

.Net Training

Intermediate Basics of .Net (C#) Part I



Agenda

- Control Statements
- Arrays
- Collection Classes
- Looping (Iteration) Statements
- Strings
- Generics
- Partial Classes

- A.K.A. Selection Statements
 - if-else
 - The **if** statement selects a statement for execution based on the value of a **Boolean** expression

```
if(a==0)
{
    ...
}
else if(a<0)
{
    ...
}
else
{
    ...
}
```

Control Statements

– switch

- The **switch** statement is a control statement that handles multiple selections and enumerations by passing control to one of the **case** statements within its body.

```
int caseSwitch = 1;
switch (caseSwitch)
{
    case 1:
        Console.WriteLine("Case 1");
        break;
    case 2:
    case 3:
        Console.WriteLine("Case 2");
        break;
    default:
        Console.WriteLine("Default case");
        break;
}
```

Control Statements

- switch
 - Control is transferred to the **case** statement which matches the value of the switch.
 - The **switch** statement can include any number of **case** instances, but no two case statements can have the same value.
 - Execution of the statement body begins at the selected statement and proceeds until the **break** statement transfers control out of the **case** body.
 - A jump statement such as a **break** is required after each **case** block, including the last block whether it is a **case** statement or a **default** statement.
 - With one exception, C# does not support an implicit fall through from one case label to another. The one exception is if a **case** statement has no code.
 - If no case expression matches the switch value, then control is transferred to the statement(s) that follow(s) the optional **default** label.
 - If there is no **default** label, control is transferred outside the **switch**.
 - The switch expression can be either a string or an integer type: sbyte, byte, short, ushort, int, uint, long, ulong, char, string.

Control Statements

- In VB6 there was a control statement known as IIF
- In C# that concept exists as well, in a different format
- `string title = paygrade > 1 ? "Boss" : "Employee";`
 - Condition
 - Value if True
 - Value if False

Arrays

- An array is a data structure that contains a number of variables of the same type. Arrays are declared with a type:
 - `type[] arrayName;`
 - E.g. `string[] s; int[] i; decimal[] d; char[] c;`
- Arrays can be single-dimensional, multi-dimensional or jagged

Jagged Arrays

- A jagged array is an array whose elements are arrays.
- The elements of a jagged array can be of different dimensions and sizes.
- A jagged array is sometimes called an "array of arrays."

```
int[][] jaggedArray = new int[3][];
```
- Before you can use jaggedArray, its elements must be initialized. You can initialize the elements like this:

```
jaggedArray[0] = new int[5];  
jaggedArray[1] = new int[4];  
jaggedArray[2] = new int[2];
```

Arrays

- The following examples create single-dimensional, multidimensional, and jagged arrays:

```
// Declare a single-dimensional array  
int[] array1 = new int[5];
```

```
// Declare and set array element values  
int[] array2 = new int[] { 1, 3, 5, 7, 9 };
```

```
// Alternative syntax  
int[] array3 = { 1, 2, 3, 4, 5, 6 };
```

```
// Declare a two dimensional array  
int[,] multiDimensionalArray1 = new int[2, 3];
```

```
// Declare and set array element values  
int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };
```

```
// Declare a jagged array  
int[][] jaggedArray = new int[6][];
```

```
// Set the values of the first array in the jagged array structure  
jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
```

Arrays

- In C#, arrays are actually objects, and not just addressable regions of contiguous memory as in C and C++
- System.Array is the abstract base type of all array types. You can use the properties, and other class members, that **Array** has
 - E.g. .Length; GetLowerBound(); GetUpperBound()
- Pop-Quiz:
 - Can a System.Array contain multiple data types?

