



Learn to Code:

The Full Beginner's Guide

By Adam Dachis of Lifehacker.com

You can contact Adam Dachis, the author of this post, at adachis@lifehacker.com. You can also follow him on [Twitter](#) and [Facebook](#)

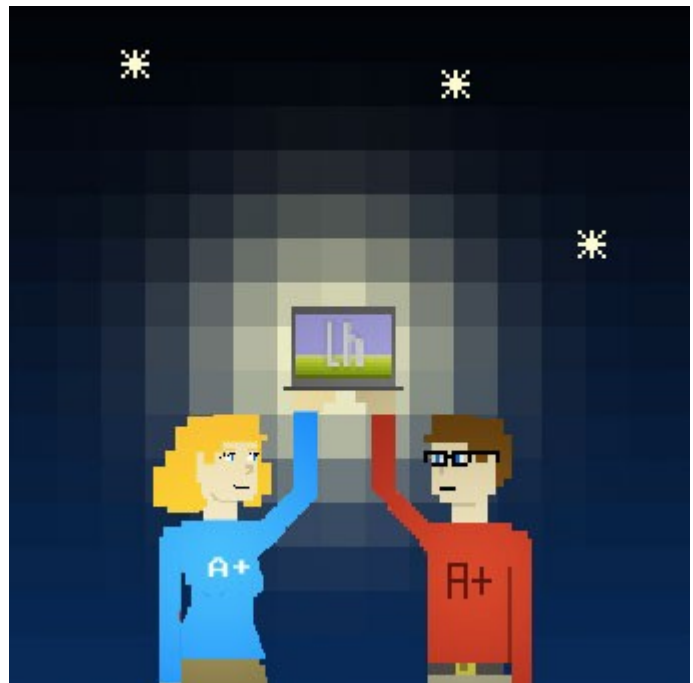
Learn How to Code Part I:

Variables and Basic Data Types

[Link to YouTube Video](#)

Want to learn how to code but don't know where to start? We've got you covered. We'll be teaching you the basics all week, and here's your first lesson.

Previously we've [provided you with some resources for learning to code](#) and [given you a broad overview of the process](#), but now it's time to get down to business: We're offering a short 101 course, step by step. You can't learn to code overnight (or in a week), but we've broken up the basics into a few lessons that will be released as the first four parts in our brand new **Lifehacker Night School** series. Each lesson will be video-based (as you can see above), but we'll also provide you with text notes and files that you can refer to as well. Each lesson is designed to be a manageable chunk of information that you can digest in under 15 minutes (and often much less). Although we're starting off our first programming lesson at 9:00 AM PST, the following three lessons will be released every day at 6:00 PM PST. Be sure to come back and visit us at the end of the day on Tuesday, Wednesday, and Thursday this week to finish learning the basics.



Our first lesson is going to be very simple and consist of learning about basic variables and data types. For the lessons in this series, we're going to use JavaScript as a model because it's a syntax that's pretty easy to understand and it's something that anyone with a text editor and a web browser can use. Because it's an [ECMA-based language](#), it makes understanding other ECMA-based languages (like [ActionScript](#)) much easier to learn. Even better, you'll find that knowing how to write JavaScript will make the transition to other [object-oriented programming languages](#) much easier. Basically, JavaScript is readily available to practically anyone with a computer and a browser, so we think it's a really good starting point. Once you have the basics down it should be easy to begin learning other languages.

Let's get started!

What Are Variables?

You can think of variables as labeled jars that store different types of data. While there are several kinds of variables, today we're only going to look at three:

- **String** - A string variable is a string of alphanumeric characters and allowed symbols that are contained within quotation marks. For example, "Hello world, I'm 102 years old today!" is an example of a string. Strings can also be contained within single quotes, which is useful if you want to have a string with a quotation like this: "I hate the snow," Laurel said.' Strings are basically used for storing text.
- **Number** - A number variable couldn't be more straightforward because all number variables store are numbers. You don't store them within quotes like strings. Instead, numbers can just be written as they are. If you want to store the number 9 in a variable, you just write 9.
- **Boolean** - A boolean variable is one of two things: *true* or *false*. This data type is kind of like an on and off switch, so you can ask true or false questions in your code. For example, you might ask "is the video currently playing?" The response you'd get would be a boolean variable. *True* would mean the video is currently playing and *false* would mean it is not.

