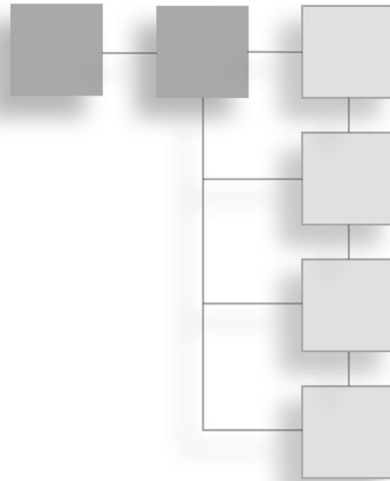


## CHAPTER 2

# THE BASICS



Chapter 1 showed you some history on why .NET and C# were created. Now it's time to dive deep into the abyss and learn just how to use C#. In this chapter, I will show you:

- How to compile and run a C# program.
- What a class is.
- What an entry point is.
- The basic data types.
- The basic mathematical and bitwise operators.
- How to declare variables and constants.
- How to perform basic typecasts.
- How to create program branches using `if` and `switch` statements.
- How to create loops using `while`, `for`, and `do-while` statements.
- How scoping works.

## Why You Should Read This Chapter

If you already know a language like C/C++ or Java, then this chapter is going to be a breeze for you. In fact, you may even be tempted to skip over this chapter. After all, the basics of most programming languages are pretty much the same within the C family of languages. Unfortunately, though, even though the syntaxes of all of the languages are close to identical, the behavior of each language is different. There's actually quite a bit about C# that is different from other languages, so it's in your best interest to go ahead and read this chapter.

## Your First C# Program

There is an ancient tradition (okay it's not that old) in computer programming that says that your first program in any language should be a "Hello World" program, a program that simply prints out a welcome message on your computer.

On the CD for this book you will find a demo entitled "HelloCSharp." You can find it in the `/Demos/Chapter02/01-HelloCSharp/` directory. The `HelloCSharp.cs` file in that directory contains the code for the program; you can open it up in any text editor or Visual Studio and view it. The code should look like this:

```
class HelloCSharp
{
    static void Main( string[] args )
    {
        System.Console.WriteLine( "Hello, C#!" );
    }
}
```

At first glance, you can see that this is about four or five lines longer than you could write it in C or C++; that's because C# is a more complicated language.

### Classes

C# is an *object-oriented* programming language, which may not mean anything to you at this point. I will go over the concepts in much more detail in Chapter 3, "A Brief Introduction to Classes," but for now, all you need to know is that C# represents its programs as objects.

The idea is to separate your programs into nouns and verbs, where every noun can be represented as an object. For example, if you make a game that has spaceships flying around, you can think of the spaceships as *objects*.

A *class* in a C# program describes a noun; it tells the computer what kind of data your objects will have and what kind of actions can be done on them. A spaceship class might tell the computer about how many people are in it, how much fuel it has left, and how fast it is going.

In C#, your entire program is actually a class. In Demo 2.1, you have the `HelloCSharp` class, which is the name of the program.

### The Entry Point

Every program has an *entry point*, the place in the code where the computer will start execution. In older languages like C and C++, the entry point was typically a global function

called `main`, but in C# it's a little different. C# doesn't allow you to have global functions, but rather it forces you to put your functions into classes, so you obviously cannot use the same method for a C# entry point. C# is like Java in this respect; the entry point for every C# program is a *static function* called `Main` inside a class, like the one you saw defined in Demo 2-1. I'll cover functions and static functions in a lot more detail in Chapter 3, so just bear with me for now.

Every C# program must have a class that has a static `Main` function; if it doesn't, then the computer won't know where to start running the program. Furthermore, you can only have one `Main` function defined in your program; if you have more than one, then the computer won't know which one to start with.

---

**note**

Technically, you can have more than one `Main` function in your program, but that just makes things messy. If you include more than one `Main`, then you need to tell your C# compiler which class contains the entry point—that's really a lot of trouble you can live without.

