JS Functions 2

Course: Java

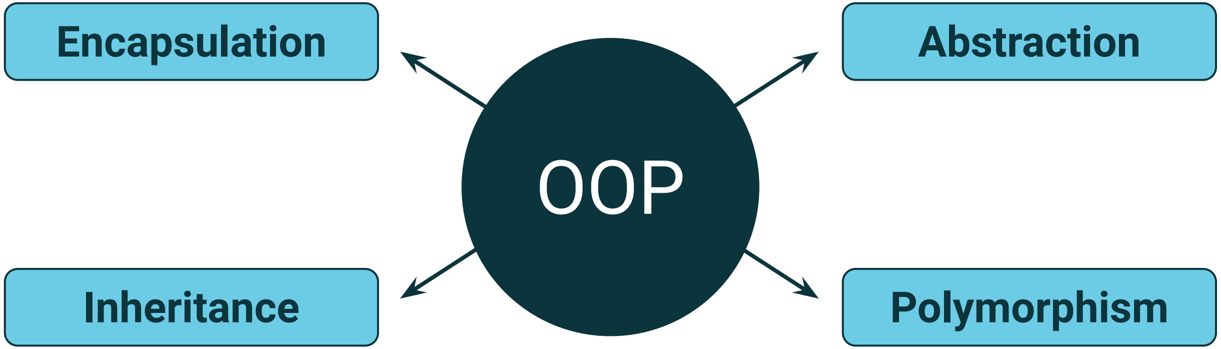
S1

WELCOME

Web Applications are Data Driven

This usually means that they work with data structures, such as arrays of objects, often with nesting.

We need better tools than `for` and `while` to manage this! So, let's explore the basics of functional programming. This will be an invaluable tool when used with (or without) OOP.



Learning Outcomes

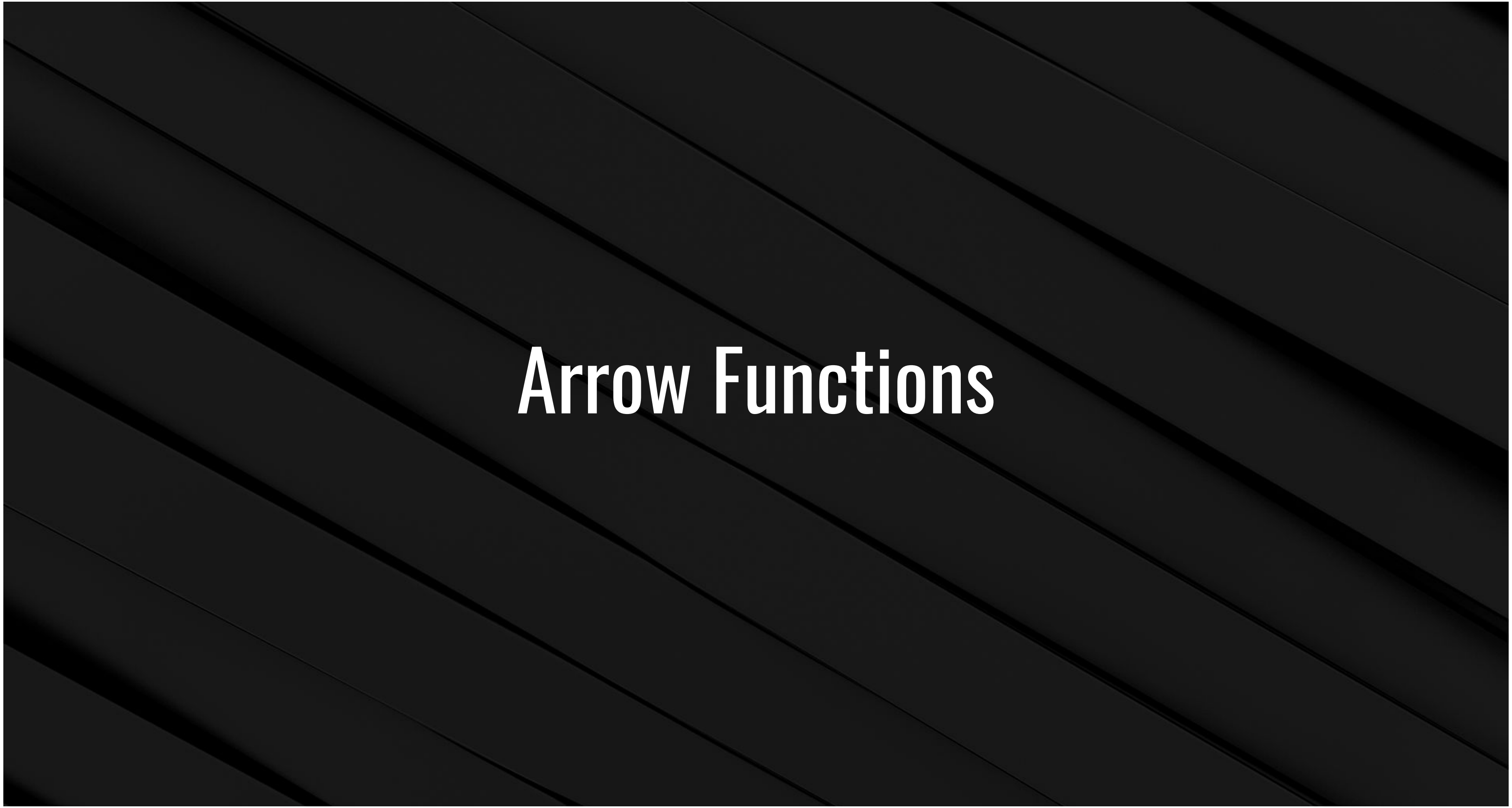
By the end of this lesson, you will be able to:

01 Use arrow functions and implicit `return`.

02 Use function factories to create objects and use ES6's object shorthand.

03 Explain what `prototype` means.

04 Use callback patterns to `map`, `filter`, `reduce`, and `sort` data structures.



Arrow Functions

Arrow Syntax for Function

Arrow syntax is part of the ES6 spec.
It uses `const` and a `=>` rather than requiring us to type out `function`.

Arrow functions do not have their own `this` bindings.
If you require `this`, you should use `function`.

There is no obligation to use arrow syntax, but it can offer great convenience.
And, less typing usually means fewer mistakes.

```
const adder = (num1, num2) => {
   return num1 + num2
 }
```

Arrow Syntax for Function

If our **function body** (inside of the `{}`) consists of only one statement, we can remove `{}` and return: `const adder = (num1, num2) => num1 + num2`.

```
const adder = (num1, num2) => {
  return num1 + num2
 }
```

This uses our single statement as an implicit `return`.
Recall that with `function`, implicit `return`s are always `undefined`.

Additionally, if our function only receives one parameter,
we can omit the `()` around it.

Arrow Syntax for Function

Me **must** use `()` if we use destructured parameters:

```
// We are destructuring an object because we have more than
2 parameters (not required, but best practice)
const greet = ({name, age, occupation}) => `
  My name is ${name}. I am ${age} years old. I am a(n)
${occupation}.
```

Activity: Refactor Some Previous Functions to Use =>

Use the code in this folder. Update `index.html` to use the correct JS file: `<script src="./refactor-arrows/age-is-minor.js"></script>`.

Use single-statement implicit `return`s whenever possible.

Suggested Time:

20 Minutes

Time to Code

Function Factory with Object Shorthand

Suggested Time:

15 Minutes

Function Factory with Object Shorthand

A **function factory** is any function that returns an object.

```
const createPerson = (name, age) => {
  return {name: name, age: age}
}
```

Function Factory with Object Shorthand

ES6 also provides **object shorthand**: `{name: name, age: age}` can be simplified to `{name, age}`.

```
// `{name}` creates a property `"name"` and for the value assign the variable `name`
const createPerson = (name, age) => ({name, age})

console.log(createPerson("Mark", 23))
```

Break

OOP and Prototypes

Time to Code

Function Factory

Suggested Time:

10 Minutes

Code: Function Factory

```
const bob = {
    id: 1134299,
    username: "bob1998",
    firstName: "Robert",
    lastName: "Kazinsky",
    title: "Associate Developer",
    intro() {
        return `My name is ${this.firstName} ${this.lastName}. I am a(n)
${this.title}.`
    }
    updateTitle(newTitle) {
        this.title = newTitle
        }
    }
```

Time to Code

Constructor Function

Suggested Time:

15 Minutes

Constructor Function

This approach simplifies the creation of objects, but the methods are still duplicated for each object.

Part of the motivation behind OOP is to allow for some abstraction of duplicated code.

The properties need to be **constructed** for each instance since they are unique, but the methods should just be inherited.

We do this via **prototypes** in JS.

JS uses **function constructors** that use the `new` keyword and `this` bindings to attach properties and/or methods to newly instantiated objects. It's still OOP—just a different underlying implementation mechanism.

JS does not apply `class`es like Java's OOP does. ES6 does introduce a `class` keyword, but this is just **syntactic sugar** for the underlying function constructors.

Syntactic Sugar

01

The term “syntactic sugar” refers to a simpler, hence sweeter, syntax for doing the same thing—**without any change to the underlying implementation**.

