform** calls the method referenced in the delegate.

The solution for this practice is located in *install_folder*\Practices\Mod05\Delegates_Solution \Dolphin.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open *install_folder* \Practices\Mod05\Delegates \Dolphin.sln.	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start Page**, click **Open Project**. **c.** In the **Open Project** dialog box, browse to *install_folder*\Practices \Mod05\Delegates, click **Dolphin.sln**, and then click **Open**. **d.** In Solution Explorer, click **Dolphin.cs**, and then press F7 to open the Code Editor. The **Dolphin** class is located in the file **Dolphin.cs**.
2. Examine the Delegate declaration.	**a.** On the **View** menu, point to **Show Tasks**, and then click **All.** **b.** Double-click the task **TODO 1: Examine the delegate: void TrickType().** Note that the delegate declaration is for a method that takes no parameters and has a void return type.
3. In the **Dolphin** class, add a property called **Trick** that allows the user of the class to assign a method to the private delegate instance member **dolphinTrick**.	**a.** Locate the task **TODO 2: Write a public property called Trick**. **b.** Add a property called **Trick** that allows the user of the class to assign a method to **dolphinTrick** by removing the comments from the lines that follow the TODO comment.
4. In **Form1.cs**, locate the method **setTrick2_Click** and write code to assign a trick to **zooDolphin.Trick**.	**a.** In Solution Explorer, click **Form1.cs**, and then press F7 to open the Code Editor. **b.** Locate the task **TODO 3: Assign a trick to the dolphin**. **c.** In the **setTrick2_Click** method, remove the **Select a trick** comment, and replace it with one of the methods that implements a trick. Trick methods are located at the bottom of this file. The **JumpThroughHoop** method is already assigned to the **Set Trick 1** button. Note that this statement creates a new Delegate and assigns it to the **Trick** property of the **zooDolphin** object.
5. Test the application.	**a.** Press F5 to test your application. **b.** In the **Dolphin Display** window, click **Perform**. Nothing should happen because no methods have been assigned to the dolphin's delegate. **c.** Click **Set Trick 1**, and then click **Perform**. **d.** Click **Set Trick 2**, and then click **Perform**.
6. Quit Visual Studio .NET, saving your solution.	**a.** On the **File** menu, click **Save All**. **b.** On the **File** menu, click **Exit**.

Review

- Using Arrays
- Using Collections
- Using Interfaces
- Using Exception Handling
- Using Delegates and Events

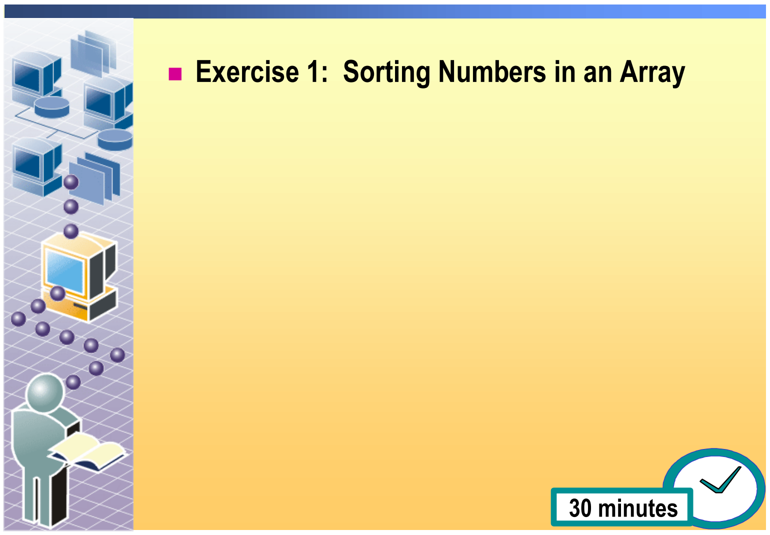
1. In the array int[] number = {1, 2, 3, 4 } how do you access the value 3?

2. Create an array that contains the integers 1, 2, and 3. Then use the **foreach** statement to iterate over the array and output the numbers to the console.