---

## Hello, C#!!

The part of the program that performs the printing is this line:

```
System.Console.WriteLine( "Hello, C#!!" );
```

This line gets the `System.Console` class—which is built into the .NET framework—and tells it to print out "Hello, C#!!" using its `WriteLine` function.

## Compiling and Running

There are a few ways you can compile this program and run it. The easiest way would be to open up a console window, find your way to the demo directory, and use the command-line C# compiler to compile the file, like this:

```
csc HelloCSharp.cs
```

The other way you could compile this program would be to load up the `01-HelloCSharp.cmbx` project file in `SharpDevelop` or the `01-HelloCSharp.sln` file in `Visual Studio.NET`, depending on which IDE you're using. You can find more detailed instructions on how to do this in Appendix B.

Now, when you run the program, you should get a simple output on your screen:

```
Hello, C#!!
```

Ta-da! You now have your very first C# program, which spits out some text to your screen!

## The Basics

Almost every programming language has common properties. For one thing, programming languages generally know how to store data. They must also operate on that data by moving it around and performing calculations on it.

### Basic Data Types

Like most programming languages, C# has a large number of built-in data types, mostly representing numbers of various formats. These are shown in Table 2.1.

#### note

C# is an *extendible* language, which means that you can create your own data types later on if you want. I'll go into much more detail on this in Chapter 3.

**Table 2.1** C# Built-in Data types

Type	Size (bytes)	Values
bool	1	true or false
byte	1	0 to 255
sbyte	1	-128 to 127
char	2	Alphanumeric characters (in Unicode)
short	2	-32,768 to 32,767
ushort	2	0 to 65,535
int	4	-2,147,483,648 to 2,147,483,647
uint	4	0 to 4,294,967,295
*float	4	-3.402823x10 <sup>38</sup> to 3.402823x10 <sup>38</sup>
long	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	8	0 to 18,446,744,073,709,551,615
*double	8	-1.79769313486232x10 <sup>308</sup> to 1.79769313486232x10 <sup>308</sup>
**decimal	16	-79,228,162,514,264,337,593,543,950,335 to 79,228,162,514,264,337,593,543,950,335

\* - These are *floating-point* formats, which can represent inexact decimal values

\*\* - This is a *fixed-point* format, which represents exact decimal values with up to 28 digits

The integer-based types (byte, short, int, long, and so on) can only store whole numbers, such as 0, 1, 2, and so on; they cannot hold decimal numbers, such as 1.5 or 3.14159.

In order to hold decimal numbers, you need to switch to either a *floating-point* or a *fixed-point* format. The exact details on how these kinds of numbers are stored is beyond the scope of this book, but there is a subtle difference that will affect scientists and mathematicians (but probably not game programmers).

#### note

Basically, floating-point numbers cannot hold precise numbers; they can only approximate decimal numbers within a certain amount of error. For example, using floats, you can represent the numbers 1.0 and 1.00000012, but you can't represent any number in between. So, if you set a float to be equal to 1.00000007, then the computer will automatically round that up to 1.00000012. Doubles are the same way, but have more precision (up to 15 digits). Decimals are encoded in a different way, and even though the .NET documentation calls them *fixed-point* numbers, they are still technically *floating-point* numbers, and they have a precision of up to 28 digits.

## Operators

*Operators* are symbols that appear in a computer language; they tell the computer to perform certain calculations on data. Operators are commonly used in math equations, so I'm sure this concept will be very familiar to you.

The C# language has a number of built-in operators in the language, and if you've ever used C++ or Java, then you probably already know most of them.

### Mathematical Operators

C# has five basic mathematical operations built into the language, as shown in Table 2.2.

