# *Chapter 3*
# *Introduction to Java and C#*

There has been significant discussion in the industry regarding the similarities between Java and C#. Some have suggested that C# is simply a clone of the Java language. Others claim that C# gleaned ideas from many languages including Java, C, C++, Modula-2, and Smalltalk. Though C# appears to have borrowed more features from Java than from its other ancestors, the C# designers attempted to remain true to the language's C++ heritage as often as possible. For instance, most of the C# language's operators, keywords, and statements are borrowed directly from C++. So, is C# just a Java clone or is it a radically new language featuring unique and innovative ideas? As is usually the case, the truth lies somewhere in the middle. Though C# unabashedly borrows many of Java's design features, it also introduces many new and innovative ideas.

As we will see in this chapter, C# adopted many of Java's most popular design features. However, these features do not always work the same in C# as they do in Java. Rather, in some areas, the C# designers attempted to "improve upon" Java by modifying certain features or adding new ones. Of course, it is up to each developer to decide whether or not these changes actually represent improvements. There are strong arguments for and against virtually all divergent design decisions made by the Java and C# development teams.

In this chapter, we will learn the fundamentals of the Java and C# languages. As we do so, we will compare and contrast the various data types, keywords, statements, and programming constructs employed by each language. Sample code will be provided to demonstrate each new concept. Since many readers are familiar with at least one of these two languages, the differences between the two languages will be highlighted throughout the chapter. This allows readers that are proficient in one language to leverage their knowledge and focus only on the areas in which Java and C# differ. Any identical features shared by these languages will not be addressed or only demonstrated briefly for the purpose of completeness.

Though introducing two languages in a single chapter is a somewhat daunting task; that is exactly what we are going to attempt here. Feel free to skim or skip any sections with which you are already familiar. This chapter consists of the following two sections:

- Language Introduction
- Language Fundamentals

## *Language Introduction*

As we will see, the syntax for Java and C# are very similar in many areas. This similarity can be traced to the fact that both languages are derived from C++. In addition, the C# designers were able to learn from Java and adopt many of its most popular features. The similarities and differences between Java and C# will become apparent in this section as we examine a basic program written in both languages. Specifically, this section covers the following topics.

- Hello, World!

- Program File Naming Conventions
- Compilation and Execution
- Main Method
- Namespaces
- Referencing Classes

## Hello World!

What better way to introduce a language than with the canonical "Hello World!" program? Since examining code is usually the quickest way to learn, let's jump right in. Code listings 3.1 and 3.2 present basic "Hello World!" programs in Java and C#, respectively.

**Listing 3.1** ***"Hello World!" Program (Java)***

```
1: public class HelloWorld
2: {
3:   public static void main(String[] args)
4:   {
5:     System.out.println("Hello World!");
6:   }
7: }
```

**Listing 3.2** ***"Hello World!" Program (C#)***

```
1: public class HelloWorld
2: {
3:   public static void Main(string[] args)
4:   {
5:     System.Console.WriteLine("Hello World!");
6:   }
7: }
```

After examining both programs, it is immediately apparent that Java and C# are close relatives in the family of programming languages. In fact, you have to look closely to find any differences at all. However, these two simple examples are a bit misleading. Less trivial programs will reveal more conspicuous differences between these languages. Let's begin our comparison by evaluating each line of these two programs.

Each program begins with a class declaration using the `class` keyword. As pure object-oriented languages, Java and C# require all code to reside within a class or similar structure. C-style global functions are not supported. In both examples, the classes are declared public. The `public` modifier indicates that the `HelloWorld` class is globally accessible by all classes.

Line 3 of `HelloWorld` differs in case only. In Java, the main method begins with a lower-case letter (e.g., `main`) while C# capitalizes this method (e.g., `Main`). The reverse holds true for the string variable. Java capitalizes the `String` data type while C# uses a lower-case `string`. Capitalization differences will become a recurring theme as we dissect each language. You will soon realize that the case conventions for Java and C# differ quite often. You should also notice that each example declares main as a `public static` method having a `void` return type. The `public` modifier indicates that this method can be called by any class. Though the main method is typically declared `public`, it is not required for compilation or execution. The `static` modifier means that `main` and `Main` are "class methods" that can be called using just the class name as opposed to an instance of the class (the `static` modifier is discussed later in this chapter). Finally, `void` indicates that the method does not return a value. The `args` parameter will be discussed in the next section.

Line 5 is where all the work gets done. The primary difference here is the name of the class and methods used to output text to the console. Though it may appear straight forward, this line can be a bit confusing. The confusion generally stems from the fact that `System` is the name of a class in Java while in C# it represents a namespace. This disparity introduces some subtle differences that are not immediately apparent upon first inspection. For instance, in the Java example, `System` is the name of a class that contains a static object variable named `out` that implements a static method called `println()`. The equivalent line in the C# version is a bit less cryptic. In C#, `System` is a namespace that contains a class called `Console` that implements a static method called `WriteLine()`. As you probably guessed, any text passed to `println()` or `WriteLine()` is sent to standard out which, by default, is directed to the system console.

**Program File Naming Conventions**

Java requires that each program file contains no more than one public class (or interface), is given the same name as the public class it defines, and ends with the `.java` extension. For instance, Listing 3.1 must be stored in a file named `HelloWorld.java` since the `HelloWorld` class is declared `public`. If the file does not contain a public class, Java does not mandate the file name other than requiring the `.java` extension. Typically, each Java program file contains only a single class definition. However, multiple classes may be defined within a single file as long as no more than one class is declared `public`. And, of course, if one class is declared `public`, the file must be named after that class.

C#, on the other hand, places no restrictions on program file names. C# program files may use any name and extension. For instance, Listing 3.2 could be stored in a file named `MyFirstCSharpProgram.txt`. Typically, however, C# program files are named according to the class or classes they define and end with a `.cs` extension. For example, Listing 3.2 would typically be named `HelloWorld.cs` (though this name is not mandatory). Additionally, a single C# program file can contain any number of class definitions, including multiple public classes.

**Compilation and Execution**

Compiling and executing a Java application requires two programs that are included in the Java SDK: `javac` and `java`. To compile a `.java` file, use the `javac` compiler like this:

```
javac HelloWorld.java
```

The `javac` compiler produces a `.class` file having the same name as each class defined within the `.java` file. For instance, the previous `javac` compilation command generates a file called `HelloWorld.class`. The compiled `HelloWorld.class` file can then be executed using the `java` interpreter program like this:

```
java HelloWorld
```

Note that the `.class` extension is assumed by the Java interpreter and should not be specified. Otherwise, the interpreter will not be able to locate the class.

Compiling a C# application requires the `csc` compiler that ships with the .NET Framework SDK. To compile a C# file, use the `csc` compiler like this:

```
csc HelloWorld.cs
```

Unlike Java, the C# compiler produces an executable `.exe` file rather than a `.class` file that must be executed by an interpreter. Therefore, C# compiled files can be executed directly from the command-line without specifying an interpreter.

> **NOTE**
> Though it may appear that C# applications run as native executables, this is actually not the case. Rather, once the .NET Framework is installed, the Windows shell recognizes .NET executables and automatically invokes the CLR to run them at the time they are executed.

This has been a very brief introduction to the Java and C# compilers and the Java interpreter. For detailed information regarding these programs, including their many command-line options, see Chapter 6.

## Main Method

Similar to C and C++, the main method is the program's entry point. This is where the runtime environment begins execution when the program is invoked. Though not required for all classes, any class that is to be executed from the command-line must contain a main method.

The main method signatures supported by Java and C# differ slightly. In Java, there is only one valid parameter list and return type for the main method. The Java main method is declared like this:

```
public static void main(String[] args)
```

In contrast, C# supports two parameter signatures and two return types for the main method. The combination of different parameters and return types create four valid declarations as follows:

```
public static void Main(string[] args)
public static void Main()
public static int Main(string[] args)
public static int Main()
```

### *Command-Line Arguments*

In both Java and C#, the `args` parameter is a string array containing all of the command-line arguments passed when the program was invoked. For example, Listing 3.3 demonstrates how the command-line arguments can be retrieved by a simple Java program. A C# example would be nearly identical with a notable exception being that the `args` variable's `length` property would begin with a capital letter.

**Listing 3.3** ***Accessing Command-Line Arguments (Java)***

```
public class Hello
{
  public static void main(String[] args)
  {
    String welcome = "Hello";

    if (args.length > 0)
      welcome += " " + args[0];

    if (args.length > 1)
      welcome += " and " + args[1];
    System.out.println(welcome);
  }
}
```

This simple program first checks for the existence of command-line arguments and then concatenates them to the end of the welcome message. Here's the output from Listing 3.3 using the following command-line parameters:

```
C:\> java Hello Java C#
Hello Java and C#
```

Typically, if a program does not require access to command-line arguments, it simply ignores the `args` parameter. However, in C#, if command-line arguments are not used, a main method declaration is provided that does not accept any input parameters. This declaration saves a little typing and may improve readability since no unused variables are introduced.

***Return Values***

You may have noticed that, unlike Java, C# supports a main method that returns an `int` value. This value can be useful to programs that invoke the class (e.g., batch files or shell scripts). For example, Listing 3.4 demonstrates a C# program that returns a different value based on the number of command-line arguments received and Listing 3.5 presents an MS-DOS batch file that alters its flow based on the value returned.

**Listing 3.4** ***Returning a Value from the Main Method (C#)***

```
public class ReturnValue
{
  public static int Main(string[] args)
  {
    if (args.Length == 0)
      return 0;
    else if (args.Length == 1)
      return 1;
    else
      return 2;
  }
}
```

**Listing 3.5** ***Reading Return Values within a Batch File***

```
@ECHO OFF

REM Pass batch file's command-line arguments to C# program (%1, %2, %3).
ReturnValue %1 %2 %3

REM Always check error levels in reverse order (highest to lowest).
IF ERRORLEVEL 2 GOTO TWO_ARG
IF ERRORLEVEL 1 GOTO ONE_ARG
IF ERRORLEVEL 0 GOTO NO_ARG

:NO_ARG
ECHO No Arguments
GOTO END

:ONE_ARG
ECHO One Argument
GOTO END
```

```
:TWO_ARG
ECHO Two or More Arguments

:END
```

The lack of a return value in Java's main method may seem like a critical oversight. However, Java provides this same functionality through a simple static method call. The `System.exit()` method exits the main method (terminating program execution) and returns an integer value to the process from which it was invoked. Listing 3.6 demonstrates the the `System.exit()` method. To test Listing 3.6, simply alter the fourth line of Listing 3.5 to read "`java ReturnValue %1 %2 %3`".

**Listing 3.6** ***Returning a Value from the Main Method (Java)***

```
public class ReturnValue
{
  public static void main(String[] args)
  {
    if (args.length == 0)
      System.exit(0);
    else if (args.length == 1)
      System.exit(1);
    else
      System.exit(2);
  }
}
```

***Multiple Main Methods (C#)***

Unlike Java, C# supports multiple public classes within a single source file and all of these classes may implement a `Main()` method. Of course, without additional information, the compiler would not know which `Main()` method is the desired entry point for the compiled assembly and, thus, would generate a compilation error. For instance, compiling Listing 3.7 without providing additional information produces the following error:

```
C:\> csc MultipleMain.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

MultipleMain.cs(3,22): error CS0017: Program 'MultipleMain.exe' has more
  than one entry point defined: 'Main1.Main(string[])'
MultipleMain.cs(11,22): error CS0017: Program 'MultipleMain.exe' has
  more than one entry point defined: 'Main2.Main(string[])'
```

**Listing 3.7** ***C# Program with Multiple Main Methods (C#)***

```
public class Main1
{
  public static void Main(string[] args)
  {
    System.Console.WriteLine("Main 1");
  }
}

public class Main2
{
  public static void Main(string[] args)
```

```
    {
      System.Console.WriteLine("Main 2");
    }
}
```

To compile a C# source file containing multiple `Main()` methods, you must tell the compiler which class contains the desired `Main()` method by using the `/main` command-line argument like this:

```
C:\> csc /main:Main1 MultipleMain.cs
```

**Namespaces**

The purpose of this section is to provide a brief introduction to how packages and namespaces are declared in Java and C# programs. See Chapter 2 for a more comprehensive discussion on packages and namespaces.

In order to organize classes and prevent naming collisions, all classes within the Java and .NET class libraries reside within a unique namespace. Like C++, .NET uses the generic term *namespace* when referring to the construct that organizes classes hierarchically. On the other hand, Java uses the term *package* to describe this concept. Listings 3.8 and 3.9 illustrate how packages and namespaces are declared in Java and C# programs, respectively. The Java example declares the `HelloWorld` class to be within the `com.sourcestream.samples` package and the C# example declares the `HelloWorld` class to be within the `SourceStream.Samples` namespace. Also notice that, unlike Java's packages, Listing 3.10 demonstrates how C# namespaces can be nested.

**Listing 3.8** ***"Hello World!" within the com.sourcestream.samples Package (Java)***

```
package com.sourcestream.samples;

public class HelloWorld
{
  public static void main(String[] args)
  {
    System.out.println("Hello World!");
  }
}
```

**Listing 3.9** ***"Hello World!" within the SourceStream.Samples Namespace (C#)***

```
namespace SourceStream.Samples
{
  public class HelloWorld
  {
    public static void Main(string[] args)
    {
      System.Console.WriteLine("Hello World!");
    }
  }
}
```

**Listing 3.10** ***"Hello World!" within the SourceStream.Samples Nested Namespace (C#)***

```
namespace SourceStream
{
  namespace Samples
  {
    public class HelloWorld
    {
      public static void Main(string[] args)
      {
        System.Console.WriteLine("Hello World!");
      }
    }
  }
}
```

**Referencing Classes**

Java uses the `import` statement to declare the packages in which the compiler should search when attempting to locate classes referenced in code. The `using` statement performs this same function in C#. The `import` and `using` statements allow classes to be referenced without fully qualifying them. For example, consider the slight alteration made to the C# "Hello World!" example shown in Listing 3.11.

**Listing 3.11** ***"Hello World!" that uses the using Statement (C#)***

```
using System;

public class HelloWorld
{
  public static void Main(string[] args)
  {
    Console.WriteLine("Hello World!");
  }
}
```

Notice that once the "`using System;`" line is added, the `Console` class can be referenced without specifying its namespace. When the compiler encounters the reference to `Console`, it attempts to locate the class within each namespace declared by a `using` statement. In this case, given that only one `using` statement has been declared, the compiler searches the `System` namespace when attempting to locate the `Console` class. If `Console` did not exist within the `System` namespace, the compiler would produce an error indicating that it was unable to locate the class. Each declared namespace requires a separate `using` statement and all `using` statements must precede all other programming elements (with the exception of comments).

Now let's turn our attention to the Java version of the "Hello World!" program. As we discussed previously, in Java, `System` is the name of a class rather than the name of a package or namespace. So, you might be wondering, to which Java package does the `System` class belong? The answer is that `System` is a class within the `java.lang` package. Given this fact, you might expect that references to `System` would need to be fully qualified using `java.lang.System`. That's a reasonable assumption but it's not necessary because the `java.lang` package is automatically imported by the Java compiler. This is due to the fact that `java.lang` is the most fundamental package in Java and virtually no application can be written without accessing some of the classes within this package. Since it is used by every application, Java relieves the programmer from having to always import this package. Regardless, it is perfectly legal to explicitly import the `java.lang` package for documentation and code readability reasons. Listing 3.12 demonstrates how the java `import` statement is used. Notice that, unlike C#, Java uses the asterisk (`*`) wildcard to indicate that an entire package is being imported as opposed to a single class.

**Listing 3.12** ***"Hello World!" that uses the import Statement (Java)***

```
import java.lang.*;

public class HelloWorld
{
  public static void main(String[] args)
  {
    System.out.println("Hello World!");
  }
}
```

Though the `import` and `using` statements indicate the packages and namespaces in which referenced classes reside, this does not include sub-packages and sub-namespaces. For instance, importing the `java.util` package will not make the `java.util.zip` package available. These packages must be declared using separate `import` statements. Likewise, just because a class uses the `System` namespace, the compiler will not automatically search the `System.Data` namespace for a referenced class. Again, these namespaces must be declared separately as shown below:

```
import java.util.*;
import java.util.zip.*;
```

```
using System;
using System.Data;
```

### *Declaring Individual Classes (Java)*

In Java, it is possible to import an entire package or just a single class. Similar functionality in C# requires the use of an alias (discussed in the next section). For example, the following Java `import` statements import one package (`java.io`) and two classes (`java.util.ArrayList` and `java.text.DateFormat`). In this way, the `ArrayList` and `DateFormat` classes as well as any classes in the `java.io` package may be referenced without fully qualifying them:

```
import java.io.*;
import java.util.ArrayList;
import java.text.DateFormat;
```

Note that the C# `using` statement, without employing an alias, supports only namespaces. A compilation error is generated if a specific class is declared with the `using` statement.

### *Aliases (C#)*

C# introduced a very useful feature known as an alias. *Aliases* allow classes and namespaces to be renamed within a C# program. This feature is helpful whenever you wish to shorten a class name, make a class name more descriptive, or avoid naming conflicts between namespaces that include classes of the same name. For example, imagine that the `SourceStream.File` and `SourceStream.Phone` namespaces both contain a class called `Directory`. In this case, it would not be possible to access the `Directory` class by simply declaring both packages with the `using` statement. This is due to the fact that any reference to `Directory` would be ambiguous since the class exists in both namespaces. At this point, you have two choices. You can either always fully qualify the class name (using `SourceStream.File.Directory` or

`SourceStream.Phone.Directory`) or, even better, you can use C#'s alias feature to provide a unique name for each class as follows:

```
using SourceStream.File;
using SourceStream.Phone;
using FileDirectory = SourceStream.File.Directory;
using PhoneDirectory = SourceStream.Phone.Directory;
```

The neat thing about aliases is that they work for namespaces as well as classes. Therefore, they can be used whenever you want to save some typing. For example, the following code provides a short alias for the lengthy namespace `SourceStream.File.Util.Zip`:

```
using Zip = SourceStream.File.Util.Zip;
```

Classes within the `SourceStream.File.Util.Zip` namespace could then be accessed the traditional way or in an abbreviated manner as follows:

```
SourceStream.File.Util.Zip.File myZip1 = new
  SourceStream.File.Util.Zip.File(); //long reference without alias

Zip.File myZip2 = new Zip.File(); //short reference with alias
```

In the previous section, we mentioned that C#'s `using` statement does not support individual classes. However, using an alias, it is possible to declare a single class. The following code demonstrates how Java and C#, respectively, are able to reference an individual class without declaring an entire package or namespace:

```
import java.util.ArrayList; //Java import
```

```
using ArrayList = System.Collections.ArrayList; //C# alias
```

The use of aliases and single class imports are recommended for readability reasons. These constructs make it very clear exactly which class is being referenced. When entire packages and namespaces are declared, it is often difficult to determine the correct package or namespace to which a particular class belongs.

