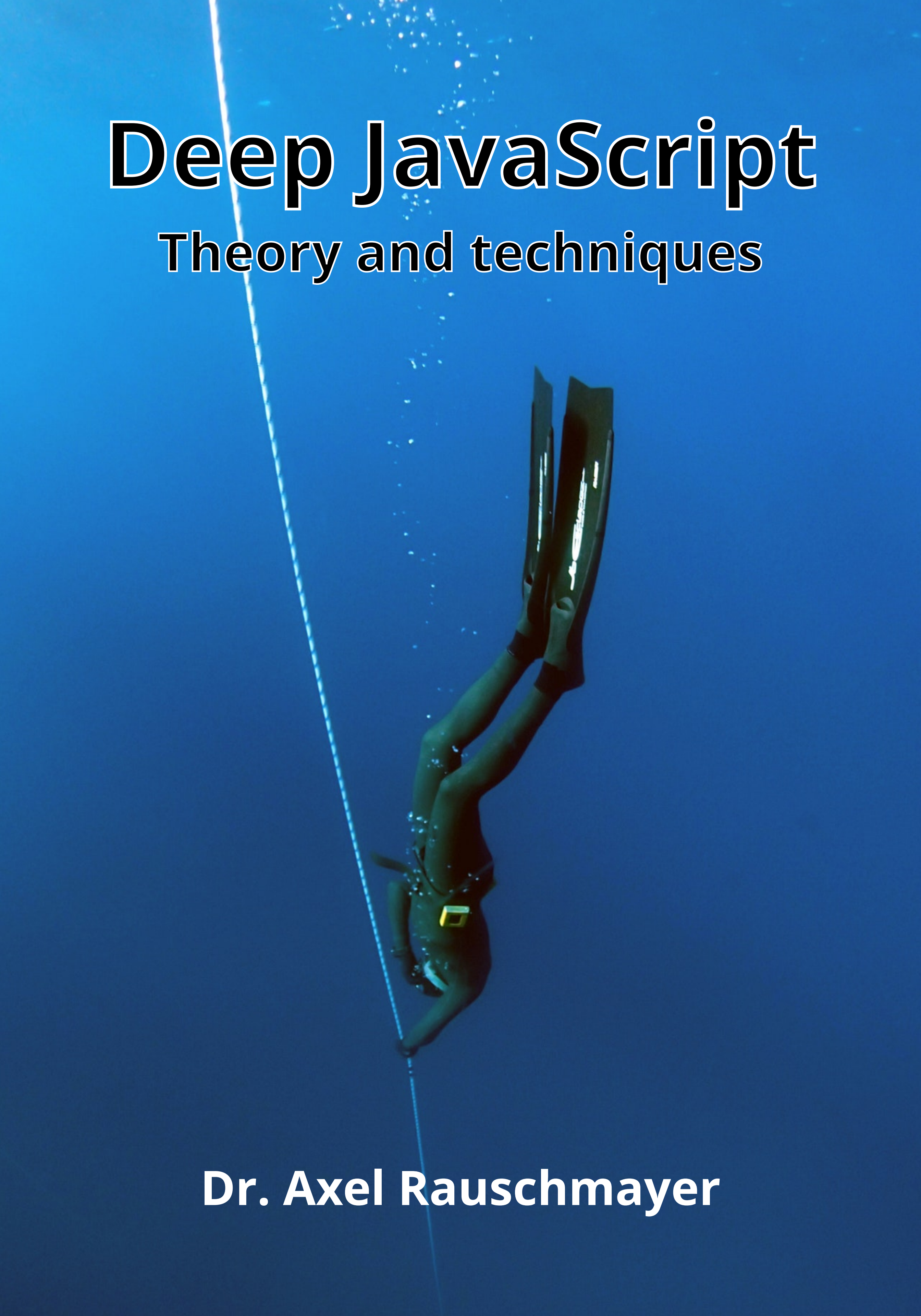

Deep JavaScript

Theory and techniques

Dr. Axel Rauschmayer

Deep JavaScript

Dr. Axel Rauschmayer

2019

Contents

Part I

Frontmatter

Chapter 1

About this book

Contents

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

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`: $0 \leq$ `l` $\leq 2^{53}-1$

- `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

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

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).

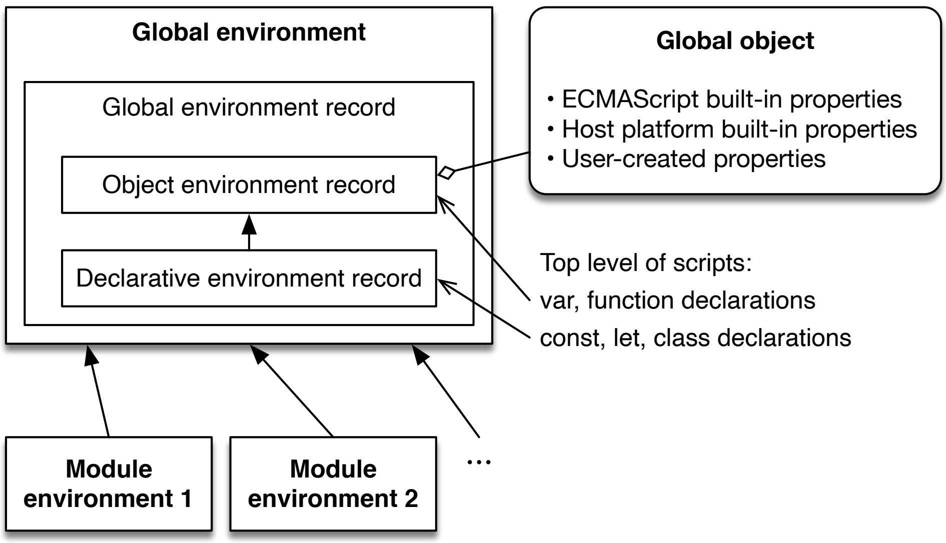
Global environment
Global environment record
Object environment record
Declarative environment record
Global object
• ECMAScript built-in properties
• Host platform built-in properties
• User-created properties
Top level of scripts:
var, function declarations
const, let, class declarations
Module environment 1
Module environment 2
…

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

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

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 §12 "Enumerability of properties".

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

- §14 "Copying instances of classes: `.clone()` vs. copy constructors" 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

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

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"
> - §14 "Copying instances of classes: `.clone()` vs. copy constructors"

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:

- §10 "Protecting objects from being changed"
- §15 "Immutable wrappers for collections"

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

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 §12 "Enumerability of properties".

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 §11.3.4 "Inherited read-only properties prevent creating own properties via assignment".

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:

- §10 "Protecting objects from being changed"
- §11 "Properties: assignment vs. definition"
- §12 "Enumerability of properties"

Chapter 10

Protecting objects from being changed

Contents

In this chapter, we'll look at how objects can be protected from being changed. Examples include: Preventing properties being added and preventing properties being changed.

Required knowledge: property attributes

For this chapter, you should be familiar with property attributes. If you aren't, check out §9 "Property attributes: an introduction".

10.1 Levels of protection: preventing extensions, sealing, freezing

JavaScript has three levels of protecting objects:

- *Preventing extensions* makes it impossible to add new properties to an object. We can still delete and change properties, though.

    - Method: `Object.preventExtensions(obj)`
- *Sealing* prevents extensions and makes all properties *unconfigurable* (roughly: we can't change how a property works anymore).
    - Method: `Object.seal(obj)`
- *Freezing* seals an object after making all of its properties non-writable. That is, the object is not extensible, all properties are read-only and there is no way to change that.
    - Method: `Object.freeze(obj)`

10.2 Preventing extensions

```
Object.preventExtensions<T>(obj: T): T
```

This method works as follows:

- If `obj` is not an object, it returns it.
- Otherwise, it changes `obj` so that we can't add properties anymore and returns it.
- The type parameter <T> expresses that the result has the same type as the parameter.

Let's use `Object.preventExtensions()` in an example:

```
const obj = { first: 'Jane' };
Object.preventExtensions(obj);
assert.throws(
  () => obj.last = 'Doe',
  /^TypeError: Cannot add property last, object is not extensible$/);
```

We can still delete properties, though:

```
assert.deepEquals(
  Object.keys(obj), ['first']);
delete obj.first;
assert.deepEquals(
  Object.keys(obj), []);
```

10.2.1 Checking whether an object is extensible

```
Object.isExtensible(obj: any): boolean
```

Checks whether `obj` is extensible – for example:

```
> const obj = {};
> Object.isExtensible(obj)
true
> Object.preventExtensions(obj)
{}
> Object.isExtensible(obj)
false
```

10.3 Sealing

```
Object.seal<T>(obj: T): T
```

Description of this method:

- If `obj` isn't an object, it returns it.
- Otherwise, it prevents extensions of `obj`, makes all of its properties *unconfigurable*, and returns it. The properties being unconfigurable means that they can't be changed, anymore (except for their values): Read-only properties stay read-only, enumerable properties stay enumerable, etc.

The following example demonstrates that sealing makes the object non-extensible and its properties unconfigurable.

```
const obj = {
  first: 'Jane',
  last: 'Doe',
};

// Before sealing
assert.equal(Object.isExtensible(obj), true);
assert.deepEqual(
  Object.getOwnPropertyDescriptors(obj),
  {
    first: {
      value: 'Jane',
      writable: true,
      enumerable: true,
      configurable: true
    },
    last: {
      value: 'Doe',
      writable: true,
      enumerable: true,
      configurable: true
    }
  });

Object.seal(obj);

// After sealing
assert.equal(Object.isExtensible(obj), false);
assert.deepEqual(
  Object.getOwnPropertyDescriptors(obj),
  {
    first: {
      value: 'Jane',
      writable: true,
      enumerable: true,
      configurable: false
    },
    last: {
      value: 'Doe',
      writable: true,
      enumerable: true,
      configurable: false
    }
  });
```

We can still change the the value of property `.first`:

```
obj.first = 'John';
assert.deepEqual(
  obj, {first: 'John', last: 'Doe'});
```

But we can’t change its attributes:

```
assert.throws(
  () => Object.defineProperty(obj, 'first', { enumerable: false }),
  /^TypeError: Cannot redefine property: first$/);
```

10.3.1 Checking whether an object is sealed

```
Object.isSealed(obj: any): boolean
```

Checks whether `obj` is sealed – for example:

```
> const obj = {};
> Object.isSealed(obj)
false
> Object.seal(obj)
{}
> Object.isSealed(obj)
true
```

10.4 Freezing

```
Object.freeze<T>(obj: T): T;
```

- This method immediately returns `obj` if it isn’t an object.
- Otherwise, it makes all properties non-writable, seals `obj`, and returns it. That is, `obj` is not extensible, all properties are read-only and there is no way to change that.

```
const point = { x: 17, y: -5 };
Object.freeze(point);

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

```
assert.throws(
  () => Object.defineProperty(point, 'x', {enumerable: false}),
  /^TypeError: Cannot redefine property: x$/);

assert.throws(
  () => point.z = 4,
  /^TypeError: Cannot add property z, object is not extensible$/);
```

10.4.1 Checking whether an object is frozen

```
Object.isFrozen(obj: any): boolean
```

Checks whether `obj` is frozen – for example:

```
> const point = { x: 17, y: -5 };
> Object.isFrozen(point)
false
> Object.freeze(point)
{ x: 17, y: -5 }
> Object.isFrozen(point)
true
```

10.4.2 Freezing is shallow

`Object.freeze(obj)` only freezes `obj` and its properties. It does not freeze the values of those properties – for example:

```
const teacher = {
  name: 'Edna Krabappel',
  students: ['Bart'],
};
Object.freeze(teacher);

// We can’t change own properties:
assert.throws(
  () => teacher.name = 'Elizabeth Hoover',
  /^TypeError: Cannot assign to read only property 'name'/);

// Alas, we can still change values of own properties:
teacher.students.push('Lisa');
assert.deepEqual(
  teacher, {
    name: 'Edna Krabappel',
    students: ['Bart', 'Lisa'],
  });
```

10.4.3 Implementing deep freezing

If we want deep freezing, we need to implement it ourselves:

```
function deepFreeze(value) {
  if (Array.isArray(value)) {
    for (const element of value) {
      deepFreeze(element);
    }
    Object.freeze(value);
  } else if (typeof value === 'object' && value !== null) {
    for (const v of Object.values(value)) {
      deepFreeze(v);
    }
    Object.freeze(value);
  } else {
    // Nothing to do: primitive values are already immutable
  }
  return value;
}
```

Revisiting the example from the previous section, we can check if `deepFreeze()` really freezes deeply:

```
const teacher = {
  name: 'Edna Krabappel',
  students: ['Bart'],
};
deepFreeze(teacher);

assert.throws(
  () => teacher.students.push('Lisa'),
  /^TypeError: Cannot add property 1, object is not extensible$/);
```

10.5 Further reading

- More information on patterns for preventing data structures from being changed: §15 "Immutable wrappers for collections"

Chapter 11

Properties: assignment vs. definition

Contents

There are two ways of creating or changing a property `prop` of an object `obj`:

- *Assigning*: `obj.prop = true`
- *Defining*: `Object.defineProperty(obj, '', {value: true})`

This chapter explains how they work.

Required knowledge: property attributes and property descriptors

For this chapter, you should be familiar with property attributes and property descriptors. If you aren't, check out §9 "Property attributes: an introduction".

11.1 Assignment vs. definition

11.1.1 Assignment

We use the assignment operator `=` to assign a value `value` to a property `.prop` of an object `obj`:

```
obj.prop = value
```

This operator works differently depending on what `.prop` looks like:

- Changing properties: If there is an own data property `.prop`, assignment changes its value to `value`.
- Invoking setters: If there is an own or inherited setter for `.prop`, assignment invokes that setter.
- Creating properties: If there is no own data property `.prop` and no own or inherited setter for it, assignment creates a new own data property.

That is, the main purpose of assignment is making changes. That's why it supports setters.

11.1.2 Definition

To define a property with the key `propKey` of an object `obj`, we use an operation such as the following method:

```
Object.defineProperty(obj, propKey, propDesc)
```

This method works differently depending on what the property looks like:

- Changing properties: If an own property with key `propKey` exists, defining changes its property attributes as specified by the property descriptor `propDesc` (if possible).
- Creating properties: Otherwise, defining creates an own property with the attributes specified by `propDesc` (if possible).

That is, the main purpose of definition is to create an own property (even if there is an inherited setter, which it ignores) and to change property attributes.

11.2 Assignment and definition in theory (optional)

Property descriptors in the ECMAScript specification

> In specification operations, property descriptors are not JavaScript objects but *Records*, a spec-internal data structure that has *fields*. The keys of fields are written in double brackets. For example, `Desc.[[Configurable]]` accesses the field `.[[Configurable]]` of `Desc`. These records are translated to and from JavaScript objects when interacting with the outside world.

11.2.1 Assigning to a property

The actual work of assigning to a property is handled via the following operation in the ECMAScript specification:

```
OrdinarySetWithOwnDescriptor(O, P, V, Receiver, ownDesc)
```

These are the parameters:

- `O` is the object that is currently being visited.
- `P` is the key of the property that we are assigning to.
- `V` is the value we are assigning.
- `Receiver` is the object where the assignment started.
- `ownDesc` is the descriptor of `O[P]` or `null` if that property doesn't exist.

The return value is a boolean that indicates whether or not the operation succeeded. As explained later in this chapter, strict-mode assignment throws a `TypeError` if `OrdinarySetWithOwnDescriptor()` fails.

This is a high-level summary of the algorithm:

- It traverses the prototype chain of `Receiver` until it finds a property whose key is `P`. The traversal is done by calling `OrdinarySetWithOwnDescriptor()` recursively. During recursion, `O` changes and points to the object that is currently being visited, but `Receiver` stays the same.
- Depending on what the traversal finds, an own property is created in `Receiver` (where recursion started) or something else happens.

In more detail, this algorithm works as follows:

- If `ownDesc` is `undefined`, then we haven't yet found a property with key `P`:
  - If `O` has a prototype `parent`, then we return `parent.[[Set]](P, V, Receiver)`. This continues our search. The method call usually ends up invoking `OrdinarySetWithOwnDescriptor()` recursively.
  - Otherwise, our search for `P` has failed and we set `ownDesc` as follows:
    ```
    {
      [[Value]]: undefined, [[Writable]]: true,
      [[Enumerable]]: true, [[Configurable]]: true
    }
    ```
    With this `ownDesc`, the next `if` statement will create an own property in `Receiver`.
- If `ownDesc` specifies a data property, then we have found a property:
  - If `ownDesc.[[Writable]]` is `false`, return `false`. This means that any non-writable property `P` (own or inherited!) prevents assignment.
  - Let `existingDescriptor` be `Receiver.[[GetOwnProperty]](P)`. That is, retrieve the descriptor of the property where the assignment started. We now

have:
    * The current object `O` and the current property descriptor `ownDesc` on one hand.
    * The original object `Receiver` and the original property descriptor `existingDescriptor` on the other hand.
  - If `existingDescriptor` is not `undefined`:
    * (If we get here, then we are still at the beginning of the prototype chain – we only recurse if `Receiver` does not have a property `P`.)
    * The following two `if` conditions should never be `true` because `ownDesc` and `existingDesc` should be equal:
      · If `existingDescriptor` specifies an accessor, return `false`.
      · If `existingDescriptor.[[Writable]]` is `false`, return `false`.
    * Return `Receiver.[[DefineOwnProperty]](P, { [[Value]]: V })`. This internal method performs definition, which we use to change the value of property `Receiver[P]`. The definition algorithm is described in the next subsection.
  - Else:
    * (If we get here, then `Receiver` does not have an own property with key `P`.)
    * Return `CreateDataProperty(Receiver, P, V)`. (This operation creates an own data property in its first argument.)
- (If we get here, then `ownDesc` describes an accessor property that is own or inherited.)
- Let `setter` be `ownDesc.[[Set]]`.
- If `setter` is `undefined`, return `false`.
- Perform `Call(setter, Receiver, «V»)`. `Call()` invokes the function object `setter` with `this` set to `Receiver` and the single parameter `V` (French quotes «» are used for lists in the specification).
- Return `true`.

11.2.1.1 How do we get from an assignment to `OrdinarySetWithOwnDescriptor()`?

Evaluating an assignment without destructuring involves the following steps:

- In the spec, evaluation starts in the section on the runtime semantics of `AssignmentExpression`. This section handles providing names for anonymous functions, destructuring, and more.
- If there is no destructuring pattern, then `PutValue()` is used to make the assignment.
- For property assignments, `PutValue()` invokes the internal method `.[[Set]]()`.
- For ordinary objects, `.[[Set]]()` calls `OrdinarySet()` (which calls `OrdinarySetWithOwnDescriptor()`) and returns the result.

Notably, `PutValue()` throws a `TypeError` in strict mode if the result of `.[[Set]]()` is `false`.

11.2.2 Defining a property

The actual work of defining a property is handled via the following operation in the ECMAScript specification:

```
ValidateAndApplyPropertyDescriptor(O, P, extensible, Desc, current)
```

The parameters are:

- The object `O` where we want to define a property. There is a special validation-only mode where `O` is `undefined`. We are ignoring this mode here.
- The property key `P` of the property we want to define.
- `extensible` indicates if `O` is extensible.
- `Desc` is a property descriptor specifying the attributes we want the property to have.
- `current` contains the property descriptor of an own property `O[P]` if it exists. Otherwise, `current` is `undefined`.

The result of the operation is a boolean that indicates if it succeeded. Failure can have different consequences. Some callers ignore the result. Others, such as `Object.defineProperty()`, throw an exception if the result is `false`.

This is a summary of the algorithm:

- If `current` is `undefined`, then property `P` does not currently exist and must be created.
  - If `extensible` is `false`, return `false` indicating that the property could not be added.
  - Otherwise, check `Desc` and create either a data property or an accessor property.
  - Return `true`.
- If `Desc` doesn't have any fields, return `true` indicating that the operation succeeded (because no changes had to be made).
- If `current.[[Configurable]]` is `false`:
  - (`Desc` is not allowed to change attributes other than `value`.)
  - If `Desc.[[Configurable]]` exists, it must have the same value as `current.[[Configurable]]`. If not, return `false`.
  - Same check: `Desc.[[Enumerable]]`
- Next, we *validate* the property descriptor `Desc`: Can the attributes described by `current` be changed to the values specified by `Desc`? If not, return `false`. If yes, go on.
  - If the descriptor is *generic* (with no attributes specific to data properties or accessor properties), then validation is successful and we can move on.
  - Else if one descriptor specifies a data property and the other an accessor property:
    - The current property must be configurable (otherwise its attributes can't be changed as necessary). If not, `false` is returned.

    * Change the current property from a data property to an accessor property or vice versa. When doing so, the values of `.[[Configurable]]` and `.[[Enumerable]]` are preserved, all other attributes get default values (`undefined` for object-valued attributes, `false` for boolean-valued attributes).
  - Else if both descriptors specify data properties:
    * If both `current.[[Configurable]]` and `current.[[Writable]]` are `false`, then no changes are allowed and `Desc` and `current` must specify the same attributes:
      · (Due to `current.[[Configurable]]` being `false`, `Desc.[[Configurable]]` and `Desc.[[Enumerable]]` were already checked previously and have the correct values.)
      · If `Desc.[[Writable]]` exists and is `true`, then return `false`.
      · If `Desc.[[Value]]` exists and does not have the same value as `current.[[Value]]`, then return `false`.
      · There is nothing more to do. Return `true` indicating that the algorithm succeeded.
      · (Note that normally, we can't change any attributes of a non-configurable property other than its value. The one exception to this rule is that we can always go from writable to non-writable. This algorithm handles this exception correctly.)
  - Else (both descriptors specify accessor properties):
    * If `current.[[Configurable]]` is `false`, then no changes are allowed and `Desc` and `current` must specify the same attributes:
      · (Due to `current.[[Configurable]]` being `false`, `Desc.[[Configurable]]` and `Desc.[[Enumerable]]` were already checked previously and have the correct values.)
      · If `Desc.[[Set]]` exists, it must have the same value as `current.[[Set]]`. If not, return `false`.
      · Same check: `Desc.[[Get]]`
      · There is nothing more to do. Return `true` indicating that the algorithm succeeded.

- Set the attributes of the property with key `P` to the values specified by `Desc`. Due to validation, we can be sure that all of the changes are allowed.

- Return `true`.

11.3 Definition and assignment in practice

This section describes some consequences of how property definition and assignment work.

11.3.1 Only definition allows us to create a property with arbitrary attributes

If we create an own property via assignment, it always creates properties whose attributes `writable`, `enumerable`, and `configurable` are all `true`.

```
const obj = {};
obj.dataProp = 'abc';
assert.deepEqual(
  Object.getOwnPropertyDescriptor(obj, 'dataProp'),
  {
    value: 'abc',
    writable: true,
    enumerable: true,
    configurable: true,
  });
```

Therefore, if we want to specify arbitrary attributes, we must use definition.

And while we can create getters and setters inside object literals, we can't add them later via assignment. Here, too, we need definition.

11.3.2 The assignment operator does not change properties in prototypes

Let us consider the following setup, where `obj` inherits the property `prop` from `proto`.

```
const proto = { prop: 'a' };
const obj = Object.create(proto);
```

We can't (destructively) change `proto.prop` by assigning to `obj.prop`. Doing so creates a new own property:

```
assert.deepEqual(
  Object.keys(obj), []);

obj.prop = 'b';

// The assignment worked:
assert.equal(obj.prop, 'b');

// But we created an own property and overrode proto.prop,
// we did not change it:
assert.deepEqual(
  Object.keys(obj), ['prop']);
assert.equal(proto.prop, 'a');
```

The rationale for this behavior is as follows: Prototypes can have properties whose values are shared by all of their descendants. If we want to change such a property in only one descendant, we must do so non-destructively, via overriding. Then the change does not affect the other descendants.

11.3.3 Assignment calls setters, definition doesn't

What is the difference between defining the property `.prop` of `obj` versus assigning to it?

If we define, then our intention is to either create or change an own (non-inherited) property of `obj`. Therefore, definition ignores the inherited setter for `.prop` in the following example:

```
let setterWasCalled = false;
const proto = {
  get prop() {
    return 'protoGetter';
  },
  set prop(x) {
    setterWasCalled = true;
  },
};
const obj = Object.create(proto);

assert.equal(obj.prop, 'protoGetter');

// Defining obj.prop:
Object.defineProperty(
  obj, 'prop', { value: 'objData' });
assert.equal(setterWasCalled, false);

// We have overridden the getter:
assert.equal(obj.prop, 'objData');
```

If, instead, we assign to `.prop`, then our intention is often to change something that already exists and that change should be handled by the setter:

```
let setterWasCalled = false;
const proto = {
  get prop() {
    return 'protoGetter';
  },
  set prop(x) {
    setterWasCalled = true;
  },
};
const obj = Object.create(proto);

assert.equal(obj.prop, 'protoGetter');

// Assigning to obj.prop:
obj.prop = 'objData';
assert.equal(setterWasCalled, true);

// The getter still active:
assert.equal(obj.prop, 'protoGetter');
```

11.3.4 Inherited read-only properties prevent creating own properties via assignment

What happens if `.prop` is read-only in a prototype?

```
const proto = Object.defineProperty(
  {}, 'prop', {
    value: 'protoValue',
    writable: false,
  });
```

In any object that inherits the read-only `.prop` from `proto`, we can't use assignment to create an own property with the same key – for example:

```
const obj = Object.create(proto);
assert.throws(
  () => obj.prop = 'objValue',
  /^TypeError: Cannot assign to read only property 'prop'/);
```

Why can't we assign? The rationale is that overriding an inherited property by creating an own property can be seen as non-destructively changing the inherited property. Arguably, if a property is non-writable, we shouldn't be able to do that.

However, defining `.prop` still works and lets us override:

```
Object.defineProperty(
  obj, 'prop', { value: 'objValue' });
assert.equal(obj.prop, 'objValue');
```

Accessor properties that don't have a setter are also considered to be read-only:

```
const proto = {
  get prop() {
    return 'protoValue';
  }
};
const obj = Object.create(proto);
assert.throws(
  () => obj.prop = 'objValue',
  /^TypeError: Cannot set property prop of #<Object> which has only a getter$/);
```

The "override mistake": pros and cons

The fact that read-only properties prevent assignment earlier in the prototype chain, has been given the name *override mistake*:

- It was introduced in ECMAScript 5.1.
- On one hand, this behavior is consistent with how prototypal inheritance and setters work. (So, arguably, it is *not* a mistake.)
- On the other hand, with the behavior, deep-freezing the global object causes unwanted side-effects.

- There was an attempt to change the behavior, but that broke the library Lodash and was abandoned (pull request on GitHub).
- Background knowledge:
  - Pull request on GitHub
  - Wiki page on ECMAScript.org (archived)

11.4 Which language constructs use definition, which assignment?

In this section, we examine where the language uses definition and where it uses assignment. We detect which operation is used by tracking whether or not inherited setters are called. See §11.3.3 "Assignment calls setters, definition doesn't" for more information.

11.4.1 The properties of an object literal are added via definition

When we create properties via an object literal, JavaScript always uses definition (and therefore never calls inherited setters):

```
let lastSetterArgument;
const proto = {
  set prop(x) {
    lastSetterArgument = x;
  },
};
const obj = {
  __proto__: proto,
  prop: 'abc',
};
assert.equal(lastSetterArgument, undefined);
```

11.4.2 The assignment operator = always uses assignment

The assignment operator = always uses assignment to create or change properties.

