3.7 Simple Exception Handling

- An exception is an unexpected error that happens while a program is running
- If an exception is not handled by the program, the program will abruptly halt
- C# allows you to write codes that responds to exceptions. Such codes are known as exception handlers.
- In C# the structure is called a `try-catch` statement

```
try { }
catch { }
```

- The try block is where you place the statements that could have exception
- The catch block is where you place statements as response to the exception when it happens

Exception Handling

- Types of catch:
  - catch without parentheses:
    - `catch { }`
    - This catch block will catch all kinds of exceptions that are produced by the try block
  - `catch(ExceptionType exceptionVariable) { }`
    - This catch block will only catch exceptions of the specified type. If you use the base class Exception, this will also catch all exceptions.
- You can chain catch blocks (as many as you want) and they will be executed from top to bottom.
- If you are expecting multiple types of exceptions to be thrown, it's always better to catch specific exceptions and handle them case by case instead of having a general exception handling block.
- When chaining multiple catch blocks remember to order them starting with the more specific to the more general catch statements. The first catch encountered that applies will execute, even if a more specific one exists below it.

Exception Handling

Exceptions:

- When an exception is caught you can make use of the caught exception object (an instance of the Exception class or a class that inherits from Exception) to get more insight into what went wrong.
- Some useful properties:
  - Message: contains the exception message, a short description of the error that caused the exception.
  - StackTrace: shows a trace of method calls from the current block up until the statement where the error occurred. This helps trace the original statement where the exception happened and simplifies debugging.

Exception Handling

Exceptions:

- When an exception is caught you can make use of the caught exception object (an instance of the Exception class or a class that inherits from Exception) to get more insight into what went wrong.
- Some useful properties:
  - Message: contains the exception message, a short description of the error that caused the exception.
  - StackTrace: shows a trace of method calls from the current block up until the statement where the error occurred. This helps trace the original statement where the exception happened and simplifies debugging.

The finally block:

- When an exception is thrown in the try block, any remaining statements following that statement will now be executed, and program control is transferred to the catch block. In some cases you need to make sure some code is ALWAYS executed, whether or not there was an exception. This code is placed in the finally block.
- Finally can be used with a try block or with try and catch blocks.

Throwing an Exception

- In the following example, the user may entered invalid data (e.g. null) to the `milesText` control. In this case, an exception happens (which is commonly said to "throw an exception").
- The program then jumps to the catch block.
- You can use the following to display an exception's default error message:

```
catch (Exception ex)
{
  MessageBox.Show(ex.Message);
}
```

```
try
{
 double miles;
 double gallons;
 double mpg;

 miles = double.Parse(milesTextBox.Text);
 gallons = double.Parse(gallonsTextBox.Text);
 mpg = miles / gallons;
 mpgLabel.Text = mpg.ToString();
}
catch
{
 MessageBox.Show("Invalid data was entered.");
}
```

AUBURN UNIVERSITY

Throwing an Exception

- Just like you can catch an exception, you can also throw your own exception.
- To throw your own exception use the following format:
- **`throw exceptionObject`**
- Example:

```
throw new Exception("Exception Message");
```

7.1 Value Types and Reference Types

- The data types in C# and the .NET Framework fall into two categories: **values types** and **reference types**
- A variable that is used to hold a value, such as 23, 15.87, "Hello", etc. is a value type of variable
  - They actually hold data
- A variable that is used to reference an object is commonly called a reference variable
  - Reference variables can be used only to reference objects. They do not hold data.

How a Value Type Works

- When you declare a value type variable, the compiler allocates a chunk of memory that is big enough for the variable
- The memory that is allocated for a value type variable is the actual location that will hold the value assigned to the variable
- When you are working with a value type, you are using a variable that holds a piece of data
- Value type of variable actually holds the data

How a Reference Type Works

AUBURN UNIVERSITY

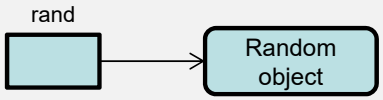
- When you work with a reference type, you use two things:
  - An object that is created in memory
  - A variable that references the object
- The object that is created in memory holds data. You need a way to refer to it.
  - A variable is then created to hold a value called **reference**
  - A reference variable does not hold an actual piece of data, it simply refers to the data
  - A reference type links the variable that holds actual data to the object
- If a kite is the object, then the spool of string that holds the kite is the reference

