Instructor: Bruce Reynolds

Introduction to Applications in C# Class 3

Concepts from Last Week

- Control Structures
  - Do / While / Switch
- Class vs. Instance
  - Instance methods, static methods
- Reference vs. Value Types
  - The new operator and null

Homework Review

- Follow Microsoft C# Coding Conventions
  - Published in the help and on MSDN
  - Use default Code Editor settings for indentation
    - Four-character indents, tabs saved as spaces
  - Identifier naming conventions
    - Mostly PascalCase, for types, methods, and fields
    - camelCase only for locals, method parameters, and private instance variables
    - No underscores

Concepts for Week 3

- Arrays
- File I/O
- Exception Handling
- Collections and Foreach

Arrays

Arrays

- An array is a data structure that contains multiple variables.
- The variables, called elements, are accessed through an index value.
- All elements in the array have the same type.
- The number of elements in the array is fixed.

Array Example

```
int[] threeInts;          //declaration of the array variable
threeInts = new int[3]; //initialization of the array variable
threeInts[0] = 3;         //initialization of the 0th element
threeInts[1] = 7;         //initialization of the 1st element
threeInts[2] = 23;        //initialization of the 2nd element
// accessing the three elements
Console.WriteLine("The three ints are {0}, {1}, and {2}.",
threeInts[0], threeInts[1], threeInts[2]);
```

Declaring An Array

- In its most basic form, the syntax is:
  - *type [] identifier;*
- The type can be a value type (int, bool), or reference type (Random, Uri).

Initializing An Array

- An array is a reference type, so you use new to initialize the array variable.

```
threeInts = new int[3];
```

- In this case, you've created an array with 3 elements, each of type int.
- The default value for int is `0`; so all three elements have a value of `0`. (But it's always better programming to explicitly initialize variables.)

Accessing and Initializing Elements

- You must also initialize each element of the array.
- Individual elements of the array are accessed through the array access operator [] (AKA square brackets):

```
threeInts[2] = 23;
```

- The elements are indexed 0, 1, 2, …
- Trying to access an element beyond (higher) than what was declared is an error. (And this is where it's easier than C++.)

Initializing in this Example

```
threeInts[0] = 3;        //initialization of the 0th element
threeInts[1] = 7;        //initialization of the 1st element
threeInts[2] = 23;       //initialization of the 2nd element
```

Using the Debugger

- Set a breakpoint on the first line of code (F9).
- Run the code with debugging (F5).
- Step over each line of code (F10).
- You can examine each element:
  - In the code editor.
  - In the Locals window.

Accessing Elements

- You can also access elements through an expression:

```
int number = 2;
threeInts[number] = 4;
threeInts[0 + 1] = 25;
```

Accessing in this Example

```
Console.WriteLine("The three ints are {0}, {1}, and {2}.",
    threeInts[0], threeInts[1], threeInts[2]);
```

Using the Array Initializer

- You can use the array initializer:

```
int[] threeInts = new int[3] { 3, 7, 23 };
for (int index = 0; index < 3; index++)
{
    Console.WriteLine(threeInts[index]);
}
```

- Or this shorthand notation:

```
int[] threeInts = { 3, 7, 23 };
for (int index = 0; index < 3; index++)
{
    Console.WriteLine(threeInts[index]);
}
```

Arrays and the For Statement

- Because array elements are indexed sequentially, the for statement is a natural way to access the elements in the array.

```
int[] threeInts = { 3, 7, 23 };
//print the values
for (int index = 0; index < 3; index++)
{
    Console.WriteLine(threeInts[index]);
}
```

Arrays and the For Statement

- You are less likely to make an indexing error (AKA off-by-one error) if you use the Length property of the array. (This only works for one-dimensional arrays. More on that later.)

```
int[] threeInts = { 3, 7, 23 };
//total the values
int total = 0;
for (int index = 0; index < threeInts.Length; index++)
{
    total += threeInts[index];
}
Console.WriteLine("The total is {0}.", total);
```

Demo: Array and For Statement

- You can set the element values in a for statement:

```
int[] threeInts = new int[3];
//initialize with the multiples of 5
for (int index = 0; index < 3; index++)
{
    threeInts[index] = index * 5;
}

for (int index = 0; index < 3; index++)
{
    Console.WriteLine(threeInts[index]);
}
```

Arrays and .NET

- An array is an instance of the System.Array class.
  - http://msdn.microsoft.com/en-us/library/system.array.aspx
- You get some interesting features:
  - IndexOf
  - Sort
  - Reverse
  - Length
  - Resize

Sorting an Array

- Some types have a sorting algorithm built into them (IComparable). For these types, you can sort the array, using the **static** Sort method:

```
//sort and print the values
Array.Sort(threeInts);
for (int index = 0; index < 3; index++)
{
    Console.WriteLine(threeInts[index]);
}
```

Reverse an Array

```
//reverse and print the values
Array.Reverse(threeInts);
for (int index = 0; index < 3; index++)
{
    Console.WriteLine(threeInts[index]);
}
```

