Chapter 15

Formatting and Conversion

Formatting and Conversion

Objectives

After completing this unit you will be able to:

- **Use the formatting features of C# to control how your output looks.**
- **Use *System.Convert* to do type conversions.**
- **Define your own type conversions.**

Introduction to Formatting

- **For simple programs where we are not concerned with appearance, the default formatting provided by .NET library classes may be quite adequate.**
- **When you do wish to control formatting, there are some simple facilities provided by the .NET library classes that you can use.**
- **Every class in C# inherits the *ToString* method of the object base class, so there is always a string representation available.**
- **We will discuss how to control the format of numeric types and how to align strings, illustrating this using our case study.**
- **Many conversions among the standard types can be accomplished with the *System.Convert* class.**
- **Finally, we will see how you can define conversion operations in your own classes.**

ToString

- **The fundamental issue in formatting is obtaining an appropriate string representation of an object.**
- **In C# every class ultimately derives from object, which has a method *ToString*, which returns a string representation of the object.**
  - Since primitive data types can be treated as classes, we can also apply the **ToString** method to data types like **int**, **decimal**, and **float**.
  - If we use the object in a context where a string is expected, the **ToString** method will be called automatically to obtain its string representation.
- **Obtaining a string representation of a number should not be confused with *converting* a number to a string.**
  - A string and a number are fundamentally different data types, and in C# there is no implicit conversion from one to the other.
  - Conversion is reserved for situations like converting an integer into a wider integer, or an integer to a floating point number, and so on.
  - We will discuss conversions later in the chapter.
- **The program *NumberToString* illustrates obtaining a string representation in several simple scenarios.**

ToString in Your Own Class

- **The *ToString* method is available in every class in C#, including classes you define yourself.**
  - As an example, consider the simple hotel class in **HotelToString\Step1**.

```
// Hotel.cs – Step 1

using System;

public class Hotel
{
   private string city;
   private string name;
   public Hotel(string city, string name)
   {
      this.city = city;
      this.name = name;
   }
}
```

  - In the test program we create two hotel objects and write them out with **WriteLine**, relying on **ToString** to implicitly obtain the string representation for us.

ToString in Your Own Class (Cont'd)

- **The *ToString* method being used is the one defined in the base class *object* from which our *Hotel* class implicitly inherits.**
  - The base class knows nothing about the particulars of our class, so it does something very simple: It displays the name of the class. (You may wonder how the base class knows the name of our particular class. It obtains the name using a feature in .NET known as *reflection*, which we will touch upon in Chapter 20.)
  - The **ToString** method is virtual, which means we may override it in our own class to display a more meaningful representation.
- **The new version of our class is implemented in the directory *HotelToString\Step2*.**

Using Placeholders

- **When doing output with *WriteLine*, we usually use placeholders, {0}, {1}, and so on.**
  - This output mechanism uses **ToString** under the hood.
- ***HotelToString\Step3* illustrates using placeholders in the test program.**

```
// HotelToString.cs - Step 3

using System;

public class HotelToString
{
   public static void Main()
   {
      Hotel alpha = new Hotel("Atlanta", "Dixie");
      Hotel beta = new Hotel("Boston", "Yankee");
      Console.WriteLine("Hotel alpha is {0}",
         alpha);
      Console.WriteLine("Hotel beta is {0}", beta);
   }
}
```

- **Output is:**

```
Hotel alpha is Atlanta Dixie
Hotel beta is Boston Yankee
```

Format Strings

- **C# has extensive formatting capabilities, which you can control through placeholders and format strings.**
  - Simple placeholders: {n}, where n is 0, 1, 2, and so on.
  - Control width: {n,w}, where w is width (positive for right justified and negative for left justified).
  - Format string: {n:S}, where S is a format string.
  - Width and format string: {n,w:S}.
- **A format string consists of a format character followed by an optional precision specifier.**
  - The table shows the available format characters.

