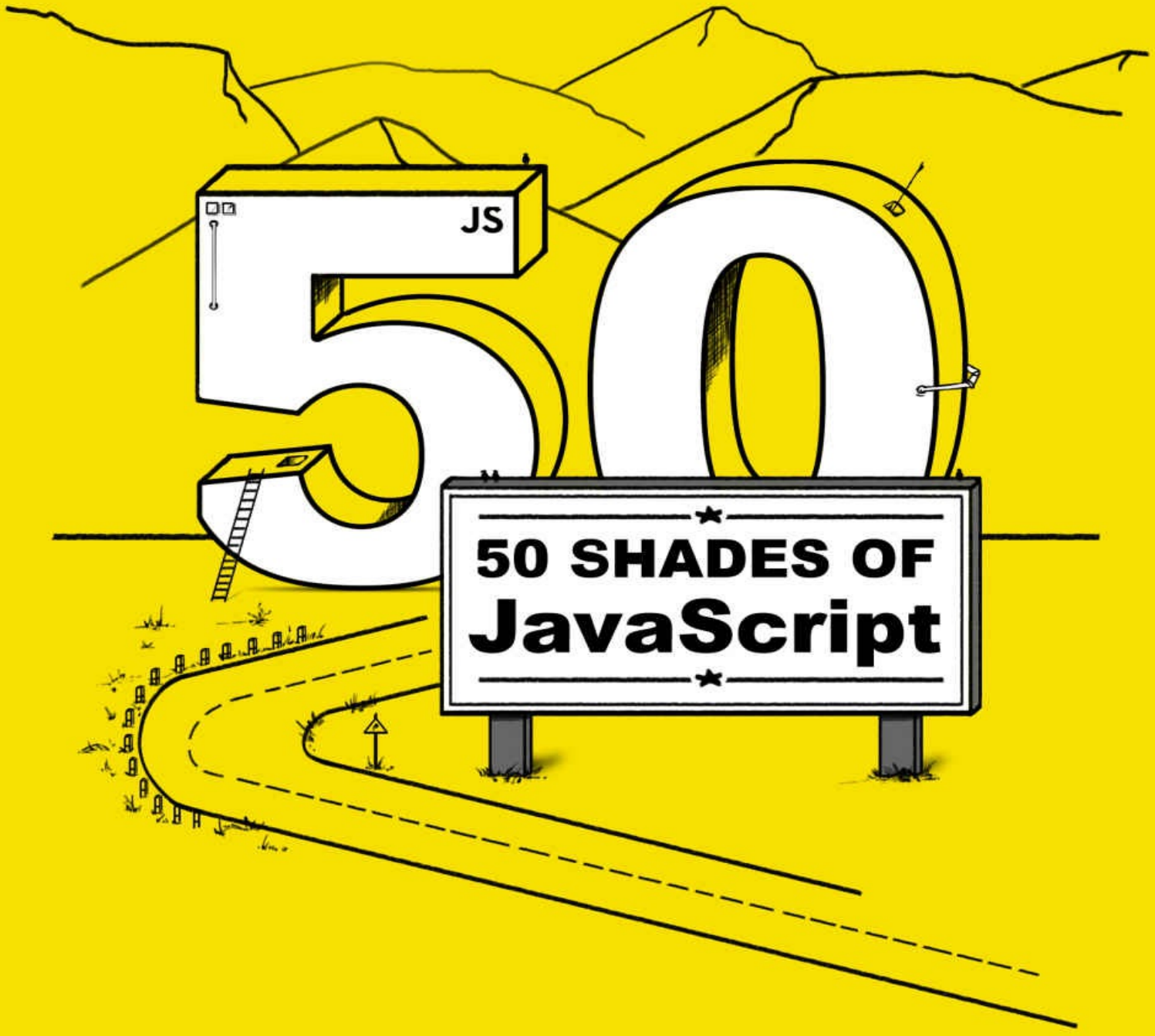


50 of the most interesting features, concepts, and patterns in JavaScript.

Written and illustrated by Krasimir Tsonev.

50 shades of JavaScript

Krasimir Tsonev

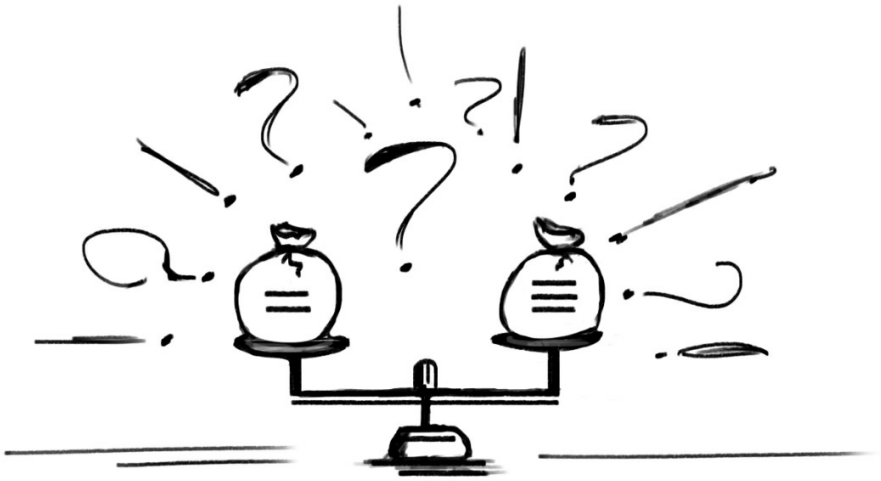
50 shades of JavaScript

Hey,

My name is Krasimir Tsonev. I work with JavaScript for more than a decade and I love it. The 50 stories in this book are stuff that I encountered in my daily work and remembered. Some of them are basic features of the language but I found them important and interesting. I wrote this book for people that are already using JavaScript and have some experience with it.

Happy JavaScripting and remember - I can’t teach you anything, I can only make you think.

Basics



Strict equality

I’m writing JavaScript for a long time and one of the first things that I learned is to use `===` (triple equals) instead of `==` (double equals). It is the be-on-the-safe-side mental model. For example:

```
user.id === id
```

over

```
user.id == id
```

Such equality comparisons are a source of a lot of jokes. Very often we stumble on statements that look illogical. For example, what if we compare an empty string with a boolean `false`:

```
console.log("" == false); // true
```

This comparison is `true` because we are using double equals or the so-called loose equality. The result is `true` because both values are converted to a common type and then compared. The above in reality looks more like:

```
console.log(Boolean("") === false); // true
```

After the conversion JavaScript engine uses `===` (strict equality).

So, use triple equals to make your code explicit. This will save you from some weird bugs.


I'm not just a Comma!

Comma operator

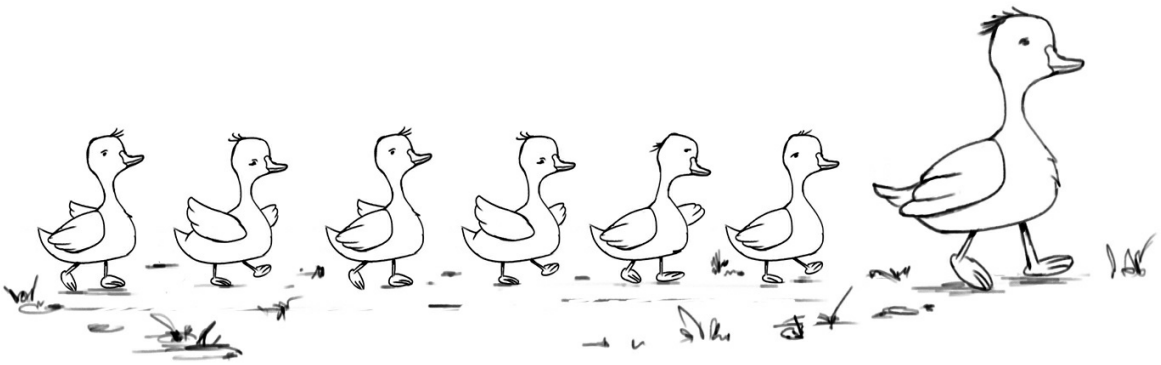
I think the comma operator in JavaScript is underestimated. The popular perception is that the comma is used to separate function arguments or items in an array. However, there is another use case.

```
const page = {
  visits: 42
};
const newPageView = () => {
  return page.visits += 1, page.visits;
};

console.log(newPageView()); // 43;
console.log(newPageView()); // 44;
```

`newPageView` is a function that wants to increase the `visits` field but also wants to return it. The comma operator evaluates each of its operands from left to right and returns the last one as a result. That is why the result of the function is the new value of the property.

Someone will ask when such type of writing is useful. The truth is that the comma operator makes sense the most in places where we write the code in one line. Like for example the ternary operator.

```
const a = () => 'a';
const b = () => 'b';
const isValid = true;
const result = isValid ? (a(), b()) : 'Nope';
console.log(result); // b
```



Spread operator

Back in the days when I started reading about the new features of JavaScript, I was the most excited about the spread operator. It allows us to expand iterable (or strings) in various places. I'm very often using this operator when constructing an object. For example:

```
const name = {
  firstName: "Krasimir",
  lastName: "Tsonev"
};

// Adding fields to an object
const profile = { age: 36, ...name };
console.log(profile); // age, first, last name
```

It works well also when we want to overwrite properties. Let's say that we want to update the `age` field of the `profile` above. We may do the following:

```
const updates = { age: 37 };
const updatedProfile = { ...profile, ...updates };
console.log(updatedProfile); // age=37
```

Another popular use case is when we want to pass multiple arguments to a function:

```
const values = [10, 33, 42, 2, 9];
console.log(Math.max(...values));
// instead of
console.log(Math.max(10, 33, 42, 2, 9));
```

Last but not least, the operator does a good job of copying arrays. Not deep copying but just to avoid the mutation of the original object. Remember how objects and arrays are passed by reference (that's not entirely true) to functions. In such cases, we may want to work with the array but avoid changing it. Look at the following example:

```
const values = [10, 33, 42, 2, 9];
function findBiggest(arr) {
  return arr.sort((a, b) => (a > b ? -1 : 1)).shift();
}
const biggest = findBiggest(values);
console.log(biggest), // 42
console.log(values); // [33, 10, 9, 2]
```

The `findBiggest` function sorts the items of the array and returns the biggest one. The problem is that it's doing it with the `shift` method, which mutates the

object. It leaves it with fewer items.

The solve this issue we could use the spread operator.

```
const values = [10, 33, 42, 2, 9];
function findBiggest(arr) {
  return [...arr].sort((a, b) => (a > b ? -1 : 1)).shift();
}
const biggest = findBiggest(values);
console.log(biggest), // 42
console.log(values); // [10, 33, 42, 2, 9]
```


JS
name
job
USER

Destructing

I first saw destructuring in some other language. It took me some time to understand it but once I got the idea I couldn't stop thinking how nice will be to have it in JavaScript. Well, sometime after was introduced in the language. The destructing is a JavaScript expression that extracts array items or object fields into individual variables. Here is how we destruct an object:

```
const user = { name: "Krasimir", job: "dev" };
const { name, job } = user;
console.log(`${name}, position: ${job}`); // Krasimir, position: dev
```

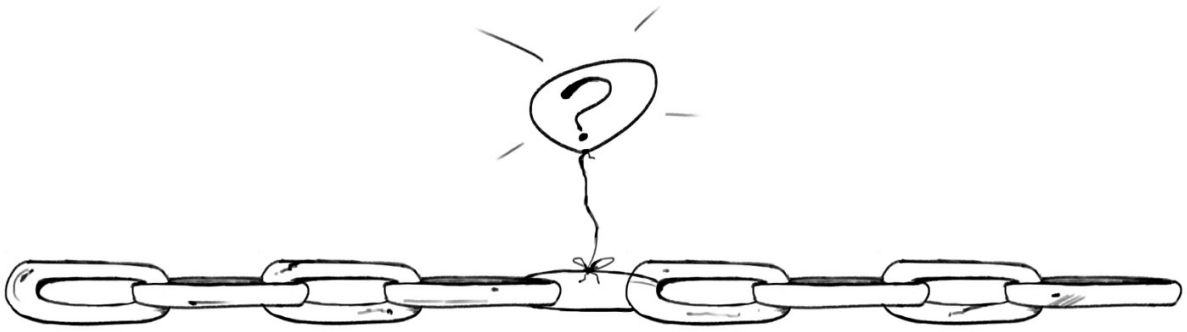
The other name of this assignment is "unpacking". As we can see here the `name` and `job` fields are coming from the `user` object.

It works similarly with arrays:

```
// destructuring array
const arr = ["oranges", "apples", "bananas"];
const [a, b, c] = arr;
console.log(`I like ${a}`); // I like oranges
```

In the end, if we don't like a field name we may create an alias. In the following example, `name` becomes `who` and `job` becomes `position`:

```
// destructuring with renaming
const user = { name: "Krasimir", job: "dev" };
const { name: who, job: position } = user;
console.log(`${who}, position: ${position}`); // Krasimir, position: dev
```



Optional chaining

My “favorite” error while I’m coding something is “Cannot read properties of undefined”. Here’s an example that produces such error:

```
const user = { name: 'Krasimir' };
console.log(user.skills[0]);
// Uncaught TypeError: Cannot read properties of undefined (reading '0')
```

To prevent such situations, we do checks or use helpers. Reading values from deeply nested properties has always been a source of bugs. Today the language has a feature that solves this problem. It is called *optional chaining*. It provides safe access to nested fields. When something along the chain is `undefined` or `null`, the whole expression results in `undefined`. For example:

```
const user = { name: 'Krasimir' };
console.log(user?.skills?.[0]); // undefined
```

By reference

By value

By reference or by value

In programming, there is the phrase “passing a variable”. That is the moment when we call a function with a given parameter. In some languages, those parameters are passed by value in others by reference. In JavaScript we have a bit of a hybrid approach.

The primitives, like numbers and strings, are passed by value and some others like object literals and arrays are passed by “copy of a reference”. Let’s illustrate this with a couple of examples.

```
var numOfUsers = 24;
function doSomething(num) {
  num += 1;
}

doSomething(numOfUsers);

console.log(numOfUsers); // 24
```

Even though we define `numOfUsers` with `var`, we are passing it by its value. Meaning that inside the function, we receive its value, but we can not modify the original variable.

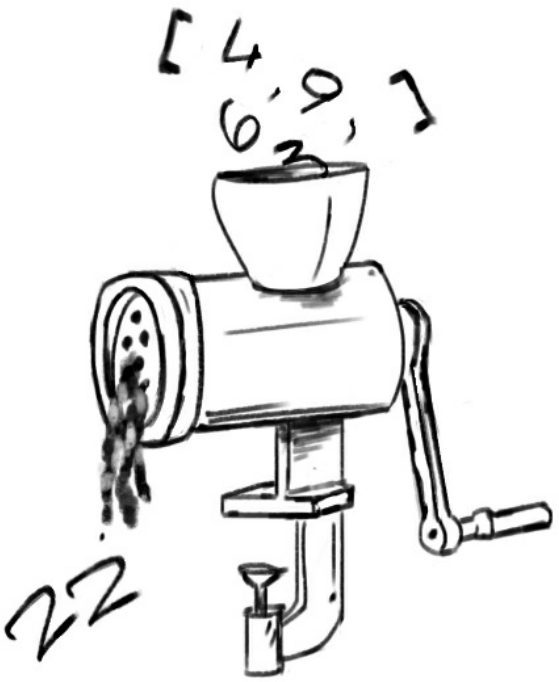
Here is another example:

```
const user = { score: 78 }
function mutates(obj) {
  obj.score += 12;
}

mutates(user);
console.log(user.score); // 90
```

Here `user` is passed by a copy of a reference. We can mutate its fields, but we can’t amend the original definition. The following snippet proves that:

```
const user = { score: 78 }
function doesntMutate(obj) {
  obj = { score: 120 }
}

doesntMutate(user);
console.log(user.score); // 78
```

We have to make the note that in some other languages the `obj = { score: 120 }` assignment will change the value of the `user` object.


[4, 9, 6, 3]
22

Reducing

Very often, we have to transform objects from one shape to another. My favorite tool for such transformations is the `reduce` function. Let's say that we have the following array of objects:

```
const positions = [
  { languages: ["HTML", "JavaScript", "PHP"], years: 4 },
  { languages: ["JavaScript", "PHP"], years: 2 },
  { languages: ["C", "PHP"], years: 3 },
];
```

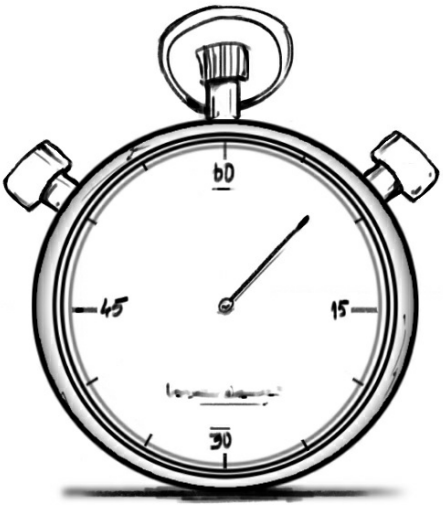
And we need the total number of years for each of the languages. The solution involves looping over the items and some sort of aggregation. Exactly what `reduce` is made for:

```
const yearsOfWriting = positions.reduce((res, { languages, years }) => {
  languages.forEach((lang) => {
    if (!res[lang]) res[lang] = 0;
    res[lang] += years;
  });
  return res;
}, {});

console.log(yearsOfWriting); // { HTML: 4, JavaScript: 6, PHP: 9, C: 3 }
```

`reduce` is a method of the Array prototype. It allows us to accumulate a value by looping over the items of the array.