02

By this definition, arrow syntax is **not** syntactic sugar, because the underlying functionality is affected due to losing `this` bindings.

Syntactic Sugar

01 By convention, function constructors are capitalized. This serves as a reminder that we need to use `new`.

02 Since we need `this` bindings, we will not see arrow functions for function constructors in most cases.

03 The name is usually a noun that represents the type of object being created.

Time to Code

class

Suggested Time:

10 Minutes

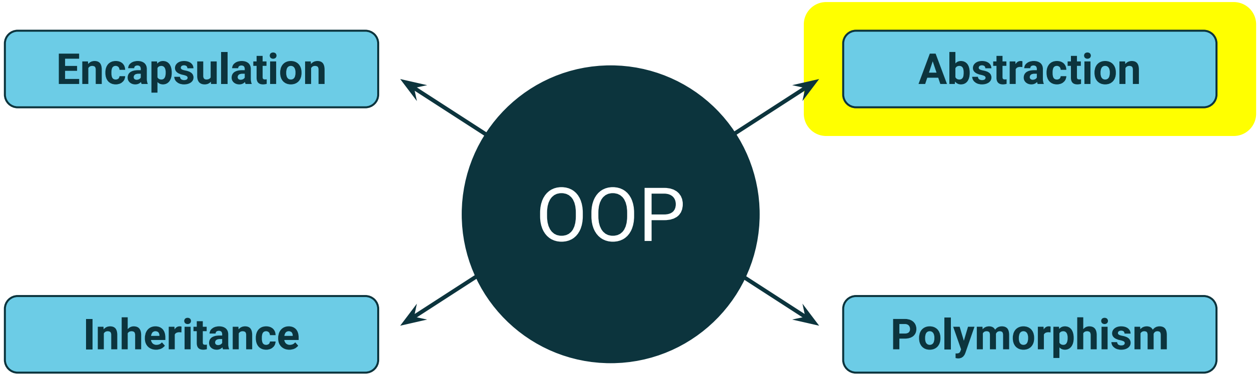
JS OOP Highlights and prototype

The purpose of OOP is to create objects!

JS OOP Highlights and prototype

Using `{}` makes creating objects in JS simple.

We can abstract objects into function constructors (we can also implement them using ES6 `class`, but we're still implementing function constructors).

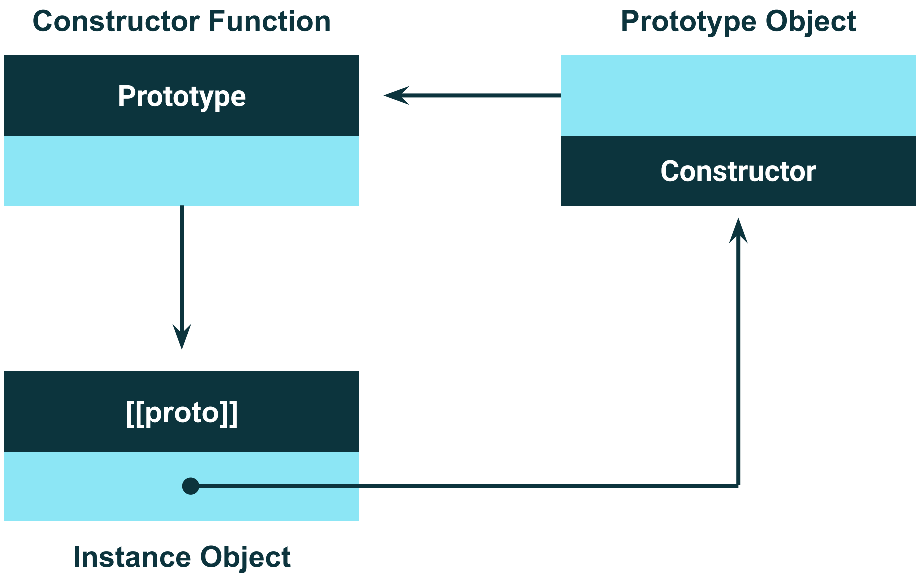


JS OOP Highlights and prototype

Everything in JS is an object—except primitives.

This includes, for instance, arrays.
In fact, `const nums = [1, 2, 4];` is syntactic sugar for `const nums = new Array(1, 2, 4);`.

Properties and methods are attached to a constructor function's `prototype` (or can be implemented on an individual, per-object basis).



JS OOP Highlights and prototype

The array methods that we explored previously, such as `push` and `pop`, are all attached to `Array.prototype`.

```
class Employee {
  constructor({
    id,
    username,
    firstName,
    lastName,
    title,
  } = {}) {
    this.username = username;
    this.firstName = firstName;
    this.lastName = lastName;
    this.title = title;
  }
```

Break

Callback Patterns with Mutations

Callback Patterns with Mutations

Earlier, we learned about callback functions in the context of asynchronous DOM events.