**Table 2.2** Basic Mathematical Operators in C#

Operator	Symbol	Example	Result
Addition	+	5 + 6	11
Subtraction	-	6 - 5	1
Multiplication	*	6 * 7	42
Division	/	8 / 4	2
Modulus	%	9 % 3	0
Increment	++	10++	11
Decrement	--	10--	9

The first four operators are no-brainers, or at least they ought to be. The fifth operator may be new to you if you haven't done a lot of programming before. Modulus is sometimes

## 18 Chapter 2 ■ The Basics

known as “the remainder operator” or “the clock operator.” Basically, the result from a modulus operation is the same as the remainder if you took the first number and divided it by the second. In the example given in Table 2.2, 3 divides into 9 evenly, so the remainder is 0. If you took  $10 \% 3$ , the result would be 1, as the remainder of  $10/3$  is 1.

**note**

---

Modulus is often called the clock operator because you can easily calculate the result using a clock. For example, take the calculation  $13 \% 12$ . Imagine you have the hand of a clock starting at 12, and you move it forward one hour every time you count up by 1. So when you count to 1, the hand will be at 1, and when you count to 2, the hand will be at 2, and so on. Eventually, when you get to 12, the hand will be at 12 again, and when you count to 13, the hand moves back to 1. So the result of  $13 \% 12$  is 1.

---

**note**

---

The increment and decrement operators actually each have two different versions: the post- and pre- versions. For example,  $++x$  is the pre-increment version, and  $x++$  is the post-increment version. The difference is when the operators actually perform their calculations. For example, if  $x$  is 10 and you write  $y = x++$ , then the computer first puts the value of  $x$  into  $y$  and then increments  $x$ , leaving  $y$  equal to 10 and  $x$  equal to 11 when the code is done. On the other hand,  $y = ++x$  performs the increment first and performs the assignment later, leaving both  $x$  and  $y$  equal to 11. This is another holdover from C, and can make it ugly and difficult to read, so I don't really recommend using these operators too much.

---

You should note that all mathematical operators have alternate versions that allow you to directly modify a variable (see more about variables later on in this chapter). For example, if you wanted to add 10 to  $x$ , you could do this:

```
x = x + 10;
```

But that's somewhat clunky and redundant. Instead, you can write this:

```
x += 10;
```

All of the other math operators have similar versions:

```
x *= 10; // multiply by 10
x /= 10; // divide by 10
x -= 10; // subtract 10
x %= 10; // modulus by 10
x >>= 2; // shift down by 2
x <<= 2; // shift up by 2
```

### Bitwise Math Operators

In addition to the standard math operators, there are also bitwise math operators, which perform binary math operations on numbers. The basic bitwise operators in C# are listed in Table 2.3.

**Table 2.3** Basic Bitwise Operators in C#

Operator	Symbol	Example	Result
Binary And	&	6 & 10	2
Binary Or		6   10	14
Binary Xor	^	6 ^ 10	12
Binary Not	~	~7*	248

\* - this example is performed on a byte

Bitwise math operators have alternate versions as well:

```
x &= 10; // and by 10
x |= 10; // or by 10
x ^= 10; // xor by 10
```

### Shifting Operators

There are two shifting operators, << and >>. These operators shift the bits in a number up or down, resulting in the following equations:

- -  $x \ll y$  is the same as  $x * 2^y$
- -  $x \gg y$  is the same as  $x / 2^y$

So  $5 \ll 3$  is the same as  $5 * 8$ , or 40, and  $40 \gg 3$  is the same as  $40 / 8$ , or 5.

#### note

Bitshifting is a lot faster than straight multiplication or division, but it's rarely used anymore. The speed savings just aren't that spectacular, and it makes your programs harder to read, anyway.

### Logical Operators

There are a few common logical operators that perform comparisons on things and return the Boolean values `true` or `false`, depending on the outcome. Table 2.4 lists the logical operators.

