LESSON 2

==============================

DATA TYPES

Built-In Data Types

C# provides all the data types that are available in Java, and adds support for unsigned numerals and a new 128-bit high-precision floating-point type.

For each primitive data type in Java, the core class library provides a wrapper class that represents it as a Java object. For example, the Int32 class wraps the **int** data type, and the Double class wraps the double data type.

On the other hand, all primitive data types in C# are objects in the System namespace. For each data type, a short name, or alias, is provided. For instance, **int** is the short name for **System.Int32** and **double** is the short form of **System.Double**.

The list of C# data types and their aliases is provided in the following table. As you can see, the first eight of these correspond to the primitive types available in Java. Note, however, that Java's **boolean** is called **bool** in C#.

LESSON 2

= =

BUILT-IN DATA TYPES

Short Name	.NET Class	Type	Width (Bits)	Range (bits)
byte	Byte	Unsigned integer	8	0 to 255
sbyte	SByte	Signed integer	8	-128 to 127
int	Int32	Signed integer	32	-2,147,483,648 to 2,147,483,647
uint	UInt32	Unsigned integer	32	0 to 4294967295
short	Int16	Signed integer	16	-32,768 to 32,767
ushort	UInt16	Unsigned integer	16	0 to 65535
long	Int64	Signed integer	64	-9223372036854775808 to 9223372036854775807
ulong	UInt64	Unsigned integer	64	0 to 18446744073709551615
float	Single	Single-precision floating point type	32	-3.402823e38 to 3.402823e38
double	Double	Double-precision floating point type	64	-1.79769313486232e308 to 1.79769313486232e308
char	Char	A single Unicode character	16	Unicode symbols used in text
bool	Boolean	Logical Boolean type	8	True or false
object	Object	Base type of all other types		
string	String	A sequence of characters		
decimal	Decimal	Precise fractional or integral type that can represent decimal numbers with 29 significant digits	128	±1.0 × 10e−28 to ±7.9 × 10e28

LESSON 2

OPERATORS

In C#, an *operator* is a program element that is applied to one or more *operands* in an expression or statement. Operators that take one operand, such as the increment operator (`++`) or `new`, are referred to as *unary* operators. Operators that take two operands, such as arithmetic operators (`+`, `-`, `*`, `/`), are referred to as *binary* operators. One operator, the conditional operator (`?:`), takes three operands and is the sole ternary operator in C#.

The following C# statement contains a single unary operator and a single operand. The increment operator, `++`, modifies the value of the operand `y`

Operators, Evaluation, and Operator Precedence

An operand can be a valid expression that is composed of any length of code, and it can comprise any number of sub expressions. In an expression that contains multiple operators, the order in which the operators are applied is determined by *operator precedence*, *associativity*, and parentheses.

Each operator has a defined precedence. In an expression that contains multiple operators that have different precedence levels, the precedence of the operators determines the order in which the operators are evaluated. For example, the following statement assigns 3 to `n1`.

```
n1 = 11 - 2 * 4;
```

The multiplication is executed first because multiplication takes precedence over subtraction.

Category	**Operators**
arithmetic	**-, +, *, /, %, ++, --**
logical	**&&, ||, !**
binary	**&, |, ^, ~, <<, >>**
comparison	**==,!=, >, <, >=, <=**
assignment	**=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=**
string concatenation	**+**
type conversion	**(type), as, is, typeof, sizeof**
other	**., new, (), [], ?:, ??**

LESSON 2

Default Variable Values

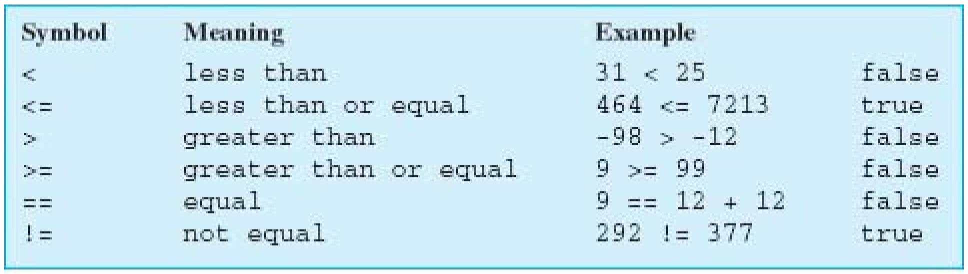
Each data type in C# has a **default value** (default initialization) which is used when there is no explicitly set value for a given variable. We can use the following table to see the default values of the types, which we already got familiar with:

Type	Default Value	Type	Default Value
sbyte	0	**float**	0.0f
byte	0	**double**	0.0d
short	0	**decimal**	0.0m
ushort	0	**bool**	false
int	0	**char**	'\u0000'
uint	0u	**string**	null
long	0L	**object**	null
ulong	0u		

RELATIONAL OPERATORS AND EXPRESSIONS

Arithmetic operators take numeric operands and give numeric results. For example, the value of 3+4 is an integer, 7. By contrast, an expression such as 3<4, stating that 3 is less than 4, gives the value true, and the expression 7<2 gives the value false. Type bool, named for the British mathematician and logician, George Boole (1815-1864), provides two values, true and false, which we use to express the value of relational and logical expressions.

