

## Chapter 2

# Programs, Data, and Pretty Colors

### In this chapter:

Introduction .....	19
Making a Game Display .....	20
Controlling Color .....	24
Conclusion .....	39
Pop Quiz .....	40

- Explore how games actually work.
- See how data is stored in a program.
- Discover how colors are managed on computers.
- Find out about classes and methods.
- Write some code that controls color.
- Write some code that makes decisions.
- Create a funky color-changing mood light.

## Introduction

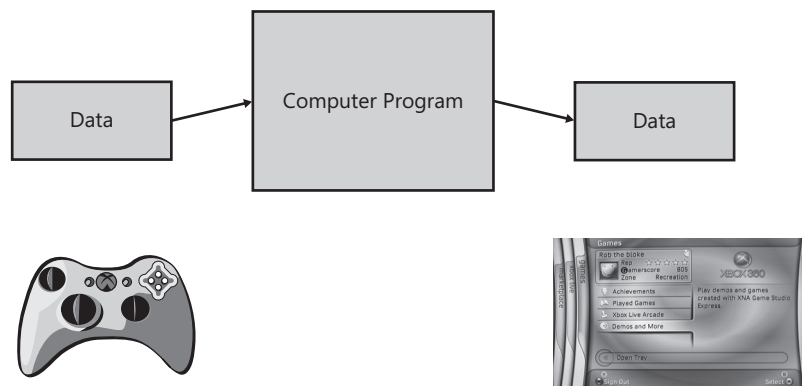
You now know how to create an XNA program and run it on an Xbox 360 or PC. The program only turns the screen blue, but you could call it a start. Next, you are going to figure out how game programs are constructed. Then you'll play with colors and find out how XNA stores color information and how C# stores data.

### Program Project: A Mood Light

Your first project is going to be a program that turns an Xbox display (the bigger the better) into a mood light. These are the things that they have on spaceships, where a chandelier would actually not work very well. Instead, the spaceship will have a panel on the wall that can be set to glow in different colors and brightness levels or perhaps even change color over time. This is probably not a very efficient way of lighting a building—you are using one of the most powerful game consoles ever made to replace a lamp—but it will be a fun exercise and may even lead to a game idea or two along the way.

## 20 Part I Getting Started

Before going any farther, you need to consider what a game program does. Computer programs in general read data in, do something with it, and then send data out. This is true whether the computer is working out the company wages or timing the ignition spark in a car engine. Figure 2-1 shows how this works with respect to games programs. The gamepad provides the input data to the game, and the display screen shows the output.



**Figure 2-1** An Xbox game as a computer program

Later versions of games might have other inputs and outputs too; for example, if you are playing on Xbox Live, your console is receiving information about other players in your networked game. For now, start by considering only the output from your game. Later you'll take a look at where the input values come from.

## Making a Game Display

To see how a game program can produce a display, you need to look inside one of the C# programs that XNA built. At the end of Chapter 1, “Computers, Xboxes, C#, XNA, and You,” you used XNA Game Studio Express to create a game program. Now you are going to take a look at this program and discover how it works.

The file that contains the game behavior is called, not surprisingly, **Game1.cs**. The name **Game1** was generated automatically when the project was created; the **.cs** part is the *file extension* for C# programs. If you want to take a look inside this file, start up XNA Game Studio Express and open the solution you created in Chapter 1 from the Solution Explorer. You can find the Solution Explorer, as shown in Figure 2-2, in the top right-hand corner of the XNA Game Studio Express screen. If you double-click the name of the file that you want to work with, the file opens in the editing window.

If you take a look at the content of **Game1.cs**, which drew that impressive blue screen, you can see how the program works. The program code that XNA Game Studio Express created when you made an empty game contains the following method:

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.CornflowerBlue);
    // TODO: Add your drawing code here
    base.Draw(gameTime);
}
```

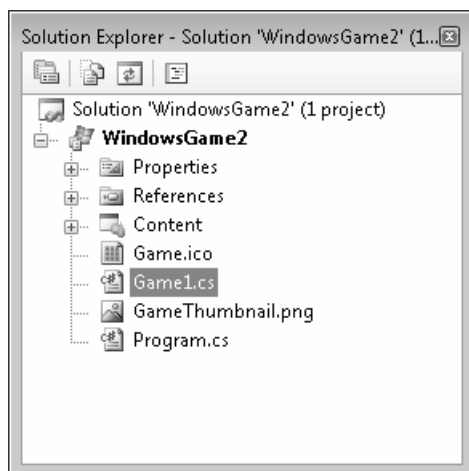


Figure 2-2 Solution Explorer

A *method* is a named part of a program. In this case the method has the name **Draw** (you can ignore the **protected override void** part for now). All you need to know at the moment is that when XNA wants to draw the screen, it will use this method. You can change what gets drawn by altering the content of this method. At the moment you just get a blue screen; if you look at the second line of the preceding code, you can see where the blue screen comes from.

## Statements in the Draw Method

The **Draw** method contains a block of statements. C# programs are expressed as a series of statements that are separated by the semicolon (;). Each *statement* describes a single action that your program needs to do. There are a number of different kinds of statements; you'll discover new ones as you learn more about programming. The statements are organized into a single block. A *block* is a way to lump statements together. The start of a block is marked with an open curly bracket character (`{`), and the end of the block is marked with a closing curly bracket (`}`). These curly kinds of brackets are sometimes called *braces*. The C# compiler, which is trying to convert the program text into something that the Xbox can actually run, will notice and complain if you use the wrong kind of bracket.

In the previous code, there is also a *comment*. Comments are ignored by the compiler; they let you put text into your program to describe the program or remind you to do things. In the previous code, the comment is a "TODO," which tells the programmer they need to do something. In this case, the programmer must add drawing statements at that position in the program file.

**The Great Programmer Speaks: Comments Are Cool** Our Great Programmer likes comments. She says that a well-written program is like a story in the way that the purpose of each part is described. She says that she will be looking at our code and making sure that we put the right kind of comments in.

From the point of view of changing the color of your screen, the statement that is most interesting is this one:

