Chapter 2. Your First C# Program

In the previous chapter, you typed in, compiled, and ran your first C# program. We had you do that because we wanted you to understand how those steps work before we start using C# to solve more problems. Now that you've done that, though, it's time to take a closer look at C# programs in general.

2.1. What Does a C# Program Look Like?

This is actually a trickier question than you might think! Because C# is an object-oriented language, we're going to end up writing lots of things called *classes*. We use classes as building blocks for our problem solutions, so they're a critical part of our C# programs. We'll talk about classes lots more throughout the book, including later in this chapter, so let's put off that discussion for a while longer.

Although our programs will use lots of classes, we also need a .cs file we can actually run. Lots of people call this program the *application class* because it's the class that essentially runs the program (another word for application). If we think of our program as a software application or app (which it is), the application class terminology makes sense. The Program.cs class you typed in followed the "typical" format for such an application class; the syntax for a C# application class is provided below. We'll discuss all the parts in further detail, so don't worry if you don't understand them all right away.

SYNTAX: C# Application Class

```
using statements for namespaces

namespace NamespaceName
{
    class documentation comment

    class ApplicationClassName
    {
        method documentation comment

        static void Main(string[] args)
        {
            constant declarations

            variable declarations

            executable statements
        }
    }
}
```

Since we'll be providing lots of "syntax descriptions" in the coming chapters, we should discuss the notation a bit. When we list items in italics (e.g., *`class documentation comment`* or *`ApplicationClassName`*), that means you, the programmer, decide what goes there. When we list words without any special formatting (such as `namespace`, `class`, `static`, and `void`), those words need to appear EXACTLY as written. That's because these are special words in C# (they're called *keywords*), so you need to use them just as they appear in the syntax descriptions.

Okay, let's discuss each of the parts of a C# application class in greater detail.

2.2. Using Other Namespaces and Classes

The first thing we see in the application class is the place where we say we'll be `using` other namespaces and classes. Namespaces and classes are collections of useful C# code that someone else has already written, tested, and debugged (so that you don't have to!). In fact, all C# development environments come with a set of namespaces and classes that you'll find very useful. One example that we'll use a little later is a class that will generate random numbers for us. That class (called, surprisingly enough, `Random`) is found in the `System` namespace, so when we need it, we'll include the following line in our program:

```
using System;
```

This tells the compiler that you want access to the classes in the `System` namespace. We'll introduce you to other namespaces provided in C# as we need them. Of course, MonoGame gives us even more namespaces to use as we develop games, and we'll learn about those namespaces as we need them as well.

So there are lots of handy namespaces and classes available to us, but how do we find out more about them? By looking at the documentation that you installed when you installed the IDE, of course! You'll probably find that you use the documentation a lot as you learn to program in C# – and even when you're an experienced C# programmer – so let's walk through the process now.

The typical way you'll use the documentation is by clicking Help > View Help from the IDE. The pane on the left lets you explore the documentation by browsing through different categories, but you'll typically be looking for information about a specific class or namespace rather than just browsing. Let's try to find out more about the `Random` class. Type Random Class in the Search box in the upper left corner and press <Enter> (or click the magnifying glass). After the search results are returned, clicking on the "Random Class (System)" result in the results pane brings you to the documentation for the `Random` class. You should always make sure the C# tab – inside the yellow circle in Figure 2.1 – is selected to get the C# syntax. The documentation page you get should look like Figure 2.1 (without the yellow circle, of course), but remember that your documentation may look slightly different if you're using online help.