- This pattern is also useful for iterating over arrays—we no longer need to use `for`/`while`.
- With this pattern, for **each item** in an array, we call back a function.
- This iteration itself happens synchronously but still applies the callback function.

Time to Code

forEach

Suggested Time:

10 Minutes

Time to Code

`forEach <li>`

Suggested Time:

20 Minutes

Code: forEach <li>

“The `Element.classList` is a read-only property that returns a...collection of the `class` attributes of the element. This can then be used to manipulate the class list.” — MDN

```
/**
 * 1. Grab all of the `<li>s` in `document`.
 * 2. For each one of these, access its `classList` property and `add`
`.text-info`.
 */
document.querySelectorAll('li').forEach(li => {
  // `li` is a of type `Element` - use `Element.classList`
  li.classList.add("text-info")
 })
```

Activity: Using forEach

Write your JS Code here. Assuming that you have done `npm start`, navigate to this page on your `localhost` web server.

You will notice some `<section>`s, each with three `<p>`s, with some filler text.

Inside of the JS file, you'll see:

- TODO: Apply `.lead` ONLY to the first paragraph in each section.
- TODO: `Use addEventListener` to add the class `.bg-info` to all paragraphs on `mouseover` and `mouseout`

Hint: `section p:first-of-type.`

Suggested Time:

30 Minutes

Break

Array Methods

Each of these array methods will still apply callback functions.

The difference is that no mutation will be necessary.

map

Do not confuse this with the data structure `Map`.

Those are less frequently used.

map

We will only work with:

```
Array.prototype.map
```

👍

NOT:

```
TypedArray.prototype.map
```

👎

map

The data structure `Map` is capitalized, indicating the instantiation of a new type of object.

`Array.prototype.map` is a specific method attached to the constructor function `Array`.

```
const nums = [1, 2, 3, 4, 5, 6, 7];

const doubled = nums.map(num => num * 2)
```

map

The syntax of `map` is similar to `forEach`, but it **returns a whole new array** with our results.

No mutation is necessary.

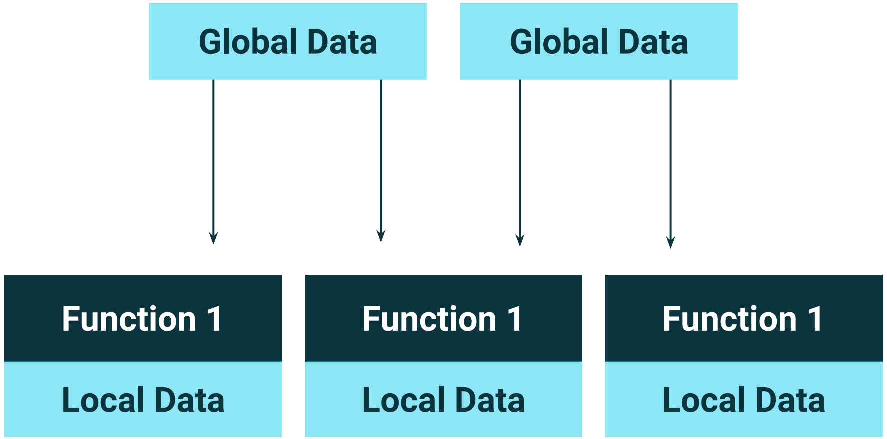
Often, we can write very clean-looking arrow functions that use only one line of code.

```
const nums = [1, 2, 3, 4, 5, 6, 7];

const doubled = nums.map(num => num * 2)
```

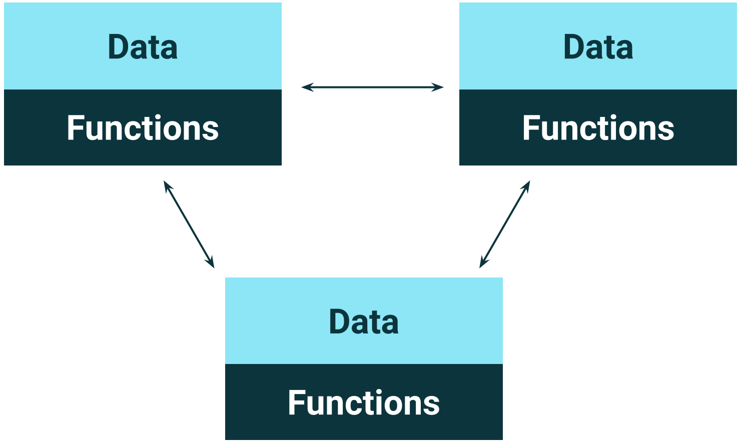
map

In addition to OOP (and imperative programming styles, such as with `for`), JS also allows for a **functional programming** style.

