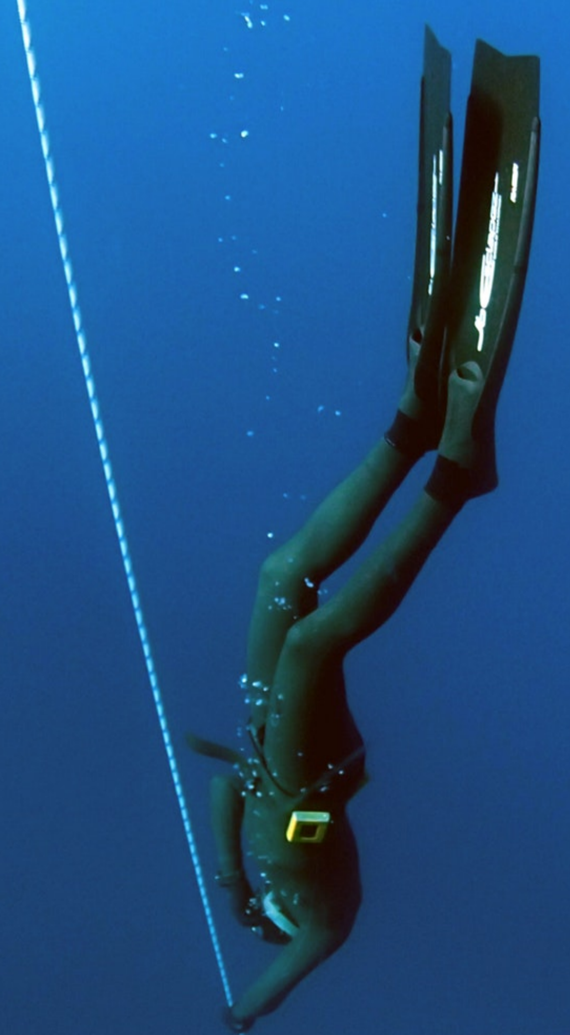
Deep JavaScript

Theory and techniques



Dr. Axel Rauschmayer

Deep JavaScript

Dr. Axel Rauschmayer

2019

Copyright © 2019 by Dr. Axel Rauschmayer
Cover photo by Jakob Boman on Unsplash

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the publisher except for the use of brief quotations in a book review or scholarly journal.

exploringjs.com

Contents

I Frontmatter 5

1 About this book 7
1.1 Where is the homepage of this book? 7
1.2 What is in this book? 7
1.3 What do I get for my money? 8
1.4 How can I preview the content? 8
1.5 How do I report errors? 8
1.6 Tips for reading 8
1.7 Notations and conventions 8
1.8 Acknowledgement 10

II Types, values, variables 11

2 Type coercion in JavaScript 13
2.1 What is type coercion? 13
2.2 Operations that help implement coercion in the ECMAScript specification 16
2.3 Intermission: expressing specification algorithms in JavaScript 18
2.4 Example coercion algorithms 19
2.5 Operations that coerce 27
2.6 Glossary: terms related to type conversion 30

3 The destructuring algorithm 31
3.1 Preparing for the pattern matching algorithm 31
3.2 The pattern matching algorithm 33
3.3 Empty object patterns and Array patterns 35
3.4 Applying the algorithm 36

4 A detailed look at global variables 39
4.1 Scopes 39
4.2 Lexical environments 40
4.3 The global object 40
4.4 In browsers, `globalThis` does not point directly to the global object 40
4.5 The global environment 41
4.6 Conclusion: Why does JavaScript have both normal global variables and the global object? 43
4.7 Further reading and sources of this chapter 45

5 `%` is a remainder operator, not a modulo operator 47
- 5.1 Remainder operator `rem` vs. modulo operator `mod` . . . 47
- 5.2 An intuitive understanding of the remainder operation . . . 48
- 5.3 An intuitive understanding of the modulo operation . . . 48
- 5.4 Similarities and differences between `rem` and `mod` . . . 49
- 5.5 The equations behind remainder and modulo . . . 49
- 5.6 Where are `rem` and `mod` used in programming languages? . . . 51
- 5.7 Further reading and sources of this chapter . . . 52

III Working with data 53

6 Copying objects and Arrays 55
- 6.1 Shallow copying vs. deep copying . . . 55
- 6.2 Shallow copying in JavaScript . . . 56
- 6.3 Deep copying in JavaScript . . . 60
- 6.4 Further reading . . . 63

7 Updating data destructively and non-destructively 65
- 7.1 Examples: updating an object destructively and non-destructively . . . 65
- 7.2 Examples: updating an Array destructively and non-destructively . . . 66
- 7.3 Manual deep updating . . . 67
- 7.4 Implementing generic deep updating . . . 67

8 The problems of shared mutable state and how to avoid them 69
- 8.1 What is shared mutable state and why is it problematic? . . . 69
- 8.2 Avoiding sharing by copying data . . . 71
- 8.3 Avoiding mutations by updating non-destructively . . . 73
- 8.4 Preventing mutations by making data immutable . . . 74
- 8.5 Libraries for avoiding shared mutable state . . . 74

IV OOP: object property attributes 77

9 Property attributes: an introduction 79
- 9.1 The structure of objects . . . 80
- 9.2 Property descriptors . . . 82
- 9.3 Retrieving descriptors for properties . . . 83
- 9.4 Defining properties via descriptors . . . 84
- 9.5 `Object.create()`: Creating objects via descriptors . . . 86
- 9.6 Use cases for `Object.getOwnPropertyDescriptors()` . . . 87
- 9.7 Omitting descriptor properties . . . 89
- 9.8 What property attributes do built-in constructs use? . . . 90
- 9.9 API: property descriptors . . . 93
- 9.10 Further reading . . . 95

10 Where are the remaining chapters? 97

Part I

Frontmatter

Chapter 1

About this book

Contents

- 1.1 Where is the homepage of this book? 7
- 1.2 What is in this book? 7
- 1.3 What do I get for my money? 8
- 1.4 How can I preview the content? 8
- 1.5 How do I report errors? 8
- 1.6 Tips for reading 8
- 1.7 Notations and conventions 8
  - 1.7.1 What is a type signature? Why am I seeing static types in this book? 8
  - 1.7.2 What do the notes with icons mean? 9
- 1.8 Acknowledgement 10

1.1 Where is the homepage of this book?

The homepage of “Deep JavaScript” is `exploringjs.com/deep-js/`

1.2 What is in this book?

This book dives deeply into JavaScript:

- It teaches practical techniques for using the language better.
- It teaches how the language works and why. What it teaches is firmly grounded in the ECMAScript specification (which the book explains and refers to).
- It covers only the language (ignoring platform-specific features such as browser APIs) but not exhaustively. Instead, it focuses on a selection of important topics.

1.3 What do I get for my money?

If you buy this book, you get:

- The current content in four DRM-free versions:
  - PDF file
  - ZIP archive with ad-free HTML
  - EPUB file
  - MOBI file
- Any future content that is added to this edition. How much I can add depends on the sales of this book.

The current price is introductory. It will increase as more content is added.

1.4 How can I preview the content?

On the homepage of this book, there are extensive previews for all versions of this book.

1.5 How do I report errors?

- The HTML version of this book has links to comments at the end of each chapter.
- They jump to GitHub issues, which you can also access directly.

1.6 Tips for reading

- You can read the chapters in any order. Each one is self-contained but occasionally, there are references to other chapters with further information.
- The headings of some sections are marked with "(optional)" meaning that they are non-essential. You will still understand the remainders of their chapters if you skip them.

1.7 Notations and conventions

1.7.1 What is a type signature? Why am I seeing static types in this book?

For example, you may see:

```
Number.isFinite(num: number): boolean
```

That is called the *type signature* of `Number.isFinite()`. This notation, especially the static types `number` of `num` and `boolean` of the result, are not real JavaScript. The notation is borrowed from the compile-to-JavaScript language TypeScript (which is mostly just JavaScript plus static typing).

Why is this notation being used? It helps give you a quick idea of how a function works. The notation is explained in detail in a 2ality blog post, but is usually relatively intuitive.

1.7.2 What do the notes with icons mean?

Reading instructions

Explains how to best read the content.

External content

Points to additional, external, content.

Tip

Gives a tip related to the current content.

Question

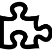
Asks and answers a question pertinent to the current content (think FAQ).

Warning

Warns about pitfalls, etc.

Details

Provides additional details, complementing the current content. It is similar to a footnote.

 **Exercise**

Mentions the path of a test-driven exercise that you can do at that point.

Quiz

Indicates that there is a quiz for the current (part of a) chapter.

1.8 Acknowledgement

- Thanks to Allen Wirfs-Brock for his advice via Twitter and blog post comments. It helped make this book better.
- More people who contributed are acknowledged in the chapters.

Part II

Types, values, variables

Chapter 2

Type coercion in JavaScript

Contents

2.1 What is type coercion? . . . 13
2.1.1 Dealing with type coercion . . . 15
2.2 Operations that help implement coercion in the ECMAScript specification . . . 16
2.2.1 Converting to primitive types and objects . . . 16
2.2.2 Converting to numeric types . . . 16
2.2.3 Converting to property keys . . . 17
2.2.4 Converting to Array indices . . . 17
2.2.5 Converting to Typed Array elements . . . 18
2.3 Intermission: expressing specification algorithms in JavaScript . . . 18
2.4 Example coercion algorithms . . . 19
2.4.1 `ToPrimitive()` . . . 19
2.4.2 `ToString()` and related operations . . . 22
2.4.3 `ToPropertyKey()` . . . 26
2.4.4 `ToNumeric()` and related operations . . . 26
2.5 Operations that coerce . . . 27
2.5.1 Addition operator (+) . . . 27
2.5.2 Abstract Equality Comparison (==) . . . 28
2.6 Glossary: terms related to type conversion . . . 30

In this chapter, we examine the role of *type coercion* in JavaScript. We will go relatively deeply into this subject and, e.g., look into how the ECMAScript specification handles coercion.

2.1 What is type coercion?

Each operation (function, operator, etc.) expects its parameters to have certain types. If a value doesn't have the right type for a parameter, three common options for, e.g., a function are:

1. The function can throw an exception:

```
function multiply(x, y) {
  if (typeof x !== 'number' || typeof y !== 'number') {
    throw new TypeError();
  }
  // ···
}
```

2. The function can return an error value:

```
function multiply(x, y) {
  if (typeof x !== 'number' || typeof y !== 'number') {
    return NaN;
  }
  // ···
}
```

3. The function can convert its arguments to useful values:

```
function multiply(x, y) {
  if (typeof x !== 'number') {
    x = Number(x);
  }
  if (typeof y !== 'number') {
    y = Number(y);
  }
  // ···
}
```

In (3), the operation performs an implicit type conversion. That is called *type coercion*.

JavaScript initially didn't have exceptions, which is why it uses coercion and error values for most of its operations:

```
// Coercion
assert.equal(3 * true, 3);

// Error values
assert.equal(1 / 0, Infinity);
assert.equal(Number('xyz'), NaN);
```

However, there are also cases (especially when it comes to newer features) where it throws exceptions if an argument doesn't have the right type:

- Accessing properties of `null` or `undefined`:

```
> undefined.prop
TypeError: Cannot read property 'prop' of undefined
> null.prop
TypeError: Cannot read property 'prop' of null
> 'prop' in null
TypeError: Cannot use 'in' operator to search for 'prop' in null
```

- Using symbols:

```
> 6 / Symbol()
TypeError: Cannot convert a Symbol value to a number
```

- Mixing bigints and numbers:

```
> 6 / 3n
TypeError: Cannot mix BigInt and other types
```

- New-calling or function-calling values that don't support that operation:

```
> 123()
TypeError: 123 is not a function
> (class {})()
TypeError: Class constructor  cannot be invoked without 'new'

> new 123
TypeError: 123 is not a constructor
> new (() => {})
TypeError: (intermediate value) is not a constructor
```

- Changing read-only properties (only throws in strict mode):

```
> 'abc'.length = 1
TypeError: Cannot assign to read only property 'length'
> Object.freeze({prop:3}).prop = 1
TypeError: Cannot assign to read only property 'prop'
```

2.1.1 Dealing with type coercion

Two common ways of dealing with coercion are:

- A caller can explicitly convert values so that they have the right types. For example, in the following interaction, we want to multiply two numbers encoded as strings:

```
let x = '3';
let y = '2';
assert.equal(Number(x) * Number(y), 6);
```

- A caller can let the operation make the conversion for them:

```
let x = '3';
let y = '2';
assert.equal(x * y, 6);
```

I usually prefer the former, because it clarifies my intention: I expect `x` and `y` not to be numbers, but want to multiply two numbers.

2.2 Operations that help implement coercion in the ECMAScript specification

The following sections describe the most important internal functions used by the ECMAScript specification to convert actual parameters to expected types.

For example, in TypeScript, we would write:

```
function isNaN(number: number) {
  // ···
}
```

In the specification, this looks as follows (translated to JavaScript, so that it is easier to understand):

```
function isNaN(number) {
  let num = ToNumber(number);
  // ···
}
```

2.2.1 Converting to primitive types and objects

Whenever primitive types or objects are expected, the following conversion functions are used:

- `ToBoolean()`
- `ToNumber()`
- `ToBigInt()`
- `ToString()`
- `ToObject()`

These internal functions have analogs in JavaScript that are very similar:

```
> Boolean(0)
false
> Boolean(1)
true

> Number('123')
123
```

After the introduction of bigints, which exists alongside numbers, the specification often uses `ToNumeric()` where it previously used `ToNumber()`. Read on for more information.

2.2.2 Converting to numeric types

At the moment, JavaScript has two built-in numeric types: number and bigint.