So how do you put a variable to your code (or *declare* a variable, as it's more traditionally called)? In JavaScript, all you need to do is this:

```
myVariable = "Hello world!";
```

There are a few of things to notice here. First, the name `myVariable`. All programming languages have something called reserved words, which means you can't use them as variable names. What they varies, but if the name is sufficiently generic, there's a chance it could be a reserved word. To avoid using reserved words and screwing up your code, just decide on a naming scheme for your variables. I've put "my" in front of my example variable, but you'll probably want to come up with something else. Second, you'll notice a semicolon at the end of the line. A semicolon is like a period at the end of a sentence in many programming languages, and that is definitely the case in JavaScript. In nearly every situation, you need to end your code sentences with a semicolon so your computer doesn't get confused when reading it. The semicolon tells the computer, "Okay I'm all done with this statement." (Note: JavaScript is forgiving, and sometimes you can get away without the semicolon, but it's good practice.)

To Var or Not to Var

In JavaScript, you can define a variable as `myVariable = "something";` or `var myVariable = "something";`, the difference being the word `var` preceding the statement. When you're declaring variables in a script outside of a function, this distinction is pretty much irrelevant. When you're declaring a variable inside a function and *do not* use `var` this creates a global variable. Global variables can be accessed from anywhere in your code, whereas local variables (such as the ones defined in functions) can only be accessed within their own scope (e.g. if a variable is local to a function, only that function can use it). This is not an important distinction right this minute, but when we learn about functions later it'll be good to know.

One more thing to note is that JavaScript is a loosely-typed language. There are (basically) two kinds of languages: loosely-typed and strictly-typed. An example of a strictly-typed language is ActionScript (the language Flash apps are written in), and the same variable declaration we just wrote would look like this in ActionScript 3:

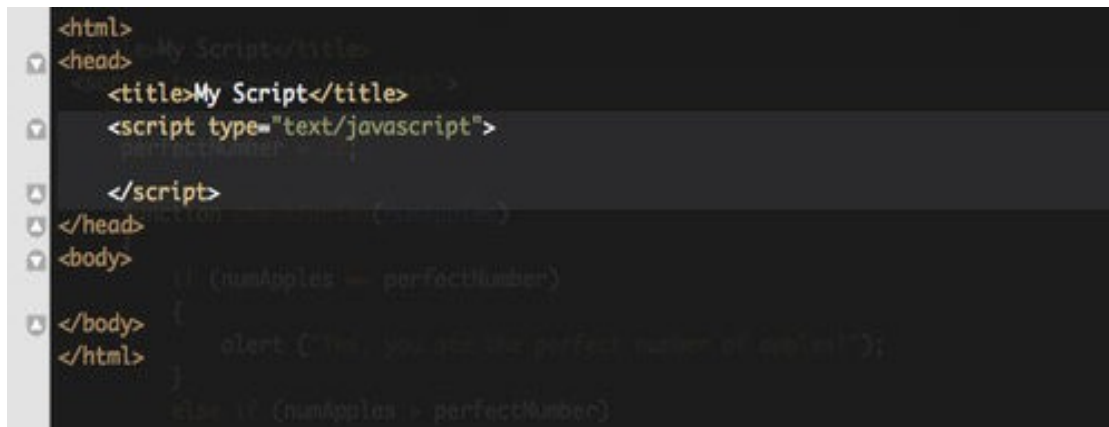
```
var myVariable:String = "Hello world!";
```

The additions you're seeing are the word *var* and the word *String* (with a colon in front of it). The word *var* tells the computer we're about to declare a variable. The *:String* attached to the variable's name tells the computer what type of variable it is and to not accept any other type. This is the reason for the term *strictly-typed*. A loosely-typed language like JavaScript is more flexible and doesn't require any of that. This makes your code more flexible, but some will argue it will also make it more error-prone. We're not going to get into the pros and cons of strictly- and loosely-typed languages here, but it's good to be aware of the basic differences now as you will most likely encounter them in your programming endeavors.

Now that you understand what variables are and how they work, we can try using them in some actual JavaScript code.

Creating Variables and Using the JavaScript Alert() Function

Let's create a simple HTML document that we can use to test our variables:

A screenshot of a code editor showing an HTML document. The code is as follows:

```
<html>
<head>
  <title>My Script</title>
  <script type="text/javascript">
    // JavaScript code here
  </script>
</head>
<body>
  // JavaScript code here
</body>
</html>
```

You're going to want to have a better-defined HTML document when you're actually writing code, but for our purposes this will work just fine. Save the code above as a file called `myscript.html` (or anything you want that ends in `.html` and doesn't contain spaces or special characters) and open it up in your web browser. You'll see absolutely nothing other than "My Script" in the title bar. We still have more work to do. First, let's declare a variable inside of the script tag:

```
myVariable = 5;
```

Here we've just declared a number. Let's look at other variable types we can declare

```
myNumber = 5;  
myString = "Hello world!";  
myBoolean = true;
```

That gives us a number, a string, and a boolean. Now let's take that myString variable and actually do something with it:

```
myNumber = 5;  
myString = "Hello world!";  
myBoolean = true;  
alert(myString);
```

You'll notice I've added the line `alert(myString);`. This calls a built-in JavaScript function (we'll learn more about these later) called `alert()` which creates a pop-up dialogue box for users to interact with.