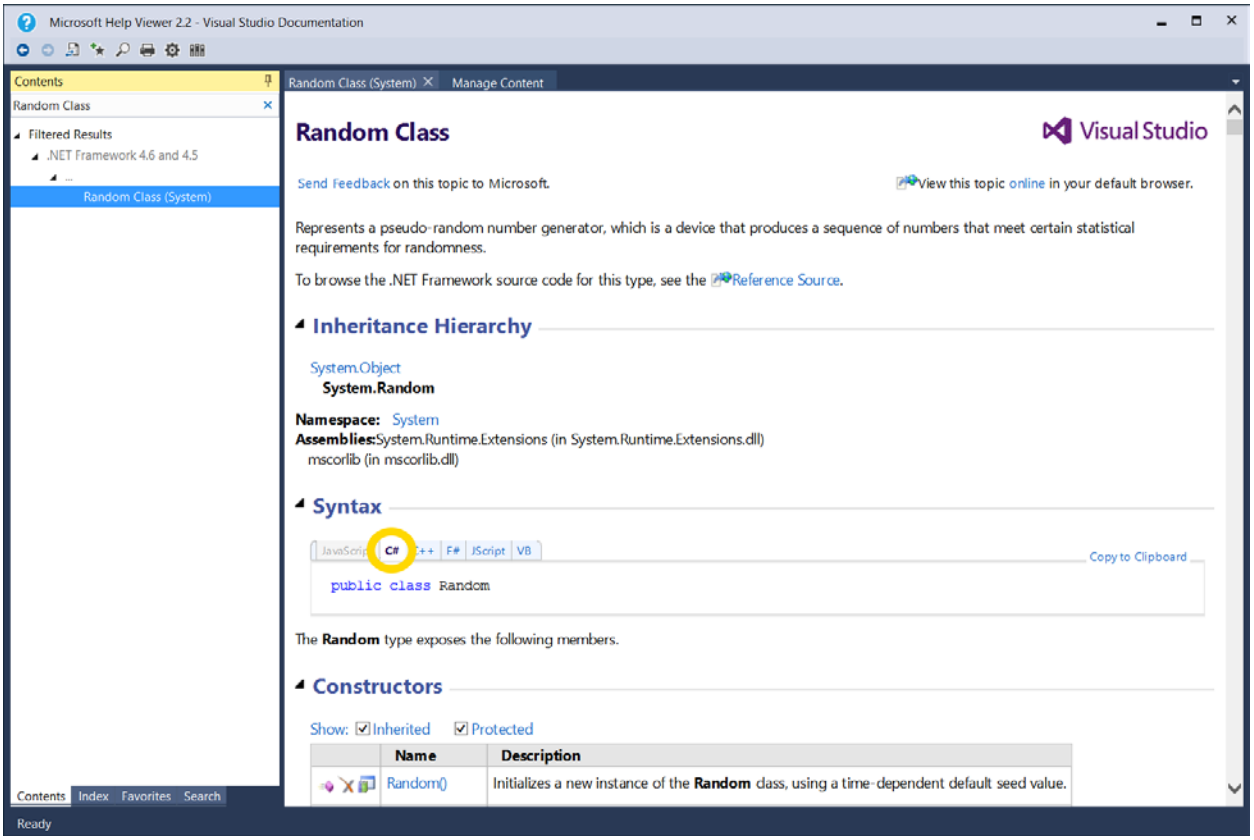

Figure 2.1. Random Class Documentation Window

You can then explore the functionality that's provided by the class by clicking on the links in the documentation. We'll look at using documentation in more detail later in this chapter.

2.3. Namespace

The next item in the syntax description is the line that tells what namespace this class belongs to. As discussed in the previous section, namespaces provide a way to group related chunks of C# code together. Most of the code you write as you work (slog?) through this book will be contained in a single namespace, and the IDE will automatically generate this part of your code based on the name you decided to use when you created the project. You generally won't have to touch this code at all.

2.4. Comments

The next item in the syntax description is the class documentation comment. Comments that start with three slashes, `///`, are called documentation comments; XML tags are used within the documentation comments.

Although comments aren't actually required by the language, it's always a good idea to provide documentation within your programs. The class documentation comment should contain a description of the class. Here's the class documentation comment from our program:

```
/// <summary>
/// A class to print a rain message
/// </summary>
```

The class documentation comment should start with the line `///<summary>` and end with the line `///</summary>`. The good news is that when you type three slashes on the line above the line that starts with `class`, the IDE automatically generates and pre-populates the documentation comment; you just fill in the actual details. You should also notice that the IDE colors the text you enter for comments green.

Now, you might be wondering why we call these documentation comments. They actually do more than just making it easier to understand the source code. There are tools available (Sandcastle is one of them) that can be used to process all your source code and generate navigable documentation in a compiled HTML file. Although we won't actually teach you how to use Sandcastle in this book[1], when you start developing games in larger teams of programmers (larger meaning more than just you!) you'll want to generate the documentation so that other programmers using your code don't have to actually read your source code to figure out how to use it. So you should definitely start using documentation comments now.

There are also two other kinds of comments we include in the programs we write. For each method in our solution (like the `Main` method), we write a documentation comment similar to the class documentation comment. Method documentation comments contain a description of the method and information about parameters and return values for the method. Don't worry; we'll cover all these ideas in great detail as we need them.

The last kind of comment we include in our program is called a *line comment*. Line comments start with a double slash `//`, followed by whatever descriptive text you choose to add after the double slash. Every few lines of C# code you write should have a comment above them explaining what those lines are supposed to do. It's not necessary to provide a line comment for every line of code, because that actually just ends up making things more confusing. There are no set rules about how much or little you should comment your code with line comments, but providing a line comment every 3-5 lines of code is probably a good rule of thumb. It's also possible to add a line comment at the end of a line that also contains other C# code, but we typically don't do that in this book.

On some occasions, you may find that you want to use several lines of comments to explain the code after the comment. We actually try to avoid that in this book (and our coding in general), but if you need to you can make your comment span multiple lines by starting with `/*`, inserting your comment, then ending it with `*/`.

2.5. Class

[1] We used Sandcastle to generate all the documentation we provide for the code we wrote for the book, though.

The next item in the syntax description is the line that defines the class. The IDE colors class names teal.

2.6. Identifiers

Whenever we need to name something in our program, such as *`ApplicationClassName`*, we need to use an *identifier* (which is just another word for name). C# has a few rules for which identifiers are legal and which aren't.

Identifiers can contain any number of letters, numbers, and underscores. They can't start with a number, and they can't contain any spaces. Identifiers can't be keywords (remember, those special C# words like `using`, etc.). How do you know if you're using a C# keyword as an identifier by mistake? You can easily tell because the IDE colors all the keywords in your program blue.