- `ToNumeric()` returns a numeric value `num`. Its callers usually invoke a method `mthd` of the specification type of `num`:

  ```
  Type(num)::mthd(···)
  ```

Among others, the following operations use `ToNumeric`:

- Prefix and postfix ++ operator
- * operator

- `ToInteger(x)` is used whenever a number without a fraction is expected. The range of the result is often restricted further afterwards.
  - It calls `ToNumber(x)` and removes the fraction (similar to `Math.trunc()`).
  - Operations that use `ToInteger()`:
    - `Number.prototype.toString(radix?)`
    - `String.prototype.codePointAt(pos)`
    - `Array.prototype.slice(start, end)`
    - Etc.
- `ToInt32()`, `ToUint32()` coerce numbers to 32-bit integers and are used by bitwise operators (see tbl. 2.1).
  - `ToInt32()`: signed, range $[-2^{31}, 2^{31}-1]$ (limits are included)
  - `ToUint32()`: unsigned (hence the `U`), range $[0, 2^{32}-1]$ (limits are included)

Table 2.1: Coercion of the operands of bitwise number operators (BigInt operators don't limit the number of bits).

Operator	Left operand	Right operand	result type
<<	`ToInt32()`	`ToUint32()`	`Int32`
signed >>	`ToInt32()`	`ToUint32()`	`Int32`
unsigned >>>	`ToInt32()`	`ToUint32()`	`Uint32`
&, ^, |	`ToInt32()`	`ToUint32()`	`Int32`
~	—	`ToInt32()`	`Int32`

2.2.3 Converting to property keys

`ToPropertyKey()` returns a string or a symbol and is used by:

- The bracket operator `[]`
- Computed property keys in object literals
- The left-hand side of the `in` operator
- `Object.defineProperty(_, P, _)`
- `Object.fromEntries()`
- `Object.getOwnPropertyDescriptor()`
- `Object.prototype.hasOwnProperty()`
- `Object.prototype.propertyIsEnumerable()`
- Several methods of `Reflect`

2.2.4 Converting to Array indices

- `ToLength()` is used (directly) mainly for string indices.
  - Helper function for `ToIndex()`
  - Range of result `l`: $0 \le l \le 2^{53}-1$

- `ToIndex()` is used for Typed Array indices.
  - Main difference with `ToLength()`: throws an exception if argument is out of range.
  - Range of result `i`: $0 \leq i \leq 2^{53}-1$
- `ToUint32()` is used for Array indices.
  - Range of result `i`: $0 \leq i < 2^{32}-1$ (the upper limit is excluded, to leave room for the `.length`)

2.2.5 Converting to Typed Array elements

When we set the value of a Typed Array element, one of the following conversion functions is used:

- `ToInt8()`
- `ToUint8()`
- `ToUint8Clamp()`
- `ToInt16()`
- `ToUint16()`
- `ToInt32()`
- `ToUint32()`
- `ToBigInt64()`
- `ToBigUint64()`

2.3 Intermission: expressing specification algorithms in JavaScript

In the remainder of this chapter, we'll encounter several specification algorithms, but "implemented" as JavaScript. The following list shows how some frequently used patterns are translated from specification to JavaScript:

- Spec: If Type(value) is String
  JavaScript: `if (TypeOf(value) === 'string')`
  (very loose translation; `TypeOf()` is defined below)

- Spec: If IsCallable(method) is true
  JavaScript: `if (IsCallable(method))`
  (`IsCallable()` is defined below)

- Spec: Let numValue be ToNumber(value)
  JavaScript: `let numValue = Number(value)`

- Spec: Let isArray be IsArray(O)
  JavaScript: `let isArray = Array.isArray(O)`

- Spec: If O has a [[NumberData]] internal slot
  JavaScript: `if ('__NumberData__' in O)`

- Spec: Let tag be Get(O, @@toStringTag)
  JavaScript: `let tag = O[Symbol.toStringTag]`

- Spec: Return the string-concatenation of “[object ”, tag, and ”]”.
  JavaScript: `return '[object ' + tag + ']';`

`let` (and not `const`) is used to match the language of the specification.

Some things are omitted – for example, the ReturnIfAbrupt shorthands `?` and `!`.

```
/**
 * An improved version of typeof
 */
function TypeOf(value) {
  const result = typeof value;
  switch (result) {
    case 'function':
      return 'object';
    case 'object':
      if (value === null) {
        return 'null';
      } else {
        return 'object';
      }
    default:
      return result;
  }
}

function IsCallable(x) {
  return typeof x === 'function';
}
```

2.4 Example coercion algorithms

2.4.1 `ToPrimitive()`

The operation `ToPrimitive()` is an intermediate step for many coercion algorithms (some of which we’ll see later in this chapter). It converts an arbitrary values to primitive values.

`ToPrimitive()` is used often in the spec because most operators can only work with primitive values. For example, we can use the addition operator (+) to add numbers and to concatenate strings, but we can’t use it to concatenate Arrays.

This is what the JavaScript version of `ToPrimitive()` looks like:

```
/**
 * @param hint Which type is preferred for the result:
 *             string, number, or don’t care?
 */
function ToPrimitive(input: any,
  hint: 'string'|'number'|'default' = 'default') {
    if (TypeOf(input) === 'object') {
```

```
      let exoticToPrim = input[Symbol.toPrimitive]; // (A)
      if (exoticToPrim !== undefined) {
        let result = exoticToPrim.call(input, hint);
        if (TypeOf(result) !== 'object') {
          return result;
        }
        throw new TypeError();
      }
      if (hint === 'default') {
        hint = 'number';
      }
      return OrdinaryToPrimitive(input, hint);
    } else {
      // input is already primitive
      return input;
    }
  }
```

`ToPrimitive()` lets objects override the conversion to primitive via `Symbol.toPrimitive` (line A). If an object doesn't do that, it is passed on to `OrdinaryToPrimitive()`:

```
function OrdinaryToPrimitive(O: object, hint: 'string' | 'number') {
  let methodNames;
  if (hint === 'string') {
    methodNames = ['toString', 'valueOf'];
  } else {
    methodNames = ['valueOf', 'toString'];
  }
  for (let name of methodNames) {
    let method = O[name];
    if (IsCallable(method)) {
      let result = method.call(O);
      if (TypeOf(result) !== 'object') {
        return result;
      }
    }
  }
  throw new TypeError();
}
```

2.4.1.1 Which hints do callers of `ToPrimitive()` use?

The parameter `hint` can have one of three values:

- `'number'` means: if possible, `input` should be converted to a number.
- `'string'` means: if possible, `input` should be converted to a string.
- `'default'` means: there is no preference for either numbers or strings.

These are a few examples of how various operations use `ToPrimitive()`:

- `hint === 'number'`. The following operations prefer numbers:

    - `ToNumeric()`
    - `ToNumber()`
    - `ToBigInt()`, `BigInt()`
    - Abstract Relational Comparison (<)
- `hint === 'string'`. The following operations prefer strings:
    - `ToString()`
    - `ToPropertyKey()`
- `hint === 'default'`. The following operations are neutral w.r.t. the type of the returned primitive value:
    - Abstract Equality Comparison (==)
    - Addition Operator (+)
    - `new Date(value)` (`value` can be either a number or a string)

As we have seen, the default behavior is for `'default'` being handled as if it were `'number'`. Only instances of `Symbol` and `Date` override this behavior (shown later).

2.4.1.2 Which methods are called to convert objects to Primitives?

If the conversion to primitive isn't overridden via `Symbol.toPrimitive`, `OrdinaryToPrimitive()` calls either or both of the following two methods:

- `'toString'` is called first if `hint` indicates that we'd like the primitive value to be a string.
- `'valueOf'` is called first if `hint` indicates that we'd like the primitive value to be a number.

The following code demonstrates how that works:

```
const obj = {
  toString() { return 'a' },
  valueOf() { return 1 },
};

// String() prefers strings:
assert.equal(String(obj), 'a');

// Number() prefers numbers:
assert.equal(Number(obj), 1);
```

A method with the property key `Symbol.toPrimitive` overrides the normal conversion to primitive. That is only done twice in the standard library:

- `Symbol.prototype[Symbol.toPrimitive](hint)`
    - If the receiver is an instance of `Symbol`, this method always returns the wrapped symbol.
    - The rationale is that instances of `Symbol` have a `.toString()` method that returns strings. But even if `hint` is `'string'`, `.toString()` should not be called so that we don't accidentally convert instances of `Symbol` to strings (which are a completely different kind of property key).
- `Date.prototype[Symbol.toPrimitive](hint)`
    - Explained in more detail next.

2.4.1.3 `Date.prototype[Symbol.toPrimitive]()`

This is how Dates handle being converted to primitive values:

```
Date.prototype[Symbol.toPrimitive] = function (
  hint: 'default' | 'string' | 'number') {
    let O = this;
    if (TypeOf(O) !== 'object') {
      throw new TypeError();
    }
    let tryFirst;
    if (hint === 'string' || hint === 'default') {
      tryFirst = 'string';
    } else if (hint === 'number') {
      tryFirst = 'number';
    } else {
      throw new TypeError();
    }
    return OrdinaryToPrimitive(O, tryFirst);
  };
```

The only difference with the default algorithm is that `'default'` becomes `'string'` (and not `'number'`). This can be observed if we use operations that set `hint` to `'default'`:

- The `==` operator coerces objects to primitives (with a default hint) if the other operand is a primitive value other than `undefined`, `null`, and `boolean`. In the following interaction, we can see that the result of coercing the date is a string:

  ```
  const d = new Date('2222-03-27');
  assert.equal(
    d == 'Wed Mar 27 2222 01:00:00 GMT+0100'
         + ' (Central European Standard Time)',
    true);
  ```

- The `+` operator coerces both operands to primitives (with a default hint). If one of the results is a string, it performs string concatenation (otherwise it performs numeric addition). In the following interaction, we can see that the result of coercing the date is a string because the operator returns a string.

  ```
  const d = new Date('2222-03-27');
  assert.equal(
    123 + d,
    '123Wed Mar 27 2222 01:00:00 GMT+0100'
      + ' (Central European Standard Time)');
  ```

2.4.2 `ToString()` and related operations

This is the JavaScript version of `ToString()`:

```
function ToString(argument) {
  if (argument === undefined) {
    return 'undefined';
```

```
  } else if (argument === null) {
    return 'null';
  } else if (argument === true) {
    return 'true';
  } else if (argument === false) {
    return 'false';
  } else if (TypeOf(argument) === 'number') {
    return Number.toString(argument);
  } else if (TypeOf(argument) === 'string') {
    return argument;
  } else if (TypeOf(argument) === 'symbol') {
    throw new TypeError();
  } else if (TypeOf(argument) === 'bigint') {
    return BigInt.toString(argument);
  } else {
    // argument is an object
    let primValue = ToPrimitive(argument, 'string'); // (A)
    return ToString(primValue);
  }
}
```

Note how this function uses `ToPrimitive()` as an intermediate step for objects, before converting the primitive result to a string (line A).

`ToString()` deviates in an interesting way from how `String()` works: If `argument` is a symbol, the former throws a `TypeError` while the latter doesn't. Why is that? The default for symbols is that converting them to strings throws exceptions:

```
> const sym = Symbol('sym');

> ''+sym
TypeError: Cannot convert a Symbol value to a string
> `${sym}`
TypeError: Cannot convert a Symbol value to a string
```

That default is overridden in `String()` and `Symbol.prototype.toString()` (both are described in the next subsections):

```
> String(sym)
'Symbol(sym)'
> sym.toString()
'Symbol(sym)'
```

2.4.2.1 `String()`

```
function String(value) {
  let s;
  if (value === undefined) {
    s = '';
  } else {
    if (new.target === undefined && TypeOf(value) === 'symbol') {
```

```
      // This function was function-called and value is a symbol
      return SymbolDescriptiveString(value);
    }
    s = ToString(value);
  }
  if (new.target === undefined) {
    // This function was function-called
    return s;
  }
  // This function was new-called
  return StringCreate(s, new.target.prototype); // simplified!
}
```

`String()` works differently, depending on whether it is invoked via a function call or via `new`. It uses `new.target` to distinguish the two.

These are the helper functions `StringCreate()` and `SymbolDescriptiveString()`:

```
/**
 * Creates a String instance that wraps `value`
 * and has the given protoype.
 */
function StringCreate(value, prototype) {
  // ···
}

function SymbolDescriptiveString(sym) {
  assert.equal(TypeOf(sym), 'symbol');
  let desc = sym.description;
  if (desc === undefined) {
    desc = '';
  }
  assert.equal(TypeOf(desc), 'string');
  return 'Symbol('+desc+')';
}
```

2.4.2.2 `Symbol.prototype.toString()`

In addition to `String()`, we can also use method `.toString()` to convert a symbol to a string. Its specification looks as follows.

```
Symbol.prototype.toString = function () {
  let sym = thisSymbolValue(this);
  return SymbolDescriptiveString(sym);
};
function thisSymbolValue(value) {
  if (TypeOf(value) === 'symbol') {
    return value;
  }
  if (TypeOf(value) === 'object' && '__SymbolData__' in value) {
```

```
      let s = value.__SymbolData__;
      assert.equal(TypeOf(s), 'symbol');
      return s;
    }
  }
```

2.4.2.3 `Object.prototype.toString`

The default specification for `.toString()` looks as follows:

```
  Object.prototype.toString = function () {
    if (this === undefined) {
      return '[object Undefined]';
    }
    if (this === null) {
      return '[object Null]';
    }
    let O = ToObject(this);
    let isArray = Array.isArray(O);
    let builtinTag;
    if (isArray) {
      builtinTag = 'Array';
    } else if ('__ParameterMap__' in O) {
      builtinTag = 'Arguments';
    } else if ('__Call__' in O) {
      builtinTag = 'Function';
    } else if ('__ErrorData__' in O) {
      builtinTag = 'Error';
    } else if ('__BooleanData__' in O) {
      builtinTag = 'Boolean';
    } else if ('__NumberData__' in O) {
      builtinTag = 'Number';
    } else if ('__StringData__' in O) {
      builtinTag = 'String';
    } else if ('__DateValue__' in O) {
      builtinTag = 'Date';
    } else if ('__RegExpMatcher__' in O) {
      builtinTag = 'RegExp';
    } else {
      builtinTag = 'Object';
    }
    let tag = O[Symbol.toStringTag];
    if (TypeOf(tag) !== 'string') {
      tag = builtinTag;
    }
    return '[object ' + tag + ']';
  };
```

This operation is used if we convert plain objects to strings:

```
> String({})
'[object Object]'
```

By default, it is also used if we convert instances of classes to strings:

```
class MyClass {}
assert.equal(
  String(new MyClass()), '[object Object]');
```

Normally, we would override `.toString()` in order to configure the string representation of `MyClass`, but we can also change what comes after "`object`" inside the string with the square brackets:

```
class MyClass {}
MyClass.prototype[Symbol.toStringTag] = 'Custom!';
assert.equal(
  String(new MyClass()), '[object Custom!]');
```

It is interesting to compare the overriding versions of `.toString()` with the original version in `Object.prototype`:

```
> ['a', 'b'].toString()
'a,b'
> Object.prototype.toString.call(['a', 'b'])
'[object Array]'

> /^abc$/.toString()
'/^abc$/'
> Object.prototype.toString.call(/^abc$/)
'[object RegExp]'
```

2.4.3 `ToPropertyKey()`

`ToPropertyKey()` is used by, among others, the bracket operator. This is how it works:

```
function ToPropertyKey(argument) {
  let key = ToPrimitive(argument, 'string'); // (A)
  if (TypeOf(key) === 'symbol') {
    return key;
  }
  return ToString(key);
}
```

Once again, objects are converted to primitives before working with primitives.