```
let lastSetterArgument;
const proto = {
  set prop(x) {
    lastSetterArgument = x;
  },
};
const obj = Object.create(proto);

// Normal assignment:
obj.prop = 'abc';
assert.equal(lastSetterArgument, 'abc');

// Assigning via destructuring:
```

```
[obj.prop] = ['def'];
assert.equal(lastSetterArgument, 'def');
```

11.4.3 Public class fields are added via definition

Alas, even though public class fields have the same syntax as assignment, they do *not* use assignment to create properties, they use definition (like properties in object literals):

```
let lastSetterArgument1;
let lastSetterArgument2;
class A {
  set prop1(x) {
    lastSetterArgument1 = x;
  }
  set prop2(x) {
    lastSetterArgument2 = x;
  }
}
class B extends A {
  prop1 = 'one';
  constructor() {
    super();
    this.prop2 = 'two';
  }
}
new B();

// The public class field uses definition:
assert.equal(lastSetterArgument1, undefined);
// Inside the constructor, we trigger assignment:
assert.equal(lastSetterArgument2, 'two');
```

11.5 Further reading and sources of this chapter

- Section "Prototype chains" in "JavaScript for impatient programmers"
- Email by Allen Wirfs-Brock to the es-discuss mailing list: "The distinction between assignment and definition […] was not very important when all ES had was data properties and there was no way for ES code to manipulate property attributes." [That changed with ECMAScript 5.]

Chapter 12

Enumerability of properties

Contents

Enumerability is an *attribute* of object properties. In this chapter, we take a closer look at how it is used and how it influences operations such as `Object.keys()` and `Object.assign()`.

Required knowledge: property attributes

For this chapter, you should be familiar with property attributes. If you aren't, check out §9 "Property attributes: an introduction".

12.1 How enumerability affects property-iterating constructs

To demonstrate how various operations are affected by enumerability, we use the following object `obj` whose prototype is `proto`.

```
const protoEnumSymbolKey = Symbol('protoEnumSymbolKey');
const protoNonEnumSymbolKey = Symbol('protoNonEnumSymbolKey');
const proto = Object.defineProperties({}, {
  protoEnumStringKey: {
    value: 'protoEnumStringKeyValue',
    enumerable: true,
  },
  [protoEnumSymbolKey]: {
    value: 'protoEnumSymbolKeyValue',
    enumerable: true,
  },
  protoNonEnumStringKey: {
    value: 'protoNonEnumStringKeyValue',
    enumerable: false,
  },
  [protoNonEnumSymbolKey]: {
    value: 'protoNonEnumSymbolKeyValue',
    enumerable: false,
  },
});

const objEnumSymbolKey = Symbol('objEnumSymbolKey');
const objNonEnumSymbolKey = Symbol('objNonEnumSymbolKey');
const obj = Object.create(proto, {
  objEnumStringKey: {
    value: 'objEnumStringKeyValue',
    enumerable: true,
  },
  [objEnumSymbolKey]: {
    value: 'objEnumSymbolKeyValue',
    enumerable: true,
  },
  objNonEnumStringKey: {
    value: 'objNonEnumStringKeyValue',
    enumerable: false,
  },
  [objNonEnumSymbolKey]: {
    value: 'objNonEnumSymbolKeyValue',
    enumerable: false,
  },
});
```

12.1.1 Operations that only consider enumerable properties

Operation		String keys	Symbol keys	Inherited

Table 12.1: Operations that ignore non-enumerable properties.

Operation		String keys	Symbol keys	Inherited
`Object.keys()`	ES5	✔	✘	✘
`Object.values()`	ES2017	✔	✘	✘
`Object.entries()`	ES2017	✔	✘	✘
Spreading `{...x}`	ES2018	✔	✔	✘
`Object.assign()`	ES6	✔	✔	✘
`JSON.stringify()`	ES5	✔	✘	✘
`for-in`	ES1	✔	✘	✔

The following operations (summarized in tbl. 12.1) only consider enumerable properties:

- `Object.keys()` [ES5] returns the keys of enumerable own string-keyed properties.

  ```
  > Object.keys(obj)
  [ 'objEnumStringKey' ]
  ```

- `Object.values()` [ES2017] returns the values of enumerable own string-keyed properties.

  ```
  > Object.values(obj)
  [ 'objEnumStringKeyValue' ]
  ```

- `Object.entries()` [ES2017] returns key-value pairs for enumerable own string-keyed properties. (Note that `Object.fromEntries()` does accept symbols as keys, but only creates enumerable properties.)

  ```
  > Object.entries(obj)
  [ [ 'objEnumStringKey', 'objEnumStringKeyValue' ] ]
  ```

- Spreading into object literals [ES2018] only considers own enumerable properties (with string keys or symbol keys).

  ```
  > const copy = {...obj};
  > Reflect.ownKeys(copy)
  [ 'objEnumStringKey', objEnumSymbolKey ]
  ```

- `Object.assign()` [ES6] only copies enumerable own properties (with either string keys or symbol keys).

  ```
  > const copy = Object.assign({}, obj);
  > Reflect.ownKeys(copy)
  [ 'objEnumStringKey', objEnumSymbolKey ]
  ```

- `JSON.stringify()` [ES5] only stringifies enumerable own properties with string keys.

  ```
  > JSON.stringify(obj)
  '{"objEnumStringKey":"objEnumStringKeyValue"}'
  ```

- `for-in` loop [ES1] traverses the keys of own and inherited enumerable string-keyed properties.

  ```
  const propKeys = [];
  for (const propKey in obj) {
    propKeys.push(propKey);
  }
  assert.deepEqual(
    propKeys, ['objEnumStringKey', 'protoEnumStringKey']);
  ```

`for-in` is the only built-in operation where enumerability matters for inherited properties. All other operations only work with own properties.

12.1.2 Operations that consider both enumerable and non-enumerable properties

Table 12.2: Operations that consider both enumerable and non-enumerable properties.

Operation		Str. keys	Sym. keys	Inherited
`Object.getOwnPropertyNames()`	ES5	✔	✘	✘
`Object.getOwnPropertySymbols()`	ES6	✘	✔	✘
`Reflect.ownKeys()`	ES6	✔	✔	✘
`Object.getOwnPropertyDescriptors()`	ES2017	✔	✔	✘

The following operations (summarized in tbl. 12.2) consider both enumerable and non-enumerable properties:

- `Object.getOwnPropertyNames()` [ES5] lists the keys of all own string-keyed properties.

  ```
  > Object.getOwnPropertyNames(obj)
  [ 'objEnumStringKey', 'objNonEnumStringKey' ]
  ```

- `Object.getOwnPropertySymbols()` [ES6] lists the keys of all own symbol-keyed properties.

  ```
  > Object.getOwnPropertySymbols(obj)
  [ objEnumSymbolKey, objNonEnumSymbolKey ]
  ```

- `Reflect.ownKeys()` [ES6] lists the keys of all own properties.

  ```
  > Reflect.ownKeys(obj)
  [
    'objEnumStringKey',
    'objNonEnumStringKey',
    objEnumSymbolKey,
    objNonEnumSymbolKey
  ]
  ```

- `Object.getOwnPropertyDescriptors()` [ES2017] lists the property descriptors of all own properties.

  ```
  > Object.getOwnPropertyDescriptors(obj)
  {
    objEnumStringKey: {
      value: 'objEnumStringKeyValue',
      writable: false,
      enumerable: true,
      configurable: false
    },
    objNonEnumStringKey: {
      value: 'objNonEnumStringKeyValue',
      writable: false,
      enumerable: false,
      configurable: false
    },
    [objEnumSymbolKey]: {
      value: 'objEnumSymbolKeyValue',
      writable: false,
      enumerable: true,
      configurable: false
    },
    [objNonEnumSymbolKey]: {
      value: 'objNonEnumSymbolKeyValue',
      writable: false,
      enumerable: false,
      configurable: false
    }
  }
  ```

12.1.3 Naming rules for introspective operations

Introspection enables a program to examine the structure of values at runtime. It is *metaprogramming*: Normal programming is about writing programs; metaprogramming is about examining and/or changing programs.

In JavaScript, common introspective operations have short names, while rarely used operations have long names. Ignoring non-enumerable properties is the norm, which is why operations that do that have short names and operations that don't, long names:

- `Object.keys()` ignores non-enumerable properties.
- `Object.getOwnPropertyNames()` lists the string keys of all own properties.

However, `Reflect` methods (such as `Reflect.ownKeys()`) deviate from this rule because `Reflect` provides operations that are more "meta" and related to Proxies.

Additionally, the following distinction is made (since ES6, which introduced symbols):

- *Property keys* are either strings or symbols.
- *Property names* are property keys that are strings.

- *Property symbols* are property keys that are symbols.

Therefore, a better name for `Object.keys()` would now be `Object.names()`.

12.2 The enumerability of pre-defined and created properties

In this section, we'll abbreviate `Object.getOwnPropertyDescriptor()` like this:

```
const desc = Object.getOwnPropertyDescriptor.bind(Object);
```

Most data properties are created with the following attributes:

```
{
  writable: true,
  enumerable: false,
  configurable: true,
}
```

That includes:

- Assignment
- Object literals
- Public class fields
- `Object.fromEntries()`

The most important non-enumerable properties are:

- Prototype properties of built-in classes

  ```
  > desc(Object.prototype, 'toString').enumerable
  false
  ```

- Prototype properties created via user-defined classes

  ```
  > desc(class {foo() {}}.prototype, 'foo').enumerable
  false
  ```

- Property `.length` of Arrays:

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

- Property `.length` of strings (note that all properties of primitive values are read-only):

  ```
  > Object.getOwnPropertyDescriptor('', 'length')
  { value: 0, writable: false, enumerable: false, configurable: false }
  ```

We'll look at the use cases for enumerability next, which will tell us why some properties are enumerable and others aren't.

12.3 Use cases for enumerability

Enumerability is an inconsistent feature. It does have use cases, but there is always some kind of caveat. In this section, we look at the use cases and the caveats.

12.3.1 Use case: Hiding properties from the `for-in` loop

The `for-in` loop traverses *all* enumerable string-keyed properties of an object, own and inherited ones. Therefore, the attribute `enumerable` is used to hide properties that should not be traversed. That was the reason for introducing enumerability in ECMAScript 1.

In general, it is best to avoid `for-in`. The next two subsections explain why. The following function will help us demonstrate how `for-in` works.

```
function listPropertiesViaForIn(obj) {
  const result = [];
  for (const key in obj) {
    result.push(key);
  }
  return result;
}
```

12.3.1.1 The caveats of using `for-in` for objects

`for-in` iterates over all properties, including inherited ones:

```
const proto = {enumerableProtoProp: 1};
const obj = {
  __proto__: proto,
  enumerableObjProp: 2,
};
assert.deepEqual(
  listPropertiesViaForIn(obj),
  ['enumerableObjProp', 'enumerableProtoProp']);
```

With normal plain objects, `for-in` doesn't see inherited methods such as `Object.prototype.toString()` because they are all non-enumerable:

```
const obj = {};
assert.deepEqual(
  listPropertiesViaForIn(obj),
  []);
```

In user-defined classes, all inherited properties are also non-enumerable and therefore ignored:

```
class Person {
  constructor(first, last) {
    this.first = first;
    this.last = last;
  }
  getName() {
      return this.first + ' ' + this.last;
    }
  }
  const jane = new Person('Jane', 'Doe');
  assert.deepEqual(
    listPropertiesViaForIn(jane),
    ['first', 'last']);
```

Conclusion: In objects, `for-in` considers inherited properties and we usually want to ignore those. Then it is better to combine a `for-of` loop with `Object.keys()`, `Object.entries()`, etc.

12.3.1.2 The caveats of using `for-in` for Arrays

The own property `.length` is non-enumerable in Arrays and strings and therefore ignored by `for-in`:

```
> listPropertiesViaForIn(['a', 'b'])
[ '0', '1' ]
> listPropertiesViaForIn('ab')
[ '0', '1' ]
```

However, it is generally not safe to use `for-in` to iterate over the indices of an Array because it considers both inherited and own properties that are not indices. The following example demonstrate what happens if an Array has an own non-index property:

```
const arr1 = ['a', 'b'];
assert.deepEqual(
  listPropertiesViaForIn(arr1),
  ['0', '1']);

const arr2 = ['a', 'b'];
arr2.nonIndexProp = 'yes';
assert.deepEqual(
  listPropertiesViaForIn(arr2),
  ['0', '1', 'nonIndexProp']);
```

Conclusion: `for-in` should not be used for iterating over the indices of an Array because it considers both index properties and non-index properties:

- If you are interested in the keys of an Array, use the Array method `.keys()`:

  ```
  > [...['a', 'b', 'c'].keys()]
  [ 0, 1, 2 ]
  ```

- If you want to iterate over the elements of an Array, use a `for-of` loop, which has the added benefit of also working with other iterable data structures.

12.3.2 Use case: Marking properties as not to be copied

By making properties non-enumerable, we can hide them from some copying operations. Let us first examine two historical copying operations before moving on to more modern copying operations.

12.3.2.1 Historical copying operation: Prototype's `Object.extend()`

Prototype is a JavaScript framework that was created by Sam Stephenson in February 2005 as part of the foundation for Ajax support in Ruby on Rails.

Prototype's `Object.extend(destination, source)` copies all enumerable own and inherited properties of `source` into own properties of `destination`. It is implemented as follows:

```
function extend(destination, source) {
  for (var property in source)
    destination[property] = source[property];
  return destination;
}
```

If we use `Object.extend()` with an object, we can see that it copies inherited properties into own properties and ignores non-enumerable properties (it also ignores symbol-keyed properties). All of this is due to how `for-in` works.