Creating a Reference Type

- Two steps are typically required:
  - Declare a reference variable
  - Create an object and associate it with the reference variable
- An example is the **`Random`** class

```
Random rand = new Random();
```

  - The `Random rand` part declares a variable named `rand`
  - The `new Random()` part creates an object and returns a reference to the object
  - The = operator assigns the reference that was returned from the `new` operator to the `rand` variable



7.2 Array Basics

- An **array** allows you to store a group of items of the same data type together in memory
- Processing a large number of items in an array is usually easier than processing a large number of items stored in separated variables
  - This is because each variable can only hold one data:

```
int number1 = 99;
int number2 = 100;
```

  - Each variable is a separated item that must be declared and individually processed
  - Variables are not ideal for storing and processing lists of data

Array Basics (Cont'd)

- Arrays are reference type objects
- To create an array, you need to:
  - declare a reference type object
  - create the object and associate it with the reference variable
- In C#, the generic format to declare a reference variable for an array is:

  DataType`[]` *arrayName*;
- For example, `int[] numbersArray;`
- The generic format to create the array object and associate it with the variable is:

  arrayName `=` `new` *DataType*`[`*ArraySize*`];`
- The *new* keyword creates an object in memory; it also returns a reference to that array. For Example, `numbersArray = new int[6];`

Array Basics (Cont'd)

- In the previous example, there are two statements:

```
int[] numbersArray;
numbersArray = new int[6];
```

- There two statements can be combined into one statement:

```
int[] numbersArray = new int[6];
```

- You can create arrays of any data type

```
double[] temperatures = new double[100];
decimal[] prices = new decimal[50];
string[] nameArray = new string[1200];
```

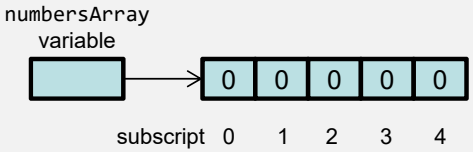
- An array's size declarator must be a positive integer and can be a literal value

```
const int SIZE = 6;
int[] numbersArray = new int[SIZE];
```

It is a preferred practice to used a named constant as size declarator

Array Elements

- The storage locations in an array are known as **elements**
- In memory, an array's elements are located in consecutive memory locations
- Each element in an array is assigned a unique number known as a **subscript**
  - Subscripts are used to identify specific elements in an array Subscripts start with 0. The element has subscript 0, the nth has n-1.



When you create a numeric array in C#, its elements are set to the value of 0 by default

AUBURN UNIVERSITY

Working with Array Elements

- Given the following code, you can access each individual element by using their subscript

```
const int SIZE = 5;
int numbersArray = new int[5];
numbersArray[0] = 20;
numbersArray[1] = 20;
numbersArray[2] = 20;
numbersArray[3] = 20;
numbersArray[4] = 20;
```

- To get the value of the 3rd element, for example, use:

```
numbersArray[2]
```

Array Initialization

- When you create an array, you can optionally initialize it with a group of values

```
const int SIZE = 5;
int[] numbersArray = new int[SIZE] { 10, 20, 30, 40, 50 };
```

- Or simply,

```
int[] numbersArray = new int[] { 10, 20, 30, 40, 50 };
```

- And even,

```
int[] numbersArray = { 10, 20, 30, 40, 50 };
```

- All three produce the same results

Using a Loop to Step Through an Array

- Arrays commonly use int as subscript. You can then create a loop to step through the array. For example,

```
const int SIZE = 3;
int[] myValues = new int[SIZE];
for (int index = 0; index < SIZE; index++)
{
     myValues[index] = 99;
}
```

- This example assigns 99 to each element as value
- Notice that the number of iterations cannot exceed the array size; otherwise, an exception will be thrown at runtime

```
for (int index = 0; index <= SIZE; index++) // will cause exception
{ … }
```

The Length Property

- In C#, all arrays have a **Length** property that is set to the number of elements in the array

```
double[] temperatures = new double[25];
```

- The output of the following is 25

```
MessageBox.Show(temperatures.Length.ToString());
```

- The Length property can be useful when processing the entire array

```
for (int index =0; index < temperatures.Length; index++)
{
     MessageBox.Show(temperatures.Length.ToString());
}
```

