Happy Learn JavaScript Tutorial Vol 1

http://www.happylearnjavascripttutorial.com/
Written and Illustrated by *GetContented*
*(*enquiries@getcontented.com.au*)*

Updated on: March 31, 2016.

Contents

1 How to learn JavaScript enjoyably

1.1 Fascination

Just like you, when we discovered computers we were taken in, fascinated by their potential. We watched happily as programs seemed to make anything possible. Such amazing works by the programmers who created them.

1.2 Wish to Create

Delight quickly changed to desire: desire to write our own programs, but how best to begin? Discovering many books, we filled our heads with knowledge, sadly not finding much guidance about proceeding with the practical craft.

1.3 Too Many Details?

Trying our hand at making programs ourselves, we discovered details cluttered us. Losing our delight, we'd stop. We simply didn't understand how to use our dusty book-learning. Our programs stank.

1.4 The Journey Begins

We felt bound by our knowledge, needing to return to the freedom of before we began, so we endured tedious practice and failure, making the theory our own, slowly understanding. Again we found no guide lighting our path, but after an arduous journey, finally the fog lifted and we could write excellent programs and still be excited and joyous.

1.5 And Now You...

So you too want to become a programmer. Luckily, you've found this guide, crafted by people who have lived this path to the end, and would save you the pain and boredom we endured.

1.6 No Magic, But Why Pain?

There's no magic here, but there are more-, or less-difficult paths to choose. If you want mastery, you always need deep practice, but why should this mean pain and boredom? Small steps will be our guide, and fun our companion. But, how to journey?

1.7 Precise Language

Far more precise language than ours is needed to program. To write in such a language, we need to know the correct words and their strict arrangements; and how to tease our intent apart and clothe it nicely as a program. This is not all.

1.8 Taking Care

A little knowledge, that dangerous thing, produces some success, and worlds of possibilities arise, bringing excitement with them. The eager beginner quickly gets into a flurried muddle, as enthusiasm has them tackling too much too soon. Initial elation slowly turns into bitter disappointment and they give up or worse, spread hate. We don't want this for you.

1.9 Two Phases of Learning

Instead, your learning will proceed in two staggered stages. Each lesson will introduce several programs. Reading, understanding and typing these in yourself will embed them in your own experience. You need this practice to recognise the pieces and to know what they do. We'll then adjust them slightly and see how they change.

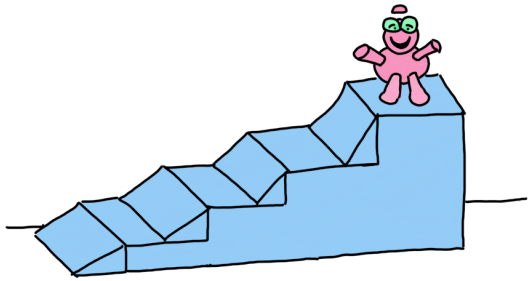
1.10 Stages Build Skill

This proceeds in a graded, staged way. Using real-world problems to illustrate simple solutions with the language constructs seen so far, our reading will slowly increase in difficulty until we have seen the core of the language and beyond.

1.11 Simple, but Fun Examples

The second stage starts further along, when enough reading means you know how to do small things. Again, we'll take care to work within enjoyable limits as we show you how begin to build a path the other way: from problem solving, to intention, to code. This

stage will solidify your understanding. Proceeding, we slowly take the training wheels off and before you know it, you'll be able to make some well-designed programs that read well, are easy to understand and are enjoyable to change.



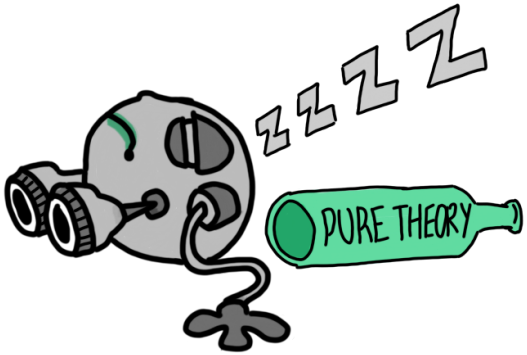
Many books don't address the subject of how to craft solutions, or they just leave you with nothing but some exercises and your own intuition. Most are focussed heavily on programming language topics first, and how you use them to do things second, if at all. You'll notice we're primarily interested in you learning how to do useful things, rather than the language for its own sake. Practical things will anchor the language more in your memory and experience.

1.12 Progressive Learning

Our material is cleverly crafted to gradually introduce you to the entire language. We do this over the course of the various, interesting examples which are present in every chapter, across all the volumes. We chose this way because the other way bores people to sleep, which is inconsiderate and tedious.

1.13 Natural Assmiliation

Countless people have found programming difficult to learn because of boring examples, unpolished writing, or material being organised for language features rather than learner interest.



On the other hand, we've seen great success in material that uses varied repetition, amusing useful pictures, fun examples using real-world topics and small graded steps. This guided our choices in building the entire series, one volume of which you have in your hands.

1.14 Motivation is King

If motivation really is what pulls us through practice, then we sincerely hope we've inspired motivation and excitement in you with this series, and wish you never forget the love that we all share for learning and programming. May it guide you always.

