avionschool

Lesson 7.0 JavaScript Types

BATCH 4

DECEMBER 05, 2020

Types

- Coercion confusion is perhaps one of the most profound frustrations for JavaScript developers. It has often been criticized as being so dangerous as to be considered a flaw in the design of the language, to be shunned and avoided.

- But as get to have a full understanding of JavaScript types, we get to flip the perspective of why coercion is bad, and see its power and usefulness.

Primitives

```
var a;
typeof a;               // "undefined"

a = "hello world";
typeof a;               // "string"

a = 42;
typeof a;               // "number"

a = true;
typeof a;               // "boolean"

a = null;
typeof a;               // "object" -- weird, bug

a = undefined;
typeof a;               // "undefined"

a = { b: "c" };
typeof a;               // "object"

// added in ES6!
typeof Symbol()         // "symbol"
```

Primitives

```
var a = null;

(!a && typeof a === "object"); // true
```

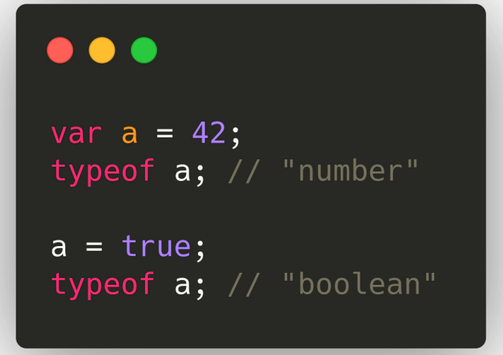
Primitives

Primitives

```
typeof function a(){ /* .. */ } === "function"; // ?

function a(b,c) {
    /* .. */
}

a.length; // ?
```

Values as Types

- In JavaScript, variables don't have types -- values have types. Variables can hold any value, at any time. "what's the type of the value in the variable?"



typeof

- operator inspects the type of the given value, and always returns one of seven string values

BUILT-IN

`null`

`undefined`

`boolean`

`number`

`string`

`object`

`symbol` -- added in ES6!

***Note**: All of these types except object are called "primitives".*

typeof

`null`

- It's special -- special in the sense that it's buggy when combined with the typeof operator:

```
typeof null === "object"; // true
```

- null is the only primitive value that is "falsy" but that also returns "object" from the typeof check.

Primitives

`undefined`

- Variables that have no value currently, actually have the undefined.

undefined vs undeclared

- An "undefined" variable is one that has been declared in the accessible scope, but at the moment has no other value in it. By contrast, an "undeclared" variable is one that has not been formally declared in the accessible scope.

```
var a;

a; // undefined
b; // ReferenceError: b is not defined
```

```
var a;

typeof a; // "undefined"

var b = 42;
var c;

// later
b = c;

typeof b; // "undefined"
typeof c; // "undefined"
```

Natives

Here's a list of the most commonly used natives:

`String()` `Number()` `Boolean()` `Array()` `Object()` `Function()` `RegExp()` `Date()` `Error()` `Symbol()`

```
var s = new String( "Hello World!" );

console.log( s.toString() ); // "Hello World!"
```

It is true that each of these natives can be used as a native constructor. But what's being constructed may be different than you think.

```
var a = new String( "abc" );
typeof a; // "object" ... not "String"
a instanceof String; // true
Object.prototype.toString.call( a ); // "[object String]"
```

`new String("abc")` creates a string wrapper object around `"abc"`, not just the primitive `"abc"` value itself

Natives

`Boolean`

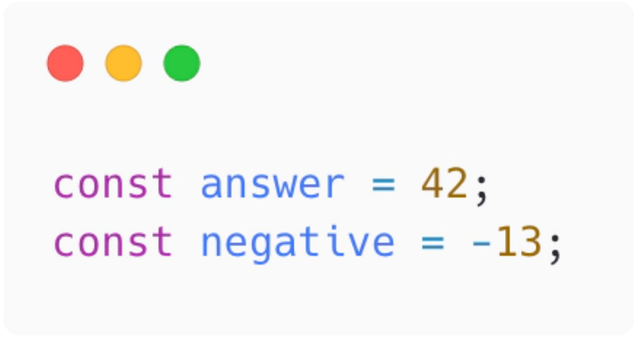
- Holds the value of either True or False / 1 or 0 / ON or OFF
- Low-level programming language

```
const iLoveJavascript = true;
```