2.4.4 `ToNumeric()` and related operations

`ToNumeric()` is used by, among others, by the multiplication operator (`*`). This is how it works:

```
function ToNumeric(value) {
  let primValue = ToPrimitive(value, 'number');
  if (TypeOf(primValue) === 'bigint') {
```

```
    return primValue;
  }
  return ToNumber(primValue);
}
```

2.4.4.1 `ToNumber()`

`ToNumber()` works as follows:

```
function ToNumber(argument) {
  if (argument === undefined) {
    return NaN;
  } else if (argument === null) {
    return +0;
  } else if (argument === true) {
    return 1;
  } else if (argument === false) {
    return +0;
  } else if (TypeOf(argument) === 'number') {
    return argument;
  } else if (TypeOf(argument) === 'string') {
    return parseTheString(argument); // not shown here
  } else if (TypeOf(argument) === 'symbol') {
    throw new TypeError();
  } else if (TypeOf(argument) === 'bigint') {
    throw new TypeError();
  } else {
    // argument is an object
    let primValue = ToPrimitive(argument, 'number');
    return ToNumber(primValue);
  }
}
```

The structure of `ToNumber()` is similar to the structure of `ToString()`.

2.5 Operations that coerce

2.5.1 Addition operator (+)

This is how JavaScript's addition operator is specified:

```
function Addition(leftHandSide, rightHandSide) {
  let lprim = ToPrimitive(leftHandSide);
  let rprim = ToPrimitive(rightHandSide);
  if (TypeOf(lprim) === 'string' || TypeOf(rprim) === 'string') { // (A)
    return ToString(lprim) + ToString(rprim);
  }
  let lnum = ToNumeric(lprim);
  let rnum = ToNumeric(rprim);
```

```
    if (TypeOf(lnum) !== TypeOf(rnum)) {
      throw new TypeError();
    }
    let T = Type(lnum);
    return T.add(lnum, rnum); // (B)
  }
```

Steps of this algorithm:

- Both operands are converted to primitive values.
- If one of the results is a string, both are converted to strings and concatenated (line A).
- Otherwise, both operands are converted to numeric values and added (line B). `Type()` returns the ECMAScript specification type of `lnum`. `.add()` is a method of numeric types.

2.5.2 Abstract Equality Comparison (==)

```
/** Loose equality (==) */
function abstractEqualityComparison(x, y) {
  if (TypeOf(x) === TypeOf(y)) {
    // Use strict equality (===)
    return strictEqualityComparison(x, y);
  }

  // Comparing null with undefined
  if (x === null && y === undefined) {
    return true;
  }
  if (x === undefined && y === null) {
    return true;
  }

  // Comparing a number and a string
  if (TypeOf(x) === 'number' && TypeOf(y) === 'string') {
    return abstractEqualityComparison(x, Number(y));
  }
  if (TypeOf(x) === 'string' && TypeOf(y) === 'number') {
    return abstractEqualityComparison(Number(x), y);
  }

  // Comparing a bigint and a string
  if (TypeOf(x) === 'bigint' && TypeOf(y) === 'string') {
    let n = StringToBigInt(y);
    if (Number.isNaN(n)) {
      return false;
    }
    return abstractEqualityComparison(x, n);
  }
```

```
  if (TypeOf(x) === 'string' && TypeOf(y) === 'bigint') {
    return abstractEqualityComparison(y, x);
  }

  // Comparing a boolean with a non-boolean
  if (TypeOf(x) === 'boolean') {
    return abstractEqualityComparison(Number(x), y);
  }
  if (TypeOf(y) === 'boolean') {
    return abstractEqualityComparison(x, Number(y));
  }

  // Comparing an object with a primitive
  // (other than undefined, null, a boolean)
  if (['string', 'number', 'bigint', 'symbol'].includes(TypeOf(x))
    && TypeOf(y) === 'object') {
      return abstractEqualityComparison(x, ToPrimitive(y));
    }
  if (TypeOf(x) === 'object'
    && ['string', 'number', 'bigint', 'symbol'].includes(TypeOf(y))) {
      return abstractEqualityComparison(ToPrimitive(x), y);
    }

  // Comparing a bigint with a number
  if ((TypeOf(x) === 'bigint' && TypeOf(y) === 'number')
    || (TypeOf(x) === 'number' && TypeOf(y) === 'bigint')) {
      if ([NaN, +Infinity, -Infinity].includes(x)
        || [NaN, +Infinity, -Infinity].includes(y)) {
          return false;
        }
      if (isSameMathematicalValue(x, y)) {
        return true;
      } else {
        return false;
      }
    }

  return false;
}
```

The following operations are not shown here:

- `strictEqualityComparison()`
- `StringToBigInt()`
- `isSameMathematicalValue()`

2.6 Glossary: terms related to type conversion

Now that we have taken a closer look at how JavaScript's type coercion works, let's conclude with a brief glossary of terms related to type conversion:

- In *type conversion*, we want the output value to have a given type. If the input value already has that type, it is simply returned unchanged. Otherwise, it is converted to a value that has the desired type.
- *Explicit type conversion* means that the programmer uses an operation (a function, an operator, etc.) to trigger a type conversion. Explicit conversions can be:
  - *Checked*: If a value can't be converted, an exception is thrown.
  - *Unchecked*: If a value can't be converted, an error value is returned.
- What *type casting* is, depends on the programming language. For example, in Java, it is explicit checked type conversion.
- *Type coercion* is implicit type conversion: An operation automatically converts its arguments to the types it needs. Can be checked or unchecked or something in-between.

[Source: Wikipedia]

Chapter 3

The destructuring algorithm

Contents

3.1 Preparing for the pattern matching algorithm . . . 31
3.1.1 Using declarative rules for specifying the matching algorithm 32
3.1.2 Evaluating expressions based on the declarative rules . . . 33
3.2 The pattern matching algorithm . . . 33
3.2.1 Patterns . . . 33
3.2.2 Rules for variable . . . 33
3.2.3 Rules for object patterns . . . 33
3.2.4 Rules for Array patterns . . . 34
3.3 Empty object patterns and Array patterns . . . 35
3.4 Applying the algorithm . . . 36
3.4.1 Background: passing parameters via matching . . . 36
3.4.2 Using `move2()` . . . 37
3.4.3 Using `move1()` . . . 37
3.4.4 Conclusion: Default values are a feature of pattern parts . . . 38

In this chapter, we look at destructuring from a different angle: as a recursive pattern matching algorithm.

The algorithm will give us a better understanding of default values. That will be useful at the end, where we'll try to figure out how the following two functions differ:

```
function move({x=0, y=0} = {})         { ··· }
function move({x, y} = { x: 0, y: 0 }) { ··· }
```

3.1 Preparing for the pattern matching algorithm

A destructuring assignment looks like this:

```
«pattern» = «value»
```

We want to use `pattern` to extract data from `value`.

We will now look at an algorithm for performing this kind of assignment. This algorithm is known in functional programming as *pattern matching* (short: *matching*). It specifies the operator ← ("match against") that matches a `pattern` against a `value` and assigns to variables while doing so:

```
«pattern» ← «value»
```

We will only explore destructuring assignment, but destructuring variable declarations and destructuring parameter definitions work similarly. We won't go into advanced features, either: Computed property keys, property value shorthands, and object properties and array elements as assignment targets, are beyond the scope of this chapter.

The specification for the match operator consists of declarative rules that descend into the structures of both operands. The declarative notation may take some getting used to, but it makes the specification more concise.

3.1.1 Using declarative rules for specifying the matching algorithm

The declarative rules used in this chapter operate on input and produce the result of the algorithm via side effects. This is one such rule (which we'll see again later):

- (2c) `{key: «pattern», «properties»} ← obj`

  ```
  «pattern» ← obj.key
  {«properties»} ← obj
  ```

This rule has the following parts:

- (2c) is the *number* of the rule. The number is used to refer to the rule.
- The *head* (first line) describes what the input must look like so that this rule can be applied.
- The *body* (remaining lines) describes what happens if the rule is applied.

In rule (2c), the head means that this rule can be applied if there is an object pattern with at least one property (whose key is `key`) and zero or more remaining properties. The effect of this rule is that execution continues with the property value pattern being matched against `obj.key` and the remaining properties being matched against `obj`.

Let's consider one more rule from this chapter:

- (2e) `{} ← obj` (no properties left)

  ```
  // We are finished
  ```

In rule (2e), the head means that this rule is executed if the empty object pattern `{}` is matched against a value `obj`. The body means that, in this case, we are done.

Together, rule (2c) and rule (2e) form a declarative loop that iterates over the properties of the pattern on the left-hand side of the arrow.

3.1.2 Evaluating expressions based on the declarative rules

The complete algorithm is specified via a sequence of declarative rules. Let's assume we want to evaluate the following matching expression:

```
{first: f, last: l} ← obj
```

To apply a sequence of rules, we go over them from top to bottom and execute the first applicable rule. If there is a matching expression in the body of that rule, the rules are applied again. And so on.

Sometimes the head includes a condition that also determines if a rule is applicable – for example:

- (3a) `[«elements»] ← non_iterable`
  `if (!isIterable(non_iterable))`

  ```
  throw new TypeError();
  ```

3.2 The pattern matching algorithm

3.2.1 Patterns

A pattern is either:

- A variable: `x`
- An object pattern: `{«properties»}`
- An Array pattern: `[«elements»]`

The next three sections specify rules for handling these three cases in matching expressions.

3.2.2 Rules for variable

- (1) `x ← value` (including `undefined` and `null`)

  ```
  x = value
  ```

3.2.3 Rules for object patterns

- (2a) `{«properties»} ← undefined` (illegal value)

  ```
  throw new TypeError();
  ```

- (2b) `{«properties»} ← null` (illegal value)

  ```
  throw new TypeError();
  ```

- (2c) `{key: «pattern», «properties»} ← obj`

  ```
  «pattern» ← obj.key
  {«properties»} ← obj
  ```

- (2d) `{key: «pattern» = default_value, «properties»} ← obj`

```
const tmp = obj.key;
if (tmp !== undefined) {
  «pattern» ← tmp
} else {
  «pattern» ← default_value
}
{«properties»} ← obj
```

- (2e) `{} ← obj` (no properties left)

```
// We are finished
```

Rules 2a and 2b deal with illegal values. Rules 2c–2e loop over the properties of the pattern. In rule 2d, we can see that a default value provides an alternative to match against if there is no matching property in `obj`.

3.2.4 Rules for Array patterns

Array pattern and iterable. The algorithm for Array destructuring starts with an Array pattern and an iterable:

- (3a) `[«elements»] ← non_iterable` (illegal value)
  `if (!isIterable(non_iterable))`

```
throw new TypeError();
```

- (3b) `[«elements»] ← iterable`
  `if (isIterable(iterable))`

```
const iterator = iterable[Symbol.iterator]();
«elements» ← iterator
```

Helper function:

```
function isIterable(value) {
  return (value !== null
    && typeof value === 'object'
    && typeof value[Symbol.iterator] === 'function');
}
```

Array elements and iterator. The algorithm continues with:

- The elements of the pattern (left-hand side of the arrow)
- The iterator that was obtained from the iterable (right-hand side of the arrow)

These are the rules:

- (3c) `«pattern», «elements» ← iterator`

```
«pattern» ← getNext(iterator) // undefined after last item
«elements» ← iterator
```

- (3d) `«pattern» = default_value, «elements» ← iterator`

```
const tmp = getNext(iterator);  // undefined after last item
if (tmp !== undefined) {
```

```
  «pattern» ← tmp
} else {
  «pattern» ← default_value
}
«elements» ← iterator
```

- (3e) `, «elements» ← iterator` (hole, elision)

```
getNext(iterator); // skip
«elements» ← iterator
```

- (3f) `...«pattern» ← iterator` (always last part!)

```
const tmp = [];
for (const elem of iterator) {
  tmp.push(elem);
}
«pattern» ← tmp
```

- (3g) `← iterator` (no elements left)

```
// We are finished
```

Helper function:

```
function getNext(iterator) {
  const {done,value} = iterator.next();
  return (done ? undefined : value);
}
```

An iterator being finished is similar to missing properties in objects.