## *Language Fundamentals*

There are certain common elements within a programming language that can be considered fundamental. These elements usually include data types, variable scope, flow control statements, expression syntax, and equality checking; to name just a few. The purpose of this section is to familiarize you with the basic elements of the Java and C# languages. Specifically, this section covers the following topics.

- Basic Data Types
- Complex Data Types
- Variable Scope
- Variable Modifiers
- Type Conversions
- Constants

- Operators
- Branch Statements
- Iteration Statements
- Jump Statements
- Equality Checking
- Preprocessor Directives
- Exception Handling
- Comments
- Assertions

## Basic Data Types

Java and C# both define basic data types. These basic types are known as *primitives* in Java and *simple types* in .NET. All basic data types have a few traits in common. First, basic data types are atomic. Unlike classes and structs, basic types are not an aggregation of simpler types and cannot be decomposed. Second, by default, all basic data types are passed by value rather than reference. Whenever a basic data type is passed from one method to another, each method receives its own copy of the variable. Third, all basic data types are stack allocated while most user-defined data types are heap allocated. And finally, basic data types are initialized using the equals (=) operator rather than the `new` operator as demonstrated here (identical in Java and C#):

```
int i = 123;
long l = 1234568790;
float f = 123.45F; //float literals must use the "F" suffix
double d = 1234567890.1234;
```

In addition, it is possible to initialize multiple variables of the same type on a single line like this:

```
int i = 123, j = 234, k = 345;
```

Basic data types that are declared but not initialized are assigned a default value of zero (or, in the case of a boolean type, false).

> **NOTE**
> The *heap* is a managed pool of memory in which long-term objects are stored. The heap can be dynamically sized at runtime. Objects stored in the heap exist independently of any method. In contrast, the *stack* is a reserved area of memory of fixed size used to store variables local to a particular method. When the method exits, all stack variables associated with it are lost.

Java and .NET strictly define the size and sign of each basic data type in order to ensure architecture neutrality and platform independence. In contrast, the size and sign of C and C++ basic types may vary between platforms. For example, in C, an `int` type may consume 16, 32, or 64 bits depending on the platform. On the other hand, an `int` in Java and C# is fixed at 32 bits regardless of the platform.

Though the Java and C# basic data types are very similar, there is a distinct difference. Java primitives are special types that do not correspond to any class. Conversely, C# simple types are actually aliases to special value type classes defined by the .NET Framework. For instance, the C# `int`, `long`, `float`, and `double` types serve as aliases for the `System.Int32`, `System.Int64`, `System.Single`, and `System.Double` .NET classes, respectively. All .NET basic data types reside under the `System` namespace. Because they are actually classes, you can invoke methods on C# basic data types much like you would any other class. For example, the following C# code converts an integer to a string using the `ToString()` method (strings are discussed in the upcoming Complex Data Types section):

```
int num = 123;
string myString = num.ToString();
```

Or, you can skip the variable declaration altogether and invoke the `ToString()` method directly on the number itself:

```
string myString = 123.ToString();
```

Since the C# simple types are just aliases for .NET classes, it is perfectly legal to use their corresponding classes rather than the alias (if you don't mind some extra typing):

```
System.Int32 num = 10;
System.String myString = num.ToString();
```

Java designers attempted to minimize the number of basic data types in order to simplify the language. In contrast, C# designers decided to provide a richer set of basic data types that include additional unsigned types that are not available in Java. Table 3.1 details each basic data type supported by Java and C#.

**Table 3.1** ***Java and C# Basic Data Types***

| Java Data Type | C# Data Type | C# Type Alias Of | Description |
|---|---|---|---|
| `byte` | `sbyte` | `System.SByte` | 8-bit signed integer |
| `--` | `byte` | `System.Byte` | 8-bit unsigned integer |
| `short` | `short` | `System.Int16` | 16-bit signed integer |
| `--` | `ushort` | `System.UInt16` | 16-bit unsigned integer |
| `int` | `int` | `System.Int32` | 32-bit signed integer |
| `--` | `uint` | `System.UInt32` | 32-bit unsigned integer |
| `long` | `long` | `System.Int64` | 64-bit signed integer |
| `--` | `ulong` | `System.UInt64` | 64-bit unsigned integer |
| `float` | `float` | `System.Single` | 32-bit floating point |
| `double` | `double` | `System.Double` | 64-bit floating point |
| `boolean` | `bool` | `System.Boolean` | Boolean (true or false) |
| `char` | `char` | `System.Char` | 16-bit unicode character |
| `--` | `decimal` | `System.Decimal` | 128-bit signed number |

***Integer***

Java supports four integer types: `byte`, `short`, `int`, and `long`. C# adds support for the signed or unsigned counterpart for each of the Java integer types. Thus, C# supports the following eight integer types: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, and `ulong`. Each Java data type maps to the corresponding C# type of the same name with the exception of `byte`. The `byte` data type is signed in Java (ranging from -128 to 127) but unsigned in C# (ranging from 0 to 255). For a signed `byte` that is equivalent to Java's version, use C#'s `sbyte` type.

An explicit cast is required whenever an integer conversion would result in a loss of precision. For instance, to assign a `double` value to an `int`, the `double` must be explicitly cast to an `int`. On the other hand, no cast is necessary to assign an `int` value to a `double`. The following code demonstrates this:

```
int i = 123;
double d = 1234567890.1234;

double d2 = i; //no cast required
int i2 = (int)d2; //explicit cast required
```

Java and C# integer types can be initialized using decimal (Base 10) or hexadecimal (Base 16) values. Decimal values are assumed by default and, therefore, require no special treatment. For hexadecimal values, use the `0x` prefix like this:

```
int decimal = 255;
int hex = 0xFF;
```

### *Floating Point*

Java and C# both support two floating point data types: `float` and `double`. As previously demonstrated, the `F` suffix must be used whenever a `float` value is assigned like this:

```
float f = 123.45F;
```

In addition to the floating point types, C# supports a 128-bit signed number named `decimal`. The `decimal` data type is intended for high precision calculations, such as large financial transactions, and is accurate to 28 digits. The `M` suffix must be used whenever a `decimal` value is assigned:

```
decimal value = 999999999.99M;
```

### *Boolean*

Java and C# both define a boolean data type but name it differently. Java uses the term `boolean` and C# uses `bool`. Java and C# boolean values are not interchangeable with integers in the same way they are in C and C++. A non-zero integer value does not equate to true. Instead, you must create a boolean expression when testing for non-zero values. For instance, to interpret a non-zero value us true, a boolean expression like this is required:

```
if (i != 0)
```

Similarly, a boolean variable cannot be cast to or from any other type. To convert between boolean and integer values similar to C (i.e., zero is false and non-zero is true), expressions like the following are required:

```
b = (i != 0); //converts an int to a boolean
i = b ? 1:0; //converts a boolean to an int using ternary operator
```

### *Character*

Java and C# both support a character data type named `char`. The `char` type consumes 16 bits and represents a single Unicode character. Using an explicit cast, a `char` variable can be converted to any integer type and vice-versa:

```
byte b = 65;
```

```
char c = (char)b; //cast byte to char
short s = (short)c; //cast char to short
```

A character literal is contained within single quotes. For example, the following code creates a `char` array containing four characters (arrays are discussed in the upcoming Complex Data Types section):

```
char[] chars = new char[4];
chars[0] = 'D';
chars[1] = 'a';
chars[2] = 't';
chars[3] = 'a';
```

In addition to character literals, `char` values can be assigned using hexadecimal, Unicode, or escape sequence values as follows:

```
char hex = '\x0009';
char unicode = '\u0009';
char escape = ‘\t’;
```

Table 3.2 documents the escape sequence, hexadecimal, and Unicode values for some special characters that are commonly assigned to `char` variables.

**Table 3.2** ***Escape Sequence, Unicode, and Hex Values for Special Characters***

| Escape Sequence | Unicode Value | Hex Value | Description |
|---|---|---|---|
| `\\` | `\u005c` | `\x005c` | Backslash |
| `\′` | `\u0027` | `\x0027` | Single quote |
| `\″` | `\u0022` | `\x0022` | Double quote |
| `\b` | `\u0008` | `\x0008` | Backspace |
| `\f` | `\u000c` | `\x000c` | Form feed |
| `\n` | `\u000a` | `\x000a` | Line feed |
| `\r` | `\u000d` | `\x000d` | Carriage return |
| `\t` | `\u0009` | `\x0009` | Tab |

## Complex Data Types

Data types that are capable of storing more than just a single number or character are referred to as complex data types. In Java, all complex data types are reference types. That is, they are heap allocated objects that are passed by reference. Until they are initialized, all reference types are assigned a default value of `null`. Complex reference types include strings, arrays, and classes. C# supports these same reference types as well as two complex value types: structs and enumerations. Structs and enumerations are stack allocated objects that are passed by value. In this section, we will examine each of these complex data types.

### *Strings*

Java and C# strings differ greatly from the manner in which strings are implemented in C and C++. While C-style strings are just an array of characters, Java and C# strings are first-class objects. Unlike C and C++, Java and C# strings are type safe, bounds checked, and easy to process. Common functionality such as copying, searching, substitution, and concatenation are built into the string class.

Though strings are objects, C# makes them appear as if they are basic data types. This is due to the fact that the `string` data type in C# is an alias for the `System.String` class. In addition to defining a `string` alias, C# makes using strings easier by allowing them to be created without using the `new` operator. For example, in C# strings are initialized like this:

```
string message = “Hello World!”;
```

In Java, strings are declared using the `String` class that resides within the `java.lang` package (remember that `java.lang` is automatically imported so references to `String` need not be fully qualified). Since Java uses the name of an object rather than an alias, a Java `String` begins with a capital letter and is initialized like this:

```
String message = “Hello World!”;
```

Additionally, you can initialize a Java String just like you would create a normal object using the new operator like this:

```
String message = new String(“Hello World!”);
```

Be aware that the `System.String` .NET class does not support initializing strings in this manner.

As you can see, Java and C# both treat strings a little differently than normal objects. This is for good reason because once you start looking into strings you will see that they are not normal objects. Unlike most objects, strings are immutable. Once created, a string can never be altered. If you attempt to alter a string by character substitution or concatenation, Java and C# automatically create an entirely new string. For instance, consider the following C# code:

```
string message = "Hello";
message += " World";
message += "!";
```

Though this code appears to alter the `message` string, it actually creates three separate strings: “Hello”, “Hello World”, and “Hello World!”. Each of these strings is stored in a string table and are reused whenever an identical string is created. For example, consider this Java code:

```
String message1 = “Hello”;
String message2 = “Hello”;
```

Though it appears that this code creates two separate strings, it actually creates two references to the same string object. This is because whenever a string is initialized, Java and .NET first check the string table to determine if an identical string has already been created. If so, a reference to this string is returned. Otherwise, a new string object is created and added to the string table and a reference to the new string is returned.

As previously demonstrated, string concatenation can be accomplished using the `+=` operator. However, since this operation creates a new string for each concatenation, it is extremely inefficient. In Chapter 5, we will learn how to grow a string without having to create a new object for each new concatenation.

### *Arrays*

In C and C++, an array is simply a pointer to a location in memory where a series of elements are stored. In contrast, Java and C# arrays are reference objects that are allocated on the heap. Java and C# arrays are created by declaring the number and type of elements the array will contain. For example, the following code creates an `int` array of 10 elements:

```
int[] myArray = new int[10];
```

It is typically considered good form to associate the array brackets with the type of array being declared (as opposed to the name of the array). In fact, C# syntax requires this form and will generate a compilation error if the brackets are attached to the array name. However, either form is legal in Java. For instance, in Java, the previous array could also have been declared like this:

```
int myArray[] = new int[10];
```

Like C and C++, Java and C# array indexes are zero-based. Therefore, to access the first and last elements of `myArray`, use indexes 0 and 9, respectively:

```
int firstValue = myArray[0];
int lastValue = myArray[9];
```

### Array Initialization

Unlike C and C++, all elements within Java and C# arrays are automatically initialized to their default values when the array is created. For instance, all elements of an `int` array are automatically initialized to zero and all elements of a string array are initialized to `null`.

Similar to C and C++, Java and C# arrays can be initialized at the same time they are declared. The following C# code declares `int` and `string` arrays each consisting of three elements:

```
int[] intArray = new int[] {1, 2, 3};
string[] stringArray = new string[] {"one", "two", "three"};
```

Java and C# also support the following abbreviated array declaration syntax:

```
int[] intArray = {1, 2, 3};
string[] stringArray = {"one", "two", "three"};
```

The first declaration style is a bit more descriptive. However, since it requires less typing, the second style is more common.

### Multidimensional Arrays

A multidimensional array is essentially an array of arrays. There are two types of multidimensional arrays: rectangular and jagged. In order for a multidimensional array to be *rectangular*, each array within the multidimensional array must contain the same number of elements. In other words, each row must contain the same number of columns. Otherwise, the array is considered *jagged* (sometimes referred to as *ragged*).

Java and C# use slightly different syntax when declaring rectangular arrays. This is how a 3 by 2 rectangular array is declared in Java:

```
int[][] ra = new int[3][2];
```

Likewise, this is how a 3 by 2 rectangular array is declared in C#:

```
int[,] ra = new int[3,2];
```

Both of these declarations create a 3 element `int` array whose elements each contain a 2 element `int` array. The following Java code declares and populates a 3 by 2 rectangular array:

```
int[][] ra = new int[3][2];
ra[0][0] = 1;
ra[0][1] = 2;
ra[1][0] = 3;
ra[1][1] = 4;
ra[2][0] = 5;
ra[2][1] = 6;
```

This is how a 3 by 2 rectangular array is declared and populated in C#:

```
int[,] ra = new int[3,2];
ra[0,0] = 1;
ra[0,1] = 2;
ra[1,0] = 3;
ra[1,1] = 4;
ra[2,0] = 5;
ra[2,1] = 6;
```

Additionally, the following Java and C# examples demonstrate how rectangular arrays can be initialized at declaration time:

```
int[][] ra = {{1,2}, {3,4}, {5,6}}; //initializes Java rectangular array
```

```
int[,] ra = {{1,2}, {3,4}, {5,6}}; //initializes C# rectangular array
```

In the same way that Java and C# syntax differs when declaring a rectangular array, the syntax also differs when referencing elements of the array. The following code demonstrates how an element within a rectangular array is accessed in Java and C#:

```
int i = a[1][0]; //accessing a rectangular array element in Java
```

```
int i = a[1,0]; //accessing a rectangular array element in C#
```

Though only two-dimensional arrays have been presented, there is no pre-set limit to the number of dimensions supported by Java and C# arrays. For example, the following Java and C# code declares a 4 by 3 by 2 array:

```
int[][][] ra = new int[4][3][2]; //Java three-dimensional array
```

```
int[,,] ra = new int[4,3,2]; //C# three-dimensional array
```

Jagged arrays are declared identically in both Java and C#. For example, the following code declares a two-dimensional jagged array:

```
int[][] ja = new int[3][];
```

Notice that the size of the second dimension is not declared. The size of the second dimension must be explicitly declared for each of the three elements in the first dimension. For example, the following code declares a different size for each of the second dimension arrays:

```
int[][] ja = new int[3][];
ja[0] = new int[4];
ja[1] = new int[3];
ja[2] = new int[2];
```

This same jagged array can be declared on one line as follows:

```
int[][] ja = {new int[4], new int[3], new int[2]};
```

Like rectangular arrays, jagged arrays can be declared to support more than just two dimensions. For example, the following code declares a three-dimensional jagged array:

```
int[][][] ja = new int[4][][];
```

Note that the size of the first dimension must always be specified.

Unlike rectangular arrays, the syntax for accessing elements of a jagged array is identical in Java and C#. For instance, the first column of the second row of a two-dimensional jagged array can be accessed like this:

```
int i = ja[1][0];
```

### *Classes*

Java and C# both support the `class` complex data type. A *class* is a collection of data fields and the methods that operate on that data. A concrete instance of a class is known as an *object*. In other words, classes are the templates from which objects are instantiated. Instantiated classes (i.e., objects) are complex types that are allocated on the heap and passed by reference.

Unlike C and C++, Java and C# do not support global functions or constants. Rather, every Java and C# program must contain at least one class and all data structures must belong to a class. The class is the central unit around which object-oriented programs are built. Listing 3.13 presents a simple class that contains two public fields (`x` and `y`) and two public methods (`add()` and `multiply()`). This class is identical in Java and C#.

**Listing 3.13** ***Simple Class (Java, C#)***

```
public class Calculate
{
  public int x;
  public int y;

  public int add()
  {
    return x + y;
  }

  public int multiply()
  {
    return x * y;
  }
}
```

Listings 3.14 and 3.15 demonstrate how the `Calculate` class can be used from a Java and C# program, respectively.

**Listing 3.14** ***Java Program that uses the Calculate Class (Java)***

```
public class Test
{
  public static void main(String[] args)
  {
    Calculate calc = new Calculate();
    calc.x = 5;
    calc.y = 5;
    int result = calc.add();
    System.out.println(calc.x + " + " + calc.y + " = " + result);

    calc.x = 6;
    calc.y = 6;
    result = calc.multiply();
    System.out.println(calc.x + " * " + calc.y + " = " + result);
  }
}
```

**Listing 3.15** ***C# Program that uses the Calculate Class (C#)***

```
using System;

public class TestCalculate
{
  public static void Main()
  {
    Calculate calc = new Calculate();

    calc.x = 5;
    calc.y = 5;
    int result = calc.add();
    Console.WriteLine("{0} + {1} = {2}", calc.x, calc.y, result);

    calc.x = 6;
    calc.y = 6;
    result = calc.multiply();
    Console.WriteLine("{0} * {1} = {2}", calc.x, calc.y, result);
  }
}
```

The output from Listings 3.14 and 3.15 looks like this:

```
5 + 5 = 10
6 * 6 = 36
```

This has been a very brief introduction to the `class` complex data type. For a more thorough discussion (including coverage of inheritance and interfaces), see the Object-Oriented Programming section later in this chapter.

