CHAPTER 1

YOUR FIRST ARDUINO SKETCH

Tools needed:

- Computer
- USB A to B cable

Parts needed:

- Arduino UNO (1)

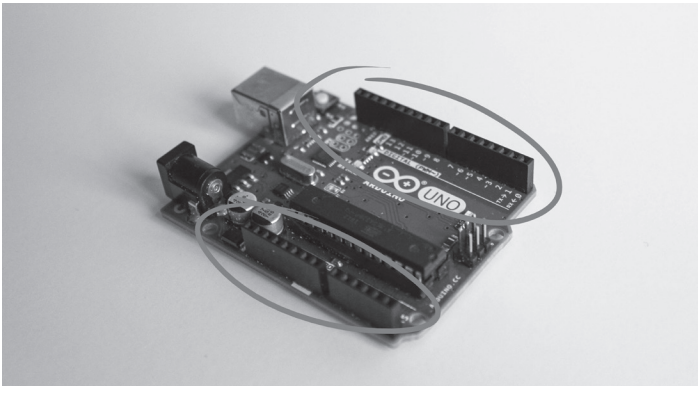
It's time to write your first Arduino sketch! A **sketch** is the Arduino term for the code that you upload to the device in order to make it behave how you want it to.

Here's how this is going to go: I'm going to give you a series of steps to follow in order to make the sketch work. You will follow them. Once you've completed all of them and gotten this first project to behave how we want it to, we'll double back, and I'll explain the code line-by-line. Alright? All right. Let's dive in.

For this first sketch as well as the next one, I'm going to be a little more thorough in my explanation than I will be in later projects. It's important to have a strong understanding of the basics before moving on to more complicated topics.

The first project we're going to complete is very simple—we are going to make a light on the Arduino blink on and off. Essentially, the Arduino will act like a switch that we can control using some code.

The little holes on top of the black bars lining the edges of the Arduino are called **female header pins**, or just **pins** for short. They are used to attach electronic circuitry to the Arduino by allowing wires to plug directly into them. Each of these pins has a number to identify it, which is written in white right next to the pin.



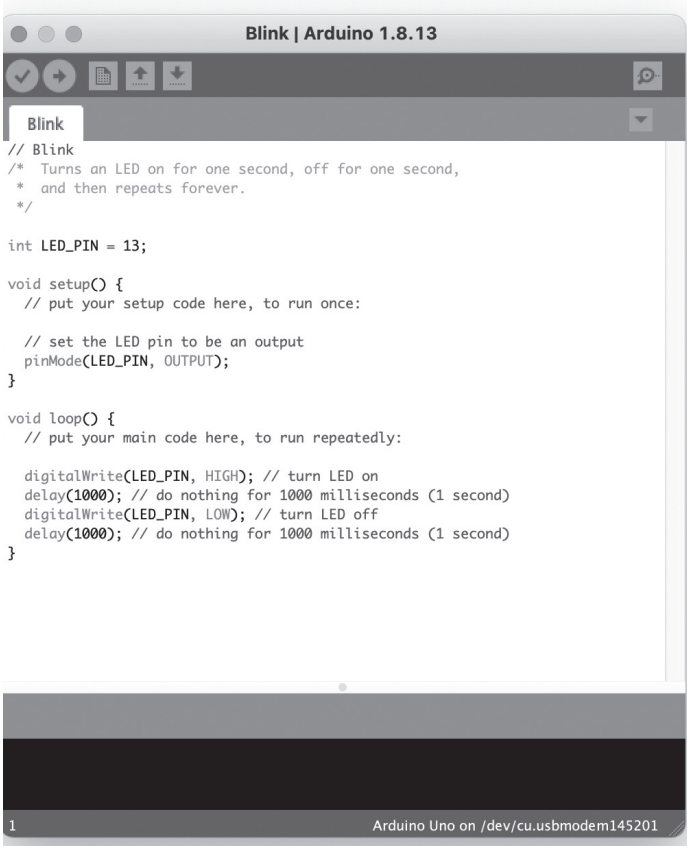
In this sketch, though, we won't be attaching any wires to anything—instead, we're going to work with the Arduino's **onboard LED**, which is an LED light that is hard-wired to pin 13 on the Arduino (it's labeled with an L on the surface of the Arduino board). Because it's already connected to pin 13, we won't need to add any extra electronics.