AUBURN UNIVERSITY

Using the `foreach` Loop with Arrays

- C# provides a special loop called ***`foreach`*** to simplify array processing
- The *`foreach`* loop is designed to work a temporary, read-only variable known as iteration variable. A generic format is:

```
foreach (Type VariableName in ArrayName)
{
    statement(s);
}
```

- *`Type`* is the data type of the array
- *`VariableName`* is the name of the temporary iteration variable
- **`in`** is a keyword that must appear
- *`ArrayName`* is the name of array to process

Using the foreach Loop with Arrays

```
int[] numbers = { 3, 6, 9 };

foreach (int val in numbers)
{
    MessageBox.Show(val.ToString());
}
```

Copyright © 2017 Pearson Education, Inc.

7.3 Working with Files and Array

- The book demonstrates how to write an array's contents to a file. Here is the code:

```
int[] numbers = { 10, 20, 30, 40, 50 };
StreamWriter outputFile;
outputFile = File.CreateText("Values.txt");
for (int index = 0; index < numbers.Length; index++)
{
   outputFile.WriteLine(numbers[index]);
}
outputFile.Close();
```

Reading Values from a File to an Array

- The book demonstrates how to read values from a file and store them in an array. Here is the code:

```
const in SIZE= 5;
int[] numbers = new int[SIZE];
int index = 0;
StreamReader inputFile;
inputFile = File.OpenText("Values.txt");

while (index < numbers.Length && !inputFile.EndOfStream)
{
   numbers[index] = int.Parse(inputFile.ReadLine());
   index++;
}
inputFile.Close();
```

7.4 Passing Array as Argument to Methods

- An entire array can be passed as one argument to a method

```
string[] people = { "Bill", "Jill", "Phil", "Will" };
ShowArray(people);
private void ShowArray(string[] strArray)
{
    foreach (string str in strArray)
    {
        MessageBox.Show(str);
    }
}
```

- It can also be passed individually

```
for (int index = 0; index < people.Length; index++)
{
   ShowPeople(people[index]);
}
```

```
private void ShowPeople(string str)
{
    MessageBox.Show(str);
}
```

7.5 Some Useful Array Algorithms

- The Sequential Search uses a loop to sequential step through an array, starting with the first element. For example,

```
bool found = false;
int index = 0;
int position = -1;
while (!found && index < sArray.Length)
{
    if (sArray[index] == value)
    {
        found = true;
        position = index;
    }
    index++;
}
```

- This code searches a string array for a special value. If the value is found its position is return; otherwise -1 is returned.
- This algorithm is not efficient

Useful Array Algorithms (Cont'd)

- Copying an array – create a second array and copy the individual element of the source array to the target array

```
for (int index = 0; index < firstArray.Length; index++)
{
   secondArray[index] = firstArray[index];
}
```

- Comparing Arrays – You must compare each element of the arrays

```
if (firstArray.Length != secondArray.Length) { return false; }
if (firstArray[index] != secondArray[index]) { return false; }
```

- Totaling the value of an array – Use a loop with an accumulator variable

```
for (int index = 0; index < units.Length; index++)
{
   total += units[index];
}
```

To find the average, use:

```
total / units.Length;
```

Finding the Highest Value in an Array

- Create a variable to hold the found highest value
- Start with assuming the element 0 is the highest
- Use a loop to step through element 1 and the rest
- Each time the loop iterates, it compares an array element to the variable holding the found highest value. If the array element is greater than the found highest value, assign the value of that array element to the variable holding the found highest value.
- When the loop finished, the highest value is found

```
int[] numbers = { 8, 1, 12, 6, 2, … }
int highest = numbers[0];
for (int index=1; index < numbers.Length; index++)
{
   if (numbers[index] > highest)
   {
     highest = numbers[index]
   }
}
```

Finding the Lowest Value in an Array

- Similarly, the lowest can be found

```
int[] numbers = { 8, 1, 12, 6, 2, … }
int lowest = numbers[0];

for (int index=1; index < numbers.Length; index++)
{
   if (numbers[index] < lowest)
   {
     lowest = numbers[index]
   }
}
```

AUBURN UNIVERSITY

7.6 Advanced Algorithms for Sorting and Searching

- The selection sort works by:
  - locating the smallest value and moving it to element 0
  - Then find the second smallest and move it to element 1
  - Continue the process until all the elements have been placed in their proper order