3. Name two collection objects in **System.Collections** namespace, and describe how these classes track objects.

4. What is a delegate, what is the benefit of using a delegate, and when should you use it?

Lab 5:1: Using Arrays



- Exercise 1: Sorting Numbers in an Array

30 minutes

Objectives

After completing this lab, you will be able to use an array to sort numbers.

Prerequisites

Before working on this lab, you must have the ability to:

- Declare and initialize arrays.
- Assign to and from arrays.

Note This lab focuses on the concepts in this module and as a result may not comply with Microsoft security recommendations.

Estimated time to complete this lab: 30 minutes

Exercise 0
Lab Setup

The Lab Setup section lists the tasks that you must perform before you begin the lab.

Task	Detailed steps
▪ Log on to Windows as **Student** with a password of **P@ssw0rd**.	▪ Log on to Windows with the following account: • User name: **Student** • Password: **P@ssw0rd** Note that the 0 in the password is a zero.

Note that by default the *install_folder* is C:\Program Files\Msdntrain\2609.

Exercise 1
Sorting Numbers in an Array

In this exercise, you will sort numbers in an array.

Scenario

Although C# and the .NET Framework provide sorting algorithms, as a programmer you will find it useful to understand how data is held in an array and what happens when that data is sorted.

In this lab, you will implement the **BubbleSort** method as shown in the following code:

```
private void BubbleSort( int[] anArray)
```

The **anArray** integer array contains randomly generated numbers. Your **BubbleSort** method will sort the numbers in the array, with the lower numbers at the beginning of the array.

You will use the following simple algorithm to sort the numbers in the array:

```
For every element in array a
    For every unsorted element in array a
        If a[n] > a[n + 1]
            swap them
```

To move the lowest numbers to the beginning of the array, pass through the array multiple times in the **BubbleSort** method, each time moving the highest unsorted number to the end of the list.

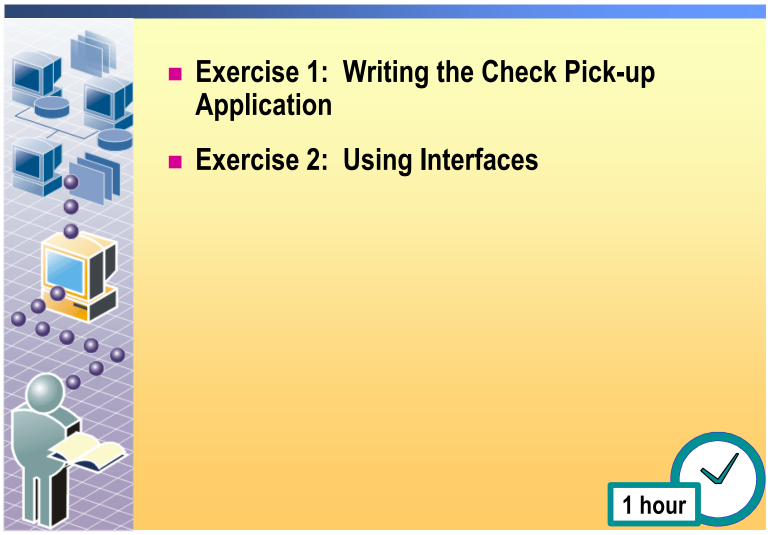
Within the loop, you examine each element *n* in array *a* and compare the element to the one that follows it, *n* + 1. If *a*[*n*] is greater than *a*[*n* + 1], the elements must be swapped. Continue to move down the list comparing adjacent elements. In this way, higher values are moved down the list until they are in place.

On each pass, you must compare only the unsorted elements of the array.

