Lecture 4b

Using Arrays and Collections

OBJECTIVES

In this lecture you will learn:

- **Declare, initialize, copy, and use array variables**.
- **Declare, initialize, copy, and use variables of various collection types**

References:

Chapter 10, John Sharp, “*Microsoft Visual C# 2008 Step By Step*”, **Microsoft Press, 2008 ISBN-13: 978-1-7356-2430-6**

Outline

1 Introduction

All the examples of variables you have seen so far have one thing in common- they hold information about a single item (an *int*, *a float*, *a Circle*, *a Time*, *and so on)*.

What happens if you need to manipulate sets of items?

How many variables do you need?

How should you name them?

How would you avoid very repetitive code?

2 Arrays in C#.NET

An *array is an unordered sequence of elements*. ***All the elements in an array have the same*** **type** (unlike the fields in a structure or class, which can have different types).

The elements of an array **live in a contiguous block of memory and are accessed by using an integer index** (unlike fields in a structure or class, which are accessed by name).

2 Arrays in C#.NET

Declaring Array Variables

You declare an array **variable by specifying the name of the element type**, followed by a **pair of square brackets**, followed by the **variable name**. The **square brackets signify that the variable is an array**.

For **example**, to declare an **array** of ***`int`*** *elements named* ***`pins`***, *you would write:*

```
int[] pins; // Personal Identification Numbers
```

2 Arrays in C#.NET

C and C++ programmers should note that **the size of the array** is **not part** of **the declaration**

Java programmers should note that **you must place the square brackets** *before* the **array name**

Note:

You are **not restricted to primitive types** as array elements. You can also **create arrays of structures**, **enumerations**, and **classes**. For **example**, you can create an array of **`Time`** structures like this:

```
Time[] times;
```

2 Arrays in C#.NET

Tip

It is **often useful to give array elements plural names**, such as ***`places`*** *(where each element is a Place)*, ***`people`*** *(where each element is a Person)*, *or* ***`times`*** *(where each element is a Time)*.

3 Creating an Array Instance

Arrays are reference types, regardless of the type of their elements.

This means **that an array variable** *refers to the* ***contiguous block of memory holding the array elements on the heap*** (just as a class variable refers to an object on the heap) and **does not hold its array elements directly on the stack** (as a structure does).

3 Creating an Array Instance

Remember that when you declare a class variable, memory is not allocated for the object until you create the instance by using *new*.

Arrays follow the same rules- ***when you declare*** **an array variable, you do not declare its size**.

You **specify the size of an array only when you actually create an array instance**

