Preboot - Functions (Concepts)

Let's say that you are writing a piece of code that has five different places where it needs to print your name. As mentioned above, for code that you expect to call often, separate this out into a different part of your file, so these lines of code don't need to be duplicated each time you print your name. This is called a **FUNCTION.** Creating (or *declaring*) a function could look like this:

```
function sayMyName( )
{
    console.log("My name is Martin");
}
```

By using the special **function** word, you tell JavaScript that what follows is a set of source code that can be called at any time by simply referring to the **sayMyName** label. Note: *the code above does not actually call the function immediately;* it sets the function up for other code to use (call) it later.

'Calling' the function is also referred to as *'running*' or *'executing'* the function. If the above is how you declare a function, then the below is how you actually *run* that function:

```
sayMyName();
```

That's it! All you need to do is call that label, followed by open and close parentheses. The parentheses are what tell the computer to execute the function with that label, so don't forget those.

One last thing: there is nothing stopping a function from calling other functions (or in certain special situations, even calling itself!). You can see above that the **sayMyName** function does, in fact, call the built-in **console.log** function. Naturally, you would not expect **console.log** to run *until* you actually called it; in the same way, any code that you write will only start running when some other code calls it (maybe part of the computer browser or the operating system). So, except for the very first piece of code that runs when a hardware device starts up, all other code runs only because other code called it.

To review: when we *declare* a function, it allows some other *caller* to execute our function from some other place in the code, at some other time. It does not run the function immediately. So, if your source code file contains this:

```
function sayMyName( )
{
    console.log("My name is Martin");
}
```

…and then you execute that source code file, nothing would actually appear in the developer console. This is because no code ever called **sayMyName()**. You set it up but never used it.

Preboot - Loops (Concepts)

Sometimes you will have lines of code that you want to run more than once in succession. It would be very wasteful to simply copy-and-paste that code over and over. Plus, if you ever needed to change the code, you would need to change all those lines one by one. What a mess! Instead, you can indicate that a section of code should be executed some number of times. Consider the following: “Do the next thing I tell you four times: hop on one foot.” That would be much better than “Hop on one foot. Hop on one foot. Hop on one foot. Hop on one foot.” (even though it is just as silly) Programming languages have the concept of a *LOOP* that is essentially a section of code that will be executed a certain number of times. There are a few different types of loops. The most common are **FOR** and **WHILE** *loops*. Shall we explore each of them in turn? Yes, let’s.

FOR Loops

FOR loops are useful when you know how many times those lines of code will run. **WHILE loops** are slightly better when you don’t know how many times to loop, but you will loop while a certain test continues to be true. To create a **FOR loop**, in addition to the code to be looped, you specify three things within parentheses following the **FOR:** any initial setup, a test that must be true in order to start the loop, and any code to be run at end of each time through the loop. Here is an annotated example:

```
//     A ;         B ;         D
for (var num = 1; num < 6; num = num + 1)
{
    // C
    console.log("I'm counting! The number is ", num);
}
// E
console.log("We are done. Goodbye world!");
```

The above will execute in this sequence: A - B-C-D - B-C-D - B-C-D - B-C-D - B-C-D - B - E.

Let’s walk through this **FOR** loop in detail. Up front, local variable **num** is created and set to a value of 1. This *step A* happens exactly once, then we start looping. *Step B*: we compare **num** to 6. If it is less than 6, then the code within curly braces (*step C*) is executed, and then 1 is added to **num** (*step D*). We then return to step B. When the test at step B fails, we immediately exit without executing step C or step D. At that point, execution continues from step E, following our closed-curly-brace.