It is possible to use the function on objects too. We need a little bit of help. If we pass the object to `Object.keys`, we will get back an array of all the keys/fields. Then we can iterate them. Let's continue the example and ask another question - how many years in total?

```
const yearsOfWriting = { HTML: 4, JavaScript: 6, PHP: 9, C: 3 };
const totalYears = Object.keys(yearsOfWriting).reduce((total, key) => {
  return total + yearsOfWriting[key];
}, 0);

console.log(totalYears); // 22
```


60
45
15
30

Async/await

The topic of asynchrony in JavaScript is huge. That is because the language offers many mechanisms for handling asynchronous operations. Over the years, those mechanisms change. Historically, the native APIs introduced the first one. They accept a function that is fired sometime in the future. We name this function a *callback*.

```
const fs = require('fs');

fs.readFile('./content.md', function callback(error, result) {
  if (error) {
    // in case of error
  } else {
    // use the data
  }
})
```

The `callback` function is triggered when the operation has some development. Either the file is successfully read or there is an error. It was a well-established practice to accept the potential error as a first argument. This encourages error handling. Using a lot of callbacks though leads to deeply nested functions, producing the so-called “callback hell”.

```
functionA(param, function (err, result) {
  functionB(param, function (err, result) {
    functionC(param, function (err, result) {
      // And so on...
    });
  });
});
```

The API that was supposed to fix the callback hell was promises. The promise is an object that represents the future of our operation. It may succeed or fail. The object has three states - resolved, rejected, and pending.

Here’s the same example converted to use promises.

```
import { readFile } from 'fs/promises';
readFile('./content.md')
  .then(data => { ... })
  .catch(err => { ... });
```

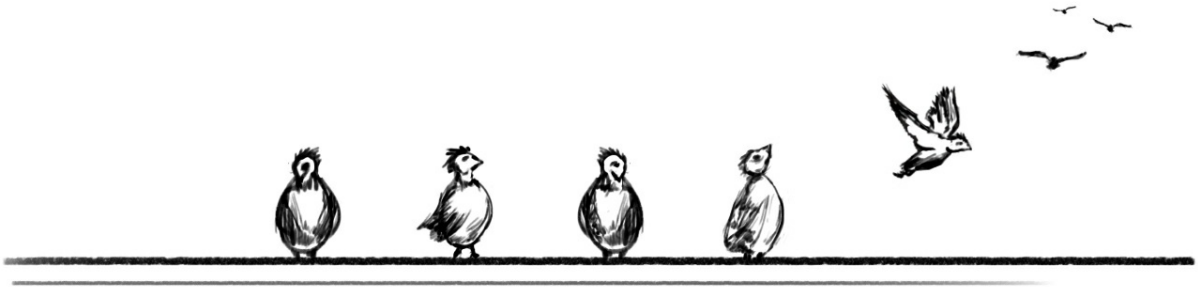
The promises don’t have the problem of the callbacks, but they are not perfect. Once we start nesting or chaining promises we may end up with pretty messy code.

Nowadays, almost everyone is using another API - the `async/await`. This API allows us to define a function as asynchronous with the `async` keyword in front. Then inside, we can place the `await` keyword before each of our promise. This will make the function *pause* till the promise is resolved. Again, here's the same example but re-written with `async/await`.

```
import { readFile } from 'fs/promises';

async function getContent() {
  try {
    const data = await readFile('./content.md');
    // use the data
  } catch(err) {
    // in case of error
  }
}

getContent();
```

We should again mention that the `await` keyword may be placed before a promise. `readFile` in our case is a function that returns a promise. Another important fact is that every `async` function returns a promise. For example, the `getResult` below doesn't return 10, but a promise that is resolved with a value of 10.

```
async function getResult() {
  return 10;
}
console.log(getResult());
// Promise<fulfilled: 10>
```



Iterable protocol

The iterable protocol is so underrated. I don't see people using it, which makes me sad because it's a wonderful instrument. It defines how we loop over a specific object. Or in other words, it allows us to create custom iterating behavior. We have to create a property called `@@iterator` (`Symbol.iterator` is a shortcut for this key). That property must equal a zero-argument function that returns an object matching the iterable protocol. Here is an example:

```
const user = {
  data: ["JavaScript", "HTML", "CSS"],
  [Symbol.iterator]: function () {
    let i = 0;
    return {
      next: () => ({
        value: this.data[i++],
        done: i > this.data.length,
      }),
    };
  },
};
for (const skill of user) { console.log(skill); }
```

The protocol requires that we return an object with a `next` method. That function should result in another object that has `value` and `done` fields. The `value` could be anything, and `done` is a boolean that indicates whether the iteration finishes.

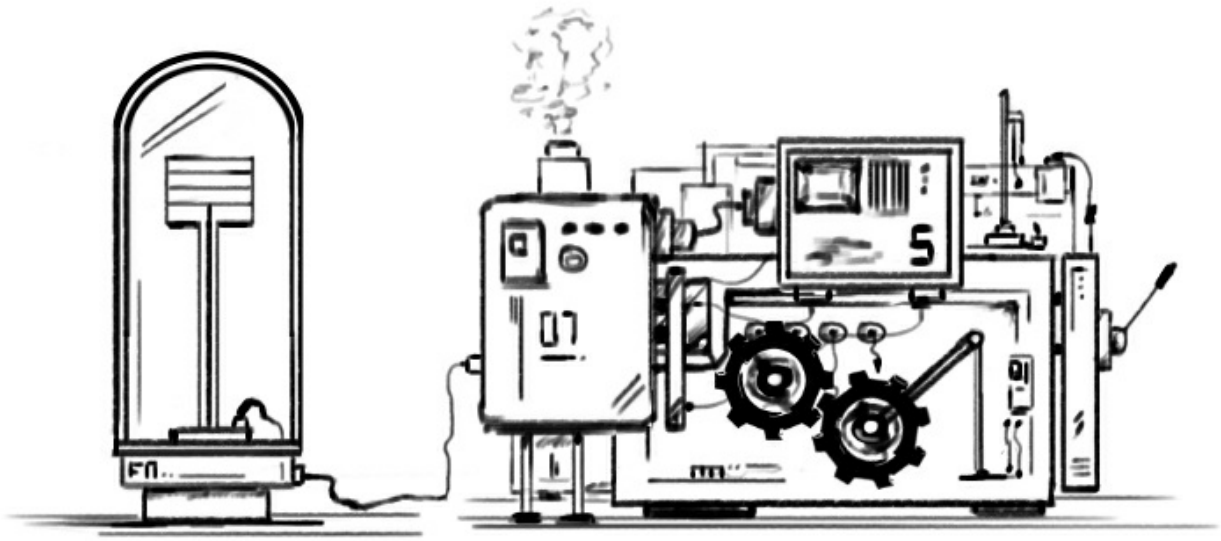
Notice, in the example above, how `user` is not an array, but we can use it as such. That is because we defined a custom iterator. The script outputs `"JavaScript"`, `"HTML"` and `"CSS"` (in that order).

This feature comes in handy if we have complex data structures and we want to access deeply nested properties. I like to use the iterable protocol to facilitate the destructing of my objects.

```
const user = {
  name: { first: 'Krasimir', last: 'Tsonev' },
  position: 'engineer',
  [Symbol.iterator]: function () {
    let i = 0;
    return {
      next: () => {
        i++;
        if (i === 1) {
          return { value: `${this.name.first} ${this.name.last}`, done: false };
        } else if (i === 2) {
```

```
          return {value: this.position, done: false };
        }
        return { done: true }
      }
    }
  }
}
const [name, position] = user;
console.log(`${name}: ${position}`); // Krasimir Tsonev: engineer
```

By definition, we can destruct every iterable object. And if our objects are not iterable, we can make them so by using the technique above - defining an iterable protocol.



Generators

Similar to the iterable protocol, the usage of generators in JavaScript is not very popular. The community is not betting a lot on this API. I think that it has potential. Especially for handling asynchronous processes.

The generator functions are a special type of function which if called returns an iterable generator. This means that the code inside the function is not executed immediately. We have to call the `next` method of the generator. Then the execution continues until a `yield` statement appears. The following code snippet illustrates the workflow:

```
function* calculate() {
  yield 10;
  const result = yield 5;
  console.log(`Result = ${result}`);
}

const g = calculate();
const res1 = g.next(); // {value: 10, done: false}
const res2 = g.next(); // {value: 5, done: false}
g.next(res1.value + res2.value); // Result = 15
```

Let’s read this example line by line and explain what happens:

- `const generator = calculate()` - at this point the function’s body is not executed. We just created the generator. In fact `calculate` may create as many generators as we want. They will be different iterables.
- `const res1 = generator.next()` - now the function runs and its execution stops at the first `yield` statement. Whatever we `yield` appears in the `value` field of the returned object.
- `const res2 = generator.next()` - same as the above line except that we have `5` as a value. And we have to say that the function is paused right before `result` receives its value. At this point, `results` constant is still not defined. Only the following `next()` call outside will complete the assignment.
- `generator.next(res1.value + res2.value)` - this line resumes the generator by sending the sum of the previously exported integers.

The ability of the generator to send and receive data is what makes it unique and opens the door for some interesting implementations. The most common

one is the implementation of the command pattern. Think about how we have to do something which consists of a couple of steps. For example, we want to get an image URL from a remote server and create an `<img>` tag. Here is how this looks like with a generator.

```
commander(robot());

function* robot() {
  const catURL = yield ["get-cat"];
  const imgTag = yield ["format", catURL];
  console.log(imgTag);
}
async function commander(gen, passBack) {
  let state = gen.next(passBack);
  switch (state.value ? state.value[0] : null) {
    case "get-cat":
      const res = await fetch("https://api.thecatapi.com/v1/images/search");
      const data = await res.json();
      return commander(gen, data[0].url);
    case "format":
      return commander(gen, `<img src="${state.value[1]}" />`);
  }
  if (!state.done) {
    commander(gen);
  }
}
// after a second or two we'll get:
// <img src="https://cdn2.thecatapi.com/images/<random id here>.jpg" />
```

Our generator function `robot` sends commands out and asynchronously receives results. `get-cat` is a command that performs an HTTPS request to thecatapi.com and returns a random image URL. `format` is another command and it returns the image tag. Notice how generators, similarly to the async/await syntax, make our functions look synchronous.

Browser APIs


{...}
SDA

Printing JSON

This chapter of this book addresses the most used debugging tool of every developer - printing to the console. In JavaScript, we work with data structures all the time. Very often we need to look inside objects. One of my favorite ways to print an object is to use `JSON.stringify` API.

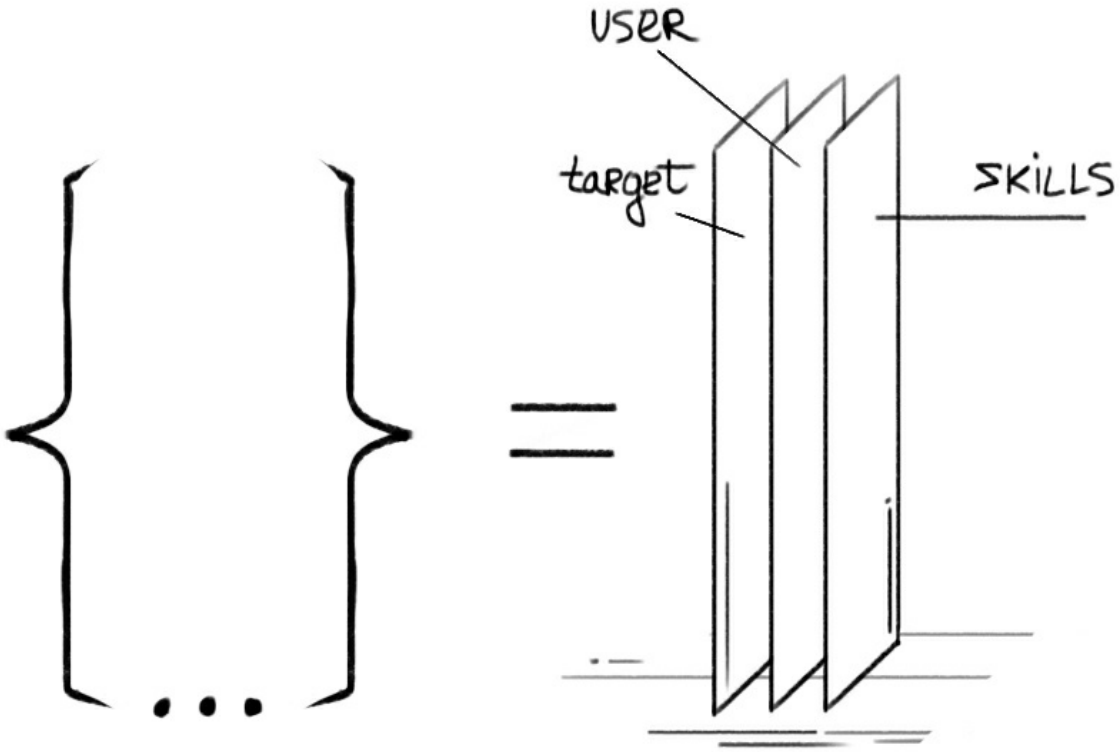
```
const user = {
  name: "Molecule Man",
  age: 29,
  secretIdentity: "Dan Jukes",
  powers: ["Radiation resistance", "Radiation blast"],
};

console.log(JSON.stringify(user, null, 2));
```

Notice the `null` and `2` arguments. Without them, everything will appear in one line. The second argument (`null`) is for a replacer function. That comes in handy when we want to have custom traversing. A couple of times I had to write my own replacer. That was when I wanted to treat some of the properties in a specific way. The other, more popular, use case is to solve the problem of having a circular data structure. The DOM element is a good example of such a structure because every DOM element points to its parent which points to its children and so on.

Another quick trick that is possible with `JSON.stringify` and `JSON.parse` is cloning an object. For example:

```
const user1 = {
  name: "Molecule Man",
  age: 29,
  secretIdentity: "Dan Jukes",
  powers: ["Radiation resistance", "Radiation blast"],
};
const user2 = JSON.parse(JSON.stringify(user1));

console.log(user1 === user2); // false
```

We have to clarify that this doesn’t work every time. If the object is a complex one with the mentioned above circular dependencies the process will fail.


USER
target
SKILLS

Object.assign

There are some APIs that I use very often. `Object.assign` is one of them. The use case for me is almost always constructing a new object by merging various properties. Here are the basics:

```
const skills = { JavaScript: true, GraphQL: false };
const user = { name: 'Krasimir' };

const profile = Object.assign({}, user, skills);
// { "name":"Krasimir", "JavaScript":true, "GraphQL":false }
```

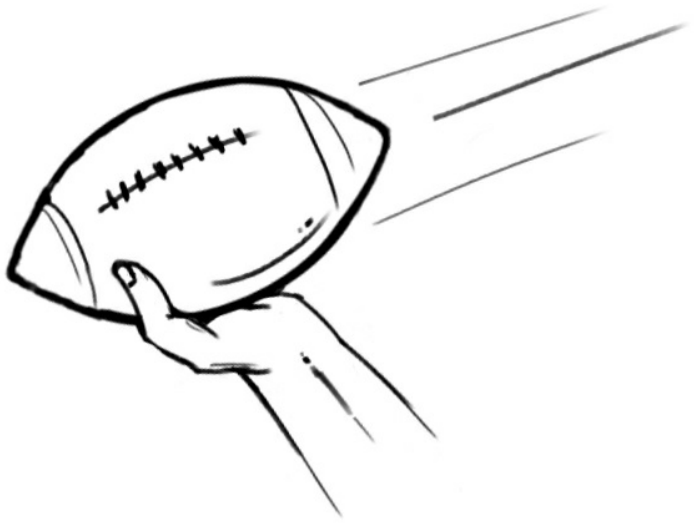
There are a couple of aspects that make this method useful. We can for example set a default value of a field and then overwrite it:

```
const defaults = { JavaScript: false, GraphQL: false, name: 'unknown' };
const skills = { JavaScript: true };

const profile = Object.assign(defaults, skills);
// { "JavaScript":true, "GraphQL":false, "name":"unknown" }
```

The falsy arguments are ignored so we can add a field only if it exists with a neat one-liner:

```
function createUser(accessToken) {
  const user = Object.assign(
    { name: "Krasimir" },
    accessToken && { accessToken }
  );
  return user;
}

createUser('xxx'); // { name:"Krasimir", accessToken:"xxx" }
createUser(); // { name:"Krasimir" }
```

Overall I use this API as a safety net. Especially in the cases where data is missing or is incomplete.