Here are some identifier examples:

Legal identifiers: `Weapon`, `playerName`, `Spell`, `TotalDelayMilliseconds`
Illegal identifiers: `2_Names`, `int`

One other comment about identifiers. C# is *case sensitive*, which means that the identifier `playerName` is NOT the same as the identifier `playername`. Be really careful about this, and be sure to use a consistent capitalization style to avoid confusion.

2.7. The Main Method

All the application classes we write will have a method called `Main`. The `Main` method is the main entry point for the application; when we run our program, it will simply do the things we told it to do in the `Main` method. You always want to declare the first line of the `Main` method exactly as shown in the syntax description:

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

We'll talk about the words in front of `Main` (`static`, `void`) as we need them. The part that says (`string[] args`) lets people use something called "command-line arguments." We won't be using command-line arguments in this book, but C# requires that we include them for our `Main` method whether or not we use them. Bottom line – as long as you declare the first line of the `Main` method EXACTLY as shown in the syntax description, you'll be fine. This is made even easier by the fact that the IDE automatically provides this to you, so the only way to screw it up is for you to go change that line somehow.

2.8. Variables and Constants

Computers were originally designed to do mathematical calculations, so it shouldn't come as a surprise that many of the programs you write will also do calculations. You should remember from your studies of Einstein's work that

Energy = mass * c^2

Energy and mass are the variables in the equation, while c (for the speed of light) is a constant in the equation. C# lets us declare the required variables and constants as follows:

```
const double C = 3.0E+8;
double energy;
double mass;
```

Don't worry about what `double` means yet – we'll get to data types in Chapter 3. At this point, you should simply realize that C# lets us declare both variables and constants.

2.9. Console Output

It's nice to have a program that goes off and crunches numbers, but it's even nicer to have the program tell us the answer after it figures it out! And how does the program get the numbers in the first place? That's what input and output are all about. In this section, we'll talk about how we do console output; we'll cover input and other forms of output as we need them throughout the book.

Input and Output (commonly called IO) is of course also important in game development. Gamers need to interact with the game world – that's pretty much the point of playing a game – so we'll talk about some standard ways to get and process user input once we start using MonoGame. We'll also obviously talk lots about game output as well. Although countless hours of productivity have been lost to people playing games that say things like "You've conquered the Orc. Do you want to leave the meadow or stay and rest?", modern gamers typically expect graphics, audio, and in some cases force feedback in their game output. This section only covers output to the console, but later in the book we'll cover more typical forms of game output.

To do our output, we'll use the `Console.Write` and `Console.WriteLine` methods, which are contained in the `Console` class. Basically, the `Console.Write` method outputs whatever we tell it to print on a line on the screen, then leaves the cursor on that line so we can print more on that line. The `Console.WriteLine` method moves the cursor to the next line after printing what we tell it to.

Before we look at the details for using these methods, we should talk briefly about what methods are. A method is simply a self-contained piece of code that does something useful for us (like displaying output on a screen). You may also hear people talking about functions rather than methods; for our purposes, they're the same, though method is the correct term in the C# world.

The thing that makes methods generally useful is our ability to provide information to the method when we use it. For example, a `Console.WriteLine` method that always prints "diaf, n00b" might be amusing for a while, but we're going to eventually need to output other text to

the screen. From the method's perspective, information is passed into the method using *parameters*. From the perspective of the code using the method, information is passed into the method using *arguments*. Let's look at an example.

To figure out what information a method expects us to provide, we need to read the documentation. To get to the documentation page shown in Figure 2.2, we searched on Console Class, then clicked on the result labeled Console Class (System).

