Javascript Basics

Primitive Variables

In computing, a variable is a reference to a position in memory that defines a particular value. They can reference other variables, and their values, or at least the value they're referencing, can be changed (with some exceptions which we'll get into in future sessions).

There are a few different ways to declare a variable in Javascript. We'll start with a simple example:

```
var myVariable
```

This code declares the generic variable "myVariable" exists, but doesn't assign a value to it.

Variable names must start with a letter and contain no spaces. Some special characters, like underscores, are allowed. Variable names can contain numbers as long as it isn't the first character. The most common Javascript naming convention for a mutable variable is camelCase, or having every word in a variable name capitalized except for the first one. If you're unfamiliar with variable naming conventions, I encourage you to look them up.

To assign a value to the variable, use the equals sign:

```
var myVariable = 20
```

So this line declares the "myVariable" variable and assigns a value of "20" to it.

So what good does this do us? We can use variables as placeholders for values in functions, equations, and more.

```
var myVariable = 20
console.log(myVariable)
```

Variables can also be set to reference another variable, or to the result of another variable:

```
var myVariable = 20
var myOtherVariable = myVariable + 10
console.log(myOtherVariable)
```

One important note: there's a difference between declaring and defining a variable. I'll break it up into extra lines so you can see the difference:

```
var myVariable
myVariable = 20
console.log(myVariable)
```

The first line *declares* the variable, telling the interpreter that it exists. The next line *defines* the value of the variable to be 20. The last line displays a message box containing the value of the variable.

As the name suggests, the value that variables reference can be changed at any time:

```
var myVariable
myVariable = 20
console.log(myVariable)
myVariable = 10
wconsole.log(yVariable)
```

Now that we know a bit how variables work, let's move on to talking about types of values that can be assigned to variables. Javascipt ES5 defines 5 primitive datatypes and one special type:

- Number
- String
- Boolean
- Null
- Undefined
- Object (the special type)

Functions are considered an Object data type that have the ability to be called. We'll talk about functions later.

ES6 also adds another type, but it's out of scope for this session:

- Symbol

The type of a value determines how it will compare or combine with other values. We can use a couple of operators to demonstrate this: the arithmetic addition `+` operator and the loose equality or double-equals `==` operator. Think of operators like built in special functions that take in input and generate output.

Let me write a small section of code that alerts the result of adding two variables together, and code that will display "true" or "false" depending on whether the variables are "almost" equal:

```
var a = 1
var b = 2
console.log(a + b)
console.log(a == b)
```

Number

In this example, both variables `a` and `b` are of the number type. According to MDN, a Number is "a numeric data type in the double-precision 64-bit floating point format". In English, it's any number between about negative 9 quadrillion and positive 9 quadrillion (including decimals) depending on the computer.

I'm not going to go over the quirks of floating point math in this session. Just know that a number is any numeric value. For those who have a Java or C background, there is no distinct primitive type for Integers (unsigned or otherwise) or floats vs doubles. A number is a number is a number.

String

I'll rewrite the code a bit to define two identical strings as values:

```
var a = 'this is a string'
var b = 'this is a string'
console.log(a + b)
console.log(a == b)
```

Strings are a sequence of characters "strung" together, and should be used to represent textual data.

The equality operator more or less works as you would expect: the two strings are compared, and if every character in the string is identical, then the strings are equal. However, the addition operator does something interesting: it displays the results of a combined (or concatenated) string. Meaning it behaves differently than the Number type. This is important to remember, and is the source for a lot of common issues in Javascript. For example, what do you think would happen if I changed the string values to '1' and '2'?

```
var a = '1'
var b = '2'
console.log(a + b)
console.log(a == b)
```

So '1' and '2', as strings, are combined to form '1' + '2', or '12'. Always ensure that the data types are right when trying to perform arithmetical operations.

Another thing to note: Javascript is a loosely typed language. This means that a variable can be set to any data type without declaring what it needs to be first, and it is allowed to change to a different type. A variable can start as a string and turn into a number for example:

```
var a = '1'
a = 10
var b = '2'
b = 20
console.log(a + b)
console.log(a == b)
```

Strictly typed languages like C# or C++ (at least traditionally) will not allow this.

Boolean

Named after George Boole, a boolean is a data type with two possible data types: `true` or `false`. It is perhaps the most fundamental data type there is in computing.

I'll adjust the above code a bit:

```
var a = true
var b = false
console.log(a + b)
console.log(a == b)
```

When adding two booleans together, the boolean is "cast" into a number: 0 for false, and 1 for true.

