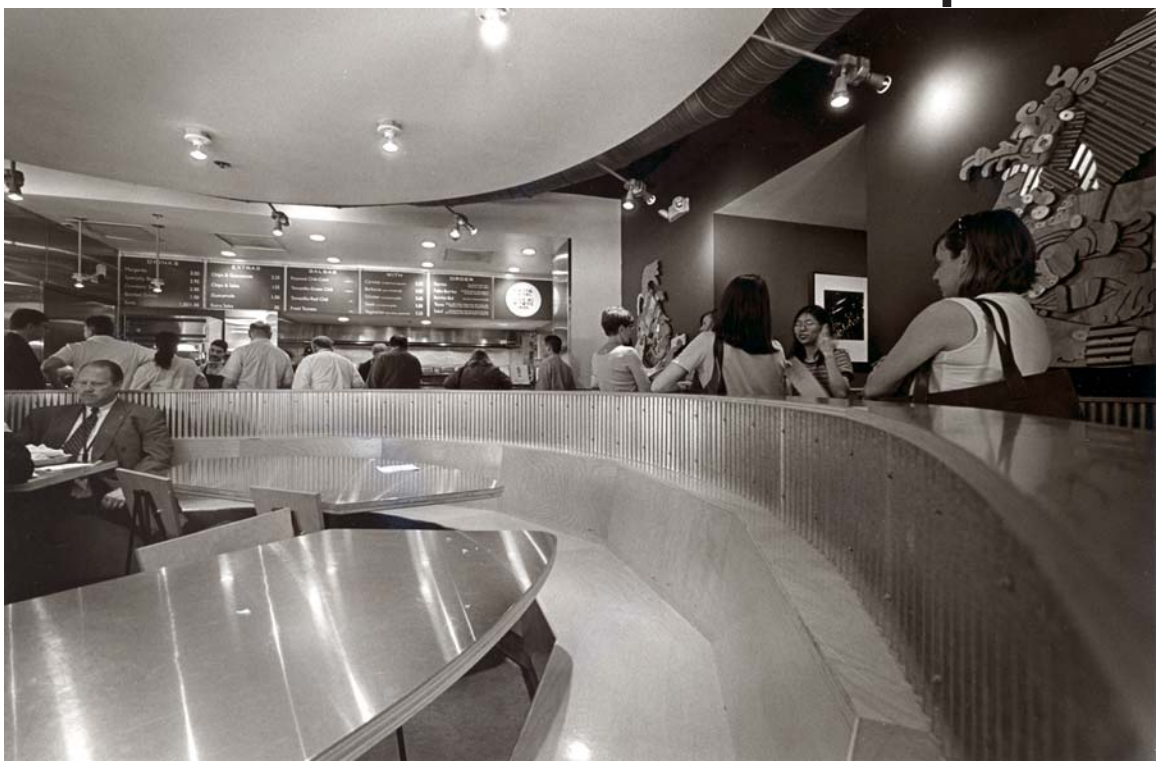
Chapter 6



Voigtlander Bessa-L / 15mm Super Wide-Heliar

Chipotle – Rosslyn, VA

Simple C# Programs

Learning Objectives

- *State the required parts of a simple console application*
- *State the definition of the terms: "Application", "Assembly", and "Module"*
- *State the purpose of the Main() method*
- *Describe the differences between the four different versions of the Main() method*
- *State the purpose of the "using" directive*
- *Describe the differences between value types and reference types*
- *State the purpose of statements, expressions, and operators*
- *State the purpose of the "new" operator*
- *Apply the "new" operator to dynamically create objects in memory*
- *List and describe the use of the C# operators*
- *List and describe the use of the C# reserved keywords*
- *Demonstrate your ability to create simple C# programs*
- *Demonstrate your ability to compile C# programs using the command-line compiler*

Introduction

This chapter lays a solid foundation for the understanding of the material contained within the remaining chapters of this book. Here you will learn the fundamental concepts crucial to building C# applications. As the old adage goes, you must learn to crawl before you can walk. Soon you will be running. But in the meantime, you will begin to wonder whether you will ever get off the floor!

Try as I may to make the material contained here easy to understand and free of confusing concepts, I am hindered in doing so by the very nature of the C# language. For example, the simplest program you can write in C# must be contained within a class. Thus, the concept of a class is forced upon you when it would be nice to delay its discussion until later.

The primary challenge facing both students and teachers of a modern object-oriented programming language like C# is the multitude of complexities presented by both the language itself and its accompanying collection of framework classes, referred to in the case of C# as the .NET Framework Application Programming Interface (API). I will mitigate this complexity in this chapter by keeping the example programs small and concise, and by limiting the use of .NET Framework API classes to those required for simple console input and output.

I will focus my efforts on helping you understand the C# type structure and understanding the differences between value types and reference types. I will explain to you the purpose of the Main() method, and show you how to use value type and reference type objects within a Main() method. I will also show you how to use variables and constants in simple programs. I will then discuss the C# language operators and demonstrate their use.

If you are completely new to programming, even the material I talk about in this chapter can be intimidating. Be patient and keep at it. A keen grasp of the fundamentals pays big dividends when you start to tackle more complex concepts.

What Is A C# Program?

When I say to you, "Write a program in C# to do this or that...," what do I mean? There are many answers to this question, and all of them are correct. Each depends on the complexity of the problem being solved and the particular approach you might take towards its solution. For example, as you will soon see later in this chapter, if I ask you to write a program that adds two numbers and displays the result on the screen, you can write this program as a console application contained in one class. The effort spent analyzing the problem (*i.e.*, adding two numbers and displaying the sum) will be minimal.

Another approach to writing the simple adding program might involve the use of graphical user interface (GUI) components so that users could enter the numbers to be added in a familiar window interface. This version of the program could be written either as one class or as multiple classes. It depends on how you approach the design of the program. The GUI version of the program also would use more of the .NET API classes to create the window and handle user interactions within the interface.

The approach you take to the design of a program depends largely on how much you know about designing programs. As you progress through this book, you will learn the C# language and program design concepts hand-in-hand. At first you will see examples of simple, one-class programs. As you are introduced gradually to object-oriented programming concepts, your knowledge of program design will increase and you will be able to build more complex programs.

So, when I say, "Write a program in C# to do this or that..., " what you do might be as simple as creating one class and adding a few lines of code to do a simple operation. This simple program will be contained in one source file. For more complex programming projects, you may need to spend considerable time analyzing the problem at hand and designing a suitable solution using object-oriented analysis, design, and programming techniques. The resulting program may be spread across multiple files. But first, you must learn to crawl.

A Simple Console Application

In this section you will learn how to build simple, one-class console applications. This one-class program design will serve as the basis for demonstrating many fundamental concepts throughout this and several later chapters.

Definition Of Terms: Application, Assembly, Module, and Entry Point

An *application* is an *assembly* that has an *entry point*. In the words of the Common Language Infrastructure (CLI) standard: "An assembly is a configured set of loadable code modules and other resources that together implement a unit of functionality," and, "A module is a single file that can be executed by the virtual execution system (VES)." (See Chapter 4 for a discussion of the VES). These are nice definitions, but are rather technical and somewhat misleading — and in dire need of explanation.

The entry point is a fancy name for where a program can begin execution. When an application is loaded into the VES, it has to start execution at some point in the code. This point is referred to as the entry point. The entry point is nothing more than a special method known as the Main() method. I will talk more about the Main() method shortly.

Essentially, you can compile a C# program into either an *assembly* or a *module*. If your program has one class that contains a Main() method, then it can be compiled directly into an assembly. This is a file that has a ".exe" filename extension. You can execute this file by typing its name at a command prompt or by double-clicking its icon. If your program has no Main() method, you will get the following error message when you compile:

```
'[assembly file name]' does not contain a static 'Main' method suitable for an entry point
```

A program with no Main() method can be compiled into a module using the csc compiler's `/target:module` argument. But even if a module contains executable code, it can't be loaded and run standalone by the virtual execution system if it doesn't have a Main() method. The VES doesn't know where to start execution.

Modules can be added to assemblies. In fact, modules of different languages that conform to the CLI specification can be combined with modules written in C# to form an executable assembly. Cross-language compatibility is one of the promises of both the .NET and the broader CLI initiative.

Structure Of A Simple Application

Example 6.1 gives the code for a simple C# application.

6.1 SimpleApp.cs

```
1    using System;
2
3    public class SimpleApp {
4       static void Main() {
5            Console.WriteLine("Howdy Stranger!");
6         }
7    }
```

Referring to Example 6.1 — SimpleApp is the name of the class that contains the Main() method. On line 1, a *using directive* signals the compiler that this source file refers to classes and constructs declared within the System namespace. (I cover namespaces in Chapter 9.) Specifically, in this short program I am using the System.Console class to get input from and send output to the command console.

Line 3 includes the keywords *public* and *class* to declare the class SimpleApp. At the end of line 3 there appears an opening curly brace '`{`'. This signals the beginning of SimpleApp's class *body*. Everything belonging to a class, that is all *fields*, *properties*, *methods*, etc., appear in the class body between the opening and closing curly braces.