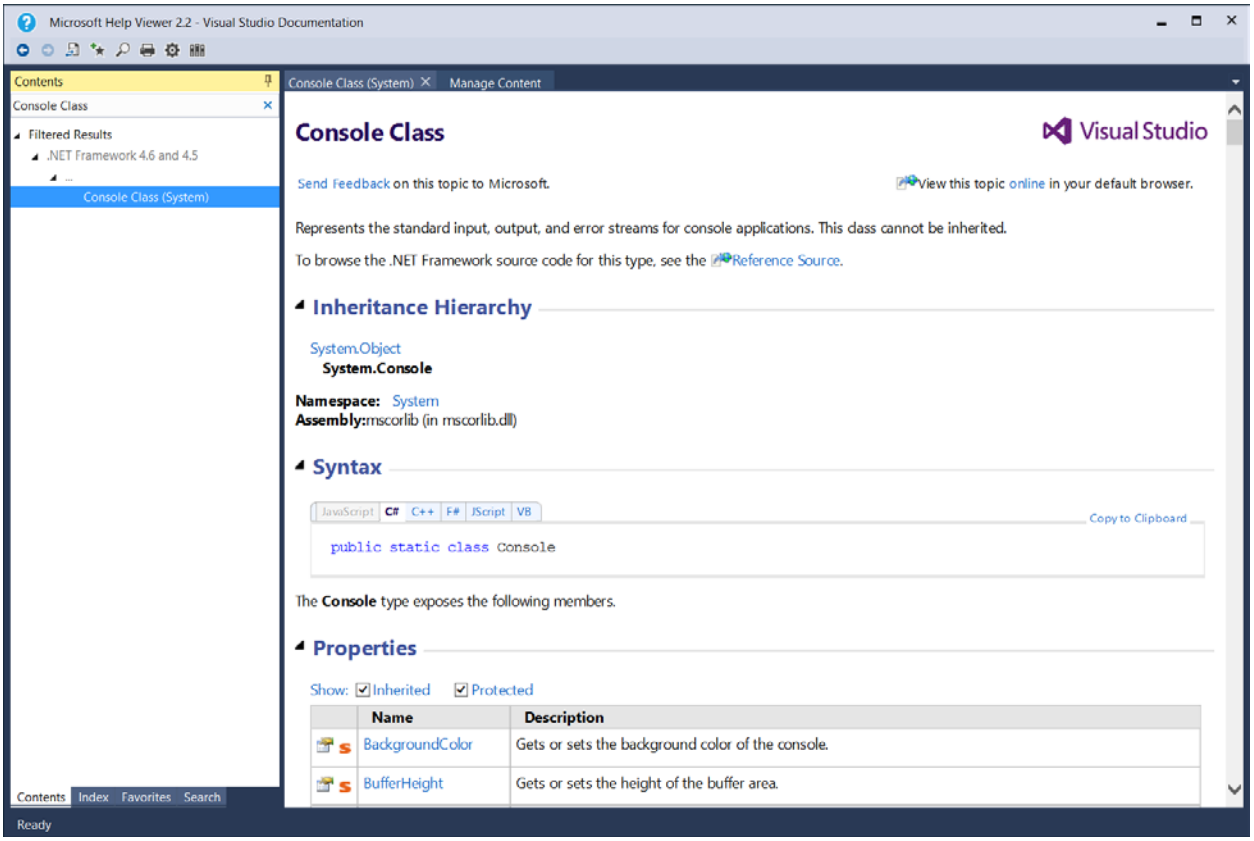

Figure 2.2. Console Class Documentation

As you can see, there are lots of things we can tell the `Console` class to do, but for now we're interested in how to use the `WriteLine` method. Luckily, everything is in alphabetical order so we can easily scroll down to `WriteLine` in the Methods area of the documentation; see Figure 2.3.

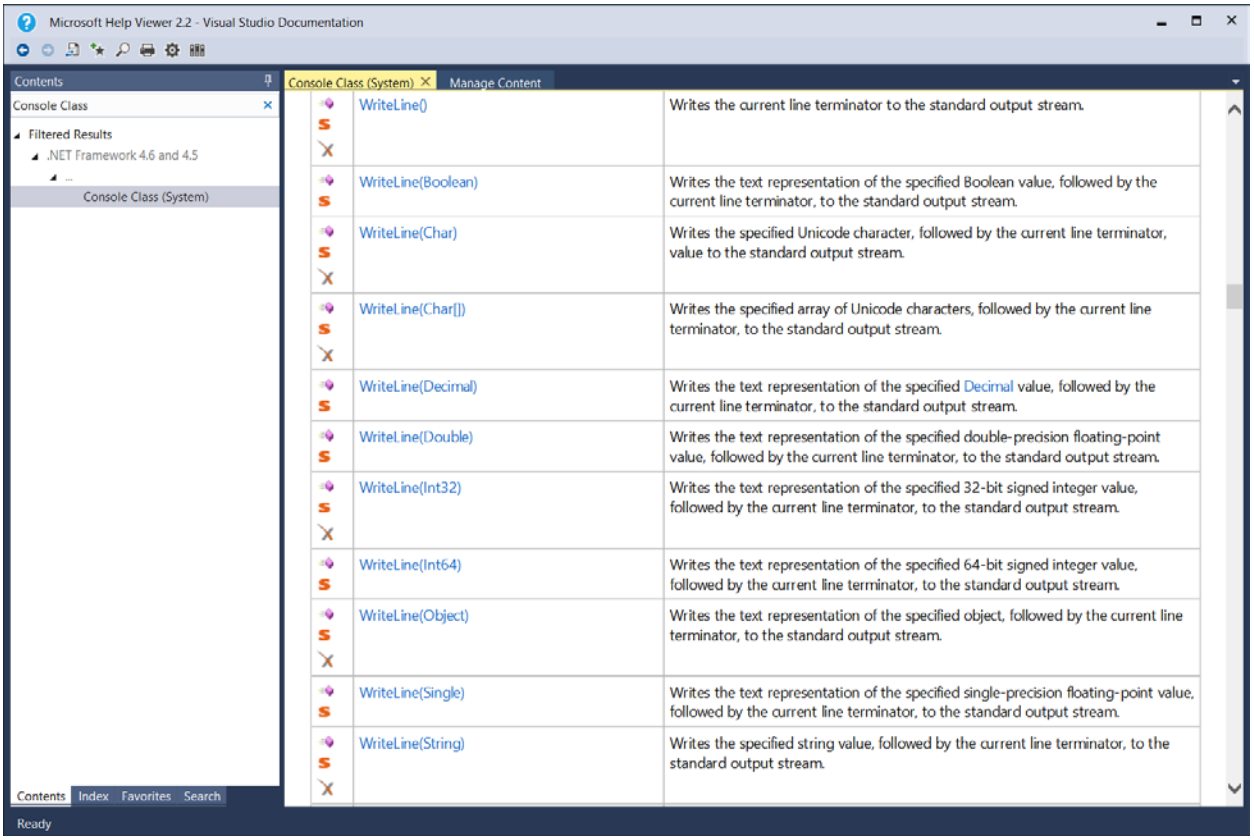

Figure 2.3. Console WriteLine Area

Holy smokes! There appear to be lots of ways we can use the `WriteLine` method! We'll learn about the `String` class in a later chapter, so for now you'll just have to trust us that we need to output a string.