***Structures (C#)***

In C, a structure is a collection of variables grouped within a single data type. A structure is defined using the `struct` keyword. Though not supported by Java, C# adopts and enhances the `struct` data type. In fact, the C# structure differs significantly from the C version. The C# `struct` type is very similar to a `class`. Like a `class`, the `struct` type stores data and methods that operate on that data. A structure may contain the same members as a class such as fields, methods, constructors, properties, and constants. Structures can also implement multiple interfaces just like a class.

Of course, there are several critical differences between structures and classes. The principle distinction is that, like a C# basic type, a structure is a value type that is allocated on the stack rather than the heap. Since accessing values on the stack is faster than retrieving objects from the heap, there are times when structures can provide better performance than classes. However, since a new copy must be created whenever a `struct` is assigned to another `struct` or passed to a method, structures are usually kept small in order to minimize memory consumption. Listing 3.16 demonstrates how a simple structure is defined:

**Listing 3.16** ***Simple Struct Example (C#)***

```
public struct Point
{
  public int x;
  public int y;
}
```

Unlike Java where the number of primitives is fixed, structures allow the C# programmer to define new types that behave just like built-in basic data types. Like basic data types, structures can be initialized without using the `new` operator. Listing 3.17 presents a simple program that determines the distance between two points using the `Point` structure. Notice that the structure can be used without explicitly instantiating it.

**Listing 3.17** ***Using the Point Struct (C#)***

```
using System;

public class DistanceBetweenPoints
{
  public static void Main()
  {
    Point point1;
    point1.x = 5;
    point1.y = 10;

    Point point2;
    point2.x = 10;
    point2.y = 20;

    //calculate horizontal and vertical distances
    int xdiff = Math.Abs(point2.x - point1.x);
    int ydiff = Math.Abs(point2.y - point1.y);

    //calculate total distance between points
    double distance = Math.Sqrt(Math.Pow(xdiff, 2) +
      Math.Pow(ydiff, 2));

    Console.WriteLine("Distance between ({0},{1}) and " +
      "({2},{3}) is {4}", point1.x, point1.y, point2.x,
      point2.y, distance);
  }
}
```

Though it is not required, it is possible to instantiate a structure using the `new` operator as follows:

```
Point point1 = new Point();
point1.x = 5;
point1.y = 10;
```

The `new` operator is useful whenever a structure implements a constructor that defines one ore more parameters. For instance, by defining the appropriate constructor (constructors will be discussed a little later in this chapter), the `Point` structure could be instantiated and initialized as follows:

```
Point point1 = new Point(5, 10);
```

In order to illustrate the primary difference between structures and classes, Listing 3.18 demonstrates how structures are value types and classes are reference types.

**Listing 3.18** ***Comparing Structs and Classes (C#)***

```
using System;

public class StructClass
{
  public static void Main()
  {
    PointStruct struct1;
    struct1.x = 1;
    struct1.y = 2;

    PointStruct struct2 = struct1;
    struct2.x = 10;
    struct2.y = 20;

    PointClass class1 = new PointClass();
    class1.x = 3;
    class1.y = 4;

    PointClass class2 = class1;
    class2.x = 30;
    class2.y = 40;

    Console.WriteLine("Struct1: ({0}, {1})", struct1.x,
      struct1.y);
    Console.WriteLine("Struct2: ({0}, {1})", struct2.x,
      struct2.y);
    Console.WriteLine("Class1: ({0}, {1})", class1.x, class1.y);
    Console.WriteLine("Class2: ({0}, {1})", class2.x, class2.y);
  }
}

public struct PointStruct
{
  public int x;
  public int y;
}

public class PointClass
{
  public int x;
  public int y;
}
```

The output from Listing 3.18 looks like this:

```
Struct1: (1, 2)
```

```
Struct2: (10, 20)
Class1: (30, 40)
Class2: (30, 40)
```

Notice that assigning `struct1` to `struct2` results in an entirely new copy of the structure. Because it is a new copy, changing the value of `struct2` does not affect the value of `struct1`. On the other hand, assigning `class1` to `class2` only copies the reference. No new object is created. Since both references point to the same object, changing the value of `class2` also alters the value of `class1`.

Though they are very similar to classes in many ways, structures do not support inheritance. That is, unlike classes, structures cannot inherit functionality from any type nor can any type inherit from a structure. However, like all C# data types, structures are derived from `System.Object` and have access to all of `Object`'s methods.

Now that you know what a structure is, you might be wondering when to use it. Typically, structures are useful whenever it is desirable to pass a parameter by value rather than reference. For instance, sometimes it is necessary to ensure that when a particular object is passed to a method, it will not be modified (since any modification may affect code outside of the method). In order to guarantee that a reference type will not be altered, a defensive copy of the object must be created and passed to the method in question. In this manner, the method can only alter the copy while the original object is undisturbed. In contrast, because structures are passed by value, no defensive copy is required. A copy is made automatically when the method is invoked. Furthermore, since structures are allocated on the stack, they do not create objects in the heap that must be garbage collected. Considering the overhead required to create, maintain, and destroy objects in the heap, there are times when a structure may be more efficient than a class.

Alright, so now you might be wondering when not to use a structure. Though they have some advantages over classes, structures are usually used sparingly. This is because every time a structure is assigned to a variable or passed to a method, a new structure must be created and populated with the entire contents of the original structure. Copying large structures can be processor and memory intensive. For this reason, structures are usually limited to representing small and simple data types.

### *Enumerations (C#)*

In C, an enumeration is a data type that can contain only a single integer value from a finite collection of named constants. Though not supported in Java, C# implements the enumeration data type similar to C. Defined using the `enum` keyword, enumerations provide a type-safe method of limiting the valid values a variable may contain. If an enumeration is assigned a value outside of the valid range, an error is generated at compile time. An enumeration type that can be assigned one of four valid values is defined like this:

```
public enum Direction
{
  NORTH,
  SOUTH,
  EAST,
  WEST
}
```

The `Direction` enumeration can then be declared and initialized like this:

```
Direction go = Direction.SOUTH;
```

Keep in mind that `NORTH`, `SOUTH`, `EAST`, and `WEST` are the only values that can be assigned to the `Direction` enumeration. Attempting to assign any other value will generate a compilation error.

All enumeration constants are assigned an integer value. By default, each constant will be assigned an incremental integer starting at zero. Therefore, the integer values of the `Direction` enumeration's constants `NORTH`, `SOUTH`, `EAST`, and `WEST` are 0, 1, 2, and 3, respectively. If the automatically assigned values are not appropriate, you can also assign a custom integer value to each constant like this:

```
public enum Direction
{
  NORTH = 10,
  SOUTH = 20,
  EAST = 30,
  WEST = 40
}
```

Enumerations improve code readability and reduce bugs by eliminating "magic numbers" from your code and ensuring that variables can only contain expected values. Listing 3.19 demonstrates the use of an enumeration. Note that the `switch` statement will be discussed a little later in this chapter.

**Listing 3.19** ***Using Enumerations (C#)***

```
using System;

public enum Direction
{
  NORTH,
  SOUTH,
  EAST,
  WEST
}

public class EnumerationExample
{
  public static void Main()
  {
    Direction n = Direction.NORTH;
    Direction s = Direction.SOUTH;
    Direction e = Direction.EAST;
    Direction w = Direction.WEST;

    Console.WriteLine("Direction: {0}, Value: {1}", n, (int)n);
    Console.WriteLine("Direction: {0}, Value: {1}", s, (int)s);
    Console.WriteLine("Direction: {0}, Value: {1}", e, (int)e);
    Console.WriteLine("Direction: {0}, Value: {1}", w, (int)w);
    Console.WriteLine();

    go(Direction.SOUTH);
  }

  public static void go(Direction d)
  {
    switch (d)
    {
      case Direction.NORTH:
        Console.WriteLine("Moving North!");
        break;
      case Direction.SOUTH:
        Console.WriteLine("Moving South!");
        break;
      case Direction.EAST:
        Console.WriteLine("Moving East!");
        break;
      case Direction.WEST:
        Console.WriteLine("Moving West!");
        break;
    }
  }
}
```

The output from Listing 3.19 looks like this:

```
Direction: NORTH, Value: 0
Direction: SOUTH, Value: 1
Direction: EAST, Value: 2
Direction: WEST, Value: 3

Moving South!
```

You might be wondering how Java developers manage to program without enumerations. Typically, special Java classes that expose public constants are used as substitutes for enumerations. For example, the following Java class defines the constants contained in the `Direction` enumeration and is used in a similar manner:

```
public class Direction
{
  public static final int NORTH = 0;
  public static final int SOUTH = 1;
  public static final int EAST = 2;
  public static final int WEST = 3;
}
```

Though it appears equivalent, there is a distinct disadvantage to using public constants within a class in place of a proper enumeration. The problem with this approach is that it is not type safe. That is, any method that expects a `Direction` value must accept an integer parameter. Since the integer range is far broader than the range of valid values, there is no guarantee that the value passed to the method is valid.

The following class demonstrates another popular method used by Java programmers to overcome the lack of enumerations:

```
public final class Direction
{
  private Direction() //private constructor
  {
  }

  public static final Direction NORTH  = new Direction();
  public static final Direction SOUTH = new Direction();
  public static final Direction EAST = new Direction();
  public static final Direction WEST = new Direction();
}
```

This class allows a value to be passed in a type-safe manner similar to enumerations. Since it is declared `final` and the default constructor is private, the class cannot be extended or instantiated (these modifiers are discussed later in this chapter). Rather, the only possible instantiations of this class are the four public objects contained within it. Therefore, a method that accepts a `Direction` parameter can be assured that its value is one of the four valid objects. Listing 3.20 demonstrates how the `Direction` class is used.

**Listing 3.19** ***Passing a Type Safe Value (Java)***

```
public class DirectionTest
{
  public static final void main(String[] args)
  {
    Direction d = Direction.SOUTH;

    go(d);
  }

  public static void go(Direction tsd)
  {
    if (tsd == Direction.NORTH)
      System.out.println("Moving North!");
    else if (tsd == Direction.SOUTH)
      System.out.println("Moving South!");
    else if (tsd == Direction.EAST)
      System.out.println("Moving East!");
    else if (tsd == Direction.WEST)
      System.out.println("Moving West!");
  }
}
```

Though it provides a type-safe way of passing parameters, this method is still lacking when compared with an enumeration. First, unlike enumerations, integer values are not used for comparison. Rather, this method relies on comparing object references. Using object references can be problematic when used in distributed environments or when classes are serialized and deserialized. This is due to the fact that when a class is passed to a remote method, it is marshaled across the network and then reconstructed as a new instance at the destination. Since a remote call always results in a new instance, the object reference comparison will return false. To solve this problem, integer values can be added to each object as shown here:

```
public final class Direction
{
  public final int value;

  private Direction(int value)
  {
    this.value = value;
  }

  public static final Direction NORTH  = new Direction(0);
  public static final Direction SOUTH = new Direction(1);
  public static final Direction EAST = new Direction(2);
  public static final Direction WEST = new Direction(3);
}
```

**NOTE**
When assigning the value of a method parameter to an instance variable, it is a common practice to use the same name for both variables and make the assignment as follows:

```
      private Direction(int value)
      {
        this.value = value;
      }
```

`this.value` refers to the instance variable while `value` refers to the local method parameter.

Since the `value` integer variable is marked `final`, the client can read but not alter its value. This class requires the following changes to the `DirectionTest` program:

```
public class DirectionTest
{
  public static final void main(String[] args)
  {
    Direction d = Direction.SOUTH;

    go(d);
  }

  public static void go(Direction d)
  {
    if (d.value == Direction.NORTH.value)
      System.out.println("Moving North!");
    else if (d.value == Direction.SOUTH.value)
      System.out.println("Moving South!");
    else if (d.value == Direction.EAST.value)
      System.out.println("Moving East!");
    else if (d.value == Direction.WEST.value)
      System.out.println("Moving West!");
  }
}
```

At last, we have a type-safe manner of passing values that works equally well whether used locally or passed to a remote method. However, as you can see, overcoming the lack of a true enumeration type requires considerable effort on the Java programmer's part. For this reason, many Java programmers choose to use the public constants technique, sacrificing type safety for simplicity.

## Variables

In Java and C#, there are four primary types of variables: class, instance, parameter, and local. *Class variables* are declared using the `static` modifier and are initialized one time when the class is first accessed. Class variables are declared outside of any method and are shared across all class instances. *Instance variables* are declared outside of any method and are initialized each time the class is instantiated. In contrast to class and instance variables, *parameter variables* and *local variables* are declared within a method and are initialized each time the method is invoked. The following class declares a class, instance, parameter, and local variable named `classVar`, `instanceVar`, `paramVar`, and `localVar`, respectively:

```
public class Variables
{
  static int classVar;
  int instanceVar;

  public void evaluate(int paramVar)
  {
    int localVar;
  }
}
```

***Modifiers***

In addition to `static`, Java and C# support other variable modifiers. Most of these other modifiers are known as *access modifiers* because they affect the accessibility of a variable. Table 3.3 details the variable modifiers defined by Java and C#.

**Table 3.3** ***Java and C# Variable Modifiers***

| **Java Modifier** | **C# Modifier** | **Description** |
|---|---|---|
| `private` | `private` | Accessible from within the variable's class. |
| `private` | *none* | Accessible from within the variable's class (same as `private`). |
| `--` | `protected` | Accessible from within the variable's class or any derived class. |
| *none* | `internal` | Accessible by any class within the same package or assembly. |
| `protected` | `protected internal` | Accessible from within the variable's class or any derived class or from any class within the same package or assembly. |
| `public` | `public` | Accessible from any class. |
| `static` | `static` | Indicates a static class-level variable (as opposed to an instance variable). |
| `transient` | `[NonSerialized]` | Indicates a variable whose value is not part of the object's state and should not be serialized. |
| `volatile` | `--` | Indicates that the field is not thread-safe. |

The "`--`" shown in Table 3.3 indicates that there is no equivalent to the specified characteristics described in the Description column. The "none" modifiers indicate the default accessibility levels of variables that do not explicitly specify an access modifier. Java and C# differ in their treatment of these variables. Absent an access modifier, C# variables default to private access while Java variables are accessible to all classes within the same package (sometimes called *package access*).

Additionally, Java and C# define modifiers that prevent a variable's value from being altered. These modifiers allow for the declaration of constants and read-only variables. Constants and read-only variables both have fixed values. The difference is that constants are static and must be initialized when they are declared. On the other hand, read-only variables are instance variables that may be initialized at declaration, in the class's constructor, or in a static initializer (constructors and static initializers are covered in the Object-Oriented Programming section later in this chapter). Of course, once a read-only variable's value is set, it cannot be changed.

Let's start by examining constants. In Java, a constant is declared like this:

```
static final PI = 3.14159;
```

The `final` modifier indicates that once this variable's value has been set, it cannot be changed. The `static` modifier indicates that a single copy of this variable is shared across all class instances. In C#, a constant is declared using the `const` modifier like this:

```
const PI = 3.14159;
```

The `const` modifier is implicitly static. If a constant is not assigned a value at the time it is declared, the compiler will generate an error. Lastly, in addition to literals, constants can be initialized using other constants like this (in Java):

```
static final double RADIUS = 10;
static final double PI = 3.14159;
static final double CIRCUMFRENCE = RADIUS * 2 * PI;
```

Or like this in C#:

```
const double RADIUS = 10;
const double PI = 3.14159;
const double CIRCUMFRENCE = RADIUS * 2 * PI;
```

To declare an instance variable as read-only, use the `final` modifier in Java (without the `static` modifier) like this:

```
final customerID;
```

Or the `readonly` modifier in C# like this:

```
readonly customerID;
```

***Scope***

Unlike C and C++, Java and C# both require local variables within a given scope to be named in a manner that allows them to be referenced unambiguously. For instance, the following code does not compile because the variable `j` is declared twice within the same scope:

```
public void evaluate()
{
  int j = 10;

  for (int i = 0; i < 10; i++)
  {
    int j = i * 2; //declaring j again causes a compile error
  }
}
```

In contrast, local variables can hide variables of the same name that are declared outside of the method. The compiler does not object to this type of variable hiding because class and instance variables can be uniquely referenced using the name of the class or the `this` keyword, respectively. For example, the `bankNum` and `accountNum` method parameters in the following class hide the instance variables of the same name:

```
public class VariableScope
{
  static int bankNum;
  int accountNum;

  public void evaluate(int bankNum, int accountNum)
  {
    //use class name to assign parameter value to class variable
    VariableScope.bankNum = bankNum;

    //use "this" to assign parameter value to instance variable
    this.accountNum = accountNum;
  }
}
```

**NOTE**
In Java, class variables can be referenced using either the class name or the `this` keyword (e.g., `VariableScope.bankNum` or `this.bankNum`). On the other hand, in C#, the `this` keyword can be used to reference instance variables only. When qualifying a class variable, the class name must be used.

## Type Conversions

A *type conversion* is the act of converting one data type to another. In Java and C#, some type conversions are simply implied while others must be explicitly declared. Additionally, C# supports user-defined type conversions and automatic boxing. We will investigate each of these concepts in the following sections.

### *Implicit*

An *implicit conversion*, also known as an *automatic conversion*, is a type conversion that is performed automatically and can be executed without data loss. In order to guarantee that no data is lost, implicit conversions can only be made from the base type to a type of equal or greater range and precision. For example, an `int` can be implicitly converted to a `long` because the range of a `long` spans all possible values that may be assigned to an `int`. In contrast, an `int` cannot be automatically converted to a `byte` because the `byte` data type's range is not large enough to store all possible `int` values. Similarly, since it would result in the truncation of any values to the right of the decimal place, a `float` cannot be implicitly converted to a `long`. The following code demonstrates some implicit conversions between integer types:

```
byte b = 10;
short s = b;
int i = s;
long l = i;
```

An implicit conversion between floating point types looks like this:

```
float f = 10F;
double d = f;
```

In addition to the basic Java primitives, C# supports corresponding unsigned types as well. In order to determine when an implicit cast is legal, use the same criteria as previously described. That is, if a data type's entire range fits within the type's range, an implicit conversion can be performed. Therefore, it is not possible to implicitly convert between `int` and `uint` data types because neither type's range completely overlaps the other. However, a `uint` may be implicitly converted to a `long` because its range is a proper subset of the `long` data type's range. Table 3.4 documents the types to which each of the basic value types may be implicitly converted.