The start of the Main() method begins on line 4. The keywords *static* and *void* are used to declare the Main() method. The Main() method, as you can see, contains an opening and closing parentheses "`( )`". The parentheses denote the beginning and ending of an optional *method parameter-list*. A *parameter* represents an object that will be passed to a method for processing when the method is called. (I'll talk more about Main() method parameters later.)

At the end of line 4, an opening curly brace denotes the beginning of the Main() method body. Any code appearing between the Main() method's opening and closing curly braces belongs to the Main() method. In the case of Example 6.1, the Main() method contains one line of code, line 5, which is a method call to the Console class's Write-

Line() method. The WriteLine() method writes different types of objects to the console. In this case, I'm writing a string of characters (*i.e.,* a String object) to the console. Line 6 contains the Main() method's closing curly brace '}' and line 7 contains the SimpleApp class's closing curly brace.

To compile this program at the command prompt, you would save the source code in a file named SimpleApp.cs and use the csc compiler tool like so:

```
csc SimpleApp.cs
```

This creates an assembly named SimpleApp.exe. Figure 6-1 shows the results of running this program.

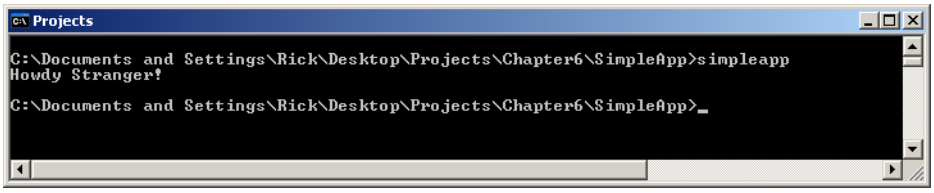


Figure 6-1: Results of Running Example 6.1

Purpose Of The Main() Method

The purpose of the Main() method is to provide an entry point for application execution. As I stated earlier, without a Main() method, the virtual execution system has no way of knowing where to start running a program.

Main() Method Signatures

The Main() method can have the following four *signatures*:

```
static void Main() { }
static void Main(string[] args) { }
static int Main() { }
static int Main(string[] args) { }
```

The term *method signature* refers to the combination of a method's name and its parameter list. A method's return type is *not* considered part of its signature, but the Main() method can optionally return an integer value, which yields four different versions. As these four method signatures show, the Main() method can return void (nothing) or optionally an integer value, and take either no parameters or a string array parameter.

The purpose of the string array parameter is to enable the passing of command-line arguments to the program. I will show you how to do this in Chapter 8 after you learn about arrays.

The SimpleApp class shown in Example 6.1 used the first version of Main(). It could have easily used the other versions as well. Example 6.2 shows the SimpleApp class employing the second version of the Main() method.

6.2 SimpleApp (Version 2)

```
1    using System;
2
3    public class SimpleApp {
4       static void Main(string[] args) {
5            Console.WriteLine("Howdy Stranger!");
6         }
7    }
```

Referring to Example 6.2 — this version of the SimpleApp class produces the same output when executed as that shown in Figure 6-1. In this case, the string array parameter named args is ignored.

Keep in mind that the only four versions of the Main() method authorized as entry points are those shown above. If you tried to use a method named Main() that took a different type or number of parameters, then you would receive a compiler warning. Let's see what happens if we try to use a different Main() method argument type. Example 6.3 gives the code.

6.3 SimpleApp (Version 3)

```
1    using System;
2
3    public class SimpleApp {
4       static void Main(int i) { // will not compile!
5            Console.WriteLine("Howdy Stranger!");
6         }
7    }
```


Referring to Example 6.3 — the Main() method string parameter has been replaced with an integer parameter. When you compile this version of the program, it produces the error messages shown in Figure 6-2.

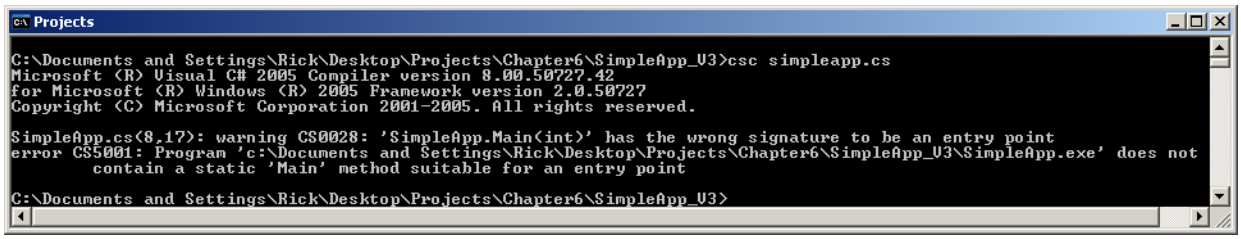


Figure 6-2: Results of Compiling Example 6.3 with Improper Main() Method Signature

Quick Review

A simple C# application is a class that contains a Main() method. The purpose of the Main() method is to provide an entry point for program execution. There are four authorized Main() method signatures. A class that contains a Main() method can be compiled into an executable assembly. A class with no Main() method can be compiled into a module. Modules can be added to assemblies. Modules created in CLI compliant languages other than C# can be compiled with C# modules to form executable assemblies.

Identifiers And Reserved Keywords

Table 6-1 lists C# language reserved keywords.

abstract	do	in	protected	true
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe
catch	false	namespace	short	ushort
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	void
const	for	operator	string	volatile
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	implicit	private	throw	

Table 6-1: C# Reserved Keywords

Referring to Table 6-1 — there's no need to memorize the entire list. In time, as you write increasingly complex programs, you will come to know most of them intimately. The important thing to note right now is that reserved keywords have special meaning in the C# language. You can't hijack them for your own purpose.

In the SimpleApp code shown in Example 6.1, you saw several keywords put to use. These included *class*, *public*, *static*, *void*, *string*, and *using*. The *class* keyword is used to introduce a new class type name, in this case the string of characters "SimpleApp". The string of characters "SimpleApp" is known as an *identifier*. The act of programming requires you to invent names for lots of things in your programs like variables, constants, class and method names. So long as the names you choose for these objects are different from the reserved keywords, you'll be fine. But what would happen if you were to try and introduce a new name for an object within your program that has already been reserved? Let's see what happens. Example 6.4 gives the code for a naughty little program that tries to declare a class named "class".

6.4 Naughty Program

```
1    using System;
2
3    public class class { // <-- will cause an error when compiled
4      static void Main(){
5        Console.WriteLine("Bad, bad program...!");
6      }
7    }
```

Referring to Example 6.4 — on line 3 an attempt is made to introduce a new class named "class". But since class is a reserved keyword, this causes the compiler to pitch quite a fit, as is shown in Figure 6-3.

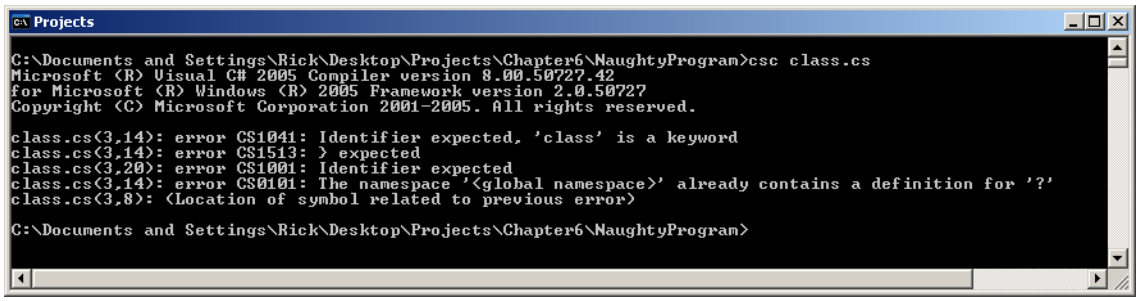


Figure 6-3: Errors Produced when Attempting to Reintroduce a Reserved Keyword

Identifier Naming Rules

It's easy to avoid trouble in formulating identifier names if you take the time to give the objects in your programs names that make sense within the context of the problem at hand. Creating valid identifiers is easy, as you'll see. What takes a little more skill is effectively naming objects within a program that correspond to real world objects in the problem domain. For more information on this topic, see the discussion on isomorphic mapping in Chapter 1.

Identifiers can start with a letter or the underscore '_' character. The starting letter can be uppercase or lowercase. The starting character can be followed by any number of letters, underscores and decimal digits. Unicode character escape sequences can be used as well, but putting these in your identifiers makes them difficult to read and understand.

Although I said earlier that reserved keywords cannot be used as identifiers, I will recant somewhat and say that if you add the '`@`' character in front of a keyword, you can use it as an identifier. Example 6.5 shows how this is done.

6.5 Somewhat Bad Program

```
1    using System;
2
3    public class @class { // <-- This will work...!
4      static void Main(){
5        Console.WriteLine("Works but not recommended...!");
6      }
7    }
```

Referring to Example 6.5 — the '`@`' character is added to the beginning of the second occurrence of the class keyword on line 3, which now forms a valid identifier. (The identifier is "`@class`".) Although this works, I don't recommend doing this as the inevitable result will be code that's hard to read, understand, and maintain. I leave it up to you to compile Example 6.5 and see for yourself the results of its execution.