You never really want to use these in practice because—as any internet user likely knows—alert boxes are very annoying, but they make for a good way to test your code, to make sure it works, while you're writing. The parenthesis following alert allow you to provide alert with data it might need. Not all functions will require that you give it information, but alert needs to know what to alert the user. In this case, we gave it myString, so the user will receive a popup notification that says "Hello world!" Try this with your other variables to get popups with a number and a boolean value.

What is "Hello world!" all about?

Writing a simple program that says "Hello world!" is generally the first thing every programmer does when they're learning how to code. It's not necessary, but it's sort of a tradition and an initiation into the club.

Why give alert() a variable and not just give it the contents of the variable? Well, if you said `alert("Hello world!")` you'd get the same result in this example, but variables are called variables because they *vary*. The idea is that the contents, or values, of these variables will change as users interact with the programs you write.

Learn to Code Part II:

Working With Variables

[Link to YouTube video](#)

In our second "Learn to Code" lesson, we'll be taking a look at how to actually work with the [variables and data types we learned about in the first lesson](#). Get excited—it's time to put your new knowledge to work!

These lessons work best with the video, which you can see above, but we're also providing text for reference below. Even if you do prefer to read, the videos will be more explicit and demonstrate how to do everything we're discussing. If the text seems a bit too complicated, be sure to watch the video.

Creating a Variable Statement

We're going to create a variable statement so you can see how useful variables can be. Everything we'll be doing in this lesson will take place between the `<script>` tags you created in your `myscript.html` in the [previous lesson](#). First we're going to create four variables and then we're going to use them to make a sentence. Here are the four variables you want to create:

```
var myName = "Adam";  
var foodType = "apples";  
var numberEaten = 3;  
var numberTotal = 7;
```

You don't have to set their values to what you see above. For example, if your name is Melissa then you might want to set the value of the `myName` variable to "Melissa" instead. If your name is really terrible and embarrassing (e.g. Herpe), it's okay if you use mine. Anyway, let's make a new variable called `myStatement` and combine our four variables to make a sentence. To do that we just set `myStatement` like this:

```
var myStatement = myName + " ate " + numberEaten + " " + foodType + ".";
```

You'll notice that I added some words and spaces to make it a coherent sentence and a period so it's properly punctuated. You can assemble this sentence however you'd like, but the above example will give you something that works well. Now, let's test it. Use the `alert()` function to view what we came up with:

```
alert(myStatement);
```

What you should see in the alert is: Adam ate 3 apples.

Okay, great, so you just wrote a useless sentence and didn't even use one of the variables. Now what? Let's do a little math. Try this sentence:

```
var myStatement = myName + " ate " + numberEaten + " " + foodType +  
", leaving " + (numberTotal - numberEaten) + " " + foodType + " left  
over.";
```

What just changed? Well, we altered the statement a little bit to also inform the user of how many apples were left over after you or I had our way with the apples available to us. The important thing to notice here is the part of the code that says

`(numberTotal - numberEaten)`. When you add a string variable to practically anything (even a number), it creates another string. If you add two numbers together, it actually does the math (meaning $2 + 3$ will equal 5 and not 23). Because we have other strings in the mix, we need to make sure this mathematical operation is carried out as math *first*. By putting the mathematical statement in parenthesis, the computer will know to perform that operation first before considering it part of the big long string variable we're creating. Obviously we're not adding these two numbers together, but the same rules apply for all mathematical operations.



With the changes to `myStatement` made, `alert(myStatement)` should now show: Adam ate 3 apples, leaving 4 apples left over.

That's all we're really going to cover today, but before you call it quits be sure to change the contents of your variables and see how `myStatement` automatically updates when you reload the page in your web browser. This is your first small look at what makes programming so useful. Ready for what's next? Go check out [Lesson 3](#), which tackles arrays and logic statements. This is where things get a bit more challenging, but also a lot more fun.

Learn to Code Part III:

Arrays and Logic Statements

[Link to YouTube Video](#)

You've mastered the basics of variables and made it half way through our course, but are you up to the challenge of arrays and logic statements? Of course you are. Let's get started!

These lessons work best with the video, which you can see above, but we're also providing text for reference below. Even if you do prefer to read, the videos will be more explicit and demonstrate how to do everything we're discussing. If the text seems a bit too complicated, be sure to watch the video.