2 Display a Message

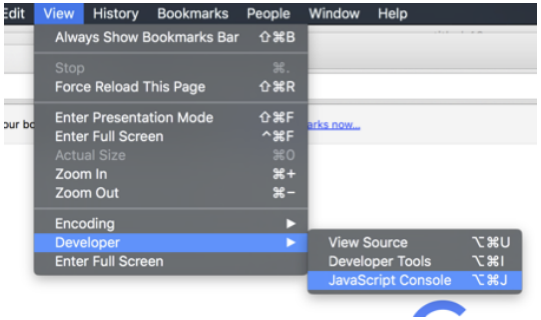
This chapter is where you start to become a beginner programmer by learning how to read your first few simple programs.

Using programs that put an alert on the screen, our aim for this lesson is to get you familiar with some basic JavaScript and the accompanying terms.

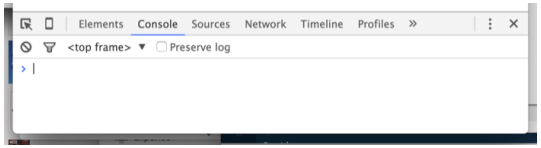
We'll open the Chrome web-browser now, because it's easier to start programming in. If you don't have it, visit `https://www.google.com/chrome/`.

2.1 Let's Begin

Each window in Chrome has a place to enter small programs directly called the **JavaScript console**, which is normally hidden. Let's go to it now:



Open a new window with `File... New Window`, then a console with `View... Developer... JavaScript console`. To select it, click beside the `>` symbol:



We can type any text we like here, and pressing the return key, the browser will interpret it as JavaScript. If it makes sense, our program will do its work (**run** or **execute**).

2.2 Our First Program

Below you'll see your first **program** — just one line. Type it into the console carefully and press return to run it. If you make a mistake, try again on the next line, or start again.

```
alert();
```

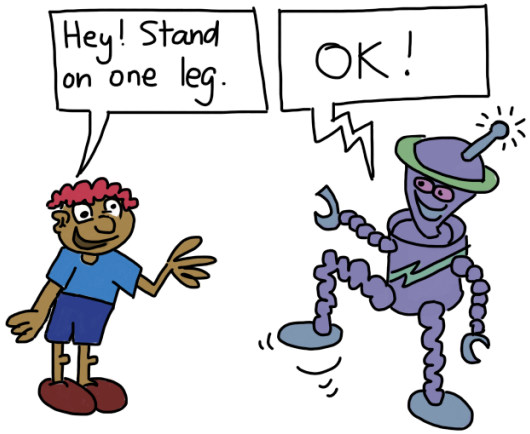
Congratulations on your first step into the world of JavaScript programming! We're now going to walk through what this line means and explain some of the terms and pieces common to all programs.

2.3 What Happened?

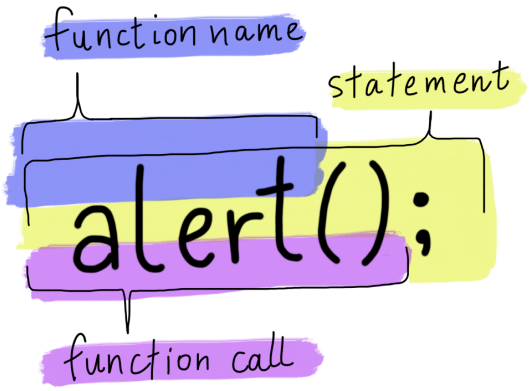
When you pressed return, a **dialog box** appeared, so-named because the computer is "talking" to us, and needs a response. We give one by clicking the button on that window which will **dismiss** it, and it will disappear. This tells the computer you've seen the window.

After it's gone, you'll see undefined is "printed" to the console. This is normal, but for now ignore this.

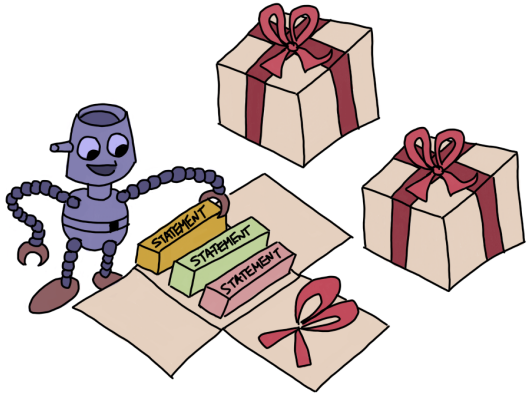
What actually happened, though? Well, JavaScript programs are written as a sequence of **statements** of actions to take. These end in a semi-colon, and each tells the computer what to do.



For simplicity, our first program is only one statement. We'll see programs with more soon.



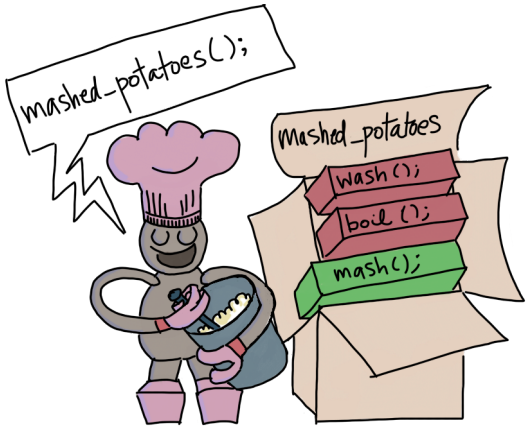
When you run a program, JavaScript executes its statements. In our case, we have a **function call** in our statement, so it looks at the word alert, and knows we're talking about its built-in alert **function**. A function is a kind of block, or parcel of code, like a mini-program!



