Chapter 11

Characters and Strings

Characters and Strings

Objectives

After completing this unit you will be able to:

- **Use the *string* class to address several common programming problems.**
- **Use the *StringBuilder* class in situations requiring a mutable string.**

Characters

- **C# provides the primitive data type char to represent individual characters. A character enclosed in single quotes represents a character literal.**

```
char ch1 = 'a';
```

- **A C# *char* is represented internally as an unsigned two-byte integer. You can cast back and forth between *char* and integer data types.**

```
char ch1 = 'a';
int n = (int) ch1;
n++;
ch1 = (char) n;       // ch1 is now 'b'
```

- **The relational operators ==, <, >, and so on apply to *char*.**

```
char ch1 = 'a';
char ch2 = 'b'
if (ch1 < ch2)        // expression is true
    ...
```

Sample Program

```
// CharTest.cs

using System;

public class CharTest
{
   public static void Main(string[] args)
   {
      char ch1 = 'a';
      char ch2 = 'b';
      Console.WriteLine("ch1 = {0}, ch2 = {1}",
         ch1, ch2);

      // demonstrate inequality for char
      if (ch1 < ch2)
         Console.WriteLine(ch1 + " < " + ch2);
      else if (ch1 == ch2)
         Console.WriteLine(ch1 + " == " + ch2);
      else
         Console.WriteLine(ch1 + " > " + ch2);

      // demonstrate casting between integers
      int n = (int) ch1;
      n++;
      ch1 = (char) n;
      Console.Write("After increment: ");
      Console.WriteLine("ch1 = {0}, ch2 = {1}",
         ch1, ch2);
      if (ch1 < ch2)
         Console.WriteLine(ch1 + " < " + ch2);
      else if (ch1 == ch2)
         Console.WriteLine(ch1 + " == " + ch2);
      else
         Console.WriteLine(ch1 + " > " + ch2);
   }
}
```

Character Codes

- **The integer corresponding to a character is referred to as its *character code*.**
  - You can easily write a C# program to display character codes.
- **The program *CharCode* provides an illustration.**

```
// CharCode.cs

using System;

public class CharCode
{
   public static void Main(string[] args)
   {
      byte nA = (byte) 'A';
      byte nZ = (byte) 'Z';
      for (int i = nA; i <= nZ; i++)
      {
         Console.Write(i + "  ");
         Console.WriteLine((char) i);
      }
      char ch = '\u0041';
      Console.WriteLine(ch);
   }
}
```

- **The output gives you a table of character codes for 'A' through 'Z'.**
  - The last line shows the character 'A' via a special Unicode escape sequence for characters (discussed in the next section).

ASCII and Unicode

- **Traditionally, a one-byte character code called ASCII has been used to represent characters.**
  - ASCII code is simple and compact.
  - But ASCII cannot be used to represent many different alphabets used throughout the world.
- **Modern computer systems prefer to use a two-byte character code called Unicode.**
  - Most modern (and many ancient) alphabets can be represented by Unicode characters.
  - ASCII is a subset of Unicode, corresponding to the first 255 Unicode character codes.
  - For more information on Unicode, you can visit the Web site *www.unicode.org*.
  - C# uses Unicode to represent characters.

Escape Sequences

- **You can represent any Unicode character in a C# program by using the special escape sequence beginning with \u followed by hexadecimal digits.**

```
char A = '\u0041';   // 41 (hex) is 65 (dec) or
'A'
```

- **Special escape sequences are provided for a number of standard non-printing characters and for characters like quotation marks that would be difficult to represent otherwise.**

Escape Character	Name	Value
\’	Single quote	0x0027
\”	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

- **The program *Escape* illustrates a few escape sequences in a C# program.**

Strings

- **More useful in programs than individual characters are strings of characters.**
- **C# provides a *string* type, which is an alias for the *String* class in the *System* namespace.**
  - As a class type, **string** is a reference type.
  - Much string functionality, available in all .NET languages, is provided by the **String** class.
  - The C# compiler provides additional support to make working with strings more concise and intuitive.
- **In this section we will first outline the main features of the *String* class.**
- **We will then look at string input, at the additional support provided by C#, and at the issues of string equality.**
- **The following section surveys some of the useful methods of the *String* class.**

String Class

- **The *String* class inherits directly from *Object* and is a sealed class, which means that you cannot further inherit from *String*.**
  - We will discuss inheritance and sealed classes in Chapters 13 and 14.
  - When a class is sealed, the compiler can perform certain optimizations to make methods in the class more efficient.
- **Instances of *String* are immutable, which means that once a string object is created, it cannot be changed during its lifetime.**
  - Operations that appear to modify a string actually return a new string object.
  - If, for the sake of efficiency, you need to modify a string-like object directly, you can make use of the **StringBuilder** class, which we will discuss in a later section.

