C# Basics

Array & List

Value Types and Reference Types

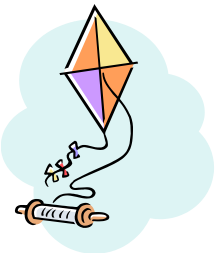
- The data types in C# and the .NET Framework fall into two categories: **values types** and **reference types**
- A variable that is used to hold a value, such as 23, 15.87, “Hello”, etc. is a value type of variable
  - They actually hold data
- A variable that is used to reference an object is commonly 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

- 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 site is the reference

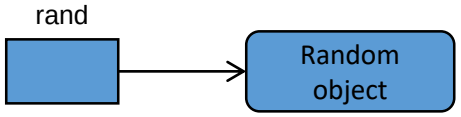


Creating a Reference Type

- 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



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

- 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

- 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

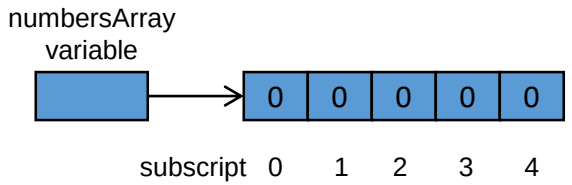
```
double[] temperatures = new double[100];
decimal[] prices = new decimal[50];
string[] nameArray = new string[1200];
```

- 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];
```

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

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());
}
```

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);
}
```

```
int[] numbers = { 3, 6, 9 };
foreach (int val in numbers)
{
    MessageBox.Show(val.ToString());
}
```

- *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

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
```

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]);
}
```