When it gets to the round brackets written after **alert** (these brackets are called **parentheses**), it knows it means we're telling it to **call** that function.

2.4 Calling a Function

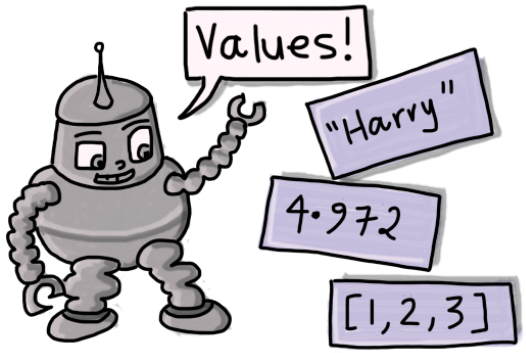
Calling a function means executing all of the **statements** inside the definition of that function, in the proper sequence: evaluating its contents and responding with a return value.



The term **value** in JavaScript just means a piece of data, such as the number 100 or the word "cats".

2.5 Values

This particular built-in function displays an alert box on the screen and waits for you to click the button. Until you click it, JavaScript pauses program execution. When you do, the alert function responds with the undefined **value**.



The undefined value is returned when the computer hasn't got anything more meaningful to respond with. It's returned here because all function calls must return a value.

2.6 The Next Step

Putting a blank alert on the screen is ok, but it'd probably be more useful and interesting if we could actually display a text message.

Let's call the alert function with a message now:

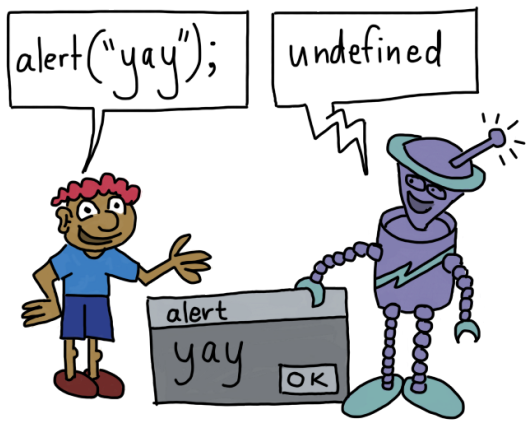
```
alert("hi");
```

If you **run** this program (by typing it into the console, then pressing the return key), an alert will pop up with the message in it. As before when you dismiss it undefined is printed in the console.

We can tell from doing this that the message shown is whatever value we put in quotes in-between the parentheses (the rounded brackets).

2.7 Function Parameters

So the alert function can take a parameter, or not: the message to display. JavaScript programmers call the parameters that a function takes its **arguments**.



What about if we write two statements on the one line:

```
alert("hello") ; alert("hello, again") ;
```

If you try this, you'll see that it does the left statement first, then the statement on the right, displaying an alert each time and waiting for you to dismiss it: it runs them in sequence.

So we can see that the semi-colon tells JavaScript where the end of a statement is, even if there are more than one of them on a single line of **code**.

You may notice that extra spaces don't change anything for JavaScript. We usually won't write our code like this, but it's good to know it's possible, so you don't get confused when you read other people's programs that do this.

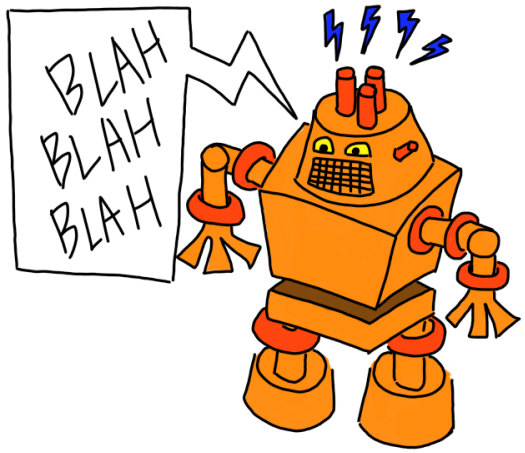
Well we made it to the end of the first lesson. Well done! Take a breather, and then try the homework.

2.8 Homework

Your homework is to type the code in and try it if you haven't. This will make it more your experience. Then try writing five different alerts with your own messages in them. Don't forget the quotation marks around the messages, or you'll get an error. We'd also like you to do an internet search for JavaScript code and make sure you can see the statements in any code you see in a few pages. You shouldn't try anything more complicated yet. We want to make sure you only do a little bit at a time and not get confused.