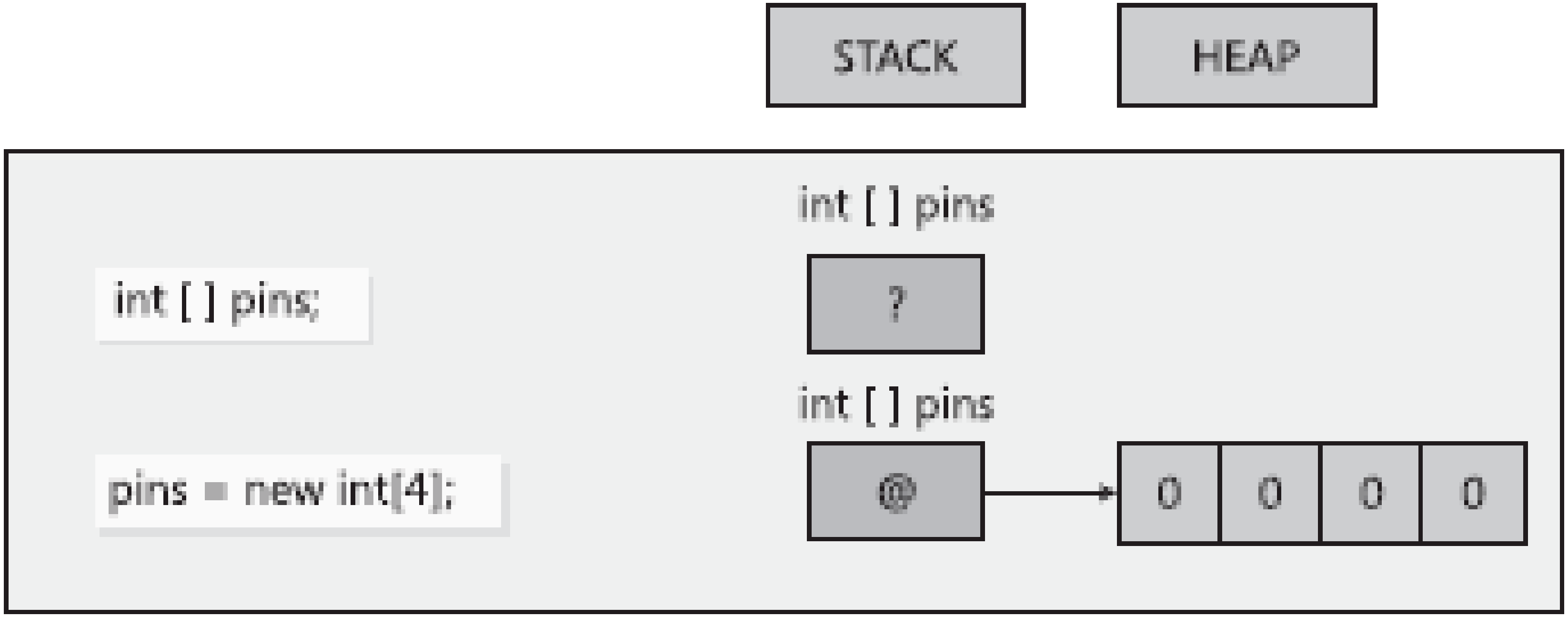
3 Creating an Array Instance

To create an array instance, you use the **`new`** *keyword followed by the name of the element* type, followed by the **`size`** of the array you're creating between **`square brackets`**. **Creating an array also initializes its elements by using the now familiar default values (`0`, `null`**, *or* **`false`**, *depending* on whether the type is numeric, a reference, or a **`Boolean`**, respectively).

For example, to create and initialize a **`new`** array of four integers for the **`pins`** *variable declared earlier*, you write this:

```
pins = new int[4];
```

3 Creating an Array Instance



3 Creating an Array Instance

The size of an array instance does not have to be a constant- it can be calculated at run time, as shown in this example:

```
int size =
    int.Parse(Console.ReadLine());
int[] pins = new int[size];
```

You're **allowed to create an array whose size is 0**. It's useful in situations where the **size of the array is determined dynamically** and could be 0.

An **array of size 0** is not a `null` *array*.

3 Creating an Array Instance

It’s also possible to create **multidimensional arrays**.

For example, to create a two-dimensional array, you create an array that requires two integer indexes.

Detailed discussion of multidimensional arrays is beyond the scope of this lecture, but here’s an example:

```
int[,] table = new int[4,6];
```

3 Creating an Array Instance

Initializing Array Variables

When you create an array instance, **all the elements of the array instance are initialized to a default value depending on their type**. You can modify this behavior and initialize the elements of an array to specific values if you prefer. You achieve this by providing a **comma separated list of values** between a pair of braces. For example, to initialize **`pins`** *to an array of* four **`int`** *variables whose values are* **`9, 3, 7,`** *and* **`2`**, *you would write this:*

```
int[] pins = new int[4]{ 9, 3, 7, 2 };
```

3 Creating an Array Instance

Initializing Array Variables

The values between the braces do not have to be constants. They can be values calculated at run time, as shown in this example:

```
Random r = new Random();
int[] pins = new int[4]{ r.Next() % 10,
                         r.Next() % 10,
                         r.Next() % 10,
                         r.Next() % 10 };
```

3 Creating an Array Instance

Initializing Array Variables

The number of values between the braces must exactly match the size of the array instance being created:

```
int[] pins = new int[3]{ 9, 3, 7, 2 }; // compile-time error
int[] pins = new int[]{ 9, 3, 7 };     // okay
int[] pins = new int[4]{ 9, 3, 7, 2 }; // okay
int[] pins = { 9, 3, 7, 2 };           // okay
Time[] schedule = { new Time(12,30), new Time(5,30) };
```

4 Creating an Implicitly Typed Array

The **element type** when you declare an array **must match the type of elements** that you attempt to store in the array.