The solution code for this lab is provided in *install_folder*\Labfiles\Lab05_1\Exercise1 \Solution_Code\Bubble Sort.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open *install_folder* \Labfiles\Lab05_1\Exercise1 \Bubble Sort.sln.	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start Page**, click **Open Project**. **c.** In the **Open Project** dialog box, browse to i*nstall_folder*\Labfiles \Lab05_1\Exercise1, click **Bubble Sort.sln**, and then click **Open**.
2. Examine the code and run the sample application.	**a.** In Solution Explorer, click **Bubble.cs**, and then press F7. **b.** Locate the **sort_Click** method, located at the bottom of the file, and examine the code. The **sort_Click** method is called when you click the **Sort** button in the application. It generates 1000 random integers and places in them in array **a**. It then calls the **BubbleSort** method, passing the array. Finally it lists the contents of the array in the application window. **c.** Build and run the application by pressing F5. **d.** In the application window, click **Sort**. **e.** Close the application window.
3. Write the **BubbleSort** method.	■ Locate the **BubbleSort** method and write code to sort the elements in the array, using the algorithm outlined in the scenario.
4. Test your method by running the program and clicking the **Sort** button.	**a.** In Visual Studio .NET, press F5 to compile and run your program. **b.** In the **Sorting an Array** window, click **Sort**.
5. Quit Visual Studio .NET, saving your solution.	**a.** On the **File** menu, click **Save All**. **b.** On the **File** menu, click **Exit**.

Lab 5.2 (optional): Using Indexers and Interfaces



- Exercise 1: Writing the Check Pick-up Application
- Exercise 2: Using Interfaces

Objectives

After completing this lab, you will be able to:

- Use collection classes to manage objects.
- Write indexers to provide indexed access to the data in your objects.
- Implement interfaces for an object.
- Use exception handling to handle unexpected errors.

Note This lab focuses on the concepts in this module and as a result may not comply with Microsoft security recommendations.

Prerequisites

Before working on this lab, you must have the ability to:

- Declare, initialize, and use indexers.
- Implement interfaces.

Scenario

When bank customers who order traveler's checks ask to pick them up in person, the check numbers and amounts are placed in a list for the bank teller who subsequently retrieves the checks and places them in an envelope for the customers.

You have been asked to create a prototype for a system that reads the list of checks and displays them to the teller. This system consists of three parts:

- The user interface where the data operator will retrieve, view, and clear the check information, which is provided to you.
- The component that generates the list of checks, which is simulated and provided to you.
- A data structure that holds the collection of the checks that the teller retrieved but has not yet processed.

Estimated time to complete this lab: 60 minutes

Exercise 0
Lab Setup

The Lab Setup section lists the tasks that you must perform before you begin the lab.

Task	Detailed steps
▪ Log on to Windows as **Student** with a password of **P@ssw0rd**.	▪ Log on to Windows with the following account: • User name: **Student** • Password: **P@ssw0rd** Note that the 0 in the password is a zero.

Note that by default the *install_folder* is C:\Program Files\Msdntrain\2609.

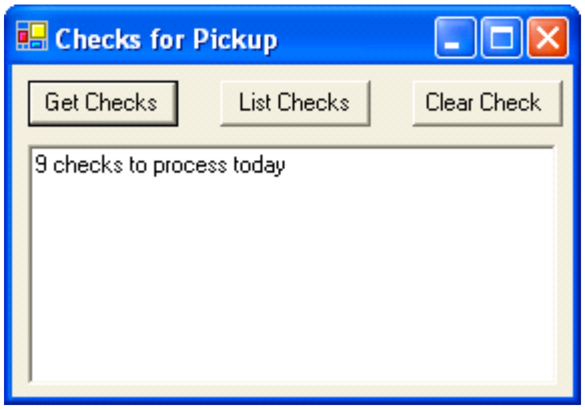
Exercise 1
Writing the Check Pick-up Application

In this exercise, you will write the traveler's check customer pick-up application.

Scenario

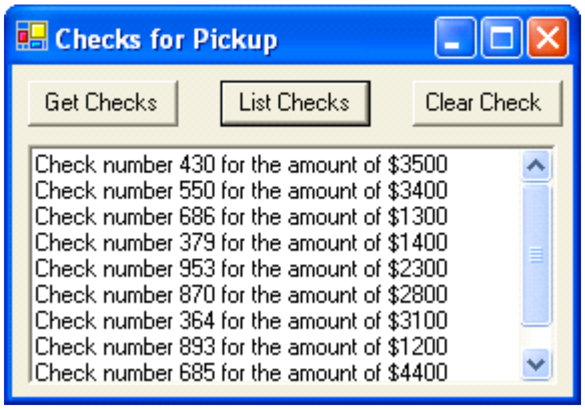
A bank provides a service that allows users to order traveler's checks by using the Web site, and then pick them up in person.