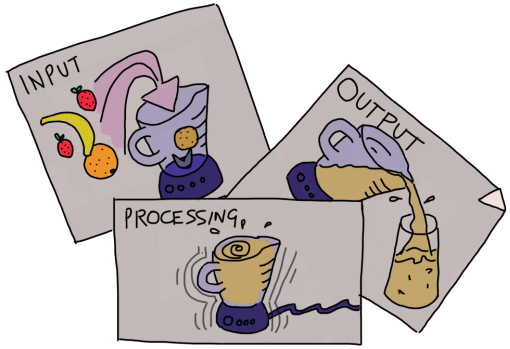
3 ChatterBot Mark 1

Welcome back. You've seen some simple code, so using a silly little talky program, we're now going to show you how to write code in files, get input from the user, join text together, do some simple math, and introduce **variables**, too!

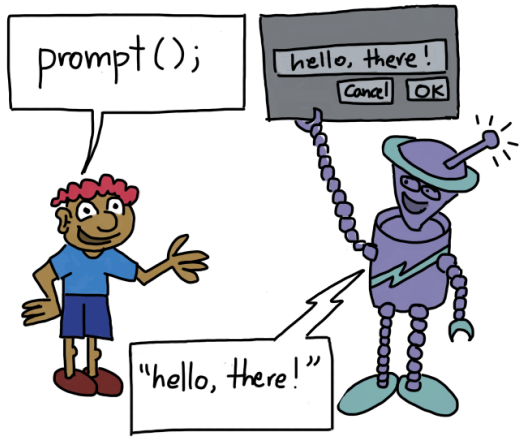


The ChatterBot program's purpose is to ask a few questions and give a reply.

It may surprise you to find out that computers can only perform four main basic tasks: **input**, **processing**, **storage** and **output**. Whatever it appears they're doing from moment to moment, however complex it is, it will just be a combination of these four tasks. We already saw some output, ChatterBot will use the remaining three, too.



We're going to use a text editor — an application for editing unformatted text — to make our source code. We can't use a **word processor** because its saved format is incorrect. Download the one we'll use so it'll be ready when we need it from the website `https://atom.io` — just use the download button on the front page.



Meantime, we'll play with some new functions, so open the JavaScript console again. It's normal to not remember how, just go back and remind yourself.

Take a look at the code below. It's a statement with a function call again. This one is named prompt. As before, the semi-colon marks the end of the statement, and the parentheses show that we're calling the function.

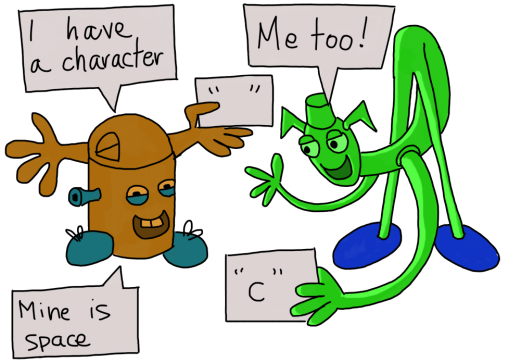
3.1 The prompt Function

```
prompt();
```

When you hit return, a box comes up with a **field**, an area to type into. Type `"jelly"`, and press OK. The console prints your entered string `"jelly"` as its response.

3.2 Strings and Characters

We've seen strings before. The quotes tell JavaScript what's inside is text data; a series of characters. If you don't know, a **character** is any unit of written data, such as the letter `"F"`, and even non-textual ones, like the character for new-line, or the space that appears between words.



This `prompt` function is like `alert`: we can call it with or without an argument. If you give it one, it'll use the argument in the window to prompt for input.

Of course the main difference compared to `alert` is that `prompt` lets the user input some text. On pressing OK, that input becomes

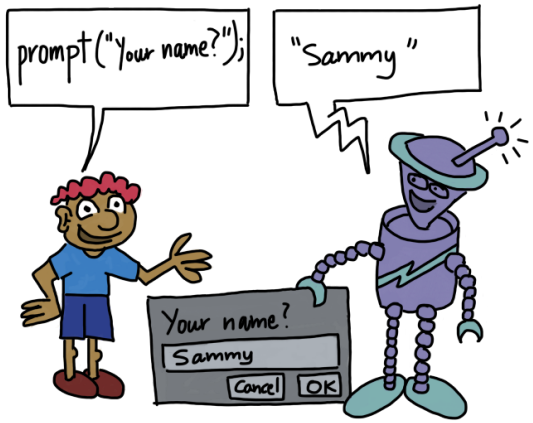
the string value returned. If cancel is pressed instead, a `null` value is returned rather than a string — this is an intentionally empty value. By contrast, we saw that JavaScript uses `undefined` when the `alert` function returned because it didn't actually return a value at all. Don't worry too much about this, it will become clearer when we see it more.

Try running this again a few times, pressing cancel or OK, and using different string values in the prompt dialog box.

What about when we call prompt with a string as its argument?

```
prompt("Your name?");
```

As expected, the string appears in the dialog box above the field. We can see prompt is one way to have the computer collect **input** from the user into our programs. We'll see how we can use these responses shortly.



By now Atom should have downloaded, so install and open it.

3.3 Programming from Files

We'll use it to create a web page to hold our ChatterBot program because it'll be larger than just a few lines entered into the browser.

Make a new file, and save it as chatterbot.html in a location you'll be able to find later.