Natives

`Number`

- Accepts integer, decimal, and real numbers
- Does not use quotation marks
- Limitation: +/- 9007199254740991



```
const answer = 42;
const negative = -13;
```

Natives

`String`

- Blocks of text
- Defined with quotation marks (' ') or (" ")

```
const dog = 'fido';
```

Natives

`Object`

- Objects group together a set of variables and functions to create a model
- In an object, variables and functions take on new names.

```
const newObj = {};
```

Key:value pairs

- Objects use a concept called "key:value pairs.
- The key is the identifier and the value is the value we want to save to that key.
- Objects can hold many key:value pairs, they must be separated by a comma (no semi-colons inside of an object!).

```
const school = {
    //PROPERTIES
    name: 'Avion School',
    courseSlots: 30,
    students: 25,
    courses: ['Web Dev', 'Data Science', 'UX/UI'],
    //METHOD
    checkAvailability: function(){
        return this.courseSlots - this.students;
    }
};
```

Natives

`Symbol`

- Symbol is a new data type as of ES6
- Symbols are special "unique" (not strictly guaranteed!) values that can be used as properties on objects with little fear of any collision.
- Symbols are not objects, they are simple scalar primitives.

```
var mysym = Symbol( "my own symbol" );
mysym;               // Symbol(my own symbol)
mysym.toString();    // "Symbol(my own symbol)"
typeof mysym;        // "symbol"
```

Boxing Wrappers

- Primitive values don't have properties or methods, so to access .length or .toString() you need an object wrapper around the value.
- Thankfully, JS will automatically box (aka wrap) the primitive value to fulfill such accesses.

```
var a = "abc";

a.length; // 3
a.toUpperCase(); // "ABC"
```

Natives as Constructors

Array(..)

```
var a = new Array( 1, 2, 3 );
a; // [1, 2, 3]

var b = [1, 2, 3];
b; // [1, 2, 3]
```

The `Array(..)` constructor does not require the `new` keyword in front of it. If you omit it, it will behave as if you have used it anyway.

So `Array(1,2,3)` is the same outcome as `new Array(1,2,3)`.

Natives as Constructors

Object(..), Function(..), and RegExp(..)

The `Object(..)` / `Function(..)` / `RegExp(..)` constructors are also generally optional (and thus should usually be avoided unless specifically called for):

```
var c = new Object();
c.foo = "bar";
c; // { foo: "bar" }

var d = { foo: "bar" };
d; // { foo: "bar" }

var e = new Function( "a", "return a * 2;" );
var f = function(a) { return a * 2; };
function g(a) { return a * 2; }

var h = new RegExp( "^a*b+", "g" );
var i = /^a*b+/g;
```

There's practically no reason to ever use the `new Object()` constructor form, especially since it forces you to add properties one-by-one instead of many at once in the object literal form.

The `Function` constructor is helpful only in the rarest of cases, where you need to dynamically define a function's parameters and/or its function body. **Do not just treat `Function(..)` as an alternate form of `eval(..)`**. You will almost never need to dynamically define a function in this way.

Natives as Constructors

Object(..), Function(..), and RegExp(..)

Regular expressions defined in the literal form `(/^a*b+/g)` are strongly preferred, not just for ease of syntax but for performance reasons -- the JS engine precompiles and caches them before code execution. Unlike the other constructor forms we've seen so far, `RegExp(..)` has some reasonable utility: to dynamically define the pattern for a regular expression.

```
var name = "Kyle";
var namePattern = new RegExp( "\\b(?:" + name + ")+\\b", "ig" );

var matches = someText.match( namePattern );
```

This kind of scenario legitimately occurs in JS programs from time to time, so you'd need to use the `new RegExp("pattern","flags")` form.