```
const proto = Object.defineProperties({}, {
  enumProtoProp: {
    value: 1,
    enumerable: true,
  },
  nonEnumProtoProp: {
    value: 2,
    enumerable: false,
  },
});
const obj = Object.create(proto, {
  enumObjProp: {
    value: 3,
    enumerable: true,
  },
  nonEnumObjProp: {
    value: 4,
    enumerable: false,
  },
});

assert.deepEqual(
  extend({}, obj),
  {enumObjProp: 3, enumProtoProp: 1});
```

12.3.2.2 Historical copying operation: jQuery's `$.extend()`

jQuery's `$.extend(target, source1, source2, ···)` works similar to `Object.extend()`:

- It copies all enumerable own and inherited properties of `source1` into own properties of `target`.
- Then it does the same with `source2`.
- Etc.

12.3.2.3 The downsides of enumerability-driven copying

Basing copying on enumerability has several downsides:

- While enumerability is useful for hiding inherited properties, it is mainly used in this manner because we usually only want to copy own properties into own properties. The same effect can be better achieved by ignoring inherited properties.
- Which properties to copy often depends on the task at hand; it rarely makes sense to have a single flag for all use cases. A better choice is to provide a copying operation with a *predicate* (a callback that returns a boolean) that tells it when to ignore properties.
- Enumerability conveniently hides the own property `.length` of Arrays when copying. But that is an incredibly rare exceptional case: a magic property that both influences sibling properties and is influenced by them. If we implement this kind of magic ourselves, we will use (inherited) getters and/or setters, not (own) data properties.

12.3.2.4 `Object.assign()` [ES5]

In ES6, `Object.assign(target, source_1, source_2, ···)` can be used to merge the sources into the target. All own enumerable properties of the sources are considered (with either string keys or symbol keys). `Object.assign()` uses a "get" operation to read a value from a source and a "set" operation to write a value to the target.

With regard to enumerability, `Object.assign()` continues the tradition of `Object.extend()` and `$.extend()`. Quoting Yehuda Katz:

> `Object.assign` would pave the cowpath of all of the `extend()` APIs already in circulation. We thought the precedent of not copying enumerable methods in those cases was enough reason for `Object.assign` to have this behavior.

In other words: `Object.assign()` was created with an upgrade path from `$.extend()` (and similar) in mind. Its approach is cleaner than `$.extend`'s because it ignores inherited properties.

12.3.2.5 A rare example of non-enumerability being useful when copying

Cases where non-enumerability helps are few. One rare example is a recent issue that the liberary `fs-extra` had:

- The built-in Node.js module `fs` has a property `.promises` that contains an object with a Promise-based version of the `fs` API. At the time of the issue, reading `.promise` led to the following warning being logged to the console:

```
ExperimentalWarning: The fs.promises API is experimental
```

- In addition to providing its own functionality, `fs-extra` also re-exports everything that's in `fs`. For CommonJS modules, that means copying all properties of `fs` into the `module.exports` of `fs-extra` (via `Object.assign()`). And when `fs-extra` did that, it triggered the warning. That was confusing because it happened every time `fs-extra` was loaded.

- A quick fix was to make property `fs.promises` non-enumerable. Afterwards, `fs-extra` ignored it.

12.3.3 Marking properties as private

If we make a property non-enumerable, it can't by seen by `Object.keys()`, the `for-in` loop, etc., anymore. With regard to those mechanisms, the property is private.

However, there are several problems with this approach:

- When copying an object, we normally want to copy private properties. That clashes with making properties non-enumerable that shouldn't be copied (see previous section).
- The property isn't really private. Getting, setting and several other mechanisms make no distinction between enumerable and non-enumerable properties.
- When working with code either as source or interactively, we can't immediately see whether a property is enumerable or not. A naming convention (such as prefixing property names with an underscore) is easier to discover.
- We can't use enumerability to distinguish between public and private methods because methods in prototypes are non-enumerable by default.

12.3.4 Hiding own properties from `JSON.stringify()`

`JSON.stringify()` does not include properties in its output that are non-enumerable. We can therefore use enumerability to determine which own properties should be exported to JSON. This use case is similar to the previous one, marking properties as private. But it is also different because this is more about exporting and slightly different considerations apply. For example: Can an object be completely reconstructed from JSON?

As an alternative to enumerability, an object can implement the method `.toJSON()` and `JSON.stringify()` stringifies whatever that method returns, instead of the object itself. The next example demonstrates how that works.

```
class Point {
  static fromJSON(json) {
    return new Point(json[0], json[1]);
  }
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  toJSON() {
    return [this.x, this.y];
  }
}
```

```
assert.equal(
  JSON.stringify(new Point(8, -3)),
  '[8,-3]'
);
```

I find `toJSON()` cleaner than enumerability. It also gives us more freedom w.r.t. what the storage format should look like.

12.4 Conclusion

We have seen that almost all applications for non-enumerability are work-arounds that now have other and better solutions.

For our own code, we can usually pretend that enumerability doesn't exist:

- Creating properties via object literals and assignment always creates enumerable properties.
- Prototype properties created via classes are always non-enumerable.

That is, we automatically follow best practices.

Part V

OOP: techniques

Chapter 13

Techniques for instantiating classes

Contents

In this chapter, we examine several approaches for creating instances of classes: Constructors, factory functions, etc. We do so by solving one concrete problem several times. The focus of this chapter is on classes, which is why alternatives to classes are ignored.

13.1 The problem: initializing a property asynchronously

The following container class is supposed to receive the contents of its property `.data` asynchronously. This is our first attempt:

```
class DataContainer {
  #data; // (A)
  constructor() {
    Promise.resolve('downloaded')
```

```
        .then(data => this.#data = data); // (B)
    }
    getData() {
      return 'DATA: '+this.#data; // (C)
    }
  }
```

Key issue of this code: Property `.data` is initially `undefined`.

```
const dc = new DataContainer();
assert.equal(dc.getData(), 'DATA: undefined');
setTimeout(() => assert.equal(
  dc.getData(), 'DATA: downloaded'), 0);
```

In line A, we declare the private field `.#data` that we use in line B and line C.

The Promise inside the constructor of `DataContainer` is settled asynchronously, which is why we can only see the final value of `.data` if we finish the current task and start a new one, via `setTimeout()`. In other words, the instance of `DataContainer` is not completely initialized, yet, when we first see it.

13.2 Solution: Promise-based constructor

What if we delay access to the instance of `DataContainer` until it is fully initialized? We can achieve that by returning a Promise from the constructor. By default, a constructor returns a new instance of the class that it is part of. We can override that if we explicitly return an object:

```
class DataContainer {
  #data;
  constructor() {
    return Promise.resolve('downloaded')
      .then(data => {
        this.#data = data;
        return this; // (A)
      });
  }
  getData() {
    return 'DATA: '+this.#data;
  }
}
new DataContainer()
  .then(dc => assert.equal( // (B)
    dc.getData(), 'DATA: downloaded'));
```

Now we have to wait until we can access our instance (line B). It is passed on to us after the data is "downloaded" (line A). There are two possible sources of errors in this code:

- The download may fail and produce a rejection.
- An exception may be thrown in the body of the first `.then()` callback.

In either case, the errors become rejections of the Promise that is returned from the constructor.

Pros and cons:

- A benefit of this approach is that we can only access the instance once it is fully initialized. And there is no other way of creating instances of `DataContainer`.
- A disadvantage is that it may be surprising to have a constructor return a Promise instead of an instance.

13.2.1 Using an immediately-invoked asynchronous arrow function

Instead of using the Promise API directly to create the Promise that is returned from the constructor, we can also use an asynchronous arrow function that we invoke immediately:

```
constructor() {
  return (async () => {
    this.#data = await Promise.resolve('downloaded');
    return this;
  })();
}
```

13.3 Solution: static factory method

A *static factory method* of a class `C` creates instances of `C` and is an alternative to using `new C()`. Common names for static factory methods in JavaScript:

- `.create()`: Creates a new instance. Example: `Object.create()`
- `.from()`: Creates a new instance based on a different object, by copying and/or converting it. Example: `Array.from()`
- `.of()`: Creates a new instance by assembling values specified via arguments. Example: `Array.of()`

In the following example, `DataContainer.create()` is a static factory method. It returns Promises for instances of `DataContainer`:

```
class DataContainer {
  #data;
  static async create() {
    const data = await Promise.resolve('downloaded');
    return new this(data);
  }
  constructor(data) {
    this.#data = data;
  }
  getData() {
    return 'DATA: '+this.#data;
  }
}
```

```
DataContainer.create()
  .then(dc => assert.equal(
    dc.getData(), 'DATA: downloaded'));
```

This time, all asynchronous functionality is contained in `.create()`, which enables the rest of the class to be completely synchronous and therefore simpler.

Pros and cons:

- A benefit of this approach is that the constructor becomes simple.
- A disadvantage of this approach is that it's now possible to create instances that are incorrectly set up, via `new DataContainer()`.

13.3.1 Improvement: private constructor via secret token

If we want to ensure that instances are always correctly set up, we must ensure that only `DataContainer.create()` can invoke the constructor of `DataContainer`. We can achieve that via a secret token:

```
const secretToken = Symbol('secretToken');
class DataContainer {
  #data;
  static async create() {
    const data = await Promise.resolve('downloaded');
    return new this(secretToken, data);
  }
  constructor(token, data) {
    if (token !== secretToken) {
      throw new Error('Constructor is private');
    }
    this.#data = data;
  }
  getData() {
    return 'DATA: '+this.#data;
  }
}
DataContainer.create()
  .then(dc => assert.equal(
    dc.getData(), 'DATA: downloaded'));
```

If `secretToken` and `DataContainer` reside in the same module and only the latter is exported, then outside parties don't have access to `secretToken` and therefore can't create instances of `DataContainer`.

Pros and cons:

- Benefit: safe and straightforward.
- Disadvantage: slightly verbose.

13.3.2 Improvement: constructor throws, factory method borrows the class prototype

The following variant of our solution disables the constructor of `DataContainer` and uses a trick to create instances of it another way (line A):

```
class DataContainer {
  static async create() {
    const data = await Promise.resolve('downloaded');
    return Object.create(this.prototype)._init(data); // (A)
  }
  constructor() {
    throw new Error('Constructor is private');
  }
  _init(data) {
    this._data = data;
    return this;
  }
  getData() {
    return 'DATA: '+this._data;
  }
}
DataContainer.create()
  .then(dc => {
    assert.equal(dc instanceof DataContainer, true); // (B)
    assert.equal(
      dc.getData(), 'DATA: downloaded');
  });
```

Internally, an instance of `DataContainer` is any object whose prototype is `DataContainer.prototype`. That's why we can create instances via `Object.create()` (line A) and that's why `instanceof` works in line B.

Pros and cons:

- Benefit: elegant; `instanceof` works.
- Disadvantages:
  - Creating instances is not completely prevented. To be fair, though, the work-around via `Object.create()` can also be used for our previous solutions.
  - We can't use private fields and private methods in `DataContainer`, because those are only set up correctly for instances that were created via the constructor.

13.3.3 Improvement: instances are inactive by default, activated by factory method

Another, more verbose variant is that, by default, instances are switched off via the flag `.#active`. The initialization method `.#init()` that switches them on cannot be accessed externally, but `Data.container()` can invoke it:

```
class DataContainer {
  #data;
  static async create() {
    const data = await Promise.resolve('downloaded');
    return new this().#init(data);
  }

  #active = false;
  constructor() {
  }
  #init(data) {
    this.#active = true;
    this.#data = data;
    return this;
  }
  getData() {
    this.#check();
    return 'DATA: '+this.#data;
  }
  #check() {
    if (!this.#active) {
      throw new Error('Not created by factory');
    }
  }
}
DataContainer.create()
  .then(dc => assert.equal(
    dc.getData(), 'DATA: downloaded'));
```

The flag `.#active` is enforced via the private method `.#check()` which must be invoked at the beginning of every method.

The major downside of this solution is its verbosity. There is also a risk of forgetting to invoke `.#check()` in each method.

13.3.4 Variant: separate factory function

For completeness sake, I'll show another variant: Instead of using a static method as a factory you can also use a separate stand-alone function.