**Table 3.4** ***Implicit Conversions for Java and C# (italics indicates a C# only data type)***

| **From Data Type** | **To Data Type** |
|---|---|
| `byte` | `short`, *`ushort`*, `int`, *`uint`*, `long`, *`ulong`*, `float`, `double`, *`decimal`* |
| *`sbyte`* | `short`, `int`, `long`, `float`, `double`, *`decimal`* |
| `short` | `int`, `long`, `float`, `double`, *`decimal`* |
| *`ushort`* | `int`, *`uint`*, `long`, *`ulong`*, `float`, `double`, *`decimal`* |
| `int` | `long`, `float`, `double`, *`decimal`* |
| *`uint`* | `long`, *`ulong`*, `float`, `double`, *`decimal`* |

| | |
|---|---|
| long, *ulong* | float, double, *decimal* |
| float | double |
| char | *ushort*, int, *uint*, long, *ulong*, float, double, *decimal* |

***Explicit***

An *explicit conversion*, also known as a *casted conversion*, is a type conversion that requires the programmer to explicitly indicate the data type to which a variable should be converted. Explicit conversions are required whenever a conversion may result in a loss of data. For example, an explicit conversion is required when converting a `long` to an `int` since this operation could potentially result in data loss. An explicit conversion uses the cast syntax introduced by the C language. An explicit type cast is accomplished as follows:

```
int i = 100;
short s = (short)i;
```

Since the `short` data type's range does not fully span the range of an `int`, an explicit cast is required. This requirement helps ensure that the programmer is aware of the potential for data loss. As you would expect, after the conversion, the variable `s` contains the same value as the variable `i` (they both contain the value 100). However, this may not always be the case. Consider the following code:

```
int i = 32768;
short s = (short)i;
```

Since its upper limit is 32767, the `short` data type is unable to hold the value stored in the `int`. In order to accommodate the explicit cast, the value "wraps" to the `short` data type's lower limit. Therefore, while the `int` contains the value 32768, the short contains the value -32768. In this case, data was lost (i.e., the `short` does not contain the same value as the `int`).

Let's look at another example. The following code explicitly converts a `double` to a `long`:

```
double d = 12.99;
long l = (long)d;
```

Once this code executes, the variable `d` will contain the value 12.99 and the variable `l` will contain the value 12. Since a `long` does not support fractional values, all values to the right of the decimal place are truncated. At times, this truncation can be useful and may be desired. For example, the following code rounds the `double` value to the nearest whole dollar amount:

```
double price = 12.99;
long roundPrice = (long)(price + 0.5);
```

Though not required, explicit casts can be used wherever an implicit conversion is allowed. For instance, the following code explicitly converts a `short` to an `int` even though this conversion could have been performed implicitly:

```
short s = 10;
int i = (int)s;
```

***User-Defined (C#)***

Unlike Java, C# allows the programmer to define custom conversion rules for a class. Conversion rules can be defined for both implicit and explicit casts. To define how a class behaves when it is implicitly cast to another type, use the `implicit operator` method modifier like this:

```
public static implicit operator int(Customer c)
{
  return c.customerId;
}
```

The runtime environment automatically calls this method whenever the class is implicitly cast to an `int`. This allows the class to seamlessly return a suitable integer value. Since the `implicit operator` method is static, an instance of the class being cast is passed in as a parameter. Similarly, explicit casts are defined using the `explicit operator` method modifier as follows:

```
public static explicit operator string(Customer c)
{
  return c.customerName;
}
```

Like the `implicit operator` method, this method is called whenever an instance of its class is cast to a `string`.

**NOTE**
The name of a user-defined conversion method represents the type to which the object is being converted. The parameter passed to the conversion method is the object that is being converted.

To illustrate user-defined conversions, consider the following class:

```
public class Customer
{
  public int customerId = 512;
  public string customerName = "Kennedy Callaway";

  public Customer(int customerId, string customerName)
  {
    this.customerId = customerId;
    this.customerName = customerName;
  }

  public static implicit operator int(Customer c)
  {
    return c.customerId;
  }

  public static explicit operator string(Customer c)
  {
    return c.customerName;
  }
}
```

The `Customer` class is initialized using a public constructor (constructors are covered in the Object-Oriented Programming section later in this chapter). This class also defines methods to support implicitly converting the class to an `int` or explicitly converting it to a `string`. The following program demonstrates how these user-defined conversions are used:

```
using System;

public class ConversionTest
{
  public static void Main()
  {
    Customer c = new Customer(512, "Kennedy Callaway");

    int num = c;
    string name = (string)c;

    Console.WriteLine("Customer #: {0}\nCustomer Name: {1}",
      num, name);
  }
}
```

The output from `ConversionTest` looks like this:

```
Customer #: 512
Customer Name: Kennedy Callaway
```

The `ConversionTest` program demonstrates how user-defined conversions can simplify code. In this example, we were able to implicitly cast the `Customer` object to an `int` and explicitly cast it to a `string`. Remember that since the `string` conversion method was defined as `explicit`, an explicit cast is required or a compiler error is generated.

> **NOTE**
> Implicit conversion methods should only be defined for conversions that are guaranteed to succeed without any data loss. This is necessary since implicit conversions can take place inconspicuously. Any conversion that may lose data or throw an exception should be defined as `explicit`.

Not only can behavior be defined for converting a class to another type, it can also be declared for converting another type to a particular class. For example, the following class defines two `explicit` conversion methods that allow the class to be converted to and from a `string` using explicit casts:

```
public class Person
{
  public string name;

  public Person(string name)
  {
    this.name = name;
  }

  public static explicit operator string(Person p)
  {
    return p.name;
  }

  public static explicit operator Person(string name)
  {
    return new Person(name);
  }
}
```

After implementing the `explicit operator` methods, the `Person` class can be used like this:

```
using System;

public class Test
{
  public static void Main()
  {
    string myName = "Dustin Callaway";

    Person p = (Person)myName;
    string name = (string)p;

    Console.WriteLine("Person: {0}\nString: {1}", p.name, name);
  }
}
```

***Boxing (C#)***

In Java, in order to use a primitive value as a reference type, it must be placed within an object wrapper class. Java defines eight wrapper classes (all in the `java.lang` package) for each of the primitive types: `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, and `Boolean`. For example, if you want to store an `int` type in a collection or pass it to a method that accepts an `Object`, it must first be placed into a wrapper class like this:

```
int i = 10;
Integer bi = new Integer(i); //wrap integer
int j = bi.intValue(); //unwrap integer
```

After wrapping the `int`, it can be treated just like other `Object` types. For instance, though you can't call the `toString()` method on the primitive variable `i`, you can call `toString()` on the big integer `bi` (e.g., `bi.toString()`).

Like Java, a C# value type must be placed within a reference type before it can be treated as an object. However, unlike Java, the process of inserting a value type into an object wrapper is performed automatically. Similarly, the original value can be extracted from the object wrapper using a simple cast. In .NET, the process of inserting a value type into and extracting it from a wrapper class is known as *boxing* and *unboxing*, respectively. These terms are derived from the fact that when the value type needs to behave like an object, it is essentially placed within an object box. This box remembers the data type of its contents. When the box is converted back into its corresponding value type, it automatically extracts its contents and returns the appropriate value type (effectively "unboxing" its internal value). Consider the following code:

```
int i = 5;
object o = i; //boxes integer into an object
int j = (int)o; //unboxes integer out of the object
```

Note that boxing can be performed implicitly while unboxing requires an explicit cast.

Finally, keep in mind that the object box only stores a copy of the value type. Altering the original variable does not affect the object's value and vice-versa. The following code demonstrates this fact:

```
int i = 10;
object o = i;
i++;
```

```
System.Console.WriteLine("i = {0}, o = {1}", i, o);
```

When executed, the previous code produces the following output:

```
i = 11, o = 10
```

Notice that the value of object `o` did not change even though the value of variable `i` was altered.

> **NOTE**
> Boxing and unboxing works for all value types, including structs.

## Basic Operators

Java and C# support a similar set of basic operators that in most cases behave identically. The basic operators can be grouped into five categories: assignment, arithmetic, relational, logical, and bitwise.

### *Assignment*

Assignment operators are used to set a variable's value. Table 3.5 details the assignment operators defined by Java and C#. The compound assignment operators (e.g., `+=`, `-=`, `*=`, `/=`, etc.) perform the specified operation on the variable and then assign the result to the variable. For instance, this assignment:

```
i += 5;
```

is shorthand for:

```
i = i + 5;
```

The increment and decrement operators can be declared before (prefix) or after (postfix) the variable. A prefix declaration performs the operation first and then the assignment while a postfix declaration performs the assignment first and then the operation. For example, consider the following code:

```
i = 5;
j = 5;
k = i++; //postfix operation
l = ++j; //prefix operation
```

After this code is executed, the variables will have the following values:

```
i = 6
j = 6
k = 5
l = 6
```

Notice that both `i` and `j` were incremented. However, the variable `k` was assigned the value of `i` before it was incremented. In contrast, the variable `l` was assigned the value of `j` after it was incremented.

**Table 3.5** ***Assignment Operators (Java, C#)***

| **Operator** | **Description** |
|---|---|
| `=` | Simple assignment: `i = 5;` |
| `+=` | Addition assignment: `i += 5;` |

| | |
|---|---|
| `-=` | Subtraction assignment: `i -= 5;` |
| `*=` | Multiplication assignment: `i *= 5;` |
| `/=` | Division assignment: `i /= 5;` |
| `%=` | Modulus assignment (remainder): `i %= 5;` |
| `&=` | Logical AND assignment: `i &= 1;` |
| `\|=` | Logical OR assignment: `i \|= 1;` |
| `^=` | Logical exclusive OR assignment: `i ^= 1;` |
| `<<=` | Shift left assignment: `i <<= 1;` |
| `>>=` | Shift right assignment: `i >>= 1;` |
| `++` | Increment assignment: `i++;` or `++i;` |
| `--` | Decrement assignment: `i--;` or `--i;` |

***Arithmetic***

Arithmetic operators are used to perform mathematical calculations on numeric data types. Table 3.6 presents the arithmetic operators supported by Java and C#.

**Table 3.6** ***Arithmetic Operators (Java, C#)***

| **Operator** | **Description** |
|---|---|
| `+` | Addition (also used for string concatenation): `i = j + 5;` |
| `-` | Subtraction: `i = j - 5;` |
| `*` | Multiplication: `i  = j * 5;` |
| `/` | Division: `i = j / 5;` |
| `%` | Modulus (remainder): `i = j % 5;` |

***Relational***

Relational operators evaluate the relationship between two variables and return a boolean value indicating whether the stated relationship is true or false. Table 3.7 documents the relational operators defined by Java and C#.

**Table 3.7** ***Relational Operators (Java, C#)***

| **Operator** | **Description** |
|---|---|
| `==` | Equal to: `if (i == j)` |
| `!=` | Not equal to: `if (i != j)` |
| `<` | Less than: `if (i < j)` |
| `>` | Greater than: `if (i > j)` |
| `<=` | Less than or equal to: `if (i <= j)` |
| `>=` | Greater than or equal to: `if (i >= j)` |

***Logical***

Logical operators perform and/or comparisons and negation operations. Table 3.8 details the logical operators defined by Java and C#.

**Table 3.8** ***Logical Operators (Java, C#)***

| **Operator** | **Applicable Types** | **Description** |
|---|---|---|
| `&` | numeric, boolean | Bitwise AND: `i = j & k;` |
| `\|` | numeric, boolean | Bitwise OR: `i = j \| k;` |
| `^` | numeric, boolean | Bitwise XOR: `i = j ^ k;` |
| `~` | numeric | Bitwise compliment (inverts value of each bit): `i = ~i;` |

| | | |
|---|---|---|
| `!` | boolean | Logical negation: `i = !i;` |
| `&&` | boolean | Logical AND: `if (i && j)` |
| `\|\|` | boolean | Logical OR: `if (i \|\| j)` |

***Bitwise***
Bitwise operators shift the bits of numerical values to the left or right. Table 3.9 documents the bitwise operators supported by Java and C#.

**Table 3.9** ***Bitwise Operators***

| **Operator** | **Language** | **Description** |
|---|---|---|
| `<<` | Java, C# | Shift bits left: `i = j << 1; //shift left 1 bit` |
| `>>` | Java, C# | Shift bits right: `i = j >> 2; //shift right 2 bits` |
| `>>>` | Java | Shift bits right unsigned: `i = j >>> 1;` |

***Ternary Operator***
The ternary operator, also called the conditional operator, selects an expression to evaluate based on a boolean comparison. The ternary operator is formed as follows:

```
boolean_expression ? true_expression : false_expression;
```

For instance, consider the following code:

```
int var = (j < 100) ? 5 : 10;
```

In this example, if the expression `(j < 100)` evaluates to true, the variable `var` is assigned the value 5. Otherwise, it is assigned the value 10.

***Precedence***
Operator precedence determines the order in which elements of an expression are evaluated. With the exception of assignment operators, all Java and C# operators are evaluated left to right (i.e., left associative). In contrast, the assignment operators are performed right to left (i.e., right associative). Listed highest precedence to lowest, Table 3.10 lists operators grouped according to precedence. Operators within a group are evaluated according to their associative characteristics (right to left for assignment operators and left to right for all others).

**Table 3.10** ***Operator Precedence (Java, C#)***

| **Precedence** | **Description** |
|---|---|
| Highest | `() . []` |
| \| | `++ -- ! ~` |
| \| | `* / %` |
| \| | `+ -` |
| \| | `<< >> >>>` |
| \| | `< > <= >=` |
| \| | `== !=` |
| \| | `&` |
| \| | `^` |
| \| | `\|` |
| \| | `&&` |
| \| | `\|\|` |
| \| | `?:` |

|  |  |
|---|---|
| Lowest | = *= /= %= += -= <<= >>= &= ^= \|= |

***Overloading (C#)***

Unlike Java, C# supports operator overloading. Operator overloading allows the developer to define an operator's function when applied to a particular type of object. For example, the following method within the `Rectangle` class overloads the + operator:

```
public static Rectangle operator + (Rectangle a, Rectangle b)
{
  int newHeight = a.height + b.height;
  int newWidth = a.width + b.width;

  return new Rectangle(newHeight, newWidth);
}
```

This example indicates that when two `Rectangle` objects are added using the `+` operator, a new `Rectangle` is returned. The height and width of the new `Rectangle` is equal to the sum of the original `Rectangle`s' height and width values. The `operator` keyword in the method definition indicates that an operator overload is being defined. All operator overloads are declared as `public` and `static`.

In addition to arithmetic operators, comparison operators can be overridden as well. However, unlike arithmetic operators, comparison overloads are required to return a boolean value. To illustrate, overloading of the `+`, `-`, `==`, and `!=` operators is demonstrated in Listing 3.20.

**Listing 3.20** ***Overloading the Addition and Subtraction Operators (C#)***

```
public class Rectangle
{
  public int height;
  public int width;

  public Rectangle(int height, int width)
  {
    this.height = height;
    this.width = width;
  }

  //overload addition operator
  public static Rectangle operator + (Rectangle a, Rectangle b)
  {
    int newHeight = a.height + b.height;
    int newWidth = a.width + b.width;

    return new Rectangle(newHeight, newWidth);
  }

  //overload subtraction operator
  public static Rectangle operator - (Rectangle a, Rectangle b)
  {
    int newHeight = a.height - b.height;
    int newWidth = a.width - b.width;

    return new Rectangle(newHeight, newWidth);
  }

  //overload equals comparison operator
  public static bool operator == (Rectangle a, Rectangle b)
  {
    if ((a.height == b.height) && (a.width == b.width))
    {
      return true;
    }
    else
    {
      return false;
    }
  }

  //overload not equals comparison operator
  public static bool operator != (Rectangle a, Rectangle b)
  {
    if ((a.height == b.height) && (a.width == b.width))
    {
      return false;
    }
    else
    {
      return true;
    }
  }

  public static void Main()
  {
    Rectangle rec1 = new Rectangle(5, 10);
    Rectangle rec2 = new Rectangle(15, 25);
    Rectangle rec3 = new Rectangle(15, 25);

    Rectangle add = rec1 + rec2;
    Rectangle subtract = rec2 - rec1;

    System.Console.WriteLine("Addition: ({0}, {1})", add.height,
      add.width);
    System.Console.WriteLine("Subtraction: ({0}, {1})",
      subtract.height, subtract.width);

    System.Console.WriteLine("Equal: {0}", (rec2 == rec3));
    System.Console.WriteLine("Not Equal: {0}", (rec2 != rec3));
  }
}
```

When executed, Listing 3.20 generates the following output:

```
Addition: (20, 35)
Subtraction: (10, 15)
Equal: True
Not Equal: False
```

**NOTE**
It is recommended that whenever the == operator is overloaded, the `Equals()` method inherited from `System.Object` should be overridden so that == and `Equals()` always return the same value.

Since Java does not support operator overloading, you might be wondering how a Java program might be able to add two Rectangle objects. In Java, it is customary to define method to accomplish this sort of thing. For example, rather than overloading the + and – operators, Listing

3.21 defines add() and subtract() methods that accomplish the same thing (though perhaps not quite as elegantly).

**Listing 3.21** ***Adding and Subtracting Objects with Overloading Operators (Java)***

```
public class Rectangle
{
  public int height;
  public int width;

  public Rectangle(int height, int width)
  {
    this.height = height;
    this.width = width;
  }

  public static Rectangle add(Rectangle a, Rectangle b)
  {
    int newHeight = a.height + b.height;
    int newWidth = a.width + b.width;

    return new Rectangle(newHeight, newWidth);
  }

  public static Rectangle subtract(Rectangle a, Rectangle b)
  {
    int newHeight = a.height - b.height;
    int newWidth = a.width - b.width;

    return new Rectangle(newHeight, newWidth);
  }

  public static void main(String[] args)
  {
    Rectangle rec1 = new Rectangle(5, 10);
    Rectangle rec2 = new Rectangle(15, 25);

    Rectangle add = Rectangle.add(rec1, rec2);
    Rectangle subtract = Rectangle.subtract(rec2, rec1);

    System.out.println("Addition: (" + add.height + ", " +
      add.width + ")");
    System.out.println("Subtraction: (" + subtract.height +
      ", " + subtract.width + ")");
  }
}
```

In order to allow the programmer to define how a class is evaluated within a comparison, C# allows the `true` and `false` operators to be overloaded. These operators are usually overloaded when it is necessary to define whether a null value should evaluate to true or false. Listing 3.22 demonstrates overloading of the `true` and `false` operators.