Natives as Constructors

Symbol(..)

Symbols can be used as property names, but you cannot see or access the actual value of a symbol from your program, nor from the developer console. If you evaluate a symbol in the developer console, what's shown looks like `Symbol(Symbol.create)`, for example.

The `Symbol(..)` native "constructor" is unique in that you're not allowed to use `new` with it, as doing so will throw an error.

```
var mysym = Symbol( "my own symbol" );
mysym;               // Symbol(my own symbol)
mysym.toString();    // "Symbol(my own symbol)"
typeof mysym;        // "symbol"

var a = { };
a[mysym] = "foobar";

Object.getOwnPropertySymbols( a );
// [ Symbol(my own symbol) ]
```

Coercion

- Converting a value from one type to another is often called "type casting," when done explicitly, and "coercion" when done implicitly (forced by the rules of how a value is used).

```
var a = 42;

var b = a + "";          // implicit coercion

var c = String( a );     // explicit coercion
```

Abstract Value Operations

ToString

- When any non-string value is coerced to a string representation

```
// multiplying `1.07` by `1000`, seven times over
var a = 1.07 * 1000 * 1000 * 1000 * 1000 * 1000 *
1000 * 1000;

// seven times three digits => 21 digits
a.toString(); // "1.07e21"
```

Abstract Value Operations

ToNumber

- If any non-number value is used in a way that requires it to be a number, such as a mathematical operation

```
var a = {
    valueOf: function(){
        return "42";
    }
};

var b = {
    toString: function(){
        return "42";
    }
};

var c = [4,2];
c.toString = function(){
    return this.join( "" ); // "42"
};

Number( a );            // 42
Number( b );            // 42
Number( c );            // 42
Number( "" );           // 0
Number( [] );           // 0
Number( [ "abc" ] );    // NaN
```

Abstract Value Operations

ToBoolean

First and foremost, JS has actual keywords `true` and `false`, and they behave exactly as you'd expect of `boolean` values. It's a common misconception that the values `1` and `0` are identical to `true`/`false`.

While that may be true in other languages, in JS the `numbers` are `numbers` and the `booleans` are `booleans`. You can coerce `1` to `true` (and vice versa) or `0` to `false` (and vice versa). But they're not the same.

FALSY VALUES

All of JavaScript's values can be divided into two categories:

- values that will become `false` if coerced to `boolean`
- everything else (which will obviously become `true`)

So-called "falsy" values list:

`undefined`

`null`

`false`

`+0`, `0`, and `NaN`

`""`

Anything not explicitly on the falsy list is therefore truthy.

FALSY OBJECTS

A "falsy object" is a value that looks and acts like a normal object (properties, etc.), but when you coerce it to a `boolean`, it coerces to a `false` value.

```
var a = new Boolean( false );
var b = new Number( 0 );
var c = new String( "" );

var d = Boolean( a && b && c );

d; // truex
```

TRUTHY VALUES

a value is truthy if it's not on the falsy list.

```
var a = "false";
var b = "0";
var c = "' '";

var d = Boolean( a && b && c );

d;
```

```
var a = [];               // empty array -- truthy or falsy?
var b = {};               // empty object -- truthy or falsy?
var c = function(){};     // empty function -- truthy or falsy?

var d = Boolean( a && b && c );

d;
```

Explicit Coercion

- Explicit coercion refers to type conversions that are obvious and explicit. There's a wide range of type conversion usage that clearly falls under the explicit coercion category for most developers.