The **Get Checks** button retrieves the checks that are waiting for collection and then displays the number of retrieved checks. The code that generates the checks is provided to you. The application is shown in the following illustration:



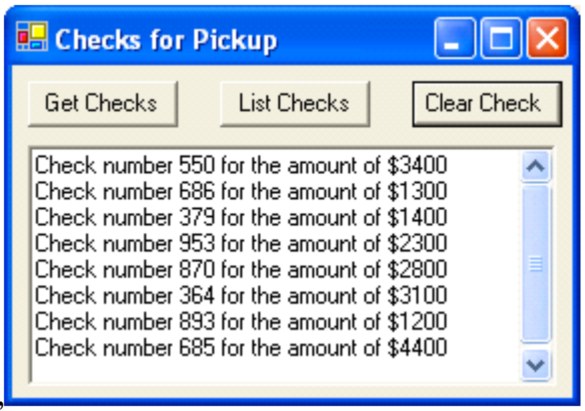
```
9 checks to process today
```

The **List Checks** button lists the checks that were retrieved, as shown in the following illustration:



```
Check number 430 for the amount of $3500
Check number 550 for the amount of $3400
Check number 686 for the amount of $1300
Check number 379 for the amount of $1400
Check number 953 for the amount of $2300
Check number 870 for the amount of $2800
Check number 364 for the amount of $3100
Check number 893 for the amount of $1200
Check number 685 for the amount of $4400
```

The **Clear Check** button removes a check from the top of the list:



```
Check number 550 for the amount of $3400
Check number 686 for the amount of $1300
Check number 379 for the amount of $1400
Check number 953 for the amount of $2300
Check number 870 for the amount of $2800
Check number 364 for the amount of $3100
Check number 893 for the amount of $1200
Check number 685 for the amount of $4400
```

,

You have been provided with a sample user interface to the traveler's check customer pick-up application. It currently consists of the following C# source files:

- *Checks.cs.* Implement your check collection class in this file. The file contains two classes, **Checks** and **TravelerCheck**. **TravelerCheck** is a simple implementation of the information that is required to locate a traveler's check, and **Checks** is the class that you will implement. You do not need to modify the **TravelerCheck** class. **Checks** will maintain a collection of **TravelerCheck** objects, and provide methods and properties that allow users of the class to do the following:
  - Add **TravelerCheck** objects to the **Checks** collection
  - Reference **TravelerCheck** objects in **Checks** by index, for example:

    ```
    myTravelerCheck = todaysChecks[0];
    ```

  - Remove **TravelerCheck** objects from **Checks**
- *Form1.cs.* The main window. The code that calls your check collection class is in this file.
- *CheckSource.cs.* Contains the code that generates the list of checks that will be collected today. It contains one class, **CheckSource**, with one method, **Next**, that you use to retrieve checks, as shown in the following code:

  ```
  // source is the CheckSource object
  TravelerCheck someCheck = source.Next();
          while (someCheck != null ) {
              someCheck = source.Next();
              // add to todaysChecks
          }
  ```

You will not need to modify the **CheckSource** class.