**Listing 3.22** ***Overloading the true and false Operators (C#)***

```
public class TrueFalseTest
{
  public bool value;

  public TrueFalseTest(bool value)
  {
    this.value = value;
  }

  //overload true comparison operator
  public static bool operator true (TrueFalseTest tf)
  {
    if (tf == null)
    {
      return true; //"Object is true" statement is true
    }
    else
    {
      return tf.value;
    }
  }

  //overload false comparison operator
  public static bool operator false (TrueFalseTest tf)
  {
    if (tf == null)
    {
      return false; //"Object is false" statement is false
    }
    else
    {
      return tf.value;
    }
  }

  public static void Main()
  {
    TrueFalseTest tfTrue = new TrueFalseTest(true);
    TrueFalseTest tfFalse = new TrueFalseTest(false);
    TrueFalseTest tfNull = null;

    if (tfTrue)
    {
      System.Console.WriteLine("tfTrue is true");
    }
    else
    {
      System.Console.WriteLine("tfTrue is false");
    }

    if (tfFalse)
    {
      System.Console.WriteLine("tfFalse is true");
    }
    else
    {
      System.Console.WriteLine("tfFalse is false");
    }

    if (tfNull)
    {
      System.Console.WriteLine("tfNull is true");
    }
    else
    {
      System.Console.WriteLine("tfNull is false");
    }
  }
```

```
}
```

Listing 3.22 indicates that a null value will be evaluated to true. This is accomplished by returning true from the `true` overload method and false from the `false` overload method whenever a null value is compared. Upon first glance, you will probably find this very confusing. These values can be very difficult to understand. Think of it like this: the true method returns an answer to the question “is this object true?” and the false method returns an answer to the question “is this object false?” Therefore, if the object evaluates to true, the true overload method returns true and the false overload method returns false.

In C#, not all operators can be overloaded. Table 3.11 presents the operators for which overloading is supported. Though the compound assignment operators, such as `+=`, `-=`, `*=`, and `/=`, cannot be overloaded, they are evaluated according to the behavior defined for operators that can be overloaded (e.g., `+`, `-` , `*`, `/`).

**Table 3.11** ***Operators that can be Overloaded (C#)***

| **Group** | **Operators** |
|---|---|
| Arithmetic | `+ ++ - -- * / %` |
| Logical | `& \| ^ ! ~ true false` |
| Bitwise | `<< >>` |
| Comparison | `== != < <= > >=` |

## Branch Statements

Branch statements are used to alter a program’s flow of control. The branch statements supported by Java and C# include the `if` and `switch` statements.

### *if…else*

The `if` and `else` statements are used identically in Java and C#. The expressions processed by an `if` statement must evaluate to a boolean value. The following code briefly illustrates the `if` and `else` statements:

```
if (score > 90)
  message = "Excellent";
else if (score > 80)
  message = "Good";
else if (score > 70)
  message = "Average";
else
  message = "Needs improvement";
```

Unlike C, a non-zero integer value does not evaluate to true. Rather, you must fashion an expression that explicitly checks for a non-zero value like this:

```
if (i != 0)
  //do something
```

### *switch*

The `switch` statement evaluates an expression and directs program flow to the appropriate `case` block. For example, the following `switch` statement evaluates the integer variable `count` and executes the corresponding `case` statement:

```
int count = getCount();
```

```
switch (count)
{
  case 0:
    message = "No Strikes.";
    break;
  case 1:
    message = "Strike One!";
    break;
  case 2:
    message = "Strike Two!";
    break;
  case 3:
    message = "Strike Three!";
    break;
  default:
    message = "Take Your Base!";
    break;
}
```

The values in case statements must be constants. The `default` statement serves as a "catch all" that is executed if none of the `case` statements apply. Though it typically comes last, the `default` block may be defined before or after any of the `case` blocks.

The `switch` statement is often preferable to a long series of `if..else` statements. Though its usage is similar, the `switch` statement behaves a little differently in Java and C#. For instance, if a `case` does not end with a `break` statement, Java allows program flow to "fall through" to the next `case`. To illustrate, if the `count` variable equals 2, the following code will print "Strike Two!" and "Strike Three!":

```
int count = getCount();

switch (count)
{
  case 1:
    System.out.println("Strike One!");
    break;
  case 2:
    System.out.println("Strike Two!");
  case 3:
    System.out.println("Strike Three!");
    break;
}
```

In contrast, unless the `case` block is empty, C# requires that each `case` ends with a `break` or `goto` statement. This requirement is meant to eliminate a common source of inconspicuous bugs resulting from the accidental omission of the `break` statement. If this rule is violated, the compiler will generate an error. Fortunately, since C# (as well as Java) allows empty `case` blocks to fall through, the following code is perfectly legal:

```
int num = getNum();

switch (num)
{
  case 1:
  case 3:
  case 5:
    message = "Odd Number";
    break;
  case 2:
  case 4:
  case 6:
    message = "Even Number";
    break;
  default:
    message = "Not Sure";
    break;
}
```

As mentioned previously, in C# all non-empty `case` blocks must end with a `break` or `goto` statement. Since program flow cannot implicitly pass from one non-empty case block to the next, the `goto` statement allows the programmer to explicitly state this desire. For example, the following code demonstrates how one `case` block can explicitly jump to another:

```
int num = getNum();

switch (num)
{
  case 1:
    Console.WriteLine("Hello.");
    goto case 3;
  case 2:
    Console.WriteLine("How do you do?");
    break;
  case 3:
    Console.WriteLine("Good bye.");
    break;
}
```

In this example, `case` 1 jumps directly to `case` 3 (printing both “Hello.” and “Good bye.”). Though it is a reserved word, Java does not currently support the `goto` statement in any form.

The last significant difference between the Java and C# `switch` statements involves support for data types. The C# `switch` statement supports a much broader range of data types. For example, the Java `switch` statement supports the `byte`, `short`, `int`, and `char` data types. On the other hand, the C# `switch` statement supports `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `enum`, and `string` data types. The C# `switch` statement’s support for the `string` data type is especially noteworthy. The following code demonstrates C#’s ability to switch on strings:

```
string time = getTimeOfDay();

switch (time)
{
  case "Morning":
    Console.WriteLine("Good morning.");
    break;
  case "Afternoon":
    Console.WriteLine("Good afternoon.");
    break;
  case "Evening":
    Console.WriteLine("Good evening.");
    break;
}
```

Note that `string` comparisons in `switch` statements are case sensitive.

## Iteration Statements

An *iteration statement*, also known as a *loop*, is used to execute a block of code multiple times. The number of times the code is executed depends on the parameters of the iteration statement. Java and C# support the `for`, `while`, and `do`..`while` statements. In addition, C# supports the `foreach` iteration statement.

### ***`for`***

The `for` statement executes a block of code a specified number of times. The `for` statement accepts three semi-colon delimited expressions. The first expression is used to initialize the looping variable. The second expression evaluates whether or not the `for` statement should continue to loop. The final expression, usually used to increment the value of the looping variable, is executed after each iteration. All three expressions are optional. The for statement works identically in Java and C#. The following Java code demonstrates a typical `for` loop that prints numbers from 1 to 10:

```
for (int i = 1; i <= 10; i++)
{
  System.out.println(i);
}
```

This `for` loop initializes the integer `i` to 1, increments it after each iteration, and stops looping when `i` is greater than 10.

### ***`while`***

The `while` statement continues to iterate as long as its boolean expression evaluates to true. The `while` statement works the same in Java and C#. The following C# `while` loop prints the numbers 1 through 10:

```
int i = 1;

while (i <= 10)
{
  System.Console.WriteLine(i++);
}
```

### ***`do`...`while`***

The `do`...`while` loop is identical to the `while` loop except that its boolean expression is checked after each iteration rather than before. The `do`...`while` loop works identically in Java and C#. The following Java code illustrates the `do`...`while` loop:

```
int i = 1;

do
{
  System.out.println(i++);
} while (i <= 10);
```

### ***`foreach`* *(C#)***

The `foreach` statement iterates over a collection of objects. Though it is supported in C# and a number of other languages, the `foreach` construct is not supported in Java. The foreach statement assumes the following form:

```
foreach (type identifier in collection)
```

The *`type`* parameter represents the type of objects stored in the collection. The *`identifier`* is an arbitrary name by which each object in the iteration can be referenced. Finally, the *`collection`* parameter indicates the collection across which the `foreach` statement will iterate. The following code demonstrates this useful looping construct:

```
string[] colors = {"red", "green", "blue"};

foreach (string color in colors)
{
  System.Console.WriteLine(color);
}
```

Notice how elegantly the `foreach` loop iterates over a `string` array. Lacking the `foreach` statement, Java requires a `for` loop to accomplish the same task:

```
String[] colors = {"red", "green", "blue"};

for (int i = 0; i < colors.length; i++)
{
  String color = colors[i];
  System.out.println(color);
}
```

### Jump Statements

Jump statements alter a program's flow of execution by jumping to a new location. Java and C# support the `break`, `continue`, and `return` statements. Additionally, C# supports the `goto` statement.

#### ***`break`***

The `break` statement transfers control from within a block to the line immediately following it. The `break` statement can be used to exit `for`, `do`, `while`, and `switch` blocks. For example, the following `while` loop will execute until the `break` statement is encountered (when the `getNumberOfStrikes()` method returns a value of 3 or higher):

```
while (true)
{
  int strikes = getNumberOfStrikes();

  if (strikes >= 3)
    break;
}
```

Upon reaching the `break` statement, the program will exit the `while` block and continue execution with the line immediately following it. This behavior is identical in Java and C#.

Unlike C#, Java supports the concept of a labeled `break`. A labeled `break` allows the programmer to explicitly indicate the location to which the `break` should transfer control. A labeled `break` transfers control to the line immediately following the labeled block. In Java as well as C#, a line can be labeled using this syntax:

```
label : statement;
```

For example, the following `if` statement has been labeled `if_block`:

```
if_block: if (true)
{
  //do something here
}
```

To illustrate a labeled `break`, the following Java code prints the numbers 1 through 5 before breaking out of both loops and the `if` block:

```
if_block: if (true)
{
  for (int i = 1; i < 10; i++)
  {
    for (int j = 1; j < 10; j++)
    {
      if (j > 5)
        break if_block;

      System.out.println(j);
    }
  }
}
```

Remember that a labeled `break` does not jump into the specified block. Rather, execution continues immediately following the labeled block. Though C#'s `break` statement does not support labels, similar functionality can be achieved using the `goto` statement.

***`continue`***

The `continue` statement is similar to `break`. However, rather than exit a loop completely like the `break` statement, the `continue` statement transfers control back to the start of the loop. Once a `continue` statement is encountered, program execution immediately jumps to the next iteration. For example, using the `continue` statement, the following C# code prints all of the odd numbers between 1 and 10:

```
for (int i = 1; i <= 10; i++)
{
  if (i % 2 == 0) //even numbers have a remainder of zero
    continue;

  System.Console.WriteLine(i);
}
```

When the `continue` statement is encountered, control is returned to the top of the loop, the `i` variable is incremented, and the next iteration begins. This behavior is identical in Java and C#.

Like the `break` statement, the Java `continue` statement supports labels. A labeled `continue` statement immediately jumps to the next iteration of the specified loop. For example, when the labeled `continue` is encountered, the following Java code exits the inner loop and immediately begins the next iteration of the outer loop:

```
outer_loop: for (int i = 1; i <= 10; i++)
{
  for (int j = 1; j <= 10; j++)
  {
    if (i % 2 == 0)
      continue outer_loop; //if i is even, skip j iterations
    System.out.println("i=" + i + ", j=" + j);
  }
}
```

Again, C# does not support the labeled `continue` statement. Fortunately, similar functionality can be achieved using the `goto` statement.

***goto* *(C#)***

The `goto` statement transfers program control to a labeled statement. As mentioned previously, though `goto` is a reserved word in Java, it is currently not supported in any manner. In C#, the `goto` statement is typically used wherever you might use a labeled `break` or labeled `continue` in Java. For example, the following code demonstrates how the `goto` statement can be used to break out of a two loops and continue execution immediately following the outer loop:

```
for (int i = 1; i < 10; i++)
{
  for (int j = 1; j < 10; j++)
  {
    System.Console.WriteLine(j);
    if (j >= 5)
      goto exit;
  }
}

exit: ;
```

This code is functionally equivalent to Java's labeled break statement. Since a label can only be assigned to a valid statement, the `exit` label is assigned to an empty statement consisting of only a semi-colon (`;`).

As previously mentioned, the `goto` statement can also be used to duplicate functionality provided by Java's labeled `continue` statement. The following code demonstrates how this can be accomplished:

```
for (int i = 1; i <= 10; i++)
{
  for (int j = 1; j <= 10; j++)
  {
    if (i % 2 == 0)
      goto end_of_outer; //if i is even, skip j iterations

    System.Console.WriteLine("i={0}, j={1}", i, j);
  }

  System.Console.WriteLine("Finished inner loop.");

  end_of_outer: ;
}
```

Notice that the `end_of_outer` label was placed at the end of the outer loop so that execution will skip all statements within the outer loop and proceed with the next iteration. Like the previous example, this label employs an empty statement (i.e., only a semi-colon).

***return***

The `return` statement immediately exits the current method and returns control to the calling method. This statement may or may not return a value (depending on the method signature). The `return` statement behaves identically in Java and C#. The following code demonstrates the `return` statement:

```
public int compare(int value1, int value2)
{
  if (value1 < value2)
    return -1;
  else if (value1 > value2)
    return 1;
  else
    return 0; //values are equal
}
```

### Equality Checking

When it comes to checking for object equality, Java and C# are similar in some ways and very different in others. Both Java and C# support equality checking using the == and `!=` operators as well as the `equals()` and `Equals()` methods inherited from `java.lang.Object` or `System.Object`, respectively. However, given C#'s ability to override == and `!=`, there exists subtle differences in the manner that objects are compared in these two languages.

***== and != Operaters***

When used with objects, the default behavior of the == and `!=` operators check for reference equality. That is, they compare whether or not two references point to the same object in memory. However, when used with value types, these operators compare values rather than references. Listing 3.23 illustrates these concepts in Java. A similar C# program would behave identically.

**Listing 3.23** ***Checking for Reference and Value Equality (Java)***

```
public class Customer
{
  public String firstName;
  public String lastName;

  public Customer(String firstName, String lastName)
  {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public static void main(String[] args)
  {
    //check reference equality
    Customer objectA = new Customer ("Tyler", "Callaway");
    Customer objectB = new Customer ("Tyler", "Callaway");
    Customer objectC = objectA;

    if (objectA == objectB)
      System.out.println("objectA == objectB");
    else
      System.out.println("objectA != objectB");

    if (objectA == objectC)
      System.out.println("objectA == objectC");
    else
      System.out.println("objectA != objectC");
    //check value equality
    int valueA = 10;
    int valueB = 10;
    int valueC = valueA;

    if (valueA == valueB)
      System.out.println("valueA == valueB");
    else
      System.out.println("valueA != valueB");

    if (valueA == valueC)
      System.out.println("valueA == valueC");
    else
      System.out.println("valueA != valueC");
  }
}
```

The output from Listing 3.23 looks like this:

```
objectA != objectB
objectA == objectC
valueA == valueB
valueA == valueC
```

Notice that even though `objectA` and `objectB` were initialized with identical values, the `==` operator does not consider them to be equal. This is because both reference types (`objectA` and `objectB`) point to a different object in memory. On the other hand, notice that the `==` operator determines value types `valueA` and `valueB` to be equal. As you can see, the `==` operator compares references when operating on reference types and values when using value types.

To this point, equality checking in Java and C# has been virtually identical. However, there is one major difference. Unlike Java, in C# it is possible to overload the `==` and `!=` operators such that their behavior can be altered to perform in a more intuitive manner. For example, consider the `Customer` object from Listing 3.23. Perhaps it would be more intuitive to have the `==` and `!=` operators compare `Customer` objects' first and last names to determine if they are equal rather than just comparing references. In C#, this can be accomplished by overriding the `==` and `!=` operators as demonstrated in Listing 3.24.

**Listing 3.24** ***Overriding the == and != Operators (C#)***

```
using System;

public class Customer
{
  public string firstName;
  public string lastName;

  public Customer (string firstName, string lastName)
  {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public static bool operator == (Customer a, Customer b)
  {
    if (a.firstName == b.firstName && a.lastName == b.lastName)
      return true; //true to question "are these equal?"
    else
```

```
      return false;
  }

  public static bool operator != (Customer a, Customer b)
  {
    if (a.firstName == b.firstName && a.lastName == b.lastName)
      return false; //false to question "are these not equal?"
    else
      return true;
  }

  public static void Main()
  {
    Customer objectA = new Customer ("Tyler", "Callaway");
    Customer objectB = new Customer ("Tyler", "Callaway");
    Customer objectC = new Customer ("Madison", "Callaway");

    if (objectA == objectB)
      Console.WriteLine("objectA == objectB");
    else
      Console.WriteLine("objectA != objectB");

    if (objectA == objectC)
      Console.WriteLine("objectA == objectC");
    else
      Console.WriteLine("objectA != objectC");
  }
}
```

The output from Listing 3.24 looks like this:

```
objectA == objectB
objectA != objectC
```

**NOTE**
The == and `!=` operators must be overloaded together. If one is overloaded without the other, the compiler will generate an error.

The most salient difference between equality checking in Java and C# stems from C#'s ability to overload operators. Given this capability, C# code commonly checks for equality using the == and != operators. On the other hand, if a value comparison is desired, Java relies on a method call to perform this type of operation (the Java `equals()` method is discussed in the next section). For example, the following code compares two strings in Java:

```
String name = "Bob";

if (name.equals("Bob"))
  System.out.println("Hello Bob");
```

In contrast, this same comparison in C# typically looks like this:

```
string name = "Bob";

if (name == "Bob")
  System.Console.WriteLine("Hello Bob");
```

The C# designer team felt that equality checking through operator overloading is more natural than Java's method-oriented approach. Regardless, C# also supports an `Equals()` method similar to Java's `equals()` method. Finally, it should be noted that whenever you override the `==` and `!=` operators in C#, you should also override the `Equals()` method to ensure consistent behavior between these different techniques of checking for equality. In other words, the `Equals()` method should always return a value consistent with that returned by the `==` and `!=` operators.