EXPLICITLY: STRINGS <--> NUMBERS

```
var a = 42;
var b = String( a );

var c = "3.14";
var d = Number( c );

b; // "42"
d; // 3.14
```

```
var a = 42;
var b = a.toString();

var c = "3.14";
var d = +c;

b; // "42"
d; // 3.14
```

`Date` To `number`

```
var d = new Date( "Mon, 18 Aug 2014 08:53:06 CDT" );

+d; // 1408369986000
```

But coercion is not the only way to get the timestamp out of a `Date` object. A noncoercion approach is perhaps even preferable, as it's even more explicit:

```
var timestamp = new Date().getTime();
// var timestamp = (new Date()).getTime();
// var timestamp = (new Date).getTime();
```

EXPLICITLY: PARSING NUMERIC STRINGS

```
var a = "42";
var b = "42px";

Number( a );     // 42
parseInt( a );   // 42

Number( b );     // NaN
parseInt( b );   // 42
```

Tip: parseInt(..) has a twin, parseFloat(..), which (as it sounds) pulls out a floating-point number from a string.

Parsing Non-Strings

`parseInt(..)` forcibly coerces its value to a `string` to perform the parse on is quite sensible. If you pass in garbage, and you get garbage back out, don't blame the trash can -- it just did its job faithfully.

```
parseInt( 0.000008 );       // 0   ("0" from "0.000008")
parseInt( 0.0000008 );      // 8   ("8" from "8e-7")
parseInt( false, 16 );      // 250 ("fa" from "false")
parseInt( parseInt, 16 );   // 15  ("f" from "function..")

parseInt( "0x10" );         // 16
parseInt( "103", 2 );       // 2
```

`parseInt(..)` is actually pretty predictable and consistent in its behavior. If you use it correctly, you'll get sensible results. If you use it incorrectly, the crazy results you get are not the fault of JavaScript.

EXPLICITLY: * --> BOOLEAN

Just like with `String(..)` and `Number(..)` above, `Boolean(..)` (without the new `,` of course!) is an explicit way of forcing the `ToBoolean` coercion:

```
var a = "0";
var b = [];
var c = {};

var d = "";
var e = 0;
var f = null;
var g;

Boolean( a ); // true
Boolean( b ); // true
Boolean( c ); // true

Boolean( d ); // false
Boolean( e ); // false
Boolean( f ); // false
Boolean( g ); // false
```

Implicit Coercion

- Implicit coercion refers to type conversions that are hidden, with non-obvious side-effects that implicitly occur from other actions. In other words, implicit coercions are any type conversions that aren't obvious (to you).

SIMPLIFYING IMPLICITLY

```
SomeType x = SomeType( AnotherType( y ) )
```

IMPLICITLY: STRINGS <--> NUMBERS

```
var a = "42";
var b = "0";

var c = 42;
var d = 0;

a + b; // "420"
c + d; // 42
```

IMPLICITLY: BOOLEANS --> NUMBERS

```
function onlyOne() {
    var sum = 0;
    for (var i=0; i < arguments.length; i++) {
        // skip falsy values. same as treating
        // them as 0's, but avoids NaN's.
        if (arguments[i]) {
            sum += arguments[i];
        }
    }
    return sum == 1;
}

var a = true;
var b = false;

onlyOne( b, a );        // true
onlyOne( b, a, b, b, b );   // true

onlyOne( b, b );        // false
onlyOne( b, a, b, b, b, a );    // false
```

IMPLICITLY: * --> BOOLEAN

Remember, implicit coercion is what kicks in when you use a value in such a way that it forces the value to be converted. For numeric and `string` operations, it's fairly easy to see how the coercions can occur.

```
var a = 42;
var b = "abc";
var c;
var d = null;

if (a) {
    console.log( "yep" );       // yep
}

while (c) {
    console.log( "nope, never runs" );
}

c = d ? a : b;
c;                  // "abc"

if ((a && d) || c) {
    console.log( "yep" );       // yep
}
```

Symbol Coercion

```
var s1 = Symbol( "cool" );
String( s1 );                   // "Symbol(cool)"

var s2 = Symbol( "not cool" );
s2 + "";                        // TypeError
```

`symbol` values cannot coerce to `number` at all (throws an error either way), but strangely they can both explicitly and implicitly coerce to `boolean` (always `true`).

Coding Challenge

CHALLENGE

Victor and John are trying to compare this BMI (body mass index), which is calculated using the formula:

BMI = mass/height^2 = mass / (height * height). (mass in kg and height in meter).