For example, if you declare **`pins`** *to be an array of* **`int`**, *as shown* in the preceding examples, you cannot store a **`double`**, **`string`**, **`struct`**, *or anything that is not an* **`int`** *in this array. If you specify a **list of initializers** when declaring an array, you can **let the C# compiler infer*** the actual type of the elements in the array for you, as:

```
var names = new[]{"John", "Diana",
                  "James", "Francesca"};
```

4 Creating an Implicitly Typed Array

In this example, the **C# compiler determines** that the ***`names`*** *variable is an array of* ***`strings`***. *It is* worth pointing out a couple of syntactic quirks in this declaration.

__**First**__, you **omit the square brackets** from the type; the ***`names`*** *variable in this example is declared simply as* ***`var`***, *and* ***not*** ***`var[]`***.

__**Second**__, *you must specify the* ***`new`*** *operator and* ***`square`*** *brackets* ***before*** *the initializer* list.

4 Creating an Implicitly Typed Array

If you use this syntax, you **must ensure that all the initializers have the same type**.

This next example will cause the **compile-time error** "***No best type found for implicitly typed array***":

```
var bad = new[]{"John", "Diana", 99, 100};
```

4 Creating an Implicitly Typed Array

However, in some cases, the compiler will convert elements to a different type if doing so makes sense.

In the following code, the **`numbers`** *array is an array of* **`double`** *because the constants* **`3.5`** *and* **`99.999`** *are both* **`double`**, *and the C# compiler can convert the* **`integer`** *values* **`1`** and **`2`** *to* **`double`** *values:*

```
var numbers = new[]{1, 2, 3.5, 99.999};
```

4 Creating an Implicitly Typed Array

However, in some cases, the compiler will convert elements to a different type if doing so makes sense.

In the following code, the **`numbers`** *array is an array of* **`double`** *because the constants* **`3.5`** *and* **`99.999`** *are both* **`double`**, *and the C# compiler can convert the* **`integer`** *values* **`1`** and **`2`** *to* **`double`** *values:*

```
var numbers = new[]{1, 2, 3.5, 99.999};
```

4 Creating an Implicitly Typed Array

The following code creates **an array of anonymous objects**, each containing **two fields** specifying the `name` and `age` of the members of a family:

```
var names = new[]{ new { Name = "John", Age = 43 },
                   new { Name = "Diana", Age = 43 },
                   new { Name = "James", Age = 15 },
                   new { Name = "Francesca",Age =13}
                 };
```

The **fields in the anonymous types** must be the **same for each element of the array**.

5 Accessing an Individual Array Element

To **access an individual array element**, you must **provide an index** indicating which element you require.

For **example**, you can read the contents of element `2` of the `pins` *array into an* `int` variable by using the following code:

```
int myPin;
myPin = pins[2];
```

5 Accessing an Individual Array Element

Similarly, you can **change the contents of an array** by assigning a value to an indexed element:

```
myPin   = 1645;
pins[2] = myPin;
```

Array indexes are **zero-based**. The initial element of an array lives at index 0 and not index 1.

An index value of *1 accesses the second element.*

5 Accessing an Individual Array Element

If you specify an index that is **less than 0** or **greater than or equal to the length** of the array, the compiler throws an **`IndexOutOfRangeException`**, *as below:*

```
try
{
  int[] pins = { 9, 3, 7, 2 };
  Console.WriteLine(pins[4]);
  // error, the 4th and last element is at index 3
}
catch (IndexOutOfRangeException ex)
{
// a managed code programming platform
}
```

5 Iterating Through an Array

All arrays inherit methods and properties from the ***`System.Array`*** *class in the* ***Microsoft .NET Framework***.

Arrays have a number of useful **built-in properties** and **methods**.

You can examine the ***`Length`*** *property to discover how many elements an array contains and iterate* through all the elements of an array by using a ***`for`*** *statement.*

5 Iterating Through an Array

This sample code writes the array element values of the *pins array to the console:*

```
int[] pins = { 9, 3, 7, 2 };
for (int index = 0; index < pins.Length; index++)
{
   int pin = pins[index];
   Console.WriteLine(pin);
}
// output all the elements in one command
Console.Writeline("[{0}]", string.Join(",", pins));
```

5 Iterating Through an Array

This sample code writes the array element values of the *pins array to the console:*

```
int[] pins = { 9, 3, 7, 2 };
for (int index = 0; index < pins.Length; index++)
{
    int pin = pins[index];
    Console.WriteLine(pin);
}
```

Note:

`Length` is a **property** and not a **method**, which is why there are **no parentheses** when you call it.