Compiler Support

- **The C# compiler provides a number of features to make working with strings easier and more intuitive:**
  - String literals and initialization
  - Concatenation
  - Index
  - Relational operators

String Literals and Initialization

- **You can define a string literal by enclosing a string of characters in double quotes.**
  - Special characters can be represented using an escape sequence, as discussed earlier in the chapter.
  - You may also define a "verbatim" string literal using the @ symbol.
  - In a verbatim string, escape sequences are not converted but are used exactly as they appear.
  - If you want to represent a double quote inside a verbatim string, use two double quotes.
- **The proper way to initialize a string variable with a literal value is to supply the literal after an equal sign.**
  - You do not need to use **new** like you do with other data types.
  - The following provides some examples of string literals and initializing string variables:

```
string s1 = "bat";
string path1 = "c:\\OI\\CSharp\\Chap11\\Concat";
string path = @"c:\OI\CSharp\Chap11\Concat\";
string greeting = @"""Hello, world""";
```

Concatenation

- **The *String* class provides a method *Concat* for concatenating strings.**
  - In C# you can use the plus operator + to perform concatenation.
  - As we saw in Chapter 10, when + is overloaded, you automatically get an overload of the compound operator +=.
  - The program **Concat** illustrates string literals and concatenation.

Index

- **A string has a zero-based index, which can be used to access individual characters in a string.**
  - That means that the first character of the string **str** is **str**[0], the second character is **str**[1], and so on.
- **You can extract an individual character from a string using a square bracket and a zero-based index.**

```
string s1 = "bat";
char ch = s1[0]; // contains ‘b’
```

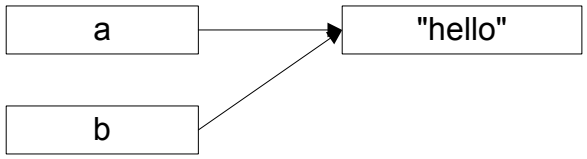
Relational Operators

- **In general, for reference types, the == and != operators check if the *object references* are the same, not whether the contents of the memory locations referred to are the same.**
  - However, the **String** class overloads these operators, so that the textual content of the strings is compared.
- **The program *StringRelation* illustrates using these relational operators on strings.**
  - The inequality operators, such as <, are not available for strings; use the **Compare** method.

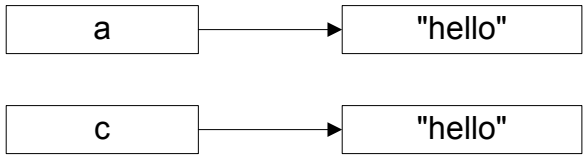
```
using System;

public class StringRelation
{
   public static void Main(string[] args)
   {
      string a1 = "hello";
      string a2 = "hello";
      string b = "HELLO";
      string c = "goodbye";
      Console.WriteLine("{0} == {1}: {2}",
         a1, a2, a1 == a2);
      Console.WriteLine("{0} == {1}: {2}",
         a1, b, a1 == b);
      Console.WriteLine("{0} != {1}: {2}",
         a1, c, a1 != c);
      //Console.WriteLine("{0} < {1}: {2}",
      //   a1, c, a1 < c);                //illegal
   }
}
```

String Equality

- **To fully understand issues of string equality, you should be aware of how the compiler stores strings.**
  - When string literals are encountered, they are entered into an internal table of string identities.
  - If a second literal is encountered with the same string data, an object reference will be returned to the existing string in the table; no second copy will be made.
  - As a result of this compiler optimization, the two object references will be the same, as represented in this figure.



- **You should not be mislead by this fact to conclude that two object references to the same string data will always be the same.**
  - If the contents of the string get determined at runtime, for example, by the user inputting the data, the compiler has no way of knowing that the second string should have an identical object references.
  - Hence you will have two distinct object references, which happen to refer to the same data, as illustrated in this figure.



