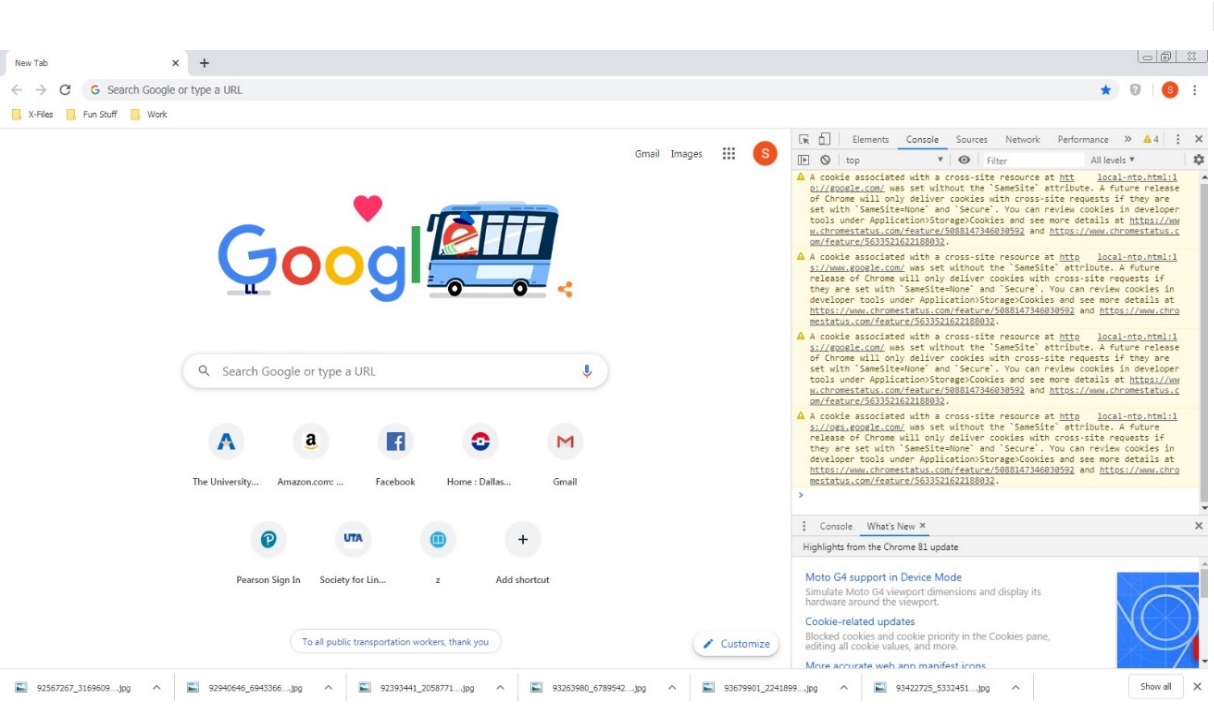
Some Basics of Javascript

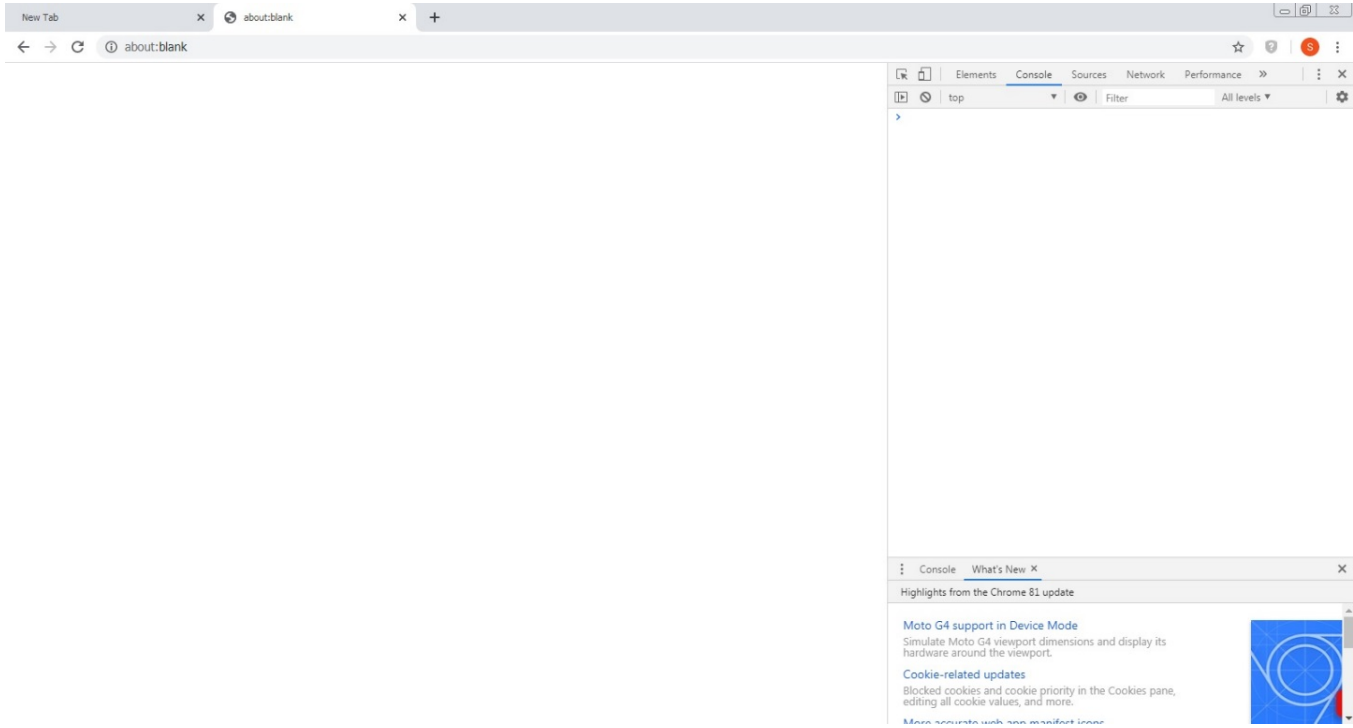
By Jason Schneider

Open the Console

In Google Chrome, you can press f12 to open the console, but you will see a lot of stuff going on that we won't need for now.

Instead type **about:blank** in the address bar, and press **f12**.





Types in Javascript

There are two types: primitive and reference.

Primitives are stored within the variable object, whereas **references** are points in memory pointed to within variable objects.

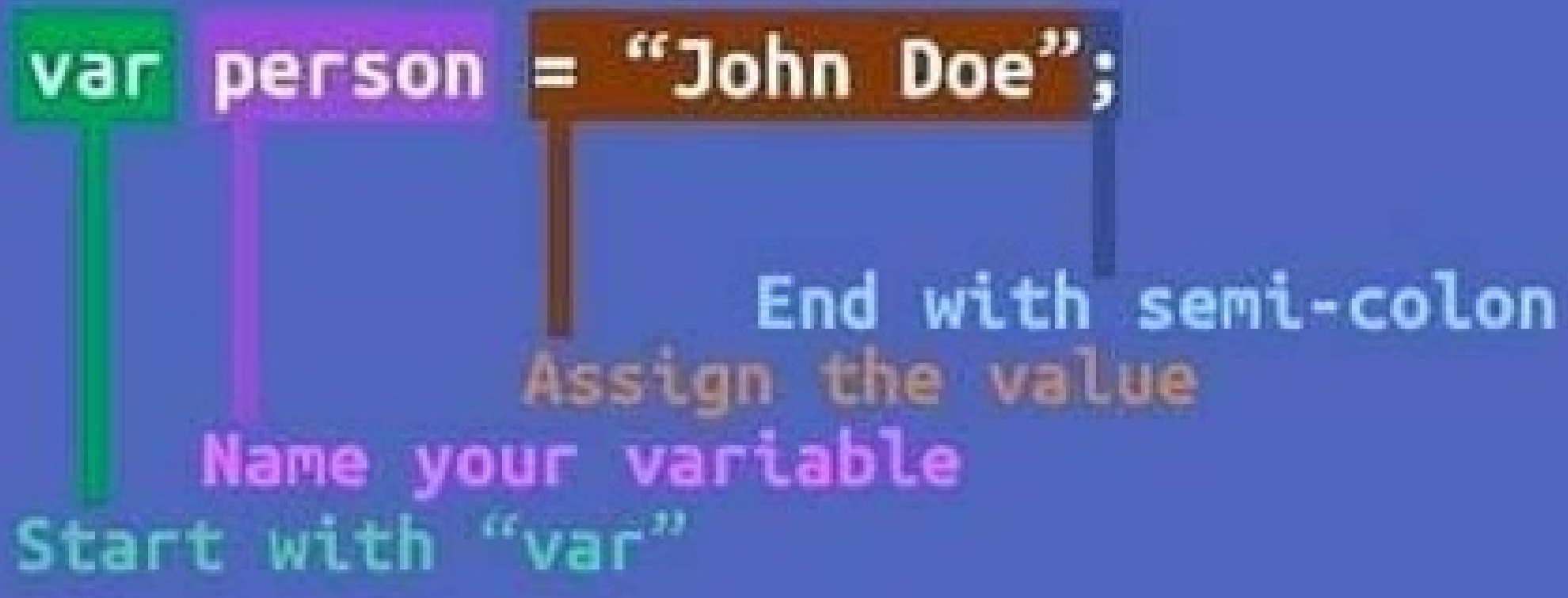
Primitives are immutable and are copied to new variables.

Referential types are mutable and point to a single point in memory, even though referenced by several variable objects.

Think of primitives like twins. When one of the twins has an accident, the other twin is not harmed.

References are like pet names a girl friend or boy friend might give to each other. One might call the other “honey”, but that doesn’t change who that person is being called honey or cause a twin to appear. That person being called honey is still Joe or Jane, but that person can take other numerous names to point to them such as honey, baby, or sugar.

HOW TO DEFINE A VARIABLE



COMMENT WITH // OR PAIR OF /* AND */

```
// This is a comment & will not be processed
var person = “John Doe”;

/* Whatever is enclosed within this pair of
comments will not be processed */
```

THE VARIOUS DATATYPES

STRING

```
var name = “John Doe”;
```

NUMBER

```
var x = 123;
```

BOOLEAN

```
var pass = true;
```

ARRAY

```
var animals = [”cat”, “dog”];
```

OBJECT

```
var person = {name:”John”, age:”99”};
```

EMPTY

```
var foo = null;
```

Primitive Types

Booleans indicate true or false values.

Numbers are any integer or floating-point numeric values.