3.3 Empty object patterns and Array patterns

Interesting consequence of the algorithm's rules: We can destructure with empty object patterns and empty Array patterns.

Given an empty object pattern `{}`: If the value to be destructured is neither `undefined` nor `null`, then nothing happens. Otherwise, a `TypeError` is thrown.

```
const {} = 123; // OK, neither undefined nor null
assert.throws(
  () => {
    const {} = null;
  },
  /^TypeError: Cannot destructure 'null' as it is null.$/)
```

Given an empty Array pattern `[]`: If the value to be destructured is iterable, then nothing happens. Otherwise, a `TypeError` is thrown.

```
const [] = 'abc'; // OK, iterable
assert.throws(
  () => {
    const [] = 123; // not iterable
```

```
  },
  /^TypeError: 123 is not iterable$/)
```

In other words: Empty destructuring patterns force values to have certain characteristics, but have no other effects.

3.4 Applying the algorithm

In JavaScript, named parameters are simulated via objects: The caller uses an object literal and the callee uses destructuring. This simulation is explained in detail in "JavaScript for impatient programmers". The following code shows an example: function `move1()` has two named parameters, `x` and `y`:

```
function move1({x=0, y=0} = {}) { // (A)
  return [x, y];
}
assert.deepEqual(
  move1({x: 3, y: 8}), [3, 8]);
assert.deepEqual(
  move1({x: 3}), [3, 0]);
assert.deepEqual(
  move1({}), [0, 0]);
assert.deepEqual(
  move1(), [0, 0]);
```

There are three default values in line A:

- The first two default values allow us to omit `x` and `y`.
- The third default value allows us to call `move1()` without parameters (as in the last line).

But why would we define the parameters as in the previous code snippet? Why not as follows?

```
function move2({x, y} = { x: 0, y: 0 }) {
  return [x, y];
}
```

To see why `move1()` is correct, we are going to use both functions in two examples. Before we do that, let's see how the passing of parameters can be explained via matching.

3.4.1 Background: passing parameters via matching

For function calls, *formal parameters* (inside function definitions) are matched against *actual parameters* (inside function calls). As an example, take the following function definition and the following function call.

```
function func(a=0, b=0) { ··· }
func(1, 2);
```

The parameters `a` and `b` are set up similarly to the following destructuring.

```
[a=0, b=0] ← [1, 2]
```

3.4.2 Using `move2()`

Let's examine how destructuring works for `move2()`.

Example 1. The function call `move2()` leads to this destructuring:

```
[{x, y} = { x: 0, y: 0 }] ← []
```

The single Array element on the left-hand side does not have a match on the right-hand side, which is why `{x,y}` is matched against the default value and not against data from the right-hand side (rules 3b, 3d):

```
{x, y} ← { x: 0, y: 0 }
```

The left-hand side contains *property value shorthands*. It is an abbreviation for:

```
{x: x, y: y} ← { x: 0, y: 0 }
```

This destructuring leads to the following two assignments (rules 2c, 1):

```
x = 0;
y = 0;
```

This is what we wanted. However, in the next example, we are not as lucky.

Example 2. Let's examine the function call `move2({z: 3})` which leads to the following destructuring:

```
[{x, y} = { x: 0, y: 0 }] ← [{z: 3}]
```

There is an Array element at index 0 on the right-hand side. Therefore, the default value is ignored and the next step is (rule 3d):

```
{x, y} ← { z: 3 }
```

That leads to both `x` and `y` being set to `undefined`, which is not what we want. The problem is that `{x,y}` is not matched against the default value, anymore, but against `{z:3}`.

3.4.3 Using `move1()`

Let's try `move1()`.

Example 1: `move1()`

```
[{x=0, y=0} = {}] ← []
```

We don't have an Array element at index 0 on the right-hand side and use the default value (rule 3d):

```
{x=0, y=0} ← {}
```

The left-hand side contains property value shorthands, which means that this destructuring is equivalent to:

```
{x: x=0, y: y=0} ← {}
```

Neither property `x` nor property `y` have a match on the right-hand side. Therefore, the default values are used and the following destructurings are performed next (rule 2d):

```
x ← 0
y ← 0
```

That leads to the following assignments (rule 1):

```
x = 0
y = 0
```

Here, we get what we want. Let's see if our luck holds with the next example.

Example 2: `move1({z: 3})`

```
[{x=0, y=0} = {}] ← [{z: 3}]
```

The first element of the Array pattern has a match on the right-hand side and that match is used to continue destructuring (rule 3d):

```
{x=0, y=0} ← {z: 3}
```

Like in example 1, there are no properties `x` and `y` on the right-hand side and the default values are used:

```
x = 0
y = 0
```

It works as desired! This time, the pattern with `x` and `y` being matched against `{z:3}` is not a problem, because they have their own local default values.

3.4.4 Conclusion: Default values are a feature of pattern parts

The examples demonstrate that default values are a feature of pattern parts (object properties or Array elements). If a part has no match or is matched against `undefined` then the default value is used. That is, the pattern is matched against the default value, instead.

Chapter 4

A detailed look at global variables

Contents

4.1 Scopes 39
4.2 Lexical environments 40
4.3 The global object 40
4.4 In browsers, `globalThis` does not point directly to the global object 40
4.5 The global environment 41
4.5.1 Script scope and module scopes 42
4.5.2 Creating variables: declarative record vs. object record 42
4.5.3 Getting or setting variables 43
4.5.4 Global ECMAScript variables and global host variables 43
4.6 Conclusion: Why does JavaScript have both normal global variables and the global object? 43
4.7 Further reading and sources of this chapter 45

In this chapter, we take a detailed look at how JavaScript's global variables work. Several interesting phenomena play a role: the scope of scripts, the so-called *global object*, and more.

4.1 Scopes

The *lexical scope* (short: *scope*) of a variable is the region of a program where it can be accessed. JavaScript's scopes are *static* (they don't change at runtime) and they can be nested – for example:

```
function func() { // (A)
  const aVariable = 1;
  if (true) { // (B)
    const anotherVariable = 2;
```

```
  }
}
```

The scope introduced by the `if` statement (line B) is nested inside the scope of function `func()` (line A).

The innermost surrounding scope of a scope S is called the *outer scope* of S. In the example, `func` is the outer scope of `if`.

4.2 Lexical environments

In the JavaScript language specification, scopes are "implemented" via *lexical environments*. They consist of two components:

- An *environment record* that maps variable names to variable values (think dictionary). This is the actual storage space for the variables of the scope. The name-value entries in the record are called *bindings*.

- A reference to the *outer environment* – the environment for the outer scope.

The tree of nested scopes is therefore represented by a tree of environments linked by outer environment references.

4.3 The global object

The global object is an object whose properties become global variables. (We'll examine soon how exactly it fits into the tree of environments.) It can be accessed via the following global variables:

- Available on all platforms: `globalThis`. The name is based on the fact that it has the same value as `this` in global scope.
- Other variables for the global object are not available on all platforms:
  - `window` is the classic way of referring to the global object. It works in normal browser code, but not in *Web Workers* (processes running concurrently to the normal browser process) and not on Node.js.
  - `self` is available everywhere in browsers (including in Web Workers). But it isn't supported by Node.js.
  - `global` is only available on Node.js.

4.4 In browsers, `globalThis` does not point directly to the global object

In browsers, `globalThis` does not point directly to the global, there is an indirection. As an example, consider an iframe on a web page:

- Whenever the `src` of the iframe changes, it gets a new global object.
- However, `globalThis` always has the same value. That value can be checked from outside the iframe, as demonstrated below (inspired by an example in the `globalThis` proposal).

File `parent.html`:

```
<iframe src="iframe.html?first"></iframe>
<script>
  const iframe = document.querySelector('iframe');
  const icw = iframe.contentWindow; // `globalThis` of iframe

  iframe.onload = () => {
    // Access properties of global object of iframe
    const firstGlobalThis = icw.globalThis;
    const firstArray = icw.Array;
    console.log(icw.iframeName); // 'first'

    iframe.onload = () => {
      const secondGlobalThis = icw.globalThis;
      const secondArray = icw.Array;

      // The global object is different
      console.log(icw.iframeName); // 'second'
      console.log(secondArray === firstArray); // false

      // But globalThis is still the same
      console.log(firstGlobalThis === secondGlobalThis); // true
    };
    iframe.src = 'iframe.html?second';
  };
</script>
```

File `iframe.html`:

```
<script>
  globalThis.iframeName = location.search.slice(1);
</script>
```

How do browsers ensure that `globalThis` doesn't change in this scenario? They internally distinguish two objects:

- `Window` is the global object. It changes whenever the location changes.
- `WindowProxy` is an object that forwards all accesses to the current `Window`. This object never changes.

In browsers, `globalThis` refers to the `WindowProxy`; everywhere else, it directly refers to the global object.

4.5 The global environment

The global scope is the "outermost" scope – it has no outer scope. Its environment is the *global environment*. Every environment is connected with the global environment via a chain of environments that are linked by outer environment references. The outer environment reference of the global environment is `null`.

The global environment record uses two environment records to manage its variables:

- An *object environment record* has the same interface as a normal environment record, but keeps its bindings in a JavaScript object. In this case, the object is the global object.
- A normal (*declarative*) environment record that has its own storage for its bindings.

Which of these two records is used when will be explained soon.

4.5.1 Script scope and module scopes

In JavaScript, we are only in global scope at the top levels of scripts. In contrast, each module has its own scope that is a subscope of the script scope.

If we ignore the relatively complicated rules for how variable bindings are added to the global environment, then global scope and module scopes work as if they were nested code blocks:

```
{ // Global scope (scope of *all* scripts)

  // (Global variables)

  { // Scope of module 1
    ...
  }
  { // Scope of module 2
    ...
  }
  // (More module scopes)
}
```

4.5.2 Creating variables: declarative record vs. object record

In order to create a variable that is truly global, we must be in global scope – which is only the case at the top level of scripts:

- Top-level `const`, `let`, and `class` create bindings in the declarative environment record.
- Top-level `var` and function declarations create bindings in the object environment record.

```
<script>
  const one = 1;
  var two = 2;
</script>
<script>
  // All scripts share the same top-level scope:
  console.log(one); // 1
  console.log(two); // 2

  // Not all declarations create properties of the global object:
```

```
  console.log(globalThis.one); // undefined
  console.log(globalThis.two); // 2
</script>
```

4.5.3 Getting or setting variables

When we get or set a variable and both environment records have a binding for that variable, then the declarative record wins:

```
<script>
  let myGlobalVariable = 1; // declarative environment record
  globalThis.myGlobalVariable = 2; // object environment record

  console.log(myGlobalVariable); // 1 (declarative record wins)
  console.log(globalThis.myGlobalVariable); // 2
</script>
```

4.5.4 Global ECMAScript variables and global host variables

In addition to variables created via `var` and function declarations, the global object contains the following properties:

- All built-in global variables of ECMAScript
- All built-in global variables of the host platform (browser, Node.js, etc.)

Using `const` or `let` guarantees that global variable declarations aren't influencing (or influenced by) the built-in global variables of ECMAScript and host platform.

For example, browsers have the global variable `.location`:

```
// Changes the location of the current document:
var location = 'https://example.com';

// Shadows window.location, doesn't change it:
let location = 'https://example.com';
```

If a variable already exists (such as `location` in this case), then a `var` declaration with an initializer behaves like an assignment. That's why we get into trouble in this example.

Note that this is only an issue in global scope. In modules, we are never in global scope (unless we use `eval()` or similar).

Fig. 4.1 summarizes everything we have learned in this section.

4.6 Conclusion: Why does JavaScript have both normal global variables and the global object?

The global object is generally considered to be a mistake. For that reason, newer constructs such as `const`, `let`, and classes create normal global variables (when in script scope).

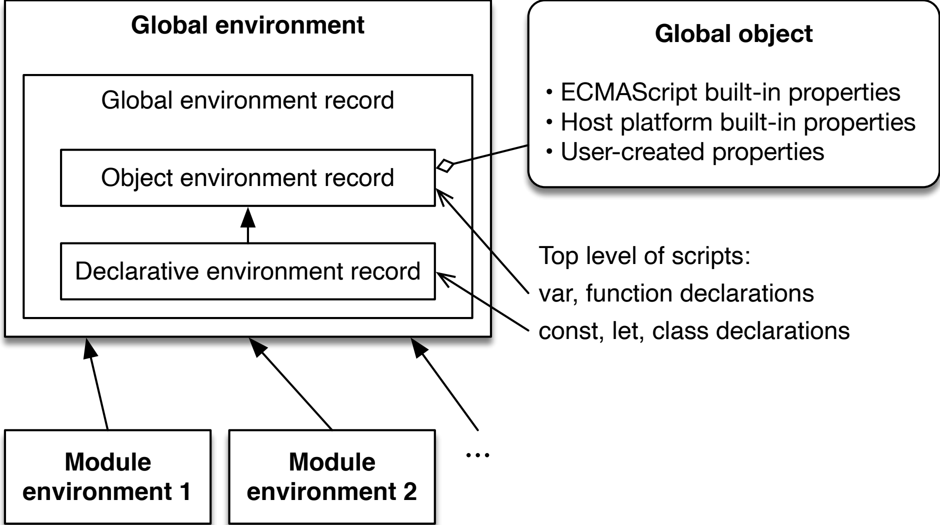


Figure 4.1: The environment for the global scope manages its bindings via a *global environment record* which in turn is based on two environment records: an *object environment record* whose bindings are stored in the global object and a *declarative environment record* that uses internal storage for its bindings. Therefore, global variables can be created by adding properties to the global object or via various declarations. The global object is initialized with the built-in global variables of ECMAScript and the host platform. Each ECMAScript module has its own environment whose outer environment is the global environment.