Arrays of Reference Types

- Declaration and initialization of the array are the same:

```
Uri[] links = new Uri[3];
```

- Initialization of the elements take into account the reference type:

```
Uri microsoft = new Uri("http://www.microsoft.com");
links[0] = microsoft;
links[1] = new Uri("http://www.washington.edu");
links[2] = new Uri("http://www.amazon.com");
```

Arrays of Reference Types

- To access an element of the array, you can set a variable, or access it through the [] operator:

```
Uri[] links = new Uri[3];
links[0] = new Uri("http://www.microsoft.com");
links[1] = new Uri("http://www.washington.edu");
links[2] = new Uri("http://www.amazon.com");
Uri firstLink = links[0];
Console.WriteLine(firstLink.AbsoluteUri);
```

Arrays of Reference Type

- You can use this shortcut to initialize:

```
Uri[] links = new Uri[] {
    new Uri("http://www.microsoft.com"),
    new Uri("http://www.washington.edu"),
    new Uri("http://www.amazon.com") };
}
```

Arrays of Strings

- Though the string is a class, arrays are declared and initialized similarly to integers.
  - This is because strings are immutable, and they have some special features built in.
- Declare, initialize, and access:

```
string[] firstNames = { "George", "Thomas", "Abraham" };
string[] lastNames = { "Washington", "Jefferson", "Lincoln" };
for (int index = 0; index < firstNames.Length; index++)
{
    Console.WriteLine("President {0} {1}",
        firstNames[index], lastNames[index]);
}
```

N-Dimensional Arrays

- Arrays can have any number of dimensions.
- Consider the 2-dimensional array, which can be thought of as a table of rows and columns.

2-Dimensional Array

- Suppose you declare this:

```
int[,] table = new int[2,2]
```

- You can think of the element values like this:

table[0,0]	table[0,1]
table[1,0]	table[1,1]

Demo: Addition Facts

- Code that prints out the addition facts:

```
int[,] additionTable = new int[10, 10];
// set the values
for (int row = 0; row < 10; row++)
{
    for (int column = 0; column < 10; column++)
    {
        additionTable[row, column] = row + column;
    }
}

// print the table
for (int row = 0; row < 10; row++)
{
    for (int column = 0; column < 10; column++)
    {
        Console.Write(additionTable[row, column] + "\t");
    }
    Console.WriteLine();
}
```

Initializing a 2-D Array

- Use the debugger to understand this code:

```
int[,] squares = {
    {1,1}, {2,4}, {3,9}, {4, 16} };

// print the array
for (int row = 0; row < 4; row++)
{
    Console.WriteLine("The square of {0} is {1}.",
        squares[row, 0], squares[row, 1]);
}
```

Arrays of Arrays

- You can declare and initialize arrays of arrays:

```
int[][] squares = new int[3][];
squares[0] = new int[] { 1, 1 };
squares[1] = new int[] { 2, 4 };
squares[2] = new int[] { 3, 9, 27 };

for (int row = 0; row < 3; row++)
{
    Console.WriteLine("The square of {0} is {1}.",
        squares[row][0], squares[row][1]);
}
```

- This is a jagged array, because all the arrays are not of the same length.

File I/O (Input/Output)

File I/O

- Before you pick a file I/O strategy, you first have to define the file type:
  - Text file (.txt)
  - Word file (.doc)
  - Database file (Access, SQL)
  - XML file (.xml)
  - HTML file (.htm)
- We'll look at text files.

Reading Files – The Most Basic

- At its most basic, you can get all the text from a file in one string using this code:

```
string text = System.IO.File.ReadAllText(
    @"C:\Users\robin\Documents\SomeText.txt");
Console.Write(text);
```

- Hint: Use the Copy Full Path feature in the code editor to get the file name.

Special Characters in String Literals

- Consider this code that we wrote:

```
string text = System.IO.File.ReadAllText(
    @"C:\Users\robin\Documents\SomeText.txt");
```

- C# has several escape sequences that you can put in strings:
  - \t for Tab
  - \n for new line
  - \" for double quote
- If you want a \ in your string, you need create a verbatim string with @.
  - Or you could use \\.

Errors in File I/O

- While it takes only one line of code to read a file, many, many things can go wrong:
  - The file may not exist.
  - You might try to write a read-only file.
  - You might not have access to the file.
  - The file may be corrupt.
- Look at the Exceptions section of the documentation to find all the things that can go wrong:
  - http://msdn.microsoft.com/en-us/library/ms143368.aspx

Exception Handling

- The runtime **throws** an **exception** when something exceptional happens.
- When an exception is thrown, execution stops unless the exception is **handled** by a **try statement**, shown on next slide.
  - If you don't have a try statement and an exception is thrown, your application stops running.
  - You want to decide how to handle exceptions before you let anyone else run your application. Your intent is to "fail gracefully."