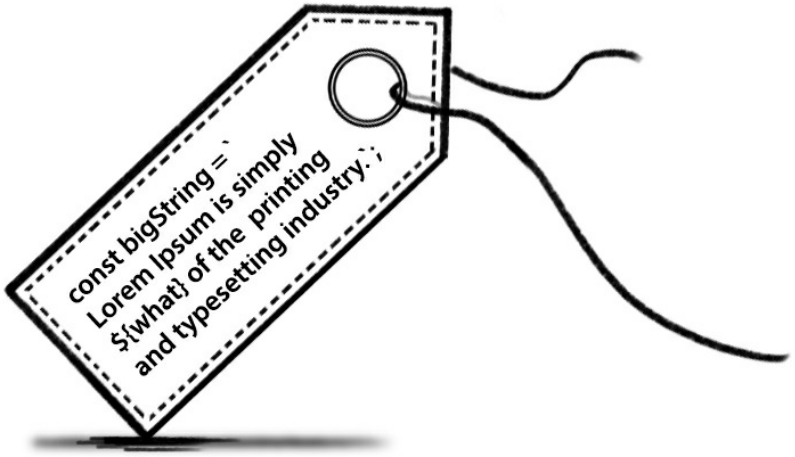
Capture groups

As most of the programming languages today JavaScript supports regular expressions. I'm writing software for more than 15 years, and I've never been so good with this. It just gets complicated quickly.

In this chapter, I want to share my favorite feature - capture groups. It can solve a whole set of problems with just one line of code. Recently I had to rename a file from `script.js` to `script.prod.js`. Capture groups are perfect for this:

```
function renameFile(file, postfix) {
  return file.replace(
    /(\.(js|ts|jsx|tsx))/i,
    `.${postfix}$1`
  );
}

console.log(renameFile("public/js/script.js", "prod"));
// public/js/script.prod.js
```

The `(\.(js|ts|jsx|tsx))` defines a capture group that matches `.js`. Later in the second argument of the `replace` method we have it as `$1`. `$2` will contain the value of the second capture group and so on.


const bigString =
Lorem Ipsum is simply
${what} of the printing
and typesetting industry.

Tagged template literal

Some time ago we weren't able to write multiline strings in JavaScript. We had to concatenate. Then the template literal was born. Suddenly our life became easier

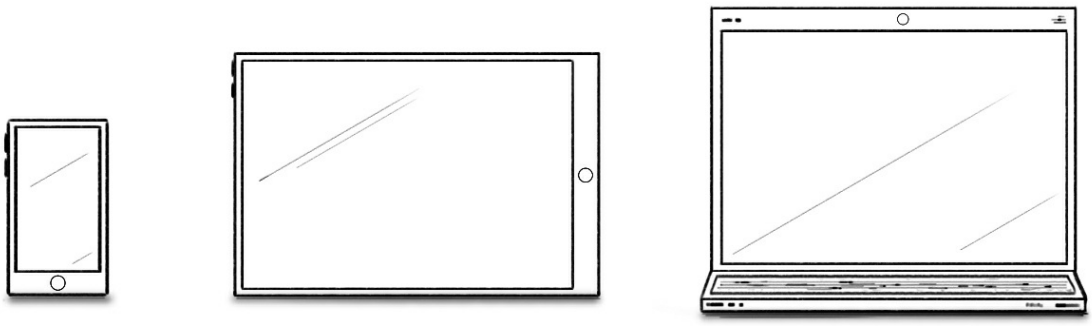
```
const what = "dummy text";
const bigString = `
  Lorem Ipsum is simply ${what} of the
  printing and typesetting industry.
`;
```

In terms of string manipulations, this is probably the best thing that happen to JavaScript in the last couple of years.

The template literals are extremely useful but tagged template literals are even better. They allow us to define a function that will process the string. The function accepts all the text and the expressions in the placeholders. Check out the example below. Instead of a primitive, we are passing a function to the placeholder. That function is fired with a `theme` object.

```
const theme = {
  brandColor1: "#BE2525",
  brandColor2: "#BE0000",
};
function css(strings, ...values) {
  return strings.reduce((res, str, i) => {
    return res + str + (values[i] ? values[i](theme) : "");
  }, "");
}
const styles = css`
  font-size: 1.2em;
  color: ${(theme) => theme.brandColor2};
`;
console.log(styles);
// font-size: 1.2em;
// color: #BE0000;
```

The JavaScript community uses this feature of the language in places where complex parsing is needed. There is all sort of libraries that rely on a tagged template. The most popular ones are implementing CSS-in-JS solutions. There we have our CSS writing as a tagged template literal. Similar to the example above.



Media query list

The first article about responsive design that I saw dates back to 2010. Ethan Marcotte is its author. Since then, the Web evolved, and so have the developers that are building websites. Nowadays, responsive architecture is not a question. It’s the default.

There are a couple of tools for implementing responsive design. Probably the main one is media queries. It’s a feature of CSS where we can specify a frame under which our elements behave or/and look differently. For example:

```
body {
  background-color: blue;
}
@media screen and (max-width: 800px) {
  body {
    background-color: red;
  }
}
```

The background color of the `<body>` is by default blue. However, if the browser has a viewport width less than 800px the color becomes red.

For a long time, this was only available in CSS. That is not the case anymore. We have a `MediaQueryList` API:

```
const mediaQueryList = window.matchMedia(
  "(max-width: 800px)"
);
const handle = (mql) => {
  if (mql.matches) {
    console.log("width <= 800px");
  } else {
    console.log("width > 800px");
  }
};

mediaQueryList.addEventListener("change", handle);
handle(mediaQueryList);
```

We create a media query list object matching our criteria. After that, we can attach a listener that will tell us when our criteria are met.

With this API we can make truly responsive applications. Not only on how the things look like but also how they work.


click

Event delegation

I find it very interesting how when technology evolves we start forgetting stuff. One of these things is the event delegation. Today, when we pretty much always use a framework, the event bubbling and capturing is not a thing anymore. The tools that we rely on are handling those processes. I remember how at interviews for front-end developer positions this was one of the common questions. And so I decided to dedicate a chapter to it.

Consider the following example:

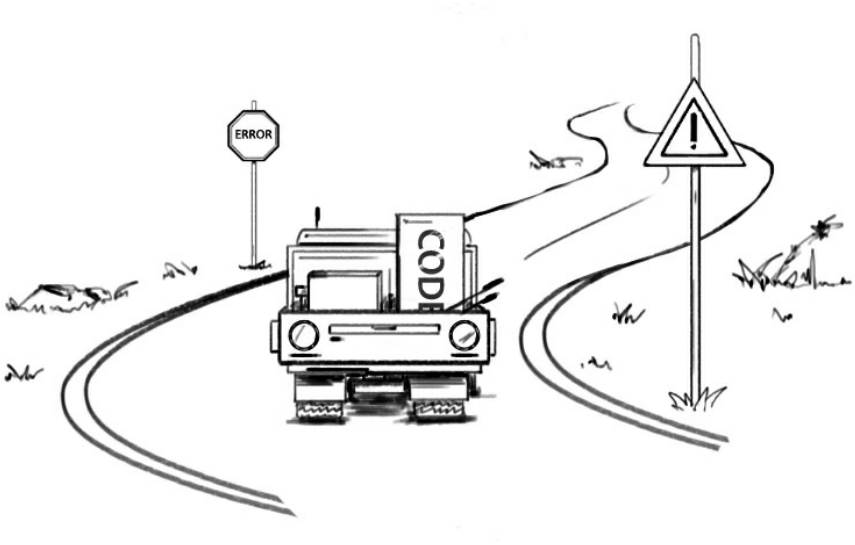
```
<div id="container">
  <header>Hello</header>
  <button id="button">call to action</button>
</div>

document
  .querySelector("#button")
  .addEventListener("click", () => {
  // ...
});
```

When the page loads initially we attach a listener to the `<button>` element. This works well unless we touch the DOM. As soon as we replace the content of the container, we have to re-attach the listener. This may seem a silly problem now, but it was a big thing a couple of years ago.

If we write vanilla JavaScript and have to manipulate the DOM elements, we should make sure that our listeners are present. One of the solutions for this is to attach the listener to a parent element that doesn't change. In our case this is `<div id="container">`. Then we rely on the `event.target` to find out where the event comes from.

```
document
  .querySelector("#container")
  .addEventListener('click', (event) => {
  // event.target....
});
```

This works because of the event delegation. If not captured the event from each element bubbles up and we are able to catch it.


ERROR
COD

Error handling

There is one thing that developers forget very often - error handling. For me, there are two rules of proper error handling. The first one is that the error should be processed in the right place. It doesn't make sense to add a try-catch block and then just log a message. The second rule is that the error should hold enough information so we can build context and react to it properly.

To illustrate the first rule we will make the assumption that we have a button which if clicked triggers a `fetch` request.

```
// somewhere in our services layer
async function getAllProducts() {
  try {
    const res = await fetch('/api/products');
    return res.json();
  } catch(error) {
    console.log('Ops! Something happen.');
  }
}
// in a React Component
<button onClick={async () => {
  const data = await getAllProducts();
  // render the data
}}>
  All products
</button>
```

This works, but it is not very useful. If the request fails and we catch the error, we can't do much with it in our service. On another side, if the try-catch block is inside the React component we will be able to render a message to the user.

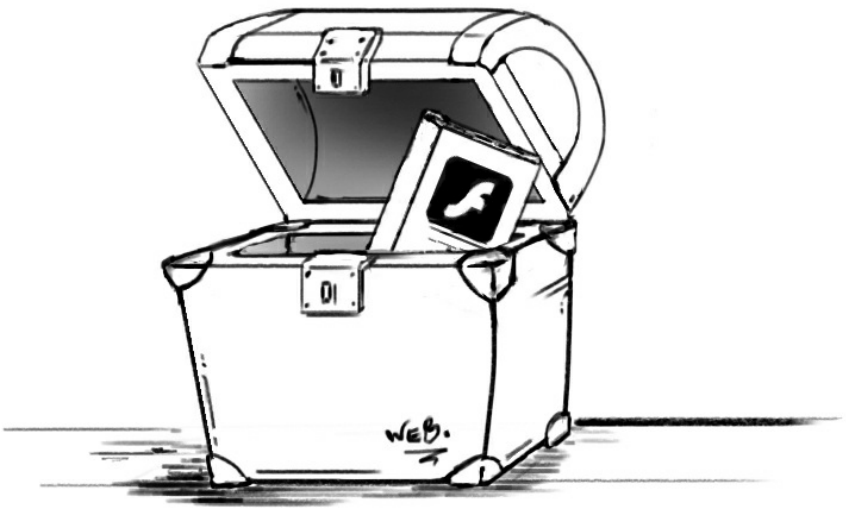
```
// somewhere in our services layer
async function getAllProducts() {
  const res = await fetch('/api/products');
  return res.json();
}
// in a React Component
<button onClick={async () => {
  try {
    const data = await getAllProducts();
    // render the data
  } catch(error) {
    // render error message
  }
}}>
  All products
</button>
```

To fulfill the second rule, I like to create a custom type of errors. While handling the error, instead of relying on the message, it is much better to use

its type. For example:

```
class NoEmailError extends Error {
  constructor() {
    super("Missing email");
    this.name = "NoEmailError";
  }
}
function validatePayload(data) {
  if (!data.email) throw new NoEmailError();
  return true;
}
(function handler() {
  try {
    validatePayload({ name: "Jules" });
  } catch (err) {
    if (err instanceof NoEmailError) {
      console.log(`Error: ${err.message}`);
    }
  }
})();
```

The error handling guarantees that our application works even if something goes wrong. We should not underestimate this part of our development process. Again, we should deal with the error in the right place and we should throw errors that hold context.


WEB.

Blast from the past

I remember the times when copying to a clipboard was done via embedding a small Flash app. Nowadays we have an API for this:

```
// Setting text to clipboard
async function copyToClipboard(text) {
  return navigator.clipboard.writeText(text);
}
// Getting text from clipboard
async function pasteFromClipboard() {
  return navigator.clipboard.readText();
}
```

Another problem that we had with Flash back in the days was the routing. Because the Flash player was just a plugin we didn't have direct access to the address bar. If the user progresses in our application and changes pages there was no way to give him/her a sharable link to that place. Then the term "deep linking" was invented. That was the process of changing the fragment of the browser's URL so we have unique addresses for each page. Today we don't use Flash anymore and we have the History API:

```
window.onpopstate = function(event) {
  console.log(`State: ${JSON.stringify(event.state)}`)
}

history.pushState({foo: "bar"}, "my title", "/foo/bar");
history.back();
```

We didn't have storage APIs too. We could pretty much use only cookies. Today there are a couple of different ways to store data in the user's browser. The most popular one is the local storage.

```
localStorage.setItem("foo", "bar");
// ...
const value = localStorage.getItem("foo");
console.log(value); // bar
```

There is also `sessionStorage` which is similar to `localStorage` but keeps the data only for the current session. Meaning that if the user closes the tab or the browser the data is gone. While the `localStorage` will store the data until it gets deleted via JavaScript or by the user manually.

In case we need more space, there is IndexedDB API. It is a low-level API for storing significant amounts of data. IndexDB is a transactional database

system. Similar to SQL. Here is one not very short example:

```
const indexedDB = window.indexedDB || window.mozIndexedDB || window.webkitIndexedDB || window.msIndexedDB ||
window.shimIndexedDB;

const request = indexedDB.open("UserProfiles", 1);

request.onupgradeneeded = function() {
  const db = request.result;
  const store = db.createObjectStore("User", { keyPath: "id" });
  const index = store.createIndex("NameIdx", ["name", "age"]);
};

request.onsuccess = function() {
  const db = request.result;
  const tx = db.transaction("User", "readwrite");
  const store = tx.objectStore("User");
  const index = store.index("NameIdx");
  store.put({id: 1234, name: "Steve", age: 32});
  store.put({id: 1235, name: "Peter", age: 34});

  const getUser = store.get(1234);
  getUser.onsuccess = function() {
    console.log(getUser.result.name);  // => "Steve"
  };

  tx.oncomplete = function() {
    db.close();
  };
}
```

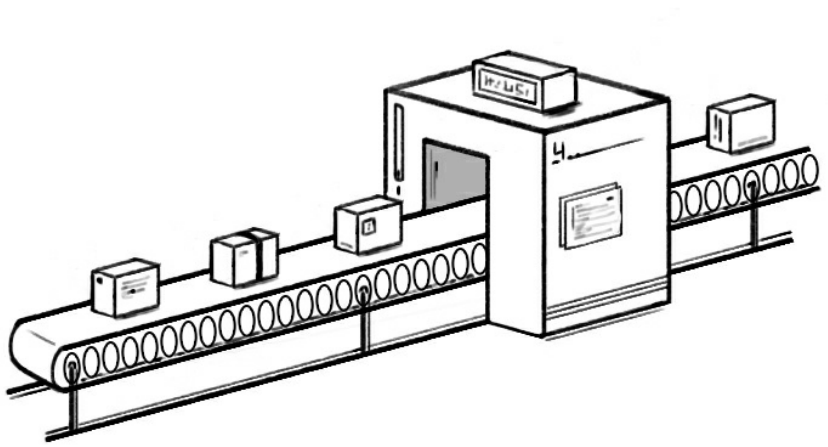
Implementations

Asynchronous Immediately Invoked Function Expression

Early in this book, we mentioned the async/await API. I remember the time when it was introduced. It made my code a bit easier to read and manage. Very often though, I was writing code in the global scope where I couldn't use `await`. In such cases, I reached out to the self-invoked async function.

```
const url = "https://api.thecatapi.com/v1/images/search";
(async function getData() {
  const res = await fetch(url);
  const data = await res.json();
  console.log(data[0].url); // cat image url
})();
```

This pattern is also known as IIFE (Immediately Invoked Function Expression). It is useful for the cases where we want to wrap logic in its dedicated scope. In the example above, `res` and `data` constants are only accessible inside the `getData` function. *(Nowadays V8 JavaScript engine supports a top level `await`. So, we don't have to use a self-invoked function anymore.)*



Asynchronous queue

Sometimes I like to solve problems that are already solved by libraries. One of those problems is handling multiple async operations one after the other. Or in other words, having an API that accepts promises at random intervals and returns the result of all of them. For example:

```
const queue = createQueue();

function nationality(name) {
  return () => fetch(`https://api.nationalize.io/?name=${name}`)
      .then((res) => res.json())
      .then((data) => `${name}: ${data.country[0].country_id}`);
}

queue.add(nationality("Krasimir"));
queue.add(nationality("Natalie"));
setTimeout(() => {
  queue.add(nationality("Hans"));
}, 10);
queue.done.then(console.log);

queue.execute();
// After a while: [ 'Krasimir: BG', 'Natalie: GB', 'Hans: FO' ]
```

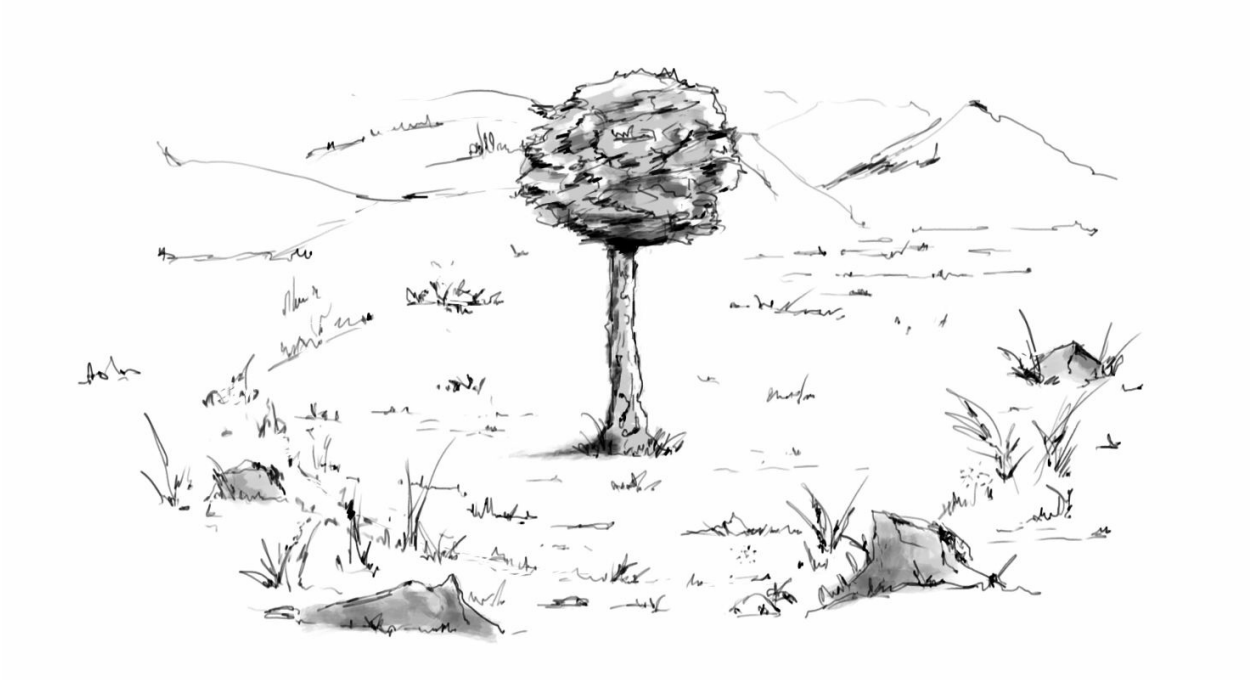
`nationality` is a function that receives a name and asks `api.nationalize.io` where that name comes from. It returns a promise. We are adding all the promises to a queue. At the end, when all the promises are resolved we get the final result.