1. Open the Arduino desktop application. If you haven't installed it yet, you can do so at https://www.arduino.cc/en/Main/Software
2. The window that opens should look something like this:

1. This is called the Arduino Integrated Development Environment (IDE) and is where you'll be writing all of your Arduino sketches. If the window that opens *doesn't* look something like this, or if no window opens at all, try the following to fix it.

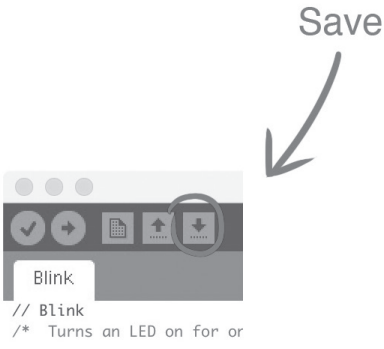
- Click the "New" icon in the sketch window. To find it, first look at the checkmark button at the top left of the window. The "New" button is two buttons to the right of that one. Clicking the "New" button opens a brand new sketch with the default code already written.
- If you can't get the Arduino desktop application to work on your computer, register for an Arduino Create account at https://store.arduino.cc/digital/create and install the plugin it asks you to. This will allow you to use Arduino from your web browser. The window looks a bit different in Arduino Create, so it might be a bit harder to follow along, but all of the same buttons will be there in some form or other.

2. Type the following code into the sketch **by hand**. The reason you are typing it in by hand rather than copy-pasting it is that writing code efficiently is difficult and takes practice, mostly due to the fact that accidentally mistyping and breaking your code is easy to do and often difficult to realize you've done, so it's in your interest to begin practicing your code-writing accuracy right now. This code is written in a programming language called C++, which is a popular and powerful language related to C and similar to Java in many ways.

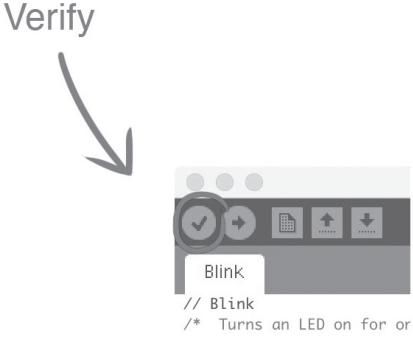


If you'd rather not type it in by hand, you can find all of the code used in this book at <web link>.

1. Go to File > Save (or press the button on the menu bar with an arrow pointing down) and save the sketch with the name "Blink" somewhere on your computer so you can easily find it again.



1. After you've saved, click the "Verify" button at the top left of the window (it looks like a checkmark) to verify the sketch.

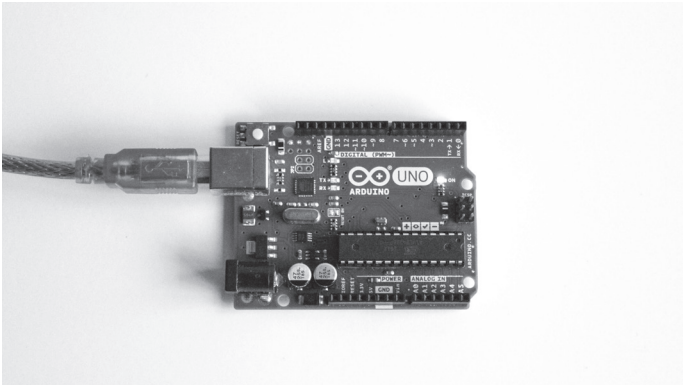


1. This will tell the computer to process your sketch to see if there are any errors. The words "Compiling sketch…" should momentarily appear on the turquoise bar below the text area. If the bar below the text window turns orange, highlights one of the lines in red, and gives you

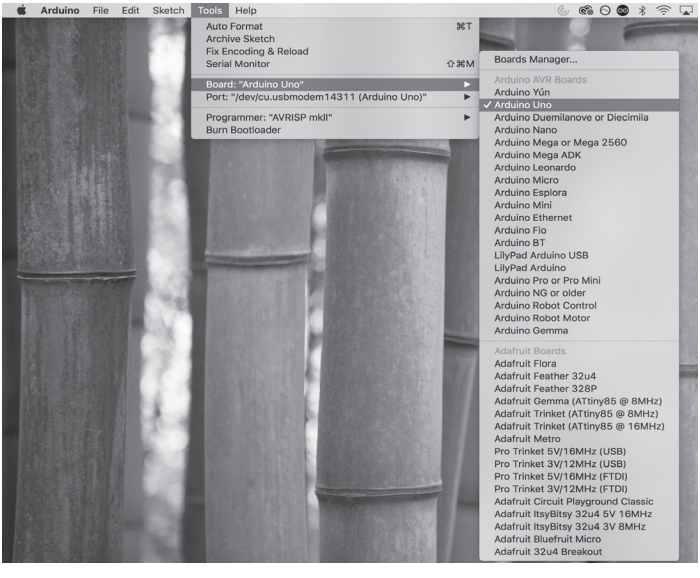
a strange error message in the place where it previously said, "compiling sketch," follow these troubleshooting steps; if it turns turquoise and says, "Done compiling," move on to step 5.