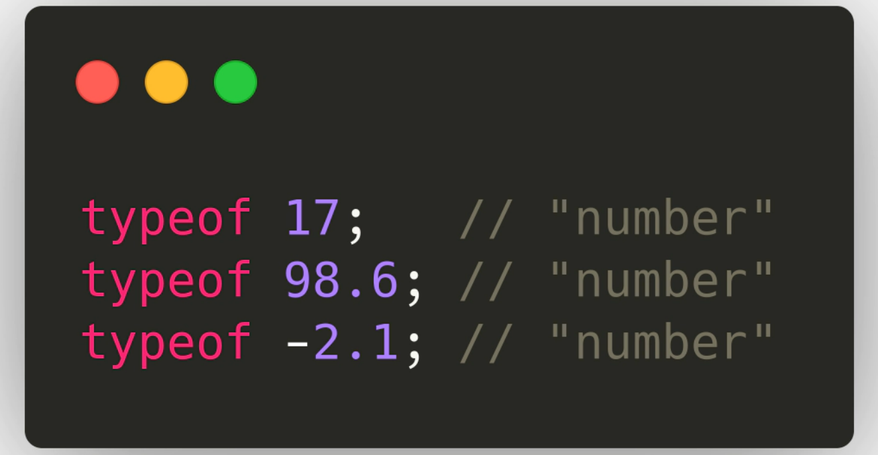
STEPS

1. Store Victor and John's mass and height in variables
2. Calculate both their BMIs
3. Create a boolean variable containing information about whether Victor has a higher BMI than John.
4. Print a string to the console containing variable from step 3. (something like "Is Victor's BMI higher than John's? True".

Effective JS #1: Understand Floating-Point Numbers

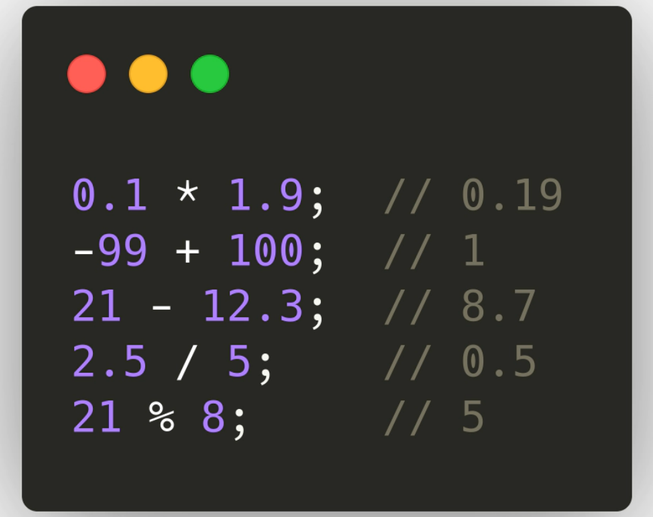
- Most programming languages have several types of numeric data, but JavaScript gets away with just one: *number*
- But in fact, all numbers in JavaScript are *double-precision floating-point* numbers- commonly known as *"doubles".*
- –9,007,199,254,740,992 (–2 ^53) to 9,007,199,254,740,992(2^53)
- 2^53 = "10000000000000000000000000000000000000000000000000000"- perfectly possible to do integer arithmetic!

Effective JS #1: Understand Floating-Point Numbers



Effective JS #1: Understand Floating-Point Numbers

- Most arithmetic operators work with integers, real numbers, or a combination of the two
- In JS, they operate on their arguments as floating-point numbers



```
0.1 * 1.9;  // 0.19
-99 + 100;  // 1
21 - 12.3;  // 8.7
2.5 / 5;    // 0.5
21 % 8;     // 5
```

Effective JS #1: Understand Floating-Point Numbers

- But bitwise arithmetic operators implicitly convert them to 32-bit integers
- All of the bitwise operators work the same way, converting their inputs to integers and performing their operations on the integer bit patterns before converting the results back to standard JavaScript floating-point numbers.