Functional Programming

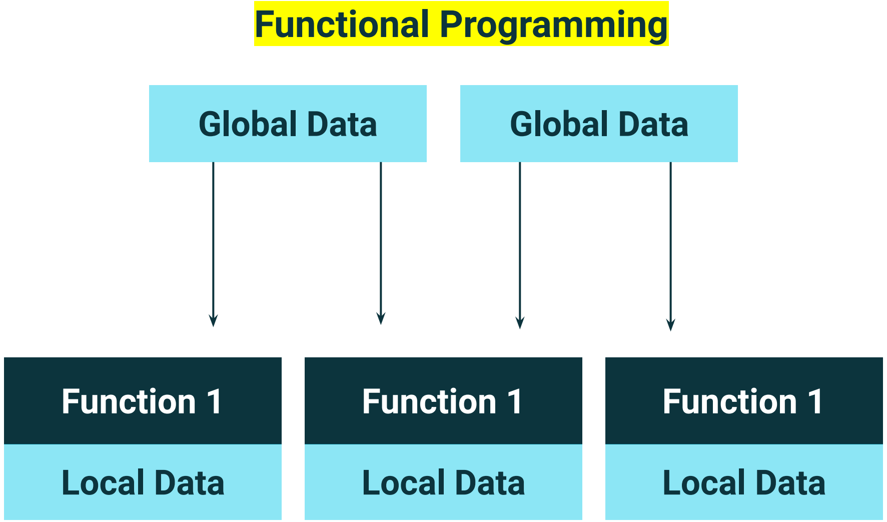


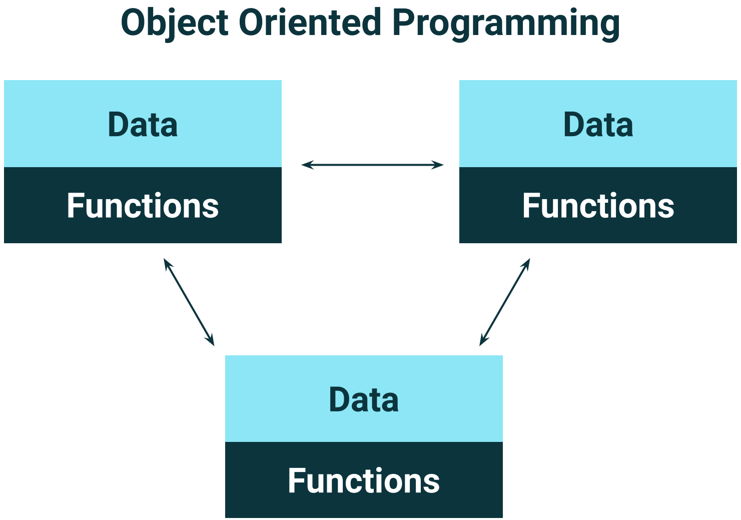
Object Oriented Programming



map

The `map` method introduces us to a more declarative syntax, where we embrace functional programming.





map

A mapped array **always** contains the same number of items as it started with. Each item is just transformed by the callback function.

01

The `map` method’s callback function is pure.

Given the same input, it will always return the same output.

02

It doesn't reach outside of its scope. It could be copied/pasted anywhere and would still work the same way—receiving a `num` and returning `num * 2`.

filter

Another common task is creating a new array (again, with no mutations) based on a predicate callback function.

A **predicate callback function** is a callback function that always returns either `true` or `false` based on some test condition.

filter

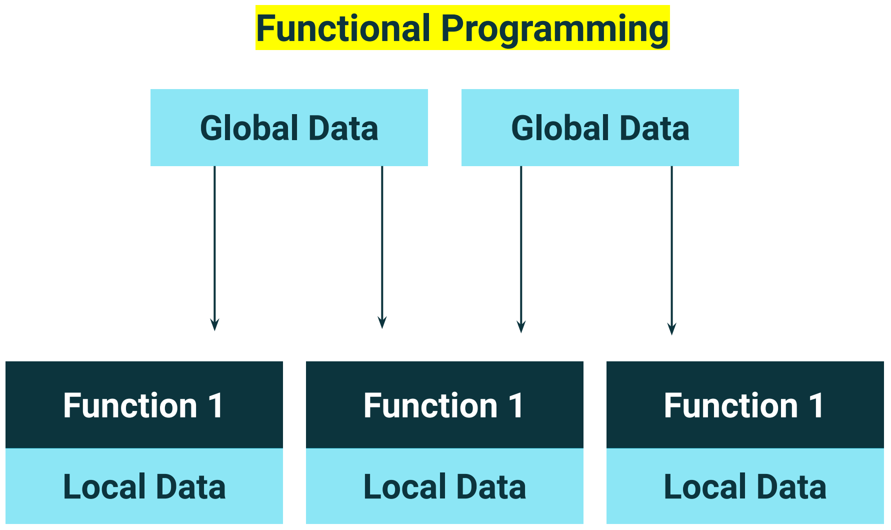
In this way, a `filter`ed array **usually has fewer items** than the original array.