Arrays as parameters

- Arrays may be passed to methods as parameters.
- As arrays are reference types, the method can change the value of the elements
- But if you assign the array to a null or a new array it has no effect on the original
- Using a **ref** parameter when passing an array allows it to be altered as a result of the call. For example, the array can be assigned the null value or can be initialized to a different array

Collection Classes

- The .NET Framework provides specialized classes for data storage and retrieval.
- These classes provide support for stacks, queues, lists, and hash tables
- They are dynamic whereas arrays are static
 - **(NO MORE REDIM)**
- Most collection classes implement the same interfaces, and these interfaces may be inherited to create new collection classes that fit more specialized data storage needs.
- Collection Classes have the following properties:
 - Collection classes are defined as part of the System.Collections or **System.Collections.Generic** namespace.
 - Most collection classes derive from the interfaces **ICollection**, **IComparer**, **IEnumerable**, **IList**, **IDictionary**, and **IDictionaryEnumerator** and their generic equivalents.

Collection Classes

- Common Collection Classes:
 - ArrayList (dynamic array)
 - Hashtable (dictionary (key-value pairs))
 - Queue (Represents a first-in, first-out collection of objects)
 - Stack (Represents last-in, first out collection of objects)
 - SortedList (combo between array and dictionary – key-value pairs, but keys are sorted)

Custom Collection Classes

- You can make your own custom Collection class by either implementing all the interfaces yourself or by inheriting from `CollectionBase` or `DictionaryBase` and overriding a few methods.
- It is not hard to do
- It is type-safe!
- But the use of Generics in Collections makes this a thing of the past

Looping Statements

- A.K.A. Iteration Statements
- Iteration statements are looping constructs that allow a developer to iterate through a result set
 - for
 - foreach
 - while
 - do
- Iteration statements cause embedded statements to be executed a number of times, subject to the loop-termination criteria. These statements are executed in order, except when a [jump statement](#) is encountered.

Looping Statements - for

- The for loop executes a statement or a block of statements repeatedly until a specified expression evaluates to false.
- The for loop is handy for iterating over arrays and for sequential processing.
- In the following example, the value of int i is written to the console and i is incremented each time through the loop by 1.

```
for (int i = 1; i <= 5; i++)  
{  
    Console.WriteLine(i);  
}
```

- To iterate backward use i--

Looping Statements - foreach

- The **foreach** statement repeats a group of embedded statements for each element in an array or an object collection.

```
int[] fibarray = new int[] { 0, 1, 2, 3, 5, 8, 13 };  
foreach (int i in fibarray)  
{  
    System.Console.WriteLine(i);  
}
```

- The **foreach** statement is used to iterate through the collection to get the desired information, but should not be used to change the contents of the collection to avoid unpredictable side effects.

Looping Statements - foreach

- For example:

```
ArrayList list = new ArrayList();  
list.AddRange(new int[] { 1, 2, 4, 5, 6, 7, 8 });  
foreach (int i in list)  
{  
    list.Remove(i);  
}
```

- The code above will throw an [InvalidOperationException](#) stating: "Collection was modified; enumeration operation may not execute."
- foreach reads the collection every time it loops

Looping Statements - while

- The **while** statement executes a statement or a block of statements until a specified expression evaluates to **false**.

```
int n = 1;
while (n < 6)
{
    Console.WriteLine("Current value of n is {0}", n);
    n++;
}
```

- Because the test of the **while** expression takes place before each execution of the loop, a **while** loop executes zero or more times

Looping Statements - do

- The **do** statement executes a statement or a block of statements enclosed in `{ }` repeatedly until a specified expression evaluates to **false**

```
int x = 0;
do
{
    Console.WriteLine(x);
    x++;
}
while (x < 5);
```

- Unlike the while statement, the body loop of the **do** statement is executed at least once regardless of the value of the expression

Jump Statements

- Jump statements allow you to break out of looping and control statements
 - break
 - continue
 - goto
 - return

Jump Statements - break

- The **break** statement terminates the closest enclosing [loop](#) or [switch](#) statement in which it appears.
- Control is passed to the statement that follows the terminated statement, if any

```
for (int i = 1; i <= 100; i++)  
{  
    if (i == 5)  
    {  
        break;  
    }  
    Console.WriteLine(i);  
}
```

- A break statement is valid in a for, foreach, do or while loop.