Table 15–1 C# Format Characters

Format Character	Meaning
C	Currency (locale specific)
D	Decimal integer
E	Exponential (scientific)
F	Fixed point
G	General (E or F)
N	Number with embedded commas
X	Hexadecimal

- **We will illustrate a number of different format scenarios by a series of programs displaying powers of two in various formats, beginning with just using simple placeholders.**

Simple Placeholders

- **The simplest way to perform output in C# is to use the *Console.WriteLine* method with placeholders {0}, {1}, and so on.**
- **Our first program is *PowerTwo\Step0*, which simply displays powers of two without any special formatting.**

```
using System;
public class PowerTwo
{
   public static void Main()
   {
      long power = 1;
      for (int i = 0; i < 16; i++)
      {
         Console.WriteLine("{0} {1}", i, power);
         power *= 2;
      }
   }
}
```

```
0 1
1 2
2 4
3 8
4 16
5 32
6 64
7 128
8 256
9 512
10 1024
11 2048
...
```

Controlling Width

- **The simplest way to control format is through a width specifier.**
  - The placeholders now have the form {n,w}, where n = 0, 1, 2, and so on and w specifies the width (positive for right justified and negative for left justified).
- **The program *PowerTwo\Step1* illustrates controlling width.**
  - The first column is printed left justified, and the second column is printed right justified.

```
using System;
public class PowerTwo
{
   public static void Main()
   {
      long power = 1;
      for (int i = 0; i < 16; i++)
      {
         // Negative value for left justification
         Console.WriteLine("{0,-3} {1,10}",
            i, power);
         power *= 2;
      }
   }
}
```

```
0            1
1            2
2            4
3            8
4           16
...
```

Format String

- **The next step in formatting is to use a format string, usually in conjunction with a width specifier.**
  - The available format characters are shown in Table 15–1 earlier in the chapter.
  - Here are some examples of placeholders with format strings.
  - In each case the width is 36.

```
{0,36:D}   Decimal integer
{0,36:N0}  Number with commas, precision 0
{0,36:N4}  Number with commas, precision 4
{0,36:X}   Hexadecimal
{0,36:F}   Fixed point
{0,36:G}   General (E or F)
{0,36:F26} Fixed point, precision 26
{0,36:E26} Exponential, precision 26
```

- **The program *PowerFormat* illustrates use of these format strings in displaying 2^{60} or 2^{-60}.**
- **For some additional illustrations of numeric formatting you may look at the programs *PowerTwo\Step2* and *NegativePower*.**

Currency

- **The "C" format character formats in a manner appropriate for currency, including using a currency symbol.**
  - Currency formatting is specific to a locale.
  - The default locale is the United States, and the currency symbol is the dollar sign $. Working with other locales is beyond the scope of this course.
  - If globalization issues are of interest to you, you may wish to study the documentation of the **System.Globalization** namespace.
  - The C format character is used in exactly the same manner as the other format characters discussed previously.
- **The program *MoneyPower* provides an illustration.**
  - This program calculates the amount of money paid to a wise man who performed a service for an Eastern monarch.

  When asked for a reward, he replied that his wants were modest: All he wanted was the amount of money equal to a penny placed on the first square of a chessboard, two pennies on the second square, four pennies on the third square, eight pennies on the fourth square, and so.
  - This program also illustrates using the decimal data type to obtain accurate representations of financial quantities, with many digits of precision. The program shows the total day by day.

Currency Format Example

```
// MoneyPower.cs

using System;

public class MoneyPower
{
   public static void Main()
   {
      decimal power = .01m;
      decimal total = 0m;
      for (int i = 1; i <= 64; i++)
      {
         total += power;
         Console.WriteLine("{0,-3} {1,30:C}",
            i, total);
         power *= 2;
      }
   }
}
```

```
1                                $0.01
2                                $0.03
3                                $0.07
4                                $0.15
5                                $0.31
6                                $0.63
7                                $1.27
8                                $2.55
...
61      $23,058,430,092,136,939.51
62      $46,116,860,184,273,879.03
63      $92,233,720,368,547,758.07
64     $184,467,440,737,095,516.15
```