Thankfully, most of the code written in modern JavaScript, lives in ECMAScript modules and CommonJS modules. Each module has its own scope, which is why the rules governing global variables rarely matter for module-based code.

4.7 Further reading and sources of this chapter

Environments and the global object in the ECMAScript specification:

- Section "Lexical Environments" provides a general overview over environments.
- Section "Global Environment Records" covers the global environment.
- Section "ECMAScript Standard Built-in Objects" describes how ECMAScript manages its built-in objects (which include the global object).

`globalThis`:

- 2ality post "ES feature: `globalThis`"
- Various ways of accessing the global `this` value: "A horrifying `globalThis` polyfill in universal JavaScript" by Mathias Bynens

The global object in browsers:

- Background on what happens in browsers: "Defining the WindowProxy, Window, and Location objects" by Anne van Kesteren
- Very technical: section "Realms, settings objects, and global objects" in the WHATWG HTML standard
- In the ECMAScript specification, we can see how web browsers customize global `this`: section "InitializeHostDefinedRealm()"

Chapter 5

`%` is a remainder operator, not a modulo operator

Contents

5.1 Remainder operator `rem` vs. modulo operator `mod` 47
5.2 An intuitive understanding of the remainder operation 48
5.3 An intuitive understanding of the modulo operation 48
5.4 Similarities and differences between `rem` and `mod` 49
5.5 The equations behind remainder and modulo 49
5.5.1 `rem` and `mod` perform integer division differently 50
5.5.2 Implementing `rem` 50
5.5.3 Implementing `mod` 50
5.6 Where are `rem` and `mod` used in programming languages? 51
5.6.1 JavaScript's `%` operator computes the remainder 51
5.6.2 Python's `%` operator computes the modulus 51
5.6.3 Uses of the modulo operation in JavaScript 51
5.7 Further reading and sources of this chapter 52

Remainder and *modulo* are two similar operations. This chapter explores how they work and reveals that JavaScript's `%` operator computes the remainder, not the modulus.

5.1 Remainder operator `rem` vs. modulo operator `mod`

In this chapter, we pretend that JavaScript has the following two operators:

- The *remainder operator* `rem`
- The *modulo operator* `mod`

That will help us examine how the underlying operations work.

5.2 An intuitive understanding of the remainder operation

The operands of the remainder operator are called *dividend* and *divisor*:

```
remainder = dividend rem divisor
```

How is the result computed? Consider the following expression:

```
7 rem 3
```

We remove 3 from the dividend until we are left with a value that is smaller than 3:

```
7 rem 3 = 4 rem 3 = 1 rem 3 = 1
```

What do we do if the dividend is negative?

```
-7 rem 3
```

This time, we add 3 to the dividend until we have a value that is smaller than -3:

```
-7 rem 3 = -4 rem 3 = -1 rem 3 = -1
```

It is insightful to map out the results for a fixed divisor:

```
x:       -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7
x rem 3: -1  0 -2 -1  0 -2 -1  0  1  2  0  1  2  0  1
```

Among the results, we can see a symmetry: `x` and `-x` produce the same results, but with opposite signs.

5.3 An intuitive understanding of the modulo operation

Once again, the operands are called *dividend* and *divisor* (hinting at how similar `rem` and `mod` are):

```
modulus = dividend mod divisor
```

Consider the following example:

```
x mod 3
```

This operation maps `x` into the range:

```
[0,3) = {0,1,2}
```

That is, zero is included (opening square bracket), 3 is excluded (closing parenthesis).

How does `mod` perform this mapping? It is easy if `x` is already inside the range:

```
> 0 mod 3
0
> 2 mod 3
2
```

If `x` is greater than or equal to the upper boundary of the range, then the upper boundary is subtracted from `x` until it fits into the range:

```
> 4 mod 3
1
> 7 mod 3
1
```

That means we are getting the following mapping for non-negative integers:

```
x:        0 1 2 3 4 5 6 7
x mod 3:  0 1 2 0 1 2 0 1
```

This mapping is extended as follows, so that it includes negative integers:

```
x:        -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7
x mod 3:   2  0  1  2  0  1  2  0  1  2  0  1  2  0  1
```

Note how the range [0,3) is repeated over and over again.

5.4 Similarities and differences between `rem` and `mod`

`rem` and `mod` are quite similar – they only differ if dividend and divisor have different signs:

- With `rem`, the result has the same sign as the dividend (first operand):

  ```
  > 5 rem 4
  1
  > -5 rem 4
  -1
  > 5 rem -4
  1
  > -5 rem -4
  -1
  ```

- With `mod`, the result has the same sign as the divisor (second operand):

  ```
  > 5 mod 4
  1
  > -5 mod 4
  3
  > 5 mod -4
  -3
  > -5 mod -4
  -1
  ```

5.5 The equations behind remainder and modulo

The following two equations define both remainder and modulo:

```
dividend = (divisor * quotient) + remainder
|remainder| < |divisor|
```

Given a dividend and a divisor, how do we compute remainder and modulus?

```
remainder = dividend - (divisor * quotient)
quotient = dividend div divisor
```

The `div` operator performs integer division.

5.5.1 `rem` and `mod` perform integer division differently

How can these equations contain the solution for *both* operations? If variable `remainder` isn't zero, the equations always allow two values for it: one positive, the other one negative. To see why, we turn to the question we are asking when determining the quotient:

> How often do we need to multiply the divisor to get as close to the dividend as possible?

For example, that gives us two different results for dividend -2 and divisor 3:

- `-2 rem 3 = -2`
  3×0 gets us close to -2. The difference between 0 and the dividend -2 is -2.
- `-2 mod 3 = 1`
  3×-1 also gets us close to -2. The difference between -3 and the dividend -2 is 1.

It also gives us two different results for dividend 2 and divisor -3:

- `2 rem -3 = 2`
  -3 × 0 gets us close to 2.
- `2 mod -3 = -1`
  -3 × -1 gets us close to 2.

The results differ depending on how the integer division `div` is implemented.

5.5.2 Implementing `rem`

The following JavaScript function implements the `rem` operator. It performs integer division by performing floating point division and rounding the result to an integer via `Math.trunc()`:

```
function rem(dividend, divisor) {
  const quotient = Math.trunc(dividend / divisor);
  return dividend - (divisor * quotient);
}
```

5.5.3 Implementing `mod`

The following function implements the `mod` operator. This time, integer division is based on `Math.floor()`:

```
function mod(dividend, divisor) {
  const quotient = Math.floor(dividend / divisor);
  return dividend - (divisor * quotient);
}
```

Note that other ways of doing integer division are possible (e.g. based on `Math.ceil()` or `Math.round()`). That means that there are more operations that are similar to `rem` and `mod`.

5.6 Where are rem and mod used in programming languages?

5.6.1 JavaScript's % operator computes the remainder

For example (Node.js REPL):

```
> 7 % 6
1
> -7 % 6
-1
```

5.6.2 Python's % operator computes the modulus

For example (Python REPL):

```
>>> 7 % 6
1
>>> -7 % 6
5
```

5.6.3 Uses of the modulo operation in JavaScript

The ECMAScript specification uses modulo several times – for example:

- To convert the operands of the >>> operator to unsigned 32-bit integers (`x mod 2**32`):

  ```
  > 2**32 >>> 0
  0
  > (2**32)+1 >>> 0
  1
  > (-1 >>> 0) === (2**32)-1
  true
  ```

- To convert arbitrary numbers so that they fit into Typed Arrays. For example, `x mod 2**8` is used to convert numbers to unsigned 8-bit integers (after first converting them to integers):

  ```
  const tarr = new Uint8Array(1);

  tarr[0] = 256;
  assert.equal(tarr[0], 0);

  tarr[0] = 257;
  assert.equal(tarr[0], 1);
  ```

5.7 Further reading and sources of this chapter

- The Wikipedia page on "modulo operation" has much information.
- The ECMAScript specification mentions that `%` is computed via "truncating division".
- Sect. "Rounding" in "JavaScript for impatient programmers" describes several ways of rounding in JavaScript.

Part III

Working with data

Chapter 6

Copying objects and Arrays

Contents

6.1 Shallow copying vs. deep copying **55**

6.2 Shallow copying in JavaScript **56**

6.2.1 Copying plain objects and Arrays via spreading 56

6.2.2 Shallow copying via `Object.assign()` (optional) 59

6.2.3 Shallow copying via `Object.getOwnPropertyDescriptors()` and `Object.defineProperties()` (optional) 59

6.3 Deep copying in JavaScript **60**

6.3.1 Manual deep copying via nested spreading 60

6.3.2 Hack: generic deep copying via JSON 60

6.3.3 Implementing generic deep copying 61

6.4 Further reading **63**

In this chapter, we will learn how to copy objects and Arrays in JavaScript.

6.1 Shallow copying vs. deep copying

There are two "depths" with which data can be copied:

- *Shallow copying* only copies the top-level entries of objects and Arrays. The entry values are still the same in original and copy.
- *Deep copying* also copies the entries of the values of the entries, etc. That is, it traverses the complete tree whose root is the value to be copied and makes copies of all nodes.

The next sections cover both kinds of copying. Unfortunately, JavaScript only has built-in support for shallow copying. If we need deep copying, we need to implement it ourselves.

6.2 Shallow copying in JavaScript

Let's look at several ways of shallowly copying data.

6.2.1 Copying plain objects and Arrays via spreading

We can spread into object literals and into Array literals to make copies:

```
const copyOfObject = {...originalObject};
const copyOfArray = [...originalArray];
```

Alas, spreading has several issues. Those will be covered in the next subsections. Among those, some are real limitations, others mere pecularities.

6.2.1.1 The prototype is not copied

For example:

```
class MyClass {}

const original = new MyClass();
assert.equal(original instanceof MyClass, true);

const copy = {...original};
assert.equal(copy instanceof MyClass, false);
```

Note that the following two expressions are equivalent:

```
obj instanceof SomeClass
SomeClass.prototype.isPrototypeOf(obj)
```

Therefore, we can fix this by giving the copy the same prototype as the original:

```
class MyClass {}

const original = new MyClass();

const copy = {
  __proto__: Object.getPrototypeOf(original),
  ...original,
};
assert.equal(copy instanceof MyClass, true);
```

Alternatively, we can set the prototype of the copy after its creation, via `Object.setPrototypeOf()`.

6.2.1.2 Many built-in objects have special "internal slots" that aren't copied

Examples of such built-in objects include regular expressions and dates. If we make a copy of them, we lose most of the data stored in them.

6.2.1.3 Only own (non-inherited) properties are copied

Given how prototype chains work, this is usually the right approach. But we still need to be aware of it. In the following example, the inherited property `.inheritedProp` of `original` is not available in `copy` because we only copy own properties and don't keep the prototype.

```
const proto = { inheritedProp: 'a' };
const original = {__proto__: proto, ownProp: 'b' };
assert.equal(original.inheritedProp, 'a');
assert.equal(original.ownProp, 'b');

const copy = {...original};
assert.equal(copy.inheritedProp, undefined);
assert.equal(copy.ownProp, 'b');
```

6.2.1.4 Only enumerable properties are copied

For example, the own property `.length` of Array instances is not enumerable and not copied:

```
const arr = ['a', 'b'];
assert.equal(arr.length, 2);
assert.equal({}.hasOwnProperty.call(arr, 'length'), true);

const copy = {...arr};
assert.equal({}.hasOwnProperty.call(copy, 'length'), false);
```

This is also rarely a limitation because most properties are enumerable. If we need to copy non-enumerable properties, we can use `Object.getOwnPropertyDescriptors()` and `Object.defineProperties()` to copy objects (how to do that is explained later):

- They consider all attributes (not just `value`) and therefore correctly copy getters, setters, read-only properties, etc.
- `Object.getOwnPropertyDescriptors()` retrieves both enumerable and non-enumerable properties.

For more information on enumerability, see [content not included].

6.2.1.5 Property attributes aren't always copied faithfully

Independently of the *attributes* of a property, its copy will always be a data property that is writable and configurable.

For example, here we create the property `original.prop` whose attributes `writable` and `configurable` are `false`:

```
const original = Object.defineProperties(
  {}, {
    prop: {
      value: 1,
      writable: false,
```

```
      configurable: false,
      enumerable: true,
    },
  });
assert.deepEqual(original, {prop: 1});
```

If we copy .prop, then writable and configurable are both true:

```
const copy = {...original};
// Attributes `writable` and `configurable` of copy are different:
assert.deepEqual(
  Object.getOwnPropertyDescriptors(copy),
  {
    prop: {
      value: 1,
      writable: true,
      configurable: true,
      enumerable: true,
    },
  });
```

As a consequence, getters and setters are not copied faithfully, either:

```
const original = {
  get myGetter() { return 123 },
  set mySetter(x) {},
};
assert.deepEqual({...original}, {
  myGetter: 123, // not a getter anymore!
  mySetter: undefined,
});
```

The aforementioned `Object.getOwnPropertyDescriptors()` and `Object.defineProperties()` always transfer own properties with all attributes intact (as shown later).

6.2.1.6 Copying is shallow

The copy has fresh versions of each key-value entry in the original, but the values of the original are not copied themselves. For example:

```
const original = {name: 'Jane', work: {employer: 'Acme'}};
const copy = {...original};

// Property .name is a copy: changing the copy
// does not affect the original
copy.name = 'John';
assert.deepEqual(original,
  {name: 'Jane', work: {employer: 'Acme'}});
assert.deepEqual(copy,
  {name: 'John', work: {employer: 'Acme'}});
```

```
// The value of .work is shared: changing the copy
// affects the original
copy.work.employer = 'Spectre';
assert.deepEqual(
  original, {name: 'Jane', work: {employer: 'Spectre'}});
assert.deepEqual(
  copy, {name: 'John', work: {employer: 'Spectre'}});
```

We'll look at deep copying later in this chapter.