```
graphics.GraphicsDevice.Clear(Color.CornflowerBlue);
```

**Clear** is a method that is part of XNA. You will see precisely how it fits into the framework later; for now, all you need to know is that the **Clear** method is given something that describes a color, and the method clears the screen to that color. At the moment, you are sending the **Clear** method the color **CornflowerBlue**, and it is clearing the screen to that color. If you want a different color, you just have to send a different value into **Clear**:

```
graphics.GraphicsDevice.Clear(Color.Red);
```

If you change the color as shown in the previous line and run the program, you should see that the screen is now set to red.

**Sample Code: Red Screen of Anger** The sample project in the 01 MoodLight Red Screen directory in the resources for this chapter will draw a red screen for you. You could run this when you felt particularly angry. You can change the color that you want to display by changing the colors used in the **Update** method; there are some comments in the code to help you with this.

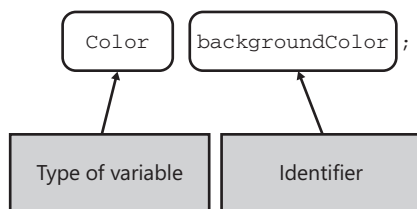
You can set the background color to a range of preset ones, but you can also design colors of your own, which brings us to our first project.

## Creating Your Own Color Values

You have seen that XNA has a set of colors built in, including one with the strange name of **Teal** (it is actually a rather boring blue/green). However, you want to make your own colors and use these in your program.

### Storing Color Values

A particular color is represented by a structure that holds the red, green, and blue intensity values. A *structure* is used to hold a number of related data items in the same way that you might write your name, address, and phone number on a piece of paper. You want to create your own colors, and you need somewhere to store the color values you create. In programming terms, this is called *declaring a variable*. Figure 2-3 shows the anatomy of the statement that declares a variable to hold a value that represents a color.



**Figure 2-3** Declaring a Color variable called backgroundColor

The *type* of the variable is set as `Color`. This determines what you can put in your variable. Having seen this declaration, the C# compiler knows that you want to create a location with the name `backgroundCo1or` in the Xbox memory, which can hold color information. In programming terms, the name of a variable is called an *identifier*. The word `backgroundCo1or` is an identifier that I've invented. When you create something for you to use in a C# program, you have to think up an identifier for it. An identifier is made up of numbers and letters and must start with a letter. The identifier should describe what you are going to use the thing for; in this program, you are storing the color that is going to be used for the background, so it can be given the identifier `backgroundCo1or`.



**Note** The C# compiler uses the type of a variable to make sure that a program never tries to do something that would be stupid. The only thing that you can put in a `Color` variable is color information. If the program tries to put something else in the `backgroundCo1or` variable, such as a player name, then the program would fail to compile. This is rather like real life, where an attempt to put an elephant in a camera case would be similarly unsuccessful.

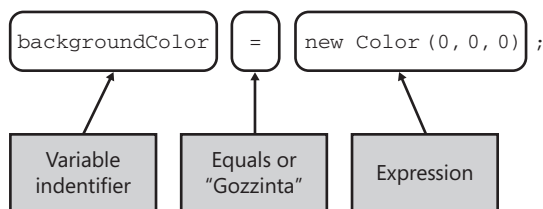
**The Great Programmer Speaks: Pick Useful Identifiers** Our Great Programmer says that there should be a special place in hell reserved for programmers who create identifiers like `X24` or `Cheese` or `count`. She says that these tell a reader of the program code nothing about what the variable is being used for. She really likes identifiers like `CarSpeed`, `backgroundCo1or`, and `accountBalance`.

## Setting a Color Value

You now have a variable that can hold the color of your background. At the moment it is not set to anything useful. So next you have to write a statement that will cause the game program to put a value into this variable. You start by creating a new `Color` that contains a particular amount of red, blue, and green. Figure 2-4 shows the anatomy of an assignment that makes a new `Color` value and then places it in the variable.

The thing that is going to be assigned is on the right-hand side of the equals. In this case you are making a new `Color` value. Don't get confused with an equals that might be used to compare two things. You should regard the equals above as being what I call a "gozzinta" operator. The value on the right of the equals "goes into" the variable on the left. You'll investigate how to compare things later. Now that you have your variable, you can use it in the game program.

## 24 Part I Getting Started



**Figure 2-4** Assigning a new color value to `backgroundColor`

```
graphics.GraphicsDevice.Clear(backgroundColor);
```

This statement calls the `Clear` method and feeds it the value of `backgroundColor`. This will cause the screen to be cleared to the new color you created. If you put these statements together, you'll get a `Draw` method that creates the `backgroundColor` variable, sets it to a value, and then clears the screen using it.

```
protected override void Draw(GameTime gameTime)
{
    Color backgroundColor;
    backgroundColor = new Color(0,0,0);
    graphics.GraphicsDevice.Clear(backgroundColor);
    base.Draw(gameTime);
}
```

If you want to find out what color you get if you make one with no red, no green, and no blue, you can run a program that uses this `Draw` method. But I don't think I'm giving too much away when I tell you that this would produce a black screen. The actual color values are given in the order red, green, and blue, and each must be in the range 0 to 255 (you'll see why this is later). By using different values when you set the `Color`, you can experiment with different displays. The color combinations obey all the rules of color combinations (for light rather than for paint) that you would expect.