```
const secretToken = Symbol('secretToken');
class DataContainer {
  #data;
  constructor(token, data) {
    if (token !== secretToken) {
      throw new Error('Constructor is private');
    }
    this.#data = data;
  }
  getData() {
      return 'DATA: '+this.#data;
    }
  }

  async function createDataContainer() {
    const data = await Promise.resolve('downloaded');
    return new DataContainer(secretToken, data);
  }

  createDataContainer()
    .then(dc => assert.equal(
      dc.getData(), 'DATA: downloaded'));
```

Stand-alone functions as factories are occasionally useful, but in this case, I prefer a static method:

- The stand-alone function can't access private members of `DataContainer`.
- I prefer the way `DataContainer.create()` looks.

13.4 Subclassing a Promise-based constructor (optional)

In general, subclassing is something to use sparingly.

With a separate factory function, it is relatively easy to extend `DataContainer`.

Alas, extending the class with the Promise-based constructor leads to severe limitations. In the following example, we subclass `DataContainer`. The subclass `SubDataContainer` has its own private field `.#moreData` that it initializes asynchronously by hooking into the Promise returned by the constructor of its superclass.

```
class DataContainer {
  #data;
  constructor() {
    return Promise.resolve('downloaded')
      .then(data => {
        this.#data = data;
        return this; // (A)
      });
  }
  getData() {
    return 'DATA: '+this.#data;
  }
}

class SubDataContainer extends DataContainer {
  #moreData;
  constructor() {
    super();
    const promise = this;
    return promise
```

```
        .then(_this => {
          return Promise.resolve('more')
            .then(moreData => {
              _this.#moreData = moreData;
              return _this;
            });
        });
    }
    getData() {
      return super.getData() + ', ' + this.#moreData;
    }
  }
```

Alas, we can't instantiate this class:

```
  assert.rejects(
    () => new SubDataContainer(),
    {
      name: 'TypeError',
      message: 'Cannot write private member #moreData ' +
        'to an object whose class did not declare it',
    }
  );
```

Why the failure? A constructor always adds its private fields to its `this`. However, here, `this` in the subconstructor is the Promise returned by the superconstructor (and not the instance of `SubDataContainer` delivered via the Promise).

However, this approach still works if `SubDataContainer` does not have any private fields.

13.5 Conclusion

For the scenario examined in this chapter, I prefer either a Promise-based constructor or a static factory method plus a private constructor via a secret token.

However, the other techniques presented here can still be useful in other scenarios.

13.6 Further reading

- Asynchronous programming:
  - Chapter "Promises for asynchronous programming" in "JavaScript for impatient programmers"
  - Chapter "Async functions" in "JavaScript for impatient programmers"
  - Section "Immediately invoked async arrow functions" in "JavaScript for impatient programmers"
- OOP:
  - Chapter "Prototype chains and classes" in "JavaScript for impatient programmers"
  - Blog post "ES proposal: private class fields"

  – Blog post "ES proposal: private methods and accessors in JavaScript classes"

Chapter 14

Copying instances of classes: `.clone()` vs. copy constructors

Contents

In this chapter, we look at two techniques for implementing copying for class instances:

- `.clone()` methods
- So-called *copy constructors*, constructors that receive another instance of the current class and use it to initialize the current instance.

14.1 `.clone()` methods

This technique introduces one method `.clone()` per class whose instances are to be copied. It returns a deep copy of `this`. The following example shows three classes that can be cloned.

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  clone() {
    return new Point(this.x, this.y);
  }
}
class Color {
  constructor(name) {
      this.name = name;
    }
    clone() {
      return new Color(this.name);
    }
  }
  class ColorPoint extends Point {
    constructor(x, y, color) {
      super(x, y);
      this.color = color;
    }
    clone() {
      return new ColorPoint(
        this.x, this.y, this.color.clone()); // (A)
    }
  }
```

Line A demonstrates an important aspect of this technique: compound instance property values must also be cloned, recursively.

14.2 Static factory methods

A *copy constructor* is a constructor that uses another instance of the current class to set up the current instance. Copy constructors are popular in static languages such as C++ and Java, where you can provide multiple versions of a constructor via *static overloading*. Here, *static* means that the choice which version to use, is made at compile time.

In JavaScript, we must make that decision at runtime and that leads to inelegant code:

```
class Point {
  constructor(...args) {
    if (args[0] instanceof Point) {
      // Copy constructor
      const [other] = args;
      this.x = other.x;
      this.y = other.y;
    } else {
      const [x, y] = args;
      this.x = x;
      this.y = y;
    }
  }
}
```

This is how you'd use this class:

```
const original = new Point(-1, 4);
const copy = new Point(original);
assert.deepEqual(copy, original);
```

Static factory methods are an alternative to constructors and work better in this case because we can directly invoke the desired functionality. (Here, *static* means that these factory methods are class methods.)

In the following example, the three classes `Point`, `Color` and `ColorPoint` each have a static factory method `.from()`:

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  static from(other) {
    return new Point(other.x, other.y);
  }
}
class Color {
  constructor(name) {
    this.name = name;
  }
  static from(other) {
    return new Color(other.name);
  }
}
class ColorPoint extends Point {
  constructor(x, y, color) {
    super(x, y);
    this.color = color;
  }
  static from(other) {
    return new ColorPoint(
      other.x, other.y, Color.from(other.color)); // (A)
  }
}
```

In line A, we once again copy recursively.

This is how `ColorPoint.from()` works:

```
const original = new ColorPoint(-1, 4, new Color('red'));
const copy = ColorPoint.from(original);
assert.deepEqual(copy, original);
```

14.3 Acknowledgements

- Ron Korvig reminded me to use static factory methods and not overloaded constructors for deep-copying in JavaScript.

Chapter 15

Immutable wrappers for collections

Contents

An immutable wrapper for a collection makes that collection immutable by wrapping it in a new object. In this chapter, we examine how that works and why it is useful.

15.1 Wrapping objects

If there is an object whose interface we'd like to reduce, we can take the following approach:

- Create a new object that stores the original in a private field. The new object is said to *wrap* the original object. The new object is called *wrapper*, the original object *wrapped object*.
- The wrapper only forwards some of the method calls it receives to the wrapped object.

This is what wrapping looks like:

```
class Wrapper {
  #wrapped;
  constructor(wrapped) {
    this.#wrapped = wrapped;
  }
  allowedMethod1(...args) {
    return this.#wrapped.allowedMethod1(...args);
    }
    allowedMethod2(...args) {
      return this.#wrapped.allowedMethod2(...args);
    }
  }
```

Related software design patterns:

- Wrapping is related to the Gang of Four design pattern *Facade*.
- We used *forwarding* to implement *delegation*. Delegation means that one object lets another object (the *delegate*) handle some of its work. It is an alternative to inheritance for sharing code.

15.1.1 Making collections immutable via wrapping

To make a collection immutable, we can use wrapping and remove all destructive operations from its interface.

One important use case for this technique is an object that has an internal mutable data structure that it wants to export safely without copying it. The export being "live" may also be a goal. The object can achieve its goals by wrapping the internal data structure and making it immutable.

The next two sections showcase immutable wrappers for Maps and Arrays. They both have the following limitations:

- They are sketches. More work is needed to make them suitable for practical use: Better checks, support for more methods, etc.
- They work shallowly: Each one makes the wrapped object immutable, but not the data it returns. This could be fixed by wrapping some of the results that are returned by methods.

15.2 An immutable wrapper for Maps

Class `ImmutableMapWrapper` produces wrappers for Maps:

```
  class ImmutableMapWrapper {
    static _setUpPrototype() {
      // Only forward non-destructive methods to the wrapped Map:
      for (const methodName of ['get', 'has', 'keys', 'size']) {
        ImmutableMapWrapper.prototype[methodName] = function (...args) {
          return this.#wrappedMap[methodName](...args);
        }
      }
    }

    #wrappedMap;
    constructor(wrappedMap) {
      this.#wrappedMap = wrappedMap;
    }
```

```
}
ImmutableMapWrapper._setUpPrototype();
```

The setup of the prototype has to be performed by a static method, because we only have access to the private field `.#wrappedMap` from inside the class.

This is `ImmutableMapWrapper` in action:

```
const map = new Map([[false, 'no'], [true, 'yes']]);
const wrapped = new ImmutableMapWrapper(map);

// Non-destructive operations work as usual:
assert.equal(
  wrapped.get(true), 'yes');
assert.equal(
  wrapped.has(false), true);
assert.deepEqual(
  [...wrapped.keys()], [false, true]);

// Destructive operations are not available:
assert.throws(
  () => wrapped.set(false, 'never!'),
  /^TypeError: wrapped.set is not a function$/);
assert.throws(
  () => wrapped.clear(),
  /^TypeError: wrapped.clear is not a function$/);
```

15.3 An immutable wrapper for Arrays

For an Array `arr`, normal wrapping is not enough because we need to intercept not just method calls, but also property accesses such as `arr[1] = true`. JavaScript proxies enable us to do this:

```
const RE_INDEX_PROP_KEY = /^[0-9]+$/;
const ALLOWED_PROPERTIES = new Set([
  'length', 'constructor', 'slice', 'concat']);

function wrapArrayImmutably(arr) {
  const handler = {
    get(target, propKey, receiver) {
      // We assume that propKey is a string (not a symbol)
      if (RE_INDEX_PROP_KEY.test(propKey) // simplified check!
        || ALLOWED_PROPERTIES.has(propKey)) {
          return Reflect.get(target, propKey, receiver);
      }
      throw new TypeError(`Property "${propKey}" can’t be accessed`);
    },
    set(target, propKey, value, receiver) {
      throw new TypeError('Setting is not allowed');
      },
      deleteProperty(target, propKey) {
        throw new TypeError('Deleting is not allowed');
      },
    };
    return new Proxy(arr, handler);
  }
```

Let’s wrap an Array:

```
  const arr = ['a', 'b', 'c'];
  const wrapped = wrapArrayImmutably(arr);

  // Non-destructive operations are allowed:
  assert.deepEqual(
    wrapped.slice(1), ['b', 'c']);
  assert.equal(
    wrapped[1], 'b');

  // Destructive operations are not allowed:
  assert.throws(
    () => wrapped[1] = 'x',
    /^TypeError: Setting is not allowed$/);
  assert.throws(
    () => wrapped.shift(),
    /^TypeError: Property "shift" can’t be accessed$/);
```

Part VI

Regular expressions

Chapter 16

Regular expressions: lookaround assertions by example

Contents

In this chapter we use examples to explore lookaround assertions in regular expressions. A lookaround assertion is non-capturing and must match (or not match) what comes before (or after) the current location in the input string.

16.1 Cheat sheet: lookaround assertions

Table 16.1: Overview of available lookaround assertions.

Pattern	Name	
`(?=«pattern»)`	Positive lookahead	ES3
`(?!«pattern»)`	Negative lookahead	ES3
`(?<=«pattern»)`	Positive lookbehind	ES2018
`(?<!«pattern»)`	Negative lookbehind	ES2018

There are four lookaround assertions (tbl. 16.1)

- Lookahead assertions (ECMAScript 3):
  - Positive lookahead: `(?=«pattern»)` matches if `pattern` matches what comes after the current location in the input string.
  - Negative lookahead: `(?!«pattern»)` matches if `pattern` does not match what comes after the current location in the input string.
- Lookbehind assertions (ECMAScript 2018):
  - Positive lookbehind: `(?<=«pattern»)` matches if `pattern` matches what comes before the current location in the input string.
  - Negative lookbehind: `(?<!«pattern»)` matches if `pattern` does not match what comes before the current location in the input string.

16.2 Warnings for this chapter

- The examples show what can be achieved via lookaround assertions. However, regular expression aren't always the best solution. Another technique, such as proper parsing, may be a better choice.
- Lookbehind assertions are a relatively new feature that may not be supported by all JavaScript engines you are targeting.
- Lookaround assertions may affect performance negatively, especially if their patterns match long strings.

16.3 Example: Specifying what comes before or after a match (positive lookaround)

In the following interaction, we extract quoted words:

```
> 'how "are" "you" doing'.match(/(?<=")[a-z]+(?=")/g)
[ 'are', 'you' ]
```

Two lookaround assertions help us here:

- `(?<=")` "must be preceded by a quote"
- `(?=")` "must be followed by a quote"

Lookaround assertions are especially convenient for `.match()` in `/g` mode, which returns whole matches (capture group 0). Whatever the pattern of a lookaround assertion matches is not captured. Without lookaround assertions, the quotes show up in the result:

```
> 'how "are" "you" doing'.match(/"([a-z]+)"/g)
[ '"are"', '"you"' ]
```

16.4 Example: Specifying what does not come before or after a match (negative lookaround)

How can we achieve the opposite of what we did in the previous section and extract all unquoted words from a string?

- Input: `'how "are" "you" doing'`
- Output: `['how', 'doing']`

Our first attempt is to simply convert positive lookaround assertions to negative lookaround assertions. Alas, that fails:

```
> 'how "are" "you" doing'.match(/(?<!")[a-z]+(?!")/g)
[ 'how', 'r', 'o', 'doing' ]
```

The problem is that we extract sequences of characters that are not bracketed by quotes. That means that in the string `'"are"'`, the "r" in the middle is considered unquoted, because it is preceded by an "a" and followed by an "e".

We can fix this by stating that prefix and suffix must be neither quote nor letter:

```
> 'how "are" "you" doing'.match(/(?<!["a-z])[a-z]+(?!["a-z])/g)
[ 'how', 'doing' ]
```

Another solution is to demand via `\b` that the sequence of characters `[a-z]+` start and end at word boundaries:

```
> 'how "are" "you" doing'.match(/(?<!")\b[a-z]+\b(?!")/g)
[ 'how', 'doing' ]
```

One thing that is nice about negative lookbehind and negative lookahead is that they also work at the beginning or end, respectively, of a string – as demonstrated in the example.

16.4.1 There are no simple alternatives to negative lookaround assertions

Negative lookaround assertions are a powerful tool and usually impossible to emulate via other regular expression means.

If we don't want to use them, we normally have to take a completely different approach. For example, in this case, we could split the string into (quoted and unquoted) words and then filter those:

```
const str = 'how "are" "you" doing';

const allWords = str.match(/"?[a-z]+"?/g);
const unquotedWords = allWords.filter(
    w => !w.startsWith('"') || !w.endsWith('"'));
  assert.deepEqual(unquotedWords, ['how', 'doing']);
```

Benefits of this approach:

- It works on older engines.
- It is easy to understand.

16.5 Interlude: pointing lookaround assertions inward

All of the examples we have seen so far have in common that the lookaround assertions dictate what must come before or after the match but without including those characters in the match.

The regular expressions shown in the remainder of this chapter are different: Their lookaround assertions point inward and restrict what's inside the match.

16.6 Example: match strings not starting with `'abc'`

Let's assume we want to match all strings that do not start with `'abc'`. Our first attempt could be the regular expression `/^(?!abc)/`.

That works well for `.test()`:

```
> /^(?!abc)/.test('xyz')
true
```

However, `.exec()` gives us an empty string:

```
> /^(?!abc)/.exec('xyz')
{ 0: '', index: 0, input: 'xyz', groups: undefined }
```

The problem is that assertions such as lookaround assertions don't expand the matched text. That is, they don't capture input characters, they only make demands about the current location in the input.

Therefore, the solution is to add a pattern that does capture input characters:

```
> /^(?!abc).*$/.exec('xyz')
{ 0: 'xyz', index: 0, input: 'xyz', groups: undefined }
```

As desired, this new regular expression rejects strings that are prefixed with `'abc'`:

```
> /^(?!abc).*$/.exec('abc')
null
> /^(?!abc).*$/.exec('abcd')
null
```

And it accepts strings that don't have the full prefix:

```
> /^(?!abc).*$/.exec('ab')
{ 0: 'ab', index: 0, input: 'ab', groups: undefined }
```

16.7 Example: match substrings that do not contain '.mjs'

In the following example, we want to find

```
import ··· from '«module-specifier»';
```

where `module-specifier` does not end with `'.mjs'`.

```
const code = `
import {transform} from './util';
import {Person} from './person.mjs';
import {zip} from 'lodash';
`.trim();
assert.deepEqual(
  code.match(/^import .*? from '[^']+(?<!\.mjs)';$/umg),
  [
    "import {transform} from './util';",
    "import {zip} from 'lodash';",
  ]);
```

Here, the lookbehind assertion `(?<!\.mjs)` acts as a *guard* and prevents that the regular expression matches strings that contain `'.mjs`' at this location.

16.8 Example: skipping lines with comments

Scenario: We want to parse lines with settings, while skipping comments. For example:

```
const RE_SETTING = /^(?!#)([^:]*):(.*)$/

const lines = [
  'indent: 2', // setting
  '# Trim trailing whitespace:', // comment
  'whitespace: trim', // setting
];
for (const line of lines) {
  const match = RE_SETTING.exec(line);
  if (match) {
    const key = JSON.stringify(match[1]);
    const value = JSON.stringify(match[2]);
    console.log(`KEY: ${key} VALUE: ${value}`);
  }
}

// Output:
// 'KEY: "indent" VALUE: " 2"'
// 'KEY: "whitespace" VALUE: " trim"'
```

How did we arrive at the regular expression `RE_SETTING`?

We started with the following regular expression for settings:

```
/^([^:]*):(.*)$/
```

Intuitively, it is a sequence of the following parts:

- Start of the line
- Non-colons (zero or more)
- A single colon
- Any characters (zero or more)
- The end of line

This regular expression does reject *some* comments:

```
> /^([^:]*):(.*)$/.test('# Comment')
false
```

But it accepts others (that have colons in them):

```
> /^([^:]*):(.*)$/.test('# Comment:')
true
```

We can fix that by prefixing `(?!#)` as a guard. Intuitively, it means: "The current location in the input string must not be followed by the character `#`."

The new regular expression works as desired:

```
> /^(?!#)([^:]*):(.*)$/.test('# Comment:')
false
```

16.9 Example: smart quotes

Let's assume we want to convert pairs of straight double quotes to curly quotes:

- Input: `` `"yes" and "no"` ``
- Output: `` `“yes” and “no”` ``

This is our first attempt:

```
> `The words "must" and "should".`.replace(/"(.*)"/g, '“$1”')
'The words “must" and "should”.'
```

Only the first quote and the last quote is curly. The problem here is that the `*` quantifier matches *greedily* (as much as possible).

If we put a question mark after the `*`, it matches *reluctantly*:

```
> `The words "must" and "should".`.replace(/"(.*?)"/g, '“$1”')
'The words “must” and “should”.'
```

16.9.1 Supporting escaping via backslashes

What if we want to allow the escaping of quotes via backslashes? We can do that by using the guard `(?<!\\)` before the quotes:

```
> const regExp = /(?<!\\)"(.*?)(?<!\\)"/g;
> String.raw`\"straight\" and "curly"`.replace(regExp, '“$1”')
'\\"straight\\" and “curly”'
```

As a post-processing step, we would still need to do:

```
.replace(/\\"/g, `"`)
```

However, this regular expression can fail when there is a backslash-escaped backslash:

```
> String.raw`Backslash: "\\"`.replace(/(?<!\\)"(.*?)(?<!\\)"/g, '“$1”')
'Backslash: "\\\\"'
```

The second backslash prevented the quotes from becoming curly.

We can fix that if we make our guard more sophisticated (`?:` makes the group non-capturing):

```
(?<=[^\\](?:\\\\)*)
```

The new guard allows pairs of backslashes before quotes:

```
> const regExp = /(?<=[^\\](?:\\\\)*)"(.*?)(?<=[^\\](?:\\\\)*)"/g;
> String.raw`Backslash: "\\"`.replace(regExp, '“$1”')
'Backslash: “\\\\”'
```

One issue remains. This guard prevents the first quote from being matched if it appears at the beginning of a string:

```
> const regExp = /(?<=[^\\](?:\\\\)*)"(.*?)(?<=[^\\](?:\\\\)*)"/g;
> `"abc"`.replace(regExp, '“$1”')
'"abc"'
```

We can fix that by changing the first guard to: `(?<=[^\\](?:\\\\)*|^)`

```
> const regExp = /(?<=[^\\](?:\\\\)*|^)"(.*?)(?<=[^\\](?:\\\\)*)"/g;
> `"abc"`.replace(regExp, '“$1”')
'“abc”'
```

16.10 Acknowledgements

- The first regular expression that handles escaped backslashes in front of quotes was proposed by `@jonasraoni` on Twitter.

16.11 Further reading

- Chapter “Regular expressions (`RegExp`)” in “JavaScript for impatient programmers”

Chapter 17

Composing regular expressions via `re-template-tag`

Contents

My small library `re-template-tag` provides a template tag function for composing regular expressions. This chapter explains how it works. The intent is to introduce the ideas behind the library (more than the library itself).

17.1 The basics

The library implements the template tag `re` for regular expressions.

Installation:

```
npm install re-template-tag
```

Importing:

```
import {re} from 're-template-tag';
```

Use: The following two expressions produce the same regular expression.

```
assert.deepEqual(
  re`/abc/gu`,
  /abc/gu);
```

17.2 A tour of the features

The unit tests are a good tour of the features of re-template-tag.

17.2.1 Main features

You can use `re` to assemble a regular expression from fragments:

```
const RE_YEAR = /([0-9]{4})/;
const RE_MONTH = /([0-9]{2})/;
const RE_DAY = /([0-9]{2})/;
const RE_DATE = re`/^${RE_YEAR}-${RE_MONTH}-${RE_DAY}$/u`;
assert.equal(RE_DATE.source, '^([0-9]{4})-([0-9]{2})-([0-9]{2})$');
```

Any strings you insert are escaped properly:

```
assert.equal(re`/-${'.'}-/u`.source, '-\\.-');
```

Flag-less mode:

```
const regexp = re`abc`;
assert.ok(regexp instanceof RegExp);
assert.equal(regexp.source, 'abc');
assert.equal(regexp.flags, '');
```

17.2.2 Details and advanced features

You can use the backslash as you would inside regular expression literals:

```
assert.equal(re`/\./u`.source, '\\.');
```

Regular expression flags (such as `/u` in the previous example) can also be computed:

```
const regexp = re`/abc/${'g'+'u'}`;
assert.ok(regexp instanceof RegExp);
assert.equal(regexp.source, 'abc');
assert.equal(regexp.flags, 'gu');
```

17.3 Why is this useful?

- You can compose regular expressions from fragments and document the fragments via comments. That makes regular expressions easier to understand.
- You can define constants for regular expression fragments and reuse them.
- You can define plain text constants via strings and insert them into regular expressions and they are escaped as necessary.

17.4 `re` and named capture groups

`re` profits from named capture groups because they make the fragments more independent.

Without named groups:

```
const RE_YEAR = /([0-9]{4})/;
const RE_MONTH = /([0-9]{2})/;
const RE_DAY = /([0-9]{2})/;
const RE_DATE = re`/^${RE_YEAR}-${RE_MONTH}-${RE_DAY}$/u`;

const match = RE_DATE.exec('2017-01-27');
assert.equal(match[1], '2017');
```

With named groups:

```
const RE_YEAR = re`(?<year>[0-9]{4})`;
const RE_MONTH = re`(?<month>[0-9]{2})`;
const RE_DAY = re`(?<day>[0-9]{2})`;
const RE_DATE = re`/${RE_YEAR}-${RE_MONTH}-${RE_DAY}/u`;

const match = RE_DATE.exec('2017-01-27');
assert.equal(match.groups.year, '2017');
```

Part VII

Miscellaneous topics

Chapter 18

Exploring Promises by implementing them

Contents

Required knowledge: Promises

For this chapter, you should be roughly familiar with Promises, but much relevant knowledge is also reviewed here. If necessary, you can read the chapter on Promises in "JavaScript for impatient programmers".

In this chapter, we will approach Promises from a different angle: Instead of using this API, we will create a simple implementation of it. This different angle once helped me greatly with making sense of Promises.

The Promise implementation is the class `ToyPromise`. In order to be easier to understand, it doesn't completely match the API. But it is close enough to still give us much insight into how Promises work.

Repository with code

`ToyPromise` is available on GitHub, in the repository `toy-promise`.

18.1 Refresher: states of Promises

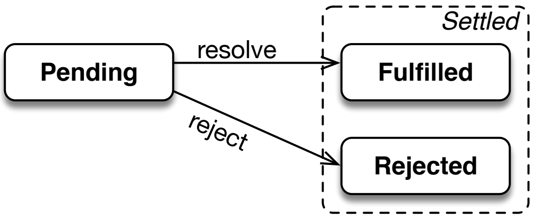

Figure 18.1: The states of a Promise (simplified version): A Promise is initially pending. If we resolve it, it becomes fulfilled. If we reject it, it becomes rejected.

We start with a simplified version of how Promise states work (fig. 18.1):

- A Promise is initially *pending*.
- If a Promise is *resolved* with a value `v`, it becomes *fulfilled* (later, we'll see that resolving can also reject). `v` is now the *fulfillment value* of the Promise.
- If a Promise is *rejected* with an error `e`, it becomes *rejected*. `e` is now the *rejection value* of the Promise.

18.2 Version 1: Stand-alone Promise

Our first implementation is a stand-alone Promise with minimal functionality:

- We can create a Promise.
- We can resolve or reject a Promise and we can only do it once.
- We can register *reactions* (callbacks) via `.then()`. Registering must do the right thing independently of whether the Promise has already been settled or not.
- `.then()` does not support chaining, yet – it does not return anything.

`ToyPromise1` is a class with three prototype methods:

- `ToyPromise1.prototype.resolve(value)`
- `ToyPromise1.prototype.reject(reason)`
- `ToyPromise1.prototype.then(onFulfilled, onRejected)`

That is, `resolve` and `reject` are methods (and not functions handed to a callback parameter of the constructor).

This is how this first implementation is used:

```
// .resolve() before .then()
const tp1 = new ToyPromise1();
tp1.resolve('abc');
tp1.then((value) => {
  assert.equal(value, 'abc');
});

// .then() before .resolve()
const tp2 = new ToyPromise1();
tp2.then((value) => {
  assert.equal(value, 'def');
});
tp2.resolve('def');
```

Fig. 18.2 illustrates how our first `ToyPromise` works.

The diagrams of the data flow in Promises are optional

The motivation for the diagrams is to have a visual explanation for how Promises work. But they are optional. If you find them confusing, you can ignore them and focus on the code.

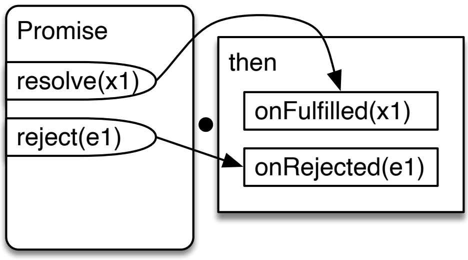

Figure 18.2: `ToyPromise1`: If a Promise is resolved, the provided value is passed on to the *fulfillment reactions* (first arguments of `.then()`). If a Promise is rejected, the provided value is passed on to the *rejection reactions* (second arguments of `.then()`).

18.2.1 Method `.then()`

Let's examine `.then()` first. It has to handle two cases:

- If the Promise is still pending, it queues invocations of `onFulfilled` and `onRejected`. They are to be used later, when the Promise is settled.
- If the Promise is already fulfilled or rejected, `onFulfilled` or `onRejected` can be invoked right away.