6.2.2 Shallow copying via `Object.assign()` (optional)

`Object.assign()` works mostly like spreading into objects. That is, the following two ways of copying are mostly equivalent:

```
const copy1 = {...original};
const copy2 = Object.assign({}, original);
```

Using a method instead of syntax has the benefit that it can be polyfilled on older JavaScript engines via a library.

`Object.assign()` is not completely like spreading, though. It differs in one, relatively subtle point: it creates properties differently.

- `Object.assign()` uses *assignment* to create the properties of the copy.
- Spreading *defines* new properties in the copy.

Among other things, assignment invokes own and inherited setters, while definition doesn't (more information on assignment vs. definition). This difference is rarely noticeable. The following code is an example, but it's contrived:

```
const original = {['__proto__']: null}; // (A)
const copy1 = {...original};
// copy1 has the own property '__proto__'
assert.deepEqual(
  Object.keys(copy1), ['__proto__']);

const copy2 = Object.assign({}, original);
// copy2 has the prototype null
assert.equal(Object.getPrototypeOf(copy2), null);
```

By using a computed property key in line A, we create `.__proto__` as an own property and don't invoke the inherited setter. However, when `Object.assign()` copies that property, it does invoke the setter. (For more information on `.__proto__`, see "JavaScript for impatient programmers".)

6.2.3 Shallow copying via `Object.getOwnPropertyDescriptors()` and `Object.defineProperties()` (optional)

JavaScript lets us create properties via *property descriptors*, objects that specify property attributes. For example, via the `Object.defineProperties()`, which we have already

seen in action. If we combine that method with `Object.getOwnPropertyDescriptors()`, we can copy more faithfully:

```
function copyAllOwnProperties(original) {
  return Object.defineProperties(
    {}, Object.getOwnPropertyDescriptors(original));
}
```

That eliminates two issues of copying objects via spreading.

First, all attributes of own properties are copied correctly. Therefore, we can now copy own getters and own setters:

```
const original = {
  get myGetter() { return 123 },
  set mySetter(x) {},
};
assert.deepEqual(copyAllOwnProperties(original), original);
```

Second, thanks to `Object.getOwnPropertyDescriptors()`, non-enumerable properties are copied, too:

```
const arr = ['a', 'b'];
assert.equal(arr.length, 2);
assert.equal({}.hasOwnProperty.call(arr, 'length'), true);

const copy = copyAllOwnProperties(arr);
assert.equal({}.hasOwnProperty.call(copy, 'length'), true);
```

6.3 Deep copying in JavaScript

Now it is time to tackle deep copying. First, we will deep-copy manually, then we'll examine generic approaches.

6.3.1 Manual deep copying via nested spreading

If we nest spreading, we get deep copies:

```
const original = {name: 'Jane', work: {employer: 'Acme'}};
const copy = {name: original.name, work: {...original.work}};

// We copied successfully:
assert.deepEqual(original, copy);
// The copy is deep:
assert.ok(original.work !== copy.work);
```

6.3.2 Hack: generic deep copying via JSON

This is a hack, but, in a pinch, it provides a quick solution: In order to deep-copy an object `original`, we first convert it to a JSON string and parse that JSON string:

```
function jsonDeepCopy(original) {
  return JSON.parse(JSON.stringify(original));
}
const original = {name: 'Jane', work: {employer: 'Acme'}};
const copy = jsonDeepCopy(original);
assert.deepEqual(original, copy);
```

The significant downside of this approach is that we can only copy properties with keys and values that are supported by JSON.

Some unsupported keys and values are simply ignored:

```
assert.deepEqual(
  jsonDeepCopy({
    // Symbols are not supported as keys
    [Symbol('a')]: 'abc',
    // Unsupported value
    b: function () {},
    // Unsupported value
    c: undefined,
  }),
  {} // empty object
);
```

Others cause exceptions:

```
assert.throws(
  () => jsonDeepCopy({a: 123n}),
  /^TypeError: Do not know how to serialize a BigInt$/);
```

6.3.3 Implementing generic deep copying

The following function generically deep-copies a value `original`:

```
function deepCopy(original) {
  if (Array.isArray(original)) {
    const copy = [];
    for (const [index, value] of original.entries()) {
      copy[index] = deepCopy(value);
    }
    return copy;
  } else if (typeof original === 'object' && original !== null) {
    const copy = {};
    for (const [key, value] of Object.entries(original)) {
      copy[key] = deepCopy(value);
    }
    return copy;
  } else {
    // Primitive value: atomic, no need to copy
    return original;
```

```
  }
}
```

The function handles three cases:

- If `original` is an Array we create a new Array and deep-copy the elements of `original` into it.
- If `original` is an object, we use a similar approach.
- If `original` is a primitive value, we don't have to do anything.

Let's try out `deepCopy()`:

```
const original = {a: 1, b: {c: 2, d: {e: 3}}};
const copy = deepCopy(original);

// Are copy and original deeply equal?
assert.deepEqual(copy, original);

// Did we really copy all levels
// (equal content, but different objects)?
assert.ok(copy     !== original);
assert.ok(copy.b   !== original.b);
assert.ok(copy.b.d !== original.b.d);
```

Note that `deepCopy()` only fixes one issue of spreading: shallow copying. All others remain: prototypes are not copied, special objects are only partially copied, non-enumerable properties are ignored, most property attributes are ignored.

Implementing copying completely generically is generally impossible: Not all data is a tree, sometimes we don't want to copy all properties, etc.

6.3.3.1 A more concise version of `deepCopy()`

We can make our previous implementation of `deepCopy()` more concise if we use `.map()` and `Object.fromEntries()`:

```
function deepCopy(original) {
  if (Array.isArray(original)) {
    return original.map(elem => deepCopy(elem));
  } else if (typeof original === 'object' && original !== null) {
    return Object.fromEntries(
      Object.entries(original)
        .map(([k, v]) => [k, deepCopy(v)]));
  } else {
    // Primitive value: atomic, no need to copy
    return original;
  }
}
```

6.4 Further reading

- [Content not included] explains class-based patterns for copying.
- Section "Spreading into object literals" in "JavaScript for impatient programmers"
- Section "Spreading into Array literals" in "JavaScript for impatient programmers"
- Section "Prototype chains" in "JavaScript for impatient programmers"

Chapter 7

Updating data destructively and non-destructively

Contents

7.1 Examples: updating an object destructively and non-destructively . 65
7.2 Examples: updating an Array destructively and non-destructively . 66
7.3 Manual deep updating 67
7.4 Implementing generic deep updating 67

In this chapter, we learn about two different ways of updating data:

- A *destructive update* of data mutates the data so that it has the desired form.
- A *non-destructive update* of data creates a copy of the data that has the desired form.

The latter way is similar to first making a copy and then changing it destructively, but it does both at the same time.

7.1 Examples: updating an object destructively and non-destructively

The following code shows a function that updates object properties destructively and uses it on an object.

```
function setPropertyDestructively(obj, key, value) {
  obj[key] = value;
  return obj;
}

const obj = {city: 'Berlin', country: 'Germany'};
setPropertyDestructively(obj, 'city', 'Munich');
assert.deepEqual(obj, {city: 'Munich', country: 'Germany'});
```

The following code demonstrates non-destructive updating of an object:

```
function setPropertyNonDestructively(obj, key, value) {
  const updatedObj = {};
  for (const [k, v] of Object.entries(obj)) {
    updatedObj[k] = (k === key ? value : v);
  }
  return updatedObj;
}

const obj = {city: 'Berlin', country: 'Germany'};
const updatedObj = setPropertyNonDestructively(obj, 'city', 'Munich');

// We have created an updated object:
assert.deepEqual(updatedObj, {city: 'Munich', country: 'Germany'});

// But we didn’t change the original:
assert.deepEqual(obj, {city: 'Berlin', country: 'Germany'});
```

Spreading makes `setPropertyNonDestructively()` more concise:

```
function setPropertyNonDestructively(obj, key, value) {
  return {...obj, [key]: value};
}
```

Both versions of `setPropertyNonDestructively()` update *shallowly*: They only change the top level of an object.

7.2 Examples: updating an Array destructively and non-destructively

The following code shows a function that updates Array elements destructively and uses it on an Array.

```
function setElementDestructively(arr, index, value) {
  arr[index] = value;
}

const arr = ['a', 'b', 'c', 'd', 'e'];
setElementDestructively(arr, 2, 'x');
assert.deepEqual(arr, ['a', 'b', 'x', 'd', 'e']);
```

The following code demonstrates non-destructive updating of an Array:

```
function setElementNonDestructively(arr, index, value) {
  const updatedArr = [];
  for (const [i, v] of arr.entries()) {
    updatedArr.push(i === index ? value : v);
  }
  return updatedArr;
```

```
}

const arr = ['a', 'b', 'c', 'd', 'e'];
const updatedArr = setElementNonDestructively(arr, 2, 'x');
assert.deepEqual(updatedArr, ['a', 'b', 'x', 'd', 'e']);
assert.deepEqual(arr, ['a', 'b', 'c', 'd', 'e']);
```

`.slice()` and spreading make `setElementNonDestructively()` more concise:

```
function setElementNonDestructively(arr, index, value) {
  return [
    ...arr.slice(0, index), value, ...arr.slice(index+1)];
}
```

Both versions of `setElementNonDestructively()` update *shallowly*: They only change the top level of an Array.

7.3 Manual deep updating

So far, we have only updated data shallowly. Let's tackle deep updating. The following code shows how to do it manually. We are changing name and employer.

```
const original = {name: 'Jane', work: {employer: 'Acme'}};
const updatedOriginal = {
  ...original,
  name: 'John',
  work: {
    ...original.work,
    employer: 'Spectre'
  },
};

assert.deepEqual(
  original, {name: 'Jane', work: {employer: 'Acme'}});
assert.deepEqual(
  updatedOriginal, {name: 'John', work: {employer: 'Spectre'}});
```

7.4 Implementing generic deep updating

The following function implements generic deep updating.

```
function deepUpdate(original, keys, value) {
  if (keys.length === 0) {
    return value;
  }
  const currentKey = keys[0];
  if (Array.isArray(original)) {
    return original.map(
      (v, index) => index === currentKey
```

```
        ? deepUpdate(v, keys.slice(1), value) // (A)
        : v); // (B)
  } else if (typeof original === 'object' && original !== null) {
    return Object.fromEntries(
      Object.entries(original).map(
        (keyValuePair) => {
          const [k,v] = keyValuePair;
          if (k === currentKey) {
            return [k, deepUpdate(v, keys.slice(1), value)]; // (C)
          } else {
            return keyValuePair; // (D)
          }
        }));
  } else {
    // Primitive value
    return original;
  }
}
```

If we see `value` as the root of a tree that we are updating, then `deepUpdate()` only deeply changes a single branch (line A and C). All other branches are copied shallowly (line B and D).

This is what using `deepUpdate()` looks like:

```
const original = {name: 'Jane', work: {employer: 'Acme'}};

const copy = deepUpdate(original, ['work', 'employer'], 'Spectre');
assert.deepEqual(copy, {name: 'Jane', work: {employer: 'Spectre'}});
assert.deepEqual(original, {name: 'Jane', work: {employer: 'Acme'}});
```

Chapter 8

The problems of shared mutable state and how to avoid them

Contents

- 8.1 **What is shared mutable state and why is it problematic?** **69**
- 8.2 **Avoiding sharing by copying data** **71**
  - 8.2.1 How does copying help with shared mutable state? 71
- 8.3 **Avoiding mutations by updating non-destructively** **73**
  - 8.3.1 How does non-destructive updating help with shared mutable state? 73
- 8.4 **Preventing mutations by making data immutable** **74**
  - 8.4.1 How does immutability help with shared mutable state? . . . 74
- 8.5 **Libraries for avoiding shared mutable state** **74**
  - 8.5.1 Immutable.js 74
  - 8.5.2 Immer 75

This chapter answers the following questions:

- What is shared mutable state?
- Why is it problematic?
- How can its problems be avoided?

8.1 What is shared mutable state and why is it problematic?

Shared mutable state works as follows:

- If two or more parties can change the same data (variables, objects, etc.).
- And if their lifetimes overlap.
- Then there is a risk of one party's modifications preventing other parties from working correctly.

Note that this definition applies to function calls, cooperative multitasking (e.g., async functions in JavaScript), etc. The risks are similar in each case.

The following code is an example. The example is not realistic, but it demonstrates the risks and is easy to understand:

```
function logElements(arr) {
  while (arr.length > 0) {
    console.log(arr.shift());
  }
}

function main() {
  const arr = ['banana', 'orange', 'apple'];

  console.log('Before sorting:');
  logElements(arr);

  arr.sort(); // changes arr

  console.log('After sorting:');
  logElements(arr); // (A)
}
main();

// Output:
// 'Before sorting:'
// 'banana'
// 'orange'
// 'apple'
// 'After sorting:'
```

In this case, there are two independent parties:

- Function `main()` wants to log an Array before and after sorting it.
- Function `logElements()` logs the elements of its parameter `arr`, but removes them while doing so.

`logElements()` breaks `main()` and causes it to log an empty Array in line A.

In the remainder of this chapter, we look at three ways of avoiding the problems of shared mutable state:

- Avoiding sharing by copying data
- Avoiding mutations by updating non-destructively
- Preventing mutations by making data immutable

In particular, we will come back to the example that we've just seen and fix it.

8.2 Avoiding sharing by copying data

Copying data is one way of avoiding sharing it.

Background

For background on copying data in JavaScript, please refer to the following two chapters in this book:

- §6 "Copying objects and Arrays"
- [Content not included]

8.2.1 How does copying help with shared mutable state?

As long as we only *read* from shared state, we don't have any problems. Before *modifying* it, we need to "un-share" it, by copying it (as deeply as necessary).