The documentation tells us that the version of the `WriteLine` method we're going to use (just past the middle of the pane on the right in Figure 2.3) has a single parameter that's a `String`. That means that when we call this method, we need to provide a single string argument to the method. For example,

```
Console.WriteLine("Hello, world");
```

will output the words "Hello, world" (without the quotes) on a single line on the screen.

So we're calling the method with a single argument, the string `"Hello, world"`. We could call the method with any other string we can think of (like, say, `"diaf, n00b"`) and the method would output that string instead.You might be wondering why we spent so much time talking about how to navigate the documentation instead of just telling you how to use the `WriteLine` method. We're glad you asked! You'll spend a significant amount of time as a game programmer reading documentation so you can use the C# and MonoGame classes properly. Learning how to use the provided documentation is therefore a really important thing for you to do.

2.10. A Word About Curly Braces

You'll notice that there are probably more curly braces in the syntax description than you're used to seeing. C# uses curly braces to block off areas of code. For example, there's an opening curly brace after the line containing the application class name, and the closing curly brace for that opening curly brace is the next to last line in the program. That means that everything between those two braces is part of the application class. Similarly, the `Main` method has an opening curly brace just below the start of the method and a closing curly brace at the end of the executable statements for the method. You'll want to make sure that you always have matching opening and closing curly braces, and that you always put them in the right places! The IDE helps with this by highlighting the corresponding opening curly brace when you type a closing curly brace.

2.11. A Matter of Style

While beginning programmers tend to think that a program that works properly is perfect, more experienced programmers also recognize the importance of style. Good programming style tells us to use variable names that are descriptive (such as `firstInitial` instead of `fi`) and to use lower case letters at the start of variable names, capital letters at the start of class names, and capital letters to start the "internal words" in those names. It also tells us to use proper indentation (we use 4 spaces in this book), good commenting, and to include "white space" (blank lines) in our programs. Following these style guidelines makes your programs easier to read and understand.

Like most style matters, programming style can be largely a matter of taste. We've selected a particular style for all the examples in this book, but your teacher may want you to use a different style; most companies developing software have coding standards that specify the style that everyone in the company should use. In any case, using reasonable, consistent style guidelines will help you develop better code.

2.12. Solving Problems One Step at a Time

So far, we've been talking about coding details without really talking about how we actually go from a problem description to the code that solves the problem. In this book, we'll do that using the following steps:

1. Understand the Problem
2. Design a Solution
3. Write Test Cases
4. Write the Code
5. Test the Code

Problem solving is an iterative process no matter what set of steps we use. For example, we may get all the way to running our test cases before realizing that we didn't design a correct solution, so we'd need to go back to that step and work our way through to the end again. Similarly, we may realize as we write our code that we forgot something in our test cases, so we'd return to that step to make our corrections. There's nothing wrong with going back to previous steps; as a

matter of fact, it's a natural part of the problem-solving process, and you'll almost always need to iterate a few times before reaching a correct problem solution.

In fact, we're going to plan on completing Steps 3, 4, and 5 many times as we solve our problems; we might even have to return to Steps 1 and 2 once in a while. Nobody in their right mind tries to write a complete test plan, write all their code, and then run all the tests against their code. Good programmers go through Steps 3, 4, and 5 with small pieces of their solution. Not only does this make solving large, complex problems more manageable, it also makes it easier to find where the bugs are when your tests fail. If you've only added a few lines of code since you last tested the code successfully, the bug is probably somewhere in those new lines of code.

Let's look at each of the five steps in more detail.

1. Understand the Problem
This certainly seems like common sense – how can you possibly solve the problem correctly if you don't know what it is? In other words, understanding the problem requires that you understand **WHAT** is required. Surprisingly, lots of people try to jump right into finding a solution before they even know what problem they're trying to solve. This may be due, at least in part, to the feeling that we're not making progress toward the solution if we're "just" thinking about the problem[2]. In any case, you need to make sure you understand the problem before moving on.

As an example, consider the following problem: "Look through a deck of cards to find the Queen of Hearts." Do you REALLY understand the problem? Do you know what to do if you're not playing with a full deck <grin>? Do you know what to do if the Queen of Hearts isn't in the deck? Do you know what to do after you find the Queen of Hearts? Even the simplest problems require you to think carefully before proceeding to the next problem-solving step.

2. Design a Solution
Once you know what the problem is, you need to formulate a solution. In other words, you need to figure out **HOW** you're going to solve the problem.

Designing your solution is, without a doubt, one of the hardest problem-solving steps. Because design is so important, we'll spend a lot of time throughout the book talking about how we go about doing this effectively. For now, let's just say that completing this step results in a set of classes, a set of objects, and a set of methods designed to solve the problem. The tricky part of this step is figuring out which classes and objects we need and how they'll interact with each other through their methods to solve the problem.