```
backgroundColor = new Color(255, 255, 0);
```

This statement would set `backgroundColor` to a color value that has the red and green values at maximum, which would be displayed as yellow.

**Sample Code: Yellow Screen of Peril** The sample project "02 MoodLight Yellow Background" creates a yellow background color and fills the screen with it. You can change the numbers in the `Update` method to make any color you like.

## Controlling Color

At this point you can see that you add C# statements to the `Draw` method to change what is drawn on the screen. You also know that XNA makes use of a `Color` structure to lump

together information that describes a particular color and that you can create your own `Color` variables that contain a specific amount of red, green, and blue. Finally, you have managed to make a program that uses a color variable to set the screen to any color you like.

Next you want the light to change color over time, to get a nice soothing mood light effect. This sounds like hard work, and like every great programmer, I really hate hard work, but actually it turns out to be quite easy. To discover how to do this, you have to find how XNA is connected to the game programs that you write. The way this works makes use of C# classes.

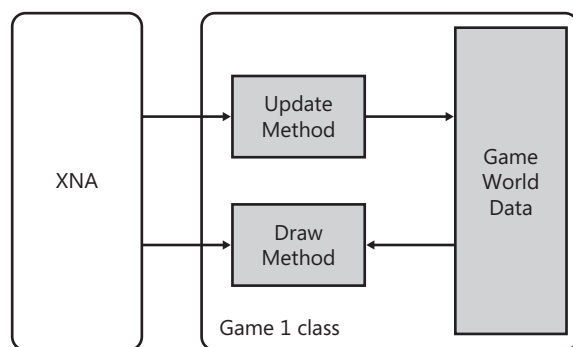
## Games and Classes

The game program is actually a *class* called `Game1`. A class is a collection of abilities (methods) and data (variables) that forms part of a program. You can put as much stuff as you like inside a single class. A class is usually constructed to look after one particular part of a system. Later on in this book you'll use classes called `GameSprite`, `Player`, and `Card`. In the commercial world, you'll find classes called `Receipt`, `Invoice`, and `StockItem`.

When XNA Game Studio Express created our project, it gave the game class the name `Game1`. You can rename this if you wish; you'll see how to do this later in the book.

## Classes and Behaviors

A behavior is something that a class can be asked to do. A particular method performs a particular behavior. You used the `Clear` behavior of the `GraphicsDevice`. When you use `Clear`, this causes the code in the `Clear` method to be obeyed to clear the screen. You don't need to know how `Clear` works; you just need to know that you can feed it with information to tell it what color you want to use. The `Game1` class provides `Update` and `Draw` behaviors (among others) so that XNA can ask `Game1` to update the state of the game and draw it on the display. Figure 2-5 shows how the `Update` and the `Draw` methods are part of the `Game1` class.



**Figure 2-5** The `Game1` class and XNA

The job of the `Update` method is to update the values in the game. The job of the `Draw` method is to use these values to draw the display. The XNA system will call `Draw` and `Update` at regular

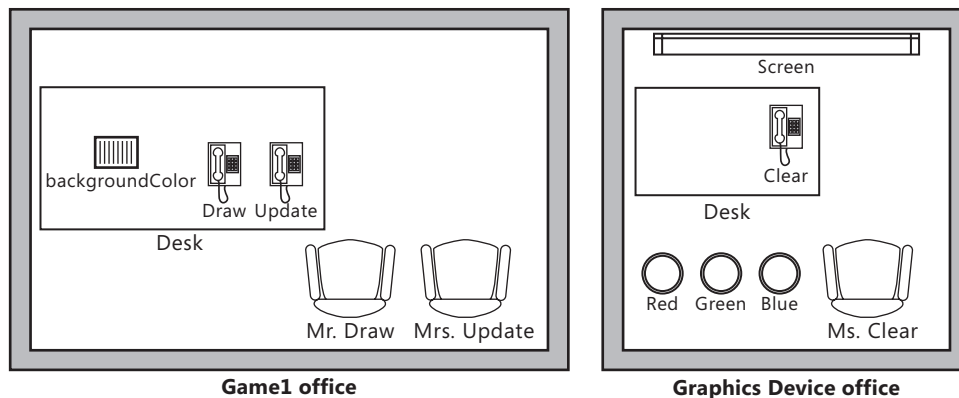
## 26 Part I Getting Started

intervals when the game is running. You have already used methods provided by other classes; you know that the `Clear` method can be called to clear the display to a particular color. `Draw` and `Update` are methods that you provide for use by XNA.

### Classes as Offices

You can think of `Update` and `Draw` as two people sitting in an office called `Game1`. Each has their own telephone. Every now and then, Mr. Draw's phone rings, and a voice on the other end of the line tells Mr. Draw that a sixtieth of a second has gone by. Mr. Draw then jumps up, gets the value of the background color from the desk, and then uses his phone to dial the number of the Ms. Clear in the `GraphicsDevice` office down the hall and asks her to clear the screen to that color. She has a set of paint cans and can fill the screen with any color that she is asked to use.

At a similar interval, the `Update` phone in the `Game1` office rings and tells Mrs. Update that a sixtieth of a second has gone by. She jumps up, goes up to the table in the office, and updates the information on the bits of paper on it. You can see how this would look in Figure 2-6.



**Figure 2-6** The `Game1` and `GraphicsDevice` classes as offices

The people/methods in our office/classes perform actions for each other, and data is just information that the class stores within itself. When a class wants to use a method, it calls it.

In our first version of the `Game1` class, the information on the table will be the color that Mr. Draw will use to color the graphics display. You change what happens when the screen is drawn by changing what Mr. Draw does (the content of the `Draw` method). You change what happens when the game itself is updated by changing what Mrs. Update does (the content of the `Update` method).

Note that nobody has to know exactly how the other methods work. Mr. Draw has no idea about cans of paint and displays, but he does know that if he asks Ms. Clear to clear with

yellow paint, this will result in a yellow screen being drawn. A call of a method is equivalent to calling up someone in an office and asking them to perform their task.

## Game World Data

You've seen that the actual state of the game is also held in the **Game1** class. In a driving game this state would include the speed of the car the player is driving, the car position on the track, and the position and speed of the other cars. This could be called the game world data. The game world data that you are going to use in the mood light is simply the red, green, and blue intensity values that will be used to color the screen. These variables can then be used by methods in the class.