Strings are characters or sequence of characters.

Null is a single value of being empty, indicated as null.

Undefined is a variable that is not initialized, (explain further here).

Primitives are stored directly into a variable object, so if a primitive is stored in one variable and another variable is set to equal that of the first variable, both variables will store separate data because the second variable copied the data in separate memory of the second variable object.

```
var color1 = "red";
var color2 = color1;

console.log(color1); //"red"
console.log(color2); //"red"

color1 = "blue";

console.log(color1); //"blue"
console.log(color2); //"red"
```

To be able to operate on variables, it is important to know something about their type.JavaScript variables can hold many data types e.g numbers, strings, objects and others.

Basic data types in Java Script

There are a number of data types in java script.But they can be grouped into three different categories.

a .primitive/primary
=>Numbers,String,Boolean
b .composite
=>Object,Array,Function
c.Special
=> Undefined,Null (Null listed as primitive in documentation but querying its typeof(),it shows as an object.Thats why I've put it in special category(devdocs.io/javascript/global_objects/null))

```
var length      =  16;                                  // Number
var lastName    =  "Johnson";                           // String
var person      =  {firstName:"John", lastName:"Doe"};  // Object
var Online      = True                                  // Boolean
```

NULL

var foo = null;

An explicit way to define an "empty nothing value".

UNDEFINED

var foo;

When a variable is not assigned any values.

EMPTY

var foo = "";

An explicit way to define an "empty string".

```
var name = "Carlos";

var firstName = name;

name = "Carla";

console.log(name); // "Carla"

console.log(firstName); // "Carlos"
```

Let’s unpack what’s going on:
We give a variable name the value of "Carlos“, and a place in memory is given to it.
Then, a copy of name's value is given to the variable firstName, and firstName is given a separate place in memory for it to reside and, thus, is not one and the same as the variable name. However, firstName also has a value of "Carlos". When we modify the value of name to "Carla“, firstName remains unaffected and still has the value of “Carlos” because it is stored in a different place in memory.

Identifying Primitive Types

Use of the typeof operator returns the type of primitives.

However, typeof will return object for null, maybe, because null is an empty object pointer.

To find whether a value is null, use the === operator to compare the value of a primitive with null. If the value is true, then it is truly null, but if the value is false, then it is not null.

```
console.log(typof "Jason"); //"string"
console.log(typof 10); //"number"
console.log(typof 5.1); //"number"
console.log(typof true); //"boolean"
console.log(typof undefined); //"undefined"
console.log(typof null); //"object"

//determine whether a value is truly null
console.log(value === null); //true or false
```

Making Comparisons

Using == converts strings into a number prior to comparing two types, whereas === does not make any coercions when making comparisons between types.

```
console.log("5" == 5); //true
console.log("5" === 5); //false

console.log(undefined == null); //true
console.log(undefined === null); //false
```

Primitive Methods

Each type of primitive has a number of methods.

Strings have methods to lower case and slice characters from a string.

Numbers have methods to convert numbers to floats and character strings.

Booleans has a method to change true or false to a string.

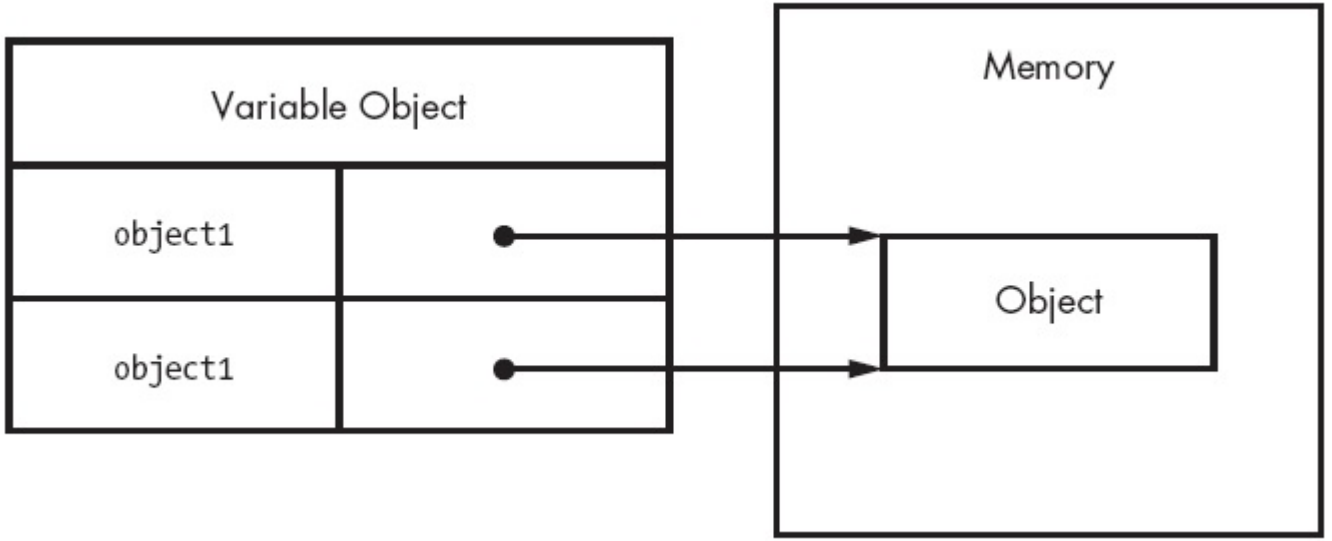
However, null and undefined do not have any methods.

```
var name = "Jason";
var lowercaseName = name.toLowerCase(); // convert to lowercase
var firstLetter = name.charAt(); //get first character
var name4 = name.charAt(2);
var name5 = name[2];
var middleOfName = name.substring(2,4); //get characters 2-3
var name7 = name.slice(2,4);

var count = 10;
var fixedCount = count.toFixed(2); //convert to "10.00"
var hexCount = count.toString(16); //convert to "a"

var flag = true;
var stringFlag = flag.toString(); //convert to "true"
```

Reference Types

Think of reference types as objects, and these objects point to data in memory rather than store the data in a variable like primitives.

