Primitive Datatypes in JS

console.log(typeof(undefined)); // undefined

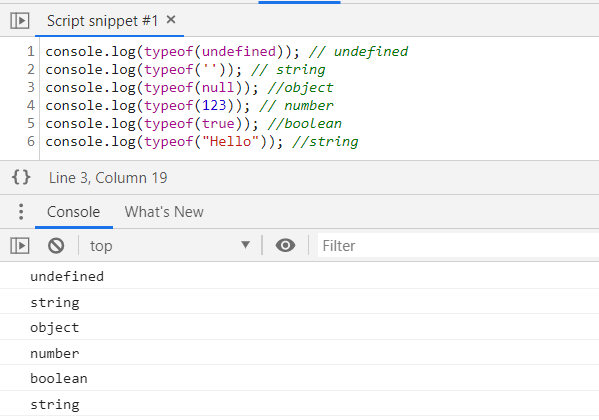
console.log(typeof('')); // string

console.log(typeof(null)); //object

console.log(typeof(123)); // number

console.log(typeof(true)); //boolean

console.log(typeof("Hello")); //string



**JavaScript values: not everything is an object**

**Update 2012-08-14:** Rewrote most of the post, added Sect. 4 explaining typeof and instanceof.

This blog post explains that JavaScript has two main kinds of values: primitive values and objects. There are several things one needs to be aware of when working with them.

**Kinds of values: primitives versus objects**

Values can be partitioned into two main kinds: primitives and objects.

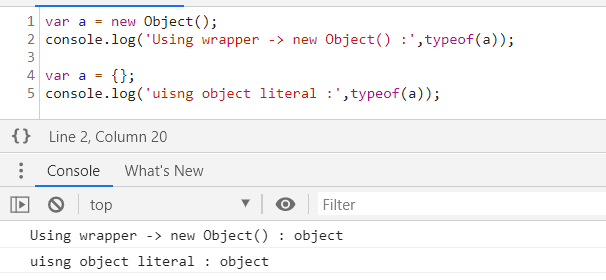
**The definition**

The usual definition for primitives and objects in JavaScript is: **The following values are primitive.**

* Strings: "abc"
* Numbers: 4, 3.57 (all numbers in JavaScript are floating point)
* Booleans: true, false
* null: usually explicitly assigned
* undefined: usually a default value, automatically assigned

**All other values are objects.** Objects can be partitioned further:

* Wrappers for primitives: Boolean, Number, String. Rarely used directly.
* Creatable by literals. The following literals produce objects that can also be created via a constructor. Use literals whenever you can.
  + [] is the same as new Array()
  + {} is the same as new Object()
  + function() {} is the same as new Function()
  + /\s\*/ is the same as new RegExp("\\s\*")
* Dates: new Date("2011-12-24")



**Different natures**

You can define primitives and objects by enumerating the primitives and defining objects as non-primitives. But you can also describe what primitives and objects are like. Let’s start with objects.

* **Objects are mutable by default:**
* > var obj = {};
* > obj.foo = 123; // write
* 123
* > obj.foo // read
* 123
* **Objects have unique identities and are compared *by reference*:** Every object you create via an expression such as a constructor or a literal is considered different from every other object; a fact that can be observed via the equality operator (===). That operator compares objects *by reference*: two objects are only equal if they have the same identity. It does not matter whether they have the same content or not.
* > {} === {}
* false
* > var obj = {};
* > obj === obj
* true
* **Variables hold references to objects:** Thus, two variables can refer to the same object – changes you make via one variable can be observed via the other variable.
* > var var1 = {};
* > var var2 = var1;
* > var1.foo = 123;
* 123
* > var2.foo
* 123

As expected, primitives are different:

* **Primitives are immutable:** any property you add will be immediately forgotten.
* > var str = "abc";
* > str.foo = 123; // write - ignored
* 123
* > str.foo // read
* undefined
* **Primitives are compared *by value*, they don’t have individual identities:** To compare two primitives, one looks at their values, their content. If their values are the same then they are considered equal.
* > "abc" === "abc"
* true

That means that the identity of a primitive is its value, it does not have an individual identity.

These last two facts combined mean that there is no way for you to tell whether a variable holds a reference to a primitive or a complete copy. Internally, the former is usually true for strings, the latter for numbers.

**Pitfall: Primitive values and their wrappers**

**Rule:**

Ignore wrapper types as much as possible. In contrast to other programming languages such as Java, you will rarely notice them.