Now type the following HTML. We're not going to explain the HTML, but It contains an embedded program in JavaScript. You'll probably recognise what this program does: it displays an alert on the screen.

```
<html>
```

```
  <body>
    <script>
      alert("Hi");
    </script>
  </body>
</html>
```

To load this into our browser, save it, go to the browser and choose Open File from the File menu. Choose the file you just saved. When you choose it, the page will load and the JavaScript will run immediately. To re-run it, simply reload the page (either with the button, or from the **View** menu).

Well done on creating your first file-based JavaScript program and running it. Next we'll modify it to ask your name using the `prompt` function.

```
<html>
  <body>
    <script>
      prompt("What's your name?");
    </script>
  </body>
</html>
```

If you save this, then use reload in the browser, it will load the new program in and run it, and so ask for your name.

When you do that and press OK, what happens to the return value from `prompt`? We didn't tell JavaScript to use it, so it just disappeared. On the console it would print out the result, but when we're inside a program the result is just thrown away. Soon we'll find out how to capture it instead.

We'd like our ChatterBot to produce a greeting alert after asking for the name of the user.

```
<html>
  <body>
    <script>
      prompt("What's your name?");
      alert("Hello, ");
    </script>
  </body>
</html>
```

When you run this, it does what we'd expect: asks for a name, and gives you a message `"Hello, "`.

However, we'd really like it to actually put the entered name into the greeting. Let's put a dummy name in first, then change it out for the actual one. We could just put it into the existing string argument to the alert, but let's do something else instead:

```
<html>
  <body>
    <script>
      prompt("What's your name?");
      alert("Hello, " + "Penelope");
    </script>
  </body>
</html>
```

3.4 String Concatenation

We used an **operator** called + here to show how to join our dummy-name `"Penelope"` to the greeting. Operators are symbols used with values to form expressions that produce other values. Try this program out.

An **expression** is any JavaScript code that the computer can work out a value for. Joining two strings with + is an expression. Function calls are expressions, too. Operators can be used to help make expressions. All expressions can be used as statements (we call them **expression statements** when they are), and can also be part of a statement, but not all statements are expressions. This will become clearer when you see more. Don't worry about it.

So we now know that the + operator used with two strings joins them into one. This joining together operation is called **concatenation**, a word originating from Latin that means "chain on to".

3.5 Some Input

This doesn't do what we want yet, though. We want the actual return value from the call to `prompt` to be concatenated to the greeting, not just `"Penelope"` every time. Let's see an adjusted program that does this in a simple way.

```
<html>
  <body>
    <script>
      alert("Hello, " + prompt("What's your name?"));
    </script>
  </body>
</html>
```

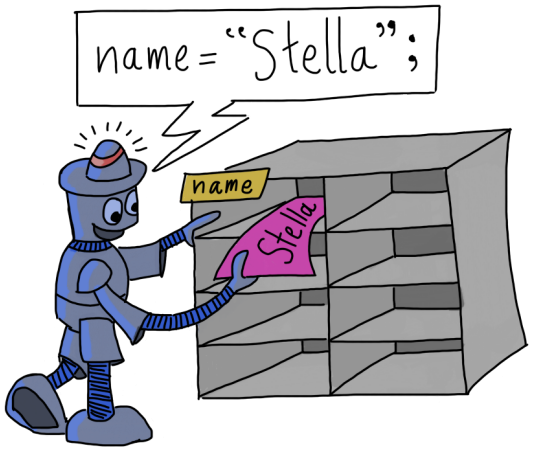
Ok this is kind of crazy. We've embedded the call to `prompt` into the call to `alert`! We know `prompt` **returns** its field contents as a string, so we just inserted the whole call to `prompt` into the `alert` function's argument concatenating it to the end of `"Hello, "`. When you run this, you'll see it does what we want.

When evaluating a function call, JavaScript looks at its arguments and evaluates them first, then feeds that result into the function. In the case of `alert`, that means `prompt` is evaluated to get its return value so we can concatenate that to `"Hello, "` and then feed that whole string into `alert`. So `prompt` happens first, and then `alert` displays its window with the greeting string!

This does work, but it's not very elegant. We programmers tend to prefer when code is clear, and this isn't very clear.

3.6 Variables

It would be nicer if we had some kind of temporary named storage location we could put the result of `prompt` into, then pass that to the call to `alert` so it would be clearer what our intention is, and what sequence things happen in.

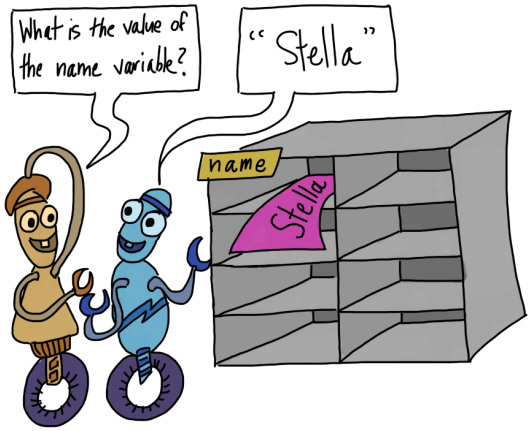


Well, JavaScript definitely has something like this. It's called a **variable**. This is a named value that can change: a value that is able to be varied! You can assign values to it, and if you refer to it, its value evaluates to whatever you last put in it.