3. Write Test Cases
Our third problem-solving step helps you make sure you've actually solved the problem. Does your solution do what it's supposed to? It's reasonable to go back and run your program (which

[2] One of my sons consistently gives me a hard time because he saw me sitting in a chair staring into space once and told me I should be working. I told him I was working because I was thinking about how to implement the artificial intelligence in one of our company's commercial games. I've never heard the end of that one!

you'll write in the next step) a few times to make sure it solves the right problem. You're not really worried about making your program more efficient here; you just want to make sure it's correct. One way to try to make sure a program does what it's supposed to do is with *testing*, which is running a program with a set of selected inputs and making sure the program gives the correct outputs. To accomplish testing, we'll write a set of *test cases* you run against your code to make sure it's working properly.

But how can you decide how to test your program if you haven't even written it yet? We'll talk a lot more about software testing throughout the book, but it turns out that you can build most if not all of your test cases just by knowing what it's supposed to do. And remember from Step 1 – if you don't actually know what your code is supposed to do, you have bigger problems than coming up with the test cases!

We'll also talk about additional test cases you might want to add to your test plan based on the actual code you write in Step 4, which means we'll regularly re-visit this step after Step 4 before moving on to running the test cases.

So we know we need to write test cases, and each test case has a set of inputs we want to use for the code being tested and the outputs we expect when we use those inputs. Choosing which inputs to use is the trick, of course, because most programs can accept a huge number of inputs; if we use all of them, testing will take a LONG time! So how do we pick which inputs to use? We'll cover that in more detail soon, but at a high level we can either use the list of requirements (the things our solution is supposed to do) to pick our inputs (typically called *black box testing*) or we can use the structure of our solution to pick our inputs (typically called *white box testing*). We'll actually use a combination of these two techniques in this book. Let's see how.

There are many kinds of testing we can do for software, but in this book we're going to use two specific kinds: *unit testing* and *functional testing*. We discuss classes in much more detail in the Chapter 4, but for now, think of a program as the application class plus a bunch of other classes that the application class uses to solve the problem. If we think of each of those classes as a small unit in our solution, it makes sense to think of the testing we do on each individual class as unit testing. For unit testing, we'll add the black box test cases when we complete this step the first time, then after we write the code we'll re-visit this step to add the white box test cases based on the structure of the code we added.

So how do we document those test cases so we can use them in Step 5 to actually test the code? There are a variety of ways we can do that. For most of the test cases in this book, we'll simply write this all down in a concise set of steps for each test case. We can then execute the test case each time we need to test our code by manually following the steps in the test case.

In practice, though, it's much more useful to automate our test cases so we can rerun them whenever we change our code without having to manually pound keys for each test case. That's where a tool like NUnit comes in. We can use NUnit to capture the inputs and expected outputs for each test case, then run the NUnit test cases whenever we need to. We won't use NUnit in this book, but we regularly use NUnit for testing in our company's game development work.

We're making this testing stuff sound pretty straightforward, but in the game domain there are lots of things that aren't nearly as simple to test. For example, we haven't talked at all about how we make sure game entities move properly graphically when we run the program. We haven't even talked about testing if our print rain message program prints the correct message to the screen. Those two examples have something in common – these kinds of tests are checking the overall functionality of the program rather than the behavior of specific classes in our solution. That's why this kind of testing is called – wait for it – functional testing.

Unfortunately, NUnit isn't a help to us here, because our functional tests need a person to watch the program while it runs to make sure it's behaving properly. That means we can't really automate these tests (there are functional testing tools available, but they're way more complicated than we want to take on here), so we have to manually run each functional test case every time. We still need to document the inputs and expected outputs for each of our functional tests, though; we'll show you the format we'll use for that soon.

Writing test cases probably sounds like a big job to you, but that's only because it IS a big job! Thoroughly testing your code is critical, though, because releasing buggy code can really hurt you and your company. This seems to be particularly true in the game industry; lots of people using office software will accept minor problems with that code, but gamers tend to be less forgiving when the game doesn't work exactly the way they think it should. We'll write both unit test cases and functional test cases as appropriate throughout this book.

4. Write the Code

This step is where you take your design and implement that design in a working C# program (computer programs are commonly called *code*). The hard part of this step is doing the detailed problem solving to figure out the precise steps you need to implement in your code to solve the problem. Certainly, you'll have to learn the required syntax for C# (just as you have to learn the grammar rules for a foreign language you try to learn), but syntax is easy. If you do a careful job figuring out the steps you need to accomplish, writing the code will be easier than actually designing the solution. One more time – the hard part in the process is figuring out the correct steps, not figuring out where all the semicolons go.

Before we discuss C# source code, though, let’s take a little time now to discuss *algorithms*. We’ll define an algorithm as "a step-by-step plan for solving a problem"; you might hear the algorithms we write below referred to as *pseudocode* as well. Rather than formally writing algorithms or pseudocode for our solutions, we'll implement the code directly because the code is really just an expression of the steps we need to take in the C# programming language. We will, however, have to figure out those steps in a very precise and detailed way. Because we don't know enough C# to write the code directly for this problem, let's actually work on a detailed algorithm instead to start practicing the thought process we'll end up going through when we write the code.