Quick Review

Identifiers are sequences of characters that represent names of objects in a program. Identifiers are used to formulate the name of classes, structures, methods, variables, constants, properties, fields, enums, etc.

Identifiers can start with either an uppercase or lowercase letter or an underscore '_' character followed by any number of letters, digits, and underscores.

Reserved keywords are identifiers that have special meaning within the C# language. You cannot reintroduce a reserved keyword as a name for an object within your program. You can, however, prepend the '@' symbol to a reserved keyword to formulate a valid identifier, but I discourage you from doing this as it renders code hard to read, understand, and maintain.

Types

C# is a strongly typed programming language. The term strongly typed means that all objects in a C# program must be associated with a particular type. An object's type is a specification of the legal operations that can be performed on that object. For example, the '+' operator can be applied to integer (int) objects, and the Append() method can be called on StringBuilder objects. Generally speaking, if you try to perform an operation on an object that its particular type does not allow, you will get a compiler error.

There are two categories of types in the C# language: *value types* and *reference types*. Figure 6-4 gives the complete C# type hierarchy.

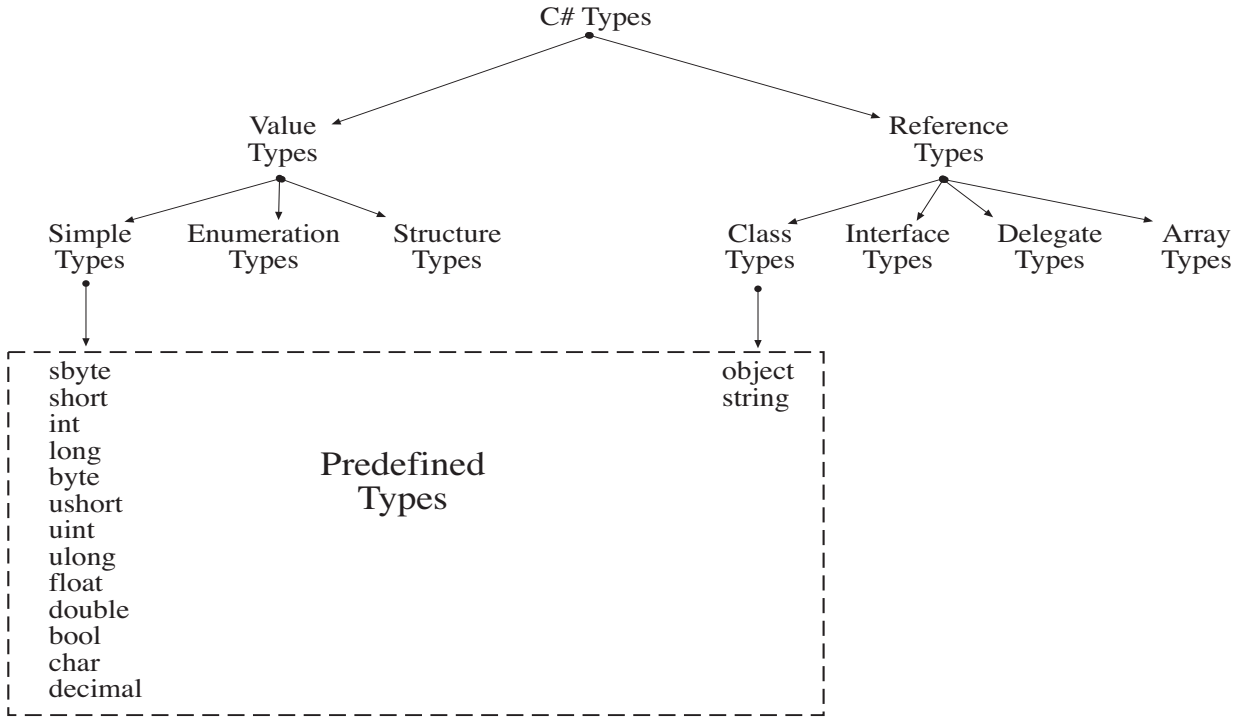


Figure 6-4: C# Type Hierarchy

Referring to Figure 6-4 — value types and reference types each have several type subcategories. The important thing to note in Figure 6-4 is the existence of the C# predefined types. These are the types that are built into the language. Notice that the predefined type names all start with lowercase letters. All but two of the predefined types are simple value types. The types *object* and *string* are class types, which is a subcategory of reference types. Value types behave differently from reference types. I explain these behavioral differences in the next section.

Value Type Variables vs. Reference Type Variables

This section explains the differences between value type and reference type variables.

Value Type Variables

A value type variable contains its very own copy of its data. Let's take a look at a simple example of value types in action.

6.6 ValueTypeTest.cs

```
1    using System;
2
3    public class ValueTypeTest {
4      static void Main(){
5        int i = 0;
6        int j = i;
7        j = j+1;
8        Console.WriteLine("The value of i is: " + i);
9        Console.WriteLine("The value of j is: " + j);
10     }
11   }
```

Referring to Example 6.6 — An integer value type variable named i is declared and initialized on line 5. The term *variable* means a named storage location in memory whose value can be changed during program execution. On line 6, another integer variable named j is declared and initialized to the value of i. On line 7, a simple addition operation is performed on the variable j adding 1 to its value. Adding 1 to the variable j does not affect the value of the variable i. The code on lines 8 and 9 print the values of i and j to the console. Figure 6-5 shows the results of running this program.

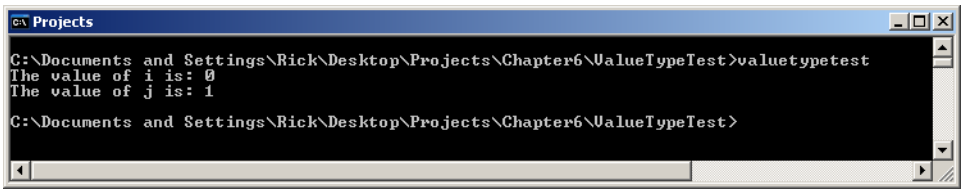


Figure 6-5: Results of Running Example 6.6

Reference Type Variables

A reference type variable contains the memory address of a reference type object. Two different reference type variables can point to the same reference type object in memory. The following program offers an example.

6.7 ReferenceTypeTest.cs

```
1    using System;
2    using System.Text;
3
4    public class ReferenceTypeTest {
5      static void Main(){
6        StringBuilder sb1 = new StringBuilder();
7        StringBuilder sb2 = sb1;
8        sb1.Append("Howdy Pawdner!");
9        Console.WriteLine(sb1);
10       Console.WriteLine(sb2);
11     }
12   }
```

Referring to Example 6.7 — on line 2, another using directive provides shortcut name access to the StringBuilder class located in the System.Text namespace. On lines 6 and 7, in the body of the Main() method, two StringBuilder reference variables named sb1 and sb2 are declared and initialized. Notice that in order to create a StringBuilder object, you must use the *new* operator as is shown on line 6. On line 7, the StringBuilder variable named sb2 is initialized to the same value as sb1. Remember, reference type variables store memory addresses to objects located in memory. So, when the value of sb1 is assigned to sb2, sb2 is being assigned a memory address. Now sb1 and sb2 both "point" to or "reference" the same StringBuilder object in memory. Any operation performed on the object pointed to by sb1 affects the object pointed to by sb2 since, in this case, it is the same object. This is what happens when the Append() method is called via the sb1 variable adding the character string "Howdy Pawdner!". Note the results of running this program shown in Figure 6-6.

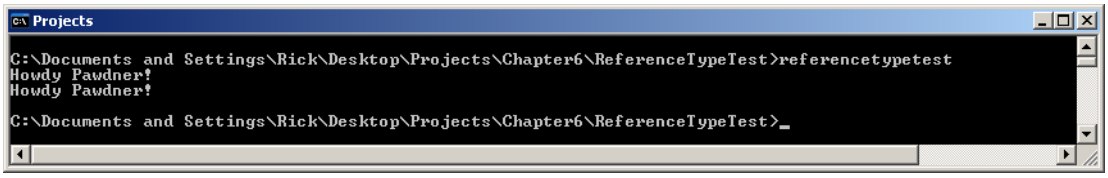


Figure 6-6: The Results of Running Example 6.7

Maybe Some Pictures Will Help

Figure 6-7 offers a simple conceptual view of value-type memory allocation based on the code presented in Example 6.6.

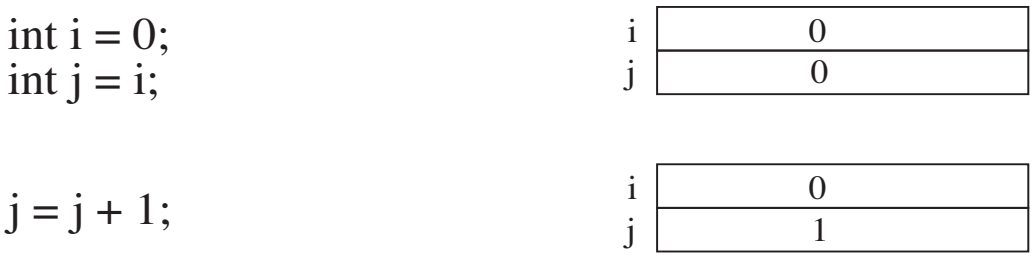


Figure 6-7: Value Type Memory Allocation

Referring to Figure 6-7 — the integer variables i and j each hold their very own copy of their assigned values. When the value of variable i is assigned to the variable j, a copy of i's value, in this case 0, is made and stored in j's memory location.