Let's take a step backward and see another program that says hello to someone named Stella, but uses a variable.

```
<html>
  <body>
    <script>
      var name = "Stella";
      alert("Hello, " + name);
    </script>
  </body>
</html>
```

Ok the first statement **declares** and **assigns** a variable named `name`. Declaration is when we use the `var` keyword to indicate that we're going to use the word `name` as a variable in our program. Assignment is when you use the `=` operator to assign a value to a variable. We're doing both at once, putting the value `"Stella"` into `name`. Here, we're using the name `name` as a variable identifier. This means it is used by Javascript to identify the variable.



This means later on when we refer to the **identifier** `name`, JavaScript looks up the value stored for it, and uses that. As you can imagine, this is incredibly useful.

On the next line, we have our old friend the `alert` function, in a statement that uses the value from the variable, the string concatenation operator `+`, and the string `"Hello, "` to produce a greeting alert.

Obviously again, we'd prefer if the variable's contents are the result of prompting the user for their name rather than just `"Stella"`:

```
<html>
  <body>
    <script>
      var name = prompt("What's your name?");
      alert("Hello, " + name);
    </script>
  </body>
</html>
```

3.7 Our Final Program

Finally, we'll see a program that also asks for your age and does some simple math on it.

```
<html>
  <body>
    <script>
      var name = prompt("What's your name?"),
          age  = prompt("What's your age?");
      alert("Hello, " + name +
            ". Half of your age is " + age / 2 +
            ". Triple your age is " + age * 3);
    </script>
  </body>
</html>
```

Notice that we have one single variable assignment statement that is declaring and defining two variables as the result of prompting the user? We can split a statement across lines like this if we need to. Notice the use of the comma to separate out definitions. We could have used two separate statements. Either is fine.

Next, we've split the statements with the `alert` on it across three lines, and we're making extensive use of the string concatenation operator to join some response string together into the one alert, and using some math operators on the age value to tell the user some things about their age. We use the `/` operator to divide one number by another, and the `*` operator to multiply two numbers.

Now we'll see a program that does exactly the same thing, but goes about it slightly differently. Use this to compare to the previous one.

```
<html>
  <body>
    <script>
      var name, age, alert_value;
      name = prompt("What's your name?");
      age = prompt("What's your age?");
      alert_value = "Hello, " + name + ". " +
                    "Half of your age is " + age / 2 ". " +
                    "Triple your age is " + age * 3);
      alert(alert_value);
    </script>
  </body>
</html>
```

We notice that we're **declaring** three variables by using the `var`

keyword. This simply tells JavaScript that we want to set these names aside as variables that we'll assign values to them later. Then, in separate statements we're assigning values to these variables. These are similar values to the program before, but we're using a variable for the call to `alert` this time instead of putting the expression directly into `alert` as an **argument**.

3.8 Homework

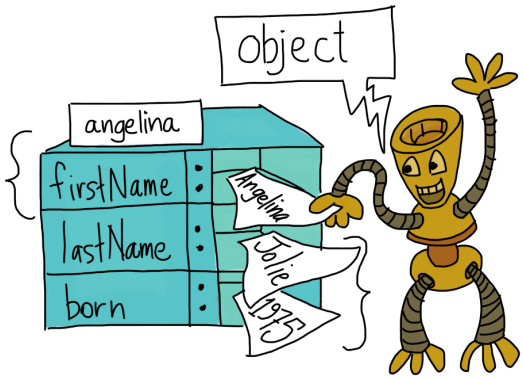
Your homework is to try adjusting the strings of either of the final programs and seeing what effect this has on the program. Don't try to do anything tricky, though. We'll see more things in good time. We don't want you to get confused or frustrated. Make sure you try inputting each of the programs in this chapter and try them all out at least once.

4 Guess the Number

Making a simple guessing game, we'll introduce **objects**, setting & retrieving **properties** on objects, calling **methods** on objects, the `Math` object, `Math.floor`, `Math.random`, the `if` statement and conditionals.

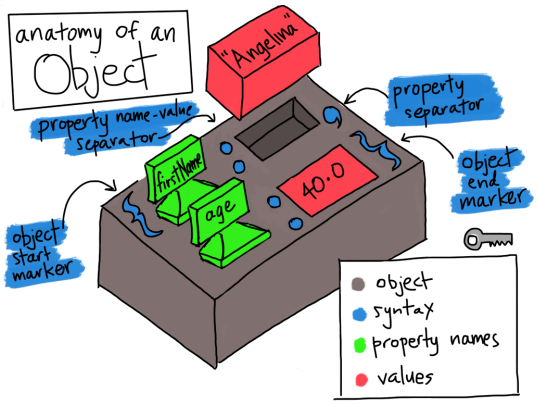
Our game will make up a secret number, and we'll take some turns guessing it. Make a new file called `guessing_game.html` and open it in our browser.

4.1 Objects



Javascript has a concept called **objects**, which is a way of bundling a bunch of named values into a single value. For example, we might decide to model some information about Angelina Jolie:

```
<html>
  <body>
    <script>
      var angelina = { firstName: "Angelina",
                       lastName: "Jolie",
                       born: 1975 };
    </script>
  </body>
</html>
```



4.2 Object Properties

This program simply creates a variable called `angelina` which is an object that bundles three **properties** together, named `firstName`, `lastName` and `born`. The fancy squiggly brackets (called braces) are how you define an **object literal** in JavaScript. Each named place for data in an object is called a **property** of the object. The properties are written as name-value pairs, and are separated from each other by commas. Each property has a name, then a colon and then its value. This is kind of like having a bunch of variables inside a variable!