- In the event of a syntax error, the IDE should highlight a line of code in red. This indicates that the error is on or adjacent to that line (for instance, sometimes the error is on the line above). The error message that shows up written on the orange bar will provide a hint about what the error is. Sometimes the error message is complicated and hard to understand, but it is often useful in the event of a simple error.
- Ninety-nine times out of 100, Arduino coding errors are due to the omission of a semicolon. If money and circumstances permit it, I suggest you tattoo a semicolon on the back of your dominant hand, and maybe your forehead as well, to remind you of this. In the event of a semicolon error, the error message will read something resembling "expected ';' before '}' token" and highlight the line immediately after the one on which the semicolon is actually missing.
- Make sure that every line you've typed in is exactly identical to the code typed in the image above. Computer code can be very finicky about accuracy.

2. If your window says, "Done compiling," then it's time to upload your sketch. Take your USB A to B cable, and plug the "B" end into your Arduino Uno. Hold off on connecting the "A" end to the computer for now.



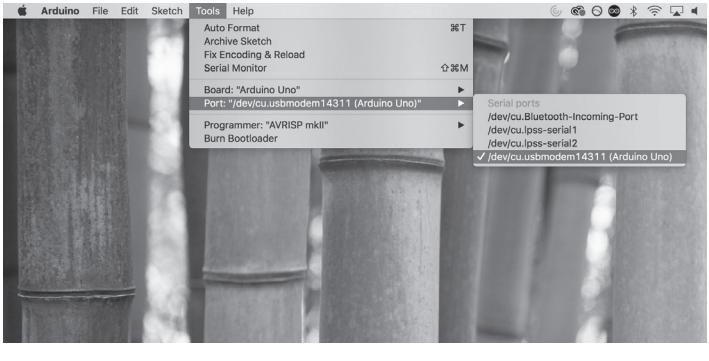
1. Now we have to make sure the computer knows what kind of Arduino board it's uploading to. Go to Tools > Board on the menu bar and select "Arduino Uno" from the rather lengthy list of possible boards.



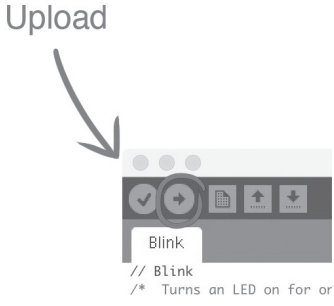
1. For the last step before we upload our code, we need to connect the Arduino to the computer and tell the computer which USB port the Arduino is attached to. The computer

will often connect to the correct port automatically, but let's make sure just in case. First, go to Tools>Port and look at the options it gives you. Take note of these options, then close the menu and plug the "A" (normal-looking) end of the USB cable into a USB port on your computer. Note that the port names on your computer may be different from the port names in this image.

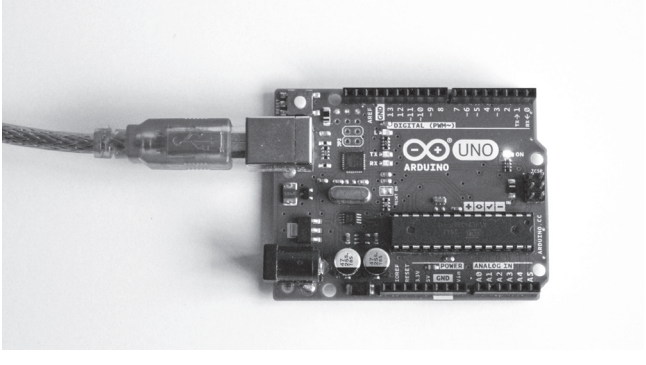
1. Now go back to Tools > Port. Notice how a new option has popped up, probably with the words "Arduino Uno" in parentheses next to it? Select that one.