```
class Game1 {  
  
    // The Game World - our color values  
    byte redIntensity;  
    byte greenIntensity;  
    byte blueIntensity;  
    // TODO: Draw method goes here  
    // TODO: Update method goes here  
}
```

This code declares three variables inside the **Game1** class. These are part of the class; they are often called *members* of the class and can be used by any methods that are also members of the class. They have the identifiers **redIntensity**, **greenIntensity**, and **blueIntensity**. You can think of these as separate pieces of paper on the desk in the **Game1** office. Figure 2-7 shows how a class can contain members.

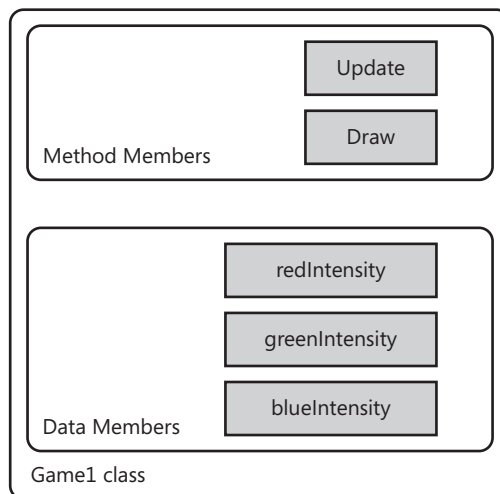


Figure 2-7 The **Game1** class and its members

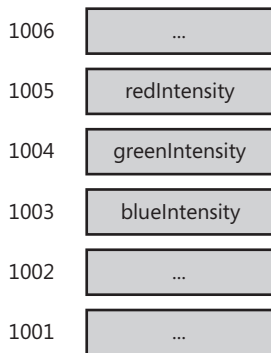
## 28 Part I Getting Started

There are two kinds of members: methods (which do something) and data (which hold information). The `Game1` class you are working on has both kinds of member; it has the `Draw` method and the `Update` method and also the three data members, which are going to be used to hold the color values for the changing background. The data members are of type `byte`.

If you refer back to Figure 2-3, you can see that a declaration is the type of the variable, followed by the identifier. Previously you have declared variables of type `Color` that can represent a color. Now you are using another type that can represent a numeric value.

### Storing Data in Computer Memory

The data for each color intensity is being held in a variable of type `byte`. The byte type is interesting because it uses 8 bits of computer memory to hold the value that it is trying to represent. Computer memory is actually a huge number of such locations, each of which is 1 byte in size. The Xbox 360 has 512 megabytes of memory. This means that the memory inside the console has about 512 million storage locations, each of which can hold a single byte value. The memory is addressed by number, and the compiler generates a program that uses a particular memory location when it accesses a particular variable. Figure 2-8 shows how this might work. The compiler has decided that `blueIntensity` is to be held in memory byte number 1003, `greenIntensity` in memory byte number 1004, and so on.



**Figure 2-8** Storing the color intensity values in memory

When the program runs, the statements that work with `redIntensity`, `blueIntensity` and `greenIntensity` will be directed to these locations in memory. Each data type uses up a particular amount of computer memory; a `byte` uses a single memory location. The `Color` type uses at least 4 bytes of memory; other types can use a lot more. When the program needs to hold a `Color` value, the compiler will allocate a number of adjacent memory locations.



**Note** In XNA you never have to worry about precisely where the compiler chooses to put things. These issues are managed automatically and hidden from the programs. In fact, the way things really work is a little more complex than the explanation here, but for now it's important for you to remember that computer data is held in memory locations of a particular size and that a particular number of memory locations are available for a program to use.

The same memory locations that store data can also be used to hold program instructions. When an Xbox game is running, it might be that half the memory space holds the game program code (the methods) and the other half the data that is being used (the variables). When a game is showing the dreaded “Loading” screen, the Xbox is actually transferring program code and data values from the game disk into its memory.

## Drawing by Using Our Color Variables

The color variables that you have created will represent the amounts of red, green, and blue that the mood light will have. You can use them in your **Draw** method to create the color to be used to clear the screen.

