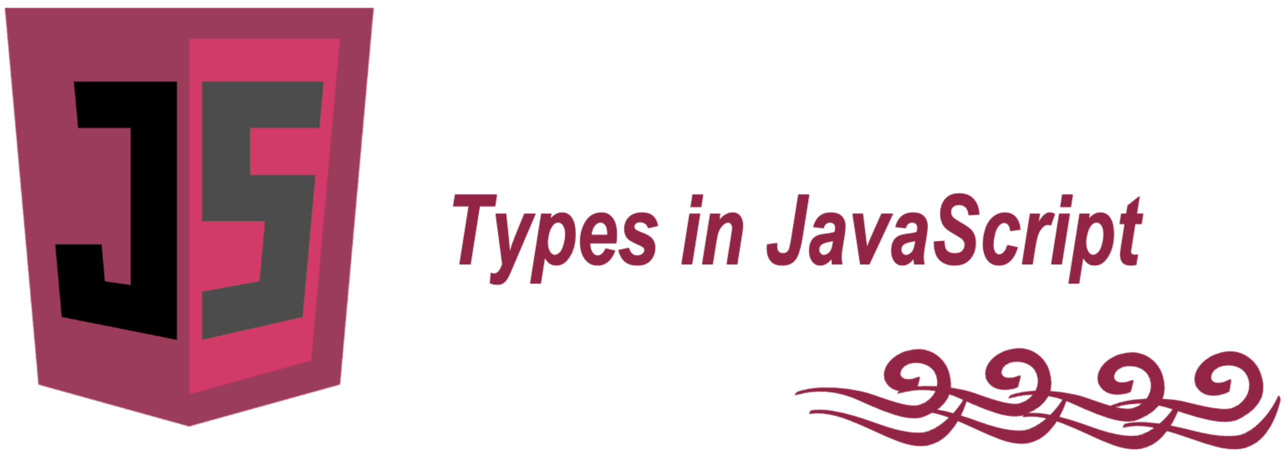
Types

In this lesson, we will explore JavaScript types.
Let's begin!



Values and variables always have a concrete type that you can access with the typeof operator, as shown is this sample:

```
var myValue = "this is a value";
console.log(typeof new Object());
console.log(typeof myValue);
console.log(typeof (42));
```

JavaScript has only **seven types** by means of the value domain of the `typeof` operator. They are:

> 1. object
> 2. function
> 3. string
> 4. number
> 5. boolean
> 6. null
> 7. undefined

⥵ The value of string, number, or boolean types are retrieved by `typeof`, respectively.

⥵ If the value is an object, or null (this value represents an empty object pointer), `typeof` returns object.

As you already learned, functions are first class citizens in JavaScript, so `typeof` retrieves function if you apply it on a function definition, as shown in this code snippet:

```
console.log(typeof fortytwo);

function fortytwo() {
    return 42;
}
```

The `undefined` type has only one special value, `undefined`. When a variable is declared but not initialized, or you refer to a non-existing property of an object value, you get back `undefined`. This code snippet shows a few samples that all produce this value:

```
var dummy;
var obj = new Object();
console.log(typeof undefined);


console.log(typeof dummy);
console.log(typeof obj.name);
```

Although `typeof` never returns `null` (for `null` it gives back `object`), the **ECMAScript** standard defines `null` as a type, which, similarly to `undefined`, has only one special value, `null`.

Primitive types as objects

Earlier you learned that you can create your own object instances with constructor functions that act as templates. JavaScript provides a few constructor functions out-of-the-box (defined by the standard). These are `Number()`, `String()`, `Boolean()`, `Object()`, `Array()`, `Function()`, `Date()`, `RegExp()`, and `Error()`.

The first three — `Number()`, `String()`, and `Boolean()`, not only construct objects, but they also provide a primitive value for a string, a number, and a Boolean.

Listing 7-9 provides a little help to understand this:

Listing 7-9: Using the `Number()`, `String()`, and `Boolean()` constructors

```
<!DOCTYPE html>
<html>
<head>
  <title>Copying values</title>
  <script>
    var num1 = new Number(3.14);
    var num2 = new Number();
    var num3 = 256;
    console.log(typeof num1);
    console.log(num1);
    console.log(num1.valueOf());
    console.log(typeof num2);
    console.log(num2);
    console.log(num2.valueOf());
    console.log(typeof num3);
    console.log(num3);
    console.log(num3.valueOf());
  </script>
</head>

<body>
  Listing 7-9: View the console output
</body>
</html>
```

The output of the code above tells us a lot about `Number()`:

console

```
object
Number {}
3.14
object
Number {}
0
number
256
256
```

Explanation

The **first line** of the code instantiates a `Number` object (`num1`) with its primitive value of `3.14`. **Lines 4-6** log the attributes of this `Number` instance. You can see that it is an object, and the `valueOf()` function returns its primitive value, `3.14`.

The **second line** of the code instantiates a `Number` object with no value specified at construction time. This kind of instantiation sets `num2` as if it were created with the `0` primitive value, as shown by the output between **line 4** and **line 6**.

The **third line** initializes the `num3` variable to a number, with a primitive value of `256`. The last line of the console output is a bit surprising because it shows that `num3` has a `valueOf()` function that retrieves `256`. *How can this be?*

If `num3` is a primitive value, it should not have properties or functions!

Special transition from primitive types to object instances

JavaScript provides a special transition from the primitive number, string, and Boolean values to corresponding `Number`, `String`, and `Boolean` object instances, and vice versa. When using literal values for a number, a string, or a Boolean, you can use the properties and operations of `Number`, `String`, and `Boolean` on these primitive values.

Behind the scenes, the JavaScript engine creates an appropriate **wrapper object** for the literal value and invokes the designated operation or gets a property value. When the on-the-fly created wrapper object is no longer needed, the engine discards it.

So, when **line 12** invoked the `valueOf()` function upon `num3`, which is a primitive value, the JavaScript engine created a wrapper `Number` object instance, invoked the `valueOf()` function, passed the result to the `console.log()` method, and then discarded the temporary wrapper instance.

Here is another example that displays the binary form of the value 42. It uses the `toString()` method of a `Number` object. This method accepts a number between two and 36, which is used as the base of the numeric representation.

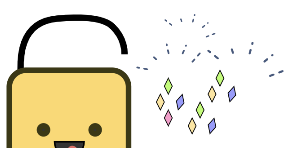
```
var number = 42;
var other = number.toString(2);
console.log(other);
console.log((42).toString(2));
```

This sample shows that you can invoke `toString()` not only on a variable, but also on a literal value.

> **NOTE:** In the last line, the parentheses around 42 are required, otherwise the JavaScript parser would think that "42" is the beginning of a floating-point number and would give an error for `toString` that is not a valid fractional part.

Achievement unlocked!

Congratulations! You've learned how to deal with different types in JavaScript.



different types in javascript.

Great work!

Give yourself a round of applause! :)

In the *next lesson*, we will learn about argument passing in functions.

See you there! :)