Defensive copying is a technique to always copy when issues *might* arise. Its objective is to keep the current entity (function, class, etc.) safe:

- Input: Copying (potentially) shared data passed to us, lets us use that data without being disturbed by an external entity.
- Output: Copying internal data before exposing it to an outside party, means that that party can't disrupt our internal activity.

Note that these measures protect us from other parties, but they also protect other parties from us.

The next sections illustrate both kinds of defensive copying.

8.2.1.1 Copying shared input

Remember that in the motivating example at the beginning of this chapter, we got into trouble because `logElements()` modified its parameter `arr`:

```
function logElements(arr) {
  while (arr.length > 0) {
    console.log(arr.shift());
  }
}
```

Let's add defensive copying to this function:

```
function logElements(arr) {
  arr = [...arr]; // defensive copy
  while (arr.length > 0) {
    console.log(arr.shift());
  }
}
```

Now `logElements()` doesn't cause problems anymore, if it is called inside `main()`:

```
function main() {
  const arr = ['banana', 'orange', 'apple'];

  console.log('Before sorting:');
  logElements(arr);

  arr.sort(); // changes arr

  console.log('After sorting:');
  logElements(arr); // (A)
}
main();

// Output:
// 'Before sorting:'
// 'banana'
// 'orange'
// 'apple'
// 'After sorting:'
// 'apple'
// 'banana'
// 'orange'
```

8.2.1.2 Copying exposed internal data

Let's start with a class `StringBuilder` that doesn't copy internal data it exposes (line A):

```
class StringBuilder {
  _data = [];
  add(str) {
    this._data.push(str);
  }
  getParts() {
    // We expose internals without copying them:
    return this._data; // (A)
  }
  toString() {
    return this._data.join('');
  }
}
```

As long as `.getParts()` isn't used, everything works well:

```
const sb1 = new StringBuilder();
sb1.add('Hello');
sb1.add(' world!');
assert.equal(sb1.toString(), 'Hello world!');
```

If, however, the result of `.getParts()` is changed (line A), then the `StringBuilder` ceases to work correctly:

```
const sb2 = new StringBuilder();
sb2.add('Hello');
sb2.add(' world!');
sb2.getParts().length = 0; // (A)
assert.equal(sb2.toString(), ''); // not OK
```

The solution is to copy the internal `._data` defensively before it is exposed (line A):

```
class StringBuilder {
  this._data = [];
  add(str) {
    this._data.push(str);
  }
  getParts() {
    // Copy defensively
    return [...this._data]; // (A)
  }
  toString() {
    return this._data.join('');
  }
}
```

Now changing the result of `.getParts()` doesn't interfere with the operation of `sb` anymore:

```
const sb = new StringBuilder();
sb.add('Hello');
sb.add(' world!');
sb.getParts().length = 0;
assert.equal(sb.toString(), 'Hello world!'); // OK
```

8.3 Avoiding mutations by updating non-destructively

We can avoid mutations if we only update data non-destructively.

Background

For more information on updating data, see §7 "Updating data destructively and non-destructively".

8.3.1 How does non-destructive updating help with shared mutable state?

With non-destructive updating, sharing data becomes unproblematic, because we never mutate the shared data. (This only works if everyone that accesses the data does that!)

Intriguingly, copying data becomes trivially simple:

```
const original = {city: 'Berlin', country: 'Germany'};
```

```
const copy = original;
```

This works, because we are only making non-destructive changes and are therefore copying the data on demand.

8.4 Preventing mutations by making data immutable

We can prevent mutations of shared data by making that data immutable.

> **Background**
>
> For background on how to make data immutable in JavaScript, please refer to the following two chapters in this book:
>
> - [Content not included]
> - [Content not included]

8.4.1 How does immutability help with shared mutable state?

If data is immutable, it can be shared without any risks. In particular, there is no need to copy defensively.

> **Non-destructive updating is an important complement to immutable data**
>
> If we combine the two, immutable data becomes virtually as versatile as mutable data but without the associated risks.

8.5 Libraries for avoiding shared mutable state

There are several libraries available for JavaScript that support immutable data with non-destructive updating. Two popular ones are:

- Immutable.js provides immutable data structures for lists, stacks, sets, maps, etc.
- Immer also supports immutability and non-destructive updating but for plain objects, Arrays, Sets, and Maps. That is, no new data structures are needed.

These libraries are described in more detail in the next two sections.

8.5.1 Immutable.js

In its repository, the library Immutable.js is described as:

> Immutable persistent data collections for JavaScript which increase efficiency and simplicity.

Immutable.js provides immutable data structures such as:

- `List`
- `Stack`

- `Set` (which is different from JavaScript's built-in `Set`)
- `Map` (which is different from JavaScript's built-in `Map`)
- Etc.

In the following example, we use an immutable `Map`:

```
import {Map} from 'immutable/dist/immutable.es.js';
const map0 = Map([
  [false, 'no'],
  [true, 'yes'],
]);

// We create a modified version of map0:
const map1 = map0.set(true, 'maybe');

// The modified version is different from the original:
assert.ok(map1 !== map0);
assert.equal(map1.equals(map0), false); // (A)

// We undo the change we just made:
const map2 = map1.set(true, 'yes');

// map2 is a different object than map0,
// but it has the same content
assert.ok(map2 !== map0);
assert.equal(map2.equals(map0), true); // (B)
```

Notes:

- Instead of modifying the receiver, "destructive" operations such as `.set()` return modified copies.
- To check if two data structures have the same content, we use the built-in `.equals()` method (line A and line B).

8.5.2 Immer

In its repository, the library Immer is described as:

> Create the next immutable state by mutating the current one.

Immer helps with non-destructively updating (potentially nested) plain objects, Arrays, Sets, and Maps. That is, there are no custom data structures involved.

This is what using Immer looks like:

```
import {produce} from 'immer/dist/immer.module.js';

const people = [
  {name: 'Jane', work: {employer: 'Acme'}},
];

const modifiedPeople = produce(people, (draft) => {
```

```
    draft[0].work.employer = 'Cyberdyne';
    draft.push({name: 'John', work: {employer: 'Spectre'}});
  });

  assert.deepEqual(modifiedPeople, [
    {name: 'Jane', work: {employer: 'Cyberdyne'}},
    {name: 'John', work: {employer: 'Spectre'}},
  ]);
  assert.deepEqual(people, [
    {name: 'Jane', work: {employer: 'Acme'}},
  ]);
```

The original data is stored in `people`. `produce()` provides us with a variable `draft`. We pretend that this variable is `people` and use operations with which we would normally make destructive changes. Immer intercepts these operations. Instead of mutating `draft`, it non-destructively changes `people`. The result is referenced by `modifiedPeople`. As a bonus, it is deeply immutable.

`assert.deepEqual()` works because Immer returns plain objects and Arrays.

Part IV

OOP: object property attributes

Chapter 9

Property attributes: an introduction

Contents

9.1 The structure of objects **80**
9.1.1 Internal slots 80
9.1.2 Property keys 81
9.1.3 Property attributes 81
9.2 Property descriptors **82**
9.3 Retrieving descriptors for properties **83**
9.3.1 `Object.getOwnPropertyDescriptor()`: retrieving a descriptor for a single property 83
9.3.2 `Object.getOwnPropertyDescriptors()`: retrieving descriptors for all properties of an object 83
9.4 Defining properties via descriptors **84**
9.4.1 `Object.defineProperty()`: defining single properties via descriptors 84
9.4.2 `Object.defineProperties()`: defining multiple properties via descriptors 86
9.5 `Object.create()`: Creating objects via descriptors **86**
9.6 Use cases for `Object.getOwnPropertyDescriptors()` **87**
9.6.1 Use case: copying properties into an object 87
9.6.2 Use case for `Object.getOwnPropertyDescriptors()`: cloning objects 89
9.7 Omitting descriptor properties **89**
9.7.1 Omitting descriptor properties when creating properties . . . 89
9.7.2 Omitting descriptor properties when changing properties . . . 90
9.8 What property attributes do built-in constructs use? **90**
9.8.1 Own properties created via assignment 90
9.8.2 Own properties created via object literals 91
9.8.3 The own property `.length` of Arrays 91

9.8.4 Prototype properties of built-in classes 91
9.8.5 Prototype properties and instance properties of user-defined classes 92
9.9 API: property descriptors 93
9.10 Further reading 95

In this chapter, we take a closer look at how the ECMAScript specification sees JavaScript objects. In particular, properties are not atomic in the spec, but composed of multiple *attributes* (think fields in a record). Even the value of a data property is stored in an attribute!

9.1 The structure of objects

In the ECMAScript specification, an object consists of:

- *Internal slots*, which are storage locations that are not accessible from JavaScript, only from operations in the specification.
- A collection of *properties*. Each property associates a *key* with *attributes* (think fields in a record).

9.1.1 Internal slots

The specification describes internal slots as follows. I added bullet points and emphasized one part:

- Internal slots correspond to internal state that is associated with objects and used by various ECMAScript specification algorithms.
- Internal slots are not object properties and they are not inherited.
- Depending upon the specific internal slot specification, such state may consist of:
  - values of any ECMAScript language type or
  - of specific ECMAScript specification type values.
- Unless explicitly specified otherwise, internal slots are allocated as part of the process of creating an object and may not be dynamically added to an object.
- Unless specified otherwise, the initial value of an internal slot is the value `undefined`.
- Various algorithms within this specification create objects that have internal slots. However, **the ECMAScript language provides no direct way to associate internal slots with an object**.
- Internal methods and internal slots are identified within this specification using names enclosed in double square brackets `[[ ]]`.

There are two kinds of internal slots:

- Method slots for manipulating objects (getting properties, setting properties, etc.)
- Data slots that store values.

Ordinary objects have the following data slots:

- `.[[Prototype]]: null | object`
  - Stores the prototype of an object.

  - Can be accessed indirectly via `Object.getPrototypeOf()` and `Object.setPrototypeOf()`.
- `.[[Extensible]]: boolean`
  - Indicates if it is possible to add properties to an object.
  - Can be set to `false` via `Object.preventExtensions()`.
- `.[[PrivateFieldValues]]: EntryList`
  - Is used to manage private class fields.

9.1.2 Property keys

The key of a property is either:

- A string
- A symbol

9.1.3 Property attributes

There are two kinds of properties and they are characterized by their attributes:

- A *data property* stores data. Its attribute `value` holds any JavaScript value.
- An *accessor property* consists of a getter function and/or a setter function. The former is stored in the attribute `get`, the latter in the attribute `set`.

Additionally, there are attributes that both kinds of properties have. The following table lists all attributes and their default values.

Kind of property	Name and type of attribute	Default value
Data property	`value: any`	`undefined`
	`writable: boolean`	`false`
Accessor property	`get: (this: any) => any`	`undefined`
	`set: (this: any, v: any) => void`	`undefined`
All properties	`configurable: boolean`	`false`
	`enumerable: boolean`	`false`

We have already encountered the attributes `value`, `get`, and `set`. The other attributes work as follows:

- `writable` determines if the value of a data property can be changed.
- `configurable` determines if the attributes of a property can be changed. If it is `false`, then:
  - We cannot delete the property.
  - We cannot change a property from a data property to an accessor property or vice versa.
  - We cannot change any attribute other than `value`.
  - However, one more attribute change is allowed: We can change `writable` from `true` to `false`. The rationale behind this anomaly is historical: Property `.length` of Arrays has always been writable and non-configurable. Allowing its `writable` attribute to be changed enables us to freeze Arrays.
- `enumerable` influences some operations (such as `Object.keys()`). If it is `false`,

then those operations ignore the property. Most properties are enumerable (e.g. those created via assignment or object literals), which is why you'll rarely notice this attribute in practice. If you are still interested in how it works, see [content not included].

9.1.3.1 Pitfall: Inherited non-writable properties prevent creating own properties via assignment

If an inherited property is non-writable, we can't use assignment to create an own property with the same key:

```
const proto = {
  prop: 1,
};
// Make proto.prop non-writable:
Object.defineProperty(
  proto, 'prop', {writable: false});

const obj = Object.create(proto);

assert.throws(
  () => obj.prop = 2,
  /^TypeError: Cannot assign to read only property 'prop'/);
```

For more information, see [content not included].

9.2 Property descriptors

A *property descriptor* encodes the attributes of a property as a JavaScript object. Their TypeScript interfaces look as follows.

```
interface DataPropertyDescriptor {
  value?: any;
  writable?: boolean;
  configurable?: boolean;
  enumerable?: boolean;
}
interface AccessorPropertyDescriptor {
  get?: (this: any) => any;
  set?: (this: any, v: any) => void;
  configurable?: boolean;
  enumerable?: boolean;
}
type PropertyDescriptor = DataPropertyDescriptor | AccessorPropertyDescriptor;
```

The question marks indicate that all properties are optional. §9.7 "Omitting descriptor properties" describes what happens if they are omitted.

9.3 Retrieving descriptors for properties

9.3.1 `Object.getOwnPropertyDescriptor()`: retrieving a descriptor for a single property

Consider the following object:

```
const legoBrick = {
  kind: 'Plate 1x3',
  color: 'yellow',
  get description() {
    return `${this.kind} (${this.color})`;
  },
};
```

Let's first get a descriptor for the data property `.color`:

```
assert.deepEqual(
  Object.getOwnPropertyDescriptor(legoBrick, 'color'),
  {
    value: 'yellow',
    writable: true,
    enumerable: true,
    configurable: true,
  });
```

This is what the descriptor for the accessor property `.description` looks like:

```
const desc = Object.getOwnPropertyDescriptor.bind(Object);
assert.deepEqual(
  Object.getOwnPropertyDescriptor(legoBrick, 'description'),
  {
    get: desc(legoBrick, 'description').get, // (A)
    set: undefined,
    enumerable: true,
    configurable: true
  });
```

Using the utility function `desc()` in line A ensures that `.deepEqual()` works.