Effective JS #1: Understand Floating-Point Numbers

- Decimals to the right of the decimal point is stored cleanly and nicely: ones, tenth's, hundredth's thousandth's place, etc.
- In binary however, each place to the right of the decimal is worth half as the place to the left of it: halves, quarters, eights, sixteenths, thirty-seconds, sixty-fourths, etc.
- The binary floating point format is going to have to store the value as a quarter, plus a thirty-second, and a sixty-fourth, and so on, getting closer and closer to 0.3 - but never actually exactly reaching it.
- So it stores the closest approximation of 0.3 that it can in the number of bits the variable has available, and that approximation is what you're seeing when inspecting the value in your program.

Effective JS #1: Understand Floating-Point Numbers

- Decimals to the right of the decimal point is stored cleanly and nicely: ones, tenth's, hundredth's thousandth's place, etc.
- In binary however, each place to the right of the decimal is worth half as the place to the left of it: halves, quarters, eights, sixteenths, thirty-seconds, sixty-fourths, etc.
- The binary floating point format is going to have to store the value as a quarter, plus a thirty-second, and a sixty-fourth, and so on, getting closer and closer to 0.3 - but never actually exactly reaching it.
- So it stores the closest approximation of 0.3 that it can in the number of bits the variable has available, and that approximation is what you're seeing when inspecting the value in your program.

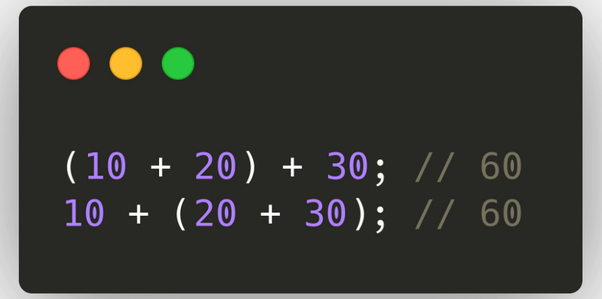
Effective JS #1: Understand Floating-Point Numbers

- When you perform a sequence of calculations, these rounding errors can accumulate, leading to less and less accurate results.
- But with floating point numbers, this is not always the case!

```
(0.1 + 0.2) + 0.3; // 0.6000000000000001
0.1 + (0.2 + 0.3); // 0.6
```

Effective JS #1: Understand Floating-Point Numbers

- When accuracy matters, it’s critical to be aware of their limitations. One useful workaround is to work with integer values wherever possible, since they can be represented without rounding. When doing calculations with money, programmers often scale numbers up to work with the currency’s smallest denomination so that they can compute with **whole numbers.**