Jump Statements - continue

- The **continue** statement passes control to the next iteration of the enclosing iteration statement in which it appears (*i.e. skips the current iteration*)

```
for (int i = 1; i <= 10; i++)  
{  
    if (i < 9)  
    {  
        continue;  
    }  
    Console.WriteLine(i);  
}
```

- In this example, the statements between continue and the end of the for body are skipped if $i < 9$.
- The output will be 9 and 10 only.

Jump Statements - goto

- The **goto** statement transfers the program control directly to a labeled statement
- A common use of **goto** is to transfer control to a specific switch-case label or the default label in a **switch** statement
- The **goto** statement is also useful to get out of deeply nested loops

Jump Statements - goto

- Example:

```
switch (n)
{
    case 1:
        cost += 25;
        break;
    case 2:
        cost += 25;
        goto case 1;
    case 3:
        cost += 50;
        goto case 1;
    default:
        Console.WriteLine("Invalid selection.");
        break;
}
```

Jump Statements - goto

- **Example:**

```
for (int i = 0; i < x; i++)  
{  
    for (int j = 0; j < y; j++)  
    {  
        if (array[i, j].Equals(myNumber))  
        {  
            goto Found;  
        }  
    }  
}
```

```
Console.WriteLine("The number {0} was not found.", myNumber);  
goto Finish;
```

Found:

```
Console.WriteLine("The number {0} is found.", myNumber);
```

Finish:

```
Console.WriteLine("End of search.");
```

Jump Statements - return

- The **return** statement terminates execution of the method in which it appears and returns control to the calling method.
- It can also return an optional value.
- If the method is a **void** type, the **return** statement can be omitted

```
double CalculateArea(int r)
{
    double area = r * r * Math.PI;
    return area;
}
```

- The return statement can also be used to exit a method from within iteration and control statements

Strings

- A C# string is a special kind of object
- A C# string is an **array of characters** declared using the **string** keyword
- A string literal is declared using quotation marks, as shown in the following example:
 - `string s = "Hello, World!";`
- String objects are *immutable*, meaning that they cannot be changed once they have been created
- Methods that act on strings actually return new string objects
- Working with large numbers of strings has a performance penalty...

String Concatenation

- To concatenate a string use the + operator
 - `s1 = s1 + s2; s1 += s2;`
- This actually creates a third string in memory
- To counteract the performance penalty use a **StringBuilder** class

```
System.Text.StringBuilder sb = new System.Text.StringBuilder();  
sb.Append("one ");  
sb.Append("two ");  
sb.Append("three");  
string str = sb.ToString();
```

Strings – Escape Character

- To include special characters in your literal string statements you can use the Escape character (`\`)
 - E.g. string `a = "Hello\r\nWorld!"`

```
Hello
World
```
 - Some examples:
 - `\r` = carriage return
 - `\n` = line feed
 - `\t` = tab
 - `\\` = `\` (so you can print a single back-slash)
 - The `@` symbol tells the string constructor to ignore escape characters and line breaks.
 - The following two strings are therefore identical:
 - `string p1 = "\\My Documents\\My Files\\";`
 - `string p2 = @"My Documents\My Files";`

String Methods

- Strings act like objects. As such they have built-in methods:
- Consider string `s`;
 - `char[] arrChr = s.ToCharArray();`
 - `bool result = s.Contains(string value);`
 - `string s2 = s.Substring(int startIndex, int length);`
 - `string s2 = s.ToUpper(); s3 = s.ToLower();`
- The [string](#) type, which is an alias for the `System.String` class, provides a number of useful methods for searching the contents of a string. For example the `IndexOf`, `LastIndexOf`, `StartsWith` and `EndsWith` methods.