The important thing to remember is that code (okay, the algorithm in this case) needs to be very detailed – if you handed the algorithm to your crazy Uncle Leeroy, could he follow it correctly? One good technique for writing correct, detailed algorithms is to think of other, similar problems you've already solved and re-use parts of those solutions. As you solve more and more problems,

you'll develop a large collection of sub-problem solutions that you can borrow from as appropriate.

OK, let's continue with our example of looking for the Queen of Hearts. We would probably solve this problem by using an object for the deck of cards, with a method that lets us search for the Queen of Hearts, so let's assume we're trying to actually figure out how to provide the method. Quick comment – no matter what design methodology you use (object-oriented or otherwise), sooner or later you have to figure out HOW to build the parts of your problem solution; that's where algorithms come in! Let's try to write an algorithm to solve the problem. Your first try might look something like this:

```
Look through the deck
If you find the Queen of Hearts, say so
```

What do you think? Is this a good algorithm? Stop nodding your head yes! This is not nearly detailed enough to be useful. Your crazy Uncle Leeroy has no idea how to "Look through the deck." Let's try to make the algorithm more detailed:

```
If the top card is the Queen of Hearts, say so
Otherwise, if the second card is the Queen of Hearts,
    say so
    Otherwise, if the third card is the Queen of
        Hearts, say so
        . . .
```

Well, this is better, but now it looks like our algorithm will be REALLY long (52 steps for a deck of 52 cards). Also, our algorithm has to know exactly how many cards are in the deck. However, we can look at our algorithm and see that we're doing the "same" thing (looking at a card) many times. Because we're doing this repeatedly, we can come up with a much cleaner algorithm:

```
While there are more cards in the deck
    If the top card is the Queen of Hearts, say so and
        discard the Queen
    Otherwise, discard the top card
```

We now have a correct, detailed algorithm that even Uncle Leeroy can follow, and it doesn't make any assumptions about the number of cards in the deck.

When we write our code, we use three types of *control structures* - *sequence*, *selection*, and *iteration*. Sequence simply means that steps in the solution are executed in sequence (e.g., one after the other, in order). Selection means that steps in the solution are executed based on some selection, or choice. Iteration means that certain steps in the solution can be executed more than once (iteratively). We'll look at these control structures and how we use them in C# in more detail in a few chapters, but you should understand that we simply use these structures as building blocks to develop our code (no matter what language we're programming in).

5. Test the Code

Finally, the last step in our problem-solving process. Now that you've implemented your solution, you run your test cases to make sure the program actually behaves the way you expected it to (and the way it's supposed to). Now, you may be surprised to discover that your code may not always work correctly the first time you try it! That's when it's time to *debug* – to find and fix the problems you discover as you test your code. Debugging is a part of EVERY programmer's life, so we'll talk about debugging techniques throughout the book as well.

2.13. Sequence Control Structure

The simplest problem solutions just start at the beginning and go to the end, one step at a time – that's what our print rain message program does. Let's take a look at another example:

Example 2.1. Reading In and Printing Out a Band Name

Problem Description: Write an algorithm that will read in the name of a band, then print that name out.

We know, you want to start slinging code, and we're still talking about algorithms. Don't worry, we'll get there soon!

Here's one way we could solve this problem:

```
Prompt for and read in band name
Print out band name
```

Note that we have to make sure we ask (prompt) for the band name before reading it in; if you walked up to someone and simply stared at them, would they know you were waiting for them to give you the name of a band? Try it and you'll see what we mean.

For many people, a "picture" of their solution helps clarify the steps and may also help them find errors in their solution. We'd therefore also like a way to represent our algorithms pictorially. One popular way to pictorially represent problem solutions is by using flowcharts, which show how the solution "flows" from one step to the next. We'll use the same idea in something called a *Control Flow Graph (CFG)*, which captures the same information as a flowchart without requiring a bunch of special symbols. A CFG graphically captures the way control flows through an algorithm (hence the name).

Let's look at the CFG for our algorithm. The CFG looks like:

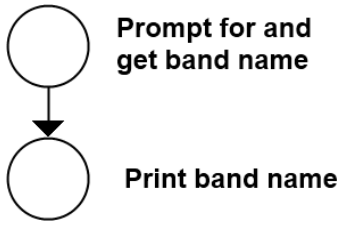

Figure 2.4. Sequence CFG

Because we have a sequence control structure (the program simply does one step after another), we have a sequence of *nodes* in the CFG. Each node represents a step in our algorithm; the *edges* in the CFG represent ways we can get from one node to another. In the algorithm above, we only have one way to get from the first step to the last step (since this is a sequential algorithm), so we simply add an edge from the first node to the last node.

The sequence control structure gives us our first building block for creating our problem solutions, but many of the problems we try to solve will also require us to make decisions or do things repetitively in our problem solution. We'll get to those control structures (selection and iteration, respectively) as we need them.

2.14. Testing Sequence Control Structures

Testing the code we write for the problem in Example 2.1. will be pretty easy – all we have to do is run it once to make sure it prompts for, gets, and correctly prints out a band name. Notice that this is a functional test case, since we're checking the functional behavior of the program rather than a specific class (unit) in the solution. We simply pick a band name and run the program once. Here's an example test case for this simple program:

Test Case 1
Checking Input and Output
Step 1. Input: The Clash for band name.
Expected Result:
```
Band Name : The Clash
```

For our functional test cases, each test case represents one complete execution of the program; no more, and no less. We only need one test case here because we only have to run the program once to fully test it.