This is where things get a bit more complicated. There's no need to be intimidated by what's to come, but just know you might hit a point of frustration because we're going to cover some of the more complex—but incredibly useful—stuff today. This is going to be the hardest (and longest) part of the beginner lesson, but you can do it. You just may need to rewind and practice a little bit more than with the previous two lessons.

First, we're going to learn about arrays. After that, we're going to take a look if statements and for loops, and also how to use those tools with your array.

Arrays

```
var myArray = (1,2,3);
```

Arrays are like mini-databases. Instead of just storing one piece of data in them, you can store several pieces of data. They're going to be your new best friend.

An array is a type of variable, but it's more like a box with a bunch of sections. Unlike the simple variables we've discussed before, arrays can contain more than one piece of data. Let's take a look at how an array can be defined and then we'll talk about what it all means.

```
var myArray = new Array("Lisa", "George", "Adam", "Paloma", "Jeffrey");
```

I've just created an array called `myArray` that contains five names. Each is separated by a comma and each name is in quotes because they're strings (if you forgot what a string is, refer back to [lesson one](#)). As you might have guessed, arrays are really useful for storing a bunch of similar data inside of one variable for easy access. It's kind of like a mini database. When you want to access an item in an array you do so by number.

```
myArray[0]
```

The above would resolve to Lisa, because Lisa is the first name in the array. If you changed the 0 to a 1, it would resolve to Adam. If you put a number beyond the number of items currently in the array (like 12), you won't get anything at all.

That's basically how arrays work. Not too tough, right? There's a ton more that you can do with arrays (e.g. sorting, splicing, searching, etc.) but we're not going to get into all of that here. If you'd like to skip ahead a little bit and take a look at array methods in JavaScript, [check this out](#). For now, though, that's all the information you really need.

What's a Method?

A method is like a function, only it's attached to a type of variable. So instead of just calling it out in the open, like `alert()`, you call a method along with a variable. `Array.join()`, for example, will take the array that `.join()` was added to and turn it into a single string. Don't worry too much about functions and methods, though. We'll cover that stuff tomorrow.

For Loops



The `for` loop is going to be the most complicated thing we're going to deal with today, and we're looking at it before the `if` statement because we're going to use an `if` statement to modify its behavior. So what is a `for` loop exactly? Basically, a `for` loop runs a chunk of code a specified number of times and counts that number as it moves along. Imagine you're a number variable called `i`. (You can pretend that stands for `I` as in yourself, but it really stands for iterator.) As `i`, you need to run a mile and you're at the starting line of a quarter-mile track. To complete a mile, that means you have to run four laps around the track. Represented as a `for` statement, that might look something like this:

```
for (i=0; i<4; i++)  
{  
    run();  
}
```

Let's break this down. When you say `for`, you're telling the computer you're declaring a `for` statement. That should make sense. That's pretty much the same as saying `var` before declaring a variable. That's the easy part. The complicated part happens inside the parentheses. Let's look at each part individually.



i=0

When you use `i` in the `for` loop, it's a variable you have to declare. It's not just there for you to use. You could use any letter, or a sequence of letters. It doesn't really matter what you call this variable, but traditionally `i` is the way to go. Because you're declaring it inside of the `for` loop, you can reuse it in another loop later. This variable is local to the `for` loop and won't conflict with stuff outside of it. What we're doing here is setting it to 0. You don't have to start at 0, but in this instance (and virtually *every other situation* where you'll use a `for` loop) it makes sense. We haven't run any laps yet, so we're starting at zero.

i<4

A `for` loop is designed to run a bunch of code until you tell it to stop. This middle portion tells the `for` loop how long to run. We only want to run a mile, which is four laps, so we're telling the `for` loop that so long as the variable `i` is less than 4, keep going. If we wanted to run two miles, we could change it to `i<8`.

i++

The last condition is very simple: increase `i` by 1 every time this loop completes. You can use `++` or `--` to increase or decrease (respectively) a number by 1 anywhere else in your code, too, but chances are you'll use it most with `for` loops. This part is very important because we're telling the `for` loop to stop running when `i` is no longer less than the number 4. Because `i` starts at 0, we need to increment `i` by one each time the loops runs or it will run forever.

Inside the Curly Braces

The curly braces `{}` currently has `run()` in it, but in this example that's not a real function. Basically, you put whatever code you want to run inside the curly braces. They're just around to section off the specific code you want to run.

One More Time, Please!

Okay, let's just run through this one more time because it's sort of complicated (or, at least, it was for me the first time I learned it):

- The `for` tells the computer that we're defining a `for` loop.
- `i=0` sets a variable called `i` equal to 0, which is our starting point.
- `i<4` tells the `for` loop to stop once `i` is no longer less than the number 4.
- `i++` tells the `for` loop to increment `i` by 1 after each time it finishes running the designated code.
- All the designated code that the `for` loop is supposed to run needs to be placed inside the curly braces.