**Table 2.4** Logical Operators in C#

Operator	Symbol	Example	Result
Equals	==	1 == 2	false
Des Not Equal	!=	1 != 2	true
Less Than	<	1 < 2	true
Greater Than	>	1 > 2	false
Less Than or Equal To	<=	1 <= 2	true
Greater Than or Equal To	>=	1 >= 2	false
Logical And	&&	true && false	false
Logical Or		true    false	true
Logical Not	!	!true	false

\* - This example is performed on a byte

## Variables

In C#, as in almost any other language, you can create instances of the basic data types, called *variables*, and perform mathematical operations on them.

Declaring a piece of data in your program is an easy thing to do. All you need to do is put in the name of the type of data, then the name of the variable you want to create after that, and then (optionally) initialize the data with a value. Here's an example:

```
int x = 10;
float y = 3.14159;
decimal z;
```

### caution

Note that if you try using a variable before initializing it (if you try using *z* from the previous code sample, for example), then you will get a compiler error in C#. Older languages, such as C and C++, would allow you to use a variable without giving it a value, which could cause a lot of errors because you never know what was in the variable if you never set it!

Here's an example using variables with the mathematical functions:

```
int x = 10 + 5;           // 15
int y = 20 * x;          // 300
int z = x / 8;           // 1
float a = (float)x / 8.0; // 1.875
x = (int)a;              // 1
```

Pay particular attention to the last two lines. These lines show you how to use *typecasts* in your program. An explanation of typecasts is coming soon.

## Constants

You can declare *constants*, pseudo-variables that cannot be changed, in your code. This is just another safety feature that's been around in computer languages for years now. For example:

```
const float pi = 3.14159;
```

Now you can use `pi` in your calculations, but you can't change its value (because changing the value of `pi` to 3.0 makes absolutely no sense!). This will cause a compiler error:

```
pi = 3.0; // ERROR!
```

### tip

---

Constants improve the readability of your programs by eliminating magic numbers. *Magic numbers* are numbers in your program that have no immediate meaning to whomever is reading it. For example, you can write `x = 103`; somewhere, but no one really knows what 103 means. It could mean the number of bullets in an ammo clip, or something else completely. Instead, you can use constants to show exactly what you mean, by defining a constant, called `const int BulletsInClip = 103;`, earlier in your program and then later using the constant `x = BulletsInClip;`. See how much more readable that is?

---

## Typecasts

Check out this code:

```
float a = 1.875;  
int x = (int)a; // 1
```

Look at the last line: the value of `a` is 1.875, a fractional number, and the last line of code is trying to put the value of `a` into `x`, which is an integer. Obviously, you can't just transfer the contents of `a` into `x`, so you need to lose some precision. Older languages, such as C/C++, would do this for you automatically, and chop 1.875 down to 1 in order to fit it into the integer (the process is called *truncation*). If you tried typing this line into a C# program, however, you would get a compiler error:

```
x = a; // error! Cannot implicitly convert type 'float' to 'int'
```

Of course, this code works perfectly well in older languages, so a lot of people will automatically dismiss C# as “difficult to use.” I can hear them now: “Can't you just automatically convert the float to the integer, you stupid compiler?”

## 22 Chapter 2 ■ The Basics

Well, the compiler isn't actually stupid; it's trying to save you some time debugging. You may not realize it, but a common source of bugs in programs is accidental truncation. You might forget that one type is an integer and some important data may get lost in the translation somewhere. So C# requires you to explicitly tell it when you want to truncate data. Tables 2.5 and 2.6 list which conversions require explicit and implicit conversions.

**Table 2.5** Explicit/Implicit Conversions, Part 1

From	byte	sbyte	short	ushort	int	uint
byte	I	E	I	I	I	I
sbyte	E	I	I	E	I	E
short	E	E	I	E	I	E
ushort	E	E	E	I	I	I
int	E	E	E	E	I	E
uint	E	E	E	E	E	I
long	E	E	E	E	E	E
ulong	E	E	E	E	E	E
float	E	E	E	E	E	E
double	E	E	E	E	E	E
decimal	E	E	E	E	E	E