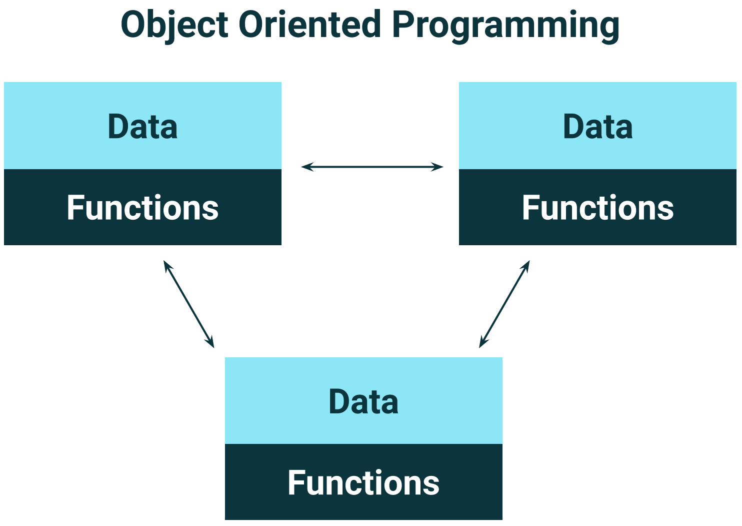
Or, it could have the same number of items if all of the items pass the predicate test in the callback function.

```
const nums = [1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12];


// Any `num` that is truthy for `num % 2` will be returned to the new array
 const oddNums = nums.filter(num => num % 2)
```

filter

Like `map`, `filter` is considered a functional approach—
it uses a pure callback function and doesn't mutate anything.





filter

By definition, `filter` never mutates anything. It returns an **entire item** based on the predicate callback function.

For this reason, it is even simpler to implement than `map`.

Destructuring is also frequently used with arrays objects.

```
const nums = [1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12];


// Any `num` that is truthy for `num % 2` will be returned to the new array
 const oddNums = nums.filter(num => num % 2)
```

reduce

Using `reduce` reduces an array's items down to a single thing.

reduce

The `reduce` method works similarly to `map` and `filter`, but its callback function takes two parameters:

01

The first represents an **accumulator** (the thing that we are reducing the array down to).

02

The second represents the **current item**.

reduce

As `reduce` iterates over the array, the accumulator gets updated in relation to the current item and gets returned until the array iteration has completed.

```
const nums = [1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12];


nums.reduce((total, num) => total += num, 0)
```

reduce

The `reduce` method also takes an optional third parameter that initializes the accumulator on the first iteration.

If we don't specify this, `reduce` will initialize the accumulator with the first item in the array. To be safe, it's usually best to explicitly specify the initial value of the accumulator using `reduce`'s third parameter.

```
const nums = [1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12];


nums.reduce((total, num) => total += num, 0)
```

Break

NodeJS

NodeJS Overview

- Up until now, we have run JS from the client side—in the browser.
- In 2011, Ryan Dahl created NodeJS. It uses Chrome's V8 engine but executes JS outside of the browser. This is how we write JS on the server side.
- Most code bases modularize code into separate files.
- Using this approach allows for a cleaner separation of concerns.
- There are two module systems in JS:
  - **CommonJS**: This module system pre-dates ES6. Node still uses it, primarily for backward-compatibility reasons. In this system, we see `require` and `module.exports` used.
  - **EcmaScript Modules**: This modern approach uses `import` and `export`. We will use this when we go back into the browser and, later, in ReactJS.

NodeJS Overview

- Much of JS is the same whether running from the browser or from Node.
- Node knows nothing about the Web API.
    - Node doesn't know DOM. It's not a “web environment.”
- Node leverages our OS's capabilities instead of a browser’s.
    - Our OS doesn't know about the DOM; therefore, Node doesn't either.
- Node does have additional capabilities, such as receiving requests and sending back responses.
    - Recall that a browser does the opposite—it sends requests and receives responses.