Let's Put It to Use

Okay, let's now make a `for` loop that loops through our array and alerts us of each name in the array.

That code should look something like this:

```
for (i=0; i<5; i++)  
{  
    alert(myArray[i]);  
}
```

This real example should look very similar to the fake example we just dissected except for one thing: `myArray[i]`. Earlier we looked at how we can access the contents of an array by number, so `myArray[0]` will give us something different than `myArray[1]`. Because `i` is a number that changes as the `for` loop runs, each time the loop will access a different point in the array. This saves you the trouble of writing the code out five times.

Okay, but what if we don't know the length of the array? Right now we know there are five elements, but if you make a change we'll either be running the loop more than we need to or we won't be running it enough. What we want to do is to run the `for` loop until we've accessed the entire array. That requires one little alteration:

```
for (i=0; i<myArray.length; i++)  
{  
    alert(myArray[i]);  
}
```

To make life easier, JavaScript (and most languages that use arrays) have a property built in to all arrays you create. (Actually, there are a bunch of different properties but we're only going to look at this one right now). This property, called `length`, gives you the number of items in the array. So, instead of saying `i<5` we'll just say `i<myArray.length` and the `for` loop will run until it's run out of items in the array.

Did you survive all that? If you did, that's mainly what you need to know about `for` loops. There are few more examples in the video up top, so be sure to check it out if you want to see a few other things you can do with them.

If Statements

if (statements) {}

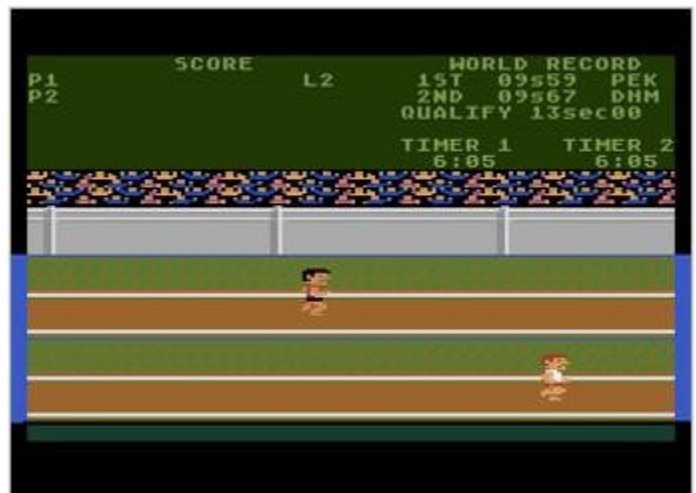
If you want to learn if statements, then read this section. (If you already know about if statements, you know that last sentence was sort of an example.)

If statements are probably the easiest type of logic statement to understand. They're powerful, too, so it can be easy to get addicted to them. Novice coders tend to cling to if statements because it seems that if you have enough of them, you can do just about anything. That is true, if you don't mind losing your mind in the process. You can write some fairly complex code that operates on if statements, but you'd need a lot of them and it will drive you crazy. So, although you're probably going to love them, don't overuse them. Too many ifs do not make for efficient, good code.

So what is an if statement? It's basically a statement that says if the specified condition is true, then run this block of code. It can also be used to say, if the first condition isn't met, do this instead. You can also check if an alternate condition is met if the first one fails. For example, if you want to wash your dog if your dog is blue, you can use an if statement to find out and wash the dog if it turns out to be blue. Here's how that could look as a piece of code:

```
if (myDog == "blue")
{
    washDog();
}
```

Like we've seen before, `if` tells the computer that we're declaring an if statement. In parentheses, we're defining the condition. You'll see we have a variable called `myDog`, which presumably contains a simple piece of information: the color of your dog represented as a string. It could be "red" or "green" or "blue", but we don't know yet. To find out, we're going to ask if `myDog` is equal to "blue" to find out if it's actually blue. To test for equality, we use `==`. If you use a single `=` then you are setting the value of a variable. If you use two `==` then you're testing to see if one variable is equal to another, or just equal to some kind of data. If the dog turns out to be blue (meaning if the condition is met and `myDog` is equal



to "blue"), then the if statement will allow the code in curly braces {} to be run. In this example, the code inside the curly braces is `washDog () ;`. While `washDog()` is not a real function (not yet, anyway), if it were it would presumably go forth and wash the blue out of your dog.