**Table 2.6** Explicit/Implicit Conversions, Part 2

From	long	ulong	float	double	decimal
byte	I	I	I	I	I
sbyte	I	E	I	I	I
short	I	E	I	I	I
ushort	I	I	I	I	I
int	I	E	I	I	I
uint	I	I	I	I	I
long	I	E	I	I	I
ulong	E	I	I	I	I
float	E	E	I	I	E
double	E	E	E	I	E
decimal	E	E	E	E	I

The charts may look confusing at first, but they are actually quite simple. For example, if you want to convert from an `int` to a `double`, look at Table 2.2, find “`int`” on the left and find “`double`” on the top. In that position is an `I`, meaning you can perform an implicit conversion:

```
int a = 10;
double b = a; // ok
```

Now say you want to convert a `double` to an `int`. Look at Table 2.1, find “`double`” on the left and “`int`” at the top. There is an `E` at that place, which means you need to perform an explicit conversion:

```
double a = 10.0;
// int b = a <-- ERROR
int b = (int)a; // ok
```

#### **note**

---

Converting from a float or a double to a decimal requires an explicit cast. This is because decimals encode data in a different way than do floats or doubles, so there is a distinct possibility of losing some data when performing the conversion. It’s probably nothing that we game programmers should be concerned with, but you should be aware of it.

---

## **Branching**

If you’ve really studied programming languages, then you know that there are three different traits that a language must have to be considered a true programming language. They are

- Sequencing
- Branching
- Repetition

You’ve already seen the first trait, sequencing, in action in Demo 2.1. *Sequencing* essentially means that the language must be able to execute commands in a given sequence.

Now I want to cover conditional statements, the use of which is known as *branching*. Essentially, branching allows a computer program to look at a given set of variables and decide whether it should continue executing or should *branch* to a different part of the program.

`C#` has a few conditional statements built in to the language; all of them were inherited from `C`, so you may be familiar with them.

## if Statements

Quite often in a program, you will want to test to see if a condition is true or not, and then take action depending on the outcome. For example, if you wanted to perform an action if a condition evaluates to true, then you would write some code like this:

```
if( x == 10 )
{
    // do something
}
```

The code checks to see if some variable named *x* has the value of 10, and then executes the code inside the brackets if, and only if, *x* is 10. This is called an `if` statement.

You can also add on an `else` clause at the end, in order to execute code in any case where *x* is *not* 10:

```
if( x == 10 )
{
    // do something
}
else
{
    // do something
}
```

So the computer executes everything in the first block when *x* is 10, and executes anything in the second block when *x* is anything but 10.

Furthermore, you can chain `elseif` statements to the end, to perform multiple inquiries:

```
if( x < 10 )
{
    // do stuff if x < 10
}
else if( x < 20 )
{
    // do stuff if 10 <= x < 20
}
else if( x < 30 )
{
    // do stuff if 20 <= x < 30
}
```

**note**

---

If you're used to a language like C++, then you know you can use numbers inside of a conditional to produce code like this: `if( x )`, where `x` is an integer. In older languages, the computer treats 0 as false and anything else as being true, meaning that if `x` is 0, then the `if` block won't execute, but it will for anything else. C# isn't like this, however, and it actually requires you to use a Boolean inside all conditional expressions. So the code will give you a compiler error in its current form. When you think about it, the old way isn't really safe anyway, because it doesn't explain exactly what you are testing. C# makes your programs safer and more readable.

---

**Switch Statements**

Using switch statements is a handy way to compare multiple outcomes of a single variable quickly. For example, if you have a variable, `x`, that variable can hold the values of 1, 2, 3 and 4, and your program will take a different course of action for each value. You can code a switch statement to do this:

```
switch( x )
{
case 1:
    // do something if 1
case 2:
    // do something if 2
case 3:
    // do something if 3
case 4:
    // do something if 4
default:
    // do something if something else
}
```

So if `x` is 2, then the code will jump to the case 2 block, and so on.