String.Format

- **The *System.String* class has several useful methods to help you with formatting tasks.**
  - We will discuss the methods **Format**, **PadLeft**, and **PadRight**.
- **Often we may wish to format numbers into a string.**
  - This task can be accomplished with the **Format** method of the **String** class.

```
public static string Format(
   string format,
   object[] args
);
```

  - The format has exactly the same syntax used in the **WriteLine** method.
- **As an illustration, consider the *MoneyReport* program the wise man wrote to prepare a report for the king stating the amount owed him.**

```
public static string CreateReport(decimal amount)
{
   string str = "Dear Your Majesty,\n\t" +
      string.Format("You owe me {0,30:C}", amount)
      + "\nSincerely, Your Humble Servant";
   return str;
}
```

PadLeft and PadRight

- **The *Format* method is useful for formatting numbers into a string.**
  - What if you want to control the formatting of just strings—i.e., make them align the way you want.
- **An easy approach is to use the *PadLeft* and *PadRight* methods, which are quite self-explanatory.**
  - Each method has two overloaded forms. Consider **PadLeft**.

```
public string PadLeft(
   int totalWidth
);
public string PadLeft(
   int totalWidth,
   char paddingChar
);
```

  - The **totalWidth** is the total number of characters in the result string, equal to the original characters plus any padding characters.
  - The default character used for padding is space, but you can change that by using the second form of the method.
- **The program *PowerTwo\Step2* illustrates use of *PadLeft* and *PadRight* to make headers line up properly with the columns of numbers underneath.**

PadLeft and PadRight (Cont'd)

```
// PowerTwo.cs - Step 2

using System;

public class PowerTwo
{
   public static void Main()
   {
      decimal power = 1;
      string header1 = "Num";
      string header2 = "Power";
      string header = header1.PadRight(4) +
         header2.PadLeft(30);
      Console.WriteLine(header);
      for (int i = 0; i < 64; i++)
      {
         Console.WriteLine("{0,-4}{1,30:N0}",
            i, power);
         power *= 2;
      }
   }
}
```

- The format **N0** is number with commas, precision 0.

```
Num                                Power
0                                      1
1                                      2
2                                      4
3                                      8
4                                     16
...
61       2,305,843,009,213,693,952
62       4,611,686,018,427,387,904
63       9,223,372,036,854,775,808
```

Bank Case Study: Step 4

- **The Step 3 version of our bank case study from Chapter 14 does not do any special formatting, and the result is not very neat.**
- **As an example, try the top level "show" command.**

```
> show
1       C: Bob  100
2       S: Mary 200
3       C: Charlie      300
```

  - The columns do not line up, and the balances are not shown as monetary amounts.
  - If we examine the code for the "show" command, we see that it uses the **GetAccounts** method of the **Bank** class. In all we have to make minor code changes in three classes: **Bank**, **Account**, and **Atm**.

- **The new versions are in the *CaseStudy* directory for Chapter 15.**
- **Please examine the code online.**

Type Conversions

- **Another important practical programming issue is converting among types.**
- **This is particularly important in C#, which is a strongly typed language.**
  - Where in some languages (such as Visual Basic 6) you can quite freely mix different data types and the compiler will usually sort things out in a reasonable manner, in C# you have to be quite precise in your use of types.
  - We discussed this issue in Chapter 4.
  - While you are becoming oriented to the C# environment, you may find that having to make explicit type conversions is somewhat of a chore, but you should soon get used to it, and your programs can benefit from the increased robustness that type safety provides.
- **This section discusses type conversions, both for built-in types and for user-defined types.**

Conversion of Built-In Types

- **The *Convert* class of the *System* namespace provides a comprehensive set of methods for converting among the built-in types.**
  - The conversion methods are static, with names of the form **ToXxxx**, where **Xxxx** is a type.
  - Note that type names are those used in the **System** namespace, not C# type names. For example, to convert to an **int**, use the method **ToInt32**.
- **An illustration is provided by our *InputWrapper* class in *CaseStudy*.**

```
using System;

class InputWrapper
{
    public int getInt(string prompt)
    {
        Console.Write(prompt);
        string buf = Console.ReadLine();
        return Convert.ToInt32(buf);
    }
    public double getDouble(string prompt)
    {
        Console.Write(prompt);
        string buf = Console.ReadLine();
        return Convert.ToDouble(buf);
    }
   ...
}
```