```
class Game1 {  
  
    // The Game World - our color values  
    byte redIntensity;  
    byte greenIntensity;  
    byte blueIntensity;  
    protected override void Draw(GameTime gameTime)  
    {  
        Color backgroundColor;  
        backgroundColor =  
            new Color(redIntensity, greenIntensity, blueIntensity);  
        graphics.GraphicsDevice.Clear(backgroundColor);  
        base.Draw(gameTime);  
    }  
    // TODO: Update method goes here  
}
```

This **Draw** method looks very much like the previous one, except that it uses member variables to define the color that is created rather than specifying particular values. Note that the actual assignment has been spread over two lines. The C# compiler is quite happy with this.

**The Great Programmer Speaks: Don't Try to Fit Everything on One Line** Our Great Programmer is very keen on sensible program layout. This means not letting program lines extend off the end of the page. She says that if the line gets too long, you should break it at a sensible point (not in the middle of an identifier) and then continue on the next line, slightly indented. She has personally checked all the program listings in this book to make sure that the layout meets her exacting requirements.

## Updating Your Colors

When the program starts, the values of byte data members are automatically set to 0. If you run a program with the **Draw** method given previously, you'll see that the screen just goes black, as a color with all the intensity values set at 0 is created and then used to clear the display. What you now need to do is get control of the **Update** process and change the colors. When an empty project is created, XNA Game Studio will create an empty **Update** method that just contains a TODO comment that tells the programmer to add the required code.

**30**    **Part I Getting Started**

```
protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back==ButtonState.Pressed)
        this.Exit();
    // TODO: Add your update logic here
    base.Update(gameTime);
}
```

The **Update** method is rather similar to **Draw** but has an extra couple of statements in it, one of which starts with the word **if**. This is the part of the code that decides when the game should end. When you ran your program, you'll have noticed that pressing the Back button on the gamepad stops the game. These two statements are the ones that give that behavior.

The first statement says “if the back button on the gamepad for player 1 is pressed, do the next statement” and the second statement says “exit the program.” Put those together, and you get a behavior that means that when the **Update** method is called, if the Back button is pressed the program will exit. You are going to spend some time on conditions later, but for now just remember that if you delete these two lines from your program, it will be impossible to stop it via the Xbox gamepad. So don't.

You may be wondering who calls **Update** and how often. The answers at the moment are “the XNA engine” and “sixty times a second.” Whenever your game is active, it will need to update the game world. This has to happen repeatedly for a game to be any fun. The XNA engine calls the **Update** method to give it a chance to perform. In a full-blown game this will involve reading the gamepad, moving all the objects in the world, checking to see if any have collided, and so on. In the mood light, the **Update** method will just change the color values that **Draw** will use to draw the display.

**Update** and **Draw** are completely separate methods, and you don't know when they are called in relation to each other. The job of **Update** is to update the values that represent the game world. The job of **Draw** is to produce a display that shows a view of that game world. This is exactly how every game works, whether it is a driving game, where the game world contains the position and speed of the car and the layout of the landscape it is driving through, or a shoot-'em-up, where the game world contains the position of all the players and any bullets they may have fired. Once you understand this principle, you are well on the way to understanding how games work. The **Update** method sets values up, and the **Draw** method uses them. To start with, you are just going to make a mood light that gets steadily brighter over time, so the first version of the **Update** method will increase the value of the red, green, and blue intensities by 1 each time that it is called.

```
protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();
}
```

```
// Make each color brighter
redIntensity++;
greenIntensity++;
blueIntensity++;

base.Update(gameTime);
}
```

The **Update** method works by using the ++ *operator*. An operator is something in the program that tells the compiler that you want to perform an operation on a particular item. In this case you're using the operator ++ on each of the intensity variables. The thing an operator works on is called an *operand*. Sometimes operators work by combining operands, and sometimes they work on a single operand. The ++ operator works only on a single operand. The **Update** method uses it three times so that each color intensity gets one bigger. This means that each time the **Update** method is called, the display should get a little bit brighter.

If you run the program with this **Update** method, you'll see that the display does get steadily brighter for about 4 seconds. Then it goes black again. This does not seem right. One of the additions would seem to be making the value much smaller rather than increasing it. To understand why this is, you need to take a look at how numbers are stored in computers.

## Memory Overflow and Data Values

You have already seen that byte values are actually represented by 8 memory bits. Now you need to understand what this means and the problems that it can cause.

A *bit* is the smallest unit of data that you can have. A bit is either on or off, in other words, it can store just two different values. The two values are often referred to as true or false. Each value will be represented by a particular voltage in the memory of the Xbox, but you need need worry about that in detail.

Think of a bit as a coin on a table. The coin can be either heads or tails, that is, in one of two possible states. If you put a second coin on the table, the two coins in combination now have four possible states: head-head, head-tail, tail-head, and tail-tail. Each coin that you add to the table doubles the number of possible states (that is, when you add the coin, you can have all the previous states with the new coin on heads plus all the previous states with the new coin on tails).

If you do the math with eight coins, you find that they can occupy 256 different states. So 8 data bits can hold 256 different values. One of these values is 0 (all false), which means that the largest possible integer value a byte can hold is 255 (all true). When the ++ operator tries to increase the value of 255, it will produce the value of 256, which cannot be represented by 8 bits. The addition process should set the value of a ninth data bit to 1 so that it can represent the value of 256, but there is no ninth bit to set, so the other eight bits are cleared to 0. This causes the value to wrap around, which means that the value in the byte goes back to 0

## 32 Part I Getting Started

again. The result of this is that the screen goes from maximum brightness to minimum brightness in a single step. The technical name for this is *overflow*.

One very important thing to note here is that no error messages are produced. The computer doesn't "know" that it has done anything wrong. Sometimes if your program does something stupid, you'll get an error, and your program will stop. However, in this case the Xbox does not seem to notice that you have just fallen off the end of a byte and will continue to run. Your program may well do the wrong thing, though. This means that your program has a bug in it. When you create the finished mood light code, you need to make sure that the values never "wrap around" like this.



**Note** Note that you have not "run out of memory." Rather, the program has tried to put too much information in a single memory location. The Xbox can work with values much larger than 256; it does this by using multiple storage locations to hold a single item. As an example, you have seen that the information to describe a color fills at least four memory locations.

**The Great Programmer Speaks: The Computer Doesn't Care** Our Great Programmer finds it very amusing when people say "the stupid computer got it wrong." She says this is not what happens. What really happened was that the person who wrote the program did a bad job. She has been known to roll around on the floor laughing when people ask her, "But why didn't the computer notice it was wrong?" She knows that the computer really doesn't know or care what a program actually does. The job of the computer is to follow the instructions the program gives it. The job of the programmer is to write instructions that are correct in every scenario.

**Sample Code: Fade from Black** The sample project in the 03 MoodLight Fade Up directory in the source code resources for this chapter will perform the fade up shown previously. It will then wrap around to black as the values in the bytes overflow.

## Making a Proper Mood Light

The fade-up part of the mood light is very good, but you don't want it to suddenly change from white to black each time around. What you would like is for it to fade smoothly up and down. If you were telling Mrs. Update what to do, you would say something like this:

"Make the value of `redIntensity` bigger each time that you are called. When the value reaches 255, start making it smaller each time you are called until it reaches 0, at which point you should start making it bigger again. Do the same with blue and green."

Mrs. Update would think about this for a while and decide that she needs to keep track of two things for each color: the current intensity value (in the range 0 to 255) and something that

lets her remember whether she is counting up or counting down for that color. Then each time she is called, she can follow a sequence like this:

1. If you are counting up, increase the value of `redIntensity`.
2. If you are counting down, decrease the value of `redIntensity`.
3. If `redIntensity` is 255, change to counting down.
4. If `redIntensity` is 0, change to counting up.

This is an *algorithm*. It provides a sequence of operations that is used to solve a problem. In this case you wanted to make the value of `redIntensity` move up to 255 and down again in steps of one.

Of course, Mrs. Update is not a person but a C# method, so now you have to convert the previous steps into C#. The first thing you need to do is work out what data you need to store. You need the intensity value and also a way of remembering if you are counting up or down.