Objects are like that of classes because Javascript objects are like hash tables. Reference types do not store object variables but holds a reference to the place in memory where the object is.




```
var object1 = new Object();
//you can create an empty object with var object1 = {}; as well
var object2 = object1;
object1.myCustomProperty = "Awesome!";
console.log(object2.myCustomProperty); // "Awesome!"

//free up memory by throwing away an object in memory
object1 = null; // dereference
```

Built-in Reference Types

Array An ordered list of numerically indexed values

Date A date and time

Error A runtime error (there are also several more specific error subtypes)

Function A function

Object A generic object

RegExp A regular expression

```
var items = new Array();
var now = new Date();
var error = new Error("Something bad happened.");
var func = new Function("console.log('Hi');");
var object = new Object();
var re = new RegExp("\\d+");
```

Variables

```
var a;                          // variable
var b = "init" ;                // string
var c = "Hi" + " " + "Joe";     // = "Hi Joe"
var d = 1 + 2 + "3";            // = "33"
var e = [2,3,5,8];              // arrav
var f = false;                  // boolean
var g = /()/;                   // RegEx
var h = function(){};           // function object
const PI = 3.14;                // constant
var a = 1, b = 2, c = a + b;    // one line
let z = 'zzz';                  // block scope local vari
```

Referential Objects

Objects can reference several properties, typically consisting of a key and value.

Keys can be strings, too, so that spaces and/or special characters between words can be used.

Alternatively, dot notation can be used to between words that make an key.

```
var book = {
name: "The Principles of Object-Oriented JavaScript",
year: 2014
};

var book = {
"name": "The Principles of Object-Oriented JavaScript",
"year": 2014
};

var book = new Object();
book.name = "The Principles of Object-Oriented
JavaScript";
book.year = 2014;
```

Referential Objects

```
var myName = {

  firstName: "Carlos"

};

var identity = myName;

myName.firstName = "Carla";

console.log(myName.firstName); //
"Carla"

console.log(identity.firstName); //
"Carla"
```

Referential types do not copy values in new place in memory. Let’s unpack what is happening here:

- First, a variable myName is declared with the value of an object which has a property called firstName. This property, or key, firstName has the value of "Carlos“, allocated in a place in memory for myName and its object.
- Then we create another variable called identity that points or references the variable myName. Notice that a new place in memory is not allocated to identity's value and only points to myName's value.
- The value of myName's firstName property can then be changed to "Carla" and replace "Carlos“ because the original place in memory created when the variable myName was created is modified, not copied, showing that referential types are mutable in contrast to primitives.

Passing Values by Reference

```
let x = {p:1}; //create new
variable x

let y =x; //y is a reference to x

x.p = 2; //change original value in
x

console.log(y.p); //2
```

```
let a = { p: 1}; //create new variable a
let b = a;
let c = b;
let d = c;
let e = d;
let f = e;
let g = f;
a.p = 5; //change original value in a
console.log(g.p); //5 now also in g.p
```

Referential types do not store data but merely point to data in memory, unlike primitives.

Exploring Objects in More Depth

Again, **objects** that have **keys and values** are mutable in Javascript. This differs from other programming languages like Python in which a data structure like objects in Javascript are immutable.

```
var person = { name: "Jason", superpower: "super strength" };
console.log(person); //{name: "Jason", superpower: "super strengt

var person2 = person;
console.log(person2.name); //”Jason”
console.log(person.name); //”Jason”
console.log(person2); //{name: "Jason", superpower: "super streng

person2.name = "Mike";
console.log(person2.name); //”Mike”
console.log(person.name); //”Mike”
console.log(person2); //{name: “Mike", superpower: "super strengt
console.log(person); //{name: “Mike", superpower: "super strength
```

Identifying Reference Types

- The typeof operator can identify objects that work as functions as the type being a function.
- However, the instanceof operator is better used with other kinds of objects because typeof will only return the type being an object if it is anything other than a function, whereas instanceof can be used to identify other reference types.

```
//typeof operator
function reflect(value) {
return value;
}
console.log(typeof reflect); // "function“

//instanceof operator
var items = [];
var object = {};
function reflect(value) {
return value;
}
console.log(items instanceof Array); // true
console.log(object instanceof Object); // true
console.log(reflect instanceof Function); // true
```

Identifying Reference Types

All reference types inherit from the general object reference type.

```
var items = [];
var object = {};
function reflect(value) {
return value;
}
console.log(items instanceof Array); // true
console.log(items instanceof Object); // true
console.log(object instanceof Object); // true
console.log(object instanceof Array); // false
console.log(reflect instanceof Function); // true
console.log(reflect instanceof Object); // true
```

Primitive Wrapper Types—The Most Confusing Part

```
var name = "Nicholas";
var firstChar =
name.charAt(0);


console.log(firstChar); // "N“
```

```
// what the JavaScript engine does
var name = "Nicholas"; //a primitive saved on a variable

var temp = new String(name); //temporary object referencing primitive
var firstChar = temp.charAt(0); //create and/or make change of primitive
temp = null; //immediately destructs the temporary object

console.log(firstChar); // "N“
```

Because the second line uses a string (a primitive) like an object, the JavaScript engine creates an instance of String so that charAt(0) will work. The String object exists only for one statement before it’s destroyed
(a process called *autoboxing*).

