Preboot - Intro.

Computers, Software, and Source Code

Computers are amazing. They rapidly perform complex calculations, store immense amounts of data, and almost instantly retrieve specific bits of information from mountains of data. In addition to being fast, they appear superhuman in their ability to use the information to make decisions. How do they do it?

Simply put, computers are machines. We as humans are skilled at creating tools that perform *specific* tasks *very* well: a toaster, for example, or a lawnmower. Computers are tools built from components such as semiconductor *chips*, and ongoing advances in material science enable these pieces to get smaller and faster with every successive year (see *Moore's Law*). This is why computers have become so breathtakingly fast, but it only partially explains why they can seem so *smart*. It doesn't really tell us why they are so universally relied upon to solve such a diverse range of problems across today's world.

Computing is exciting because computing devices are flexible – they can be *taught* to do things not imagined when they were originally built. Science fiction may become a science in the future, but for today they only know what we *tell* them; they only do as they are told. Who *teaches* them, who *tells* them what to do? Software engineers, developers, programmers! You are reading this, so you probably intend to be one too. From a computer's viewpoint, you are training to become an educator. (-:

How do programmers tell computers what to do? We create **Software**. Software is a sequence of instructions that we build and provide to a computer, which then "mindlessly" runs those instructions. Computers cannot natively understand human language, nor can humans read the language of semiconductor components. To talk with computers, we need a "go-between" format: something that software engineers can understand, yet can also be translated into machine language instructions.

Many of these "go-between" languages have been created: PHP, Python, Ruby, JavaScript, Swift, C#, Java, Perl, Erlang, Go, Rust, and others – even HTML and CSS! Each language has differing strengths and therefore is useful in different situations. Programming languages all do essentially the same things though: they read in a series of human-readable steps instructing a computer how to respond, and they translate the steps into a format the computer can understand and later execute.

All these sequences of instructions, written in programming languages like JavaScript, are what we call Source Code. How and when this source code is translated into *machine* code will depend on the language and the machine. *Interpreted languages* like PHP, Python, and Ruby translate from source code into machine code "on the fly", immediately before a computer needs it. *Compiled languages* do some or all of this translation ahead of time. As we said earlier, a computer simply follows instructions it was given. More specifically, though, it executes *(runs)* machine code that was built *(translated)* from some piece of source code: code that was written by a software engineer.

Let's teach you how to think like a computer, so you can write effective source code. We have chosen to use JavaScript as the programming language for this book, so along the way, you'll learn the specifics of that language. With very few exceptions, however, these concepts are universal.

Preboot - Code Flow

When a computer executes (runs) a piece of code, it simply reads each line from the beginning of the file, executing it in order. When the computer gets to the end of the lines to execute, it is finished with that program. There may be *other* programs running on that computer at the same time (e.g. pieces of the operating system that update the monitor screen), but as far as your program goes, when the computer's execution gets to the end of the source you've given it, the program is done and the computer has completed running your code.

It isn't necessary for your code to be purely linear from the top of the file to the end of the file, however. You can instruct the computer to execute the same section of code multiple times – this is called a program **LOOP.** Also, you can have the computer jump to a different section of your code, based on whether a certain condition is true or false – this is called an **IF-ELSE** statement (or a conditional). Finally, for code that you expect to use often, whether called by various places in your own code or perhaps even called by others' code, you can separate this out and give it a specific label so that it can be called directly. This is called a **FUNCTION**. More on each of these later.

Preboot - Variables

Imagine if you had two objects (a book and a ball) that you wanted to carry around in your hands. With two hands, it is easy enough to carry two objects. However, what if you also had a sandwich? You don’t have enough hands, so you need one or more containers for each of the objects. What if you had a box with a label on it, inside which you can put one of your objects. The box is closed, so all you see is the label, but it is easy enough to open the box and look inside. This is essentially what a *variable* does.

A variable is a specific spot in memory, with a label that you give it. You can put anything you want into that memory location and later refer to the value of that memory, by using the label. The statement:

```
var myName = 'Martin';
```

creates a variable, gives it a label of **myName,** and puts a value of **"Martin"** into that memory location. Later, to reference or inspect the value that you stored there, you simply refer to the label **myName**. For example (jumping ahead to the upcoming Printing to the Console topic), to *display* the value in the variable **myName**, do this:

```
console.log(myName);
```

Preboot - Data Types (Concept)

Containers exist to hold things. You would create a variable because you want it to store some value –some piece of information. A value could be a number, or a sentence made of text characters, or something else. Specifically, JavaScript has a *few data types,* and all values are one of those types. Of these six data types, three are important to mention right now. These are Number, String, and Boolean.