C# provides relational and equality operators, listed in Figure 3.1, which take two operands of a primitive type and produce a bool result



Symbol	Meaning	Example	
<	less than	31 < 25	false
<=	less than or equal	464 <= 7213	true
>	greater than	-98 > -12	false
>=	greater than or equal	9 >= 99	false
==	equal	9 == 12 + 12	false
!=	not equal	292 != 377	true

LESSON 2

THE AND, OR, AND NOT OPERATORS

The C# conditional operators express the familiar AND and OR operations, which we can use to write conditions such as

```
John's age is greater than 20 AND John's age is less than 35.
John's height is greater than 78.5 OR John's weight is
greater than 300.
```

Figure 4.1 shows the C# symbols for the conditional operators.

Note that the operands of the conditional operators have type bool. The expression age > 20 is either true or false, and so is age < 35.

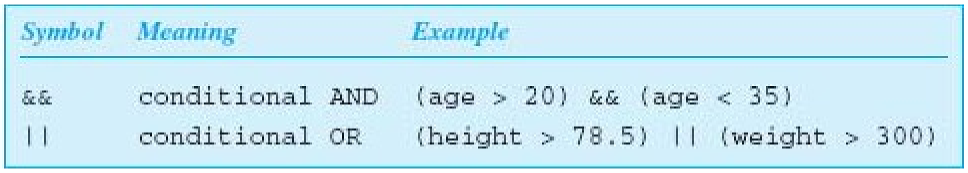
The && and || operators use two-character symbols that must be typed without any space between them. Using & & instead of && would give an error.

Conditional AND

The conditional AND expression (age > 20) && (age < 35) will be true only when both of its operands are true, and false otherwise. If the variable age has the value 25, both operands are true, and the whole && expression is true. If age has the value 17, the first operand, age > 20 is false, and the whole && expression is false. Figure 4.2 shows some sample evaluations of a conditional AND expression, illustrating how the value of an && expression depends on the values of its arguments.

Note that when the first operand is false, as it is when age is 17, we know that the conditional AND is false without even checking the value of the second operand.

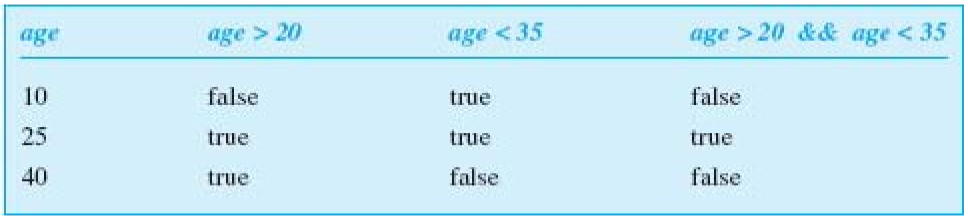
FIGURE 4.1 Conditional operators



Symbol	*Meaning*	*Example*
`&&`	conditional AND	`(age > 20) && (age < 35)`
`||`	conditional OR	`(height > 78.5) || (weight > 300)`

LESSON 2

FIGURE 4.2 Evaluating an example of a conditional AND expression



age	age > 20	age < 35	age > 20 && age < 35
10	false	true	false
25	true	true	true
40	true	false	false

Conditional OR

The conditional OR expression (height > 78.5) || (weight > 300) is true if either one of its operands is true, or if both are true. If height has the value 72 and weight has the value 310, then the first operand is false and the second operand is true, so the || expression is true. Figure 4.3 shows some sample evaluations of a conditional OR expression, illustrating how the value of an || expression depends on the values of its arguments.