9.3.2 `Object.getOwnPropertyDescriptors()`: retrieving descriptors for all properties of an object

```
const legoBrick = {
  kind: 'Plate 1x3',
  color: 'yellow',
  get description() {
    return `${this.kind} (${this.color})`;
  },
};
```

```
const desc = Object.getOwnPropertyDescriptor.bind(Object);
assert.deepEqual(
  Object.getOwnPropertyDescriptors(legoBrick),
  {
    kind: {
      value: 'Plate 1x3',
      writable: true,
      enumerable: true,
      configurable: true,
    },
    color: {
      value: 'yellow',
      writable: true,
      enumerable: true,
      configurable: true,
    },
    description: {
      get: desc(legoBrick, 'description').get, // (A)
      set: undefined,
      enumerable: true,
      configurable: true,
    },
  });
```

Using the helper function `desc()` in line A ensures that `.deepEqual()` works.

9.4 Defining properties via descriptors

If we define a property with the key `k` via a property descriptor `propDesc`, then what happens depends:

- If there is no property with key `k`, a new own property is created that has the attributes specified by `propDesc`.
- If there is a property with key `k`, defining changes the property's attributes so that they match `propDesc`.

9.4.1 `Object.defineProperty()`: defining single properties via descriptors

First, let us create a new property via a descriptor:

```
const car = {};

Object.defineProperty(car, 'color', {
  value: 'blue',
  writable: true,
  enumerable: true,
  configurable: true,
});
```

```
assert.deepEqual(
  car,
  {
    color: 'blue',
  });
```

Next, we change the kind of a property via a descriptor; we turn a data property into a getter:

```
const car = {
  color: 'blue',
};

let readCount = 0;
Object.defineProperty(car, 'color', {
  get() {
    readCount++;
    return 'red';
  },
});

assert.equal(car.color, 'red');
assert.equal(readCount, 1);
```

Lastly, we change the value of a data property via a descriptor:

```
const car = {
  color: 'blue',
};

// Use the same attributes as assignment:
Object.defineProperty(
  car, 'color', {
    value: 'green',
    writable: true,
    enumerable: true,
    configurable: true,
  });

assert.deepEqual(
  car,
  {
    color: 'green',
  });
```

We have used the same property attributes as assignment.

9.4.2 `Object.defineProperties()`: defining multiple properties via descriptors

`Object.defineProperties()` is the multi-property version of ‘`Object.defineProperty()`:

```
const legoBrick1 = {};
Object.defineProperties(
  legoBrick1,
  {
    kind: {
      value: 'Plate 1x3',
      writable: true,
      enumerable: true,
      configurable: true,
    },
    color: {
      value: 'yellow',
      writable: true,
      enumerable: true,
      configurable: true,
    },
    description: {
      get: function () {
        return `${this.kind} (${this.color})`;
      },
      enumerable: true,
      configurable: true,
    },
  });

assert.deepEqual(
  legoBrick1,
  {
    kind: 'Plate 1x3',
    color: 'yellow',
    get description() {
      return `${this.kind} (${this.color})`;
    },
  });
```

9.5 `Object.create()`: Creating objects via descriptors

`Object.create()` creates a new object. Its first argument specifies the prototype of that object. Its optional second argument specifies descriptors for the properties of that object. In the next example, we create the same object as in the previous example.

```
const legoBrick2 = Object.create(
  Object.prototype,
  {
```

```
    kind: {
      value: 'Plate 1x3',
      writable: true,
      enumerable: true,
      configurable: true,
    },
    color: {
      value: 'yellow',
      writable: true,
      enumerable: true,
      configurable: true,
    },
    description: {
      get: function () {
        return `${this.kind} (${this.color})`;
      },
      enumerable: true,
      configurable: true,
    },
  });

// Did we really create the same object?
assert.deepEqual(legoBrick1, legoBrick2); // Yes!
```

9.6 Use cases for `Object.getOwnPropertyDescriptors()`

`Object.getOwnPropertyDescriptors()` helps us with two use cases, if we combine it with `Object.defineProperties()` or `Object.create()`.

9.6.1 Use case: copying properties into an object

Since ES6, JavaScript already has had a tool method for copying properties: `Object.assign()`. However, this method uses simple get and set operations to copy a property whose key is `key`:

```
target[key] = source[key];
```

That means that it only creates a faithful copy of a property if:

- Its attribute `writable` is `true` and its attribute `enumerable` is `true` (because that's how assignment creates properties).
- It is a data property.

The following example illustrates this limitation. Object `source` has a setter whose key is `data`.

```
const source = {
  set data(value) {
    this._data = value;
  }
```

```
};

// Because there is only a setter, property `data` exists,
// but has the value `undefined`.
assert.equal('data' in source, true);
assert.equal(source.data, undefined);
```

If we use `Object.assign()` to copy property `data`, then the accessor property `data` is converted to a data property:

```
const target1 = {};
Object.assign(target1, source);

assert.deepEqual(
  Object.getOwnPropertyDescriptor(target1, 'data'),
  {
    value: undefined,
    writable: true,
    enumerable: true,
    configurable: true,
  });

// For comparison, the original:
const desc = Object.getOwnPropertyDescriptor.bind(Object);
assert.deepEqual(
  Object.getOwnPropertyDescriptor(source, 'data'),
  {
    get: undefined,
    set: desc(source, 'data').set,
    enumerable: true,
    configurable: true,
  });
```

Fortunately, using `Object.getOwnPropertyDescriptors()` together with `Object.defineProperties()` does faithfully copy the property `data`:

```
const target2 = {};
Object.defineProperties(
  target2, Object.getOwnPropertyDescriptors(source));

assert.deepEqual(
  Object.getOwnPropertyDescriptor(target2, 'data'),
  {
    get: undefined,
    set: desc(source, 'data').set,
    enumerable: true,
    configurable: true,
  });
```

9.6.1.1 Pitfall: copying methods that use `super`

A method that uses `super` is firmly connected with its *home object* (the object it is stored in). There is currently no way to copy or move such a method to a different object.

9.6.2 Use case for `Object.getOwnPropertyDescriptors()`: cloning objects

Shallow cloning is similar to copying properties, which is why `Object.getOwnPropertyDescriptors()` is a good choice here, too.

To create the clone, we use `Object.create()`:

```
const original = {
  set data(value) {
    this._data = value;
  }
};

const clone = Object.create(
  Object.getPrototypeOf(original),
  Object.getOwnPropertyDescriptors(original));

assert.deepEqual(original, clone);
```

For more information on this topic, see §6 “Copying objects and Arrays”.

9.7 Omitting descriptor properties

All properties of descriptors are optional. What happens when you omit a property depends on the operation.

9.7.1 Omitting descriptor properties when creating properties

When we create a new property via a descriptor, then omitting attributes means that their default values are used:

```
const car = {};
Object.defineProperty(
  car, 'color', {
    value: 'red',
  });
assert.deepEqual(
  Object.getOwnPropertyDescriptor(car, 'color'),
  {
    value: 'red',
    writable: false,
    enumerable: false,
    configurable: false,
  });
```

9.7.2 Omitting descriptor properties when changing properties

If instead, we change an existing property, then omitting descriptor properties means that the corresponding attributes are not touched:

```
const car = {
  color: 'yellow',
};
assert.deepEqual(
  Object.getOwnPropertyDescriptor(car, 'color'),
  {
    value: 'yellow',
    writable: true,
    enumerable: true,
    configurable: true,
  });
Object.defineProperty(
  car, 'color', {
    value: 'pink',
  });
assert.deepEqual(
  Object.getOwnPropertyDescriptor(car, 'color'),
  {
    value: 'pink',
    writable: true,
    enumerable: true,
    configurable: true,
  });
```

9.8 What property attributes do built-in constructs use?

The general rule (with few exceptions) for property attributes is:

- Properties of objects at the beginning of a prototype chain are usually writable, enumerable, and configurable.
- As described in the chapter on enumerability, most inherited properties are non-enumerable, to hide them from legacy constructs such as `for-in` loops. Inherited properties are usually writable and configurable.

9.8.1 Own properties created via assignment

```
const obj = {};
obj.prop = 3;

assert.deepEqual(
  Object.getOwnPropertyDescriptors(obj),
  {
    prop: {
```

```
      value: 3,
      writable: true,
      enumerable: true,
      configurable: true,
    }
  });
```

9.8.2 Own properties created via object literals

```
const obj = { prop: 'yes' };

assert.deepEqual(
  Object.getOwnPropertyDescriptors(obj),
  {
    prop: {
      value: 'yes',
      writable: true,
      enumerable: true,
      configurable: true
    }
  });
```

9.8.3 The own property `.length` of Arrays

The own property `.length` of Arrays is non-enumerable, so that it isn't copied by `Object.assign()`, spreading, and similar operations. It is also non-configurable:

```
> Object.getOwnPropertyDescriptor([], 'length')
{ value: 0, writable: true, enumerable: false, configurable: false }
> Object.getOwnPropertyDescriptor('abc', 'length')
{ value: 3, writable: false, enumerable: false, configurable: false }
```

`.length` is a special data property, in that it is influenced by (and influences) other own properties (specifically, index properties).

9.8.4 Prototype properties of built-in classes

```
assert.deepEqual(
  Object.getOwnPropertyDescriptor(Array.prototype, 'map'),
  {
    value: Array.prototype.map,
    writable: true,
    enumerable: false,
    configurable: true
  });
```

9.8.5 Prototype properties and instance properties of user-defined classes

```
class DataContainer {
  accessCount = 0;
  constructor(data) {
    this.data = data;
  }
  getData() {
    this.accessCount++;
    return this.data;
  }
}
assert.deepEqual(
  Object.getOwnPropertyDescriptors(DataContainer.prototype),
  {
    constructor: {
      value: DataContainer,
      writable: true,
      enumerable: false,
      configurable: true,
    },
    getData: {
      value: DataContainer.prototype.getData,
      writable: true,
      enumerable: false,
      configurable: true,
    }
  });
```

Note that all own properties of instances of `DataContainer` are writable, enumerable, and configurable:

```
const dc = new DataContainer('abc')
assert.deepEqual(
  Object.getOwnPropertyDescriptors(dc),
  {
    accessCount: {
      value: 0,
      writable: true,
      enumerable: true,
      configurable: true,
    },
    data: {
      value: 'abc',
      writable: true,
      enumerable: true,
      configurable: true,
    }
```

```
});
```

9.9 API: property descriptors

The following tool methods use property descriptors:

- `Object.defineProperty(obj: object, key: string|symbol, propDesc: PropertyDescriptor): object` [ES5]

  Creates or changes a property on `obj` whose key is `key` and whose attributes are specified via `propDesc`. Returns the modified object.

  ```
  const obj = {};
  const result = Object.defineProperty(
    obj, 'happy', {
      value: 'yes',
      writable: true,
      enumerable: true,
      configurable: true,
    });

  // obj was returned and modified:
  assert.equal(result, obj);
  assert.deepEqual(obj, {
    happy: 'yes',
  });
  ```

- `Object.defineProperties(obj: object, properties: {[k: string|symbol]: PropertyDescriptor}): object` [ES5]

  The batch version of `Object.defineProperty()`. Each property `p` of the object `properties` specifies one property that is to be added to `obj`: The key of `p` specifies the key of the property, the value of `p` is a descriptor that specifies the attributes of the property.

  ```
  const address1 = Object.defineProperties({}, {
    street: { value: 'Evergreen Terrace', enumerable: true },
    number: { value: 742, enumerable: true },
  });
  ```

- `Object.create(proto: null|object, properties?: {[k: string|symbol]: PropertyDescriptor}): object` [ES5]

  First, creates an object whose prototype is `proto`. Then, if the optional parameter `properties` has been provided, adds properties to it – in the same manner as `Object.defineProperties()`. Finally, returns the result. For example, the following code snippet produces the same result as the previous snippet:

  ```
  const address2 = Object.create(Object.prototype, {
    street: { value: 'Evergreen Terrace', enumerable: true },
    number: { value: 742, enumerable: true },
  ```

```
});
assert.deepEqual(address1, address2);
```

- `Object.getOwnPropertyDescriptor(obj: object, key: string|symbol): undefined|PropertyDescriptor` [ES5]

  Returns the descriptor of the own (non-inherited) property of `obj` whose key is `key`. If there is no such property, `undefined` is returned.

  ```
  assert.deepEqual(
    Object.getOwnPropertyDescriptor(Object.prototype, 'toString'),
    {
      value: {}.toString,
      writable: true,
      enumerable: false,
      configurable: true,
    });
  assert.equal(
    Object.getOwnPropertyDescriptor({}, 'toString'),
    undefined);
  ```

- `Object.getOwnPropertyDescriptors(obj: object): {[k: string|symbol]: PropertyDescriptor}` [ES2017]

  Returns an object where each property key `'k'` of `obj` is mapped to the property descriptor for `obj.k`. The result can be used as input for `Object.defineProperties()` and `Object.create()`.

  ```
  const propertyKey = Symbol('propertyKey');
  const obj = {
    [propertyKey]: 'abc',
    get count() { return 123 },
  };

  const desc = Object.getOwnPropertyDescriptor.bind(Object);
  assert.deepEqual(
    Object.getOwnPropertyDescriptors(obj),
    {
      [propertyKey]: {
        value: 'abc',
        writable: true,
        enumerable: true,
        configurable: true
      },
      count: {
        get: desc(obj, 'count').get, // (A)
        set: undefined,
        enumerable: true,
        configurable: true
      }
    });
  ```

Using `desc()` in line A is a work-around so that `.deepEqual()` works.

9.10 Further reading

The next three chapters provide more details on property attributes:

- [content not included]
- [content not included]
- [content not included]

Chapter 10

Where are the remaining chapters?

You are reading a preview of this book:

- The full version of this book is available for purchase.
- You can take a look at the full table of contents (also linked to from the book's homepage).