> **NOTE**
> Since Java strings are immutable and are stored in a shared string table in memory, `String` objects having exactly the same value should point to the same `String` object in memory. Therefore, it is technically possible to compare strings in Java by checking for equal references using the `==` and `!=` operators. However, this behavior is not well defined and should not be relied on. Rather, stick with the `equals()` method for comparing `String` objects.

### *`equals()` and `Equals()` Methods*

Java and C# both implement methods that facilitate equality checking between objects. Differing only in case, the Java `equals()` method and the C# `Equals()` method serve the same purpose. That is, these methods indicate whether the object passed in as a parameter is equal to the object on which the method was called. The following Java code demonstrates the `equals()` method:

```
public boolean compare(String a, String b)
{
  if (a == null || b == null)
    return false;
  else
    return a.equals(b);
}
```

To implement a functionally equivalent method in C#, simply replace the `equals()` method with `Equals()` and change the `boolean` data type to `bool`.

The default implementation of the `equals()` and `Equals()` methods (inherited from `java.lang.Object` and `System.Object`, respectively) simply determine whether or not the references point to the same object. For a more intelligent object comparison (using object values rather than just references), these methods can be overridden. In C#, remember that if the `Equals()` method is overridden, the `==` and `!=` operators should be overloaded in a similar manner in order to guarantee consistency between these comparison methods. For example, the following C# code overrides the `Equals()` method within the `Customer` class in order to compare customers by name rather than object reference:

```
public bool Equals(Customer cust)
{
  if (cust == null)
  {
    return false;
  }
  else if (this.firstName == cust.firstName &&
    this.lastName == cust.lastName)
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

In addition to the `Equals()` instance method, C# provides a static `Equals()` method that is inherited from `System.Object`. The static version of `Equals()` is a convenience method that automatically handles null references. However, unlike the code demonstrated in this section, the static `Equals()` method considers two null values to be equal. The following code demonstrates this method:

```
public bool compare(String a, String b)
{
  return System.Object.Equals(a, b);
}
```

### *`ReferenceEquals()` Method (C#)*

In C#, `ReferenceEquals()` is a static method implemented by the `System.Object` class. As you may have guessed, this method indicates whether two references refer to the same object. Like the static version of `Equals()`, this method considers two null values to be equal. It's used like this:

```
public bool compareReferences(String a, String b)
{
  return System.Object.ReferenceEquals(a, b);
}
```

## Preprocessor Directives (C#)

Unlike Java, C# supports several preprocessor directives. Preprocessor directives are basic instructions that are evaluated at compile-time. Though not included in the compiled file, these instructions typically alter the compilation process in some way. For instance, preprocessor directives can be used to perform conditional compilation or generate code-specific errors and warnings. This functionality may be useful when you wish to compile different parts of a program depending upon whether you are debugging the application or compiling the final release.

Though it supports preprocessor-type functionality, C# does not actually include a preprocessor like C and C++. Rather, in C#, the compiler itself evaluates all preprocessor directives. The supported directives include: `#define`, `#undef`, `#if`, `#elif`, `#else`, `#endif`, `#error`, `#warning`, `#region`, `#endregion`, and `#line`.

### *`#define` and `#undef`*

The `#define` directive defines a symbol that can be evaluated by other preprocessor directives. Unlike C and C++, the `#define` directive in C# cannot assign a value to a symbol. Rather, `#define` can only indicate a boolean condition—either a symbol is defined or it is not. `#define` is used like this:

```
#define DEBUG
```

This instruction tells the compiler that the `DEBUG` symbol is defined. This symbol can then be evaluated by the `#if`, `#elif`, `#else`, and `#endif` directives described in the next section. A defined symbol evaluates to *true*.

In addition to defining symbols, it is also possible to "undefine" a symbol using the `#undef` directive like this:

```
#undef DEBUG
```

This instruction tells the compiler that the `DEBUG` symbol is not defined. An undefined symbol evaluates to *false*.

Notice that preprocessor directives do not follow the standard C# coding style. Unlike standard C# code, all preprocessor directives begin with a "`#`" and do not end with a semi-colon. Additionally, the `#define` and `#undef` instructions must appear at the beginning of a C# source file before any code (including `using` and `namespace` statements) and should always occupy a line by themselves.

If a symbol has already been defined, defining it again has no effect. Likewise, `#undef` is ignored if the symbol does not exist.

### *#if, #elif, #else, and #endif*

The `#if`, `#elif`, and `#else` preprocessor directives work similarly to C#'s `if`, `else if`, and `else` statements. These directives are used to evaluate symbols and conditionally compile code. The `#endif` directive indicates the end of a conditional directive. Listing 3.25 demonstrates how these directives can be used.

**Listing 3.25** ***Conditional Compilation with Preprocessor Directives (C#)***

```
#define DEBUG
#define VERBOSE

using System;

public class ConditionalCompile
{
  public static void Main()
  {
#if DEBUG && VERBOSE
    Console.WriteLine("Debug and verbose mode!");
#elif DEBUG || VERBOSE
    Console.WriteLine("Debug or verbose mode!");
#elif DEBUG
    Console.WriteLine("Debug mode!");
#elif VERBOSE
    Console.WriteLine("Verbose mode!");
#else
    Console.WriteLine("Neither debug nor verbose mode!");
#endif
  }
}
```

Since both `DEBUG` and `VERBOSE` are defined, the output from Listing 3.25 looks like this:

```
Debug and verbose mode!
```

From Listing 3.25, notice that the `#if` and `#elif` directives may include logical operators. Table 3.12 documents the operators that are supported by these directives.

**Table 3.12** ***Logical Operators Supported by Conditional Preprocessor Directives (C#)***

| Operator | Description |
|---|---|
| `==` | True if symbol equals given value: `#if DEBUG==true` |
| `!=` | True if symbol does not equal given value: `#if DEBUG!=true` |
| `!` | Negates symbol's boolean value: `#if !DEBUG` |
| `&&` | True if both symbols are true: `#if DEBUG && VERBOSE` |
| `\|\|` | True if one or both symbols are true: `#if DEBUG \|\| VERBOSE` |

In addition to defining symbols within source code, it is also possible to define them at compile time like this:

```
csc ConditionalCompile.cs /define:DEBUG /define:VERBOSE
```

or using a semi-colon or comma-delimited list like this:

```
csc ConditionalCompile.cs /define:DEBUG,VERBOSE
```

## Exception Handling

An *exception* is an error condition that alters the normal flow of an application. When an error occurs, the program is said to have "thrown" an exception. Likewise, when an exception is handled, it is said to have been "caught." The following sections discuss how exceptions are caught, thrown, and re-thrown.

### *Catching Exceptions*

Java and C# handle exceptions using the `try`...`catch`...`finally` construct. This construct consists of three distinct blocks of code designed to keep application logic separate from error-handling code. First, the `try` block contains application code that may throw an exception. Second, the `catch` block contains error-handling code that should be executed in the event an exception is thrown in the `try` block. The `catch` block is optional if a `finally` block is specified. Last, the `finally` block contains cleanup code that is guaranteed to be executed after the `try` and `catch` blocks. The `finally` block is used to ensure that critical cleanup tasks are performed and resources released even in the case of an error. The `finally` block is optional if a `catch` block is present. If both `catch` and `finally` blocks exist, the `catch` block must come before the `finally` block. A typical `try`...`catch`...`finally` construct looks like this:

```
try
{
  //code that may or may not throw an exception
}
catch (Exception e)
{
  //code to handle exception (invoked if exception specified
  //in the catch matches exception thrown in the try block)
}
finally
{
  //cleanup code (runs whether exception occurs or not)
}
```

In the previous code, notice that the `catch` block indicates the type of exception that should be caught. The code in the `catch` block will be executed only if the thrown exception matches the exception specified by the `catch` block. A match will occur if the exceptions are identical or the thrown exception is derived from the exception specified in the `catch` block.

The previous code includes a single `catch` block. However, multiple `catch` blocks can be specified within a single `try`...`catch`...`finally` construct. If more than one is specified, `catch` blocks must be ordered from most to least specific (or lowest to highest in the object hierarchy). In other words, any time a specific exception is derived from a more general exception, the most specific exception must be caught first followed by the more general exception. This restriction is enforced by the compiler. If general exceptions were allowed to precede specific exceptions in the `catch` list, program flow would not be able to reach the specific `catch` blocks. For example, say you want to catch all exceptions but wish to provide special handling for any `NullPointerException`. Catching the `NullPointerException` first ensures that null pointers will be handled by the `catch` block specifically designed for it rather than the more general `catch` block. The following code demonstrates this principle:

```
try
{
  //code that may throw an exception
}
catch (NullPointerException e)
{
  //code to handle null pointer exception
}
catch (Exception e)
{
  //code to handle all other exceptions
}
finally
{
  //cleanup code
}
```

Code within the `try` block is scoped similarly to code within an `if`, `for`, or `while` construct. That is, variables declared within the `try` block are not visible outside of it (not even by its corresponding `catch` or `finally` blocks). Therefore, if you wish to access a variable outside of the `try` block, it must be declared prior to the start of the `try` block. For instance, database and network connections are typically declared before the `try` block so that they can be closed in the `finally` block.

Java and C# exceptions extend from the root classes `java.lang.Throwable` and `System.Exception`, respectively. Java error conditions are categorized into two types: errors and exceptions. In Java, errors are defined as unexpected conditions that occur within the runtime environment. Error conditions include “Out of Memory” and “Stack Overflow” errors. Java errors extend from `java.lang.Error` (which extends from `java.lang.Throwable`). On the other hand, Java exceptions refer to unexpected conditions that occur within an application. Exception conditions include “Index Out of Bounds” or “Null Pointer” exceptions. Java exceptions extend from `java.lang.Exception` (which extends from `java.lang.Throwable`).

C# makes no distinction between errors and exceptions. All unexpected conditions are classified as exceptions. However, similar to Java, C# exceptions can be categorized into two types of exceptions: system and application. Like Java errors, system exceptions are generated by the runtime environment. System exceptions are derived from `System.SystemException`. And similar to Java exceptions, application exceptions occur within an application. Application exceptions are derived from `System.ApplicationException`.

We have seen that both Java and C# support two distinct types of exceptions. In order to handle all error conditions, catch `java.lang.Throwable` in Java like this:

```
try
{
  //Java code that may produce an error
}
catch (java.lang.Throwable e)
{
  //handle all Java error conditions
}
finally
{
  //cleanup code
}
```

And catch `System.Exception` in C# like this:

```
try
{
  //C# code that may produce an error
}
catch (System.Exception e)
{
  //handle all C# error conditions
}
finally
{
  //cleanup code
}
```

Even simpler, if you don't need a reference to the exception, C# can catch all exceptions (including both system and application exceptions) using a generic `catch` statement that does not specify an exception type. This generic version of `catch` must always come last in the `catch` list. This feature is not supported by Java. To illustrate, the following C# code catches all exceptions:

```
try
{
  //C# code that may produce an error
}
catch
{
  //handle all C# error conditions
}
finally
{
  //cleanup code
}
```

**NOTE**
Though you can handle all error conditions in Java by catching `java.lang.Throwable`, it is usually preferable to just catch `java.lang.Exception`. This is due to the fact that, while programs can typically recover from exceptions, Java errors are usually unrecoverable. Similarly, in C#, the `System.ApplicationException`

is usually caught instead of `System.Exception` since an application usually cannot recover from a `System.SystemException`.

As you will see from the following examples, handling exceptions in Java and C# is very similar. First, let's take a look at a typical example of exception handling in Java:

```
public int divide(int dividend, int divisor)
{
  try
  {
    return dividend / divisor;
  }
  catch (ArithmeticException ae)
  {
    System.out.println("Error in divide() method: " + ae);
    return -1; //-1 indicates an error
  }
}
```

In the previous code, if the divisor is zero an `ArithmeticException` is thrown indicating that a "divide by zero" exception occurred. This exception is handled by the `catch` block. Here is an equivalent example in C#:

```
public int divide(int dividend, int divisor)
{
  try
  {
    return dividend / divisor;
  }
  catch (DivideByZeroException dbze)
  {
    Console.WriteLine("Error in divide() method: {0}", dbze);
    return -1;
  }
}
```

Notice that, other than the name of the exception and the manner that text is output to the screen, the C# and Java examples are identical.

If an exception is not caught within the method in which it is thrown, the exception is thrown to its caller and continues up the call chain until it is caught. If the exception is not caught by the application, eventually the Java or .NET runtime environment will handle it. However, since uncaught exceptions cause the runtime to present a cryptic error message and exit, it is a good idea to always catch any application-level exceptions that you can anticipate. To illustrate, Listing 3.26 demonstrates how an exception can be thrown from a C# method to its caller. Java exceptions are thrown up the call chain in a similar manner.

**Listing 3.26** ***Throwing an Exception from Method to Caller (C#)***

```
using System;

public class ExceptionTest
{
  public static void Main()
  {
    try
    {
      int a = 10;
      int b = 0;
      int result = divide(a, b);

      Console.WriteLine("{0} / {1} = {2}", a, b, result);
    }
    catch (DivideByZeroException dbze)
    {
      Console.WriteLine("Error: {0}", dbze.Message);
    }
  }

  public static int divide(int dividend, int divisor)
  {
    return dividend / divisor;
  }
}
```

The output from Listing 3.26 looks like this:

```
Error: Attempted to divide by zero.
```

***Re-Throwing Exceptions***

There may be times when you wish to catch an exception, perform some processing, and then throw the exception again (as if you never caught it in the first place). This is possible using the `throw` statement. Any exception that can be caught can also be thrown. For example, Listing 3.27 demonstrates in C# how an exception can be caught and then thrown. Exceptions can be thrown in Java in a similar manner.

**Listing 3.27** ***Explicitly Throwing an Exception from Method to Caller (C#)***

```
using System;

public class ExceptionTest
{
  public static void Main()
  {
    try
    {
      int a = 10;
      int b = 0;

      int result = divide(a, b);

      Console.WriteLine("{0} / {1} = {2}", a, b, result);
    }
    catch (Exception e) //catch all exceptions
    {
      Console.WriteLine("Error: {0}", e.Message);
    }
  }

  public static int divide(int dividend, int divisor)
  {
    try
    {
      return dividend / divisor;
    }
    catch (DivideByZeroException dbze) //catch divide by zero
    {
      Console.WriteLine("Divide by zero error in divide().");

      throw dbze;
    }
  }
}
```

The output from Listing 3.27 looks like this:

```
Divide by zero error in divide().
Error: Attempted to divide by zero.
```

In addition to re-throwing an exception by name, C# supports a generic version of `throw` that automatically re-throws the exception that was caught without explicitly specifying it. Since the generic `throw` statement simply re-throws the current exception, it can only be used within a `catch` block. For instance, the following code demonstrates how a caught exception can be re-thrown using the generic `throw` statement:

```
public int divide(int dividend, int divisor)
{
  try
  {
    return dividend / divisor;
  }
  catch (DivideByZeroException dbze) //catch divide by zero
  {
    Console.WriteLine("Divide by zero error in divide().");

    throw; //throw the exception that was caught
  }
}
```

***Throwing Exceptions***
Listing 3.27 presented how an exception can be caught and then re-thrown. However, you need not catch an exception in order to throw it. Rather, an exception can be instantiated and then thrown as shown in Listing 3.28. An comparable Java example would behave identically.

**Listing 3.28** ***Creating and Throwing an Exception (C#)***

```
using System;

public class ExceptionTest
{
  public static void Main()
  {
    try
    {
      int a = 10;
      int b = 0;

      int result = divide(a, b);

      Console.WriteLine("{0} / {1} = {2}", a, b, result);
    }
    catch (Exception e) //catch all exceptions
    {
      Console.WriteLine("Error: {0}", e.Message);
    }
  }

  public static int divide(int dividend, int divisor)
  {
    if (divisor == 0)
    {
      throw new DivideByZeroException("Divide by zero!");
    }

    return dividend / divisor;
  }
}
```

Notice that the `divide()` method in Listing 3.28 instantiates a new `DivideByZeroException` (assigning it a new a description) and throws the new exception to the calling method. The new exception is then caught by the caller. The output from Listing 3.28 looks like this:

```
Error: Divide by zero!
```

***throws* Statement (Java)**

Unlike C#, Java requires methods to either catch or declare the exceptions that they may throw. We have already discussed how exceptions are caught using the `try`...`catch`...`finally` construct. If a method doesn't catch an exception that it might throw, it must declare the exception using the `throws` keyword. The `throws` clause is part of the method definition and consists of a comma-delimited list of exceptions that may be thrown like this:

```
public String readFile(String name) throws IOException
```

If an exception declared in the `throws` clause is not handled by the client, an error is generated at compile time. For example, Listing 3.29 does not compile because the calling method does not handle the `Exception` that is declared thrown by the `divide()` method.

**Listing 3.29** ***Example That Does Not Compile Because of Checked Exception (Java)***

```
public class CheckedException
{
  public static void main(String[] args)
  {
    int result = divide(10, 5); //must catch Exception to compile
  }

  public static int divide(int dvnd, int dvsr) throws Exception
  {
    return dvnd / dvsr;
  }
}
```

The compiler generates this error when you attempt to compile Listing 3.29:

```
unreported exception java.lang.Exception; must be caught or declared to
be thrown
```

To remedy this problem, alter the `main()` method in Listing 3.29 to catch the `Exception` like this:

```
public static void main(String[] args)
```

```
{
  try
  {
    int result = divide(10, 5);
   }
   catch (Exception e)
   {
     System.out.println("Error: " + e);
   }
}
```

You might wonder why the `throws` keyword is necessary. Java designers reasoned that the exceptions a method might throw are actually part of its public interface. Clients that call this method must be aware of any exceptions that it might throw so that they can properly handle them. The `throws` keyword also encourages disciplined programming by forcing clients to handle any exceptions that may occur.

> **NOTE**
> Though including runtime exceptions in a method's `throws` clause is discouraged, it is a good practice to declare any runtime exceptions that you anticipate in a `@throws` documentation comment as described in the next section. Though not enforced by the compiler, this practice at least makes the client aware of the anticipated runtime exception.

Though Java requires a method to catch or declare the exceptions that it may throw, there is an exception to this rule. Java does not require a certain type of exception, known as a runtime exception, to be caught or declared thrown. In Java, *runtime exceptions* are errors and exceptions thrown by the runtime environment that may occur frequently throughout an application. These errors and exceptions include common problems like "out of memory", "disk full", "divide by zero", "index out of bounds", and "null pointer" conditions. All Java runtime exceptions are derived from `java.lang.Error` or `java.lang.RuntimeException` (which is derived from `java.lang.Exception`). Therefore, even if a runtime exception is declared thrown, the Java compiler does not force the client to handle the exception. As opposed to runtime exceptions, exceptions that must be caught or declared thrown are known as *checked exceptions*. Checked exceptions include all Java exceptions that do not extend from `java.lang.Error` or `java.lang.RuntimeException`.

> **NOTE**
> Though C# can not force clients to handle declared exceptions (since it doesn't support the `throws` keyword), it is possible to formally document exceptions that a C# method may throw using .NET's XML code documentation features described in the next section. Similarly, Java facilitates formal documentation of exceptions using standard Java documentation techniques described in the next section.

## Comments

Java and C# support two distinct types of comments: code comments and documentation comments. Code comments are used to convey the purpose and function of cryptic code. These comments are targeted at programmers as they review the source code. On the other hand, documentation comments are specially formatted to allow another tool to read the source code and generate external documentation in HTML or XML format. Both types of comments are discussed in this section.

***Code Comments***
*Code comments* are specially marked portions of text within a source file that are ignored by the compiler. Often times, source code can be very difficult to read and understand (even by the developer that wrote the code in the first place). Comments provide a simple and convenient method of documenting the purpose and function of code directly within the program source. Well-commented code simplifies the task of enhancing and maintaining an application.

Java and C# both support C-style code comments. There are basically two types of comments: single-line and multi-line. Single-line comments instruct the compiler to ignore all text from the "`//`" marker to the end of the line. A single-line comment can occupy its own line or be appended to the end of an existing line as illustrated here:

```
public int add(int a, int b)
{
  //calculate the sum of the parameters a and b
  int sum = a + b;

  return sum; //return result
}
```

Multi-line comments instruct the compiler to ignore all text between the "`/*`" and "`*/`" markers. Despite their name, multi-line comments are not required to span multiple lines as demonstrated here:

```
public int add(int a, int b)
{
  /* calculate the sum of the parameters a and b and provide
     comprehensive comments regarding this simple calculation */
  int sum = a + b;

  return sum; /* return result */
}
```

***Documentation Comments***
Beyond basic C-style code comments, Java and C# support innovative methods of generating external documentation from source code. Sometimes called inline documentation, documentation comments are commonly used to document a class's or library's public API. This type of documentation can be called *inline* because the documentation text itself is included inline with the code. By embedding properly formatted documentation tags, it is possible to generate HTML or XML documentation files directly from Java and C# source code.

Though their purpose is the same, Java and C# inline documentation differs significantly. For instance, Java documentation blocks are delimited by the "`/**`" and "`*/`" markers and documentation elements are specified using the "`@`*`element`*" format. On the other hand, each line of C# documentation text begins with "`///`" and documentation elements are specified as XML tags. For example, the following code illustrates documentation comments in Java:

```
/**
 * This method adds two integers and returns the result.
 *
 * @param a First integer to be added
 * @param b Second integer to be added
 * @return Sum of two integers
 */
public int add(int a, int b)
```

```
{
  return a + b;
}
```

Likewise, documentation comments in C# looks like this:

```
/// <summary>
/// This method adds two integers and returns the result.
/// </summary>
/// <param name="a">First integer to be added</param>
/// <param name="b">Second integer to be added</param>
/// <returns>Sum of two integers</returns>
public int add(int a, int b)
{
  return a + b;
}
```

Personally, I find the Java-style documentation easier to read in the source code since it is not "littered" with XML tags. However, the C# designers' decision to use well-formed XML for inline documentation offers its own advantages. For instance, since the `name` attribute of the `<param>` element is well-defined and easily parsed, the C# compiler can verify that the name of all `<param>` tags matches an existing parameter. If not, a warning is issued. Java does not offer this type of documentation validation.[1] Inline documentation is completely ignored by the Java compiler. In Java, it is the programmer's responsibility to ensure the accuracy of all inline documentation.

### Javadoc Comments

Java documentation, commonly referred to as *Javadoc comments*, provides a flexible and convenient mechanism for producing documentation from source code. For example, after properly documenting a class and all of its members, a program called *javadoc* (included with the Java Development Kit) can be used to read the source file and generate corresponding HTML documentation files that fully describe the class. These HTML documentation files are far more convenient to browse than searching through source code to determine the purpose of a class and the properties and methods that it exposes.

Javadoc comments can be described in two parts. The first part of a Javadoc comment is the free-form text description of the class or member. This description may contain HTML markup in order to specify custom formatting. For instance, HTML tags `<p>`, `<br>`, and `<code>` are just a few of the many HTML tags that are commonly used within Javadoc comments. The first sentence of the description portion of a Javadoc comment should contain a concise summary of the documented item. This is important because the *javadoc* utility uses the first sentence of a documented item when building summary and index pages. The rest of the description should provide additional detail as needed.

The second part of a Javadoc comment consists of a collection of documentation elements designated with "`@element`" tags. The *javadoc* program recognizes many different documentation elements. These elements provide the detailed information necessary to properly document a Java class or member item (e.g., a public field or method). Table 3.13 details the various documentation elements supported by the *javadoc* program (listed in the order they typically appear) and Listing 3.30 presents a fully documented Java class.

[1] Sun has developed a program called DocCheck that performs documentation validation. Unfortunately, DocCheck is a separately available Javadoc plug-in (known as a *doclet*) that is not included in the standard Java distribution.

**Table 3.13** ***Javadoc Documentation Elements (Java)***

| Element | Description |
|---|---|
| `@author` | Specifies the author of a package or class. Multiple `@author` tags are supported. Only viewable in the source code, author information is not included in the HTML documentation pages generated by `javadoc`. |
| `@version` | Indicates a version number for the package or class being documented. A typical version format looks like this: `@version 1.01, 01/26/02` |
| `@param` | Documents a single method parameter. This element should be followed by the name of the parameter followed by its description like this: `@param filename Name of file to open` |
| `@return` | Indicates the method's return value like this: `@return <code>true</code> if successful` |
| `@throws` | Specifies a checked (declared in `throws` clause) or unchecked exception thrown by a method. If the method throws multiple exceptions, use multiple `@throws` elements. This element is used like this: `@throws java.io.IOException` |
| `@see` | Indicates a related class or package that may be of interest. Multiple `@see` tags are supported. This element is followed by a fully qualified class or package name like this: `@see java.util.HashMap` |
| `@since` | Specifies the version that first introduced this item. Element is used like this: `@since 1.01` |
| `@deprecated` | Indicates that the class or method has been deprecated. Typically followed by information regarding when the item was deprecated and what replaced it (if anything): `@deprecated As of 1.01, replaced by getCustomer()` |

**Listing 3.30** ***Class Documented with Javadoc Comments (Java)***

```
/**
 * <code>IntegerMath</code> is a utility class that implements methods
 * that multiply and divide integers.
 *
 * @author Dustin R. Callaway
 * @version 1.0, 11/29/02
 */
public class IntegerMath
{
  /**
   * Multiplies two integers.
   *
   * @param a First integer to multiply
   * @param b Second integer to multiply
   * @return Result of multiplication operation
   */
  public int multiply(int a, int b)
  {
    return a * b;
  }


  /**
   * Divides two integers.
   *
   * @param a Integer dividend
   * @param b Integer divisor
   * @return Result of division operation
   * @throws java.lang.Exception
   */
  public int divide(int a, int b) throws Exception
  {
    if (b == 0)
```

```
      throw new Exception("Divisor cannot be zero!");

    return a / b;
  }
}
```

To produce Javadoc documentation, you simply run the *javadoc* utility against the source file like this:

```
javadoc IntegerMath.java
```

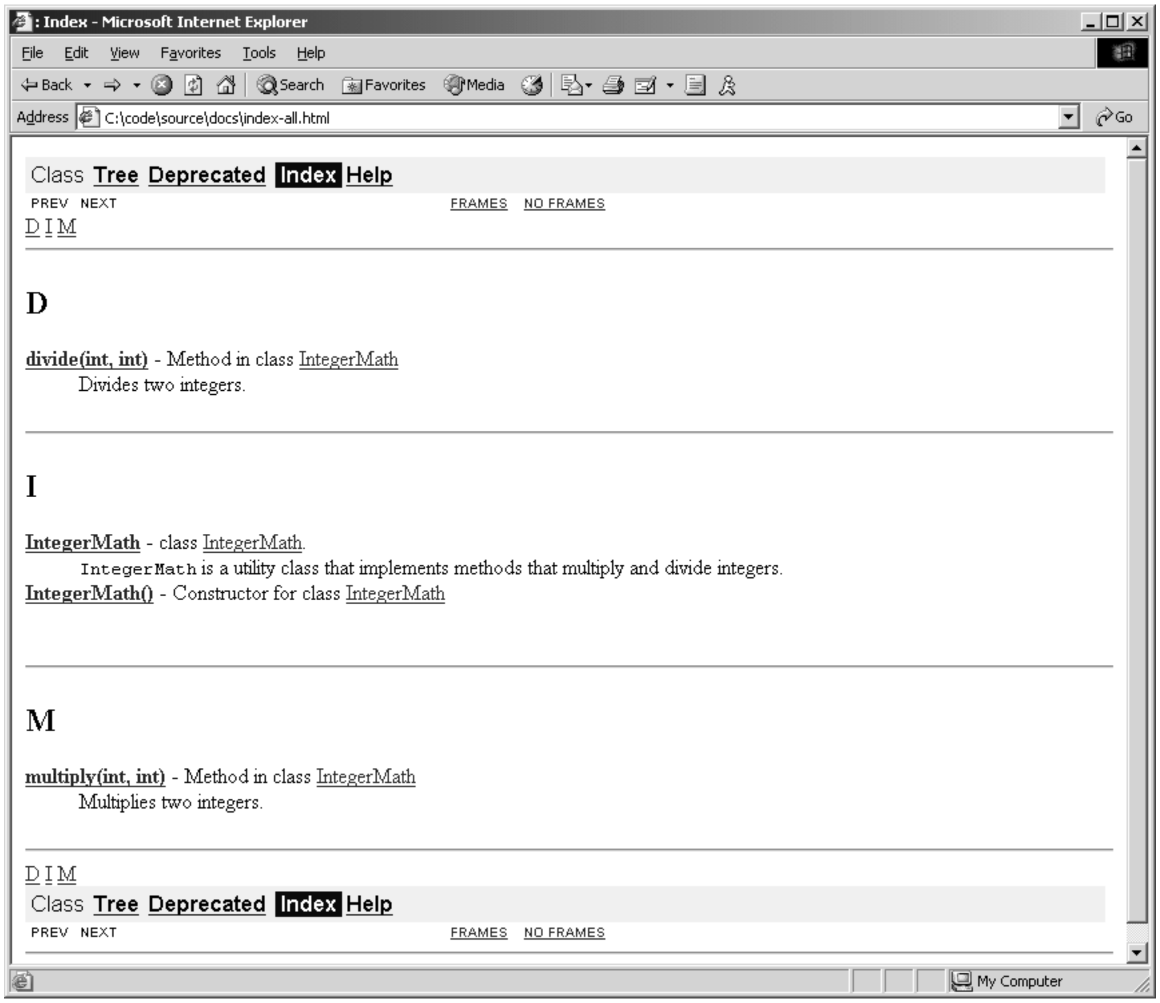
Using the `-d` switch, you can specify where the documentation files should be created:

```
javadoc -d c:\docs IntegerMath.java
```

Executing the *javadoc* utility in this manner produces the following files:

- `allclasses-frame.html`
- `index.html`
- `packages.html`
- `IntegerMath.html`
- `serialized-form.html`
- `package-list`
- `help-doc.html`
- `stylesheet.css`

As you can see, the *javadoc* utility performs a lot of work on your behalf in generating documentation. It even creates a formatted index (see Figure 3.1) and comprehensive class documentation (see Figure 3.2).


: Index - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\code\source\docs\index-all.html
Go
Class Tree Deprecated Index Help
PREV NEXT FRAMES NO FRAMES
D I M
My Computer

D

divide(int, int) - Method in class IntegerMath
Divides two integers.

I

IntegerMath - class IntegerMath.
`IntegerMath` is a utility class that implements methods that multiply and divide integers.
IntegerMath() - Constructor for class IntegerMath

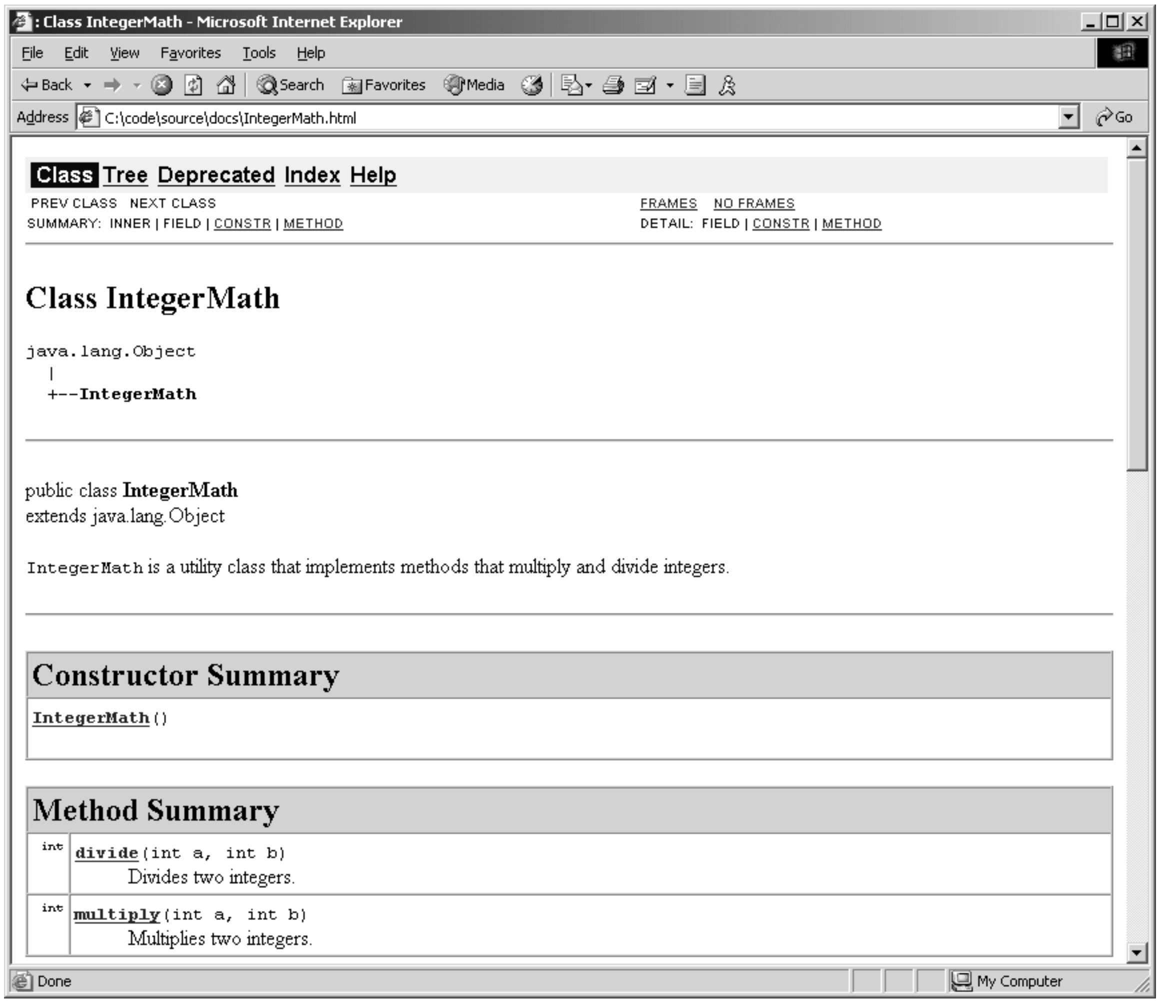
M

multiply(int, int) - Method in class IntegerMath
Multiplies two integers.

D I M

Class Tree Deprecated Index Help
PREV NEXT FRAMES NO FRAMES

**Figure 3.1** ***Index created by the javadoc utility***


: Class IntegerMath - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\code\source\docs\IntegerMath.html
Go
Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
Done
My Computer

Class IntegerMath

```
java.lang.Object
  |
  +--IntegerMath
```

public class **IntegerMath**
extends java.lang.Object

`IntegerMath` is a utility class that implements methods that multiply and divide integers.

Constructor Summary

`IntegerMath()`

Method Summary

| | |
|---|---|
| `int` | `divide(int a, int b)` Divides two integers. |
| `int` | `multiply(int a, int b)` Multiplies two integers. |

**Figure 3.2** ***Class documentation created by the javadoc utility***

**C# XML Documentation**

Like Javadoc comments, inline C# XML documentation provides a means of creating external documentation directly from source code. However, unlike Javadocs, C# utilizes well-formed XML to describe its documentation elements. Additionally, C# inline documentation produces external documentation in XML rather than HTML. In order to create HTML documentation, an XSL stylesheet must be applied to the XML output. Table 3.14 details the top-level documentation elements supported by C#. Top-level means that these elements are not embedded within another element.

**Table 3.14** ***Top-Level Documentation Elements (C#)***

| **Element** | **Attributes** | **Description** |
|---|---|---|
| `<summary>` | | Brief summary of the documented item. |
| `<remarks>` | | Detailed description of the documented item. |
| `<param>` | `name` | Describes a single method parameter. The `name` attribute is verified by the compiler (a warning is issued if the method does not define a matching parameter). |
| `<returns>` | | Describes a method's return value. |
| `<exception>` | `cref` | Documents the exceptions that a class can throw. The `cref` attribute must contain a fully qualified exception (e.g., `System.Exception`) that can be resolved from the current compilation environment. The compiler verifies that the |

|  |  |  |
|---|---|---|
|  |  | specified exception exists. |
| `<example>` |  | Presents an example of how to use the documented item. |
| `<value>` |  | Describes the value represented by a public field or property. |
| `<include>` | `file,path` | Includes documentation from an external file within the generated XML documentation. The `file` attribute indicates the name of the documentation file and the `path` attribute uses XPATH syntax to describe the portion of the documentation file to include. The compiler verifies that the specified file exists. Unlike most attributes, the value of the `file` and `path` attributes should be included within single quotes (e.g., `file='value'`). This element is used as an "empty tag" with no body text (i.e., `<include file="" path=""/>`). |
| `<permission>` | `cref` | Describes the permissions required to access the documented item. The `cref` attribute indicates a security class that can be accessed from the current compilation environment. The compiler verifies that this class exists. |

Table 3.15 presents the embedded documentation elements (i.e., these elements are embedded within another element) and Listing 3.31 presents a fully documented C# class.

**Table 3.15** ***Embedded Documentation Elements (C#)***

| **Element** | **Attributes** | **Description** |
|---|---|---|
| `<paramref>` | `name` | Use when referencing a parameter from within a top-level element (e.g., `<summary>` or `<remarks>`). The `name` attribute must match the name of an existing parameter. This is verified by the compiler. The body of this element contains the text that will be displayed. |
| `<see>` | `cref` | Provides a link to an item that is visible from the current compilation environment. The `cref` attribute references a class, method, field, or property. The compiler verifies that the `cref` reference is valid. This element is used as an "empty tag" with no body text (i.e., `<see cref=""/>`). |
| `<seealso>` | `cref` | Defines an item to display in the "See Also" section of the documentation. The `cref` attribute references a class, method, field, or property. The compiler verifies that the `cref` reference is valid. This element is used as an "empty tag" with no body text (i.e., `<seealso cref=""/>`). |
| `<c>` |  | Marks text within a line as code like this: `<c>int x = 0;</c>` |
| `<code>` |  | Marks text that spans multiple lines as code. |
| `<para>` |  | Formats text as a paragraph like this: `<para>Paragraph here.</para>` |
| `<list>` | `type` | Formats text as a bulleted or numbered list or a table. Valid values for the `type` attribute are `bullet`, `number`, or `table`. This element supports embedded elements `<listheader>` and `<item>`. Each of these embedded elements support the `<term>` and `<description>` embedded elements. |

**Listing 3.30** ***Class Documented with C# XML Documentation (C#)***