Okay, so how can we apply this in our code? Well, the video will walk you through a more complex example but we're just going to test for someone's name. Let's say you included my name (Adam) in the array and you wanted to receive an alert *only if my name comes up*. Well, we can combine your for loop and your if statement to do just that:

```
for (i=0; i<myArray.length; i++)
{
    if (myArray[i] == "Adam")
    {
        alert("I found " + myArray[i] + " in the array!");
    }
}
```

Basically, we've just put an if statement inside of the for loop, and our if statement is now asking if any position in the array is equal to Adam rather than asking if a simple variable, like `myDog`, is equal to "blue".

Got all of that? Good! If you made it through this lesson it'll be smooth-sailing with the next one. If you're ready, [move on to lesson 4](#) where we'll learn about functions and making a simple guessing game.

Learn to Code Part IV:

Understanding Functions and

Making a Guessing Game

[Link to YouTube Video](#)

If you've made it this far in our programming lessons, you've arrived at the reward. Today we're learning about functions and then we're going to make a very simple guessing game.

These lessons work best with the video, which you can see above, but we're also providing text for reference below. Even if you do prefer to read, the videos will be more explicit and demonstrate how to do everything we're discussing. If the text seems a bit too complicated, be sure to watch the video.

Today is our last-ish lesson (there will be an "epilogue" with a mini-lesson on best practices and additional resources following this one), and we're going to cover two things: functions and making a very simple guessing game. If you made it through [Lesson 3](#), chances are you'll find functions pretty easy to learn. We're going to tackle that first and then use that knowledge—plus a little HTML—to put the game together.

What the F is a Function?

Oh, come on, you already know all about functions. You've been using one this entire time: `alert()`. A function is basically a means of calling a large chunk of code with just a tiny bit of text. Functions are kind of like the [text expansion](#) of programming. While you've used `alert()`, that's just a built-in function (of which there are many in JavaScript and all programming modern object-oriented programming languages). Today, we're going to create our own function that will eventually help us make a guessing game. First, let's take a look at how a function is declared in your code:

```
function nameOfFunction(variables)
{
}

```

This should look pretty familiar. Functions look a lot like for loops and if statements. Let's break down what we're looking at. First, we start with the word `function` to tell the computer that we're defining a function. Next we have to name that function. In this example it's titled `nameOfFunction` but you'd normally name it something relative to what it does (like how `alert()` creates an alert dialogue box when used). After the name of the function are parentheses. You're not required to put anything inside the parentheses, but if you need to give the function some information (which is called passing it

variables/a variable) you need to specify the name of the variables that you'll be using inside of the function. This might be a little confusing right now, but we're going to look at it a bit more later so don't worry. Finally, we have our curly braces at the end. Everything the function needs to do will go inside of these curly braces.

Now that you know how a function is defined and what it does, let's put it to use!

Making a Simple Guessing Game in JavaScript

There are two parts to this process: the function and the interface (which is basically a form where the user can enter a number). First, we'll start with the function.

Making the Function That Runs Your Game

To make the game, we're going to need to make a function that reacts to user input. Let's put together a function that does that:

```
var perfectNumber = 12;

function checkApples(numApples)
{
    if (numApples == perfectNumber)
    {
        alert("You ate the perfect number of apples!");
    }
    else if (numApples > perfectNumber)
    {
        alert("You ate way too many apples.");
    }
    else if (numApples < perfectNumber)
    {
        alert("You didn't eat enough apples.");
    }
}
```

Okay, that's a lot to look at so let's break it down. First, I created a variable called `perfectNumber` and set it to 12. This is the number the user is going to try to guess in our game. You can set this number to anything you want, and if you want a few bonus points you can try implementing this [randRange\(\) function](#) to randomize the number each time the page loads—but let's do the basic stuff first.

After the variable `perfectNumber` is defined, we're creating a function called `checkApples` and passing it a variable called `numApples`. This means that when we call this function we'll need to give it a number, which would look something like `checkApples(5)`. That number is going to come from the user, however, so we're not going to talk about that just yet.

Inside the function is an if statement with three conditions. The first condition checks to see if `numApples`—the number variable passed to the `checkApples` function—is equal to `perfectNumber`—the variable we set earlier (that's the answer the user will be trying to guess. If that condition is met, the

user will get an alert that congratulates them on choosing the correct number. If not, the second condition will check to see if they guessed too high. If they guessed a number larger than `perfectNumber` they'll be alerted that they ate too many apples. Basically the same thing happens with the third condition, but if they guessed too low of a number.

That's all there is to the function, and it's pretty simple. But how do we request a number from the user? That's going to require combining a little JavaScript with a little HTML.