Here's my implementation of this asynchronous queue.

```
function createQueue() {
  let tasks = [], results = [], processing = false, done = () => {};
  const handle = (res) => {
    results.push(res);
    processing = false;
    execute();
  };
  function execute() {
    if (tasks.length > 0 && !processing) {
      processing = true;
      tasks.shift()().then(handle).catch(handle);
    } else if (tasks.length === 0 && !processing) {
      done(results);
    }
  }
  function add(task) { tasks.push(task); }
  return {
    add,
    execute,
    done: new Promise((cb) => (done = cb))
  }
}
```

It is defining a list of tasks and monitoring when those tasks are done. Whenever an operation is finished we check if there are any pending

promises. If not, we consider the job done. Further optimization may be to run the requests in parallel because now they wait for each other.



JavaScript module system as a singleton

There are different types of scope in JavaScript. One of them is the module scope. It is when we define a variable at the top of the file. In Node, such variable is not part of a function, class, or block scope. In some sense, this is replicated on the client when we bundle our code. Those are not accessible outside the file/module. We have to export them. Another characteristic is that they are cached. Meaning that no matter how many times we import/require the module, the top-level code executes only once. This allows us to implement the singleton pattern (later in the book we will talk about that in greater depth).

Let's say that we have a file called `registry.js` with the following content.

```
const users = [];
module.exports = {
  register(name) {
    users.push({ name });
  },
  total() {
    return users.length;
  },
};
```

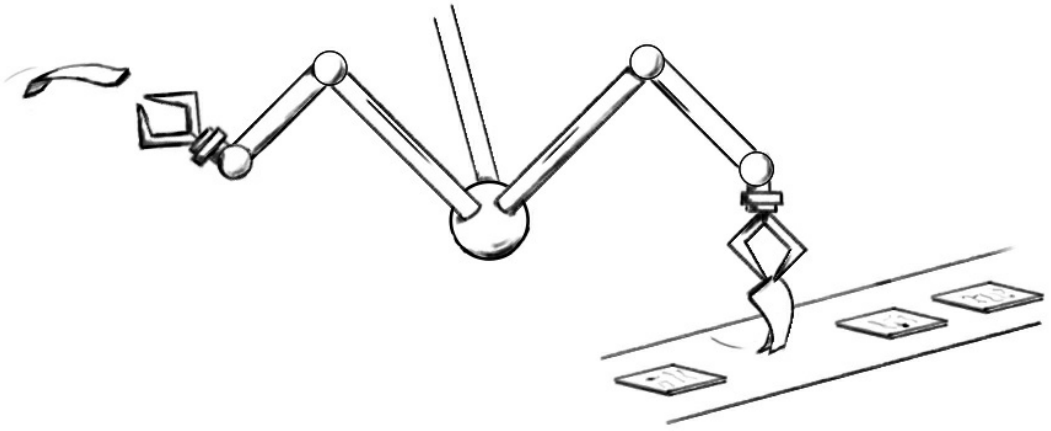
We defined the `users` constant at the top level, so it's cached. It is initialized only once. Now, let's create two other files/modules `A.js` and `B.js` that will import `registry.js`.

```
// A.js
const R = require("./registry");
R.register("Dave");
module.exports = R;

// B.js
const R = require("./registry");
R.register("Alex");
module.exports = R;
```

The `R` constant in both files is the same object. Here's the proof - we will create `index.js` that will import all three files. It will run the `total` function from the `registry.js` module.

```
const A = require("./A");
const B = require("./B");
const { total } = require("./registry");

console.log(total()); // 2
console.log(A === B); // true
```

The first log proves that we create the `users` array only once, and when we call the `register` function, we are using that same instance. The second log tells us that the exported value in the registry is cached.



Call-to-action widgets script tag replacement

Did you ever wonder how the call-to-action widgets work? You know, those little buttons for sharing content on social networks. Very often they are distributed as iframes, but sometimes we have to copy/paste script tags. And the instructions are “Place the following code where you want the button to appear”. But if that is a script tag, how does it know where to inject the button? The goal of this chapter is to give you an answer to this question.

Let’s start with the markup below:

```
<script src="./widget.js" data-color="#ff0000"></script>
<section>
  Content here
</section>
<script src="./widget.js" data-color="#00ff00"></script>
```

We want to see is a link followed by “Content here” followed by another link. Notice, that we not only want to replace those script tags. We want different results for each one of them. The color of the button should be different.

I was surprised to find out that the code behind `widget.js` could be quite short. Just 8 lines:

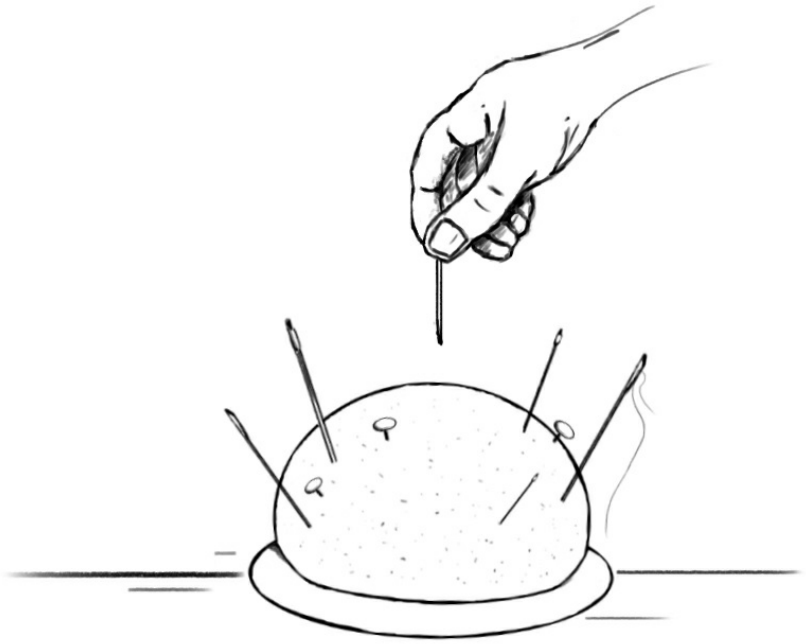
```
(async function loadContent() {
  const options = document.currentScript.dataset;
  const node = document.createElement('div');
  const style = options.color ? `color:${options.color}` : '';

  node.innerHTML = `<a href="http://mysite.com" style="${style}">click me</a>`;
  document.currentScript.parentNode.replaceChild(node, document.currentScript);
})();
```

The APIs that are used are `document.currentScript` and `element.dataset`. The first one gives us access to the element whose script we are currently processing. The `dataset` property is quick access to the data attributes of the element.

The snippet above creates a new div and injects in it a link. Then, by using `replaceChild` it swaps the script tag with that newly created element. The result is:

```
<div><a href="http://mysite.com" style="color:#ff0000">click me</a></div>
```

```
<section>
  Content here
</section>
<div><a href="http://mysite.com" style="color:#00ff00">click me</a></div>
```

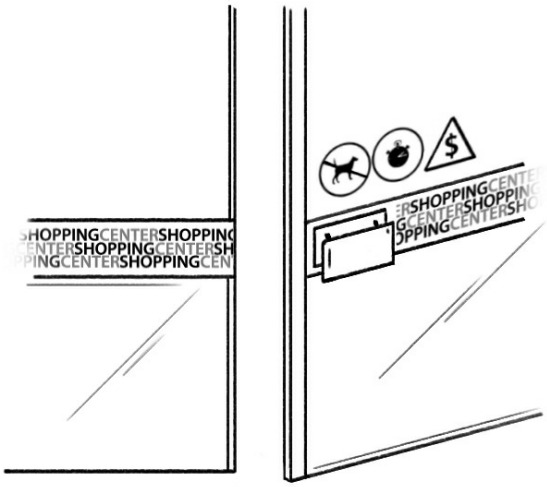


Removing fields from objects

Earlier in this book, we learned about the destructing assignment. An interesting use case of this feature is that it allows us to remove a field from an object.

```
function allButPoints(obj) {
  const { points, ...rest } = obj;
  return rest;
}

const user = {
  firstName: "David",
  lastName: "Bird",
  points: 231,
};
console.log(allButPoints(user));
// { firstName: 'David', lastName: 'Bird' }
```

I use that a lot in the world of React where a component receives a bunch of props, but I need to pass down just a few of them.

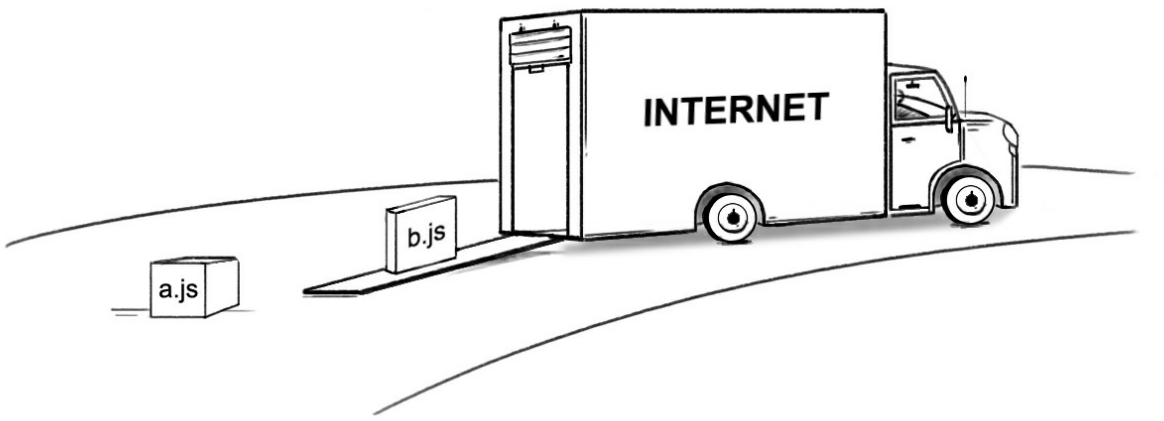

SHOPPINGCENTERSHOPPING
ENTERSHOPPINGCENTERSH
PINGCENTERSHOPPINGCEN
$

A must have function argument

As we know, JavaScript is not a strictly typed language. We don’t have types by default. That is why a big part of the community started using alternatives (like TypeScript). Which is a good solution, but most of them work at build time. Once our code is transpiled and bundled the type checks are gone. Some time ago I found a neat way to make a function argument required. Even at runtime.

```
function required() {
  throw new Error(`Missing argument.`);
}
function shoppingCenter(time, money = required()) {
  return {
    time,
    money,
  };
}

console.log(shoppingCenter("1h", 200));
// { time: '1h', money: 200 }

console.log(shoppingCenter("2h30min"));
// Error: Missing argument.
```

This may be a bit drastic because throwing an error may lead to an application crash. However, with proper error handling is I believe a good solution.


INTERNET
b.js
a.js

Loading JavaScript file dynamically

It’s funny how in the past we had to write a lot of things alone. There was no NPM or GitHub. We had to write our loader. A utility that loads JavaScript files dynamically. Today this problem is solved by numerous packages, but I figured will be interesting to share:

```
function loadJS(files, done) {
  const head = document.getElementsByTagName('head')[0];
  Promise.all(files.map(file => {
    return new Promise(loaded => {
      const s = document.createElement('script');
      s.type = "text/javascript";
      s.async = true;
      s.src = file;
      s.addEventListener('load', (e) => loaded(), false);
      head.appendChild(s);
    });
  })).then(done);
}

loadJS(["a.js", "b.js"], () => {
  console.log('Loading completed.');
});
```

The solution is to add a `<script>` tag and use the `load` event to understand when the file is downloaded.

Popular concepts

Readability

There are tons of different opinions on what is a good code. For me, the best answer is “Good code is the one that I and my teammates understand”. Readability comes to play a lot here. Some tips could make our code easier to follow.

1. When writing a function, return early. In other words, we should make our code fails fast. Check out the following example:

```
if (status === 200 || status === 202) {
    // ok
} else {
  if (status === 500) {
    // internal server error
  } else if (status === 400) {
    // not Found
  } else {
    // generic error
  }
}
```

Some people advise starting with the happy path, but I’m finding the version below better.

```
if (status === 500) {
  // internal server error
}
if (status === 400) {
  // not Found
}
if (status !== 200 && status !== 202) {
  // generic error
}
// ok
```

2. Avoid sending flags as arguments to a function. That is not because the function itself is not readable. That is because the place where you call it looks unclear.

```
function saveUser(profileData, isAdmin) {
  const user = { ...profileData, admin: isAdmin };
  // ...
}

saveUser({ name: '...' }, false);
```

See how the second argument, `false` makes you a bit nervous. That is because you don’t know what it does. To solve this problem, we may wrap that flag into an object. Like so:

```
function saveUser(profileData, { isAdmin }) {
  const user = { ...profileData, admin: isAdmin };
  // ...
```

```
}

saveUser({ name: '...' }, { isAdmin: false });
```

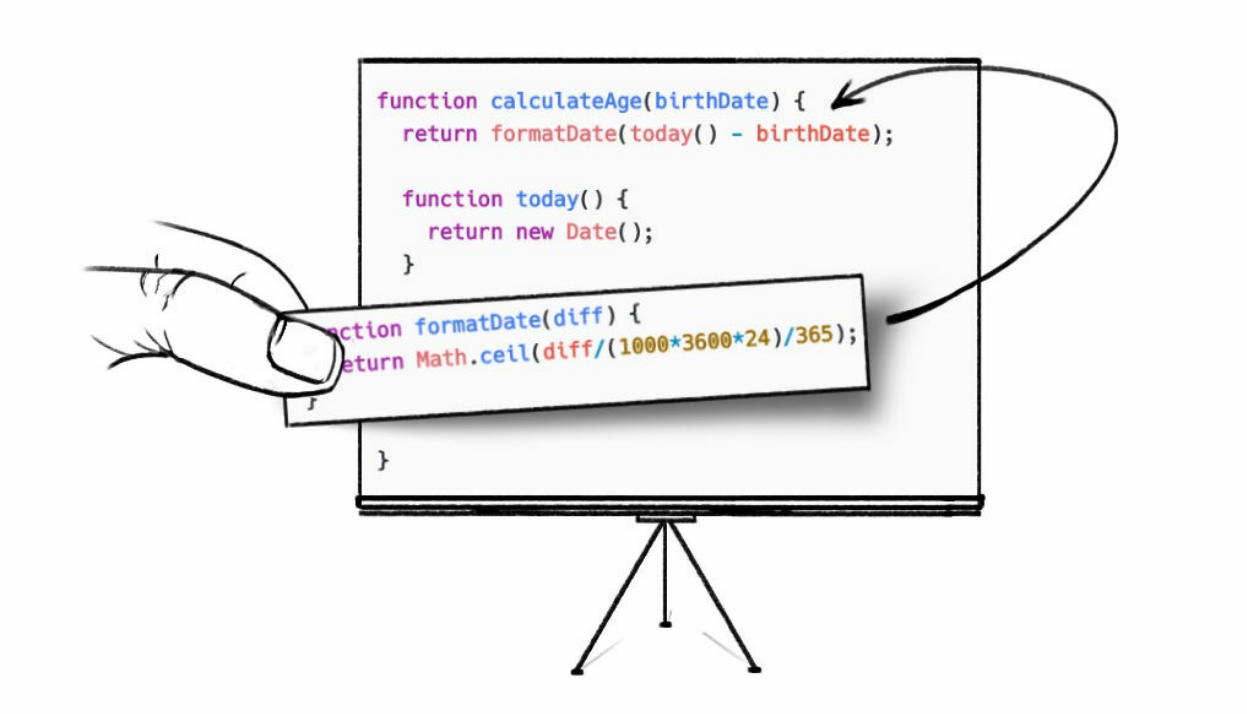
3. Last but not least, I want to mention the naming. We all know that this is one of the most challenging tasks in programming. JavaScript is not an exception. We need to name our variables and functions properly, so they bring context to the reader.

```
const arr = ['BE:Node', 'BE:PHP', 'FE:HTML', 'BE:Python', 'FE:CSS'];
const arrFiltered = arr.filter(i => i.startsWith('FE:'));

function getText(items) {
  const str = `Front-end: ${items.map(i => i.replace(/^FE:/, '')).join(', ')}`
  return str;
}
getText(arrFiltered); // Front-end: HTML, CSS
```