Said another way, we INIT, then we [TEST? – BODY – INCREMENT] while TEST is true, then we exit.

```
for(INITIALIZATION; TEST; INCREMENT/DECREMENT)
{
    // BODY of the loop –
    // this runs repeatedly as long as TEST is true
}
```

WHILE Loops

WHILE loops are similar to **FOR** loops, except with two pieces missing. First, there is no upfront setup like is built into a **FOR** loop. Also, unlike a **FOR** statement, a **WHILE** doesn’t automatically include code that is executed at the end of each loop (our D above). **WHILE** loops are great when you don’t know how many times (iterations) you will loop. Any **FOR** loop can be written as a **WHILE** loop. For example, the above **FOR** loop could be written instead as this **WHILE** loop, which would execute identically:

```
// A
var num = 1;
// B
```

```
while (num < 6)
{
    // C
    console.log("I'm counting! The number is " + num);
    // D
    num = num + 1;
}
// E
console.log("We are done. Goodbye world!");
```

Behaving identically with the above **FOR** loop, the **WHILE** code written immediately above will execute in this sequence: A - B-C-D - B-C-D - B-C-D - B-C-D - B-C-D - B - E.

Let's review before we move on. Anything we do with a **FOR** loop, we could achieve with a **WHILE** loop instead – and vice versa. So, when should we use **FOR** loops, and when should we use **WHILE** loops? Generally, use **FOR** loops when you know *exactly* how long a loop should run. Use **WHILE** loops when you have a condition that keeps the loop running (or that will cause the loop to stop), but you aren't sure exactly how many iterations that will require.

Other Loop Tips

Some developers like to increment a variable's value by running **num += 1;** this is the same as typing **num = num + 1.** You may sometimes see **num++** or even **++num;** both are equivalent to **+=**.

```
var index = 2;
index = index + 1;
index++;
// index now holds a value of 4
```

By that same token, we can decrement the value of num by running simply **num--;** or **--num;**. This is exactly the same as running **num = num – 1;** or **num -= 1.** There are ***=** and **/=** operators as well, that multiply and divide a number as you might expect.

```
var counter = 5;
counter = counter - 1; // counter now holds a value of 4
counter--; // counter is now 3
counter *= 6; // counter is 18
counter /= 2; // counter == 9
```

Furthermore, not every loop must increment by one. Can you guess what the following would output.

```
for(var num = 10; num > 2; num = num - 1)
{
    console.log('num is currently', num);
}
```

Yes, that's right: this **FOR** loop counts backwards by ones, starting at 10, while num is greater than 2. So, it would count **10,9,8,7,6,5,4,3**.

How would you print all *even* numbers from 1 to 1000000? How would you print all the *multiples* of 7 (7, 14, ...) up to 100? Understanding how to use **FOR** loops is critical, so get really familiar with this.

Preboot - Loops and Code Flow

With more complex loops, you might need to break out of a loop early or to skip the current pass but continue looping. In JavaScript, you can use special **BREAK** and **CONTINUE** keywords to do this. The **break** keyword *immediately exits the specific loop you are currently in* and continues immediately following the loop. Even the final end-loop statement (**num = num + 1** above) will not be executed.
A **continue** skips the rest of the current pass through the loop, but any loop-end statement is executed and looping will continue. With both, once they run, any subsequent code within the loop is skipped.

Here's an example. The following code prints the first two lines, but then immediately exits the loop.

```
var num = 1;
while(num < 5)
{
    if(num == 3)
    {
        break;
        // if you have code here, it will never run!
    }
    console.log("I'm counting! The number is ", num);
    num = num + 1; // if we break, these lines won't run
}
//I'm counting! The number is 1
//I'm counting! The number is 2
```

The below counts from 1 to 4, printing something about each number, but completely forgets about 3, because when num == 3, a **continue** skips the rest of the loop and proceeds (after adding 1 to num).

```
for(var num = 1; num < 5; num += 1)
{
    if(num == 3)
    {
        continue;
        // if you have additional code down here, it will never run!
    }
console.log("I'm counting! The number is ", num);
}
//I'm counting! The number is 1
//I'm counting! The number is 2
//I'm counting! The number is 4
```

Loops commonly use the **break.** Get comfortable using it to loop for a number of iterations, then exit when you encounter a certain condition. With this, it isn't preposterous to see **while(true)**! My goodness.