Conversion of User-Defined Types

- **Recall from Chapter 4 that there are two kinds of conversions in C#: implicit and explicit.**
  - An **implicit** conversion can be performed transparently by the compiler when there is no loss of information, such as in widening an integer data type.
  - Thus the following is legal:

```
int a = 718;
long b = a;
```

  - If there is a potential loss of information, the compiler will disallow a conversion, unless you explicitly tell the compiler to accept the conversion by performing a **cast** operation.
  - Such a conversion is said to be **explicit**.
  - Here is an example of an explicit conversion:

```
long c = 718;
c = c*c;
int d = (int) c;
```

- **To implement conversions in your own class you must define appropriate operators.**
  - Recall the discussion of operator overloading in Chapter 10.
  - Operators are special static methods that contain the keyword **operator**.
  - There are additional keywords, **implicit** and **explicit**, for defining implicit and explicit conversions.

Conversion of User-Defined Types (Cont'd)

- **We illustrate conversions through the class *Money*, which internally stores a money value as a *decimal*.**
  - Constructors are provided to create a **money** object from a **string**, a **double**, and a **decimal**.
  - A property is provided that can return a string representation of a **money** object. We then provide three overloaded operators to explicitly convert to **Money** and three overloaded operators to implicitly convert to **string**, **double**, and **decimal**, respectively.
- **Our example program is in the *MoneyConvert* directory. Note that the code illustrates using the *System.Convert* class.**
- **The conversion to *string* is implicit, as illustrated by the line of code**

```
string s = a;
```

  - There is an explicit cast to **string** in the **WriteLine** statements.
  - Without the cast, the call to **WriteLine** would be ambiguous, since **WriteLine** can also accept decimal and double parameters, and we define implicit conversions for these data types as well as to **string**.

User Defined Conversions: Example

```
// Money.cs

using System;

public class Money
{
   private decimal amount;
   public Money(string str)
   {
      amount = Convert.ToDecimal(str);
   }
   ...
   public static explicit operator Money(
      string str)
   {
      Money mon = new Money(str);
      return mon;
   }
   public static explicit operator Money(
      double num)
   {
      Money mon = new Money(num);
      return mon;
   }
   ...
   public static implicit operator string(
      Money mon)
   {
      return string.Format("{0:C}", mon.amount);
   }
   public static implicit operator double(
      Money mon)
   {
      return Convert.ToDouble(mon.amount);
   }
   ...
}
```

Conversion Example (Cont'd)

```
// MoneyConvert.cs

using System;

public class MoneyConvert
{
   public static void Main()
   {
      Money x = new Money("30.33");
      Console.WriteLine(x.MoneyStr);
      Money a = new Money();
      a = (Money) "40.44";
      Console.WriteLine((string) a);
      a = (Money) 50.55;
      Console.WriteLine((string) a);
      a = (Money) 60.66m;
      Console.WriteLine((string) a);
      string s = a;
      Console.WriteLine("a = (string) {0}", s);
      double b = a;
      Console.WriteLine("a (double) = {0}", b);
      decimal c = a;
      Console.WriteLine("a (decimal) = {0:C}", c);
   }
}
```

Summary

- **Fundamentally, formatting is obtaining an appropriate string representation of an object.**
- **Every class in C# inherits the *ToString* method of the *object* base class, so there is always a string representation available.**
- **This default simply returns the name of the class, so normally you will want to override the default.**
- **Numeric types may be formatted with width and precision specifiers.**
- **Strings may be aligned with *PadRight()* and *PadLeft()*.**
- **Many conversions among the standard types can be accomplished with the *System.Convert* class.**
- **You can define both explicit and implicit conversions in your own classes by providing the appropriate operators.**