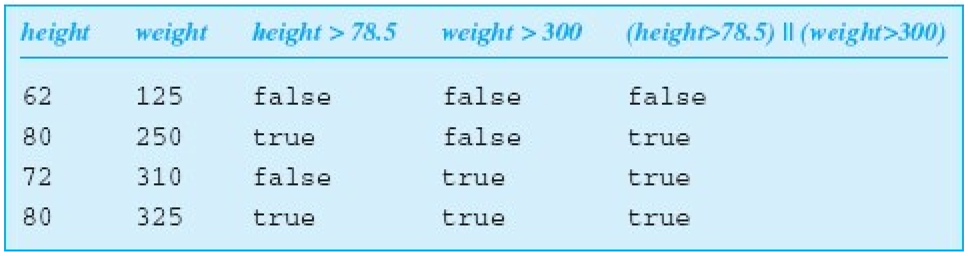
Logical Complement

C# uses the symbol ! for the logical complement, or NOT, operator, which has only one operand. The logical complement negates the value of its operand, as Figure 4.4 shows.

If the bool variable, on, has the value true, then !on is false, but if on is false, then !on is true. Example 4.1 allows the user to enter different values of height and weight and displays the value of conditional AND, conditional OR, and logical complement expressions.

To make our programs easier to read, we abstracted some input and output methods in an IO class. We used several of these methods in Chapter 3 examples. We include the code for the IO class that contains these methods in an addendum at the end of this chapter. Using the IO class is just a convenience. With a loss of readability, we could copy the code for each method instead.

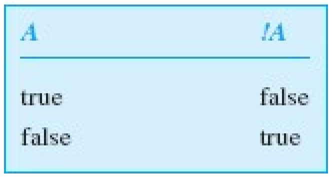
FIGURE 4.3 Evaluating an example of a conditional OR expression



height	weight	height > 78.5	weight > 300	(height>78.5) || (weight>300)
62	125	false	false	false
80	250	true	false	true
72	310	false	true	true
80	325	true	true	true

LESSON 2

FIGURE 4.4 Evaluating a logical complement expression



A	*!A*
true	false
false	true

When using the .NET Framework SDK, we compile Measure.cs using the command

```
csc /r:IO.dll Measure.cs
```

where the /r option indicates that Measure.cs references the file IO.dll containing the compiled code for the IO class. When using Visual C#, we first create a new project, then click on *Project, Add, Add Existing Item* to add Measure.cs to the project. We click on *Project, Add Reference* to add the IO.dll file containing the IO library.

LESSON 2

= =

EXAMPLE 4.1 Measure.cs

```
/*  Evaluates AND, OR, and NOT expressions
 *  involving height and weight measurements.
 */

using System;
public class Measure {

     /* Inputs a height and a weight.
      * Computes boolean-valued expressions.
      */
  public static void Main() {
    double height =
      IO.GetDouble("Enter a height in inches: ");      // Note 1
    double weight = IO.GetDouble("Enter a weight in pounds: ");
    Console.Write("height < 65 && weight < 130 is ");  // Note 2
    Console.WriteLine((height < 65) && (weight  < 130));
    Console.Write("height < 65 || weight  < 130 is ");
    Console.WriteLine((height < 65) || (weight  < 130));
    bool heavy = (weight > 250);                       // Note 3
    Console.WriteLine("heavy is " + heavy);
    Console.WriteLine("!heavy is " + !heavy);
  }
}
```

INPUT AND OUTPUT

The programs in the last section specify the values for variables and constants in the program. To change the initialization

int number1 = 25;

to use the number 30 instead, we would have to edit the program, making the change from 25 to 30, and recompile it again. In this section, we learn to enter values from the keyboard so we can run the program again with different input values without the need to recompile. We used the WriteLine statement without much explanation in the last section, but here we will explore possibilities for output

LESSON 2

==================================

Inputting from the Console

The term **console** reminds older veterans of the early computers that could only display characters and not graphics. We use it now to refer to a window that only displays characters and not graphics

We wish to enter data from the keyboard into the console. A good program will prompt the user with a message describing the data to enter. For example, the statement

```
System.Console.Write("Enter your name: ");
```

displays the message

```
String name = System.Console.ReadLine();
```

and remains on the same line, where the user will enter his or her name.

The statement

```
String name = System.Console.ReadLine();
```

returns a line of text that the user enters by typing the characters on the keyboard and pressing the *Enter* key. When the executing program reaches this ReadLine method, it waits for the user to enter a line of text, which it returns to the program, assigning it to the variable name. The String type represents text. We will learn more about it later. Just to verify that we have received the data, we use the WriteLine method to display it in the console.