If we want to get one of the property values out, we can do so pretty easily by using a dot and its name, like this:

```
<html>
  <body>
    <script>
      var angelina = { firstName: "Angelina",
                       lastName: "Jolie",
                       born: 1975 };
      alert(angelina.firstName + " " +
            angelina.lastName + " was born in " +
            angelina.born);
    </script>
  </body>
</html>
```

Running this will alert `"Angelina Jolie was born in 1975"`. So you can use `angelina.firstName` to get the data out of the `firstName` property of the `angelina` object. This

works for whatever the property is named. You can also use `angelina.lastName = "Angoyloona"` to set the last name to something else (Angoyloona here). This will actually change the value of the `lastName` property of the object.

4.3 Object Methods

Functions are also values, so if an object's property is a function, it's called a **method** and we can call it just like a regular function. We can retrieve it using the same **syntax** that we saw for retrieving property values, but if we add parentheses to the end, we can call it just like any function.

Ok, that's enough to do with Angelina.

4.4 Math and Random

JavaScript has a whole range of objects built into it with properties and methods for doing all kinds of things.

For our guessing game we're going to use an object called `Math` that has a method called `random`. Each time `random` is called, it will return a new random number value between `0` and `1` (but never actually `1`).

Let's see a program that alerts a random number every time you refresh the page:

```
<html>
  <body>
    <script>
      alert(Math.random());
    </script>
  </body>
</html>
```

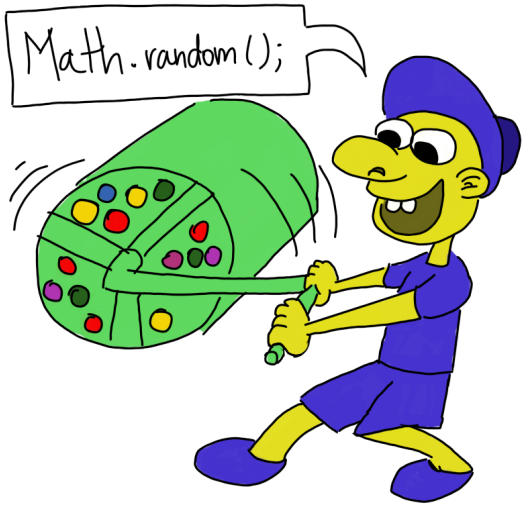
Try it out a few times, by loading then reloading the page.

We'd like our program to give us a number between `1` and `10`, so let's start see how to build toward this functionality.

4.5 Cleaning The Number Up

By using the `*` operator to multiply the result of `random` by `10` — we can call this "scaling" the result — we can get a number between `0` and `9`. We'll use some variables to show our steps, too:

```
<html>
  <body>
    <script>
      var secret_float =
        Math.random();
      var secret_float_scaled =
        secret_float * 10;
      alert(secret_float_scaled);
    </script>
  </body>
</html>
```



4.6 Floats

Why name them float? Well, these numbers are called **floating point** numbers, it's just the name of the type of numbers, they're sort of like decimals, but less accurate. Our number is scaled by a factor of 10, then displayed as an alert.

Ok this number is between 0 and 9, so we have to add one to get it between 1 and 10. We'll call this shifting it, and name our variable accordingly:

```
<html>
  <body>
    <script>
      var secret_float = Math.random();
      var secret_float_scaled =
        secret_float * 10;
      var secret_float_scaled_shifted =
        secret_float_scaled + 1;
      alert(secret_float_scaled_shifted);
    </script>
  </body>
</html>
```

If you want to, reload the page with this new program until you're confident it's giving us different numbers between 1 and 10.

Ok, but we don't actually want the numbers after the decimal point. Removing them can be done by another method of the `Math` object called `floor`, which will take anything after the decimal point and round it down to zero. This means we now have the secret number we want for our guessing game!

```
<html>
  <body>
    <script>
      var secret_float = Math.random();
      var secret_float_scaled =
        secret_float * 10;
      var secret_float_scaled_shifted =
        secret_float_scaled + 1;
      var secret_number =
        Math.floor(secret_float_scaled_shifted);
      alert(secret_number);
    </script>
  </body>
</html>
```

4.7 A Little Refactoring

Ok we'll stop at this point and do some of what's called **refactoring**. This is where we improve the way the code works. In our case, we'll collapse the four lines into one single variable assignment and expression, for simplicity:

```
<html>
  <body>
    <script>
      var secret_number =
        Math.floor(1 + Math.random() * 10);
      alert(secret_number);
    </script>
  </body>
</html>
```

This is an expression that does all of the previous code in one single step. It calls `Math.floor` on the result of calling `Math.random`, multiplying by 10 and adding 1 to it. We can do this because each expression results in a value, which in turn is fed in as argument(s) to another function or expression.