The solution for this lab is provided in *install_folder*\Labfiles\Lab05_2\Exercise1\Solution_Code \Checkprocess.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open *install_folder*\Labfiles \Lab05_2\Exercise1 \Checkprocess.sln.	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start** Page, click **Open Project**. **c.** In the **Open Project** dialog box, browse to *install_folder*\Labfiles \Lab05_2\Exercise1, click **Checkprocess.sln**, and then click **Open**.
2. Review the code in Form1.cs and in Checks.cs.	**a.** In Solution Explorer, click **Form1.cs**, and then press F7 to open the Code Editor. **b.** Find the constructor for Form1 and note that the objects **source** and **todaysChecks** are created. • The **source** object is used to produce a list of checks, by using the algorithm described above. • The **todaysChecks** object is an instance of the **Checks** class which you will implement. **c.** Locate the **getChecks_Click**, **listChecks_Click**, and **clearCheck_click** methods, and then review the comments and code in them. **d.** In Solution Explorer, click **Checks.cs**, and then press F7 to open the Code Editor. Review the code. This file contains two classes, **TravelerCheck** and **Checks**. The **TravelerCheck** class represents a single traveler's check, and the **Checks** class represents the collection of checks.
3. Write code that allows the user of the **Checks** class to add **TravelerCheck** objects to a data structure maintained within the class.	**a.** Choose an appropriate data structure for the **Checks** class, and use it to maintain a collection of **TravelerCheck** objects: • Declare the data structure below the declaration of **Count**. • Add a constructor and initialize the data structure in the constructor. **b.** Write a method that adds items to the data structure.
4. Complete the code in the **getChecks_Click** method, and then test your code.	**a.** On the **View** menu, point to **Show Tasks**, and then click **All**. **b.** In the Task List, double-click **TODO 1: Get the list of checks from the check source**. **c.** Complete the **getChecks_Click** method. **d.** Build your application. **e.** Run your application, and then click **Get Checks**. You should see a message indicating that you have some checks to process.

Tasks	Detailed steps
5. Complete the code in the **listChecks_Click** method, and then test your code.	**a.** On the **View** menu, point to **Show Tasks**, and then click **All**. **b.** In the **Task List**, double-click **TODO 2: Write an indexer and use it to list the contents**. After you write the indexer, you can use the code that is commented out in the method to display the list of checks. **c.** Complete the **listChecks_Click** method. **d.** Build your application. **e.** Run your application, click **Get Checks**, and then click **List Checks**. You should see a list of the checks to process.
6. Write a method that removes the top item from the list of checks.	**a.** In Solution Explorer, click **Checks.cs**, and then press F7. **b.** In the **Checks** class, write a method that removes the top item from the list of checks.
7. Complete the code in the **clearCheck_Click** method, and then test your code.	**a.** On the **View** menu, point to **Show Tasks**, and then click **All**. **b.** In the Task List, double-click **TODO 3: Remove the top check**. **c.** Complete the method **clearCheck_Click**. **d.** Include code to refresh the display. An easy way to do this is to simulate a click of the **List Checks** button: `listChecks.PerformClick();` **e.** Build your application. **f.** Run your application, and then click **Clear Check**. You should see that the top check is removed from the list.
8. Save your solution.	▪ On the **File** menu, click **Save All**.

Exercise 2
Using Interfaces

In this exercise, you will implement the **IEnumerable** interface for the **Checks** class, and test this interface by replacing the **for** statement in **listChecks_Click** with a **foreach** statement.

You can either continue to use your application from Exercise 1, or you can use the starter code that is provided.

The solution for this lab is provided in *install_folder*\Labfiles\Lab05_2\Exercise2\Solution_Code\Checkprocess.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. In Visual Studio .NET, either use the application that you wrote in Exercise 1, or open *install_folder*\Labfiles\Lab05_2\Exercise2\Checkprocess.sln.	**a.** If you want to use the starter code, then in Visual Studio .NET, on the **File** menu, point to **Open**, and then click **Project**. **b.** In the **Open Project** dialog box, browse to *install_folder*\Labfiles\Lab05_2\Exercise2. **c.** Click **Checkprocess.sln**, and then click **Open**.
2. Modify the **Checks** class so that it implements the **IEnumerable** interface.	**a.** Change the **Checks** class so that it inherits **IEnumerable**. **b.** Implement **GetEnumerator**. **c.** In the **Checks** class, create a **CheckEnumerator** class that implements **IEnumerator**.
3. Test your code by modifying the **listChecks_Click** method in the **Form1** class so that it uses a **foreach** loop.	**a.** Delete the **for** statement in your code and substitute a **foreach** statement. **b.** Build your application, and then run it. **c.** Check **Get Checks,** and then click **List Checks**. You should see a list of the checks to process.
4. Save your solution, and then quit Visual Studio .NET.	**a.** On the **File** menu, click **Save All**. **b.** On the **File** menu, click **Exit**.