Making a User Input Form

Making a form is something you might've done before if you've ever used HTML. It's pretty easy. First, you'll need to put this in the `<body>` of your HTML document:

```
<form>
  <input type="text" name="numApples" id="numApples" />
  <input type="Submit" name="Submit" />
</form>
```

That will get you a super simple form with a text box and a submit button, but it won't do anything. First, we need to add some stuff to the form tag to connect the input to your `checkApples()` function:

```
<form method="POST" name="applesForm" onSubmit="checkApples(document.applesForm.numApples);">
```

So what did we just add, exactly? First, we told the form to use the POST method. This isn't terribly important because it'll work either way, but the POST method is a bit cleaner for what you're doing here. POST won't put all the variables submitted in a form into the URL, but GET—your other option—will. Second, we named the form `applesForm` because we're going to need to refer to it by name when specifying where the user input text box is in the document. Lastly, we have something called `onSubmit` which lets us execute some JavaScript code when the submit button is pressed. In most forms there would also be a property called `action`, telling the form where to go after submitting, but we just want to stay on the page so we don't need it. Getting back to `onSubmit`, you can see we've basically set it to the `checkApples()` function. Instead of giving `checkApples()` a number, however, we've given it `document.applesForm.numApples.value`. This is how we're referencing the user input text box in our form. We're doing this by using the Document Object Model, or DOM, in JavaScript. Let's break this down:

- **document** - Your HTML document.
- **applesForm** - The name of the form we created.
- **numApples** - The name of the text field we created for user input, which is inside of the `applesForm`.
- **value** - We don't want the text field we created, but what's inside of the text field. You need to attach `value` to the end of a text field you can get the value of it and not a reference to the text field object that contains it.

Once you've got all of this in your code, you're done! Save the page, reload it in the browser, and try

guessing your number. Aim too high, too low, and then guess it correctly. You should find that you get the responses you defined in your code alerted to you each time you guess. So, congratulations, you just made your first game!

It's Almost Time to Say Goodbye

Well that about does it for our basic coding lessons. We hope you've enjoyed them and feel ready to approach more complicated stuff as you begin your foray into the programming world. By popular demand, we'll be doing one more post tomorrow about a few programming best practices—mainly commenting your code—as well as including additional resources to help you learn more about everything we've discussed and where to go next.

Learn to Code Epilogue:

Best Practices and Additional Resources

[Link to YouTube Video](#)

Congratulations, you've learned the basics of programming! That's wonderful, but you'd better not go out into the world and write crappy code. Before we set you free, here are some best practices and good things to keep in mind.

Best Practices

Comment Your Code and Comment It Well

You can comment your code in two ways: one way for single-line comments and one way for multi-line comments. Single-line comments start with `//` in JavaScript. Other languages use other characters, like the `#` sign, so be sure to check before you start putting forward slashes everywhere. To make a multi-line comment, you just put your comment in between `/*` and `*/`. Here's an example of both:

```
// Single-line comment
/* Multi-line comment
Another line
One more! */
```

You'll mostly use single-line comments for making actually comments about your code and use multi-line comments to remove parts of your code without deleting them. This makes it easier to isolate a problem when you're debugging.

So how do you write good comments? Well, you're probably going to feel inclined in the beginning to write comments that explain how everything works. If that helps you remember, that's not a bad thing to do *in the beginning*. For the most part, however, you're going to be naming your functions and variables clearly and you won't need to explain how something works or what a particular function does because that clarity already exists. What you want to explain in your comments is information the code can't already tell you: *why* you made the choices you made. Anything that will help another programmer (or yourself, when you look back at this code months/years later) better understand your code is worth putting in a comment. If it's redundant information, it's not helpful. If it helps bring clarity to the code you've written, it is.

Don't Use the `eval()` Function

Oh boy, the `eval()` function isn't the easiest thing to explain. If this explanation doesn't make enough

sense and the video doesn't get the idea across either, just remember not to use `eval()` whenever possible. The main reason is that `eval()` slows down your code. The other main reason is that you can almost always find a better way to get the job done than to use `eval()`.

So what does `eval()` do? Well, it's short for evaluate and it evaluates a string as if it were a variable. So let's say you had a variable called `numberOfApples` and wanted to alert it to a user. You could do it like this:

```
alert(numberOfApples);
```

What if you put `numberOfApples` in quotes? Well, the alert wouldn't alert whatever number you set `numberOfApples` to, but instead just alert the text `"numberOfApples"`. `Eval` solves that problem:

```
alert(eval("numberOfApples"));
```

Why would you ever use this? Well, someday you'll probably find a reason why `eval()` will be useful to you. Generally it's because you don't necessarily know the name of the variable, or the name of the variable you need is contained in another variable. It gets complicated and we're certainly not going to get into it here, so just remember: *don't resort to using `eval()` unless there is no other way*—and I can't think of a situation where you wouldn't have an alternative.