This code is not bad but what if we change the naming a bit:

```
const languages = ['BE:Node', 'BE:PHP', 'FE:HTML', 'BE:Python', 'FE:CSS'];
const FELanguages = languages.filter(lang => lang.startsWith('FE:'));

function formatLanguagesText(languages) {
  const str = `Front-end: ${
    languages.map(lang => lang.replace(/^FE:/, '')).join(', ')
  }`
  return str;
}
formatLanguagesText(FELanguages); // Front-end: HTML, CSS
```

`arr` and `arrFiltered` constants are just too generic and will quickly lose meaning. `getText` function is indeed about generating a string, but again it doesn't bring enough context. So, `languages`, `FELanguages` and `formatLanguagesText` are a bit longer but give a better idea of what we mean with this code.


function calculateAge(birthDate) {
return formatDate(today() - birthDate);
function today() {
return new Date();
}
ction formatDate(diff) {
eturn Math.ceil(diff/(1000*3600*24)/365);
}

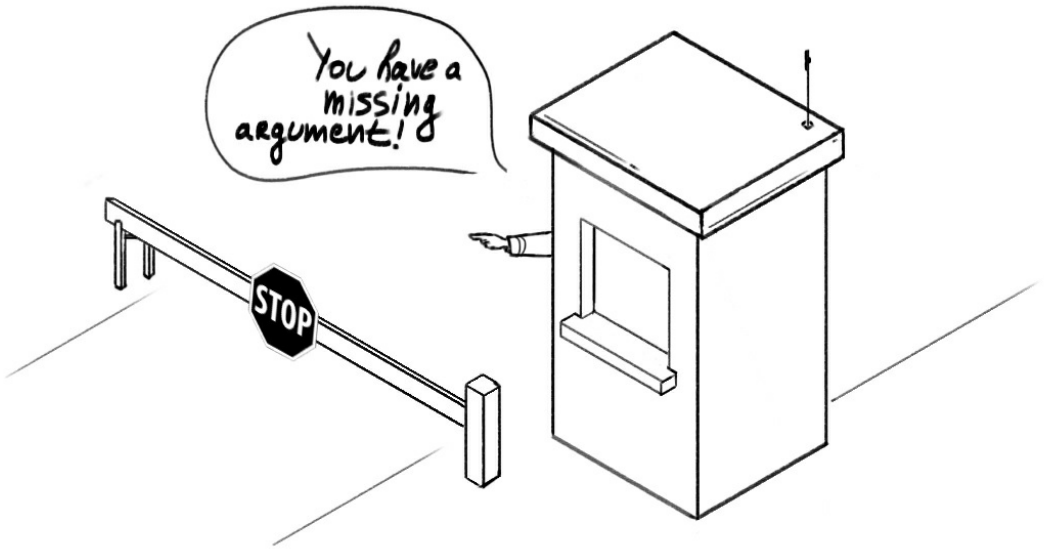
The return statement is not the end

Even though I’m writing JavaScript for many years, I’m still learning. And I’m getting surprised by simple things. It could be a pattern or just a style of writing.

```
function calculateAge(birthDate) {
  return formatDate(today() - birthDate);

  function today() {
    return new Date();
  }

  function formatDate(diff) {
    return Math.ceil(diff / (1000 * 3600 * 24) / 365);
  }

}

const age = calculateAge(new Date(1984, 1, 1));
console.log(`You are approx ${age} years old.`);
// result: You are approx 38 years old
```

See how in `calculateAge` we have the return statement quite early. After that though, we have two function definitions. This example shows us that the code after the `return` statement is not dead. Some things are getting *hoisted* at the top of the function, and we can use them.

Hoisting is the engine mechanism where it allocates memory before the code execution. Or in other words, the interpreter “moves” things at the top of the current scope. In our case the `today` and `formatDate` functions.


You have a missing argument!
STOP

Always get a value

JavaScript by default has no type definitions. It has been like that since day one. Today we can solve that by using TypeScript. However, as we mentioned a few chapters back, those are solutions that work at build time. When our code runs in the browser, we still have to deal with the good old vanilla JavaScript.

Let’s say that we face up a situation where we really must have value. For the functions this means setting up a default value like so:

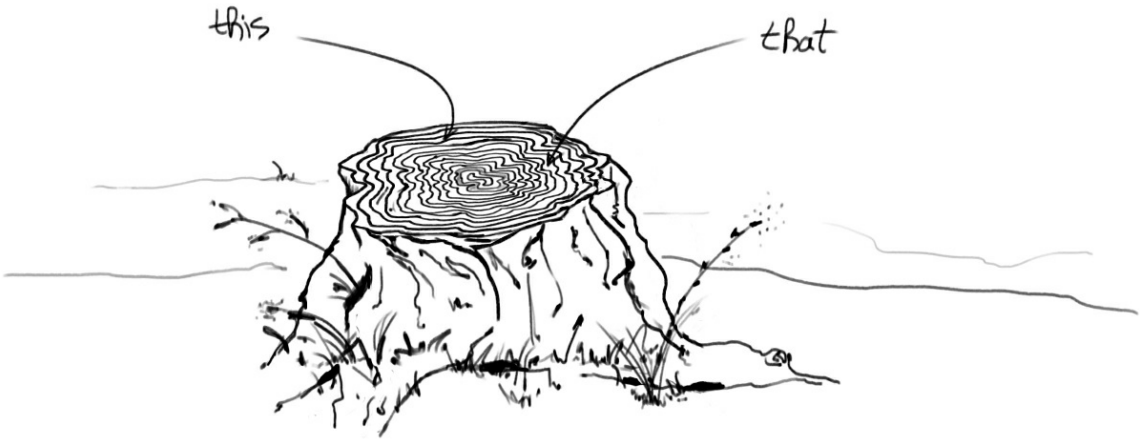
```
function calculate(value = 0) {
  return value * 2;
}
```

For everything else we may use the logical OR operator:

```
const user = { age: 37 };
const name = user.firstName || "unknown";
console.log(name);
```

These two options work if we have a default value. But what if we don’t have one, and we have to force the developer to pass an argument. We can do it via this neat trick.

```
const required = () => {
  throw new Error("Please provide a number.");
};
const calculate = (value = required()) => {
  return value + 42;
};
console.log(calculate(8)); // 50
console.log(calculate(28)); // 70
calculate(); // throws error
```


this
that

This

In every JavaScript book or course, there is always one chapter for the `this` keyword. These days I feel that the community pushes the language to more functional paradigms. We are not using the `this` keyword so much. And I bet it still confuses people because its meaning is different based on the context. So, let's start with a few examples where `this` refers to the global scope:

```
function A() {
  console.log(this); // this = global object (window)
}
const B = () => console.log(this); // this = global object (window)
const C = {
  method: () => console.log(this), // this = global object (window)
};
```

In all these cases, we call the functions without having a concrete context. That is why the global scope is picked. *(Here we have to mention that in strict mode, `this` will be equal to `undefined`)*

When the function is part of a class or an object, we have context. Then `this` points to the specific object or instance of the class.

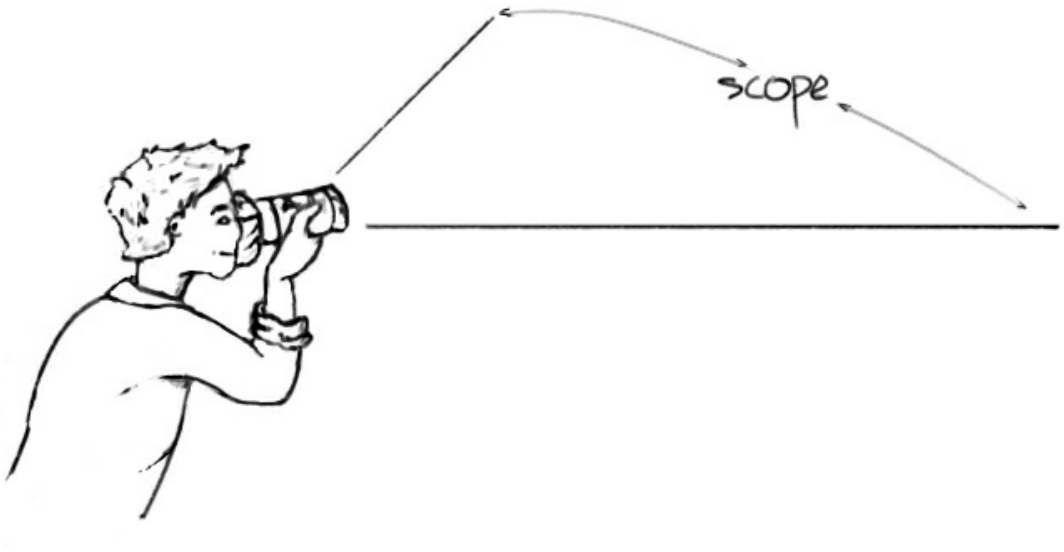
```
class D {
  run() {
    console.log(this); // this = instance of the class
  }
}

const E = {
  method() {
    console.log(this); // this = the constant E itself
  },
};

function F() {
  console.log(this); // this = an instance of the F prototype
}
new F();
```

JavaScript also offers the ability to set the scope manually. The first argument of `apply` and `call` becomes `this` of the function.

```
const foo = { id: 'foo' };
function G() {
  console.log(this); // this = foo
}
G.apply(foo);
G.call(foo);
```

And at the end, if we don’t want to call the function but define its `this`, we can use the `bind` method. We will talk about this API in the next chapters.

```
function H() {
  console.log(this.name);
}
const HFunc = H.bind({ name: 'foobar'});
HFunc(); // foobar
```


scope

Scope

At the job interviews for JavaScript developers, two questions are almost always asked - the first one is about the `this` keyword (covered in the previous chapter), and the other one is about scope. I like to think that the scope is an environment that holds a specific set of variables, constants, functions, etc. And as part of that environment, all the actors inside are visible to each other. Understanding scope is important mainly because of that visibility. We have to know well what we can access and whatnot.

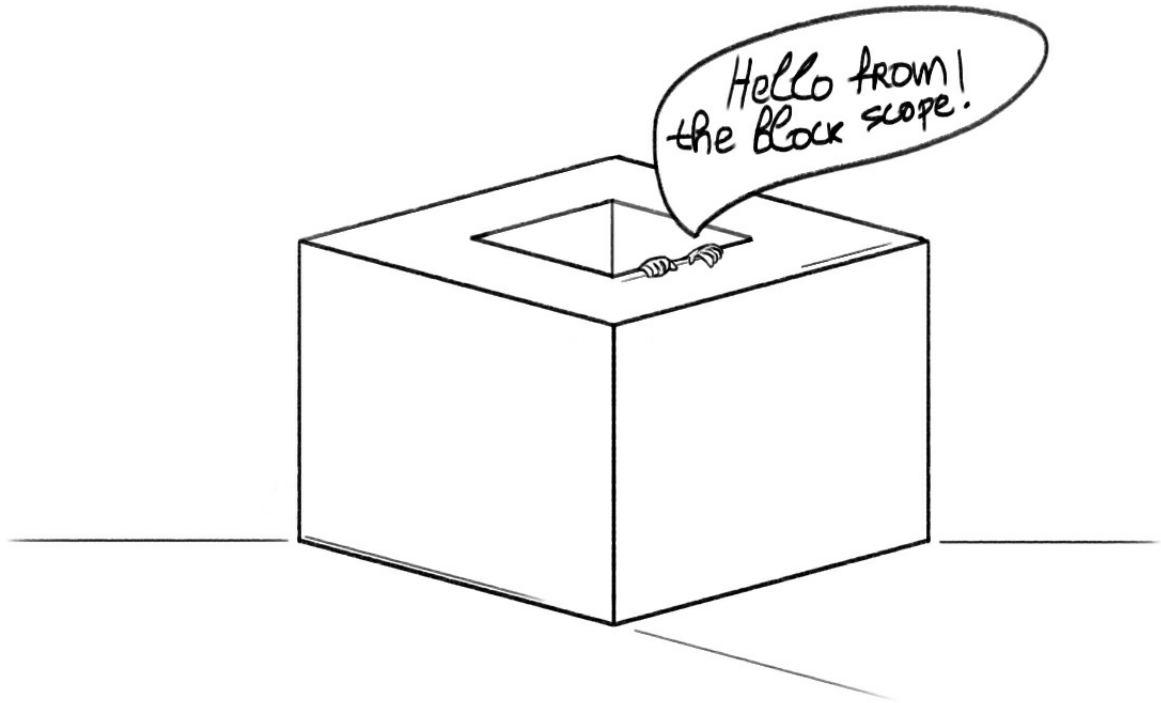
There are four different scopes: global, function, block, and module. Let's illustrate each one of them with a quick example:

```
let total = 0;
function calculate(a, b, c) {
  const sum = a + b + c;
  if (a > b) {
    const diff = a - b;
    return diff + c;
  }
  return sum;
}
```

`total` live in the global scope. Being part of that environment, it is accessible (visible) for all the nested environments (scopes). For example, we can use it from within the `if` statement. `sum` lives in the `calculate` function scope. It is not accessible outside that function. `diff` is defined in a block scope, and it is not accessible outside the `if` statement. The module scope is the one that is usually defined by a file.

```
// A.js
let inc = 0;
export default function calculate(a, b, c) {
  inc += 1;
  return a + b + c;
}

// B.js
import calculate from './A.js';
console.log(calculate(1, 2, 3));
```

In this example, the code in file `B.js` doesn't have access to the `inc` variable defined in `A.js` file.


Hello from!
the Block scope.

Manually creating block scope

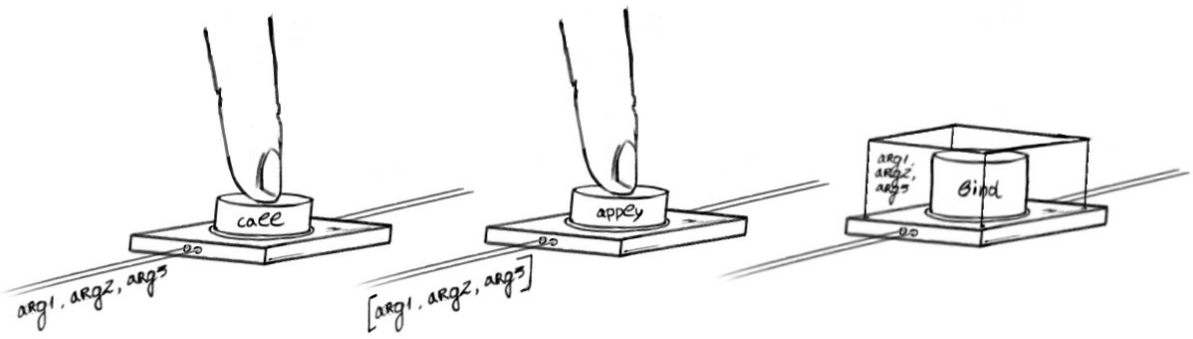
We saw the four different types of scope. I want to share a small tip regarding the block scope. And specifically when we need to create it on purpose. I rarely see the need for this, but it is good to know. Check out the following situation:

```
function test(operation) {
  const message = "In progress.";
  const { message, value } = operation;
  // throws: Identifier 'message' has already been declared
}
```

`message` is a constant that is defined in the `test` function scope. We can't destruct the `operation` argument because there is a naming clash.