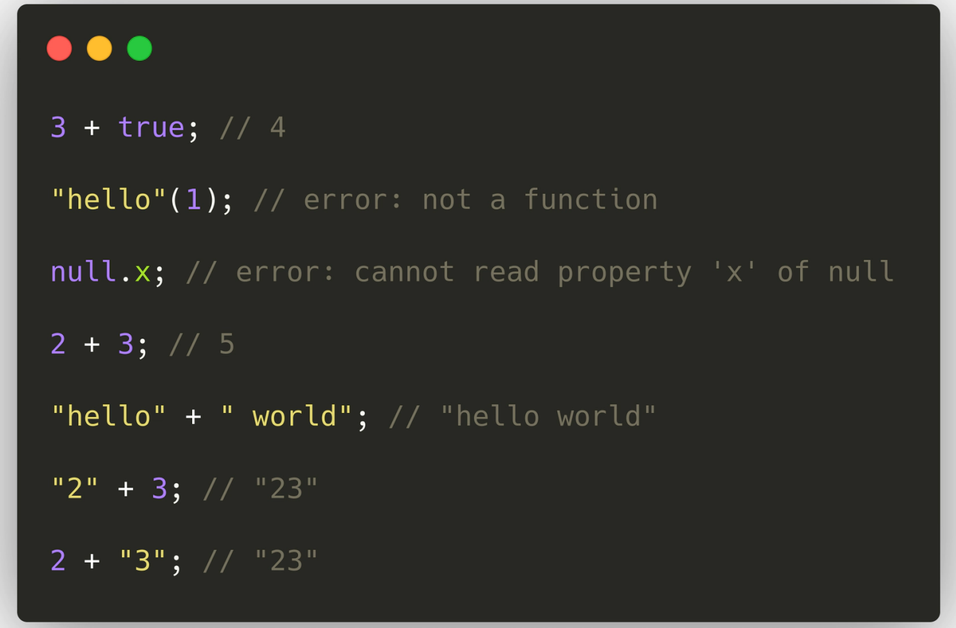


Effective JS #1: Understand Floating-Point Numbers

Review:

- JavaScript numbers are double-precision floating-point numbers.
- Integers in JavaScript are just a subset of doubles rather than a separate datatype.
- Bitwise operators treat numbers as if they were 32-bit signed integers.
- Be aware of limitations of precisions in floating-point arithmetic.

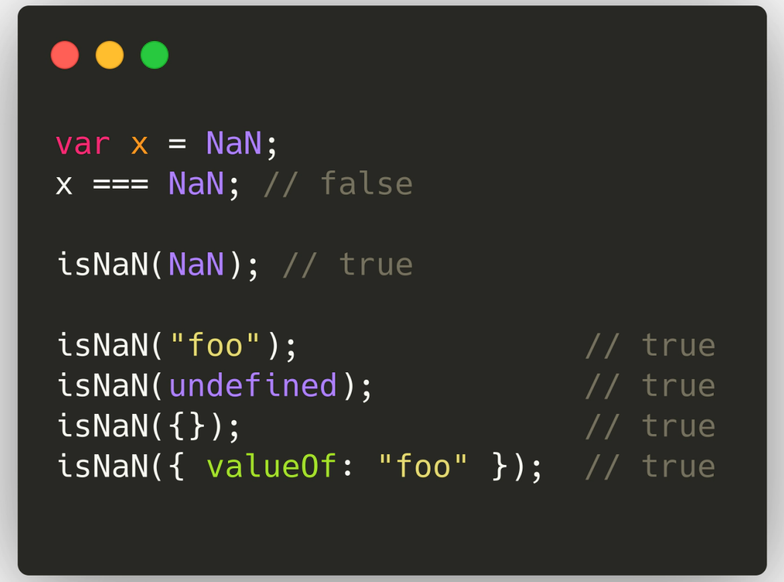
Effective JS #2: Beware of Implicit Coercions



```
3 + true; // 4

"hello"(1); // error: not a function

null.x; // error: cannot read property 'x' of null

2 + 3; // 5

"hello" + " world"; // "hello world"

"2" + 3; // "23"

2 + "3"; // "23"
```

Effective JS #2: Beware of Implicit Coercions

Effective JS #2: Beware of Implicit Coercions

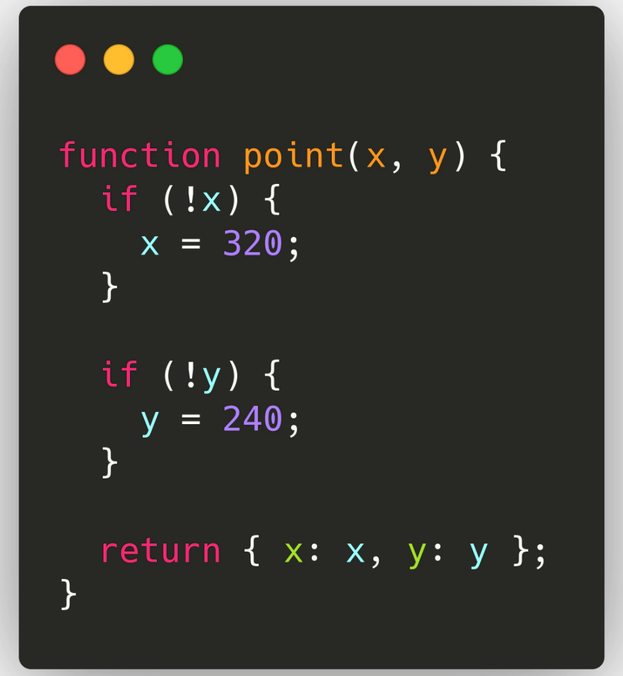


```
var x = NaN;
x === NaN; // false

isNaN(NaN); // true

isNaN("foo");                // true
isNaN(undefined);            // true
isNaN({});                   // true
isNaN({ valueOf: "foo" });   // true
```