Lab 5.3 (optional): Using Delegates and Events

- Exercise 1: Working with Events and Delegates

Objectives

After completing this lab, you will be able to:

- Create and use delegates.
- Write event procedures and send events to registered event handlers.

Note This lab focuses on the concepts in this module and as a result may not comply with Microsoft security recommendations.

Prerequisites

Before working on this lab, you must have the ability to use delegates, events, and event handlers.

Scenario

In this lab, you will write an application that allows the user to buy and sell stock by setting a buy value and a sell value.

Estimated time to complete this lab: 60 minutes

Exercise 0
Lab Setup

The Lab Setup section lists the tasks that you must perform before you begin the lab.

Task	Detailed steps
▪ Log on to Windows as **Student** with a password of **P@ssw0rd**.	▪ Log on to Windows with the following account: • User name: **Student** • Password: **P@ssw0rd** Note that the 0 in the password is a zero.

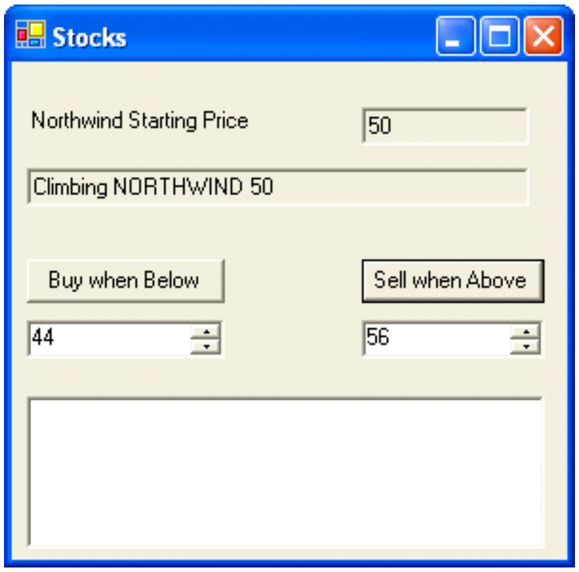
Note that by default the *install_folder* is C:\Program Files\Msdntrain\2609.

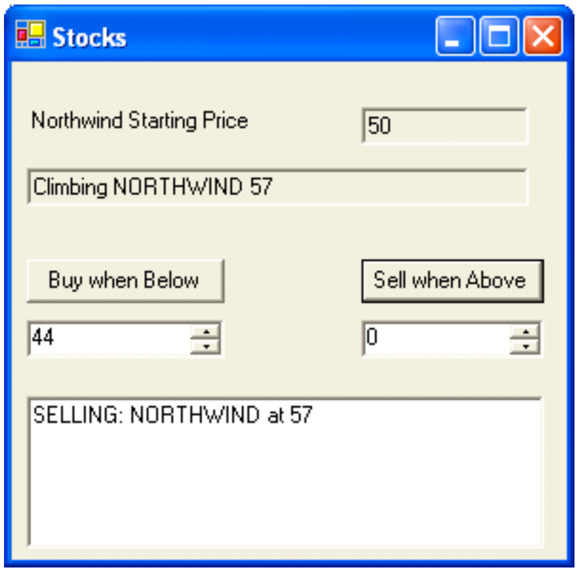
Exercise 1
Working with Events and Delegates

In this exercise, you will complete an application that simulates a detail page on a stock trader's application.

You will write code that represents a stock. The stock will raise three events: **OnStockChange**, **OnStockFall**, and **OnStockRise**. The trader's application allows the trader to use the **OnStockFall** event to buy the stock at a given price. The **OnStockRise** price is used to specify the sale price of the stock.

The application is shown in the following illustrations:





SELLING: NORTHWIND at 57

When the application runs, it monitors the current price of the stock for a fictitious company, Northwind Traders, displaying the current price in the center area. The trader can set a Buy or Sell price, by using the **NumericUpDown** boxes, and then click **Buy when Below** or **Sell when Above** to place the order. If the price drops below the buy price, stock is purchased and the order is removed. Similarly, if the price rises above the sell price, the stock is sold and the order is removed.