The Try…Catch Statement

- The basic grammar is:
  - `try` *block catch-clauses finally-clause*
- An empty try statement:

```
try
{
    // code that you want to run here
}
catch (Exception ex)
{
    // what to do if something goes wrong
}
finally
{
    // what to do always
}
```

Demo: File I/O with Try…Catch

- Run this code:

```
try
{
    string text = System.IO.File.ReadAllText(
        @"C:\Users\robin\Documents\SomeText.txt");
    Console.Write(text);
}
catch (Exception ex)
{
    // don't do this in real code
    Console.WriteLine(ex.Message);
}
```

- Try to generate these exceptions:
  - ArgumentException
  - FileNotFoundException

Try…Catch

- The catch clause is optional.
- The finally clause is optional.
- You can have multiple catch clauses.
  - Using multiple catch clauses allows you to catch specific exceptions, if the code you are running can throw more than one type of exception.

Exceptional Events

- Exceptions are expensive in programming time.
- It's better to anticipate the problem than use a try statement.
- For example, the Int32.TryParse lets you test whether Int32.Parse will not fail:

```
int value = 0;
bool success = int.TryParse("hello", out value);
if (success)
{
    Console.WriteLine(value);
}
else
{
    Console.WriteLine("Can't parse \"hello\"");
}
```

File I/O, Continued

- For many applications, you will want to break the text from the file into strings. In that case, use the ReadAllLines method. It returns an array of strings.

Writing Text to a File

- You can write a string or multiple lines.

```
// write a single string to the file
System.IO.File.WriteAllText(
    @"C:\Users\robin\Documents\SomeText.txt",
    "here is some text");

// write an array of strings
string[] firstNames = { "George", "Thomas", "Abraham" };
System.IO.File.WriteAllLines(
    @"C:\Users\robin\Documents\SomeText.txt", firstNames);
```

Writing Text to a File

- You can create a new file, or append to an existing one.

```
// append to an existing file
System.IO.File.AppendAllText(
    @"C:\Users\robin\Documents\SomeText.txt", "John");

// write the text
Console.WriteLine(System.IO.File.ReadAllText(@"C:\Users\robi
n\Documents\SomeText.txt"));
```

Using StreamReader

```
string fileName = "demo.txt";
try
{
    FileStream input = new FileStream(
        fileName, FileMode.Open, FileAccess.Read);
    StreamReader reader = new StreamReader(input);
    string line;
    while (( line = reader.ReadLine()) != null )
    {
        Console.WriteLine(line);
    }
}
catch(IOException)
{
    Console.WriteLine("Cannot open file {0}.", fileName);
}
```

Collections

The Foreach Statement

- The foreach statement is used to iterate through the members of an array or collection (more later), without using an index.
- The grammar is:

  `foreach` (*type identifier* `in` *expression*)
  embedded-statement

- *expression* must return a collection type, that is, it implements IEnumerable.

Foreach and Arrays

- You can use foreach with an array:

```
string[] firstNames = { "George", "Thomas", "Abraham" };
foreach (string name in firstNames)
{
    Console.WriteLine(name);
}
```

- No counters, no indexing!
  - But, you don't have access to the index of the elements.

Collections

- The .NET Framework includes several classes that are used to store a collection of values, much the same as an array stores multiple values.
- The classes are in two namespaces:
  - System.Collections
    - The most popular class is probably ArrayList.
  - System.Collections.Generic
    - These classes are type-safe, and are the ones we'll be using.

The List<T> Class

- The List<t> class is a data structure that holds a collection of strongly typed elements.
- The elements can be accessed by indexing.
- The elements can be searched and sorted.
- Elements can be added and removed without resizing the list.

Using the List<T> Class

```
List<int> multiples = new List<int>();
multiples.Add(5);
multiples.Add(10);
multiples.Add(15);
multiples.Add(20);

Console.WriteLine(multiples[2]);
```

DEMO: Foreach Example

```
List<int> multiples = new List<int>();
multiples.Add(5);
multiples.Add(10);
multiples.Add(15);
multiples.Add(20);

foreach (int multiple in multiples)
{
    Console.WriteLine(multiple);
}
```

Removing Elements from a List<T>

```
List<int> multiples = new List<int>();
multiples.Add(5);
multiples.Add(10);
multiples.Add(15);
multiples.Add(20);
multiples.Remove(15);

foreach (int multiple in multiples)
{
    Console.WriteLine(multiple);
}
```

Collections Initializers

- You can use this shortcut:

```
List<int> multiples = new List<int> { 5, 10, 15, 20 };

foreach (int multiple in multiples)
{
    Console.WriteLine(multiple);
}
```

Other Collections

- Stack
- Queue
- LinkedList
- Dictionary
- SortedDictionary

Reading 3

- Sample: http://msdn.microsoft.com/en-us/library/hhoeh6yz.aspx
- Spec: Chapters 12 and 16
- Deitel & Deitel
  - Chapter 8: Arrays
  - Chapter 13: Exception Handling
  - Chapter 17: Files and Streams

Assignment 3