Time to Code

Node Starter Code: tests

Suggested Time:

20 Minutes

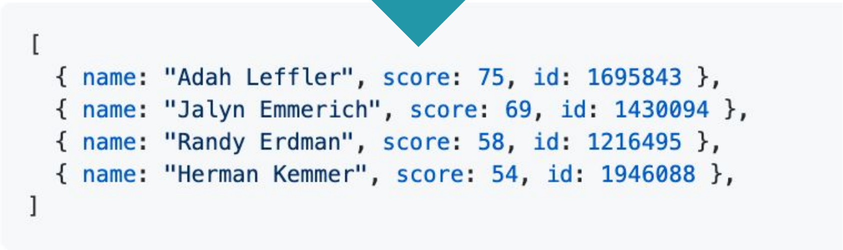
Break

JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON)

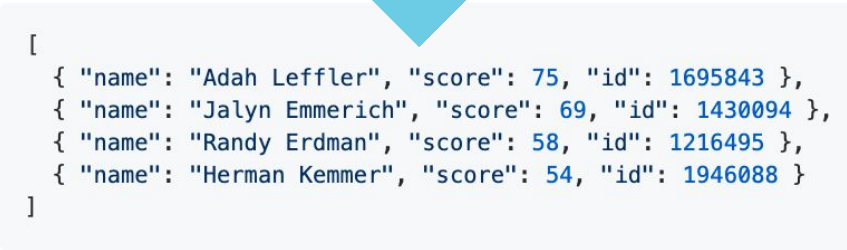
- JSON is specially formatted text. **It is not JavaScript!**
- It is notation (text) that is formatted in a style that resembles JS Objects.
  - However, almost any programming language can consume this formatted text, or notation.

Here is a snippet of the random student data that we worked with previously. We have removed the `const` part in order to show just the JS array of objects:



```
[
  { name: "Adah Leffler", score: 75, id: 1695843 },
  { name: "Jalyn Emmerich", score: 69, id: 1430094 },
  { name: "Randy Erdman", score: 58, id: 1216495 },
  { name: "Herman Kemmer", score: 54, id: 1946088 },
]
```

And, here is the same data formatted as JSON:



```
[
  { "name": "Adah Leffler", "score": 75, "id": 1695843 },
  { "name": "Jalyn Emmerich", "score": 69, "id": 1430094 },
  { "name": "Randy Erdman", "score": 58, "id": 1216495 },
  { "name": "Herman Kemmer", "score": 54, "id": 1946088 }
]
```

JavaScript Object Notation (JSON)

Notable differences between JS code and JSON:

01 In JSON, all keys must be in quotation marks. In JS, this is optional.

02 In JS, the last object in an array can have a comma after it or not.
In JSON, no dangling commas are allowed.

03 JSON doesn't typically allow comments.

JSON.stringify and JSON.parse

- It's simple to convert between JSON and JS.
- We `stringify` JS into JSON.
  - This means that we add quotes around the keys:

    ```
    JSON.stringify({ name: "Adah Leffler", score: 75, id: 1695843 })
    ```

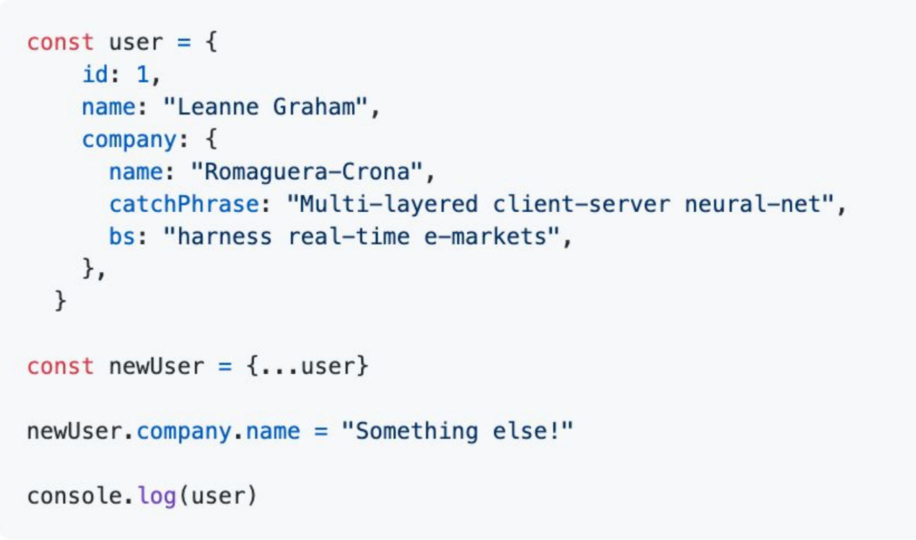
- And vice versa, we parse JSON into JS objects:

    ```
    JSON.parse({"name":"Adah Leffler","score":75,"id":1695843})
    ```

JS objects are passed by reference.
Because of this, especially for nested
objects, using `...` is not enough
to prevent unwanted mutations.