There is a catch, however. In the current state of the code in the previous block, if `x` is 2, then the code will jump right to block 2, but it will also continue on and execute the code in every block below it. This means that the code will execute code block 3, 4, and default as well. Sometimes you may want this behavior, but most of the time you won't, so you need to use the `break` keyword to break out of the switch after each block:

```
switch( x )
{
case 1:
    // do stuff
    break; // jump out of switch
```

## 26 Chapter 2 ■ The Basics

```

case 2:
    // do stuff
    break; // jump out of switch
default:
    // do stuff
    break; // optional here
}

```

**tip**


---

The `break` in the last block of the `switch` statement is optional, of course, because there is no code below it. But it's always a good idea to include the `break` anyway—just in case you end up adding more blocks later on and forget to add in the last `break`.

---

**Short-Circuit Evaluation**

Let me go off on a tangent here and go over a topic that is fairly important when evaluating conditional statements.

All C-based languages support something called *short-circuit evaluation*. This is a very helpful performance tool, but it can cause some problems for you if you want to perform some fancy code tricks.

If you know your binary math rules, then you know that with an `and` statement, if either one of the operands is false, then the entire thing is false. Table 2.7 lists the logical `and` and logical `or` result tables.

**Table 2.7** Logical and/Logical or result tables

<b>x</b>	<b>y</b>	<b>x and y</b>	<b>x or y</b>
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Look at the table, specifically the two lines where `x` is false. For the operation “`x and y`,” it doesn't matter what `y` is because the result is always going to be false. Likewise, if you look at the first two lines, you'll notice that whenever `x` is true, the operation “`x or y`” is true, no matter what `y` is.

So if you have code that looks like this:

```
if( x && y )
```

the computer will evaluate *x*, and if *x* is false, then it won't even bother to evaluate *y*.

Likewise:

```
if( x || y )
```

If *x* turns out to be true, then *y* isn't even evaluated. This is a small optimization that can speed up your programs greatly in the right circumstances. For example:

```
if( x || ( ( a && b ) && ( c || d ) ) )
```

If this code executes and finds out that *x* is true, then the whole mess on the right side will never be calculated at all.

This can be a very tricky source of bugs, but only if you write tricky-looking code. Look at this line, for example:

```
if( x || ( ( y = z ) == 10 ) )
```

If your first reaction to seeing this code is “What the hell is going on here?” then you deserve a cookie. This code is unbelievably ugly, and you can't tell what the author intended to do with it. But unfortunately, this is perfectly legal C# code, and someone somewhere will think they're hot enough to write stuff like this and get away with it.

Anyway, if *x* is true, then the computer ignores the second half of the code. But if *x* is false, then whatever is in *z* is assigned to *y*, then the result is compared to 10, giving this code the same structure as this more readable version:

```
if( x == false )  
    y = z;  
if( x || ( y == 10 ) )
```

The second version looks almost nothing like the first, so you can see how trying to do some clever tricks will get you into loads of trouble one day.

## Looping

The third trait a computer language has is *repetition*, or *looping*. Essentially, looping allows you to perform one specific task over and over again. C# has four looping mechanisms built-in; the three I'll cover in this section are inherited from the C programming language. I won't get to the fourth one until Chapter 4, “Advanced C#.”

## 28 Chapter 2 ■ The Basics

## while Loops

The first and easiest loop structure is the `while` loop. Here's an example:

```
while( x < 10 )
{
    // do stuff
}
```

Whatever is inside the brackets will be executed over and over until the value of `x` is less than 10. If `x` never gets to be equal or above 10, then the loop will loop infinitely.

## for Loops

Another popular loop is the `for` loop, which is just a different way to perform a `while` loop. The basic syntax of a `for` loop is as follows:

```
for( initialization; condition; action )
```

The initialization part of the code is executed only once, when the `for` loop is entered. This allows you to set up any variables you might need to use.

The condition part is evaluated at the beginning of each loop; and if it returns false, then the loop exits.

The action part is executed at the end of every loop.