Because this lab is about using delegates and events to place and act on the orders, the quantities of stock being bought and sold are not considered.

The code for the user interface and the stock price changes is provided to you. Your task is to implement the **OnStockRise** and **OnStockFall** events, and to register handlers for them.

How the Code Works

The **StockTicks** class contains a **StockTicker** method that regularly updates the price of a stock. When the application initializes, the Northwind Traders stock is passed to the **StockTicker**, and thereafter the **Price** property is automatically updated.

When the **Price** property changes, you should raise the **Stock** class appropriate event. For example, when the stock price goes up, raise the **OnStockRise** and **OnStockChange** events.

The **Form1** class, the code for the main window, uses the **Stock** class in the following ways:

- To update the stock price display, the **Form1** class registers for **OnStockRise** and **OnStockFall** events.

When a button is clicked, the **Form1** class also registers either a **Buy** event handler or **Sell** event handler for the events, depending on the button that is clicked. For example, the **Buy** event handler is registered for the **OnStockFall** event, and when the price drops below the value in the **NumericUpDown** box, it buys the stock.

When a **Buy** or **Sell** decision is made, you clear the order by removing the event handler.

Note: Some of the starter code in this lab uses events and properties of the **Stock** class that you will write. For that reason, the starter code will not compile until you provide those elements in the **Stock** class.

The solution for this lab is provided in *install_folder*\Labfiles\Lab05_3\Exercise1\Solution_Code \StockPrice.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open *install_folder* \Labfiles\Lab05_3\Exercise1 \StockPrice.sln.	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start** Page, click **Open Project**. **c.** In the **Open Project** dialog box, browse to *install_folder*\Labfiles \Lab05_3\Exercise1, click **StockPrice.sln**, and then click **Open**. **d.** In Solution Explorer, click **Stock.cs**, and then press F7 to open the Code Editor.
2. In the **Stock** class, declare a delegate named **StockChange**.	**a.** On the **View** menu, point to **Show Tasks**, and then click **Comment**. **b.** In the Task List, double-click **TODO 1: Declare a delegate named StockChange**. **c.** In the **Stock** class, under the comment, declare a delegate named **StockChange**.
3. Declare two event handlers named **OnStockRise**, and **OnStockFall**.	**a.** In the Task List, double-click TODO 2: Declare two more event handlers. **b.** Under the comment, declare two event handlers named **OnStockRise**, and **OnStockFall**.

Tasks	Detailed steps
4. Write a property named **Price** that raises the following events: • **OnStockRise**: raise this event when the new stock price is higher than the previous stock price. • **OnStockFall**: raise this event when the new stock price is lower than the previous stock price.	**a.** In the Task List, double-click TODO 3: Complete the property named Price. **b.** The **Price** property encapsulates the **stockPrice** private member. **c.** Raise an **OnStockRise** event every time the **Price** property is assigned a value that is higher than its previous value. **d.** Raise an **OnStockFall** event every time the **Price** property is assigned a value that is lower than its previous value.
5. In the **Form1** class, in the **buying_Click** method, write code to add the method **Buy** as an event handler for the **OnStockFall** method.	**a.** In Solution Explorer, click **Form1.cs**, and then press F7. **b.** In the Code Editor, locate the **buying_Click** method. This method is called when the user clicks the **Buy when Below** button in the user interface. **c.** Add the **Buy** method as an event handler for the **OnStockFall** method.
6. In the **Buy** method, add code to remove **Buy** from the **OnStockFall** delegate after a successful stock purchase.	**a.** Locate the **Buy** method, directly below the **buying_Click** method. **b.** Add code to remove the **Buy** method from the invocation list of **OnStockFall**.
7. Write the code to sell stock at a price specified in **sell.Value**.	▪ Locate the task TODO 4: Write code to sell at a price specified in sell.Value, and write the code.
8. Test your application.	▪ Compile and run your application.
9. Save your solution, and then quit Visual Studio .NET.	**a.** On the **File** menu, click **Save All**. **b.** On the **File** menu, click **Exit**.