5 Iterating Through an Array

The **`foreach`** *statement* ***enables*** *you to* ***iterate*** through the **elements of an array**.

For example, here's the preceding *for statement rewritten as an equivalent* **`foreach`** *statement:*

```
int[] pins = { 9, 3, 7, 2 };
foreach (int pin in pins)
{
   Console.WriteLine(pin);
}
```

5 Iterating Through an Array

The ***foreach*** *statement declares an iteration variable (in this example,* ***int pin****) that automatically* **acquires the value** of each element in the array.

The **type of this variable** must **match** **the type of the elements** in the array.

The ***foreach*** *statement* ***is the preferred way to iterate*** **through an array**; it expresses the intention of the code directly, and all of the ***for*** *loop* scaffolding drops away

5 Iterating Through an Array

Restrictions:

A **`foreach`** statement **always iterates through the whole array**. If you want to iterate through only a known portion of an array (for example, the first half) or to bypass certain elements (*for example*, *every third element*), it's easier to use a **`for`** statement.

A **`foreach`** statement always iterates from index **`0`** through index **`Length - 1`**. *If you want* to iterate backward, it's easier to use a **`for`** statement.

5 Iterating Through an Array

Restrictions:

If you need to modify the elements of the array, you'll have to use a **`for`** statement. This is because the iteration variable of the **`foreach`** statement is **a read-only copy** of each element of the array.

If the **body of the loop needs to know the index** of the element rather than just the value of the element, you'll have to use a **`for`** statement.

5 Iterating Through an Array

Example: Objects of an anonymous class

```
var names = new[]{ new { Name = "John", Age = 42 },
                   new { Name = "Diana", Age = 43 },
                   new { Name = "James", Age = 15 },
                   new { Name = "Francesca", Age = 13}
                 };
foreach (var familyMember in names)
{ // use the properties of the anonymous class
  Console.WriteLine("Name: {0}, Age: {1}",
                   familyMember.Name, familyMember.Age);
}
```

6 Copying Arrays

Arrays are reference types. This means that when you copy an array variable, you end up with two references to the same array instance.

```
int[] pins = { 9, 3, 7, 2 };
int[] alias = pins;
// alias and pins refer to
// the same array instance
```

In this example, **if you modify** the value at `pins[1]`, *the change will also be visible by reading* `alias[1]`.

6 Copying Arrays

If you want to **make a copy of the array** instance (the data on the heap) that an array variable refers to, you have to do two things.

First you need to create a new array instance of the same type and the same length as the array you are copying, as in this example:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
```

6 Copying Arrays

The **second thing** you need to do is **set the values inside the new array to the same values as the original array**.

You could do this by using a **`for`** *statement, as shown in this example:*

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
for (int i = 0; i < copy.Length; i++)
{
    copy[i] = pins[i];
}
```

6 Copying Arrays

Copying an array is actually a common requirement of many applications:

```
//using the CopyTo method
int[] pins = { 9, 3, 7, 2 };
// initialize copy first!
int[] copy = new int[pins.Length];
pins.CopyTo(copy, 0);
//using System.Array static method named Copy
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
Array.Copy(pins, copy, copy.Length);
```

6 Copying Arrays

Copying an array is actually a common requirement of many applications:

```
//using the CopyTo method
int[] pins = { 9, 3, 7, 2 };
// initialize copy first!
int[] copy = new int[pins.Length];
pins.CopyTo(copy, 0);
//using System.Array static method named Copy
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
Array.Copy(pins, copy, copy.Length);
```

7 Collection Classes in .NET

Arrays are useful, but they **have their limitations**. Arrays are only one way to collect elements of the same type.