7.6 Advanced Algorithms for Sorting and Searching

- This algorithm requires swapping of array elements which can be done by the following:

```
private void Swap(ref int a, ref int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

7.6 Advanced Algorithms for Sorting and Searching

- The selection sort algorithm:

```
int minIndex, minValue;
for (int startScan = 0; startScan < iArray.Length - 1; startScan++)
{
     minIndex = startScan;
     minValue = iArray;
     for (int index = startScan + 1; index < iArray.Length; index++)
     {
          if (iArray[index] < minValue)
          {
               minValue = iArray[index];
               minIndex = index;
          }
      }
     Swap(ref iArray[minIndex], ref iArray[starScan]);
}
```

The Binary Search Algorithm

- A clever algorithm that is much more efficient than the sequential search. Here is the pseudocode for a method that performs a binary search on an array:

```
Method BinarySeach(array, searchValue)
   set first to 0
   set last to the last subscript in the array
   set position to -1
   set found to false

   While found is not true and first is less than or equal to last
      set middle to the subscript half way between array[first] and array [last]
      If array[middle] equals searchValue
         set found to true
         set position to middle
      Else If array[middle] is greater than searchValue
         set last to middle-1
      Else
         set first to middle + 1
      End If
   End While
   Return position
End Method
```

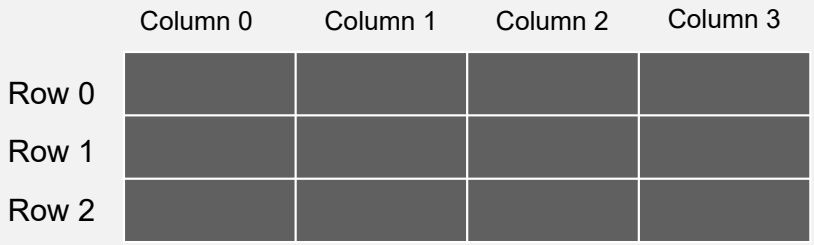
Sample Code

```
int first = 0;
int last = iArray.Length – 1;
int middle;
int position =  -1;
bool found = false;
while (!found && first <= last)
{
    middle = (first + last) / 2;
    if (iArray[middle] ==  value)
    {
        found = true;
        position = middle;
    }
   else if (iArray[middle] > value)
   {
      last = middle – 1;
   }
   else
   {
      first = middle + 1;
   }
}
```

7.7 Two-Dimensional Arrays

AUBURN UNIVERSITY

- A two-dimensional (2D) array is like several identical arrays put together
  - Can hold (store) multiple sets of data
  - Can be illustrated by a table with **rows** and **columns**
    - Row - horizontal
    - Column – vertical
  - Rows and columns are numbered 0, 1, 2, etc.



Declaring a 2D Array

- A 2D array requires two size declarators
  - The first for rows, the second for columns

```
const int ROWS = 3;
const int COLS = 4;
double[,] scores = new double[ROWS, COLS];
```

- This declares that the `scores` variable references a 2D array
  - 3 defines size of rows and 4 defines size of columns
  - Size declarators are separated by a comma (,)

	Column 0	Column 1	Column 2	Column 3
Row 0	Scores[0,0]	Scores[0,1]	Scores[0,2]	Scores[0,3]
Row 1	Scores[1,0]	Scores[1,1]	Scores[1,2]	Scores[1,3]
Row 2	Scores[2,0]	Scores[2,1]	Scores[2,2]	Scores[2,3]

AUBURN UNIVERSITY

Accessing Elements in a 2D Array

- Each element in a 2D array has two subscripts
  - One for its rows and one for its columns:
  - Elements in row 0 are:

```
score[0,0]
score[0,1]
score[0,2]
score[0,3]
```

  - Elements in row 1 are:

```
score[1,0]
score[1,1]
score[1,2]
score[1,3]
```

	Column 0	Column 1	Column 2	Column 3
Row 0	Scores[0,0]	Scores[0,1]	Scores[0,2]	Scores[0,3]
Row 1	Scores[1,0]	Scores[1,1]	Scores[1,2]	Scores[1,3]
Row 2	Scores[2,0]	Scores[2,1]	Scores[2,2]	Scores[2,3]

Assigning Values to Elements

- To assign values to the elements, use:

```
scores[0,0] = 71;
scores[0,1] = 67;
scores[0,2] = 45;
scores[0,3] = 82;
…
scores[2,3] = 61;
```

	Column 0	Column 1	Column 2	Column 3
Row 0	71	67	45	82
Row 1	54	87	90	68
Row 2	80	75	39	61

- To implicit size and initialization of 2D array, use:

```
int[,] scores = { {71, 67, 45, 82},
                  {54, 87, 90, 68},
                  {80, 75, 39, 61} };
```

AUBURN UNIVERSITY

Use Nested Loops to Process 2D Arrays


```
//To assign a random number to each element

const int ROWS = 3;
const int COLS = 4;
int[,] scores = new int[ROWS, COLS];

Random rn = new Random();

for (int row = 0; row < ROWS; row++)
{
   for (int col = 0; col < COLS; col++)
   {
      score[row, col] = rn.Next(100);
   }
}
```

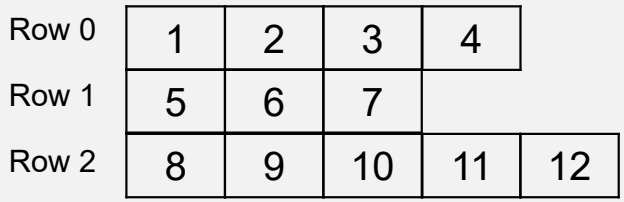
Use Nested Loops to Process 2D Arrays


```
//To add value of each element to a ListBox

const int ROWS = 3;
const int COLS = 4;
int[,] scores = new int[ROWS, COLS];

for (int row = 0; row < ROWS; row++)
{
   for (int col = 0; col < COLS; col++)
   {
      listBox.Items.Add(score[row, col].ToString());
   }
}
```

Jagged Arrays

- A jagged array is special type of 2D array
  - rows in a jagged array can have different lengths

```
int[][] jagArr = new int[3][];
jagArr[0] = new int[4] { 1, 2, 3, 4 } // 4 columns
jagArr[1] = new int[3] { 5, 6, 7 } // 3 columns
jagArr[2] = new int[5] { 8, 9, 10, 11, 12 } // 5 columns
```



AUBURN UNIVERSITY

Jagged Arrays

- To access an element, use:

```
MessageBox.Show(jagArr[1][2].ToString());
```

- To assign a value to row 0, column 3, use:

```
jarArr[0][3] = 99;
```

Row 0	1	2	3	99	
Row 1	5	6	7		
Row 2	8	9	10	11	12

AUBURN UNIVERSITY

Use Nested Loops to Process Jagged Arrays

- Jagged arrays have a **Length** property
  - Holds the number of rows
- Each row also has a Length property
  - Holds the number of columns
- The following nested loops display all the values of a jagged array:

```
for (int row = 0; row < jagArr.Length; row++)
{
   for (int col = 0; col < jagArr[row].Length; col++)
   {
      MessageBox.Show(jagArr[row][col].ToString());
   }
}
```

The List Collection

- The C# **`List`** is a class in the .NET Framework that is similar to an array with the following advantages:
  - A `List` object does not require size declaration
    - Its size is automatically adjusted
  - You can add or remove items
- Syntax to create a `List` is:

```
List<DataType> ListName = new List<DataType>();
```

  - For example,

```
List<string> names = new List<string>();  // a List that holds strings
List<int> numbers = new List<int>();      // a List that holds integers
```

AUBURN UNIVERSITY

Add or Remove Items

- To add items, use the **Add** method

```
List<string> nameList = new List<string>();
nameList.Add("Chris");
nameList.Add("Bill");
```

- To insert an item, use the **Insert** method to insert an item at a specific index

```
nameList.Insert("Joanne", 0);
```

- To remove items, use:

•**Remove** method: remove an item by its value

```
nameList.Remove("Bill");
```

•**RemoveAt** method: remove an item at a specific index in a List

```
nameList.RemoveAt(0);
```

Initializing a `List` Implicitly

- To initialize a `List` implicitly, simply defines its items when you declare it

```
List<int> numberList = new List<int>() { 1, 2, 3 };
List<string>nameList = new List<string>() { "Christ", "Kathryn", "Bill" }
```

- The **Count** property holds the number of items stored in the `List`
  - Useful in accessing all items in a `List`

```
for (int index = 0; index < nameList.Count; index++)
{
   MessageBox.Show(nameList[index]);
}
```