String Comparisons

- **As discussed, when strings are checked for equality, either through the relational operator == or through the *Equals* method, a comparison is made of the contents of the strings, not of the object references.**
  - So in both the previous cases, the strings **a** and **b** will check out as equal.
  - You have to be more careful with other reference types, where reference equality is not the same as content equality.
  - We will see an example shortly when we discuss the **StringBuilder** type.
  - Comparison operations on strings are by default case-sensitive, although there is an overloaded version of the Compare method that permits case-insensitive comparisons.
  - The empty string should be distinguished from null.
  - If a string has not been assigned, it will be a null reference.
  - Any string, including the empty string, compares greater than a null reference.
  - Two null references compare equal to each other.

String Comparison

- **The fundamental way to compare strings for equality is to use the *Equals* method of the *String* class.**
  - There are several overloaded versions of this function, including a static version that takes two **string** parameters, and a non-static version that takes one **string** parameter that is compared with the current instance.
  - These methods test if the contents of the strings are identical and are case sensitive. A **bool** value of **true** or **false** is returned.
- **If you wish to perform a case-insensitive comparison, you may use the *Compare* method.**
  - This method has several overloaded versions, all of them static.
  - Two strings, s1 and s2, are compared.
  - An integer is returned expressing the lexical relationship between the two strings, as shown in the table.

Relationship	Return Value
s1 less than s2	Negative integer
s1 equal to s2	0
s1 greater than s2	Positive integer

String Comparison (Cont'd)

- **A third parameter allows you to control the case sensitivity of the comparison.**
  - If you use only two parameters, a case-sensitive comparison is performed. The third parameter is a **bool**. A value of **false** calls for a case-sensitive comparison, and a value of **true** calls for ignoring case.
- **The program *StringCompare* illustrates a number of comparisons, using both the *Equal* and *Compare* methods.**

```
// StringCompare.cs

using System;

public class StringCompare
{
   public static void Main(string[] args)
   {
      string a1 = "hello";
      string a2 = "hello";
      string b = "HELLO";
      string c = "goodbye";
      Console.WriteLine("{0}.Equals({1}): {2}",
         a1, a2, a1.Equals(a2));
      Console.WriteLine(
         "String.Equals({0},{1}): {2}",
         a1, a2, String.Equals(a1,a2));
      Console.WriteLine("Case sensitive...");
      Console.WriteLine(
         "String.Compare({0},{1}): {2}",
         a1, b, String.Compare(a1,b));
      ...
```

String Input

- **The *Console* class has methods for inputting characters and strings.**
  - The method **Read** will read in a single character (as an **int**).
  - The method **ReadLine** will read in a line of input, terminated by a carriage return, line feed, or combination, and will return a **string**.
  - In general, the **ReadLine** method is the easier to use and synchronizes nicely with **Write** and **WriteLine**.
  - The program **ReadStrings** illustrates reading in a first name, a middle initial, and a last name.
  - All input is done via **ReadLine**.
  - The middle initial as a character is determined by extracting the character at position 0.
- **Our *InputWrapper* class, which we introduced in Chapter 2 and have used from time to time, has a method *getString*, which provides a prompt and reads in a string.**
- **For an illustration in this chapter of using *InputWrapper* for string input, see the program *StringDemo*.**

String Methods and Properties

- **In this section we will survey a few useful methods and properties of the *String* class.**
  - Many of the methods have various overloaded versions. We show a representative version.
  - Consult the online documentation for details on these and other methods.
  - The program **StringMethods** demonstrates all the examples that follow.
- **Length**

```
public int Length {get;}
```

  - This property returns the length of a string. Notice the convenient shorthand notation that is used for declaring a property.

```
string str = "hello";
int n = str.Length;           // 5
```

- **ToUpper**

```
public string ToUpper();
```

  - This method returns a new string in which all characters of the original string have been converted to upper case.

```
str = "goodbye";
str = str.ToUpper();          // GOODBYE
```

More String Methods and Properties

- **ToLower**

```
public string ToLower();
```

  - This method returns a new string in which all characters of the original string have been converted to lower case.

```
str = str.ToLower();          // goodbye
```