```
// The Game World - our color values
byte redIntensity = 0;
bool redCountingUp = true;
```

You have seen the `redIntensity` variable before; what you haven't seen is the way that you can set it to 0 when you declare it. The `redCountingUp` variable is new, though. It is of a new type (There are loads of different types, you'll be pleased to hear). This is the `bool` type, which is special because it can hold only two possible values: `true` or `false`. It allows programs to perform what is called *Boolean algebra*, which consists of calculations involving only the values `true` and `false`. Such calculations are usually used to drive decisions along the lines of "If `itIsRaining` is `true` and `robWillBeGoingOutside` is `true`, I should call the `takeMyUmbrella` method."

In this case the `bool` type is perfect since `redCountingUp` will be either `true` or `false` and nothing else. The program will use it to make decisions in the `Update` method so that it can behave according to the data. It is this ability to make decisions that makes computers truly useful in that they can change what they do in response to their situation. To make decisions in your programs, you have to use conditional statements.

## Making Decisions in Your Program

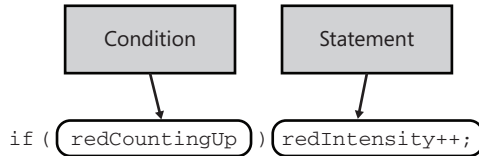
You have seen two kinds of statement so far. One calls a method to do something (you use this to call the `Clear` method), and the other changes the value of a variable (you use this to increase the intensity of our colors). Now you are going to use a conditional construction that can change what the program does depending on the particular situation.

### Creating Conditional Statements

Figure 2-9 shows how a conditional construction fits together. Conditional constructions start with the word `if`. This is followed by a condition in brackets. The condition will produce a

## 34 Part I Getting Started

Boolean result, which can be either true or false. You can use a variable of `bool` type directly here.



**Figure 2-9** The `if` condition in action

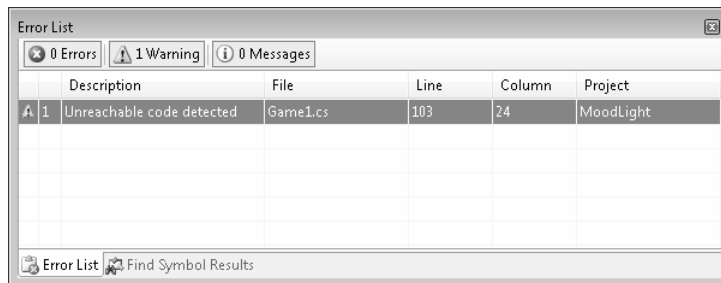
If the condition is `true` (that is, in this case the variable `redCountingUp` holds the value `true`), the statement following the condition is performed. The result of this is that when this statement is obeyed, the value of `redIntensity` will get bigger if the program is counting up. The condition can be any value that gives a Boolean result, including this rather stupid code:

```
if (true) redIntensity++;
```

This code is completely legal C# code and will compile with no problem. When the program runs, the condition will be `true`, and the statement will increase the red intensity value. This is very stupid code, though, as the test might as well not be there. You could also write the following:

```
if (false) redIntensity++;
```

In this code the statement following the condition will never be obeyed because the condition is always `false`. This C# code will compile okay, but if you look very closely at the XNA Game Studio 2.0 display, you might notice that it is trying to tell you something, as shown in Figure 2-10.



**Figure 2-10** Compiler warnings

If the Error window in Figure 2-10 is not displayed, you can open it by selecting Error List from the View menu. Alternatively, you can use the key combination `Ctrl+W` followed by `E`.

When the compiler has finished trying to convert your C# source code into a program that can be run on the computer, it will tell you how many mistakes it thinks it has found. There

are two kinds of mistakes. An *error* is a mistake that prevents what you've written being made into a program. Errors are really bad things like misspelled identifiers, using the wrong kind of brackets, and the like.

The other kind of mistake is called a *warning*. This is where the compiler thinks you may have done something stupid, but it does not prevent your program from running. Figure 2-10 shows the warning message for a program with a test for (`false`) in it.

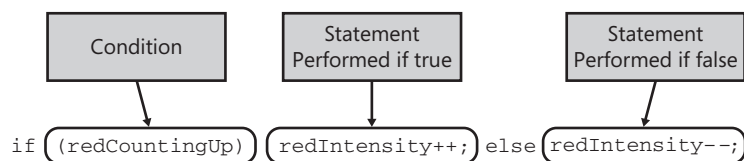
What the compiler is telling you is that it has managed to work out that the statement after the test will never be reached. This is because it's impossible for the value `false` to be true. The compiler is warning you that although the code is legal C# code, what it does might actually not be what you want.

**The Great Programmer Speaks: Warnings Should Always Be Heeded** Our Great Programmer has very strong opinions on compiler warnings; she reckons that your code should compile with no warnings at all. Warnings usually mean that your solution is imperfect in some way, and you should always take steps to investigate and resolve them.

## Adding an Else Part

The condition you have created is only half correct. If the program is not counting up, it must make the value of `redIntensity` smaller. You can use the `--` operator to do this, but you need to add extra code to the condition. You need to add an *else* part. Figure 2-11 shows another form of the `if` condition, with the `else` part added.

The two statements are separated by a new key word, `else`. The new code means that if the program is counting up (that is, `redCountingUp` is `true`), the value gets bigger, but if the program is counting down (that is, `redCountingUp` is `false`), the value gets smaller. The `else` part is optional; you must add one only if you need it.



**Figure 2-11** The `if` condition with an `else` part

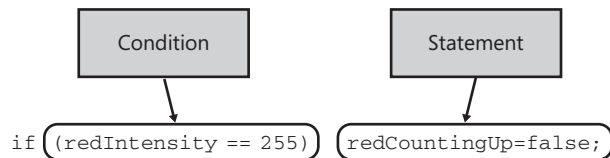
## Testing Values

The program must also manage the value in `redCountingUp` so that when it reaches the upper limit, it starts to count down, and when it reaches the lower limit, it starts to count up again. In other words,

1. When `redIntensity` reaches 255, set `redCountingUp` to `false`.
2. When `redIntensity` reaches 0, set `redCountingUp` to `true`.

## 36 Part I Getting Started

To do this you need another kind of condition, one that performs a comparison. Figure 2-12 shows how such comparisons are created. This performs the first of the previously mentioned two tests.



**Figure 2-12** Performing a comparison using the `if` condition

The key to understanding what is happening is the `==` comparison operator. When the program evaluates this condition, the values on the left and right of the `==` operator are compared. If they are the same, the result of the comparison is true, and the statement that follows the condition is performed. If they are different, the result of the comparison is false, and the statement that follows the comparison is ignored.

The sequence `==` is the comparison operator. It is completely different from the `=` operator, which we know as the “gozzinta.” It is important that you don’t get these two confused. Unfortunately, you have both a gozzinta and a comparison taking place in the `if` statement because you want to put a new value into `redCountingUp` if the comparison succeeds.

Fortunately, the compiler can usually detect when you use the wrong operator and produce a message. There are other comparison operators that can test to see if one value is greater or less than another; these are discussed later. An `if` statement that uses a condition can have an `else` part if required; it’s just that we don’t need one here. The final code to make our red intensity value move up and down ends up as follows:

```

if (redIntensity == 255) redCountingUp = false;
if (redIntensity == 0) redCountingUp = true;
if (redCountingUp) redIntensity++; else redIntensity--;

```

Note that the program needs a second test to change the direction of the counting when the bottom limit of the intensity value is reached. The tests are performed before the intensity value is updated so that if the very first value of `redIntensity` is either 0 or 255, the program will perform the wrong calculation.



**Note** Pay very careful attention to the previous three statements. Go back and read the original instructions to Mrs. Update and make sure that you are absolutely clear how these have been converted into C# statements that will perform the job.

## The Completed Mood Light

You now have the code that lets you create a smoothly pulsing mood light:

```
// The Game World - our color values
byte redIntensity=0;
bool redCountingUp = true;
byte greenIntensity = 0;
bool greenCountingUp = true;
byte blueIntensity=0;
bool blueCountingUp = true;
protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
        ButtonState.Pressed)
        this.Exit();
    // Update each color in turn
    if (redIntensity == 255) redCountingUp=false;
    if (redIntensity == 0) redCountingUp=true;
    if (redCountingUp) redIntensity++; else redIntensity--;

    if (greenIntensity == 255) greenCountingUp = false;
    if (greenIntensity == 0) greenCountingUp = true;
    if (greenCountingUp) greenIntensity++; else greenIntensity--;