The Microsoft .NET Framework provides several classes that also collect elements together in other specialized ways. These are the collection classes, and they live in the ***`System.Collections`*** *namespace and sub-namespaces*

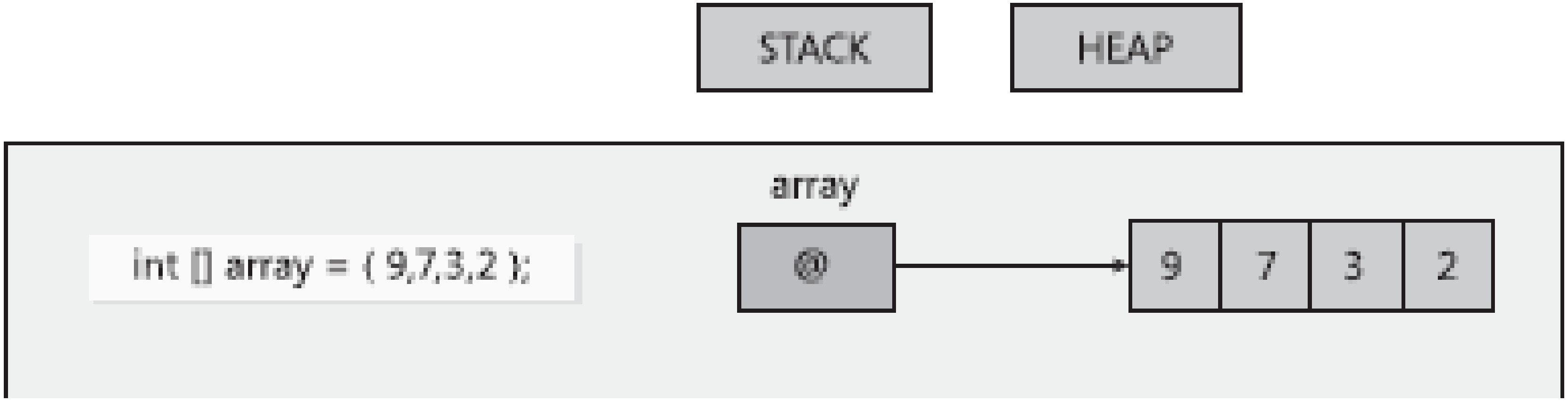
7 Collection Classes in .NET

The **basic collection classes** accept, hold, and return their elements **as objects-** that is, the **element type of a collection class** is an `object`.

To understand the implications of this, it is helpful to **contrast an array** of `int` *variables (*`int` *is a* ***value*** *type) with an* **array** *of* **objects** *(*`object` is a **reference type**).

Because `int` *is a **value type**, an array of* `int` *variables holds its* `int` *values* directly

7 Collection Classes in .NET



7 Collection Classes in .NET

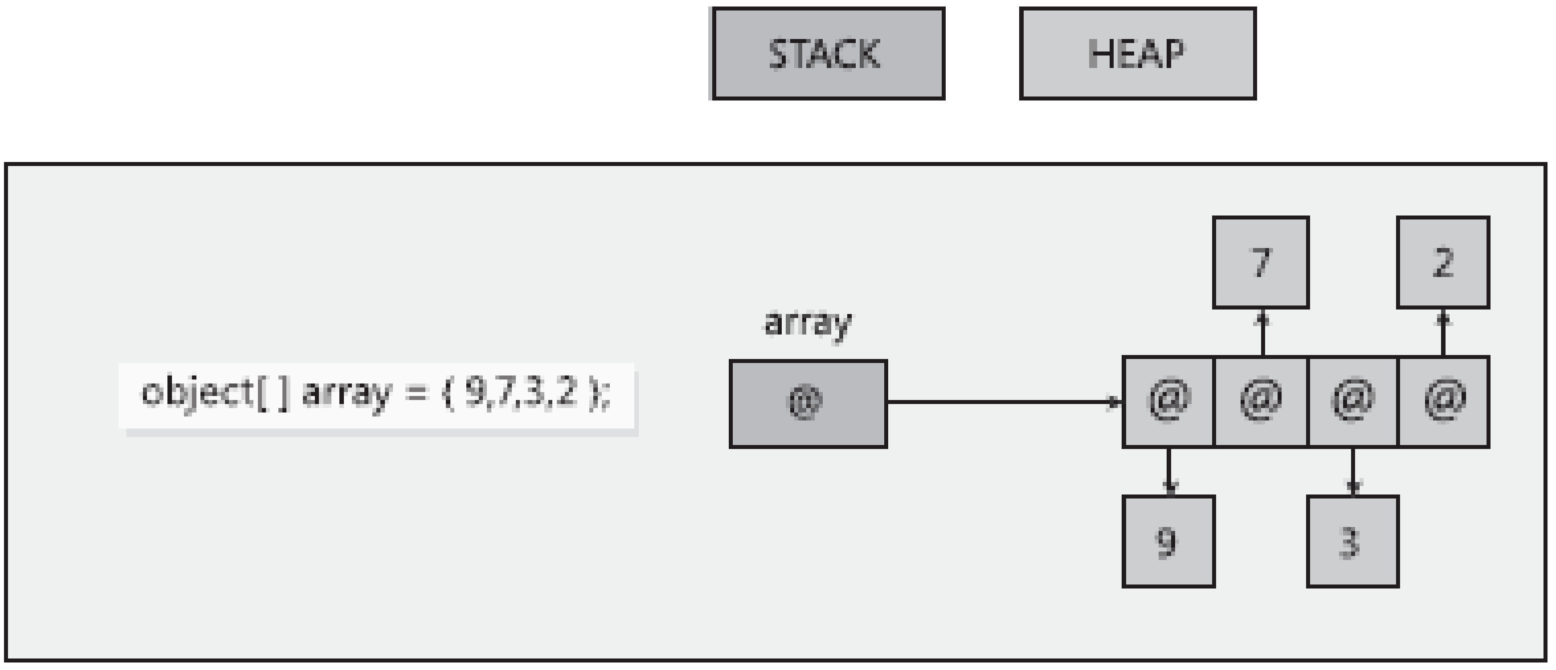
Now consider the effect **when the array is an array of `objects`.**