Figure 6-8 shows a simple conceptual view of reference type memory allocation based on the code presented in Example 6.7.

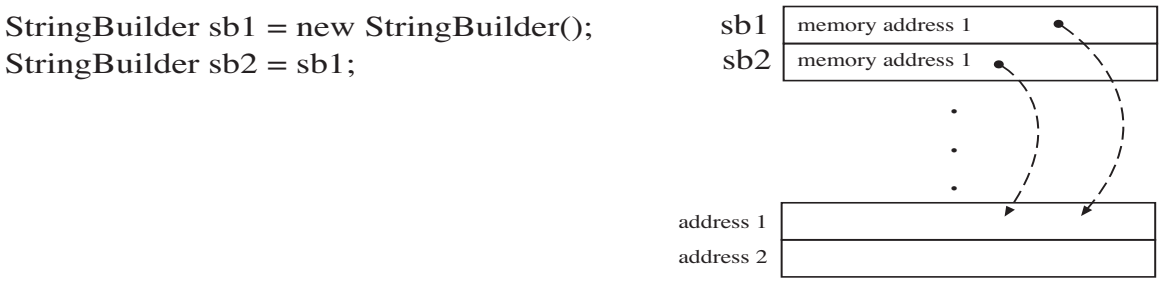


Figure 6-8: Reference Type Memory Allocation

Referring to Figure 6-8 — the StringBuilder variable sb1 contains the memory address of a StringBuilder object. The StringBuilder object was created with the expression "new StringBuilder()" which creates the object and returns the address of the object's location in memory. When the value of sb1 is assigned to the variable sb2, both variables will point to the same object in memory. When two reference variables point to the same object, any operation performed on one affects the other, as is shown in Figure 6-9.

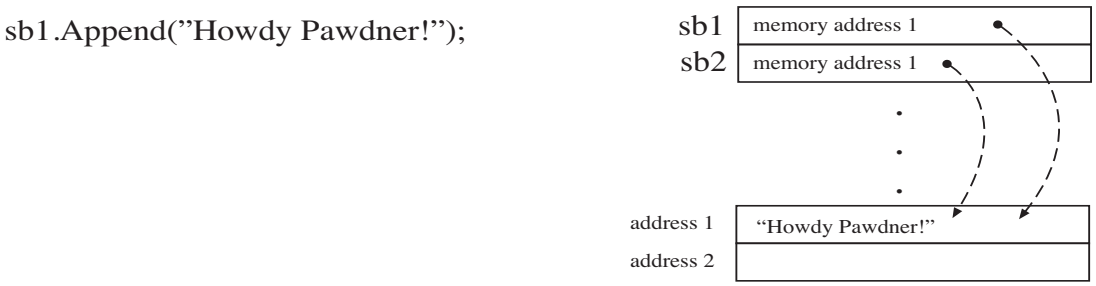


Figure 6-9: Results of Calling the Append() Method via the sb1 Variable

Mapping Predefined Types To System Structures

All the predefined types correspond to structures within the System namespace of the .NET API. For example, the predefined simple type *int* is mapped to the *System.Int32* structure. The System.Int32 structure inherits from the *System.ValueType* class, as do all value types and enumerations. Example 6.8 gives an alternative version of the ValueTypeTest code originally presented in Example 6.6.

6.8 ValueTypeTest.cs (Version 2)

```
1    using System;
2
3    public class ValueTypeTest {
4      static void Main(){
5        Int32 i = 0;
6        Int32 j = i;
7        j = j+1;
8        Console.WriteLine("The value of i is: " + i);
9        Console.WriteLine("The value of j is: " + j);
10     }
11   }
```

Referring to Example 6.8 — compare this program with Example 6.6. Notice the only difference between the two programs is the substitution here of the type Int32 for the simple type int. Table 6-2 lists the predefined types along with their corresponding System namespace structures, default values, and value ranges.

Type	Description	System Namespace Structure or Class	Default Value† / Value Range
object	The base class of all types	Object Class	Default value: null
string	A sequence of Unicode code units	String Class	Default value: null
sbyte	8-bit signed integral type	SByte Structure	Default value: 0 -128 to 127
short	16-bit signed integral type	Int16 Structure	Default value: 0 -32768 to 32767
int	32-bit signed integral type	Int32 Structure	Default value: 0 -2,147,483,648 to 2, 147,483,647
long	64-bit signed integral type	Int64 Structure	Default value: 0 -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
byte	8-bit unsigned integral type	Byte Structure	Default value: 0 0 to 255
ushort	16-bit unsigned integral type	UInt16 Structure	Default value: 0 0 to 65535
uint	32-bit unsigned integral type	UInt32 Structure	Default value: 0 0 to 4,294,967,295
ulong	64-bit unsigned integral type	UInt64 Structure	Default value: 0 0 to 18,446,744,073,709,551,615
float	single-precision floating point type	Single Structure	Default value: 0.0 -3.402823e38 to 3.402823e38
double	double-precision 64-bit floating point type	Double Structure	Default value: 0.0 -1.79769313486232e308 to 1.79769313486232e308

Table 6-2: Predefined Type Mappings, Default Values, and Value Ranges

Type	Description	System Namespace Structure or Class	Default Value† / Value Range
bool	Represents true or false	Boolean Structure	Default value: false true or false
char	character type (Unicode code unit)	Char Structure	Default value: \u0000 Any Unicode value
decimal	decimal type with at least 28 significant digits	Decimal Structure	Default value: 0 -79,228,162,514,264,337,593,543,950,335 to 79,228,162,514,264,337,593,543,950,335
† Default values are assigned to class or structure fields. Local method variables must be explicitly assigned.			

Table 6-2: Predefined Type Mappings, Default Values, and Value Ranges

Quick Review

C# has two kinds of types: value types and reference types. Value type variables contain the actual data as defined by the type. Reference type variables contain a reference to an object in memory. Two or more reference type variables can reference the same object in memory. The C# predefined types map to structures within the System namespace. System.Object is the base type for all types.

Statements, Expressions, and Operators

Statements are the fundamental building blocks of C# programs. A statement can be thought of as the smallest standalone element of a program, and programs are built using sequences of statements. The simplest type of statement is the *empty statement*. An empty statement would look like this:

```
;
```

It's just a lonely semicolon on a line by itself, although it doesn't have to be on a line by itself.

You've already seen statements in action in this chapter's example programs. The following line of code is taken from Example 6.6:

```
int i = 0;
```

This is an example of a *local variable* declaration statement. It's a local variable declaration because this line of code appeared within the body of a method, in this case, the Main() method. This variable declaration statement contains within it an expression statement. The assignment operator '=' assigns the value 0 to the variable i. Complex statements can be formed by combining statements within statements.

Notice that the statement above is terminated by the semicolon '`;`' character. The semicolon character indicates a line of execution. Note the following three lines of code:

```
1    int i = 0;
2    int j = i;
3    int j = j + 1;
```

The results of the execution of line 1 will be fully complete before line 2 begins execution. And again, the results of line 2 will be fully available when line 3 begins execution.

Statement Types

There are nineteen different types of statements in the C# language. These are listed in Table 6-3.

Statement	Statement Lists and Block Statements
	Labeled Statements and `goto` Statements
	Local Constant Declarations
	Local Variable Declarations
	Expression Statements
	`if` Statements
	`switch` Statements
	`while` Statements
	`do` Statements
	`for` Statements
	`foreach` Statements
	`break` Statements
	`continue` Statements
	`return` Statements
	`yield` Statements
	`throw` Statements and `try` Statements
	`checked` and `unchecked` Statements
	`lock` Statements
	`using` Statements

Table 6-3: C# Statement Types

Referring to Table 6-3 — a statement can be any of the statement types listed in the right column. All of these statement types are discussed throughout the book, so I will not give examples of each here. With sufficient programming experience, their use becomes second nature. I will, however, elaborate on how the different types of C# operators are used in expression statements. This is the topic of the next section.

Operators And Their Use

Table 6-4 lists the C# operators by expression category and precedence.

Expression Category	Operators
Primary	`x.y f(x) a[x] x++ x-- new typeof checked unchecked`
Unary	`+ - ! ~ ++x --x (T)x`
Multiplicative	`* / %`
Additive	`+ -`

Table 6-4: Operator Categories by Precedence

Expression Category	Operators
Shift	`<< >>`
Relational and Type-Testing	`< > <= >= is as`
Equality	`== !=`
Logical AND	`&`
Logical XOR	`^`
Logical OR	`|`
Conditional AND	`&&`
Conditional OR	`||`
Conditional	`?:`
Assignment	`= *= /= %= += -= <<= >>= &= ^= |=`

Table 6-4: Operator Categories by Precedence

Operator Precedence and Associativity

The term *operator precedence* refers to the order in which the C# compiler evaluates the operators in an expression statement. Consider for a moment the following line of code:

```
int i = 25 - 2 * 2;
```

The variable i is being assigned some value. But what value? If you leave it to the compiler to apply its precedence rules, the variable i will be assigned the value 21. The multiplication operator '`*`' has a higher precedence than the subtraction operator '`-`'. This may or may not be the way you want the expression to evaluate.