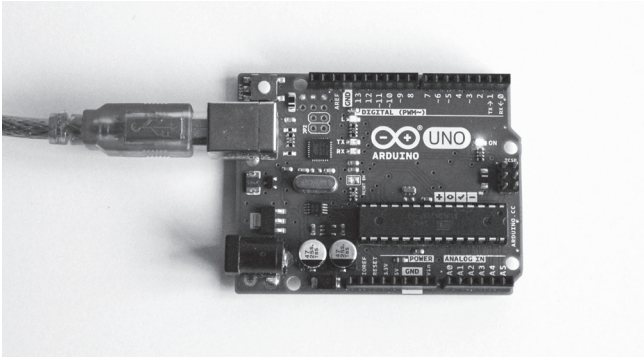
1. Upload the sketch by clicking the "Upload" button, which is directly to the right of the "Verify" button on the top left of the window and looks like an arrow pointing to the right.



1. When it's done uploading, take a close look at the top of your Arduino board. The LED should be blinking on for one second, off for one second, and then turning on and off again in a loop. If it is not doing so, refer to step 7c.



The light should blink off…



And then on, and then off again.

Congratulations! You've completed your first Arduino project! It doesn't do all that much, of course, but the things that make this simple little blinking light work make up the core principles of all Arduino programming.

Almost all of the projects we're going to work on after this find much of their foundation in this project. With that in mind, it's very important that we spend enough time on this project to understand it fully.

UNDERSTANDING THE CODE

Let's break it down line-by-line very, very carefully. The first line reads:

```
// Blink
```

This is what's called a **single-line comment**. A comment allows the coder to leave notes to themselves and others, label sections of code, and indicate the function of various sections of code.

To create a single-line comment, type two forward slashes one after the other. Everything on that line of code that appears after the two forward slashes will be grayed out by the IDE editor and ignored by the Arduino. In other words, anything you type in a comment after those two forward slashes have absolutely no effect on the code, no matter how profane or idiotic it is. (Keep in mind that if your comments are indeed profane or idiotic, they may still have an effect on anyone who reads your code.)

The next three lines are another type of comment:

```
/*  Turns an LED on for one second, off
for one second,
 *  and then repeats forever.
 */
```

This is a **multi-line comment**. A multi-line comment functions (or rather does not function at all) exactly the same as a single-line comment but can stretch across multiple lines. In order to create a multi-line comment, you have to open it with /* and close it with */. Anything after /* and before */, regardless of how many lines of code are between them, is ignored by the Arduino.

Note that when you begin typing a single line comment, the IDE automatically adds asterisks at the start of each new line until you close the comment to make it look pretty—these are not necessary for the Arduino to recognize them

as comments. The code would work exactly the same if it were written like this:

```
/*  Turns an LED on for one second, off for one second,
and then repeats forever.
*/
```

You could even move the "end comment" marker to the end of the second line, like so:

```
/*  Turns an LED on for one second, off for one second,
and then repeats forever. */
```

There's a blank line after this comment (all blank lines are ignored by the Arduino), and then we have our first line of code that the Arduino will actually read. This is where things start to get interesting. The line reads:

```
int LED_PIN = 13;
```

This line creates, or rather, **declares** a variable. **Variables** are used to store values. However, any given variable can only store a certain **type** of value. What type of value it can store is determined when you declare the variable.

In this line, we declare a variable of the type `int`, name it LED_PIN, and assign it the value 13. When the Arduino reads this line of code, it creates a little space in its memory

to store `int` values and puts the number 13 in that space. For the remainder of the code, when it encounters the term LED_PIN, it will check to see what value is stored in the space it made for LED_PIN. In this sketch, that value will always be 13 because that's the digital pin that the onboard LED is connected to.

The int variable type is short for "integer" and indicates that only integer values can be stored inside the variable. An int variable can hold any integer value, an integer being a number with no decimal points, such as 5, 0, 28, or 10,000. There is a size limit on the int type, however—it can only be used to store integer values between -32,768 and 32,767.

The second part of the statement, LED_PIN, is the name you're choosing for the variable. This can be almost anything, with a few restrictions. For example, you can't start your variable names with numbers, and they can't have any spaces or strange characters in them.

For variables that retain the same value for the entire duration of the code, the standard naming convention in C++, the programming language that Arduino programming is based on, is to make the name uppercase and to use underscores to separate words in the name.

You don't have to follow standard variable naming conventions if you don't want to, but it can be helpful to get in the habit of using them, so the code in this book will use standard conventions for variable naming, as well as other formatting. Regardless of how you name your variables, make sure any name you choose to use is descriptive enough about what it stores to allow you or someone else who might be reading your code to figure out what it's used for easily.