When working with regular objects, you can add properties at any time and they stay until you manually remove them. With primitive wrapper types, properties seem to disappear because the object on which the property was assigned is destroyed immediately afterward.

Primitive Wrapper Types—What's Happening behind The Scenes

```
var name = "Nicholas";

name.last = "Zakas";


console.log(name.last); //

undefined
```

```
// what the JavaScript engine does
var name = "Nicholas"; //a primitive saved on a variable
var temp = new String(name); //temporary object referencing primitiv
temp.last = "Zakas"; //create and/or make change of primitive
temp = null; // temporary object destroyed

var temp = new String(name);
//an empty reference temporarily created for the lookup of a property

console.log(temp.last); // undefined
temp = null;
```

Instead of assigning a new property to a string, the code actually creates a new property on a temporary object that is then destroyed. When you try to access that property later, a different object is temporarily created and the new property doesn't exist there. Although reference values are created automatically for primitive values, when instanceof checks for these types of values, the result is false:

Primitive Wrapper Types—The Weird Stuff

```
var name = "Nicholas";
var count = 10;
var found = false;

console.log(name instanceof String); //
false
console.log(count instanceof Number); //
false
console.log(found instanceof Boolean); //
false
```

```
//create primitive wrappers manually
var name = new String("Nicholas");
var count = new Number(10);
var found = new Boolean(false);

//just creates a generic object, not primitive values
console.log(typeof name); // "object"
console.log(typeof count); // "object"
console.log(typeof found); // "object"
```

The instanceof operator returns false because a temporary object is created only when a value is read. Because instanceof doesn’t actually read anything, no temporary objects are created, and it tells us the values aren’t instances of primitive wrapper types.

Operators in Javascript

ARITHMETIC

+	Plus, concatenation (addition)
++	Increment
-	Minus (subtraction)
--	Decrement
*	Multiply
/	Divide
%	Modulus (remainder)

BITWISE

&	AND
|	OR
^	XOR
~	NOT
<<	Shift left
>>	Shift right (retain sign)
>>>	Shift right (pad with zero)

CONDITIONAL

```
var x = CONDITION
      ? ON TRUE
      : ON FALSE
```

COMPARISON

>	More than
>=	More than or equals
<	Less than
<=	Less than or equals
==	Equals to
===	Equals to (with type checking)
!=	Not equals to
!==	Not equals to (with type checking)

LOGICAL

&&	AND
||	OR
!	NOT

ASSIGNMENT

=	Assign value to
+=	Add and assign
-=	Subtract and assign
*=	Multiply and assign
/=	Divide and assign
%=	Modulus and assign

Basic concepts about data Types

Does it make any sense to add "winstonmhango23" to 100 ?
Will it produce an error or will it produce a result?

Well in Java script it does,but not as you thought.It will treat the above as

```
var x = "winstonmhango23" + "100";
```

1.When adding a number and a string, JavaScript will treat the number as a string.

2.JavaScript evaluates expressions from left to right. Different sequences can produce different results:

```
var x = 16 + 4 + "Cars"; // produces 20Cars
var x = "Cars" + 16 + 4; // produces Volvo164
```

In the first example, JavaScript treats 16 and 4 as numbers, until it reaches "Cars".
In the second example, since the first operand is a string, all operands are treated as strings.

3.JavaScript has dynamic types. This means that the same variable can be used to hold different data types.

```
var x;          // Now x is undefined
x = 5;          // Now x is a Number
x = "John";     // Now x is a String
```

Coercion of Types

Javascript has an internal mechanism to keep it from breaking by coercion, but programmers should be careful because they might get something not intended.