```
then(onFulfilled, onRejected) {
  const fulfillmentTask = () => {
    if (typeof onFulfilled === 'function') {
      onFulfilled(this._promiseResult);
      }
    };
    const rejectionTask = () => {
      if (typeof onRejected === 'function') {
        onRejected(this._promiseResult);
      }
    };
    switch (this._promiseState) {
      case 'pending':
        this._fulfillmentTasks.push(fulfillmentTask);
        this._rejectionTasks.push(rejectionTask);
        break;
      case 'fulfilled':
        addToTaskQueue(fulfillmentTask);
        break;
      case 'rejected':
        addToTaskQueue(rejectionTask);
        break;
      default:
        throw new Error();
    }
  }
```

The previous code snippet uses the following helper function:

```
  function addToTaskQueue(task) {
    setTimeout(task, 0);
  }
```

Promises must always settle asynchronously. That's why we don't directly execute tasks, we add them to the task queue of the event loop (of browsers, Node.js, etc.). Note that the real Promise API doesn't use normal tasks (like `setTimeout()`), it uses *microtasks*, which are tightly coupled with the current normal task and always execute directly after it.

18.2.2 Method `.resolve()`

`.resolve()` works as follows: If the Promise is already settled, it does nothing (ensuring that a Promise can only be settled once). Otherwise, the state of the Promise changes to `'fulfilled'` and the result is cached in `this.promiseResult`. Next, all fulfillment reactions that have been enqueued so far, are invoked.

```
  resolve(value) {
    if (this._promiseState !== 'pending') return this;
    this._promiseState = 'fulfilled';
    this._promiseResult = value;
    this._clearAndEnqueueTasks(this._fulfillmentTasks);
    return this; // enable chaining
  }

  _clearAndEnqueueTasks(tasks) {
    this._fulfillmentTasks = undefined;
    this._rejectionTasks = undefined;
    tasks.map(addToTaskQueue);
  }
```

reject() is similar to resolve().

18.3 Version 2: Chaining .then() calls

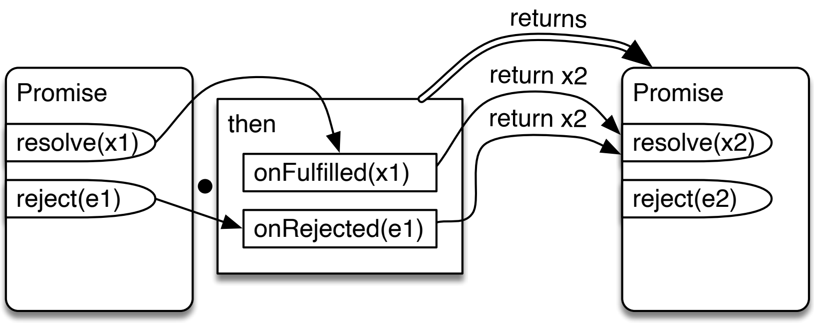

Figure 18.3: ToyPromise2 chains .then() calls: .then() now returns a Promise that is resolved by whatever value is returned by the fulfillment reaction or the rejection reaction.

The next feature we implement is chaining (fig. 18.3): A value that we return from a fulfillment reaction or a rejection reaction can be handled by a fulfilment reaction in a following .then() call. (In the next version, chaining will become much more useful, due to special support for returning Promises.)

In the following example:

- First .then(): We return a value in a fulfillment reaction.
- Second .then(): We receive that value via a fulfillment reaction.

```
new ToyPromise2()
  .resolve('result1')
  .then(x => {
    assert.equal(x, 'result1');
    return 'result2';
  })
  .then(x => {
    assert.equal(x, 'result2');
  });
```

In the following example:

- First .then(): We return a value in a rejection reaction.
- Second .then(): We receive that value via a fulfillment reaction.

```
new ToyPromise2()
  .reject('error1')
  .then(null,
      x => {
        assert.equal(x, 'error1');
        return 'result2';
      })
    .then(x => {
      assert.equal(x, 'result2');
    });
```

18.4 Convenience method `.catch()`

The new version introduces a convenience method `.catch()` that makes it easier to only provide a rejection reaction. Note that only providing a fulfillment reaction is already easy – we simply omit the second parameter of `.then()` (see previous example).

The previous example looks nicer if we use it (line A):

```
new ToyPromise2()
  .reject('error1')
  .catch(x => { // (A)
    assert.equal(x, 'error1');
    return 'result2';
  })
  .then(x => {
    assert.equal(x, 'result2');
  });
```

The following two method invocations are equivalent:

```
.catch(rejectionReaction)
.then(null, rejectionReaction)
```

This is how `.catch()` is implemented:

```
catch(onRejected) { // [new]
  return this.then(null, onRejected);
}
```

18.5 Omitting reactions

The new version also forwards fulfillments if we omit a fulfillment reaction and it forwards rejections if we omit a rejection reaction. Why is that useful?

The following example demonstrates passing on rejections:

```
someAsyncFunction()
  .then(fulfillmentReaction1)
  .then(fulfillmentReaction2)
  .catch(rejectionReaction);
```

`rejectionReaction` can now handle the rejections of `someAsyncFunction()`, `fulfillmentReaction1`, and `fulfillmentReaction2`.

The following example demonstrates passing on fulfillments:

```
someAsyncFunction()
  .catch(rejectionReaction)
  .then(fulfillmentReaction);
```

If `someAsyncFunction()` rejects its Promise, `rejectionReaction` can fix whatever is wrong and return a fulfillment value that is then handled by `fulfillmentReaction`.

If `someAsyncFunction()` fulfills its Promise, `fulfillmentReaction` can also handle it because the `.catch()` is skipped.

18.6 The implementation

How is all of this handled under the hood?

- `.then()` returns a Promise that is resolved with what either `onFulfilled` or `onRejected` return.
- If `onFulfilled` or `onRejected` are missing, whatever they would have received is passed on to the Promise returned by `.then()`.

Only `.then()` changes:

```
then(onFulfilled, onRejected) {
  const resultPromise = new ToyPromise2(); // [new]

  const fulfillmentTask = () => {
    if (typeof onFulfilled === 'function') {
      const returned = onFulfilled(this._promiseResult);
      resultPromise.resolve(returned); // [new]
    } else { // [new]
      // `onFulfilled` is missing
      // => we must pass on the fulfillment value
      resultPromise.resolve(this._promiseResult);
    }
  };

  const rejectionTask = () => {
    if (typeof onRejected === 'function') {
      const returned = onRejected(this._promiseResult);
      resultPromise.resolve(returned); // [new]
    } else { // [new]
      // `onRejected` is missing
      // => we must pass on the rejection value
      resultPromise.reject(this._promiseResult);
    }
  };

  ...
```

```
    return resultPromise; // [new]
  }
```

`.then()` creates and returns a new Promise (first line and last line of the method). Additionally:

- `fulfillmentTask` works differently. This is what now happens after fulfillment:
  - If `onFullfilled` was provided, it is called and its result is used to resolve `resultPromise`.
  - If `onFulfilled` is missing, we use the fulfillment value of the current Promise to resolve `resultPromise`.
- `rejectionTask` works differently. This is what now happens after rejection:
  - If `onRejected` was provided, it is called and its result is used to *resolve* `resultPromise`. Note that `resultPromise` is not rejected: We are assuming that `onRejected()` fixed whatever problem there was.
  - If `onRejected` is missing, we use the rejection value of the current Promise to reject `resultPromise`.

18.7 Version 3: Flattening Promises returned from `.then()` callbacks

18.7.1 Returning Promises from a callback of `.then()`

Promise-flattening is mostly about making chaining more convenient: If we want to pass on a value from one `.then()` callback to the next one, we return it in the former. After that, `.then()` puts it into the Promise that it has already returned.

This approach becomes inconvenient if we return a Promise from a `.then()` callback. For example, the result of a Promise-based function (line A):

```
asyncFunc1()
.then((result1) => {
  assert.equal(result1, 'Result of asyncFunc1()');
  return asyncFunc2(); // (A)
})
.then((result2Promise) => {
  result2Promise
  .then((result2) => { // (B)
    assert.equal(
      result2, 'Result of asyncFunc2()');
  });
});
```

This time, putting the value returned in line A into the Promise returned by `.then()` forces us to unwrap that Promise in line B. It would be nice if instead, the Promise returned in line A replaced the Promise returned by `.then()`. How exactly that could be done is not immediately clear, but if it worked, it would let us write our code like this:

```
asyncFunc1()
.then((result1) => {
    assert.equal(result1, 'Result of asyncFunc1()');
    return asyncFunc2(); // (A)
  })
  .then((result2) => {
    // result2 is the fulfillment value, not the Promise
    assert.equal(
      result2, 'Result of asyncFunc2()');
  });
```

In line A, we returned a Promise. Thanks to Promise-flattening, `result2` is the fulfillment value of that Promise, not the Promise itself.

18.7.2 Flattening makes Promise states more complicated

> **Flattening Promises in the ECMAScript specification**
>
> In the ECMAScript specification, the details of flattening Promises are described in section "Promise Objects".

How does the Promise API handle flattening?

If a Promise P is resolved with a Promise Q, then P does not wrap Q, P "becomes" Q: State and settlement value of P are now always the same as Q's. That helps us with `.then()` because `.then()` resolves the Promise it returns with the value returned by one of its callbacks.

How does P become Q? By *locking in* on Q: P becomes externally unresolvable and a settlement of Q triggers a settlement of P. Lock-in is an additional invisible Promise state that makes states more complicated.

The Promise API has one additional feature: Q doesn't have to be a Promise, only a so-called *thenable*. A thenable is an object with a method `.then()`. The reason for this added flexibility is to enable different Promise implementations to work together (which mattered when Promises were first added to the language).

Fig. 18.4 visualizes the new states.

Note that the concept of *resolving* has also become more complicated. Resolving a Promise now only means that it can't be settled directly, anymore:

- Resolving may reject a Promise: We can resolve a Promise with a rejected Promise.
- Resolving may not even settle a Promise: We can resolve a Promise with another one that is always pending.

The ECMAScript specification puts it this way: "An unresolved Promise is always in the pending state. A resolved Promise may be pending, fulfilled, or rejected."

18.7.3 Implementing Promise-flattening

Fig. 18.5 shows how `ToyPromise3` handles flattening.

We detect thenables via this function:

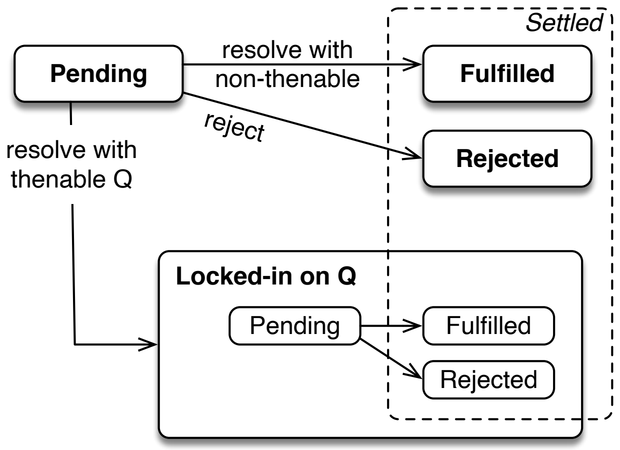

Figure 18.4: All states of a Promise: Promise-flattening introduces the invisible pseudo-state “locked-in”. That state is reached if a Promise P is resolved with a thenable Q. Afterwards, state and settlement value of P is always the same as those of Q.

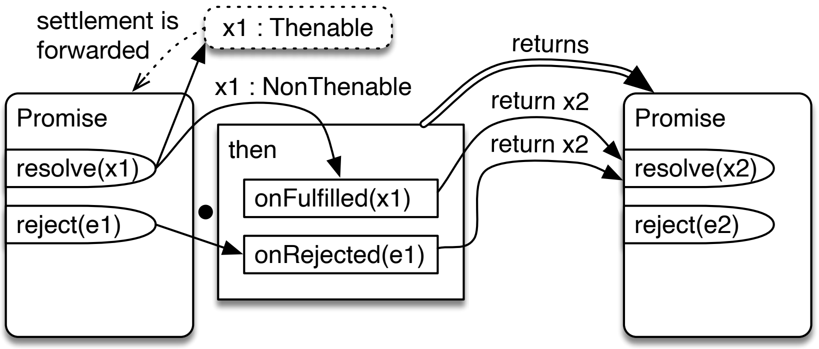

Figure 18.5: `ToyPromise3` flattens resolved Promises: If the first Promise is resolved with a thenable `x1`, it locks in on `x1` and is settled with the settlement value of `x1`. If the first Promise is resolved with a non-thenable value, everything works as it did before.