After LED_PIN, we have a single equals sign. When the Arduino sees an equals sign in the code, it knows that whatever number is on the right side of the equals is the number that should be stored in the variable named on the left side.

Finally, we have the value we're assigning to the variable, 13, followed by a semicolon (";"). **Do not forget the semicolon. Ninety-nine percent of all programming errors can be traced to forgotten semicolons.** The semicolon indicates to the compiler that a given statement has come to a close.

In summary, if you read the line

```
int LED _ PIN = 13;
```

as a sentence, it really just means "The integer called LED_PIN is equal to thirteen."

Next, let's first look at this entire block of code. A **block** is a portion of code bounded by "curly braces" (the '{' and '}' characters, respectively) that functions as a complete unit.

```
void setup() {
// put your setup code here, to run once:
// set the LED pin to be an output
 pinMode(LED _ PIN, OUTPUT);
}
```

Note that the information inside the block of code is indented—this is another formatting convention that the Arduino IDE encourages.

This entire block of code is called the **setup() function**. Every single Arduino sketch must have a setup() function. The role of setup() is to run code that initializes various things in the sketch. It runs all of the code between its open and closed "curly braces" one single time at the very beginning of the sketch and then never again until the Arduino is restarted. In its most basic form, the setup() function looks like this:

```
void setup() {

}
```

The setup() function above has nothing between its curly braces and thus does absolutely nothing. However, even if it does nothing, every sketch must have a setup() function, and every setup() function must look exactly as shown above, with a "void" initializer, the word "setup()" with the parentheses, and open and closed curly braces. Code may optionally go between the two curly braces, and there is almost always *something* between them.

In our code, we have three things between these brackets. The first two are just more comments.

```
// put your setup code here, to run once:

// set the LED pin to be an output
```

However, the last line is something new:

```
pinMode(LED _ PIN, OUTPUT);
```

This is something called a **function**. This particular function is called **pinMode()** and is built into the Arduino software. Functions are designed to, well, perform a specific function, and most have **parameters** that give them the information required to do their job. In the Arduino IDE, function names typically appear in orange, with the exception of setup()and loop(), which are special.

The function of pinMode() is basically to change one of the Arduino's settings. The Arduino UNO has an onboard LED that is connected to pin 13. We want to be able to turn this LED on and off by controlling whether power is moving through pin 13 or not. To do this, we must set the "mode" of pin 13 to OUTPUT. pinMode() allows us to set any pin to either the INPUT or OUTPUT mode, provided we tell it which pin to set the mode of and what mode to set it to. Thus, we give it the parameters by writing them in the parentheses that come after the function name like so:

```
pinMode(pin#, INPUT/OUTPUT);
```

In this case, the pin# is 13, and the mode is OUTPUT. However, we've already stored the pin number in the variable LED_PIN, so we can plug in the parameters as:

```
pinMode(LED _ PIN, OUTPUT);
```

The final block of code is called the **loop()** function:

```
void loop() {

// put your main code here, to run repeatedly:

 digitalWrite(LED_PIN, HIGH); // turn LED on

delay(1000); // do nothing for 1000 millisec-
onds (1 second)

digitalWrite(LED_PIN, LOW); // turn LED off

delay(1000); // do nothing for 1000 millisec-
onds (1 second)

}
```

The loop() function is another special function, like setup(), with the primary difference being that loop() will run all of the code between the curly braces from top to bottom, start at the top again once it reaches the bottom, and continue to loop this pattern until the Arduino is turned off. This is where most of the interesting parts of Arduino code can be found.

After the comment at the top of the block, we see a new function, **digitalWrite()**, as well as a single-line comment in a location we haven't seen one in before.

```
digitalWrite(LED_PIN, HIGH); // turn LED on
```

Single-line comments (or multi-line comments, for that matter) can begin anywhere on any line, even when that same line has code on it. Just keep in mind that anything at all,

code or otherwise, that is written to the right of the slashes will be ignored by the compiler.

```
digitalWrite() is another function much like
pinMode, but instead of changing a setting,
it controls whether one of the Arduino's pins
is high or low. Its parameters are:
```

```
digitalWrite(pin#, LOW/HIGH)
```

The first parameter, the pin#, is the pin you want to control the output of. The second parameter is the output power level you want to set that pin to. Setting this parameter to LOW makes the pin output zero volts. Setting it to HIGH makes it output five volts (or whatever the highest voltage your board can output is). If we have a component, such as an LED, connected to that same pin, this allows us to control whether power is flowing through it and thus whether the LED is operational.