You can still add integer values to this array. (*In fact, you can* ***add values of any type*** *to it*.)

When you add an integer value, it is **automatically boxed**, and the array element (an `object` reference) **refers to the boxed copy** of the integer value

The element type of all the collection classes in this lecture is an `object`. Therefore, when you **insert** a **value into a collection**, it is **always boxed**, and when you **remove** a **value** from it, you must **unbox** it by **using a cast**.

7 Collection Classes in .NET



7.1 The `ArrayList` Collection Class

There are certain **occasions** when an **ordinary array can be too restrictive**:

- If you **want to resize an array**, you have to **create a new array, copy the elements** (leaving out some if the new array is smaller), and then update any references to the original array so that they refer to the new array.
- If you **want to remove an element** from an array, you have to move all the trailing elements up by one place. Even this doesn't quite work, because you end up with two copies of the last element.
- If you **want to insert an element** into an array, you have to move elements down by one place to make a free slot. However, you lose the last element of the array!

7.1 The `ArrayList` Collection Class

To overcome these restrictions using the `ArrayList` *class:*

- You **can remove an element** from an `ArrayList` by using its Remove method. The `ArrayList` automatically reorders its elements.
- You can **add an element** to the end of an `ArrayList` by using its `Add` method. You supply the element to be added. The `ArrayList` resizes itself if necessary.
- You can **insert an element** into the middle of an `ArrayList` by using its `Insert` method. Again, the `ArrayList` resizes itself if necessary.
- You **can reference an existing element** in an `ArrayList` object by using ordinary array notation, with square brackets and the index of the element.

7.1 The *ArrayList* Collection Class

An example that shows how you can **create, manipulate, and iterate** through the contents of an *ArrayList:*

```
using System;
using System.Collections;
...
ArrayList numbers = new ArrayList();
...
// fill the ArrayList
foreach(int number in
        new int[]{10, 9, 8, 7, 7, 6, 2, 1})
{
   numbers.Add(number);
}
```

7.1 The *ArrayList* Collection Class

```
// insert an element in the last - 1  position
// in the list, and move the last item up
// (the first parameter is the position;
// the second parameter is the value being inserted)
numbers.Insert(numbers.Count-1, 99);
// remove first element whose value is 7
// (the 4th element, index 3)
numbers.Remove(7);
// remove the element that's now
// the 7th element, index 6 (10)
numbers.RemoveAt(6);
```

7.1 The *ArrayList* Collection Class

```
// iterate remaining 10 elements using a for statement
for (int i = 0; i < numbers.Count; i++)
{
  int number = (int)numbers[i];
  // notice the cast, which unboxes the value
  Console.WriteLine(number);
}
...
// iterate remaining 10 using a foreach statement
foreach (int number in numbers) // no cast needed
{
  Console.WriteLine(number);
}
```

7.1 The *ArrayList* Collection Class

It is not recommended to you use the ArrayList class for new development. Instead use the generic List<T> class.