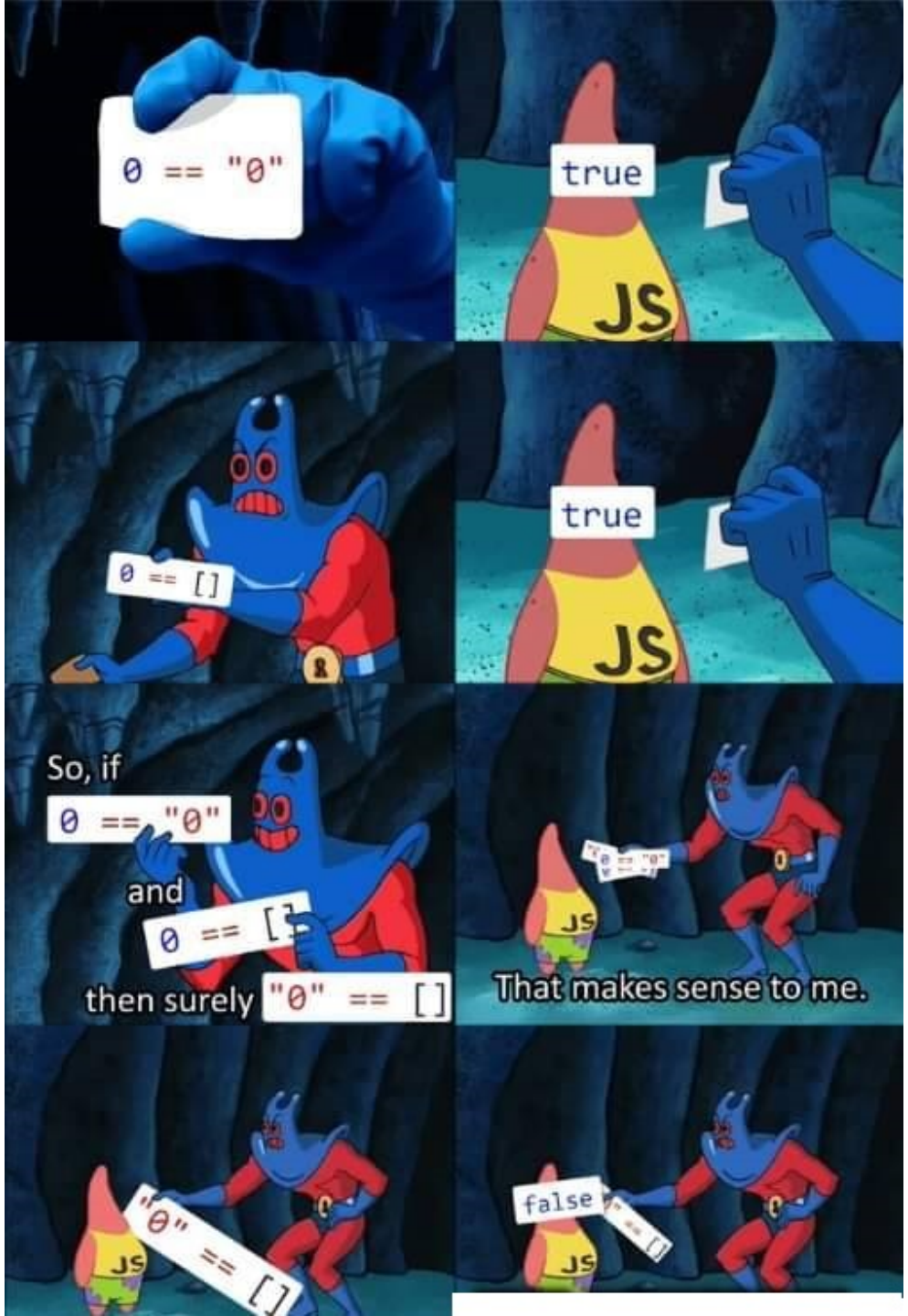
If we try the first example, Javascript will coerce those values as a string when the + operator encounters objects of incompatible types.

```
//try this first example
console.log(null + {} + true + [] + [5]);
//null[object Object]true5 <string>

//classic coercion cases sometimes produce surprising results
let a = true + 1; //becomes 1 + 1 or 2
let b = true + true; //becomes 1 + 1 or 2
let c = true + false; // becomes 1 + 0 or 1
let d = "Hello" + " " + "there."; // becomes "Hello there."
let e = "Username" + 1523462; //becomes "Username1523462"
let f = 1 / "string"; //NaN (not a number)
let g NaN === NaN; //becomes false
let h = [1] + [2]; //becomes "12" <string>
let i = Infinity; //remains Infinity
let j = [] + []; //becomes "" <string>
let o = [] + {}; //becomes [object Object]
```

- 0 == "0“ //true
- 0 == [] //true

- //zero is not a null integer just as an empty array is not really null

- "0" == [] //false



Passing Primitives in a Function

```
var myName = "Carlos";

function myNameIs(aName){

  aName = "Carla";

}

myNameIs(myName);

console.log(myName); // "Carlos"
```

A variable name is created and given the value of "Carlos". JavaScript allocates a memory spot for it.

A variable firstName is created and is given a copy of name's value. firstName has its own memory spot and is independent of name . At this moment in the code, firstName also has a value of "Carlos".

We then change the value of name to "Carla". But firstName still holds its original value, because it lives in a different memory spot.

When working with primitives, the =operator creates a copy of the original variable. That's what by value means.

With Objects, =operator works by reference.

Passing References in a Function

```
var myName = {};

function myNameIs(aName){

  aName.firstName = "Carla";

}

myNameIs(myName);

console.log(myName); // Object
{firstName: "Carla"}
```

A variable myName is created and is given the value of an object which has a property called firstName. firstName has the value of "Carlos". JavaScript allocates a memory spot for myName and the object it contains.

A variable identity is created and is pointed to myName. There is no dedicated memory space to identity's value'. It only points to myName's value.

We change the value of myName's firstName property to "Carla" instead of "Carlos".

When we log myName.firstName it displays the new value, which is pretty straightforward. But when we log identity.firstName its also displays myName.firstName's new value "Carla". This happens because identity.firstName only points to myName.firstName’s place in the memory.

When working with objects, the =operator creates an alias to the original object, it doesn’t create a new object. That’s what “by reference” means.