Therefore, the statement

```
digitalWrite(LED _ PIN, HIGH); // turn LED on
```

turns on the onboard LED connected to pin 13 (LED_PIN) by applying 5V of power across it.

The next line is as follows:

```
delay(1000); // do nothing for 1000 millisec-
onds (1 second)
```

The third and final function of this sketch, **delay()**, is even simpler than the first two. delay() causes the Arduino to

pause everything it's doing and freeze for a given number of milliseconds (1 millisecond = 1/1000 second).

Why milliseconds? Milliseconds provide a higher degree of precision than seconds, and it's a unit of time that comes in very handy when using Arduino and electronics, which can process data many thousands of times faster than a human can. In fact, there is no variant of the delay() function built into Arduino coding that lets you use a larger unit of time than the millisecond, although there is one that lets you use a smaller unit of time. delay() has a single parameter: the number of milliseconds to delay everything for.

```
delay(milliseconds)
```

In our case, the parameter is 1000, which means that when the Arduino reaches this line of code, it will bring everything to a halt for 1000 milliseconds (AKA 1 second).

The last two lines of code inside the block look very similar to the first two:

```
digitalWrite(LED_PIN, LOW); // turn LED off
```

```
delay(1000); // do nothing for 1000 milliseconds
(1 second)
```

The only difference this time around is that instead of the second parameter for digitalWrite() being set to HIGH, it is now LOW, meaning that the Arduino will set the LED_PIN (pin 10) to LOW (0V), which will turn the LED off. It will then do nothing for a second. Once the Arduino has executed these final two lines, it will hit the end of the loop() function,

as indicated by the curly brace at the end of the line of code, then go back to the top of the loop() and do everything all over again.

That was a lot to take in, so let me rewrite all of the code we just went through in English, so it makes a little more sense. This will be in the form that the compiler will read the code in, so I'll skip all the comments.

```
An integer-type variable with the name LED_PIN now has the value 13.

Begin the setup function:

LED_PIN, which is pin 13, is an output.

End the setup function.

Begin the loop function:

Set LED_PIN (pin 13) to HIGH (5V).

Do nothing for 1000 milliseconds (1 second).

Set LED_PIN (pin 13) to HIGH (5V).

Do nothing for 1000 milliseconds (1 second).

Go back to the beginning of the loop function.
```

That's really it. The code is read line by line, from top to bottom, executing each command in chronological order. The ultimate function of this code is found in the loop function:

1. Set LED_PIN to HIGH, which turns on the LED
2. Do nothing for one second, during which time the LED remains on
3. Set LED_PIN to LOW, which turns off the LED
4. Do nothing for one second, during which time the LED remains off
5. Go back to step one.

This process you've just undergone—writing a series of commands to tell a computer or Arduino exactly what you want it to do—is all coding really is. If you can understand this sketch, you're more than prepared to move on ahead.

EXTENDED LEARNING

Now that you understand how this sketch works, it might be helpful and fun to play around with the code you've already written to get used to writing your own code. Here are several simple ways you can modify the code we wrote in this project to make it work in new and interesting ways.

CHANGING THE LENGTH OF THE DELAY

What if you want your LED to blink faster? Simple: change the number in the delay() functions. The smaller the delay, the less time will pass between the light blinking on and blinking off. The larger the delay, the more time will pass between the light blinking on and blinking off.

To add an interesting twist, you could give each of the delay() functions a different parameter. If you make the delay after the light is turned on shorter than the delay after the light is turned off, what happens?

The answer is that it will turn on for a brief moment and then turn off for longer. The light will become just a little blip. Do the opposite, making the "on" delay longer than the "off" delay, and the light will flicker off occasionally.

Try changing the delay values, clicking the "verify" button (the check mark), and uploading your new code to the Arduino to see what happens.

CHANGING THE BLINK PATTERN

We've already written a sketch that turns the LED on, turns it off, and repeats. But what if you wanted a different pattern? What if you wanted to turn it on for a long time, turn it off for a long time, then rapidly turn it on and off before going back to the beginning?

If you want to increase the complexity of your blink pattern, you need to add additional digitalWrite() and delay() functions, with the appropriate parameters, to loop(). Everything inside the loop() will repeat, so as long as the code inside it is valid, it can be as long and complicated as you want it to be.