    if (blueIntensity == 255) blueCountingUp = false;
    if (blueIntensity == 0) blueCountingUp = true;
    if (blueCountingUp) blueIntensity++; else blueIntensity--;
    base.Update(gameTime);
}
protected override void Draw(GameTime gameTime)
{
    Color backgroundColor;
    backgroundColor =
        new Color(redIntensity, greenIntensity, blueIntensity);
    graphics.GraphicsDevice.Clear(backgroundColor);
    base.Draw(gameTime);
}
```

These versions of **Update** and **Draw** will produce a program that smoothly fades the Xbox screen between black and white.

**Sample Code: Mood Light** The project in the 04 MoodLight directory in the source code resources for this chapter contains the previously mentioned **Update** and **Draw** methods and provides a smoothly changing mood light that goes from dark to light and back again.

## A Proper Funky Mood Light

Going from black to white and back is all very well, but it would be nice to add some additional variety to our light. It turns out that this is very easy to achieve. At the moment, the red,

## 38 Part I Getting Started

green, and blue intensities are all the same values, counting up from 0 to 255 and back down again. This just gives shades of gray. What you want is different combinations and the color intensities going up and down at different times. You can do this by changing the starting values of the intensity values and update directions:

```
byte redIntensity=0;
bool redCountingUp = true;
byte greenIntensity = 80;
bool greenCountingUp = false;
byte blueIntensity = 160;
bool blueCountingUp = true;
```

Rather than all the colors starting at 0 and counting up, the green value now starts at 80 and counts down, and the blue value starts at 160. This means that instead of just different shades of gray, you now have lots of other colors being presented. This provides a very groovy display. If you change the values in your program to the ones shown in the previous code, you can get a much more interesting-looking display. You can even try values of your own and see what they look like.

For a much longer-lasting display, you need to change the rate at which the three colors are updated. This is not actually very hard to do, so I've written an "Ultimate Mood Light" that you can take a look at.

**ENTER PREFIX TEXT IN EDD** The project in the 05 Ultimate Mood Light directory in the source code resources for this chapter contains a new version of **Update** that changes the red, green, and blue intensities at different speeds, resulting in a display that never seems to actually repeat (although it does eventually). Take a look at the code and see if you can understand how it works.

## Finding Program Bugs

Your younger brother has been reading this book and typing in the programs on his computer. He has just told you that the book is rubbish because the programs don't work. He has written an **Update** method and is complaining that for him, the red value only ever gets brighter. You ask him to show you the code and see this:

```
if (redIntensity == 255) redCountingUp=true;
if (redIntensity == 0) redCountingUp=true;
if (redCountingUp) redIntensity++; else redIntensity--;
```

At a first glance, it looks fine, and the C# compiler is quite happy that it's legal, but obviously it's not working. There is a bug in the program. Note that the bug is not there because the computer has made a mistake, so the instructions themselves must be faulty. You don't want to bother the Great Programmer, as she seems to be busy playing Halo on her Xbox, so you take a look, bearing in mind something she said recently.

**The Great Programmer Speaks: Run Programs by Hand to Find Bugs** A good way to find out what a program is doing is to behave like the computer and “run” the program yourself. By working through the statements by hand, keeping track of the variables, and making the changes to them that the program does, you can often find out what is wrong.

Your younger brother has actually made two mistakes in copying the program from these pages. See if you can find them by working through the statements. Think a bit about the answer before taking a look at my solution, shown in Figure 2-13:

```
if (redCountingUp) redIntensity++; else redIntensity++;
if (redIntensity == 255) redCountingUp=true;
if (redIntensity == 0) redCountingup=true;
```

This makes the intensity bigger when we should be counting down

This means we keep counting up when we hit the limit

**Figure 2-13** Finding the errors in the code

I marked out the errors; see if you can understand why they had the effect they did.

## Conclusion

You have learned a lot in this chapter. You now know the fundamentals of C# programs and the XNA framework. You have seen how to identify and create variables that store data and also how to write statements that change the values of these variables. You have seen that the data in a variable is held in a location in memory, which is a certain size and has a particular capacity. If you exceed this, the value will not fit and will be changed in unexpected ways.

You know that in C#, programs are broken down into classes, each class having things it can do (methods) and things it can hold (member variables). Classes are like offices, where workers (methods) can be asked to do things. You also know that that an XNA game is a particular kind of class that contains an **Update** method that's used by XNA to update the state of the game world and a **Draw** method that's used to draw the current state of the game world. You have seen how our programs can be made to make decisions and change what they do, depending on the values of the data they hold.

## Pop Quiz

Time for another pop quiz. Have a go at the questions before you move on. When you learn to program, you find that each step builds on the last, so it's important that you understand what is in this chapter before you move on to the next. Again, all the answers are either true or false, and you can work them out from this chapter and the Glossary.

1. A program is a sequence of variables.
2. Programs are always held in a file called `Program prog`.
3. An identifier is a name that you give to something you want to use in a program.
4. Methods tell the computer how to do something.
5. The `Draw` method updates the game.
6. A block of statements is made of wood.
7. The compiler checks code comments for accuracy and spelling.
8. A `Color` value is held as a single byte.
9. The type of a variable determines what kind of data can be put into it.
10. An identifier is a name built into `C#` to identify things.
11. A variable has an identifier and a type and holds values that your program wants to work with.
12. A variable of type `bool` can hold only the values 0 and 1.
13. Conditional statements start with the word **when**.
14. An `if` condition must have an `else` part.
15. An algorithm is like a recipe.
16. The operator `=` is used to compare two values and test if they are the same.
17. A class holds method members and data members.
18. A good identifier for a class would be `PlayGame`.
19. A good identifier for a method would be `Explode`.
20. A byte holds a single bit of data.
21. The `++` operator works between two operands.
22. The `C#` compiler detects if a variable overflows when the program is running.
23. Boolean values can be either `true` or `false`.