Generally, you use `for` loops to create a loop that will go through a range of numbers for a particular variable. For example, if you want to perform 10 calculations on `x`, where `x` ranges from 0 to 9, you would create a loop like this:

```
for( int x = 0; x < 10; x++ )
{
    // do stuff
}
```

The first time the loop executes, `x` is 0, and then the next time it is 1, and so on, until it reaches 9.

You can also do some other fancy stuff, like initialize multiple variables or perform multiple actions:

```
for( int x = 0, int y = 0; x < 10; x++, y += 2 )
```

This loop creates two variables, `x` and `y`, where `x` loops from 0 to 9, and `y` loops from 0 to 18 by skipping every other number.

## do-while Loops

Sometimes in programming, a situation will arise in which you want to make absolutely certain that a loop executes at least once. Look at this code, for example:

```
int x = 0;
while( x > 0 )
{
    // this loop never gets executed
}
```

With for loops and while loops, there's always a chance that, if the condition evaluates to false, the code inside the loop will never be executed. Instead of a for loop or a while loop, you can use the do-while loop, which executes everything and checks the condition after the loop is executed. Here's an example:

```
do
{
    // loop code here
} while( condition );
```

## Break and Continue

`break` and `continue` are two useful keywords that you can use when doing stuff inside of loops to alter their flow.

### **Break**

The first is the `break` keyword, which you've already seen used inside of switch blocks. Basically, putting in a `break` will cause the program to jump to the end of the loop and exit. Here's an example:

```
for( int x = 0; x < 10; x++ )
{
    if( x == 3 )
        break;
}
```

This loop will make `x` go through values 0, 1, 2, and 3, and then quit out when `x` is 3.

### **Continue**

The other loop modifier is the `continue` keyword. This keyword causes the loop to stop executing and go back up to the top and start over. Here's an example:

```
for( int x = 0; x < 10; x++ )
{
    FunctionA();
}
```

## 30 Chapter 2 ■ The Basics

```
    if( x == 3 )
        continue;    // jump back up to top, skip anything below
    FunctionB();
}
```

Pretend that `FunctionA` and `FunctionB` actually exist for a moment. This loop will make `x` go through every number from 0 to 9. On every single iteration, `FunctionA` will be executed, but when `x` is 3, the code will skip `FunctionB()` and jump right up to the top of the loop again.

## Scoping

The term *scope*, when dealing with a computer program, refers to the place in a program where a variable is valid. For example, say you have this code in a program:

```
class ScopeDemo
{
    static void Main( string[] args )
    {    // bracket A
        int x = 10;
    }    // bracket B

    static void blah()
    {
        //    x = 20;    <--- YOU CAN'T DO THIS!
    }
}
```

The variable `x` is said to have a scope between brackets A and B. If you tried referencing `x` outside of those brackets, the C# compiler will give you a strange look and ask you what the hell you're talking about.

Seems simple enough, doesn't it? Here's another example:

```
static void Main(string[] args)
{
    if( 2 == 2 )
    {    // bracket A
        int y;
    }    // bracket B
    // y = 10;    <--- YOU CAN'T DO THIS!
}
```

The `if` block in this code will always execute because 2 is, obviously, always equal to 2; but that's beside the point. Inside of brackets A and B, a new variable, `y`, is created, and then the `if` block ends. But `y` only has a scope in between those two brackets, meaning that

nothing outside of the brackets can access it; so if you try using *y* outside of the `if` block, the computer will barf error messages all over you because it has no idea what *y* actually is.

There's still one more example I'd like to show you:

```
static void Main(string[] args)
{
    for( int x = 0; x < 10; x++ )
    {
        // do something here
    }
    // x = 10;    <--- YOU CAN'T DO THIS!
}
```

In this final example, you've created a new variable *x* *inside* the `for` statement, and you can access *x* anywhere inside the parentheses or the `for` block, but nowhere outside of it.