Create Arrays and Objects Faster

We didn't really talk much about objects but you should definitely remember arrays from [Lesson 3](#). When we created them earlier, we did it like this:

```
var myArray = new Array("item1", "item2", "etc");
```

You don't really need the `new Array` statement, however, as you can just do it like this:

```
var myArray = ["item1", "item2", "etc"];
```

You also have a similar shortcut with objects, a type of variable we haven't discussed yet. Objects are very similar to arrays in that they can hold more than one piece of data at a time, but they're accessed a little differently and you can easily name each item in your object. It's really easy to see the difference when you look at the long way of creating an object:

```
var myObject = new Object();
myObject.item01 = "Ponies";
myObject.item02 = "Unicorns";
myObject.item03 = "Rainbows";
```

The long way works fine, but here's an easier way to do it:

```
var myObject =
{
    item01: "Ponies";
    item02: "Unicorns";
    item03: "Rainbows";
}
```

These shortcuts make writing your code a bit faster and make your code a lot easier to read.

Use Your Semicolons!

JavaScript does not always requires you to end a statement with a semicolon, but you should do it anyway. We discussed this a little bit in [Lesson 1](#), but it's worth repeating. You're almost guaranteed to run into problems if you're careless about using your semicolons, so whenever you need to end a statement (a sentence of code), don't forget to drop a `;` at the end.

Additional Resources

Now that you've learned the basics, hopefully you want to learn more and start making some awesome programs. Here are a few additional resources to help you learn more JavaScript as well as some other languages.

DO NOT Use W3Schools

When you search for help online, one of the first results is often W3Schools (which I'm explicitly *not* linking to here). The short version is that it sucks. It's full of errors, it's missing information, and while it's not 100% useless it isn't a good resource. Avoid it. For the long version, visit [W3Fools](#), a site put together by the [jQuery team](#) (and some other helpers).

General Resources

- [Mozilla Web Development Resources](#)
- [CarlHProgramming Lessons](#) are a great collection of programming lessons posted to Reddit.
- [Lynda.com](#) offers a large selection of online education, but it'll cost you. The lessons are excellent, however, and I feel it's worth the cost. It costs around \$25 (depending on the type of account you get), so if you can fit a course into a month you really won't be paying that much in the end.
- [MIT OpenCourseWare's Introduction to Computer Science and Programming](#) is a great big programming learning resources complete with [video lessons](#).

JavaScript

- If you're a Firefox user, you need to have [Firebug](#). Most other browsers have built-in developer tools, but [Firebug Lite](#) is a bookmarklet that will work in pretty much any modern browser.
- [JavaScript: The Definitive Guide](#) is a great book to read if you want a bible's worth of information.
- [Douglas Crockford's "The JavaScript Programming Language"](#) videos will teach you the basics (again) and advanced stuff, too.
- [JavaScript, JavaScript](#) is a great blog by Angus Croll that'll teach you a few things with each post.
- [Learn Advanced JavaScript](#) is an effort by John Resig to, well, teach you advanced JavaScript.
- When you're ready to really step things up, downloading and learning to use libraries like [jQuery](#), [Dojo](#), [MooTools](#), and [Yui](#) will make your life much, much easier. When you get to the user interface stuff, check out [jQuery for Designers](#) for some great tutorials.

PHP

- The official [PHP Manual](#) is how I learned PHP. I just searched for functions, browsed around, and learned by example. PHP was also the first language I learned, so it's not as if I knew what I was doing. The manual is very informative and has great examples. While I'd probably recommend learning programming basics before diving right in, you just did in this series so playing around with PHP shouldn't be too tough for you.
- Zend offers up [PHP for the Absolute Beginner](#). Zend is a company, but also a powerful PHP framework. If you want to start from step zero (which isn't a bad idea), check it out.

Ruby/Rails

- [Agile Web Development with Rails](#) is how I learned Ruby and Rails. I can vouch for it being very good, although technically I read the second edition.
- [RubyOnRails.org's screencasts](#) are another great way to learn.

ActionScript (Flash/Cross-Platform AIR Apps)

- [gotoAndLearn](#) is filled with excellent tutorials that teach you how to make really useful stuff in ActionScript, whether you're using Flash or Flex to deploy to the desktop or web. One of the best ways to learn is by doing, and gotoAndLearn offers plenty of opportunities to do just that.

Mobile App Development

- [Stanford's iPhone Application Development](#) is a course on iTunes U that you can download for free. You'll also find the class resources [here](#).
- The [Android Developer](#) site is a great resource (especially the [guide](#)) for learning how to develop for the Android platform. It's available for free from Google.

Special thanks to my friend [Colin Snover](#) for his input. He currently works on [jQuery](#) and is much smarter than me. Follow him on [Twitter](#). Also, a special thanks to CnEY?! for a few of the resources listed here and making sure I had nothing good to say about W3Schools.