What Is Going On?!

```
var myName = {
  firstName: "Carla"
};

function
myNameIs(aName){
  aName = {
    nickName: "Carlita"
  };
}
```

```
myNameIs(myName);
console.log(myName); // Object {firstName: "Carla"}

//now try this
var myName = {
  firstName: "Carla"
};

function myNameIs(aName){
  aName.nickName = "Carlita";
}

myNameIs(myName);
console.log(myName); // Object {firstName: "Carla", nickName:
"Carlita"}
```

Loops

FOR LOOP

Initialize - Will run one time at the start of loop

condition - Will loop as long as this is true

increment - Will do this end of at every cycle

```
for (var i=0; i<10; i++) {
  console.log("RUN - " + i);
}
```

statements - Will do these every cycle

FOR-IN LOOP

```
var person = { "name" : "John", "age" : 123 };
for (let key in person) { console.log(key + " "); }
// Will output "name age "
```

FOR-OF LOOP

```
var animal = ["dog", "cat"];
for (let val of animal) { console.log(val + " "); }
// Will output "dog cat "
```

Loops

WHILE LOOP

```
var i = 0;
while (i<4) {
  console.log(i + " ");
  i++;
}
// Will output 0 1 2 3
```

Condition - Will loop as long as this is true

DO-WHILE LOOP

```
var i = 10;
do {
  console.log(i + " ");
  i++;
} while (i<4);
// Will output 10! DO-WHILE will run at least once
// even when condition is not met.
```

Condition - Will loop as long as this is true

BREAK & CONTINUE

```
var i = 0;
var text = "";
while (true) {
  i++;
  if (i==6) { break; }
  if (i==3) { continue; }
  text += i + " ";
}
console.log(text);
// Output 1 2 4 5
```

Break - Stop the loop

Continue - Skip cycle

Global Scope vs Local Scope

```
var x = 10;

if (x == 10) {
    const maxItems =
5;

console.log("Hello");
}
```

```
if (x == 11) {
      const maxItems = 5;
      console.log("Hello");
}

console.log(maxItems);
```

```
if (x == 10) {
     var maxItems = 5;
     console.log("Hello");
}

console.log(maxItems);
```

Functions Are Objects, Too

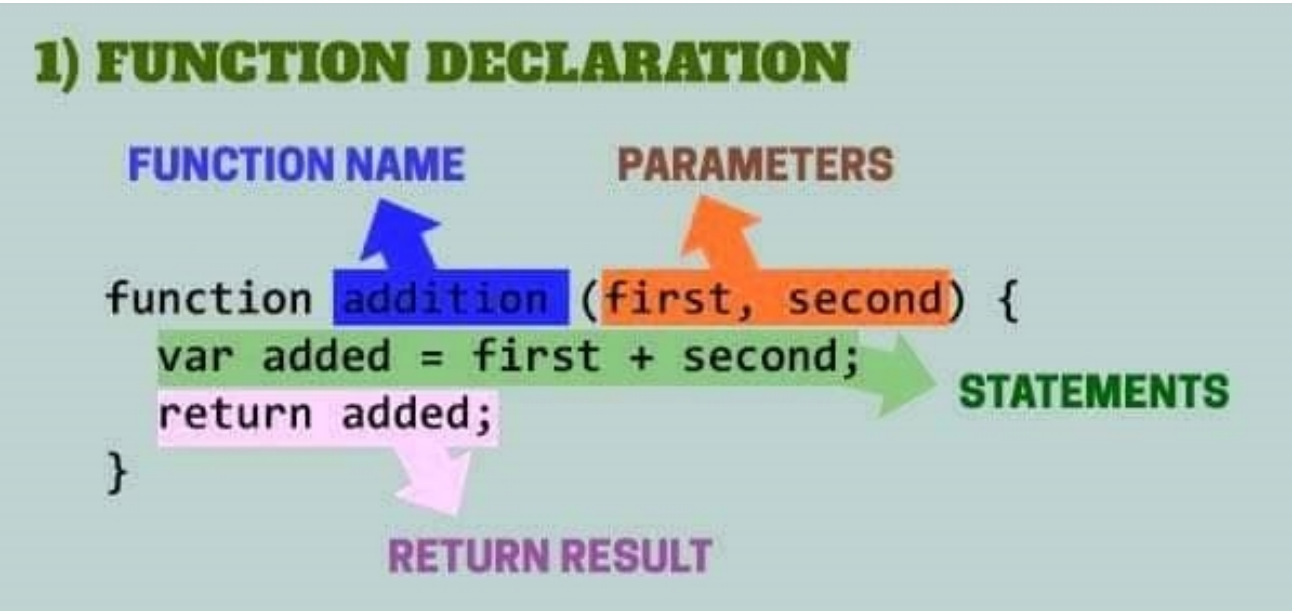
```
//function declaration
function add(num1, num2) {
      return num1 + num2;
}

//function expression
var add = function(num1, num2)
{
      return num1 + num2;
};
```