Preboot - Parameters

Being able to call another function can be helpful for eliminating a lot of duplicate source code. That said, a function that always does exactly the same thing will be useful only in specific situations. It would be better if functions were more flexible and could be customized in some way. Fortunately, you can pass values into functions, so that the functions can behave differently depending on those values. The caller simply inserts these values (called *arguments)* between the parentheses, when it executes the function. When the function is executed, those values are copied in and are available like any other variable. Specifically, inside the function, these copied-in values are referred to as *parameters.*

For example, let's say that we have pulled our friendly greeting code above into a separate function, named **greetSomeone.** This function could include a parameter that is used by the code inside to customize the greeting, just as we did in our standalone code above. Depending on the argument that the caller sends in, our function would have different outcomes. Tying together the ideas of functions, parameters, conditionals, and printing, this code could look like this:

```
function greetSomeone(person)
{
    if (person == "Martin")
    {
        console.log("Yo dawg, howz it goin?");
    }
    else
    {
        console.log("Greetings Earthling!");
    }
}
```

You might notice in the code above that there are curly braces that are not alone on their own lines, as they were in the previous code examples. The JavaScript language does not care whether you give these their own line or include them at the end of the previous line, as long as they are present. Really, braces are a way to indicate to the system some number of lines of code that it should treat as a single group. Without these, **IF..ELSE** and **WHILE** and **FOR** statements will only operate on a single line of code. Even if your loop is only a single line of code (and hence would work without braces), it is always safer to include these, in case you add more code to your loop later – and to reinforce good habits.

So, is it better to include these on their own lines, or to append them to the ends of the previous lines? This is really a matter of choice: as long as you *include them* (and you always should), JavaScript doesn't care about a few extra new-line characters or extra spaces. Over time you will develop your own **coding style**, writing source code in the way that is most understandable to you. Keep in mind though that when you join a software team, you will likely need to adopt the team's coding style (if they have one). If they don't, then even in that case you should generally match the style of existing code in the source files where you are working. So, as you develop your personal style, it is best to stay flexible.

Preboot - Review

Hopefully, this first chapter made you more comfortable with the essential building blocks of software. Below is a summary of the ideas we covered.

Computers can do amazing things but they need to be told what to do. We tell them what to do by running software. Software is generally built from source code, which is readable by humans and is a sequence of basic steps that a computer will follow exactly. There are many different software languages (such as JavaScript), with different ways of expressing these basic steps, however, most of the main concepts are universal. Our job is to break down problems into these steps, and then the computer will *run* that code when told. Generally, when source code is *run, it executes from the first line linearly to the last line. However, we can change this flow by adding "fork-in-the-road" (conditional) or "do-that-part-a-few-times" (loop) structure to our source code.*

A variable is a labeled, local space that can contain a value. We refer to a variable by its label if we want to read or change that value. Values can be a few different types, such as *numbers* or *strings* or *booleans*. A string is a sequence of characters, and a boolean is simply a true/false value. JavaScript (also known as JS) automatically changes values from one type to another as needed.

A single-equals (=) is used to set values in variables, and can be combined with normal mathematical operators **(+ - * /)**. The **==** operator compares two values, allowing JS to convert data types if needed; the **===** operator does not allow the types to be converted. We print to the developer console using the **console.log** function, which accepts a string (or converts inputs to a string).

We can examine the values of variables, and divert the flow of source code execution depending on those values, using the **IF** statement. This can be combined with an **ELSE**, to cover the other side of a conditional as well; these **IF…ELSE** statements can be nested. One can create compound comparisons using logical operators that represent *and, or* and *not* (**&&, ||, !**).

To execute a specific piece of source code multiple times, there are two different types of loops available. A **FOR** loop is particularly useful when you know exactly how many times you need to loop; in other cases, a **WHILE** loop is simple and flexible. With both kinds of loops, you can use **BREAK** and **CONTINUE** statements to change the flow of code (to exit the loop or skip a certain iteration).

We can extract a piece of source code into a **FUNCTION** so that it can easily be called repeatedly. Functions (like variables) have labels, and to call a function, we list the function name with **()** following it. A function can require one or more values from outside, and we pass those values in by using parameters, which are included between the parentheses when calling the function, and similarly specified between the parentheses when defining the function.

The **{** and **}** characters are used to group code. How you choose to space (or compact) your source code will determine your personal **coding style**.