EXAMPLE 2.5 InputName.cs

```
// Inputs text from the keyboard.
using System;                                    // Note 1
public class InputName {
  public static void Main( ) {
    Console.Write("Enter your name: ");          // Note 2
    String name = Console.ReadLine();            // Note 3
    Console.WriteLine("    Your name is {0}", name);
  }
}
```

```
Enter your name: Art
    Your name is Art
```

LESSON 2

================================

Note 1: using System;

The System namespace includes the Console class containing the Write, ReadLine, and WriteLine methods which we use in this example. This

line is a using directive. It lets us use the classes in the System namespace without the System prefix. We can use the Console class without prefixing it with System . Without this directive, we must use System.Console to refer to the Console class in the System namespace. Using the directive, we have less keyboarding to do and shorter lines

Note 2: Console.Write(“Enter your name: ”);

The Write statement leaves the cursor on the current line. By contrast, the WriteLine statement moves the cursor to the beginning of the next line. The cursor signals the position for the next input or output operation

Note 3: String name = Console.ReadLine();

When execution reaches the ReadLine statement it waits for the user to enter text and press the *Enter*key

Inputting an Integer

The ReadLine method inputs text. If we enter 23, it will return the string “23”. We are used to reading base 10 numerals, and readily understand this as a symbol for three more than the normal number of fingers and toes combined. Ancient Romans would have written XXIII to represent the same number. AC# program does not use either of these representations for integers. The int type has a Parse method that converts a base 10 representation to the 32-bit binary representation it uses

LESSON 2

==================================

EXAMPLE 2.6 InputInteger.cs

```
/*   Inputs a base 10 numeral from the console.
 *   Converts to an int and outputs twice its value.
 */

using System;
public class InputInteger {
  public static void Main( ) {
    Console.Write("Enter an integer: ");
    int  number1 = int.Parse(Console.ReadLine());
                                                    // Note 1
    int number2 = number1 + number1;
    Console.WriteLine
          ("    Twice the number  is {0}", number2);
  }
}
```

Note 1: int number1 = int.Parse(Console.ReadLine());

Type int is a shorthand for System.int32, so we could have written this method as Int32.Parse.

When we cover user interfaces later, we will have many other ways for the user to input and interact

Output to the Console

We have used the WriteLine and Write methods to write to the console. The invocation

```
System.Console.WriteLine("The number is: {0}", number1);
```

has two arguments. The first is a string and the second is a value that C# will convert to a string for display. The format specifier, {0}, indicates the position where the second argument will appear in the string

To output more than one number we use additional format specifiers, {0}, {1}, {2}, and so on. For each specifier, we include an argument to replace it in the output string. Thus if number1 is 35 and number2 is 70, the statement

LESSON 2

==================================

```
Console.WriteLine("The number is {0} and twice it is {1}.",
                  number1, number2);
```

will output

```
The number is 35 and twice it is 70.
```

We can output the data in any order. For example,

```
Console.WriteLine
        ("Twice the number is {1} and the number is {0}",
          number1, number2);
```

outputs

```
Twice the number is 70 and the number is 35.
```

The numbers 0 and 1 in {0} and {1} indicate an argument position after the format string in the WriteLine invocation. We can also specify a specific format. For example, {0:C} would display as currency. In the United States, 35 dollars displays as $35.00

EXAMPLE 2.7 OutputFormat.cs

```
// Outputs to the console.
using System;
public class OutputFormat {
  public static void Main( ) {
    int number1 = 35;

      int number2 = number1 + number1;
      Console.WriteLine
                ("The number is {0} and twice it is {1}.",
                  number1, number2);
      Console.WriteLine
                ("{0} is the number and {1} is its double.",
                  number1, number2);
      Console.WriteLine
               ("Twice the number is {1} and the number is {0}",
                  number1, number2);
      Console.WriteLine
                ("The first is {0:C} and twice it is {1:C}.",
                  number1, number2);
    }
  }
```

```
The number is 35 and twice it is 70.
35 is the number and 70 is its double.
Twice the number is 70 and the number is 35.
The first is $35.00 and twice it is $70.00.
```

LESSON 2

================================

Outputting a Table

We would like to output a table in which the columns have a fixed width. We might want to align names to the left of the column and numbers to the right. To achieve this format we use the specification {0, -10} to left-align names in a field of length 10, and {1, 10} to right-align numbers in a field of size 10

EXAMPLE 2.8 OutputTable.cs

```
/*  Formats a table with fixed column lengths and
 *  either right or left alignment.
 */

using System;
public class OutputTable {
  public static void Main( ) {
    Console.WriteLine("{0,-10}{1,10}", "Names", "Numbers" );
    Console.WriteLine
            ("{0,-10}{1,10}", "Sheila", 12345);
                                             // Note 1
    Console.WriteLine("{0,-10}{1,10}", "Frances", 241);
    Console.WriteLine("{0,-10}{1,10}", "Michael", 4141);
  }
}
```

Note 1: `Console.WriteLine`

```
            ("{0,-10}{1,10}", "Sheila", 12345);
```

We can output literals as well as values of variables. A **literal** is a source code representation of a value. Here “Sheila” is a string and 12345 is an integer