The quick fix for that is to create an alias, but we can also solve it by using a block scope.

```
function test(operation) {
  const message = "In progress.";
  {
    const { message, value } = operation;
  }
}
```


call
apply
bind
arg1, arg2, arg3
[arg1, arg2, arg3]
arg1,
arg2,
arg3

Call, apply and bind

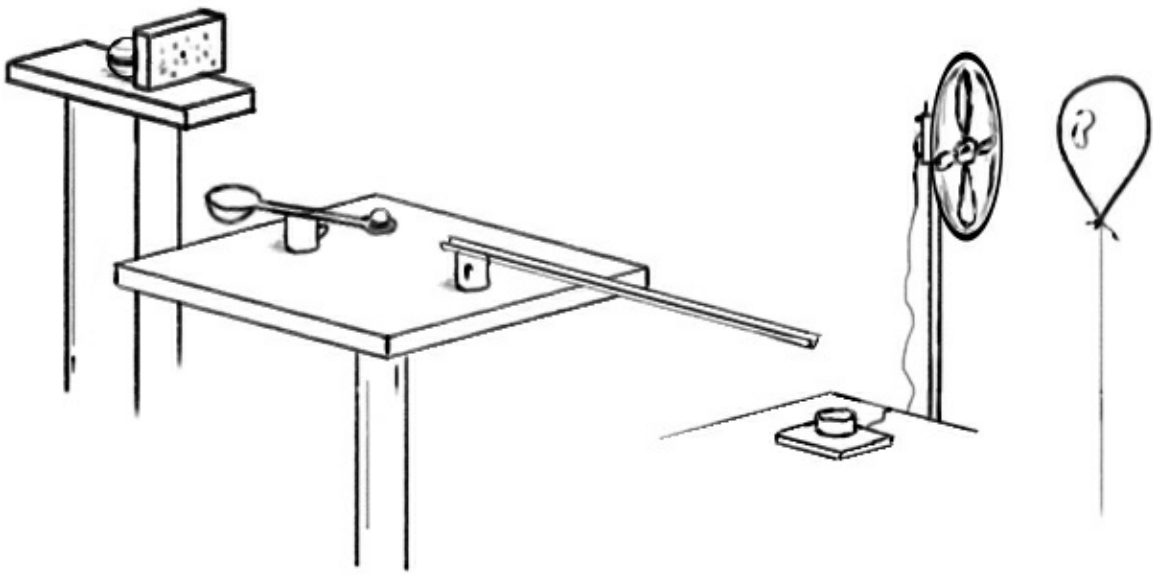
Now when we know what scope is, it's time to explore the `call`, `apply`, and `bind` methods. They are all touching on the topic of context in JavaScript, because they define the `this` of the passed functions. Consider the following example:

```
const user = { firstName: "Krasimir" };

function message(greeting) {
  console.log(`${greeting} ${this.firstName}!`);
}
message('Hey'); // Hey undefined!
```

We want the `message` function to have `user` as a context. This so we can use `this.firstName`. All the three functions above can solve that problem.

```
const user = { firstName: "Krasimir" };

function message(greeting) {
  console.log(`${greeting} ${this.firstName}!`);
}
message.call(user, 'Hey'); // Hey Krasimir!
message.apply(user, ['Hi']); // Hi Krasimir!
message.bind(user, 'Hola')(); // Hola Krasimir!
```

`call` accepts the desired `this` as a first argument followed by other parameters of the function. `apply` works the same way except that the additional parameters we pass as an array. `bind` is a bit different since it doesn't execute the function immediately. It does a partial application (something that we will talk about later). The result of `bind` is another function that we can run with predefined parameters.



Chaining

Back in the days when the Web was all made of jQuery, we were constantly using a pattern - function (or method) chaining. It looks like that:

```
$("#p1")
  .css("color", "red")
  .slideUp(2000)
  .slideDown(2000);
```

The first line (`$("#p1")`) selects a DOM element. The rest is changing the color and animating it.

We should consider method chaining when we have a lot of small functions, and they have to be executed on a single object.

Let’s see how the implementation of this pattern looks like. We will define a shopping cart factory function:

```
function ShoppingCart() {
  const products = [];
  function add(product, price) {
    products.push({ product, price });
  }
  function total() {
    return products.reduce((res, product) => (res += product.price), 0);
  }
  return { add, total }
}
```

We may create a shopping cart, add products and get the total price at the end.

```
const cart = ShoppingCart();
cart.add("t-shirt", 50)
cart.add("backpack", 120)
cart.add("socks", 7)
console.log(cart.total()); // 177
```

Now let’s make the `add` method chainable. To do that we have to return an object that itself has this `add` method. This is so we can call it again and again.

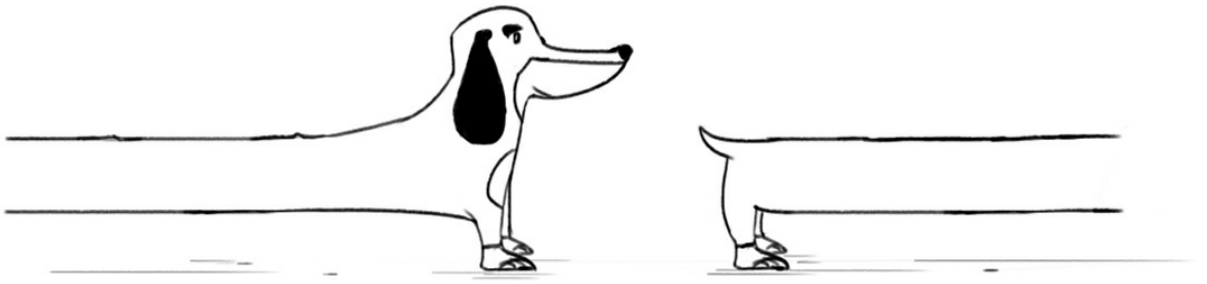
```
function ShoppingCart() {
  const products = [];
  const api = { add, total };
  function add(product, price) {
    products.push({ product, price });
    return api;
  }
  function total() {
    return products.reduce((res, product) => (res += product.price), 0);
```

```
  }
  return api;
}
```

This will allow us to rewrite our example as:

```
const cart = ShoppingCart();
const total = cart
  .add("t-shirt", 50)
  .add("backpack", 120)
  .add("socks", 7)
  .total();
console.log(total); // 177
```

Notice that `total` is not the same as `add` because it doesn’t return the same `api` object.



Recursion

Recursion is probably one of the oldest concepts in programming. It's the paradigm where a function calls itself. We usually use this technique to solve problems that require breaking them into smaller subproblems.

In JavaScript, my favorite use case is reading a deeply nested object field. Let's say that we have the following:

```
const user = {
  profile: {
    age: 36,
    name: { first: "Krasimir", last: "Tsonev" },
  },
};
```

We want to read the `last` name of the user. We have to write `user.profile.name.last`. And of course, if some of the data is missing we will get:

```
Cannot read properties of undefined (reading 'last')
```

To solve this problem, we use helpers like lodash's `get` method.

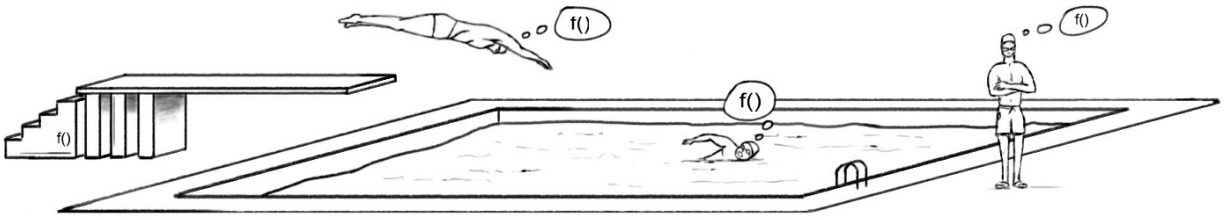
```
get(user, 'profile.name.last', 'unknown');
```

This utility safely tries to read the value and, if it doesn't exist returns the `"unknown"` string.

Here is how the implementation of such utility may look like:

```
function get(obj, path, fallback) {
  const parts = path.split(".");
  const key = parts.shift();
  if (typeof obj[key] !== "undefined") {
    return parts.length > 0 ?
      get(obj[key], parts.join("."), fallback) :
      obj[key];
  }
  return fallback;
}

console.log(get(user, "profile.name.first")); // Krasimir
console.log(get(user, "profile.age")); // 36
console.log(get(user, "profile.registered")); // undefined
console.log(get(user, "profile.registered", false)); // false
```

Notice how `get` calls itself again and again till it reaches the last part of the path.


f()
f()
f()
f()

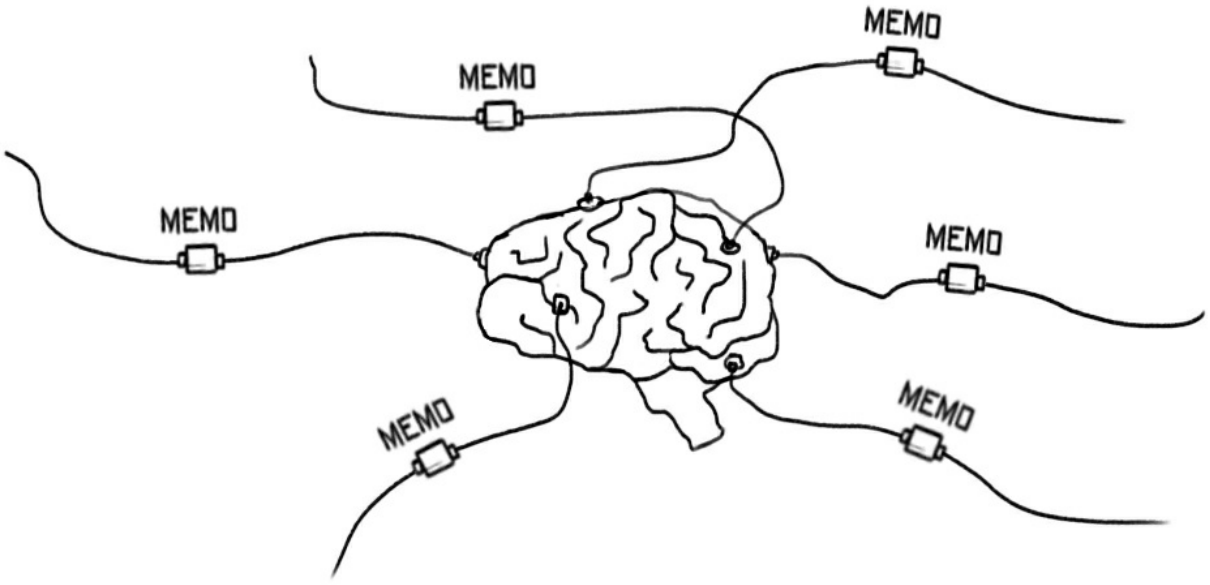
Higher order functions

The main building block of every application is the function. You've probably heard that the functions in JavaScript are first-class citizens. This means that we can assign a function to a variable and pass that variable to a method or return it as result. For example:

```
function getProducts(fetchData) {
  return async (categoryId) => {
    const data = await fetchData({ id: categoryId });
    return data.products;
  }
}
function fetchData(query) {
  return fetch(`https://site.com/api/products?id=${query.id}`);
}

const byCategoryId = getProducts(fetchData);
const shoes = await byCategoryId('XYZ');
```

Notice that `fetchData` is a function and we are passing as an argument to `getProducts`. Internally `getProducts` returns another function. In such cases, we say that `getProducts` is a higher-order function.

We are writing higher-order functions all the time. And that is because they are the natural abstraction on top of the smaller one-job methods. We rarely want to write all the logic in one place so we break it into smaller reusable functions. Then we need a glue code that operates with them. Very often that glue code consists of higher-order functions.


MEMO
MEMO
MEMO
MEMO
MEMO
MEMO
MEMO

Memoization

The previous chapter showed us that in JavaScript the main instrument is the function. And if something is getting run again and again it makes sense to try optimizing it. One way to achieve that is through *memoization*.

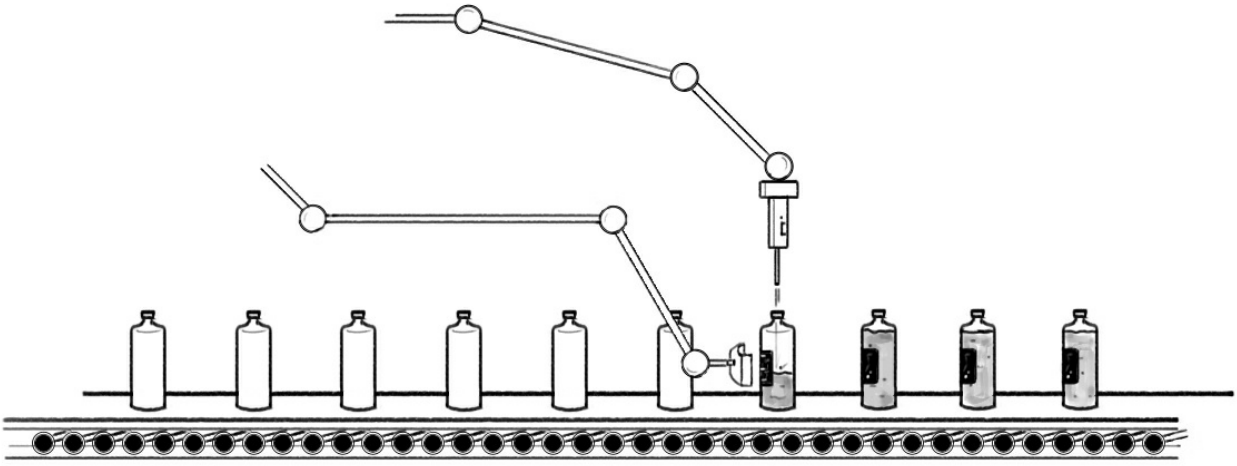
Memoization is capturing and storing the previous execution. Of course, it depends on what the function does, but sometimes we assume that if the arguments are the same the result will be also the same. In such cases, we could cache the output and return it immediately.

```
function pythagorean(a, b) {
  return Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
}
```

The `pythagorean` function does synchronous computation. The result doesn't change if the arguments are the same. It is the perfect candidate for memoization.

```
function pythagorean(a, b) {
  console.log('Doing the job ...');
  return Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
}
function memo(func) {
  var cache = {};
  return function () {
    var key = JSON.stringify(arguments);
    if (cache[key]) {
      return cache[key];
    } else {
      return (cache[key] = func.apply(null, arguments));
    }
  };
}

const mPythagorean = memo(pythagorean);

console.log(mPythagorean(4, 3)); // "Doing the job ..." followed by "5"
console.log(mPythagorean(4, 3)); // only "5"
console.log(mPythagorean(4, 3)); // only "5"
console.log(mPythagorean(4, 3)); // only "5"
```

Only the first call of `mPythagorean` does the actual work. The rest calls are cheap because `memo` returns the cached result.



Partial application

Developers that used to write in functional languages feel good with JavaScript. The language supports functional concepts. One of them is partial application.

Partial application is when we duplicate a function but with some of its arguments already applied. It sounds a bit weird, but it is a useful technique. Consider the following function:

```
function createLogMessage(type, message) {
  return `${type}: ${message}`;
}
```

and some of its potential usage:

```
console.log(createLogMessage('Error', 'Ops!'));
// Error: Ops!
console.log(createLogMessage('Info', 'Data delivered.'));
// Info: Data delivered.
```

Think about how we have to use `createLogMessage` everywhere in our application. This means specifying the type all the time. We can extract that to a constant, but it is still a duplication.

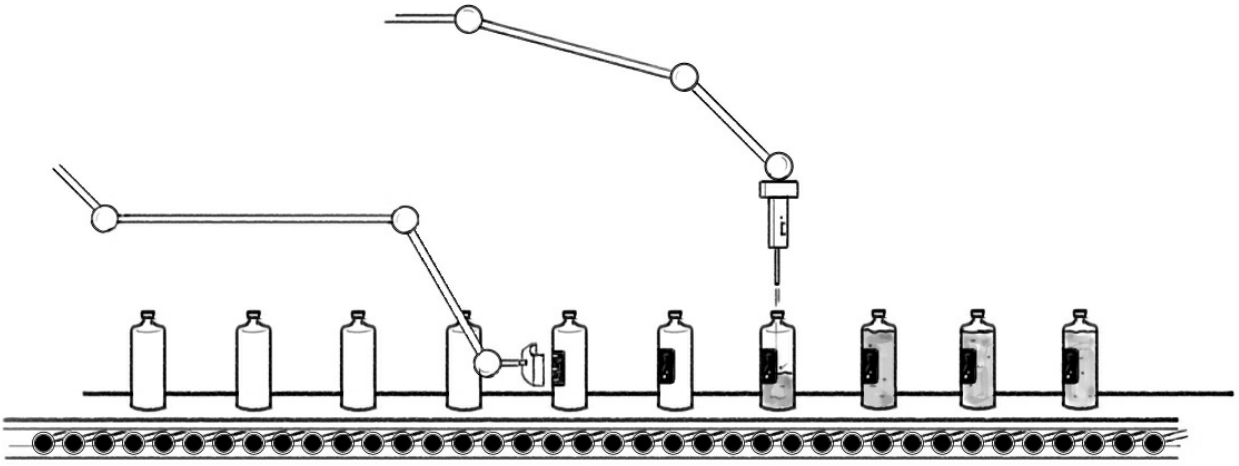
We can make our life easier by using the partial application.

```
function createLogMessage(type, message) {
  return `${type}: ${message}`;
}
function partial(func, ...args1) {
  return (...args2) => {
    return func(...args1, ...args2);
  }
}
const onError = partial(createLogMessage, 'Error');
const onInfo = partial(createLogMessage, 'Info');

console.log(onError('Ops!'));
// Error: Ops!
console.log(onInfo('Data delivered.'));
// Success: Data delivered.
```

Notice how we abstracted the original `createLogMessage` function. We not only avoid repeating ourselves, but we have these little helpers `onError` and `onSuccess`.