String Splitting

- To split a string into portions based on a delimiter use the Split() method:

```
string s = "The cat sat on the mat.";
string[] arr = s.Split(' '); //space
// arr[0] == "The";
// arr[1] == "cat";
// etc.
```

ToString()

- Most objects have a ToString() method
- This method typically converts each object to a string representation of itself
- By default most objects just print their own type
 - `MyTestObject obj; obj.ToString(); //prints MyTestObject`
- But ToString() is overridable. You can make your object print whatever you want.

```
Class MyObj
{
    override string ToString()
    {
        ...
    }
}
```

ToString()

- E.g. DateTime objects print their date in given formats:
 - `DateTime.Now.ToString();` //will print current date/time in the current O/S' date and time format as defined by user settings (5/17/2007 2:15:00 PM)
- Some objects take parameters into their ToString() methods to give more control on how they print:
 - `DateTime.Now.ToString("d");` //will print current date/time in short date pattern (5/17/2007)
 - `DateTime.Now.ToString("D");` //will print current date/time in long date pattern (Thursday, May 17th 2007)
 - `DateTime.Now.ToString("yyyy, M - d");` //will print custom format 2007, 5 - 17
 - `decimal d=200.5m; d.ToString("c");` //will print in the currency format (\$200.50)
- There are many more formats – look them up in the Visual Studio online help!!!
 - Don't spend any time creating your own formatting routines...

Generics

- New feature in C# 2.0
- Akin to Templates concept in C++
- Generics introduce the concept of type parameters
- Design classes and methods that defer the specification of one or more types until declared and instantiated by client code
- The idea is to write code that can be called without incurring the cost or risk of runtime casts or boxing operations

- Example:

// Declare the generic class

```
public class GenericList<T>
```

```
{
```

```
    void Add(T input) { }
```

```
}
```

```
class TestGenericList
```

```
{
```

```
    private class ExampleClass { }
```

```
    static void Main()
```

```
{
```

```
    // Declare a list of type int
```

```
    GenericList<int> list1 = new GenericList<int>();
```

```
    // Declare a list of type string
```

```
    GenericList<string> list2 = new GenericList<string>();
```

```
    // Declare a list of type ExampleClass
```

```
    GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
```

```
}
```

```
}
```

Generics

- Use generic types to maximize code reuse, type safety, and performance.
- The most common use of generics is to create type safe collection classes
- The .NET Framework class library contains several new generic collection classes in the System.Collections.Generic namespace.
- These should be used whenever possible in place of classes such as ArrayList in the System.Collections namespace
- You can create your own generic interfaces, classes, methods, events and delegates
- Generic classes may be constrained to enable access to methods on particular data types
- Information on the types used in a generic data type may be obtained at run-time by means of reflection

Generics

- Traditionally if you wanted a class or method that could use different types you need to use objects or common interfaces
- This approach led to the need for a lot of casting and boxing/unboxing
- With Generics you can guarantee that your code operates on a specific type and achieve type safety without the cost

Partial Classes

- Partial type definitions allow the definition of a class, struct or interface to be split into multiple files
- Splitting a class, struct or interface type over several files can be useful when working with large projects, or automatically generated code
- Prior to Partial classes, if you needed code in separate files you needed a level of inheritance
- Partial classes eliminate that need

Partial Classes

- Each source file contains a section of the class definition, and all parts are combined when the application is compiled

```
public partial class Employee : Person, IEmployee
{
    public void DoWork()
    {
    }
}
```

```
public partial class Employee : IPerson, IDataEntity
{
    public void GoToLunch()
    {
    }
}
```

Partial Classes

- All the parts must use the **partial** keyword.
- All of the parts must be available at compile time to form the final type.
- All the parts must have the same accessibility, such as public, private, and so on.
- If any of the parts are declared abstract, then the entire type is considered abstract.
- If any of the parts are declared sealed, then the entire type is considered sealed.
- If any of the parts declare a base type, then the entire type inherits that class.
- All of the parts that specify a base class must agree, but parts that omit a base class still inherit the base type.
- Parts can specify different base interfaces, and the final type implements all of the interfaces listed by all of the partial declarations.
- Any class, struct, or interface members declared in a partial definition are available to all of the other parts.
- The final type is the combination of all the parts at compile time.