The three primitive types string, number and boolean have corresponding types whose instances are objects: String, Number, Boolean. They are sometimes called *wrapper types* and converting between primitive and wrapper is simple:

* Primitive to wrapper: new String("abc")
* Wrapper to primitive: new String("abc").valueOf()

Primitive values such as "abc" are fundamentally different from wrapper instances such as new String("abc"). For example (typeof and instanceof are explained below):

> typeof "abc"

'string'

> typeof new String("abc")

'object'

> "abc" instanceof String

false

> new String("abc") instanceof String

true

> "abc" === new String("abc")

false

Wrapper instances are objects and there is no way of comparing objects in JavaScript, not even via non-strict equals == (which is more lenient than the preferred strict equals ===).

> var a = new String("abc");

> var b = new String("abc");

> a == b

false

> a == a

true

**Primitive values don’t have methods of their own**

Wrappers are rarely needed in JavaScript, as primitives can be stored anywhere without wrapping them. But primitives don’t have their own methods and borrow them from wrappers:

> "abc".charAt === String.prototype.charAt

true

There are two ways that this borrowing is done. The old way is to convert a primitive to a wrapper, on the fly. The new way (via ECMAScript 5 [strict mode](http://2ality.com/2011/01/javascripts-strict-mode-summary.html)) is to transparently use the methods from the wrapper’s prototype. The following code illustrates the difference [inspired by “[The Secret Life of JavaScript Primitives](http://javascriptweblog.wordpress.com/2010/09/27/the-secret-life-of-javascript-primitives/)”].

// Methods in Object.prototype are available to all primitives

Object.prototype.getType = function() {

return typeof this;

};

Object.prototype.getTypeStrict = function() {

"use strict";

return typeof this;

};

console.log("".getType()); // object

console.log("".getTypeStrict()); // string

**Categorizing values: typeof and instanceof**

If you want to categorize a value, you unfortunately have to be aware of the difference between primitives and objects. The typeof operator categorizes primitives and distinguishes them from objects. The instanceof operator categorizes objects and returns false for any primitive.

**typeof**

The typeof operator is mainly used to distinguish primitives from each other and from objects:

> typeof "abc"

'string'

> typeof 123

'number'

> typeof {}

'object'

> typeof []

'object'

typeof returns the following strings:

|  |  |
| --- | --- |
| **Value** | **Result** |
| Undeclared variables | "undefined" |
| undefined | "undefined" |
| null | "object" |
| Booleans | "boolean" |
| Numbers | "number" |
| String | "string" |
| Functions | "function" |
| All other values | "object" |

Comments:

* Returning "object" for null is a well-known bug that can’t be fixed, because it would break existing code. That does not mean that null is actually an object [4].
* typeof allows you to check whether a variable has been declared, without throwing an exception. No function can possibly do that, because you can’t pass it such a variable.
* > typeof undeclaredVariable
* 'undefined'
* > undeclaredVariable
* ReferenceError: undeclaredVariable is not defined
* Functions are actually objects; giving them their own category is a bit inconsistent, but sometimes useful.
* Arrays are objects and correctly categorized as such.

More information on typeof: [5] and [6].

**instanceof**

The instanceof operator is used like this:

value instanceof Constructor

The above expression returns true if value is an instance of Constructor. It is equivalent to:

Constructor.prototype.isPrototypeOf(value)

Most objects are instances of Object, because their prototype chain ends with Object.prototype. Primitives are not an instance of anything.

> "abc" instanceof Object

false

> "abc" instanceof String

false

null & Undefined

This is a very useful technique for writing programs that are more-easily debugged. An 'undefined' variable might be the result of a bug ... (how would you know?) ... but if the variable contains the value 'null,' you know that "someone, somewhere in this program, set it to 'null.'

null is [not an object](http://www.2ality.com/2011/03/javascript-values-not-everything-is.html), it is a primitive value. For example, you cannot add properties to it. Sometimes people wrongly assume that it is an object, because typeof null returns "object". But that is actually a bug (that might even be fixed in ECMAScript 6).

**What is the difference between null and undefined??**

<https://stackoverflow.com/questions/801032/why-is-null-an-object-and-whats-the-difference-between-null-and-undefined>

A property when it has no definition, is undefined. null is an object. Its type is object. null is a special value meaning "no value. undefined is not an object, it's type is undefined.

null is [not an object](http://www.2ality.com/2011/03/javascript-values-not-everything-is.html), it is a primitive value. For example, you cannot add properties to it. Sometimes people wrongly assume that it is an object, because typeof null returns "object". But that is actually a bug (that might even be fixed in ECMAScript 6).