```
function isThenable(value) { // [new]
  return typeof value === 'object' && value !== null
    && typeof value.then === 'function';
}
```

To implement lock-in, we introduce a new boolean flag `._alreadyResolved`. Setting it to `true` deactivates `.resolve()` and `.reject()` – for example:

```
resolve(value) { // [new]
  if (this._alreadyResolved) return this;
  this._alreadyResolved = true;

  if (isThenable(value)) {
    // Forward fulfillments and rejections from `value` to `this`.
    // The callbacks are always executed asynchronously
    value.then(
      (result) => this._doFulfill(result),
      (error) => this._doReject(error));
  } else {
    this._doFulfill(value);
  }

  return this; // enable chaining
}
```

If `value` is a thenable then we lock the current Promise in on it:

- If `value` is fulfilled with a result, the current Promise is also fulfilled with that result.
- If `value` is rejected with an error, the current Promise is also rejected with that error.

The settling is performed via the private methods `._doFulfill()` and `._doReject()`, to get around the protection via `._alreadyResolved`.

`._doFulfill()` is relatively simple:

```
_doFulfill(value) { // [new]
  assert.ok(!isThenable(value));
  this._promiseState = 'fulfilled';
  this._promiseResult = value;
  this._clearAndEnqueueTasks(this._fulfillmentTasks);
}
```

`.reject()` is not shown here. Its only new functionality is that it now also obeys `._-alreadyResolved`.

18.8 Version 4: Exceptions thrown in reaction callbacks

As our final feature, we'd like our Promises to handle exceptions in user code as rejections (fig. 18.6). In this chapter, "user code" means the two callback parameters of `.then()`.

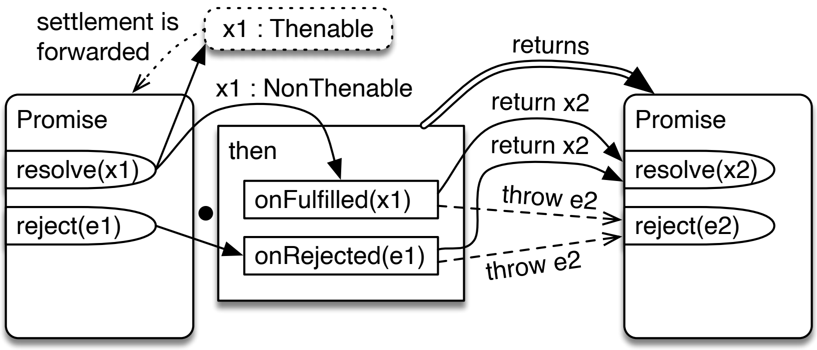

Figure 18.6: `ToyPromise4` converts exceptions in Promise reactions to rejections of the Promise returned by `.then()`.

```
new ToyPromise4()
  .resolve('a')
  .then((value) => {
    assert.equal(value, 'a');
    throw 'b'; // triggers a rejection
  })
  .catch((error) => {
    assert.equal(error, 'b');
  })
```

`.then()` now runs the Promise reactions `onFulfilled` and `onRejected` safely, via the helper method `._runReactionSafely()` – for example:

```
const fulfillmentTask = () => {
  if (typeof onFulfilled === 'function') {
    this._runReactionSafely(resultPromise, onFulfilled); // [new]
  } else {
    // `onFulfilled` is missing
    // => we must pass on the fulfillment value
    resultPromise.resolve(this._promiseResult);
  }
};
```

`._runReactionSafely()` is implemented as follows:

```
_runReactionSafely(resultPromise, reaction) { // [new]
  try {
    const returned = reaction(this._promiseResult);
    resultPromise.resolve(returned);
  } catch (e) {
    resultPromise.reject(e);
  }
}
```

18.9 Version 5: Revealing constructor pattern

We are skipping one last step: If we wanted to turn `ToyPromise` into an actual Promise implementation, we'd still need to implement the revealing constructor pattern: JavaScript Promises are not resolved and rejected via methods, but via functions that are handed to the *executor*, the callback parameter of the constructor.

```
const promise = new Promise(
  (resolve, reject) => { // executor
    // ···
  });
```

If the executor throws an exception, then `promise` is rejected.

Chapter 19

The property `.name` of functions

Contents

In this chapter, we look at the names of functions:

- All functions have the property `.name`
- This property is useful for debugging (its value shows up in stack traces) and some metaprogramming tasks (picking a function by name etc.).

19.1 Names of functions

The property `.name` of a function contains the function's name:

```
function myBeautifulFunction() {}
assert.equal(myBeautifulFunction.name, 'myBeautifulFunction');
```

Prior to ECMAScript 6, this property was already supported by most engines. With ES6, it became part of the language standard.

The language has become surprisingly good at finding names for functions, even when they are initially anonymous (e.g., arrow functions).

19.1.1 Function names are used in stack traces

One benefit of function names is that they show up in stack traces. In the following example, the argument of `callFunc()` doesn't get a name, which is why there is no function name in the second line of the stack trace.

```
function callFunc(func) {
  return func();
}

function func1() {
  throw new Error('Problem');
}

function func2() {
  // The following arrow function doesn't have a name
  callFunc(() => func1());
}

function func3() {
  func2();
}

try {
  func3();
} catch (err) {
assert.equal(err.stack, `
Error: Problem
    at func1 (ch_function-names.mjs:6:9)
    at ch_function-names.mjs:11:18
    at callFunc (ch_function-names.mjs:2:10)
    at func2 (ch_function-names.mjs:11:3)
    at func3 (ch_function-names.mjs:15:3)
    at ch_function-names.mjs:19:3
`.trim());

}
```

19.2 Constructs that provide names for functions

The following sections describe how `.name` is set up automatically by various programming constructs.

19.2.1 Function declarations

Function declarations have had a `.name` for a long time – for example:

```
function funcDecl() {}
assert.equal(funcDecl.name, 'funcDecl');
```

19.2.2 Variable declarations and anonymous functions

A function created by an anonymous function expression picks up a name if it is the initialization value of a variable declaration:

```
let letFunc = function () {};
assert.equal(letFunc.name, 'letFunc');

const constFunc = function () {};
assert.equal(constFunc.name, 'constFunc');

var varFunc = function () {};
assert.equal(varFunc.name, 'varFunc');
```

Arrow functions also get names

With regard to names, arrow functions are like anonymous function expressions:

```
const arrowFunc = () => {};
assert.equal(arrowFunc.name, 'arrowFunc');
```

From now on, whenever you see an anonymous function expression, you can assume that an arrow function works the same way.

19.2.3 Assignments and anonymous functions

Even with a normal assignment, `.name` is set up properly:

```
let assignedFunc;
assignedFunc = function () {};
assert.equal(assignedFunc.name, 'assignedFunc');
```

19.2.4 Default values and anonymous functions

If a function is a default value, it gets its name from its variable or parameter:

```
const [arrayDestructured = function () {}] = [];
assert.equal(arrayDestructured.name, 'arrayDestructured');
```

```
const {prop: objectDestructured = function () {}} = {};
assert.equal(objectDestructured.name, 'objectDestructured');

function createParamDefault(paramDefault = function () {}) {
  return paramDefault;
}
assert.equal(createParamDefault().name, 'paramDefault');
```

19.2.5 Named function expressions

The name of a named function expression is used to set up the `.name` property:

```
const varName = function funcName() {};
assert.equal(varName.name, 'funcName');
```

Because it comes first, the function expression's name `funcName` takes precedence over other names (in this case, the name `varName` of the variable). However, the name of a function expression is still only a variable inside the function expression:

```
const varName = function funcName() {
  assert.equal(funcName.name, 'funcName');
};
varName();
assert.throws(
  () => funcName,
  /^ReferenceError: funcName is not defined$/
);
```

19.2.6 Methods in object literals

If a function is the value of a property, it gets its name from that property. It doesn't matter if that happens via a method definition (line A), a traditional property definition (line B), a property definition with a computed property key (line C) or a property value shorthand (line D).

```
function shortHand() {}
const obj = {
  methodDef() {}, // (A)
  propDef: function () {}, // (B)
  ['computed' + 'Key']: function () {}, // (C)
  shortHand, // (D)
};
assert.equal(obj.methodDef.name, 'methodDef');
assert.equal(obj.propDef.name, 'propDef');
assert.equal(obj.computedKey.name, 'computedKey');
assert.equal(obj.shortHand.name, 'shortHand');
```

The names of getters are prefixed with `'get'`, the names of setters are prefixed with `'set'`:

```
const obj = {
  get myGetter() {},
  set mySetter(value) {},
};
const getter = Object.getOwnPropertyDescriptor(obj, 'myGetter').get;
assert.equal(getter.name, 'get myGetter');

const setter = Object.getOwnPropertyDescriptor(obj, 'mySetter').set;
assert.equal(setter.name, 'set mySetter');
```

19.2.7 Methods in class definitions

The naming of methods in class definitions is similar to object literals. For example, here is a class declaration (naming works the same in class expressions):

```
class ClassDecl {
  methodDef() {}
  ['computed' + 'Key']() {} // computed property key

  static staticMethod() {}
}
assert.equal(ClassDecl.prototype.methodDef.name, 'methodDef');
assert.equal(new ClassDecl().methodDef.name, 'methodDef');

assert.equal(ClassDecl.prototype.computedKey.name, 'computedKey');

assert.equal(ClassDecl.staticMethod.name, 'staticMethod');
```

Getters and setters again have the name prefixes `'get'` and `'set'`, respectively:

```
class ClassDecl {
  get myGetter() {}
  set mySetter(value) {}
}
const getter = Object.getOwnPropertyDescriptor(
  ClassDecl.prototype, 'myGetter').get;
assert.equal(getter.name, 'get myGetter');

const setter = Object.getOwnPropertyDescriptor(
  ClassDecl.prototype, 'mySetter').set;
assert.equal(setter.name, 'set mySetter');
```

19.2.7.1 Methods whose keys are symbols

The key of a method can be a symbol. The `.name` property of such a method is still a string:

- If the symbol has a description, the method's name is the description in square brackets.
- Otherwise, the method's name is the empty string (`''`).

```
const keyWithDesc = Symbol('keyWithDesc');
const keyWithoutDesc = Symbol();

const obj = {
  [keyWithDesc]() {},
  [keyWithoutDesc]() {},
};
assert.equal(obj[keyWithDesc].name, '[keyWithDesc]');
assert.equal(obj[keyWithoutDesc].name, '');
```

19.2.8 Names of classes

Under the hood, classes are functions. Property `.name` of such functions is also set up correctly:

```
class ClassDecl {}
assert.equal(ClassDecl.name, 'ClassDecl');

const AnonClassExpr = class {};
assert.equal(AnonClassExpr.name, 'AnonClassExpr');

const NamedClassExpr = class ClassName {};
assert.equal(NamedClassExpr.name, 'ClassName');
```

19.2.9 Default exports and anonymous constructs

All of the following statements set `.name` to `'default'`:

```
// Anonymous function expressions
export default function () {}
export default (function () {});

// Anonymous class expressions
export default class {}
export default (class {});

// Arrow functions
export default () => {};
```

19.2.10 Other programming constructs

- Generator functions and generator methods get their names the same way that normal functions and methods do. There are no asterisks in their names:

  ```
  function* genFuncDecl() {}
  assert.equal(genFuncDecl.name, 'genFuncDecl');
  ```

- `new Function()` produces functions whose `.name` is `'anonymous'`. A webkit bug describes why that is necessary on the web.

  ```
  assert.equal(new Function().name, 'anonymous');
  ```

- `func.bind()` produces a function whose `.name` is `'bound '+func.name`:

  ```
  function func(x) {
    return x
  }
  const bound = func.bind(undefined, 123);
  assert.equal(bound.name, 'bound func');
  ```

19.3 Things to look out for with names of functions

19.3.1 The name of a function is always assigned at creation

A function only gets a name if one of the previously mentioned patterns is used during its creation. Using one of the patterns later doesn't change anything – for example:

```
function functionFactory() {
  return function () {}; // (A)
}
const func = functionFactory(); // (B)
assert.equal(func.name, ''); // anonymous
```

Function `func` is created in line A, where it also receives its non-name, the empty string `''`. Using an assignment in line B doesn't update the initial name. Missing function names could conceivably be updated later, but doing so would impact performance negatively.

19.3.2 Caveat: minification

Function names are subject to minification, which means that they will usually change in minified code. Depending on what we want to do, we may have to manage function names via strings (which are not minified) or we may have to tell our minifier what names not to minify.

19.4 Changing the names of functions

More information on property attributes

`Object.getOwnPropertyDescriptor()` and `Object.defineProperty()` work with property attributes. For more information on those, see §9 "Property attributes: an introduction".

These are the attributes of property `.name`:

```
const func = function () {};
assert.deepEqual(
  Object.getOwnPropertyDescriptor(func, 'name'),
  {
    value: 'func',
    writable: false,
    enumerable: false,
```

```
    configurable: true,
  });
```

The property not being writable means that we can’t change its value via assignment:

```
assert.throws(
  () => func.name = 'differentName',
  /^TypeError: Cannot assign to read only property 'name'/
);
```

The property is, however, configurable, which means that we can change it by re-defining it:

```
Object.defineProperty(
  func, 'name', {
    value: 'differentName',
  });
assert.equal(func.name, 'differentName');
```

19.5 The function property `.name` in the ECMAScript specification

- The spec operation `SetFunctionName()` sets up the property `.name`. Search for its name in the spec to find out where that happens. The third parameter specifies a name prefix:
  - Getters and setters prefix `'get'` and `'set'`.
  - `Function.prototype.bind()` prefixes `'bound'`.
- Anonymous function expressions initially not having a property `.name` can be seen by looking at how functions are created in the spec:
  - The names of named function expressions are set up via `SetFunctionName()`. That operation is not invoked when creating anonymous function expressions.
  - The names of function declarations are set up elsewhere, when entering a scope (early activation).
  - Additionally, the spec explicitly states: “Anonymous functions objects that do not have a contextual name associated with them by this specification do not have a `"name"` own property but inherit the `"name"` property of `%Function.prototype%`.”
- When an arrow function is created, no name is set up, either (`SetFunctionName()` is not invoked).

Most JavaScript engines diverge from the spec and do create own properties `.name` for anonymous functions:

```
> Reflect.ownKeys(() => {})
[ 'length', 'name' ]
```