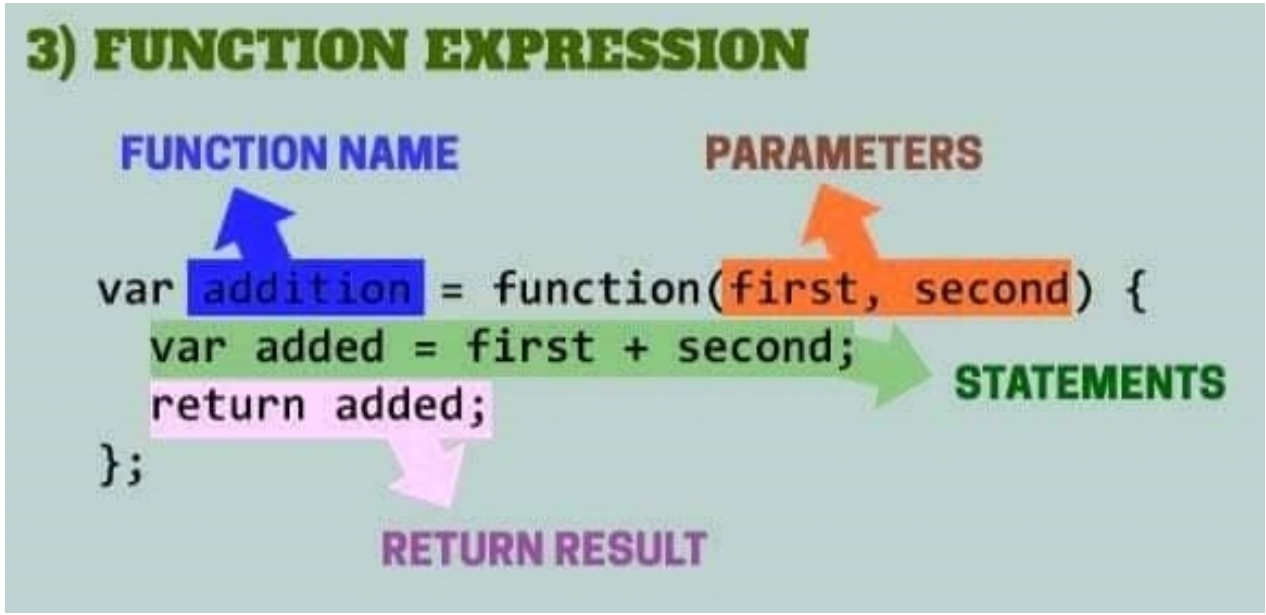
Function declarations are “hoisted” above all else whether within another function or the global scope. This is why function declarations will not cause an error when called from a previous line just like the code below.

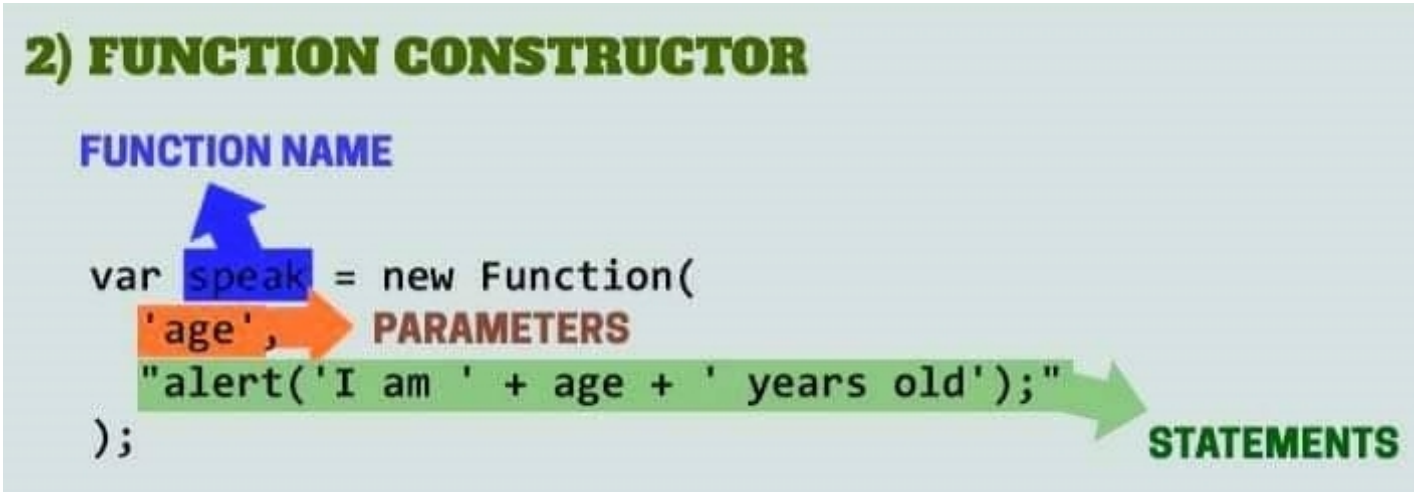
```
var result = add(5, 5);

function add(num1, num2) {
      return num1 + num2;
}
```

Function expressions, on the other hand, are not hoisted and will cause an error if the function is called in a previous line prior to the creation of the function.

Three Ways to Create Functions







Working with Arrays

A 2 WAYS TO DEFINE AN ARRAY

```
// Using the array constructor
var animals = new Array("Dog", "Cat", "Cow");

// Using the short hand
var animals = ["Dog", "Cat", "Cow"];
```

B HOW ARRAYS WORK

an array is like a container with many slots

KEY	0	1	2
VALUE	Dog	Cat	Cow

C ACCESSING ARRAY VALUES

```
// Retrieving values
alert(animals[1]); // Cat

// Replacing values
animals[2] = "Alpaca";
```

D GET ARRAY LENGTH

```
alert(animals.length); // 3
```

The Power of Javascript