The term *associativity* refers to the direction in which the C# compiler performs a series of operations. Binary arithmetic operators like the multiplication and subtraction operators used above have left-to-right associativity. For example, given an expression of the form `2-2-2` the order in which the subtraction operations are performed is `(2-2)-2`. In the example statement given above, the expression `2*2` was performed first because the multiplication operator has a higher precedence than the subtraction operator.

Assignment operations have right-to-left associativity. Thus, the compiler evaluates an expression of the form `i=j=k` as `i=(j=k)`.

Forcing Operator Precedence and Associativity Order With Parentheses

You can force the compiler to evaluate a complex expression a particular way by using parentheses. If we apply parentheses to the expression shown above in the following manner:

```
int i = (25 - 2) * 2;
```

This will cause the subtraction operator '`-`' to be evaluated before the multiplication operator '`*`', yielding the value 46. It's good programming practice to always use parentheses to show how you intend an expression to be evaluated. Doing so eliminates the possibility of making hard-to-find mistakes and makes your code easier to read and understand.

Operators and Operands

Operators are applied to operands. For example, in the following expression fragment:

```
25 - 2
```

The subtraction operator takes two operands. In the following code fragment:

```
i = 25 - 2
```

The subtraction operation with its two operands is evaluated first, yielding a value of 23. This leaves two operands for the assignment operator '=' to work on: i and 23. As you will soon see, some operators operate on one operand, some on two operands, and one on three operands.

Operator Usage Examples

In this section, I demonstrate the use of one or more operators from each of the operator categories listed in Table 6-4. You will most assuredly encounter all of these operators in more depth as you progress through the book.

Primary Expression Operators

Primary expression operators have the highest precedence. The use of parentheses with these is not usually necessary, nor legal in some cases, to force an unnatural association. You've seen several primary expression operators in action already in this chapter. These included the `new` operator and the member access '`.`' operator.

The new operator creates a reference type object. The member access operator is used to access object members. Consider, for example, the following two lines of code:

```
StringBuilder sb1 = new StringBuilder();
sb1.Append("Adding this string to the sb1 object.");
```

The `new` operator creates a new StringBuilder object in memory. The assignment operator assigns the resulting memory address to the StringBuilder reference variable sb1. The StringBuilder's Append() method is called via the sb1 variable with the help of the member access operator.

Two other primary operators you will frequently use are the postfix increment and decrement operators, '`++`' and '`--`' respectively. Example 6.9 shows the operators in use.

6.9 PrimaryOperatorTest.cs

```
1    using System;
2
3    public class PrimaryOperatorTest {
4      static void Main(){
5        int i = 0;
6        Console.WriteLine(i++); // writes value of i then increments
7        Console.WriteLine(i--); // writes value of i then decrements
8        Console.WriteLine(i);   // simply writes value of i
9      }
10   }
```

Referring to Example 6.9 — an integer variable named i is created on line 5. On lines 6 and 7 the increment and decrement operators are applied to the variable i, which is being used as an argument to the Console.WriteLine() method. Notice on line 6 that the increment operator appears to the right of i. This is the postfix application of this operator, which means "increment the value of i after the statement has been evaluated." The effects here are that the value 0 is written to the console.

On line 7, the decrement operator appears to the right of i. The effect is that the current value of i, which is now 1, is printed to the console and then decremented. Line 8 simply prints the last value of i to the screen. Figure 6-10 shows the results of running this program.

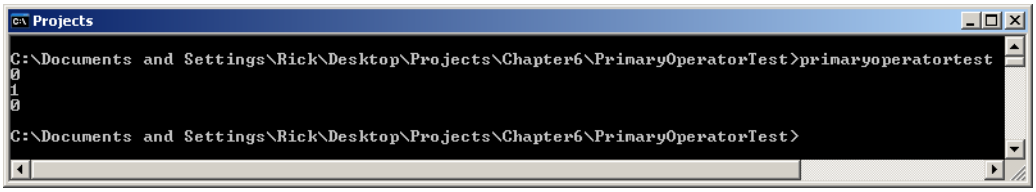


Figure 6-10: Results of Running Example 6.9

Unary Expression Operators

Unary expression operators operate on one operand. The unary expression operators include the prefix increment '`++`' and decrement '`--`', the plus '`+`' and minus '`-`', the logical negation '`!`', the bitwise complement '`~`', and the cast '`(`T_2`)`T_1'. The cast operator forces a change from one type T_1 to another type T_2. The plus and minus unary operators change the sign of integral and floating point numbers. The logical negation operator changes the value of

boolean expressions from false to true and vice versa. The bitwise complement operator switches the bit values of unsigned integral types. (*i.e.,* If a bit is set to 1 it will be changed to 0.)

Example 6.10 offers a short program showing some of these operators in action.

6.10 UnaryOperatorTest.cs

```
1    using System;
2
3    public class UnaryOperatorTest {
4      static void Main(){
5        int i = 25;
6        bool bool_var = true;
7        uint j = 1;
8        Console.WriteLine(-i);
9        Console.WriteLine(!bool_var);
10       Console.WriteLine(~j);
11     }
12   }
```

Referring to Example 6.10 — on line 5, an integer variable named i is declared and initialized to the value 25. On line 6, a boolean variable named bool_var is declared and initialized to the value true. On line 7, an unsigned integer (uint) variable named j is declared and initialized to the value 1. Each of these variables is then printed to the console after its value has been affected by the various unary operators. Figure 6-11 gives the results of running this program.

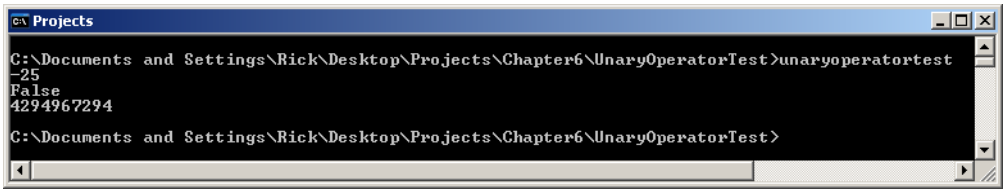


Figure 6-11: Results of Running Example 6.10

Multiplicative Expression Operators

The multiplicative expression operators include the multiplication '`*`', division '`/`', and the remainder '`%`' operators. The remainder operator '*%*' is also referred to as the *modulus* operator.

The multiplicative operators are binary operators in that they operate on two operands. You are already familiar with the notion of how the multiplication and division operators work from your elementary arithmetic background. What you need to be acutely aware of, however, is how each of these operators behaves given different types of numbers. For example, what happens if you multiply two numbers and try to assign the result into a variable type that's too small to accommodate the resultant value? What happens if you divide two integer values vs. two floating point values? If you always keep in mind the relative range of values the different simple types can represent, you will avoid most problems. Operations that attempt to assign a large value to a type that's too small to represent it will result in both a loss of precision and in a compiler warning.

The remainder operator performs a division operation on integral values and returns only the remainder. Example 6.11 shows the remainder operator in action.

6.11 RemainderOperatorTest.cs

```
1    using System;
2
3    public class RemainderOperatorTest {
4      static void Main(){
5        int i = 10;
6        int j = 5;
7        int k = 3;
8        Console.WriteLine(i%j);
9        Console.WriteLine(i%k);
10     }
11   }
```

Referring to Example 6.11 — three integer variables i, j, and k are declared and initialized to the values 10, 5, and 3, respectively. Line 8 prints out the result of the remainder operator applied to the variables i and j. Line 9 prints out the result of the remainder operator applied to the variables i and k. Figure 6-12 shows the results of running this program.

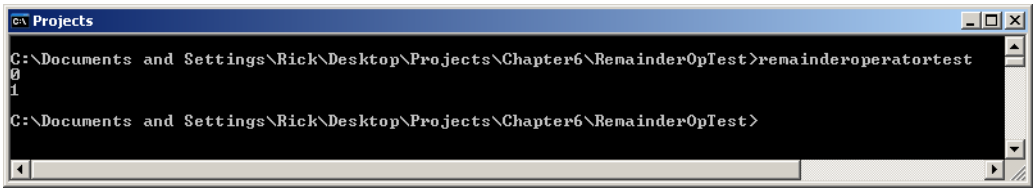


Figure 6-12: Results of Running Example 6.11

Additive Expression Operators

The additive expression operators include the arithmetic addition '+' and subtraction '-' operators. I will forego an example of these operators as they are easy and intuitive to use.

Shift Expression Operators

The shift expression operators include the left shift '<<' and right shift '>>' operators. The shift operators perform bit shifting operations.

The important thing to know about the bit shifting operators is how they behave when applied to different integral types. If the value being shifted is a signed integral type, then an arithmetic shift is performed. An arithmetic shift means that the sign of the value is preserved as the bits are shifted right. If the value being shifted is an unsigned integral type, a logical shift occurs and high-order empty bit positions are set to zero. Let's take a look at the shift operators in action in Example 6.12.