Effective JS #2: Beware of Implicit Coercions

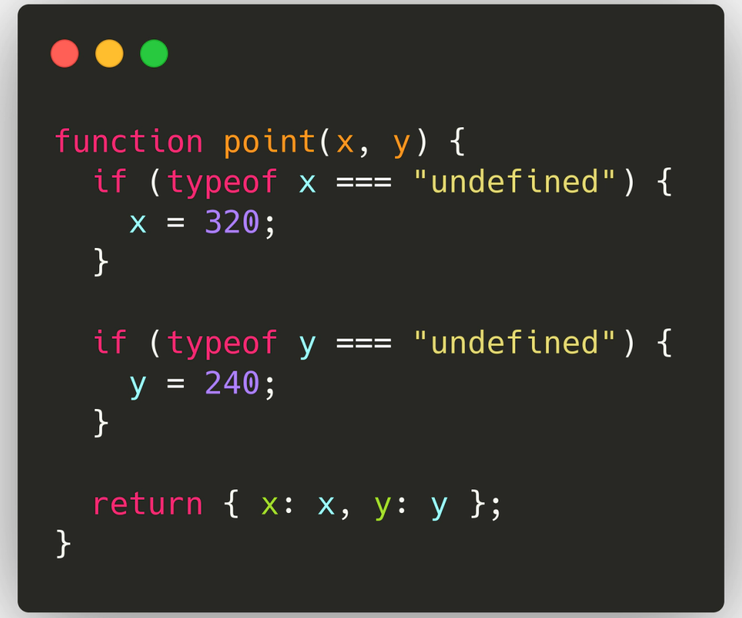
```
var a = NaN;
a !== a;                        // true

var b = "foos";
b !== b;                        // false

var c = undefined;
c !== c;                        // false

var d = {};
d !== d;                        // false

var e = {valueOf: "foo"};
e !== e;                        //false


function isReallyNaN(x) {
    return x !== x;
}
```

Effective JS #2: Beware of Implicit Coercions



```
function point(x, y) {
  if (!x) {
    x = 320;
  }

  if (!y) {
    y = 240;
  }

  return { x: x, y: y };
}
```

Effective JS #2: Beware of Implicit Coercions



```
function point(x, y) {
  if (typeof x === "undefined") {
    x = 320;
  }

  if (typeof y === "undefined") {
    y = 240;
  }

  return { x: x, y: y };
}
```

Effective JS #2: Beware of Implicit Coercions

- Type errors can be silently hidden by implicit coercions.
- The + operator is overloaded to do addition or string concatenation depending on its argument types.
- Use typeof or comparison to undefined rather than truthiness to test for undefined values.

Effective JS #3: Prefer Primitives to Object Wrappers

```
var s = new String("hello");

s + " world"; // "hello world"

s[4]; // "o"

typeof "hello"; // "string"
typeof s; // "object"

var s1 = new String("hello");
var s2 = new String("hello");
s1 === s2; // false

s1 == s2; // false

"hello".toUpperCase(); // "HELLO"

"hello".someProperty = 17;
"hello".someProperty; // undefined
```

Effective JS #3: Prefer Primitives to Object Wrappers

- Object wrappers for primitive types do not have the same behavior as their primitive values when compared for equality.
- Getting and setting properties on primitives implicitly creates object wrappers.

Effective JS #4: Avoid using == with Mixed Types

- The == operator applies a confusing set of implicit coercions when its arguments are of different types.
- Use === to make it clear to your readers that your comparison does not involve any implicit coercions.
- Use your own explicit coercions when comparing values of different types to make your program's behavior clearer.