- **Substring**

```
public string Substring(int startIndex,
                        int length);
```

  - This method returns a substring that starts from a specified index position in the value and continues for a specified length. Remember that in C# the index of the first character in a string is 0.

```
string sub = str.Substring(4,3);  // bye
```

- **IndexOf**

```
public int IndexOf(string value);
```

  - This method returns the index of the first occurrence of the specified string. If the string is not found, –1 is returned.

```
str = "goodbye";
int n1 = str.IndexOf("bye");  // 4
int n2 = str.IndexOf("boo");  // -1
```

StringBuilder Class

- **As we have discussed, instances of the *String* class are immutable.**
  - As a result, when you manipulate instances of **String** you are frequently obtaining new **String** instances.
  - Depending on your applications, creating all these instances may be expensive.
  - The .NET library provides a special class **StringBuilder** (located in the **System.Text** namespace) in which you may directly manipulate the underlying string without creating a new instance.
  - When you are done, you can create a **String** instance out of an instance of **StringBuilder** by using the **ToString** method.
- **A *StringBuilder* instance has a capacity and a maximum capacity.**
  - These capacities can be specified in a constructor when the instance is created.
  - By default, an empty **StringBuilde**r instance starts out with a capacity of 16.
  - As the stored string expands, the capacity will be increased automatically.

StringBuilderDemo

- **The program *StringBuilderDemo* provides a simple demonstration of using the *StringBuilder* class.**
  - It shows the starting capacity and the capacity after strings are appended. At the end, a **String** is returned.

```
// StringBuilderDemo.cs

using System;
using System.Text;

public class StringBuilderDemo
{
   public static void Main(string[] args)
   {
      StringBuilder build = new StringBuilder();
      Console.WriteLine("capacity = {0}",
         build.Capacity);
      build.Append(
         "This is the first sentence.\n");
      Console.WriteLine("capacity = {0}",
         build.Capacity);
      build.Append(
         "This is the second sentence.\n");
      Console.WriteLine("capacity = {0}",
         build.Capacity);
      build.Append("This is the last sentence.\n");
      Console.WriteLine("capacity = {0}",
         build.Capacity);
      string str = build.ToString();
      Console.Write(str);
   }
}
```

StringBuilder Equality

- **You should be aware that *StringBuilder* instances are object references and obey the normal rules of reference equality.**
- **Thus if you have two distinct *StringBuilder* references that happen to have the same contents, these references will not compare as equal using the relational operator ==, but they will compare as equal using the *Equals* method.**
- **The *StringBuilderCompare* program illustrates this point.**
  - It also shows how the relational operator == is overloaded to check for content equality in the case of **string**.

Programming With Strings

- **We conclude this chapter by providing several common examples of programming with strings:**
  - Command Line Arguments
  - Command Loops
  - Splitting a String