2.15. Putting It All Together

Let's walk through the entire problem-solving process for the print rain message program. Here's the problem description:

Print the lines:

```
Hello, world
Chinese Democracy is done and it's November
Is it raining?
```

to the screen.

Understand the Problem
There's not much to worry about here, since the problem seems to be easy to understand.

Design a Solution

We can easily solve this problem using an application class that prints out each line in the message, so that's what we'll do here. Don't worry; the design step will become more interesting soon! We're just going to use a single class (`Program`) with a single method (`Main`), so we can move on to the next step.

Write Test Cases

Since this program won't have any user input, all we have to do is run it to make sure it prints out the required message.

Test Case 1

Checking Message

Step 1. Input: None.

Expected Result:

```
Hello, world
Chinese Democracy is done and it's November
Is it raining?
```

Write the Code

Here's the completed code for the program:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PrintRainMessage
{
    /// <summary>
    /// A class to print a rain message
    /// </summary>
    class Program
    {
        /// <summary>
        /// Prints a rain message
        /// </summary>
        /// <param name="args">command-line arguments</param>
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, world");
            Console.WriteLine("Chinese Democracy is done and it's November");
            Console.WriteLine("Is it raining?");
            Console.WriteLine();
        }
    }
}
```

Figure 2.5. Program.cs

Test the Code

Now we simply run our program to make sure we get the results we expected. Figure 2.6 has a screen snap of the output when we run Test Case 1.

And that's it – we've used our five problem-solving steps to complete our first C# program!

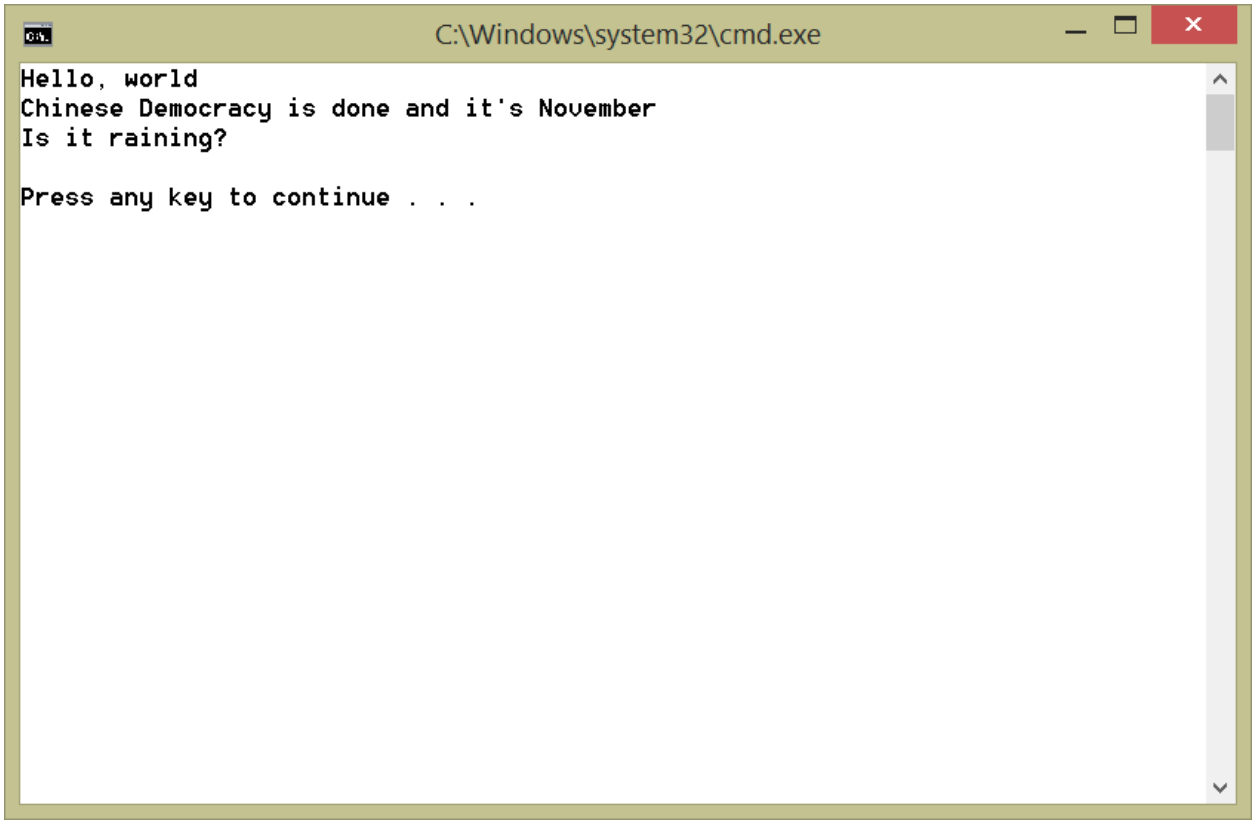

Figure 2.6. Test Case 1: Checking Message