Avoiding Mutations with JSON.stringify and JSON.parse

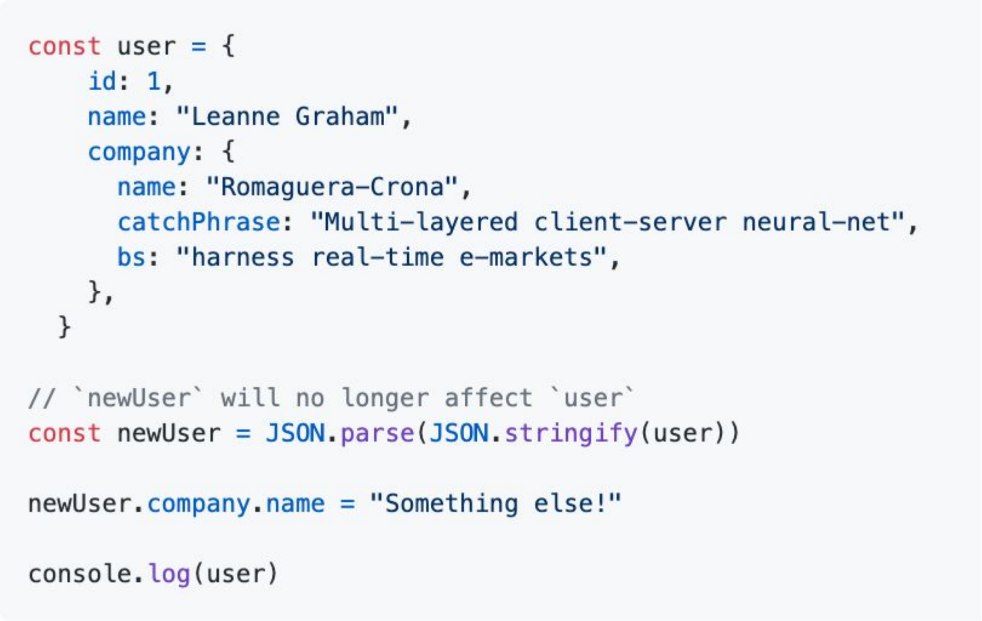
- In the example (right), we inadvertently updated `company.name` for both `user` and `newUser`, even though we applied `...` to create a new object reference.
- The reason is that `company` is a nested object, and `...` is not recursive.
- We can convert a JS object (even with nested objects) into a string with `JSON.stringify`. Turning an object into a string primitive ensures that it is no longer affiliated with any other object reference.



```
const user = {
    id: 1,
    name: "Leanne Graham",
    company: {
      name: "Romaguera-Crona",
      catchPhrase: "Multi-layered client-server neural-net",
      bs: "harness real-time e-markets",
    },
  }

const newUser = {...user}

newUser.company.name = "Something else!"

console.log(user)
```

Avoiding Mutations with JSON.stringify and JSON.parse

- Then, we can immediately `parse` the string back into a JS object.



```
const user = {
    id: 1,
    name: "Leanne Graham",
    company: {
      name: "Romaguera-Crona",
      catchPhrase: "Multi-layered client-server neural-net",
      bs: "harness real-time e-markets",
    },
  }

// `newUser` will no longer affect `user`
const newUser = JSON.parse(JSON.stringify(user))

newUser.company.name = "Something else!"

console.log(user)
```

Break

Node Activities

Time to Code

map Student Name Lengths

Suggested Time:

20 Minutes

Time to Code

Curve Each Student's Score by 10 Points

Suggested Time:

15 Minutes

Time to Code

Avoid Mutating Nested Objects in map

Suggested Time:

25 Minutes

Break

Activity: filter Student Names with length >= 10

Review the data. Notice that it's an array of objects—an extremely common data structure.

Open up our starter code. Only write code where indicated.

Suggested Time:

20 Minutes

Time to Code

filter Users Whose website.endsWith('.net')

Suggested Time:

20 Minutes

Activity: Use reduce to Calculate the Students' Average Score

The starter code is here.

Use `reduce` to update the code so that it will return the average score for all of the given students.

Hint: Divide by `students.length`.

Suggested Time:

20 minutes

Break

Time to Code

Use reduce to Calculate TLD Results

Suggested Time:

30 Minutes

RECAP

Questions?