Command Line Arguments

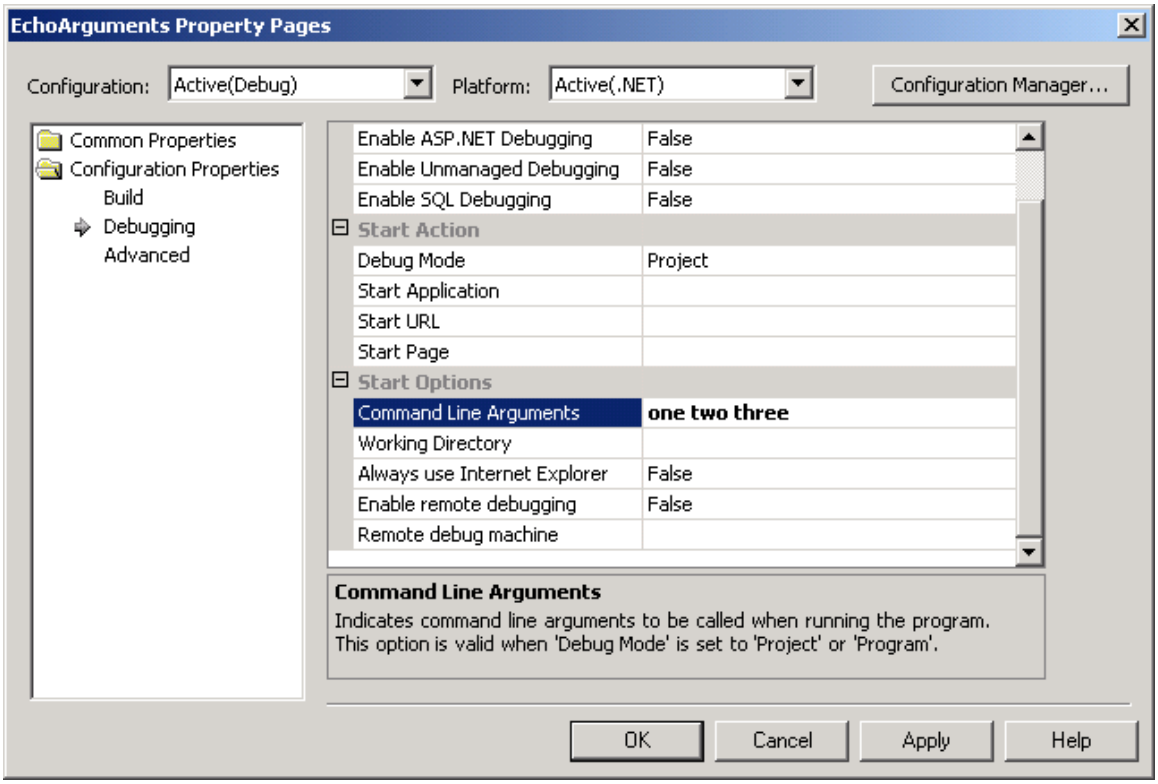
- **The *Main* method contains a *string* array.**
  - The elements of this array are the arguments passed to the program when the program is started from the command line.
- **The program *EchoArguments* writes out each argument on a separate line.**

```
// EchoArguments.cs

using System;

public class EchoArguments
{
   public static void Main(string[] args)
   {
      foreach (string arg in args)
         Console.WriteLine(arg);
   }
}
```

- **We previewed arrays and the *foreach* loop in Chapter 6, and we will discuss arrays in more detail in Chapter 12.**

Command Line Arguments in the IDE

- **You can set command line arguments in the Visual Studio IDE from the project properties.**
  - Go to Configuration Properties | Debugging



  - Save the .csproj.user file for this setting to be preserved.

Command Loops

- **A common pattern for console programs is for there to be a "command loop."**
  - Commands are entered at the keyboard in response to a prompt, the command typed in is compared to the supported commands, and the appropriate code is executed for each command.
  - This command processing continues in a loop until an appropriate command such as "quit" is entered to stop the processing.

```
Console.WriteLine("Enter command, 'quit' to exit");
cmd = iw.getString("> ");
while (! cmd.Equals("quit"))
{
   if (cmd.Equals("length"))
      // process "length" command
   else if (cmd.Equals("new"))
      // process " new " command
   else if (cmd.Equals("show"))
      // process " show " command
...
   else
      help();
   cmd = iw.getString("> ");
}
```

- **The program *StringDemo* illustrates such a command loop.**
  - This program provides an interactive demonstration of a number of methods of the **String** class.

Splitting a String

- **The *String* class provides a very useful method, *Split*, that can be used for splitting a string into substrings, based on specified characters that are used as separators.**

```
string[] Split(char[] separator);
```

  - The separators (such as blank, comma, tab, and new line) are placed in an array of characters, which is passed to the **Split** method.
  - The substrings that are delimited by these separators are returned as an array of strings.
  - If the separators do not occur, the whole string is returned as a one-element array.

- **The program *StringSplit* provides an illustration of the use of this method.**

  - If you are not accustomed to the **foreach** loop, you may wish to refer back to Chapter 6.

Summary

- **In this chapter we studied the C# data types *char* and *string*.**
- **In C# characters are represented in Unicode and take up 16 bits.**
- **The C# *string* data type is an alias for the *String* class, which has a number of useful methods.**
- **The C# compiler makes working with strings somewhat easier, allowing you to use operators for concatenation and testing for equality.**
- **Instances of *String* are immutable once they have been created. A special class *StringBuilder* can be used in situations where we want to make changes to a string without creating a new object.**
- **We concluded the chapter by looking at several common programming situations involving strings, including working with command line arguments, processing a command loop, and splitting a string into constituent parts.**