```
using System;

/// <summary>
/// <c>IntegerMath</c> is a utility class that implements methods that
/// multiply and divide integers.
/// </summary>
/// <remarks>
/// Authored by Dustin R. Callaway<br/>
/// Version 1.0, 11/29/02
/// </remarks>
public class IntegerMath
{
  /// <summary>
  /// Multiplies two integers.
  /// </summary>
  /// <param name="a">First integer to multiply</param>
  /// <param name="b">Second integer to multiply</param>
  /// <returns>Result of multiplication operation</returns>
  public int multiply(int a, int b)
  {
    return a * b;
  }

  /// <summary>
  /// Divides two integers.
  /// </summary>
  /// <param name="a">Integer dividend</param>
  /// <param name="b">Integer divisor</param>
  /// <returns>Result of division operation</returns>
  /// <exception cref="System.DivideByZeroException">
  /// Thrown if divisor is zero.
  /// </exception>
  public int divide(int a, int b)
  {
    if (b == 0)
      throw new DivideByZeroException("Divisor can't be zero!");

    return a / b;
  }
}
```

To generate XML documentation while compiling Listing 3.31, add the `/doc` switch to the compilation instruction like this:

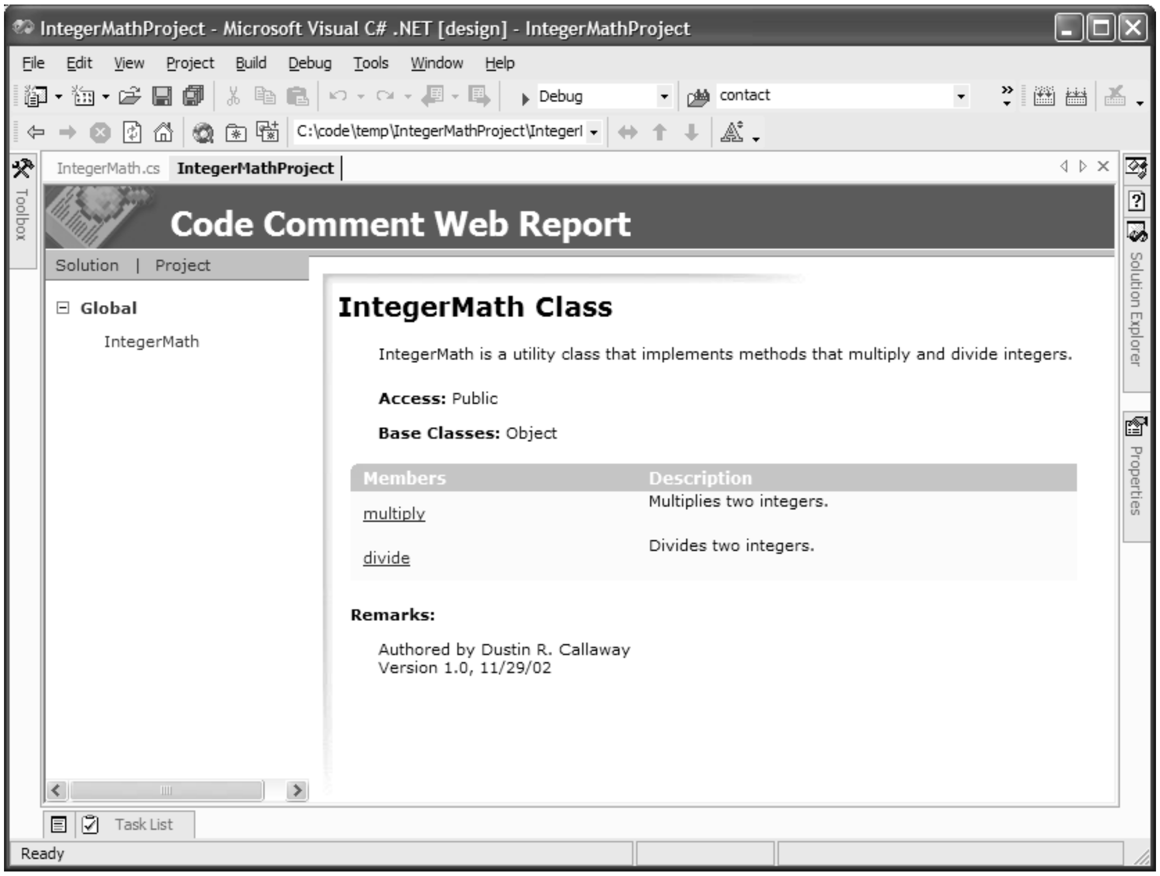
```
csc /t:library /doc:IntegerMath.xml IntegerMath.cs
```

This instruction creates an XML documentation file named `IntegerMath.xml`. The `/t:library` switch is required since the `IntegerMath` class does not include a `Main()` method. The generated XML looks like this:

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>IntegerMath</name>
  </assembly>
  <members>
    <member name="T:IntegerMath">
      <summary>
      <c>IntegerMath</c> is a utility class that implements
      methods that multiply and divide integers.
      </summary>
      <remarks>
      Authored by Dustin R. Callaway<br/>
      Version 1.0, 11/29/02
      </remarks>
    </member>
    <member
      name="M:IntegerMath.multiply(System.Int32,System.Int32)">
      <summary>
      Multiplies two integers.
      </summary>
      <param name="a">First integer to multiply</param>
      <param name="b">Second integer to multiply</param>
      <returns>Result of multiplication operation</returns>
    </member>
    <member
      name="M:IntegerMath.divide(System.Int32,System.Int32)">
      <summary>
      Divides two integers.
      </summary>
      <param name="a">Integer dividend</param>
      <param name="b">Integer divisor</param>
      <returns>Result of division operation</returns>
      <exception cref="T:System.DivideByZeroException">
      Thrown if divisor is zero.
      </exception>
    </member>
  </members>
</doc>
```

Notice that the C# compiler automatically generates a fully qualified "canonical name" for each documented member. Each canonical name begins with a `T`, `M`, or `P` indicating whether the member is a type, method, or property, respectively.

Of course, C#'s XML documentation is far more difficult to read than the HTML output generated by Java's *javadoc* utility. However, the .NET designers never intended programmers to read the XML documentation directly. Rather, using an XSL stylesheet, the XML file can be transformed into an HTML document or any other viewing format. In fact, Microsoft Visual Studio.NET provides a feature that builds "Comment Web Pages" in HTML from the XML documentation produced by the C# compiler (select **Build Comment Web Pages…** from the **Tools** menu). Figures 3.3 and 3.4 display the class and method documentation pages, respectively, generated by Visual Studio.


IntegerMathProject - Microsoft Visual C# .NET [design] - IntegerMathProject
File Edit View Project Build Debug Tools Window Help
Debug
contact
C:\code\temp\IntegerMathProject\IntegerI
IntegerMath.cs
IntegerMathProject
Toolbox
Solution Explorer
Properties
Task List
Ready

Code Comment Web Report

Solution | Project

Global

IntegerMath

IntegerMath Class

IntegerMath is a utility class that implements methods that multiply and divide integers.

**Access:** Public

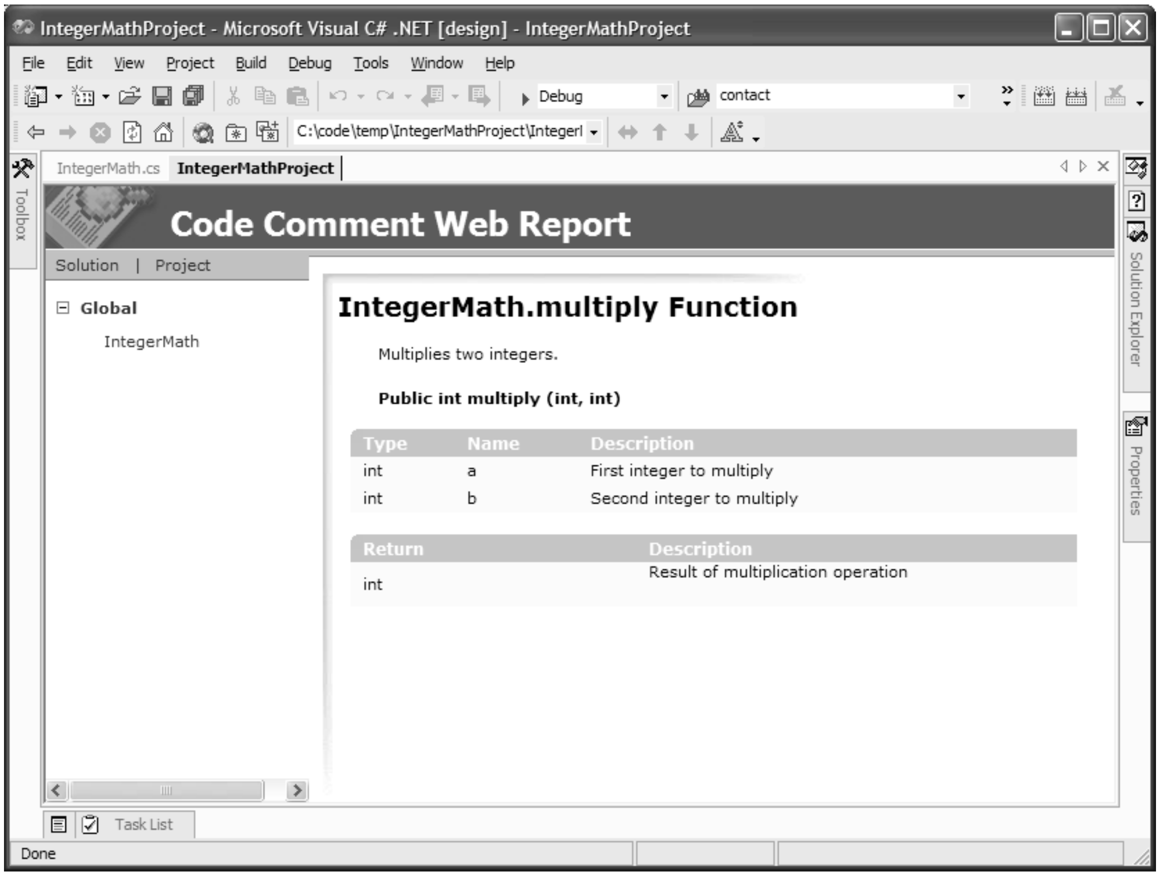
**Base Classes:** Object

| Members | Description |
|---|---|
| multiply | Multiplies two integers. |
| divide | Divides two integers. |

**Remarks:**

Authored by Dustin R. Callaway
Version 1.0, 11/29/02

**Figure 3.3** ***Class documentation page created by Visual Studio***


IntegerMathProject - Microsoft Visual C# .NET [design] - IntegerMathProject
File Edit View Project Build Debug Tools Window Help
Debug
contact
C:\code\temp\IntegerMathProject\IntegerI
IntegerMath.cs
IntegerMathProject
Toolbox
Solution Explorer
Properties
Solution | Project
Global
IntegerMath
Task List
Done

Code Comment Web Report

**IntegerMath.multiply Function**

Multiplies two integers.

**Public int multiply (int, int)**

| Type | Name | Description |
|---|---|---|
| int | a | First integer to multiply |
| int | b | Second integer to multiply |

| Return | Description |
|---|---|
| int | Result of multiplication operation |

**Figure 3.4** ***Method documentation page created by Visual Studio***

## Assertions

An *assertion* is a statement that is expected to be true. If an assertion statement does not evaluate to true, an exception is thrown. Assertions are a common way of formally adding “sanity checks” to code in order to ensure that the programmer’s assumptions are correct. Java supports assertions as an inherent part of the language. On the other hand, C# supports assertions through a system class.

### *Java*

Introduced in version 1.4, Java uses the `assert` keyword to indicate an assertion like this:

```
assert age >= 0;
```

If the assertion is not true, a `java.lang.AssertionError` is thrown.

The `assert` keyword is often used to ensure that return values or method parameters fall within an expected range. The following example uses an assertion to verify that a parameter’s value is between 0 and 100:

```
public void printPercentage(int percent)
{
  assert percent >= 0 && percent <= 100;
  System.out.println(percent + "%");
}
```

If the assertion fails, an `AssertionError` is thrown like this:

```
java.lang.AssertionError
 at MyClass.printPercentage(MyClass.java:15)
```

In addition to the boolean expression, an assertion may also include a second expression that provides text describing why the assertion failed. The descriptive text portion of an `assert` statement is separated from the boolean expression by a colon like this:

```
assert percent >= 0 && percent <= 100 : "Percentage must be " +
  "be between 0 and 100";
```

In this case, the following error is generated if the assertion fails:

```
java.lang.AssertionError: Percentage must be be between 0 and 100
 at MyClass.printPercentage(MyClass.java:15)
```

There is one caveat to mention in regard to using assertions. In order to maximize backward compatibility, the Java compiler and interpreter do not support assertions by default. Rather, options must be specified in order to enable assertions support. For instance, to compile a source file that uses the `assert` keyword, the `-source 1.4` option must be specified like this:

```
javac -source 1.4 MyClass.java
```

The `-source 1.4` option tells the compiler that the source file was written using Java version 1.4 and, therefore, the `assert` keyword should be supported. Again, by default, the compiler does not recognize the `assert` keyword.

Like the compiler, the Java interpreter must be notified when assertion support is desired. Otherwise, all `assert` statements will be ignored. To enable assertion support, the `-enableassertions` (or `-ea`, for short) option must be specified like this:

```
java -enableassertions MyClass
```

When assertions are enabled, an exception is thrown whenever an assertion fails.

***C#***

C# implements assertions differently than Java in several ways. First, C# uses a system class rather than a language keyword to declare an assertion. In C#, assertions are declared by calling one of the following `Assert()` methods defined by the `System.Diagnostics.Debug` class:

```
Assert(bool condition);
Assert(bool condition, string message);
Assert(bool condition, string message, string detailMessage);
```

To illustrate, the following code demonstrates three assertions using each of the `Assert()` method signatures:

```
Debug.Assert(age > 0);
Debug.Assert(age > 0, "Invalid Age");
Debug.Assert(age > 0, "Invalid Age", "Must be greater than 0");
```

Second, C# assertions do not throw exceptions. Rather, they just notify debug listener classes that have been registered to “listen” for debug events (including failed assertion events). .NET refers to these debug listeners as *trace listeners* since they all must extend the `System.Diagnostics.TraceListener` abstract class. The `Debug` object exposes a property named `Listeners` that represents a collection of trace listeners that are called whenever debug events occur. The `DefaultTraceListener` class is automatically added to the `Debug` object's `Listeners` collection. This listener simply displays a dialog box to the user whenever an assertion fails. Listing 3.31 demonstrates how the default listener can be "unregistered" and your own custom listener registered to listen for failed assertions.

**Listing 3.31** ***Register a new debug event listener (C#)***

```
public class AssertionTest
{
  public void printPercentage(int percentage)
  {
    Debug.Assert(percentage >=0 && percentage <= 100,
      "Invalid Percentage", "Must be between 0 and 100");

    Console.WriteLine(percentage + "%");
  }


  public static void Main()
  {
    //Create custom debug listener that writes to standard out
    TextWriterTraceListener twtc =
      new TextWriterTraceListener(Console.Out);

    Debug.Listeners.Clear(); //Unregister default debug listener
    Debug.Listeners.Add(twtc); //Register custom debug listener

    AssertionTest at = new AssertionTest();

    at.printPercentage(50);
    at.printPercentage(100);
    at.printPercentage(150);
  }
}
```

The console output produced by Listing 3.31 looks like this:

```
50%
100%
Fail: Invalid Percentage Must be between 0 and 100
150%
```

Note how the assertion warning was redirected to standard out (as instructed in Listing 3.31). Also notice that `150%` was printed to standard out even though this value failed the assertion condition. This behavior is one of the primary differences between assertions in Java and C#. While failed Java assertions throw exceptions that alter a program's flow of control, C# assertions simply notify listeners and continue normal execution.

## *Summary*

This chapter introduced the Java and C# languages and presented many of these languages' most fundamental programming constructs. Each construct was compared and contrasted. Strangely enough, Java and C# seem to be as different as they are similar. Though these two languages appear practically identical at first glance, further investigation reveals some striking differences that give each language its unique flavor. Developed first, Java introduced many new innovations and borrowed many others from previous programming languages. With the benefit of hindsight, C# adopted many of Java's most popular features and added numerous innovations of its own. Though key differences exist, these languages share enough common heritage such that familiarity with in one can greatly benefit learning of the other.

Now that we have reviewed the fundamentals of each language, in the next chapter will we will discuss the object-oriented programming features offered by Java and C#.