JavaScript has a built-in method that is doing a partial application. We mentioned it when we were talking about scope. That is the `bind` function. This last snippet we could replace with the following:

```
const onError = createLogMessage.bind(null, 'Error');
const onInfo = createLogMessage.bind(null, 'Info');
```



Currying

Another popular functional programming paradigm is currying. It is kind of similar to the partial application in the sense that it allows us to split the function call. However, this time we apply one argument at a time. While with a partial application, we may do multiple arguments for each call. Look at the `update` function below. It has three arguments:

```
const user = { age: 33, job: "developer" };

function update(user, prop, value) {
  user[prop] = value;
}
update(user, "age", 36);
```

Ideally, we don't want to pass the `user` object over and over again. A good step forward is to create a function that has it already applied and only accepts `prop` and `value`. We can do that by using partial application.

```
const user = { age: 33, job: "developer" };
function update(user, prop, value) {
  user[prop] = value;
}
const updateUser = partial(update, user);
updateUser("age", 36);

function partial(func, ...args1) {
  return (...args2) => {
    return func(...args1, ...args2);
  }
}
```

Now, let's transform this example to currying:

```
const user = { age: 33, job: "developer" };
function update(user, prop, value) {
  user[prop] = value;
}
const curriedUpdate = curry(update);
const updateUser = curriedUpdate(user);
const updateAge = updateUser('age');
updateAge(36);

function curry(f) {
  return function curried(...params) {
    if (params.length >= f.length) {
      return f.apply(this, params);
    } else {
      return function(...params2) {
        return curried.apply(this, params.concat(params2));
      }
    }
  };
}
```

Instead of `partial` we now have `curry`. This is a helper that returns a curried version of our functions. The idea is that our original function with N arguments is transformed into N functions with one argument.

f( )

Dependency injection

Dependency injection (DI) or inverse of control (IoC) is a concept that is not popular in the JavaScript ecosystem. In some cases, the framework pushes the developer to use some sort of DI. But otherwise, solving dependency management is not among the core problems of today's projects. To better understand the need for DI check the example below:

```
async function postToFacebook(message, service, settings) {
  const client = new Service({ APIKey: settings.fb.key });
  const result = await client.post(message);
  return result;
}
```

This function that posts a message to the popular social network has three dependencies. The first one is the message. We have to leave that because it is different every time. We don't know that up front. The other two are known dependencies. We know what service we will be using and, we also know the API key. So, now the question is: "Is it a job of this function to initiate the service?". Probably no. Another inconvenient fact is that we have to carry the `Service` class and the `settings` to every place where we use this function. We have a design problem.

One of the solutions is to rely on dependency injection and create the service in another place. Then *deliver* it to the function when we need it. Implementation wise we need a DI container. Something like:

```
const Container = {
  _storage: {},
  register(key, deps, func) {
    this._storage[key] = { deps, func };
  },
  get(key) {
    if (!this._storage[key]) throw new Error(`Missing ${key}`);
    const { func, deps } = this._storage[key];
    return (...args) => {
      const resolvedDeps = deps.map((key) => this.get(key));
      return func(...resolvedDeps, ...args);
    };
  },
};
```

With that, we can start and *register* our dependencies. The first one will be `settings` and then the `Service`.

```
Container.register('settings', [], () => {
```

```
  return { fb: { key: 'xxx' } }
});
Container.register('client', ['settings'], (settings) => {
  return new Service({ APIKey: settings().fb.key });
});
```

After this code, our container *knows* about all the dependencies of `postToFacebook` function. Notice that we are also defining a connection between the `client` and the `settings`. This is another dependency problem that we solve right away here at the container level.

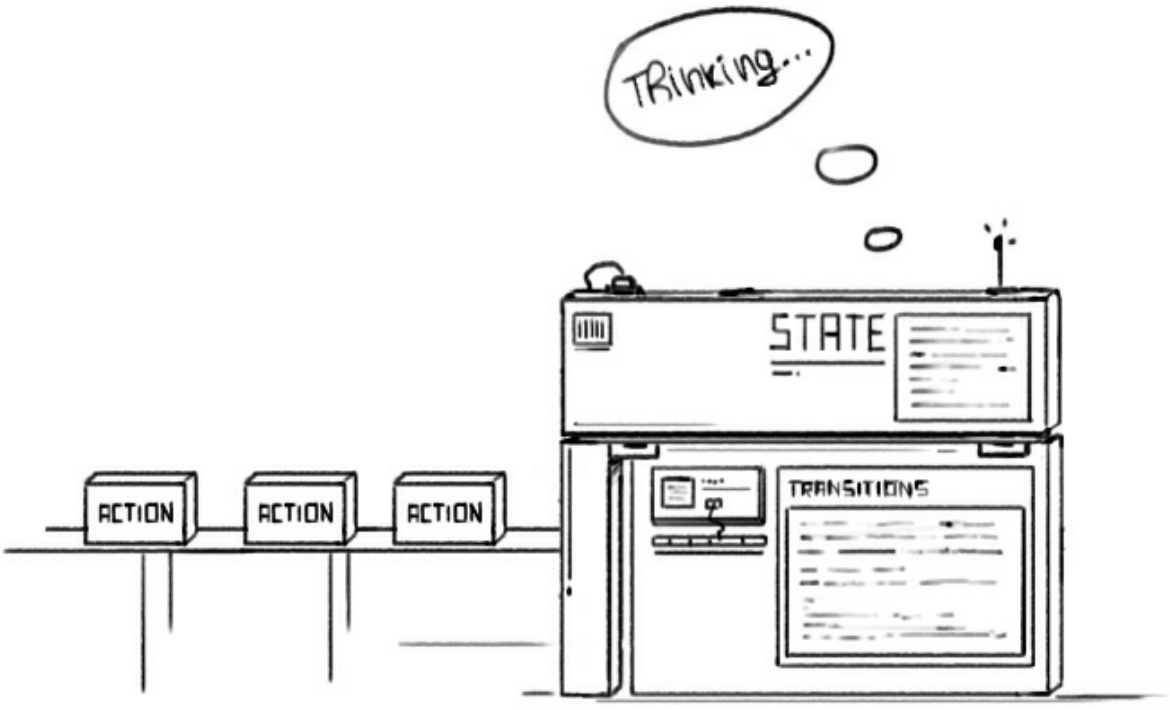
The next step is to register `postToFacebook`. This last step is important. It is the moment where we let the container inject the dependencies.

```
Container.register('postToFacebook', ['client'], async (client, message) => {
  const result = await client().post(message);
  return result;
});
```

The dependencies appear first in the list of arguments. After that, we have the function parameters. In our case, this is the `message` string. Here is how the code that uses the container looks like:

```
const publish = container.get('postToFacebook');

publish('What a beautiful day!');
```

The `get` method of the container resolves the needed dependencies and, we get back a function that requires just the `message`.


THinking...
STATE
TRANSITIONS
ACTION
ACTION
ACTION

State machines

I believe that state machines must be taught equally as every other design pattern in programming. It is interesting that, once I found this concept, I started seeing state machines everywhere. Especially in the development of user interfaces where we have to manage a lot of states. But what is a state machine? The mathematical explanation is that the state machine is a model of computation. The non-mathematical (my explanation) is that the state machine is a box that keeps your state and changes it based on input and current value.

There are different types of state machines. The one that we are interested in is called a finite state machine. As the name suggests, it has a finite number of possible states. Think about a map of what conditions your app may be and strict lines between the different values.

To illustrate the concept, we will write software for an elevator. Our elevator is a simple one and can be in the following states: “idle”, “moving”, and “broken”.

So, this is a finite number (3) of possible states. We can also define the transitions from one state to another:

```
idle --- move --> moving
moving --- stop --> idle
moving --- error --> broken
```

Here we can see one of the biggest benefits - if we implement the machine correctly, it will prevent wrong transitions. For example, we can’t go from “idle” to a “stop” state or from “move” directly to “idle”. Here is how this looks like in JavaScript:

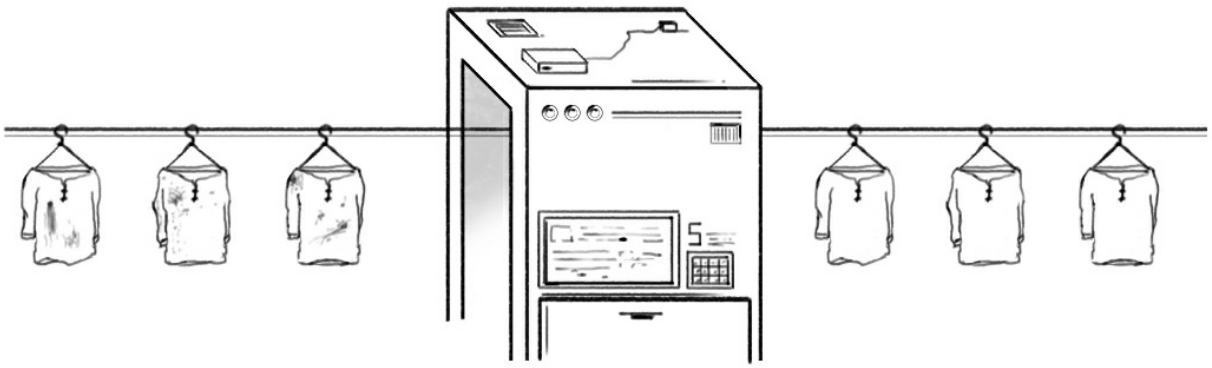
```
function createEscalator() {
  let currentState = 'idle';
  const transitions = {
    idle: {
      move: () => {
        currentState = 'moving';
      }
    },
    moving: {
      stop: () => {
```

```
        currentState = 'idle';
      },
      error: () => {
        currentState = 'broken';
      }
    },
    broken: {}
  }

  return (action) => {
    if (transitions[currentState][action]) {
      transitions[currentState][action]();
    } else {
      console.warn(
        `"${action}" is forbidden while in "${currentState}" state.`
      )
    }
  }
}
```

The API of our machine is a single function that accepts an action. Internally the machine checks if the action makes sense in the context of the current state. If not, it does nothing and prints a warning message.

```
const escalator = createEscalator();
escalator('move'); // success
escalator('move'); // "move" is forbidden while in "moving" state.
escalator('stop'); // success
escalator('move'); // success
escalator('error'); // success
escalator('stop'); // "stop" is forbidden while in "broken" state.
```

Notice that we have no transitions if the machine falls into a “broken” state. We have to restart the machine, so it starts working again.



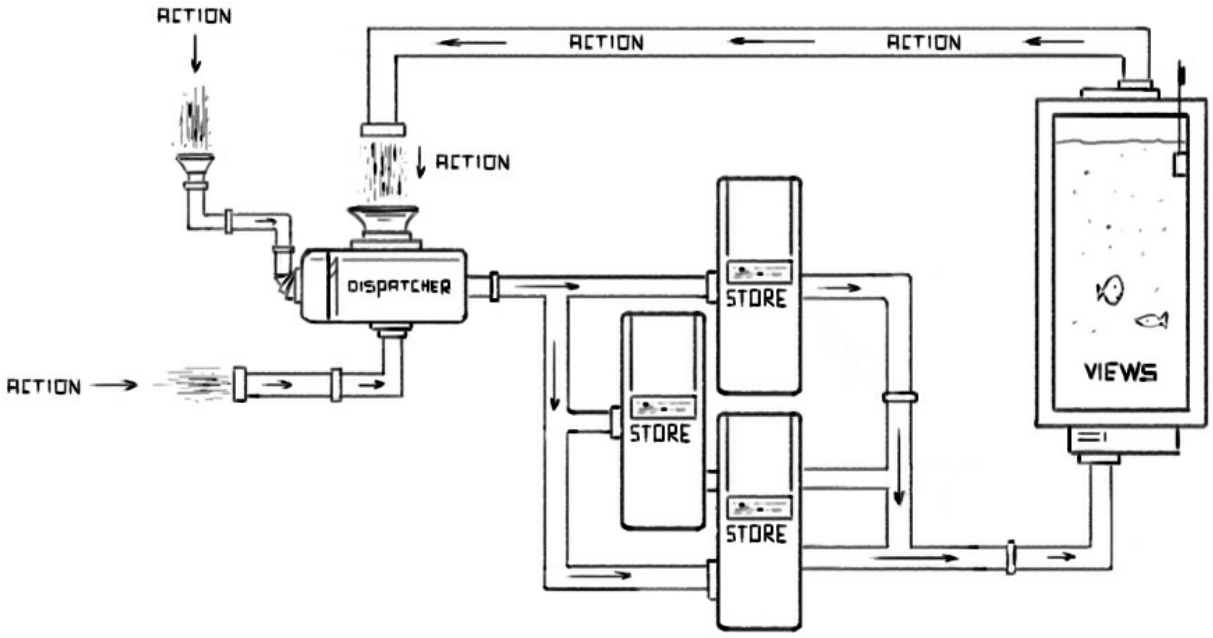
Reducers

Now when we know what a state machine is, we can talk about *reducers*. That term became popular around the Redux library, where it is the main actor. The reducer looks a lot like a state machine. By definition, the reducer is a function that accepts the current state and an action and returns a new state.

```
let state = 0;
const actions = [
  'INC', 'INC', 'FOO', 'DEC', 'INC'
];

function reducer(accumulatedValue, action) {
  if (action === 'INC') { return accumulatedValue + 1; }
  if (action === "DEC") { return accumulatedValue - 1; }
  return accumulatedValue;
}
actions.forEach(action => {
  state = reducer(state, action);
});
console.log(state); // 2
```

We don’t have that restrictive model used in a state machine, but there are some similarities. The reducer decides what will be the next state based on its current value and the incoming action.

The concept of reducers touches on another interesting topic - immutability. In libraries like Redux, it is one of the main ideas. State management where we generate a new state after each change instead of amending the current one. Think about how we have an object with key-value pairs, and instead of updating the values, we create the whole object again. Maybe this sounds like a bad idea, but in fact, it works well.


ACTION
ACTION
ACTION
ACTION
ACTION
DISPATCHER
STORE
STORE
STORE
VIEWS

Flux architecture

Flux architecture is one of those popular things that the ecosystem replaces with something new. Back in the early days of React, this pattern was the answer to many questions. Facebook's framework came with a couple of brilliant ideas, but one thing was missing - how and where we manage state. Facebook answered - Flux architecture.

The idea is to keep our state in multiple stores. There is a dispatcher that receives actions, and it forwards them to each of the stores. Internally the store decides whether the action makes sense. Maybe it changes its state, maybe not. In case of a change, it notifies the views (React components).

In this chapter, we will implement the pattern. Let's start by writing the dispatcher.

```
const D = (function () {
  const stores = [];
  return {
    register: function (store) {
      stores.push(store);
    },
    dispatch: function (action) {
      stores.forEach(function (s) {
        s.update(action);
      });
    }
  }
})();
```

There are dozen of Flux libraries and, in most of them, the dispatcher has such API. One method for *registering* a store and one for dispatching actions. We assume that the stores will have an `update` method. That is the function that will receive the action.

To make the example a bit more interesting we will use two stores. Let's say that the first will collect only numbers and the second one only letters. The view will be just a simple function displaying what's in both stores.

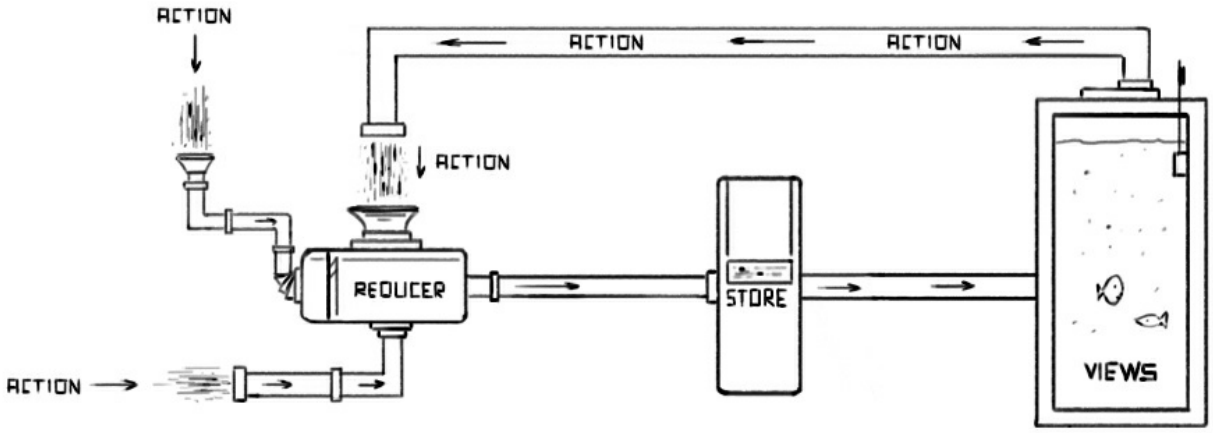
```
const Numbers = {
  data: [],
  update(action) {
    if (typeof action.payload === 'number') {
      this.data.push(action.payload); renderView();
    }
  }
```

```
}
const Letters = {
  data: [],
  update(action) {
    if (typeof action.payload === 'string') {
      this.data.push(action.payload); renderView();
    }
  }
}
function renderView() {
  console.log(Numbers.data, Letters.data);
}
```

At the end we have to dispatch some actions:

```
D.dispatch({ payload: 'A' }); // [], ['A']
D.dispatch({ payload: 'B' }); // [], ['A', 'B']
D.dispatch({ payload: 5 }); // [5], ['A', 'B']
D.dispatch({ payload: 'C' }); // [5], ['A', 'B', 'C']
D.dispatch({ payload: 6 }); // [5, 6], ['A', 'B', 'C']
```

The most important characteristic is that we have a one-direction data flow. The actions are getting fed to the stores. If the state changes, the views are re-rendered.