In Javascript, type casting is the action of turning the type of one variable into another. Because Javascript was designed more to be approachable to a wider audience and not to serve as an "enterprisey" solution like Java, Javascript is replete with implicit type casting operations. The `window.alert()` and `console.log()` function is turning the result of the conditionals into strings under the hood.

As stated earlier, resolving different types and type casting in Javascript is very complex and a very common source of problems, which is why superset languages like TypeScript exist. A good rule of thumb is to do your best to only have similar types of data interact with eachother.

On that note, I'll introduce a new type of conditional now: the strict equals (or triple equals): `===`:

```
var a = true
var b = true
console.log(a + b)
console.log(a == b)
console.log(a === b)
```

Actually I'll also introduce something else: comments. Comments are lines in a script that should be skipped. There are two ways to add comments in Javascript. I'll just cover a single line comment for now: just add two slashes on a line of code, and then everything after those slashes are skipped:

```
var a = true
var b = true
// console.log(a + b)
console.log(a == b)
console.log(a === b)
```

There, now no alert for "a + b" will appear.

Comments are useful when you need to explain why a line of code is doing something. They're purely for developer's convinience:

```
// Just running an experiment on different variable types
var a = true // Just showing that comments can be after a line that executes
var b = true
// console.log(a + b)
console.log(a == b)
console.log(a === b)
```

Anyway, back to the different types of equality. Two equals signs tell Javascript to try to resolve the variable types before checking if they're equal. Triple equals signs check to see if the data is the same type, and only if it is, performing the check.

In this case, both are true: both variables `a` and `b` are of the same type: Booleans (true or false), and they happen to be equal to one another. Nothing unusual here.

Let's change the variables a bit though:

```
// Just running an experiment on different variable types
var a = 12 // Just showing that comments can be after a line that executes
var b = '12'
// console.log(a + b)
console.log(a == b)
console.log(a === b)
```

In this case, the double-equals check returns "true", but the triple-equals check returns "false". Double-equals is casting the string into a number and then checking to see if the numbers are equal. Triple-equals notices that the types, Number and String, are not the same, and returns false outright.

In almost all cases, to avoid issues, you will want to use triple equals when comparing variables because, as the Zen of Python states, *Explicit is better than implicit*. And no, there is not a performance difference between the two operators because they are doing different things.

So why have double equals at all? Remember that Javascript was created as a language for the layman. Later when we're getting data out of input elements, the data will be in string format. For example, a number input tag might have a value of the string "12", not the number 12. Early in Javascript's development, it was decided that needing to have people learn the difference would be an obstruction. Unfortunately, this decision meant introducing weird data type issues for years to come.

But there is a legitimate use of double equals, which introduces our next Javascript data type.

Null

Null, according to MDN is "the intentional absence of an object value". It means a declared variable points to nothing.

```
// Just running an experiment on different variable types
var a // Just showing that comments can be after a line that executes
var b = null
// console.log(a + b)
console.log(a == b)
console.log(a === b)
```

With that description, it makes sense that loose equality, double equals, would return true. The variable `a` is declared, but is not defined. The variable `b` is intentionally defined as not being defined. So both `a` and `b` are variables that point to nothing, thus they're both equal.

Loosely checking equality against null is useful for checking if a variable exists, regardless of whether it lacks value intentionally or otherwise.

So why is strict equality, triple equals, returning false? Because in actuality, `a` is not `null`. Null is when data types are intentionally absent of value. `a` is implicitly without value, so what does that make it?

Undefined

If a variable has not been assigned a value, it is considered `undefined`. The most common place you'll see `undefined` variables is properties on objects that don't exist. For example, the global window object does not have a "banana" property defined, so if I did this:

```
// Just running an experiment on different variable types
var a // Just showing that comments can be after a line that executes
var b = null
console.log(window.banana)
// console.log(a + b)
console.log(a == b)
console.log(a === b)
```

It will display "undefined".

While it is possible to set a variable to the value of `undefined`, I would recommend against doing that. The purpose of an undefined value is to signify that the value has not been intentionally assigned to anything. Explicitly assigning a value to undefined goes against the spirit of this concept.

We covered a lot, so I think I'll stop here to allow you to grok the concepts. In summary, variables are a reference to a value of data in memory. There are six types of variables in ES5, and seven in ES6: Number, String, Boolean, Null, Undefined, Object, and Symbol.

Javascript is a loosely typed language and allows variables to change types and allows variables of different types to interact, which historically has lead to lots of issues. For this reason, you should be very careful when working with variables that are different types. And there's a difference between intentionally defining a variable with no value and implicitly declaring a variable with no value.

In the next session, we'll introduce the Object variable, and two of its derivatives: the array and function.