Now we can ask the user for their guess:

```
<html>
  <body>
    <script>
      var secret_number =
        Math.floor(1 + Math.random() * 10);
      var guess =
        prompt("Guess a number between 1 and 10");
    </script>
  </body>
</html>
```

4.8 The If Statement

We want the computer to tell the user if they got the answer right. To do this, we'll see a new statement called the `if` statement.

```
<html>
  <body>
    <script>
      var secret_number =
        Math.floor(1 + Math.random() * 10);
      var guess =
        prompt("Guess a number between 1 and 10");
      if(guess == secret_number) {
        alert("You got it!");
      }
    </script>
  </body>
</html>
```

We can see that `if` looks similar to a function call, except it has some braces afterward that are wrapping a statement with a call to `alert` in it. The `if` statement is how we can get our program to check the truth of something, such as an expression, and do different things depending on that check.

We can put as many statements as we like within these braces. These are not being used to show an object definition here, they're the same type of braces, but they're doing something different: they're being used to create a **block** of code. That is, this `if` statement will test the expression within its parentheses and if that expression evaluates to `true`, it will execute the block of code that follows it in sequence. If the test expression is `false`, it won't do anything.

Here we're using the `==` operator, which tests whether the expression on its left is equal to the expression to its right. Here that means `if` guess is equal to the secret number!

4.9 Giving Another Chance

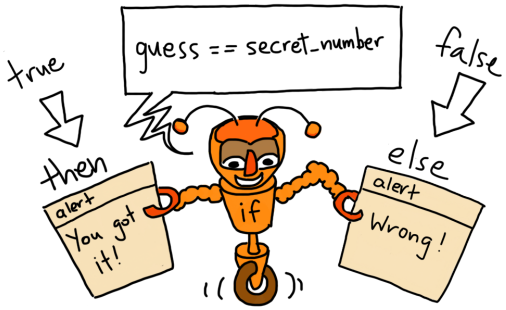
When you play this game, you'll quickly realise it's no fun: we only get one chance, and it doesn't tell us when we're wrong. Let's make it more playful.

There is more to the `if` statement... we can use an `else` statement on the end to indicate what the program should do if the test fails.

Let's put something telling us what the number was when we were wrong. This won't make it any more fun, but we'll get to that.

```
<html>
  <body>
    <script>
      var secret_number =
        Math.floor(1 + Math.random() * 10);
      var guess =
        prompt("Guess a number between 1 and 10");
      if(guess == secret_number) {
        alert("You got it!");
      } else {
        alert("Wrong. The number was " +
          secret_number);
      }
    </script>
  </body>
</html>
```

So if the test expression evaluates to `false`, the second block of statements executes. If it evaluates to `true`, the first block executes.



Notice we've also added another variable but set it to `null` for now, called `guess2`. We're going to embed another guess in the failing block, which will mean we can give the user a second chance at guessing. Don't be worried, we'll explain how it works:

```
<html>
  <body>
    <script>
      var secret_number =
        Math.floor(1 + Math.random() * 10);
      var guess =
        prompt("Guess a number between 1 and 10");
      var guess2 = null;
      if(guess == secret_number) {
        alert("You got it!");
      } else {
        if(guess < secret_number) {
          guess2 =
            prompt("Guess a number between " +
            guess + " and 10");
        } else {
          guess2 =
            prompt("Guess a number between 1 and " +
            guess);
        }
        if(guess2 == secret_number) {
          alert("You got it!");
        } else {
          alert("Wrong. The number was " +
            secret_number);
        }
      }
    </script>
  </body>
</html>
```

If you try this program out a few times you'll realise while it's still hard, it's a actually possible to guess now, because this program gives you a second chance, and tells you which side of your guess the secret number is on after your first guess.

If we just look at the block inside the first `else`, we'll see there are two `if`...`else` statements. The first compares the guess to the secret and gives the user a different prompt depending on the answer, and the second `if`...`else` statement gives them their results.

Notice that the last two `if`...`else` statements and their blocks are all included within the first else block?

Blocks can contain other blocks just fine, and can have more than one statement inside them, too.

The `if` statement is what's called a **conditional**, which means it

changes what the program does depending on some values or other expressions.

4.10 Homework

Your homework is to type each of these programs in and try them out. Also, notice in the `secret_number` expression, the `*` operator is performed before the `+` operator, because of order of operations, just like in arithmetic in maths.

The second part of your homework is to try, without looking, to create a single line program that puts a message window on the screen with a greeting on it, similarly to what we saw in the "Display a Message" chapter. Try to do it without looking it up. If you have to look, do it while looking, but then try again in an hour or so. We're trying to connect up making a single line program into your memory. Try as many times as you need to until you can do it without looking at all.

The third part of your homework is to make a single line program, but to try to break it in as many ways as you can think of. See what happens when you miss pieces off, or when you do crazy things to your program. Get familiar with all the ways you can fail. If we make failure our friend, it won't be so bad when we unexpectedly see it in a program, because we'll already have lots of experience with all the ways things can go wrong. This is the mark of a master.

5 End of the Free Sample

5.1 Thanks!

We really hope you've enjoyed this free sample of the book.

Please consider purchasing the full book, and letting as many people know about our works as possible.

This will enable us to make more great works like this for you in the future!

— GetContented

`http://happylearnjavascripttutorial.com`