The ArrayList class is designed to hold heterogeneous collections of objects. However, it does not always offer the best performance. Instead, we recommend the following:

For a heterogeneous collection of objects, use the List<Object> (in C#) type.

For a homogeneous collection of objects, use the List<T> class.

7.2 The *Queue* Collection Class

The ***Queue*** *class implements a first-in, first-out (FIFO) mechanism.*

An element is inserted ***int*** the queue **at the back** (the ***enqueue*** operation) and is **removed** from the queue at **the front** (the ***dequeue*** operation).

Here's an example of a ***queue*** and its operations:

```
using System;
using System.Collections;
...
Queue numbers = new Queue();
```

7.2 The *Queue* Collection Class

```
foreach (int number in new int[]{9, 3, 7, 2})
{
 numbers.Enqueue(number);
 Console.WriteLine(number + " has joined the
                                              queue");
}
...
// iterate through the queue
foreach (int number in numbers)
{
      Console.WriteLine(number);
}
```

7.2 The *Queue* Collection Class

```
...
// empty the queue
while (numbers.Count > 0)
{
    int number = (int)numbers.Dequeue();
    // cast required to unbox the value
    Console.WriteLine(number +
                                 " has left the queue");
}
```

7.2 The *Queue* Collection Class

It is not recommended to use the Queue class for new development.

Instead, use the generic Queue<T> class.

7.3 The `Stack` Collection Class

The *Stack class implements a last-in, first-out (LIFO) mechanism.*

An element joins the stack at the top (the push operation) and leaves the stack at the top (the pop operation).

To visualize this, think of a stack of dishes: new dishes are added to the top and dishes are removed from the top, making the last dish to be placed on the stack the first one to be removed.

7.3 The *Stack* Collection Class

```
using System.Collections;
...
Stack numbers = new Stack();
...
// fill the stack
foreach (int number in new int[]{9, 3, 7, 2})
{
    numbers.Push(number);
    Console.WriteLine(number +
                      " has been pushed on the stack");
}
...
```

7.3 The *Stack* Collection Class

```
foreach (int number in numbers)
{
  Console.WriteLine(number);
}
...
// empty the stack
while (numbers.Count > 0)
{
    int number = (int)numbers.Pop();
    Console.WriteLine(number +
                      " has been popped off the stack");
}
```

7.3 The *Stack* Collection Class

It is not recommended to use the Stack class for new development.

Instead, we recommend that you use the generic System.Collections.Generic.Stack<T> class.   .

7.4 The `Hashtable` Collection Class

The **array** and **`ArrayList`** *types provide* ***a way to map an integer index to an element***.

You **provide** an **integer index** inside square brackets (for example, **`[4]`**), and you **get** **back the element** at index **`4`** (which is actually the fifth element).

However, sometimes you might want to provide **a mapping where the type you map from** is not an **`int`** *but rather some* ***other type***, such as **`string`**, **`double`**, *or* **`Time`**.

In other languages, this is often called an ***associative array***.

7.5 The `Hashtable` Collection Class

The ***`Hashtable`*** *class provides the functionality of an* ***associative array*** *by* **internally maintaining two object arrays**, *one for* the *keys you're mapping from and one for the values you're mapping to. When you insert a* ***`key/value`*** pair into a ***`Hashtable`***, *it automatically tracks which* ***`key`*** *belongs to which* ***`value`*** *and* **enables you to retrieve the value that is associated with a specified key** quickly and easily.

There are some important **consequences** of the design of the ***`Hashtable class`*** *as displayed on the next slide.*

7.5 The `Hashtable` Collection Class

`Hashtable class` *design*

A ***`Hashtable`*** **cannot contain duplicate keys**. If you call the ***`Add`*** method to add a key that is already present in the keys array, you'll get an exception.