6.12 ShiftOperatorTest.cs

```
1    using System;
2
3    public class ShiftOperatorTest {
4      static void Main(){
5        short i =  -0x000F;
6        short j =   0x000F;
7        Console.WriteLine("The value of i before the shift: " + i);
8        Console.WriteLine("The value of j before the shift: " + j);
9        Console.WriteLine("The value of i after the shift: " + (i >> 2));
10       Console.WriteLine("The value of j after the shift: " + (j >> 2));
11     }
12   }
```

Referring to Example 6.12 — two short variables named i and j are declared and initialized using *hexadecimal literal* values representing -15 and 15 respectively. Lines 7 and 8 print these values of i and j to the console. Lines 9 and 10 print the values of i and j after the right shift operator has been applied, shifting the bits two places to the right. What do you think the new values will be? Figure 6-13 shows the results of running this program. Cover the figure and try to work it out before proceeding. A detailed explanation follows the figure.

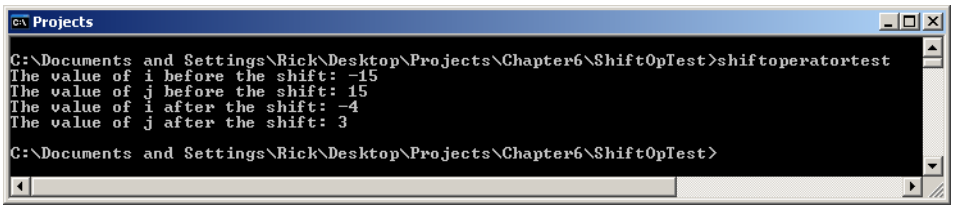


Figure 6-13: Results of Running Example 6.12

Referring to Figure 6-13 — after the shift, the value of i is -4 and the value of j is 3. Here's a brief explanation as to why they are different. The value 15 is represented in hexadecimal as the letter F. The hexadecimal value F is represented in binary as 1111. A short type is sixteen digits long, therefore the full binary for the positive number 15 is:

0000000000001111

The value of the variable i is -15. To convert the binary value 15 to -15, you need to invert the bits and add 1. This is known as 1's complement. The resulting binary value representing the number -15 looks like this:

1111111111110001

When this string of binary digits is shifted two places to the right, the new value becomes:


```
1111111111111100
```

This is the binary representation of the decimal value -4.

The value of the variable j is also shifted to the right two places, but because it's a positive value, the left-most binary digits are replaced with 0. The binary value of j after the shift looks like this:

```
0000000000000011
```

This is the binary representation for the decimal value 3.

Relational, Type-Testing, and Equality Expression Operators

This category of operators includes the comparison operators equals '==', not equals '!=', less than '<', greater than '>', less than or equal to '<='. and greater than or equal to '>='. It also includes the type testing operators '`is`' and '`as`'.

The comparison operators work on integral, floating point, decimal, and enumeration types, The '==' and '!=' operators work on boolean, reference, string, and delegate types. The behavior of these operators is summed up in Table 6-5.

Operator	Behavior	Operands
<	Returns true if left operand is less than the right operand; false otherwise	numeric types enumeration types reference types if overloaded
>	Returns true is left operand is greater than the right operand; false otherwise	numeric types enumeration types reference types if overloaded
<=	Returns true if the left operand is less than or equal to the right operand; false otherwise	numeric types enumeration types reference types if overloaded
>=	Returns true if the left operand is greater than or equal to the right operand; false otherwise	numeric types enumeration types reference types if overloaded
==	Returns true if the left operand is equal to the right operand; false otherwise	numeric types enumeration types boolean values string objects delegate types reference types if overloaded
!=	Returns true if the left operand is not equal to the right operand; false otherwise	numeric types enumeration types boolean values string objects delegate types reference types if overloaded

Table 6-5: Comparison Operator Behavior

The behavior of these operators is easy to understand in the context of numbers. However, the behavior of the '==' and '!=' operators and how they work for reference objects begs for an example. These two operators will work on string objects as expected but only because the String class provides a definition for them. In other words, string objects know how to behave when compared to each other with the '==' and '!=' operators.

User-defined classes that do not overload the '==' or '!=' operators will be compared to each other according to the rules the operators follow when comparing ordinary objects. Let's look at an example. Example 6.13 gives the code.

6.13 ReferenceEqualityTest.cs

```
1    using System;
2
3    public class ReferenceEqualityTest {
4      static void Main(){
5         Object o1 = new Object();
6         Object o2 = new Object();
7         Object o3 = o2;
8         String s1 = "Hello";
9         String s2 = "Hello";
10        String s3 = "World";
11        Console.WriteLine(o1 == o2);
12        Console.WriteLine(o1 != o2);
13        Console.WriteLine(o2 == o3);
14        Console.WriteLine(s1 == s2);
15        Console.WriteLine(s1 == s3);
16     }
17   }
```

Referring to Example 6.13 — three Object reference variables named o1 through o3 are declared and initialized on lines 5 through 7. The variables o1 and o2 point to unique objects. The variable o3 is assigned the same address as the variable o2. This means that the variables o2 and o3 now point to the same object.

On lines 8 through 10, three String variables are created. The variables s1 and s2 each point to identical string values "Hello". The variable s3 points to a string whose value is "World". Now study the results of running this program as shown in Figure 6-14.

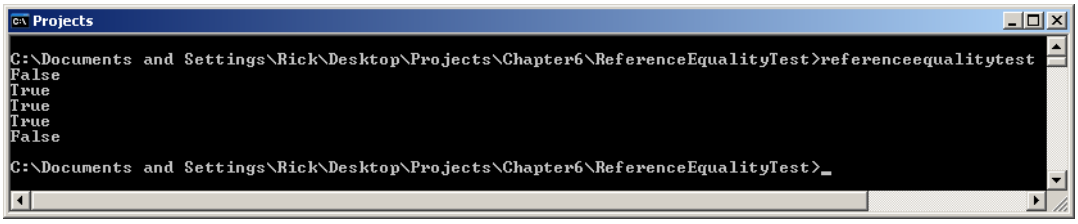


Figure 6-14: Results of Running Example 6.13

Referring to Figure 6-14 — on line 11, the expression o1 == o3 evaluates to false because the variables o1 and o2 point to different objects. The "==" operator's natural behavior as defined in the Object class is to test if we are comparing the same object. If not, the operator returns false. On the next line, the expression o1 != o2 returns true as expected. On line 3, the expression o3 == o2 returns true because the variables o2 and o3 do in fact point to the same object and so they must be equal.

These same operators behave somewhat differently when used with String objects. Notice now on line 14 that if two different Strings are compared the result will be true if their values (*i.e.*, the characters they contain) are identical. Different String objects with different values will return false, as is shown on the last line.

Logical AND, OR, and XOR Expression Operators

The logical AND '&', OR '|', and XOR '^' operators behave differently according to their parameter types. Table 6-6 summarizes the logical operator behavior.

Operator	Behavior	Operands
&	Integral operands: x & y performs bitwise logical AND Enumeration operands: x & y performs bitwise logical AND Boolean operands: Performs conditional AND comparison	int, uint, long, ulong enumeration boolean
|	Integral operands: x | y performs bitwise logical OR Enumeration operands: x | y performs bitwise logical OR Boolean operands: Performs conditional OR comparison	int, uint, long, ulong enumeration boolean

Table 6-6: Logical Operator Behavior

Operator	Behavior	Operands
^	Integral operands: x ^ y performs bitwise logical XOR Enumeration operands: x ^ y performs bitwise logical XOR Boolean operands: Performs conditional XOR comparison	int, uint, long, ulong enumeration boolean

Table 6-6: Logical Operator Behavior

Logical Operations on Integral Operands

When presented with integral operands, the logical operators perform bitwise logical operations on their operands according to the truth tables shown in Figure 6-15. Example 6.14 shows these operators in action.

6.14 LogicalOperatorTest.cs

```
1    using System;
2
3    public class LogicalOperatorTest {
4      static void Main(){
5        int i = 0xFFFF;
6        int mask_1 = 0x0000;
7        int mask_2 = 0x0003;
8        int mask_3 = 0xFFFF;
9        Console.WriteLine("FFFF & 0000 = " + (i & mask_1));
10       Console.WriteLine("FFFF | 0000 = " + (i | mask_1));
11       Console.WriteLine("FFFF & 0003 = " + (i & mask_2));
12       Console.WriteLine("FFFF | 0003 = " + (i | mask_2));
13       Console.WriteLine("FFFF ^ FFFF = " + (i ^ mask_3));
14     }
15   }
```

X	Y	X & Y
0	0	0
1	0	0
0	1	0
1	1	1

X	Y	X | Y
0	0	0
1	0	1
0	1	1
1	1	1

X	Y	X ^ Y
0	0	0
1	0	1
0	1	1
1	1	0

Figure 6-15: Logical AND, OR, and XOR Truth Tables

Referring to Example 6.14 — on line 5, an integer variable i is declared and initialized to the hexadecimal value FFFF. On lines 6 through 8, three more integer variables named mask_1 through mask_3 are declared and initialized with the hexadecimal values 0000, 0003, and FFFF, respectively. Lines 9 through 13 use the logical operators to perform bit manipulation operations on the variable i using the various mask values. Figure 6-16 shows the results of running this program.

