**# Call stack**

A call stack is a mechanism for an interpreter (like the JavaScript interpreter in a web browser) to keep track of its place in a script that calls multiple functions — what function is currently being run and what functions are called from within that function, etc.

1. When a script calls a function, the interpreter adds it to the call stack and then starts carrying out the function.

2. Any functions that are called by that function are added to the call stack further up, and run where their calls are reached.

3. When the current function is finished, the interpreter takes it off the stack and resumes execution where it left off in the last code listing.

4. If the stack takes up more space than it was assigned, a "stack overflow" error is thrown.

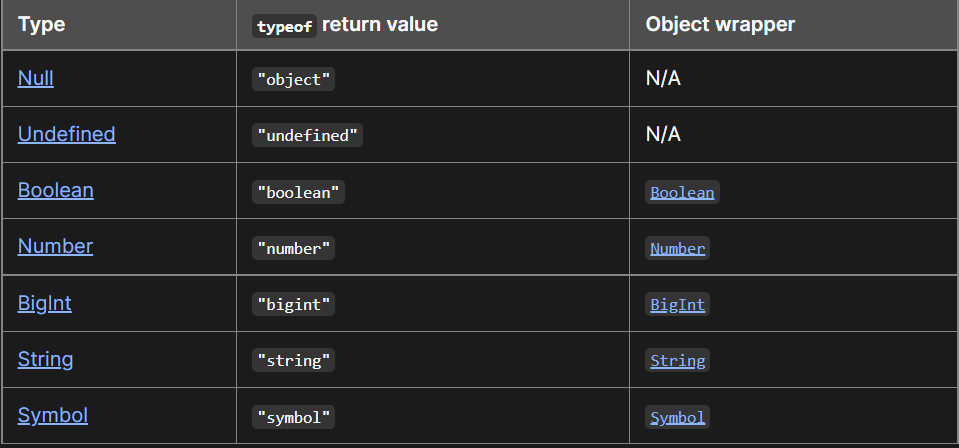
**# Primitive values**

All types except Object define immutable values represented directly at the lowest level of the language. We refer to values of these types as primitive values.

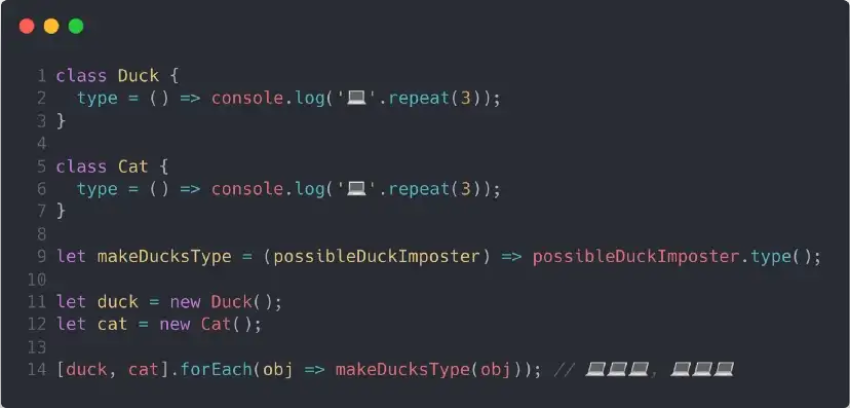
All primitive types, except null, can be tested by the typeof operator. typeof null returns "object", so one has to use === null to test for null.

All primitive types, except null and undefined, have their corresponding object wrapper types,

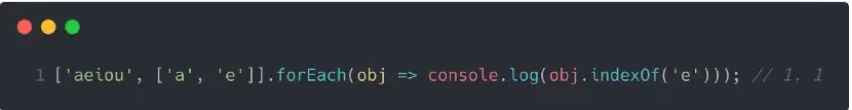
which provide useful methods for working with the primitive values.



**# Duck Typing**



duck and cat are both objects. Both of them respond to the method type . Therefore as far as JavaScript is concerned both objects are of the same type. You can call this informal grouping of unrelated objects x, group, or even duck.



We are calling forEach on an array that contains elements of two different types, a string and an array. Both elements get passed into the callback in turn and are in turn stored in or referenced by the parameter obj. Then the method indexOf checks if 'e' is found and returns the index. JavaScript doesn’t care that our elements have different types. It just cares about finding the index of 'e'.

**#** **Expression** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions\_and\_Operators#Expressions)

An expression is a valid unit of code that resolves to a value. There are two types of expressions: those that have side effects (such as assigning values) and those that purely evaluate.

x = 7 is an example of the first type

The expression 3 + 4 is an example of the second type.

**STATEMENT**

Basically, statements perform actions, they do things.

In javascript, statements can never be used where a value is expected. So they cannot be used as function arguments, right-hand side of assignments, operators operand, return values…

These are all javascript statements:

1. if
2. if-else
3. while
4. do-while
5. for
6. switch
7. for-in
8. with (deprecated)
9. debugger
10. variable declaration

Wherever JavaScript expects a statement, you can also write an expression. Such a statement is called an expression statement. The reverse does not hold: you cannot write a statement where JavaScript expects an expression. For example, an if statement cannot become the argument of a function.”

**# Nominal**

A nominal type system means that **each type is unique and even if types have the same data you cannot assign across types**.

**#** **Classes ans Factories**

ECMAScript 2015 (aka ES6) comes with the class syntax, so now we have two competing patterns for creating objects. In order to compare them, I’ll create the same object definition (TodoModel) as a class,and then as a factory function.

**factories are more robust against bugs due to avoiding the this keyword**.

Factory Method Pattern **allows the sub-classes to choose the type of objects to create**.