You can, however, use the square brackets notation (indexer) to add a ***`key/value`*** pair (as shown in the following example), **without danger of an exception**, even if the key has already been added. The indexer ***`get`*** property returns ***`null`***, when the ***`key`*** **is missing.**

You can test whether a ***`Hashtable`*** already contains a particular ***`key`*** by using the ***`ContainsKey`*** method.

7.5 The `Hashtable` Collection Class

`Hashtable class` *design*

Internally, a ***`Hashtable`*** is a sparse data structure that **operates best when it has plenty of memory to work in**. The size of a ***`Hashtable`*** in memory **can grow quite quickly** as you insert more elements.

When you use a ***`foreach`*** statement to iterate through a ***`Hashtable`***, you get back a ***`DictionaryEntry`***. The ***`DictionaryEntry`*** class **encapsulates** to the ***`key`*** and ***`value`*** elements and **provides access** to these elements in both arrays through the ***`Key`*** property and the ***`Value`*** properties

7.5 The *Hashtable* Collection Class

```
using System;
using System.Collections;
...
Hashtable ages = new Hashtable();
...
// fill the Hashtable
ages[“John”] = 42;
ages[“Diana”] = 43;
ages[“James”] = 15;
ages[“Francesca”] = 13;
...
```

7.5 The *Hashtable* Collection Class

```
// iterate using a foreach statement
// the iterator generates a DictionaryEntry
// object containing a key/value pair
foreach (DictionaryEntry element in ages)
{
  string name = (string)element.Key; //object!
  int age     = (int)element.Value; //object!
  Console.WriteLine("Name: {0}, Age: {1}",
                                   name, age);
}
```

7.5 The *Hashtable* Collection Class

It is not recommended to use the Hashtable class for new development.

Instead, we recommend that you use the generic Dictionary<TKey,TValue> class.

Note:

The Dictionary will throw an exception if you try to reference a key that doesn't exist. The Hashtable will just return null. The reason is that the value might be a value type, which cannot be null. In a Hashtable the value was always Object, so returning null was at least possible.

7.6 The *SortedList* Collection Class

The ***SortedList*** *class is very similar to the* ***Hashtable*** *class in that it enables to* ***associate keys with values***. The **main difference** is that the **keys array is always sorted**.

Like the ***Hashtable*** *class, a* ***SortedList*** *cannot* ***contain duplicate*** *keys. When you use a* ***foreach*** statement to iterate through a ***SortedList***, *you get back a* ***DictionaryEntry***.

However, the ***DictionaryEntry*** *objects will be* ***returned sorted*** *by the* ***Key property***

7.6 The *SortedList* Collection Class

```
using System;
using System.Collections;
...
SortedList ages = new SortedList();
...
// fill the SortedList
ages["John"] = 42;
ages["Diana"] = 43;
ages["James"] = 15;
ages["Francesca"] = 13;
...
```

7.6 The *SortedList* Collection Class

```
// iterate using a foreach statement
// the iterator generates a DictionaryEntry
//  object containing a key/value pair
foreach (DictionaryEntry element in ages)
{
   string name = (string)element.Key;
   int age = (int)element.Value;
   Console.WriteLine(“Name: {0}, Age: {1}”,
                                              name, age);
}
```

7.6 The *SortedList* Collection Class

It is not recommended to use the SortedList class for new development.

Instead, use the generic System.Collections.Generic.SortedList<TKey,TValue> class.

8 Using Collection Initializers

The examples in the preceding subsections have shown you how to add individual elements to a collection by using the method most appropriate to that collection (*Add for an* ***`ArrayList`***, ***`Enqueue`*** *for a* ***`Queue`***, ***`Push`*** *for a* ***`Stack`***, *and so on).*

You can ***also initialize some collection*** types when you declare them, using a syntax very similar to that supported by arrays.

```
ArrayList numbers =
            new ArrayList(){10, 9, 8, 7, 7, 6};
```

8 Using Collection Initializers

For more complex collections such as **`Hashtable`** *that take key/value pairs, you can specify* each **`key/value`** pair as an **`anonymous`** type in the initializer list, like this:

```
Hashtable ages =
            new Hashtable(){
                      {"John", 42},
                      {"Diana", 43},
                      {"James", 15},
                      {"Francesca",13}
                          };
```

The first item in each pair is the key, and the second is the value.