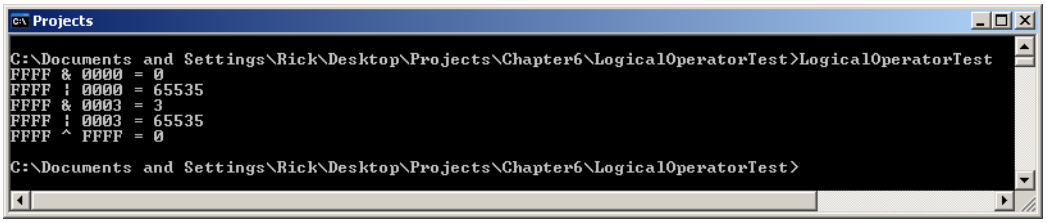


Figure 6-16: Results of Running Example 6.14

Logical Operations on Enumeration Operands

The logical operators work with enumeration type operands. An enumeration is a type that represents any one of a set of authorized values. Enumeration types are declared with the `enum` keyword and can be defined outside of or within the body of a class, as the code in Example 6.15 illustrates.

6.15 LogicalOperationEnumTest.cs

```
1    using System;
2
3    public class LogicalOperatorEnumTest {
4
5      public enum EYE_COLOR {BLACK, BROWN, HAZEL, BLUE, GREY};
6
7      static void Main(){
8        Console.WriteLine(EYE_COLOR.BLACK & EYE_COLOR.BROWN);
9        Console.WriteLine(EYE_COLOR.BROWN & EYE_COLOR.BROWN);
10       Console.WriteLine(EYE_COLOR.BLACK & EYE_COLOR.BLUE);
11       Console.WriteLine(EYE_COLOR.BLACK | EYE_COLOR.BROWN);
12       Console.WriteLine(EYE_COLOR.BROWN | EYE_COLOR.HAZEL);
13       Console.WriteLine(EYE_COLOR.BLACK | EYE_COLOR.BLUE);
14       Console.WriteLine(EYE_COLOR.BLACK ^ EYE_COLOR.BROWN);
15       Console.WriteLine(EYE_COLOR.BROWN ^ EYE_COLOR.BROWN);
16       Console.WriteLine(EYE_COLOR.BLACK ^ EYE_COLOR.BLUE);
17     }
18   }
```

Referring to Example 6.15 — an enumerated type named EYE_COLOR is declared on line 5, and within the curly braces there appear five names: BLACK, BROWN, HAZEL, BLUE, and GREY. The enumeration value BLACK equates to the value 0, which is the default value for the first enumeration value unless explicitly set to be something else. The next enumeration value BROWN is assigned the value 1, and so on. (Enumerated types are covered in more detail in Chapter 9.)

Essentially, the logical operators treat enumeration types like they treat integers, which they ultimately are. Figure 6-17 shows the results of running Example 6.15.

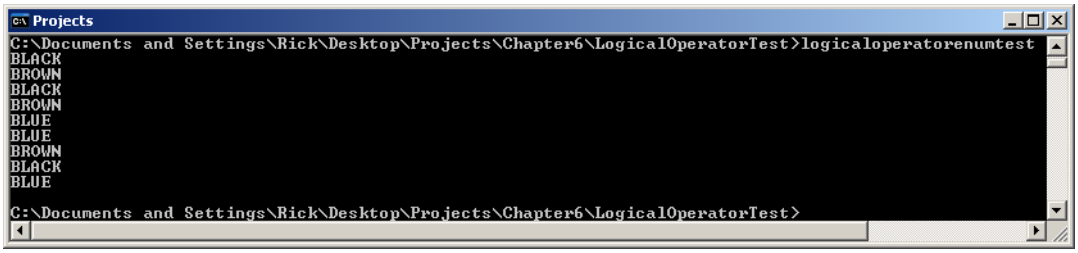


Figure 6-17: Results of Running Example 6.15

Logical Operations on Boolean Operands

The logical operators also operate on operands of type boolean. A boolean argument can be a boolean literal, a boolean variable, or a conditional expression that evaluates to a boolean value. Example 6.16 demonstrates the use of the logical operators on boolean literals. Just keep in mind that where the keywords “true” or “false” appear in the program a complex expression that evaluates to “true” or “false” could be substituted. Figure 6-18 gives the results of running this program. Compare its output with the truth tables given in Figure 6-15.

6.16 LogicalBoolTest.cs

```
1    using System;
2
3    public class LogicalBoolTest {
4       static void Main(){
5         Console.WriteLine("true  & true  = " + (true  & true));
6         Console.WriteLine("true  & false = " + (true  & false));
7         Console.WriteLine("false & true  = " + (false & true));
8         Console.WriteLine("false & false = " + (false & false));
9         Console.WriteLine("------------");
10        Console.WriteLine("true  | true  = " + (true  | true));
11        Console.WriteLine("true  | false = " + (true  | false));
12        Console.WriteLine("false | true  = " + (false | true));
13        Console.WriteLine("false | false = " + (false | false));
14        Console.WriteLine("------------");
15        Console.WriteLine("true  ^ true  = " + (true  ^ true));
16        Console.WriteLine("true  ^ false = " + (true  ^ false));
17        Console.WriteLine("false ^ true  = " + (false ^ true));
18        Console.WriteLine("false ^ false = " + (false ^ false));
19      }
20   }
```

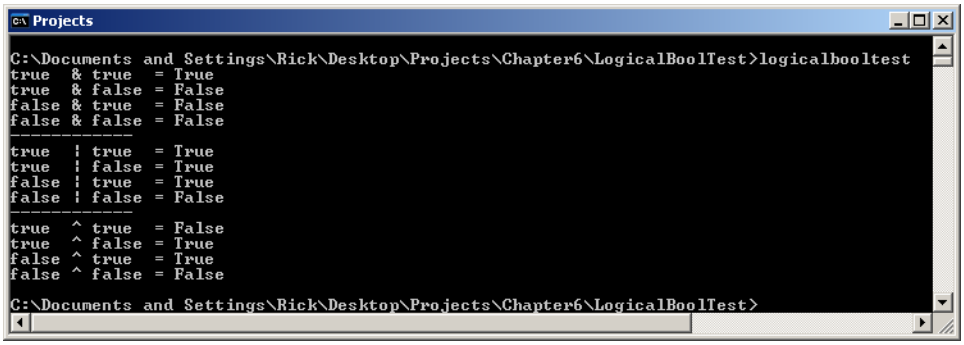



Figure 6-18: Results of Running Example 6.16

Conditional AND and OR Expression Operators

The conditional operators AND '`&&`' and OR '`||`' are also referred to as the *short-circuiting* logical operators. The reason for this alternate name is that they will skip the evaluation of the second operand if the expression can be completely evaluated solely on the value of the first operand. For example, the second operand in the expression (`true || true`) can be safely skipped because the OR operator requires only one true operand. However, the second operand in the expression (`false || true`) must be evaluated since the first operand was false. These operators are demonstrated in Example 6.17. Figure 6-19 gives the results of running this program.

6.17 ConditionalOpsTest.cs

```
1    using System;
2
3    public class ConditionalOpsTest {
4       static void Main(){
5         Console.WriteLine("true  && true  = " + (true  && true));
6         Console.WriteLine("true  && false = " + (true  && false));
7         Console.WriteLine("false && true  = " + (false && true));
8         Console.WriteLine("false && false = " + (false  && false));
9         Console.WriteLine("--------------------");
10        Console.WriteLine("true  || true  = " + (true  || true));
11        Console.WriteLine("true  || false = " + (true  || false));
12        Console.WriteLine("false || true  = " + (false || true));
13        Console.WriteLine("false || false = " + (false || false));
14      }
15   }
```

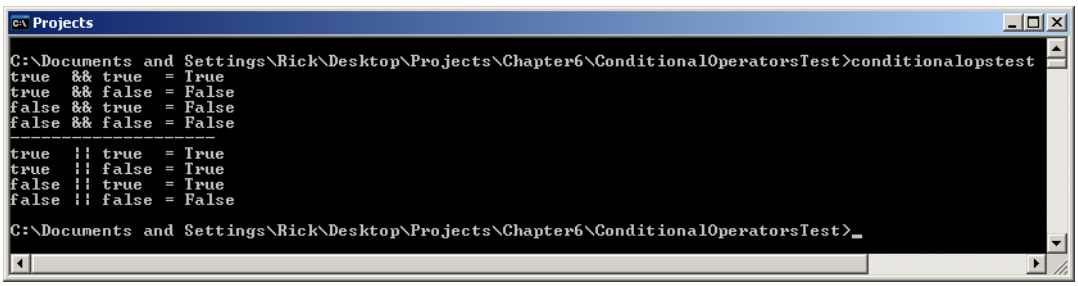


Figure 6-19: Results of Running Example 6.17

Conditional (Ternary) Expression Operator

The conditional operator '`?:`', also referred to as the *ternary* operator, takes one boolean operand expression. Based on the results of its evaluation, it returns one of two possible expressions. For example, consider the following ternary operator statement:

```
(boolean conditional expression) ? expression A : expression B;
```

If the conditional expression evaluates to true, then expression A is evaluated and returned as a result of the operation. If the conditional expression evaluates to false, then expression B is evaluated and returned instead. The ternary operator never evaluates both alternate expressions. Example 6.18 shows the ternary operator in use.