In JavaScript, a *Number* can store a huge range of numerical values from extremely large values to microscopically small ones, to incredibly negative values as well. If you know other programming languages, you might be accustomed to making a distinction between integers and floating-point numbers – JavaScript makes no such distinction.

A *String* is any sequence of characters, contained between quotation marks. In JavaScript, you can use either single-quotes or double-quotes. Either way, just make sure to close the string the same way you opened it. **'Word'** and **"wurd"** are both fine, but **'weird"** and **"whoa!'** are not.

Finally, a *Boolean* has only two possible values: **true** and **false**. You can think of a Boolean like a traditional light switch, or perhaps a yes/no question on a test. Just as a light switch can be either *on* or *off,* and just as a yes/no question can be answered with either yes or no, likewise a Boolean must have a value of either **true** or **false** – there is nothing in-between.

One of the main things we do with variables, once they contain information of a certain data type, is to compare them. This is our next section.

Preboot - Equals

Not All Equals Signs Are the Same!

In many programming languages, you see both **=** and **==.** These mean different things! Code **(X = Y)** can be described as *"Set the value of X to become the value of Y"*, and you can describe **(X == Y)** as *"Is the value of X equivalent to the value of Y?"* It is more common – but less helpful right now while learning these concepts – verbalize these as *"Assign Y to X" and "Are X and Y equal?"*, respectively.

Use **=** to **set** things, **==** to **test** things. *Single-equals is for assignment; double-equals is for comparison.*

Many programming languages are extremely picky about data types. When asked to combine two values that have different data types, some languages will halt with an error rather than do so. JavaScript, however, is not so strict. In fact, you could say that JavaScript is very loosely typed: it very willingly changes a variable's data type, whenever needed. Remember the **==** operator that we described previously? It *actually* means *"after converting X and Y to the same data type, are their values equivalent?"* If you want *strict comparison* without converting data types, use the === operator.

Generally, === is advised, unless you explicitly intend to equate values of differing types, such as **{1,true,"1"}** or **{0,false,"0"},** etc. (If this last sentence doesn't make sense yet, don't worry.)

Quick quiz:

- How many inputs are accepted by the **==** operator and the **===** operator, respectively?
- Do inputs to **==** and **===** operators need to be the same data type, for the operators to function?
- What is the data type of the output value produced by the **==** and **===** operators?

Hey! Don't just read on: *jot down answers* for those questions before moving on.

Done? OK, good….

Answers:

- The **==** and **===** operators both accept two values (one before the operator, and one after).
- *No*, two values need not be the same type for these operators. However, if they are not, **===** always returns **false**. The **==** internally converts values to the same type before comparing.
- The **==** and **===** operators both return a *Boolean* value (**true** or **false**).

Now that you know just a little about Numbers, Strings and Booleans, let's start using them.

Preboot - Console

Printing to the Console

graphical user interface. However, when we are first learning how to program, we start by having our programs write simple text messages (strings!) to the screen. In fact, the very first program that most people write in a new language is relatively well known as the “Hello World” program. In JavaScript, we can quickly send a text string to the *developer console*, which is where errors, warnings and other messages about our program go as well. This is not something that a normal user would ever look at, but it is the easiest way for us to print variables or other messages.

To log a message to the console, we use **console.log()**. Within the parentheses of this call, we put any message we want to be displayed. The **console.log** function always takes in a string. If we send it something that isn’t a string, JavaScript will first convert it to a string that it can print. It’s very obliging that way. So, our message to be logged could be a literal value (**42** or **"Hello"**), or a variable like this:

```
console.log("Hello World!");
var message = "Welcome to the Dojo";
console.log(message);
```

We can also combine literal strings and variables into a larger string for **console.log,** simply by adding them together. If you had a variable **numDays**, for example, you could log a message like this:

```
var numDays = 40;
console.log("It rained for " + numDays + " days and nights!");
```

Notice that we put a space at the end of our **"It rained for "** string so that the console would log **"It rained for 40 days and nights!"** instead of **"It rained for40days and nights!"**

So, what would the following code print?

```
var greeting = "howdy";
console.log("greeting" + greeting);
```

It would print **"greetinghowdy"**, since we ask it to combine the literal string **"greeting"** with the value of the variable greeting, which is **"howdy"**. Make sense?

One last side note: if you *really insist* upon writing a string to the actual web page, you could use the old-fashioned function **document.write(),** such as **document.write("Day #" + numDays).** However, you won’t use this function when creating real web pages, so why get into that habit now?