ACTION
ACTION
ACTION
ACTION
REDUCER
STORE
VIEWS
ACTION

Redux

In this book, the Flux architecture chapter is before Redux on purpose. Historically Redux came after Flux and, it has almost the same structure. The one-way direction data flow is still here. We again dispatch actions and, our views subscribe to store changes. In Redux though, we have just one store. To keep it organized, we have to create the so-called “slices”. Each slice has its function that accepts actions and the current state. Based on that takes a decision what will be the next state. Those functions are called reducers.

Similar to Flux, Redux can be implemented in a lot of different ways. The pattern became a standard. Because of that, the community develops all sorts of tooling and helpers for it. The code that you are going to see here lacks action creators and selectors. Those are, as the name suggests, for creating actions and getting data from the store. For the sake of simplicity, we are not going to implement them. We will use the example from the previous chapter where we solved the collecting of numbers in one place and letters in another.

The following function gives us the backbone of the Redux pattern - dispatching of actions, updating the state in an immutable fashion, and subscribing for changes.

```
function configure(slices) {
  const state = {};
  const views = [];
  return {
    dispatch(action) {
      Object.keys(slices).forEach(key => {
        state[key] = slices[key](state[key], action);
      });
      views.forEach(view => view(state));
    },
    connect(view) {
      views.push(view);
    }
  }
}
```

Our store will have a slice that keeps the letters and another that keeps the numbers.

```
const { dispatch, connect } = configure({
  letters(state = [], action) {
    if (typeof action.payload === 'string') {
      return [...state, action.payload];
    }
```

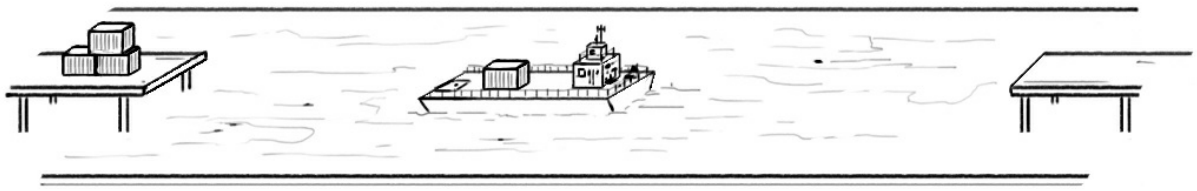
```
    return state;
  },
  numbers(state = [], action) {
    if (typeof action.payload === 'number') {
      return [...state, action.payload];
    }
    return state;
  }
});
```

What we pass to the `configure` function are our reducers. They receive the current state and the action. As a result, they should return a new version of the state (or the same array if nothing is changed).

Here is a demo of how everything works:

```
connect(state => console.log(
  `${state.letters} ${state.numbers}`
));
dispatch({ payload: 'A' }); // A
dispatch({ payload: 'B' }); // A,B
dispatch({ payload: 5 }); // A,B 5
dispatch({ payload: 'C' }); // A,B,C 5
dispatch({ payload: 10 }); // A,B,C 5,10
```

Our “view” here is just a function that prints the values from the state. In the actual application, that is probably a component that renders information on the DOM.



Communicating sequential processes

CSP (communicating sequential processes) is a formal language for describing patterns of interaction in concurrent systems. It's used in Go, Crystal, Clojure's core.async, and a couple of other places.

The idea is nothing so complicated, but it offers some interesting capabilities. Think about a channel that we can use to transfer messages. We attach publishers and subscribers. Nothing unusual. Every event-based system works like that. However, in CSP those two groups are synchronized. Meaning that publishing is not possible until we have a subscriber that awaits the message.

Here is a simple implementation of such a channel:

```
function createChannel() {
  const puts = [], takes = [];
  return {
    put: (data) => new Promise(resolvePut =>
      takes.length > 0 ?
        (takes.shift()(data), resolvePut()) :
        (puts.push(() => resolvePut()))
    ),
    take: () => new Promise(resolveTake =>
      puts.length > 0 ?
        resolveTake(puts.shift()()) :
        takes.push(resolveTake)
    )
  }
}
```

The `puts` array represents the messages that we want to send. `takes` contains the subscribers that wait for those messages. Notice how in each of the two methods (`put` and `take`) we first check whether there are consumers on the other side. Again, we can't put something on the channel if there is no one to take it. And we can't take it if there is nothing in the channel.

Here is one possible use case:

```
async function A() {
  console.log('Waiting for values');
  console.log(`Receiving ${await channel.take()}`);
  console.log(`Receiving ${await channel.take()}`);
}
async function B() {
  console.log('Sending "foo"');
  await channel.put('foo');
  console.log('Sending "bar"');
  await channel.put('bar');
```

```
  console.log('Messaging over.');
}

A(); B();
// Waiting for values
// Sending "foo"
// Receiving foo
// Sending "bar"
// Receiving bar
// Messaging over.
```

We execute the two functions at the same time, but on every line, they wait for each other.

Design patterns


I'm the One!

Singleton

In this section of the book, we will learn about six of my favorite design patterns. The first one is called “singleton”. Whenever we force the creation of one and only one instance of a class, we use the singleton pattern. Think about a class that we can initialize only once.

In JavaScript the implementation of this pattern is very often done via static function of the class.

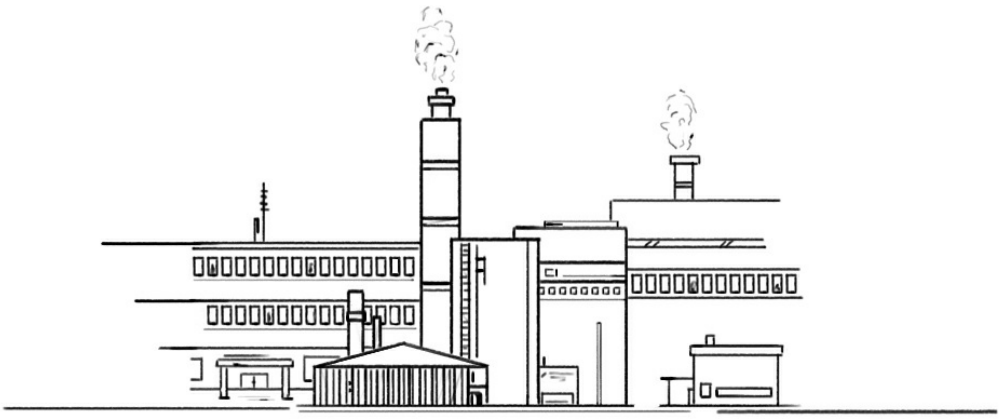
```
let cache;
class A {
  static getInstance() {
    if (cache) return cache;
    return cache = new A();
  }
}

const x = A.getInstance();
const y = A.getInstance();

console.log(x === y); // true
```

The `cache` variable keeps the created instance of the class. If we attempt to create a new one, we will get the same result. That is why `x` and `y` point to the same object.

Another popular way to create a singleton is to use a factory function:

```
let cache;
class A {
  // ...
}

function getInstance() {
  if (cache) return cache;
  return cache = new A();
}

const x = getInstance();
const y = getInstance();

console.log(x === y); // true
```

I prefer this approach because it eliminates the option of someone using `new <MyClass>` directly.

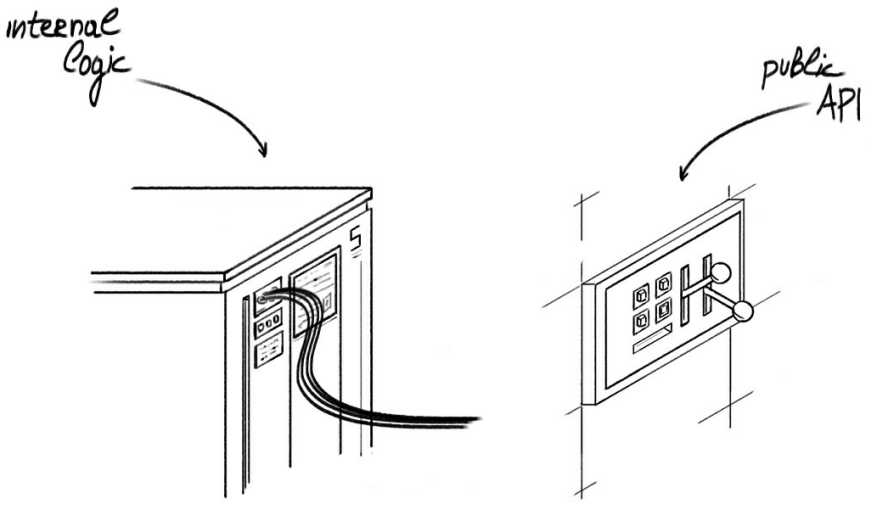


Factory pattern

The factory pattern, same as a singleton, is into the category of creational patterns. Those are patterns that are focused on the initialization of the objects. In the world of JavaScript this could be a function that receives dependencies and returns the desired instance.

```
function vehicleFactory(engine) {
  return (type) => {
    switch(type) {
      case 'car': return new Car(engine);
      case 'track': return new Track(engine);
    }
    throw new Error(`Unrecognized type ${type}`);
  }
}
function Car(engine) { this.what = 'car'; }
function Track(engine) { this.what = 'track'; }
const engine = { vroom: true };
const factory = vehicleFactory(engine);
const A = factory('car');
const B = factory('track');
console.log(A.what, B.what); // car track
```

We use the factory pattern when we are not sure what kind of objects we need. Often this information comes at runtime. That is why the factories usually hold a `switch` type of logic.


internal
logic
public
API

Module revealing pattern

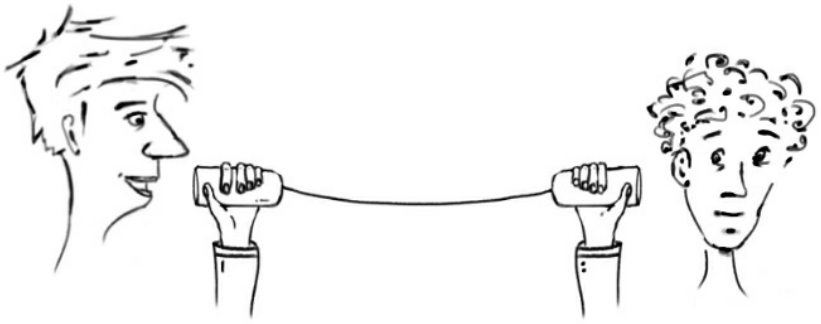
The revealing module pattern is the one that I use the most in my daily job. Back in the days when we didn't have classes, this pattern was my main tool for mimicking OOP.

The idea here is to create a scope and return an API. The scope encapsulates the logic and, the object that we return gives access to our *public* interface.

Imagine we have to write a music player:

```
function Player() {
  let trackId;
  function play(id) {
    trackId = id;
    console.log(`Track #${trackId} is started`);
  }
  function stop() {
    console.log(`Track #${trackId} stopped`);
  }
  function status() {
    console.log(`Processing track #${trackId}`);
  }
  return {
    play,
    stop,
    status
  };
}
const p = Player();

p.play(4); // Track #4 is started
p.status(); // Processing track #4
p.status(); // Processing track #4
p.stop(); // Track #4 stopped
```

We have scoped the variable `trackId` that is hidden for the outside world. The functions `play`, `stop`, and `status` are our public API. It is similar to writing a class. The `Player` function acts as a constructor because whenever we call it we will get a new *instance*. With its own `trackId` and public methods.



Publisher/Subscriber pattern

Often in programming, we use design patterns without even knowing that they exist. The publish/subscribe pattern is one of them. I'm pretty sure that every web developer has to work with the DOM and has to listen for events like `click` or `load`. Whenever we use `addEventListener` we are using the publish/subscribe pattern.

The idea here is that we have objects (subscribers) that need some information and other objects that are ready to provide it (publishers). The pattern implements a mechanism/channel between them so they can communicate via messages. The message is often called *event*.

```
const Bus = {
  _subscribers_: {},
  subscribe(type, callback) {
    if (!this._subscribers_[type]) this._subscribers_[type] = [];
    this._subscribers_[type].push(callback);
    return () => {
      this._subscribers_[type] = this._subscribers_[type].filter((c) => c !== callback);
    };
  },
  dispatch(type, payload) {
    if (this._subscribers_[type]) {
      this._subscribers_[type].forEach((c) => c(payload));
    }
  },
};
```

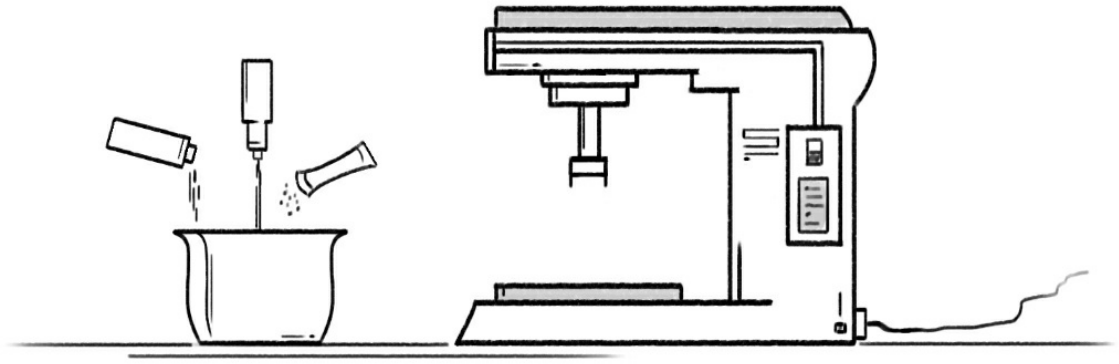
If you read the chapter for the Flux architecture, you will recognize a similar API. We need a local variable to keep the subscribers and a method for dispatching events.

```
const unsubscribe = Bus.subscribe("hey", (data) => {
  console.log(`Hello, ${data.name}`);
});
Bus.dispatch("hey", { name: "Alan" }); // Hello, Alan
Bus.dispatch("hey", { name: "Margie" }); // Hello, Margie
unsubscribe();
Bus.dispatch("hey", { name: "George" }); // nothing
```

A typical characteristic of this pattern is organizing the events into topics. In our example, we pass the topic via the `type` argument. This way we have a bucket of publishers and subscribers without having conflicts.

There is also usually an unsubscribing functionality. In the code above, the `subscribe` method returns a function that effectively removes the listener.

The biggest advantage of this concept is also its weakness. The different parties are isolated from each other. Often the subscribe doesn’t know who the sender is. This may look like a good side effect, but when the application grows, it becomes a problem. It is difficult to track data flows and debugging issues.



Mixins

A mixin is a set of helpers that we can inherit. In the world of JavaScript, this is often an object that we merge with another one. The power of the mixins comes from the fact that we may create many of them. By default in JavaScript, we can inherit from one class. This creates limitations. The mixins don't have such problems because we can apply multiple to a single object.

Let's look at the following car engine case.

```
function Car() {
  this.speed = 50;
}
function engineMixin(obj) {
  return Object.assign(obj, {
    forward() { console.log(`forward ${this.speed} km/h`); },
    backward() { console.log(`backward ${this.speed} km/h`); },
  });
}
function extrasMixin(obj) {
  return Object.assign(obj, {
    horn() { console.log("Beeeeeeeep!"); },
  });
}
```

If we create a car (via `new Car()` statement) we will get just an object with a `speed` property. However, if we pass the car to `engineMixin` and `extrasMixin` mixins we will enhance the car's API with three new methods.

```
const car = extrasMixin(engineMixin(new Car()));
car.forward(); // forward 50 km/h
car.horn(); // Beeeeeeeep!
car.backward(); // backward 50 km/h
```

We have to use this pattern wisely. The common problems of the mixins are with the dependencies management, level of complexity, and name clashing. If a team decides to adopt the mixin pattern, the developers need to create conventions, so they avoid such issues.

Command pattern done with generators

I spent some time thinking about how to finish the book. What will be the last chapter? And to be honest, most of those 50 topics deserve to close the book. I picked the command pattern implemented with generators. I saw it for the first time when I used the redux-saga library. It is based entirely on generators. I became a fan of the approach.

The idea of the command pattern is to separate the objects that trigger logic from the objects that implement it. The trivial variant involves a function and a switch statement that decides what exactly needs to be done. The following snippet shows the triggering bit:

```
function* logic() {
  const name = yield ["get:name"];
  const user = yield ["get:user", name];
  console.log(`Hey ${user.firstName} ${user.lastName}`);
}
```

`"get:name"` and `"get:user"` are the so-called commands. Those are the instruction. The idea is to get a name as a string and then receive a user object containing that same string split into first and last name. See how here we don't do the name fetching and splitting. We also don't create the `user` object. We only declare what we want to happen without doing it.

Here is the generator that receives the commands and executes them:

```
function processor(gen, result) {
  let status = gen.next(result);
  if (!status.done) {
    const [command, param] = status.value;
    switch (command) {
      case "get:name":
        return processor(gen, "Tracy King");
      case "get:user":
        const [firstName, lastName] = param.split(" ");
        return processor(gen, { firstName, lastName });
    }
  }
}

processor(logic()); // Hey Tracy King
```

I don't see this often, but the generators are a way to go over a set of instructions (potentially asynchronous). In our case, the `processor` function iterates the generator by reading the commands and passing the result back.

You’ve made it! Thank you for reading all these 50 short stories. I hope my book made you a better JavaScript developer.