6.18 TernaryOperatorTest.cs

```
1    using System;
2
3    public class TernaryOperatorTest {
4      static void Main(){
5        Console.WriteLine(true  ? "Return this string if true" : "Return this string if false");
6        Console.WriteLine(false ? "Return this string if true" : "Return this string if false");
7        Console.WriteLine();
8        int i = 3;
9        int j = 7;
10       Console.WriteLine((i < j)  ? "Return this string if true" : "Return this string if false");
11       Console.WriteLine((i > j)  ? "Return this string if true" : "Return this string if false");
12     }
13   }
```

Referring to Example 6.18 — lines 5 and 6 utilize the boolean literals “true” and “false” as arguments to the ternary operator’s conditional expression. Since true is always true, the first string will be returned. On line 6, the second string will always be returned. On lines 8 and 9, two integer variables i and j are declared and initialized to the values 3 and 7, respectively. On lines 10 and 11, these variables are used to demonstrate how an actual conditional expression might be constructed.

The use of the boolean literals on lines 5 and 6 triggers a compiler warning that says it has detected unreachable code, as Figure 6-20 shows. Figure 6-21 shows the results of running the program.

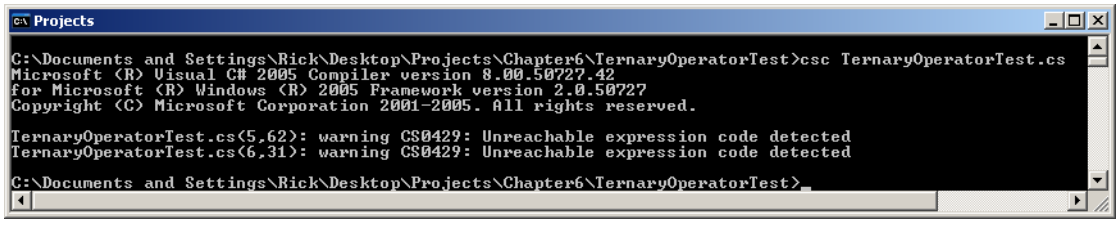


Figure 6-20: Compiler Warning due to Unreachable Code

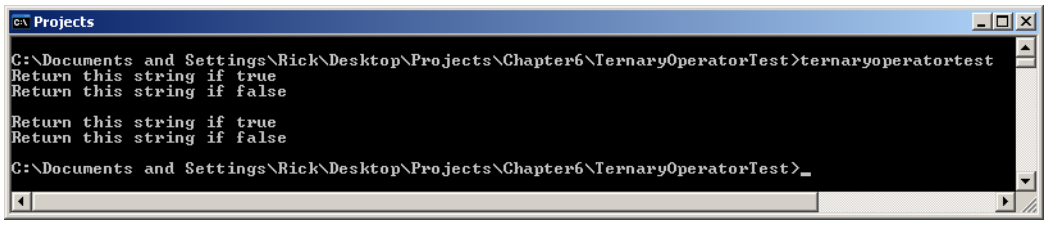


Figure 6-21: Results of Running Example 6.18

Assignment Expression Operators

The assignment expression operators include both the simple assignment operator `'='`, which you have seen used throughout this chapter, and the compound assignment operators `+=`. `-=`. `*=`. `/=`, `%=`. `<<=`, `>>=`, `&=`, `|=`, and `^=`. The ‘+=’ operator is also overloaded to include event assignment, which I will cover in depth in Chapter 12.

The compound operators, as their name suggests, combine the indicated operation with an assignment. This makes for a convenient shorthand way of doing things. For example, the expression `i = i + 1` can be written `i += 1` with the help of the compound operator. Example 6.19 demonstrates the use of several compound operators. Figure 6-22 shows the results of running this program.

6.19 AssignmentOpsTest.cs

```
1    using System;
2
3    public class AssignmentOpsTest {
4      static void Main(){
5        int i = 0;
6        Console.WriteLine("The value of i initially = " + i);
7        Console.WriteLine("i += 1 = " + (i += 1));
8        Console.WriteLine("i -= 1 = " + (i -= 1));
```



```
9         Console.WriteLine("i += 2 = " + (i += 2));
10        Console.WriteLine("i *= 2 = " + (i *= 2));
11        Console.WriteLine("i /= 2 = " + (i /= 2));
12      }
13    }
```

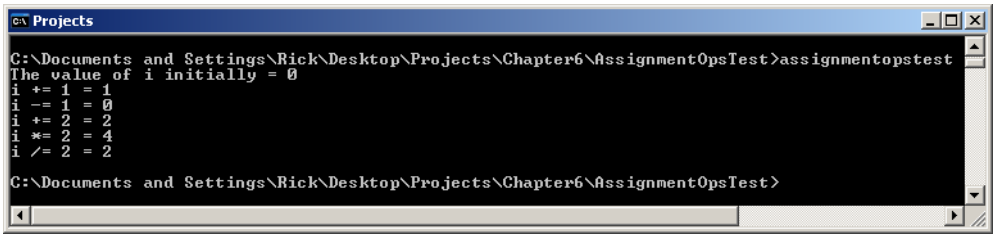


Figure 6-22: Results of Running Example 6.19

Quick Review

Statements are the fundamental building blocks of C# programs. A statement can be thought of as the smallest standalone element of a program. Programs are built using sequences of statements. C# offers nineteen different types of statements.

The term *operator precedence* refers to the order in which the C# compiler evaluates the operators appearing in an expression statement. The term *associativity* refers to the direction in which the C# compiler performs a series of operations.

Summary

A simple C# application is a class that contains a Main() method. The purpose of the Main() method is to provide an entry point for program execution. There are four authorized versions of the Main() method; each version has a different method signature.

A class that contains a Main() method can be compiled into an executable assembly. A class with no Main() method can be compiled into a module. Modules can be added to assemblies. Modules created in CLI-compliant languages other than C# can be compiled with C# modules to form executable assemblies.

Identifiers are sequences of characters that represent names of objects in a program. Identifiers are used to name classes, structures, methods, variables, constants, properties, fields, enums, etc. Identifiers can start with either an uppercase or lowercase letter or an underscore '_' character followed by any number of letters, digits, and underscores.

Reserved keywords are identifiers that have special meaning within the C# language. You cannot reintroduce a reserved keyword as a name for an object within your program. You can, however, prepend the '@' symbol to a reserved keyword to formulate a valid identifier, however, I discourage doing this as it renders code hard to read, understand, and maintain.

C# has two kinds of types: value types and reference types. Value type variables contain the actual data as defined by the type. Reference type variables contain a reference to an object in memory. Two or more reference type variables could reference the same object in memory. The C# predefined types map to structures within the System namespace. System.object is the base type for all other types.

Skill-Building Exercises

1. **API Drill:** Visit the Microsoft Developer Network (MSDN) [www.msdn.com] and research the C# predefined type structures located in the System namespace. Use Table 6-2 as a guide. Take note of the methods and fields each

structure makes available for use. Track down and study any interfaces these structures may implement.

2. **Practice Makes Perfect:** Compile and run each of the example programs listed in this chapter.

3. **Type Ranges:** Write a program that displays to the console a list of the predefined numeric types and shows their minimum and maximum values. **Hint:** Pay attention to what you discovered in Skill-Building Exercise #1!

Suggested Projects

1. **Average Five Numbers:** Write a program that computes the average of five floating point numbers and writes the answer to the console.

2. **Compute The Area:** Write a program that computes the area of a rectangle or square given the input height and width.

3. **Find The Greatest Value:** Write a program that compares the values of two integer variables and returns the larger of the two. Use the ternary conditional operator to perform the comparison.

4. **Compute Time To Travel:** Write a program that computes the *time* required to travel a given *distance* in miles at a certain *speed* in miles/hour. The equation required is:

$$t = d / s$$

5. **Compute Average Speed:** Write a program that computes the *speed* required to travel a certain *distance* in a given amount of *time*. The equation required is:

$$s = d / t$$

6. **Compute Fuel Efficiency:** Write a program that takes miles traveled since last fill-up and gallons of gas required to fill your car's tank. Calculate how many miles/gallon your car is getting between fill ups. Write the results to the console.

7. **Division By Shifting:** Write a program that divides an integer by 2 using the right shift operator. Explain why shifting a number to the right performs a division. What happens when you shift a number to the left? **Hint:** Think in terms of binary digits.

Self-Test Questions

1. What two kinds of types does C# support?

2. How many predefined types does C# support.

3. Describe in your own words how two reference type variables might end up referencing the same object.

4. What's the difference between a value type and a reference type?

5. What character is used to terminate a statement?

6. What's the purpose of the *new* operator.

7. How many different forms of the Main() method does C# support?

8. What's the purpose of a Main() method?

9. What's the purpose of the *using* directive.

10. Can a reserved keyword be used as an identifier? Explain your answer.

References

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-334.htm]

Microsoft Developer Network (MSDN) [http://www.msdn.com]

Notes