## Summary

Computer languages are very complex, and no one can ever fully understand an entire language anymore—they're just far too complex nowadays. Luckily, you won't need many of the features in a language, so you don't have to be a versed expert in the language in order to use it—that's what reference manuals are for.

This chapter is enough to get you started on making some simple C# programs, but you really can't do anything really complex yet. But that's okay; you're only two chapters into the book!

This chapter has shown you how to create your very first C# program and compile it, and has introduced you to the very basic concepts of the language, such as the basic data types, mathematical operators, conditional statements, and looping statements. In the next chapter you'll go on to even more advanced topics.

## What You Learned

The main concepts that you should have picked up from this chapter are:

- How to compile and run a C# program.
- Every program has a main class that defines an entry point, where program execution starts.
- There are many built-in numeric data types in C#.
- Short-circuit evaluation can be used to speed up your programs, but may introduce unforeseen flaws.
- Constants make your programs easier to read.

## 32 Chapter 2 ■ The Basics

- Typecasts are strict in C# when compared to C/C++, because you might accidentally lose data if you're not paying close enough attention.
- Scoping allows you to manage your variables in an efficient manner.

### Review Questions

These review questions test your knowledge of the important concepts explained in this chapter. The answers can be found in Appendix A.

- 2.1. Every C# program requires at least one main static class. (True/False)
- 2.2. Booleans are only 1 bit in size. (True/False)
- 2.3. Unsigned integers can hold numbers up to around 4 billion. (True/False)
- 2.4. Floating point numbers hold exact representations of numbers. (True/False)
- 2.5. Why can't you use variables before they have been assigned a value?
- 2.6. Why do constants make your programs easier to read?
- 2.7. Is the following code valid?

```
int x = 10;
float y = 20;
x = y;
```

(Yes/No)

- 2.8. What is the value of *x* after this code is done?

```
int x = 10;
if( x == 10 )
    x = 20;
```

- 2.9. Assume that *c* is 0. What are the values of the variables after this code is done, and why?

```
int w = 0, x = 0, y = 0, z = 0;
switch( c )
{
case 0:
    w = 10;
case 1:
    x = 10;
case 2:
    y = 10;
    break;
case 3:
    z = 10;
    break;
}
```

2.10. Now assume that `c` is 2 and the code from Question 2.9 is run again. What are the values of the variables `w`, `x`, `y`, and `z`?

2.11. Does the computer compare the value of `x` and 10 in this example?

```
int x = 10, y = 20;
if( y == 20 && x == 10 )
    x = 20;
```

2.12. Does the computer compare the value of `x` and 10 in this example?

```
int x = 10, y = 20;
if( y == 20 || x == 10 )
    x = 20;
```

2.13. When this code is completed, what is the value of `x`?

```
int x = 0;
while( x < 10 )
    x++;
```

2.14. For each loop, does the value of `x` increase before `FunctionA` executes or after it executes?

```
for( int x = 0; x < 10; x++ )
{
    FunctionA();
}
```

2.15. Rewrite the code in Question 2.14 using a `while` loop instead.

2.16. How many times is `FunctionA` executed?

```
int x = 0;
do
{
    FunctionA();
} while( x == 1 );
```

2.17. What is the value of `x` after the following code is done?

```
int x = 0;
for( int y = 0; y < 10; y += 2 )
{
    if( y == 4 )
        break;
    x++;
}
```

2.18. What is the value of `x` after the following code is done?

```
int x = 0;
for( int y = 0; y < 10; y += 2 )
```

## 34 Chapter 2 ■ The Basics

```
{  
    if( y == 4 )  
        continue;  
    x++;  
}
```

2.19. Is this code valid? (Assume that `FunctionA` exists.)

```
for( int y = 0; y < 10; y++ )  
{  
    FunctionA();  
}  
y = 0;
```

### On Your Own

Play around with the looping structures to find out what they do exactly. Sometimes they can be a little bit difficult to pick up for an absolute beginner, and it's very important that you learn exactly how they operate before you start making serious code.