Mobile CSS

Prepared by
Mark Gurevich

Canva

Devices

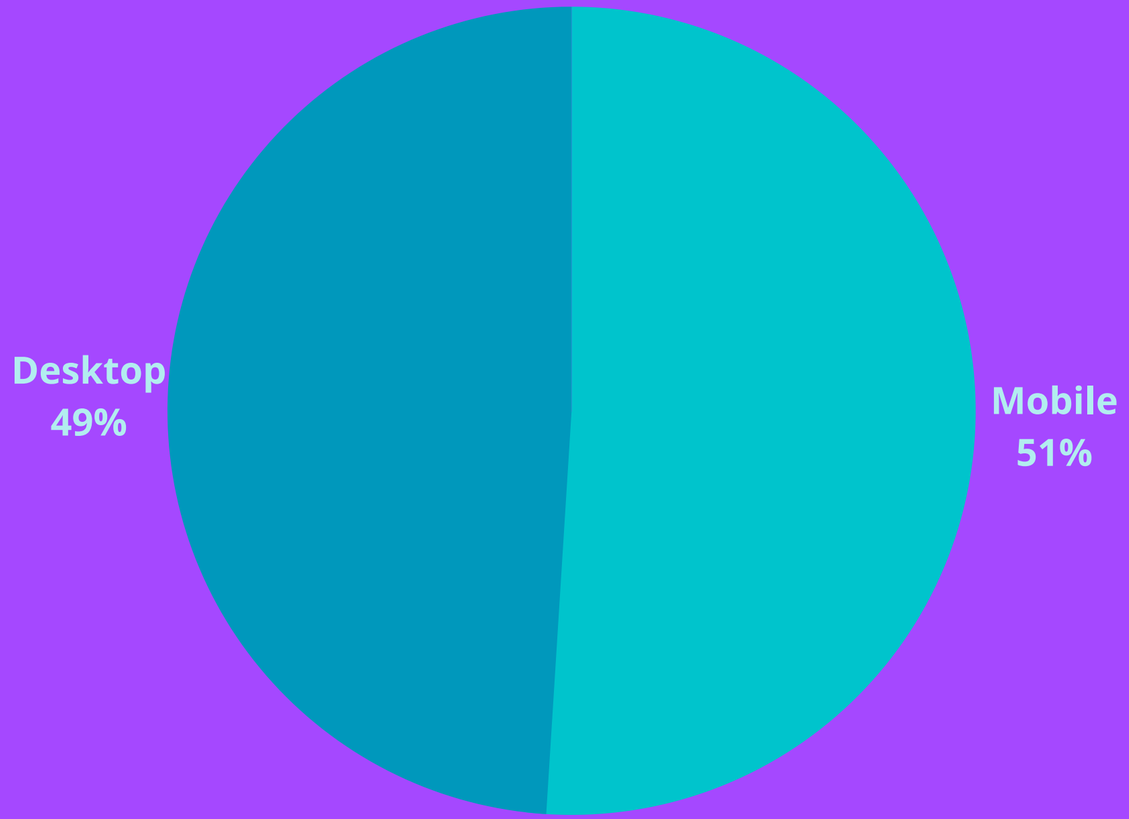
Users could open a website on a lot of different devices: desktop, mobile, tablet, tv...

And developers responsibility is to build a website in the way that it would works properly wherever it has been opened

Mobile

Mobile web worldwide traffic grows every year

And it's already ahead of desktop



Mobile version

One of the ways to provide a good experience to mobile users is to build a separate version of the website which will be optimised to mobile devices

When a user opens your website from mobile device they will be redirected to mobile version (usually leaves on sub-domain which starts with "m." - m.website.com)

Mobile version

Pros:

- Well optimised for mobile
- Possible to build a different UX flow for mobile and desktop
- Easier to debug

Cons:

- Have to build 2 websites
  - Time
  - Make sure that changes were applied on both versions
- False detections
- Handle SEO problems

Responsive Website

The another way to build a website which will works good on both mobile and desktop is to make it responsive by CSS

It will change how it looks like based on the size of the screen

Responsive Website

Pros:

- Time
- Always have the same functionality whatever device is used

Cons:

- Extra code that used only for the one version of the website
- Harder to provide the best UX in all the cases

Media queries

Allows to create CSS rules which are applied to the document only when device reach specific criteria

```
.article {
  padding: 5px 10px;
}

@media (min-width: 600px) {
  .article {
    padding: 10px 20px;
  }
}
```

Media Types

```
// All the devices
@media all { ... }

// Print mode
@media print { ... }

// Screen devices
@media screen { ... }

// Speech synthesizers
@media speech { ... }
```

Media Features

```
// 500px and narrower (e.g. a phone)
@media (max-width: 500px) { ... }

// 501px and wider
@media (min-width: 501px) { ... }

// Primary input can hover
@media (hover: hover) { ... }

// Dark mode preference
@media (prefers-color-scheme: dark) ...
```

Multiple criteria

```
@media screen
  and (min-width: 320px)
  and (orientation: portrait) {
    ...
  }
```

Negate query

```
// Invert the whole media query
@media not screen
  and (min-width: 320px)
  and (orientation: portrait) {
    ...
  }
```

Combine queries

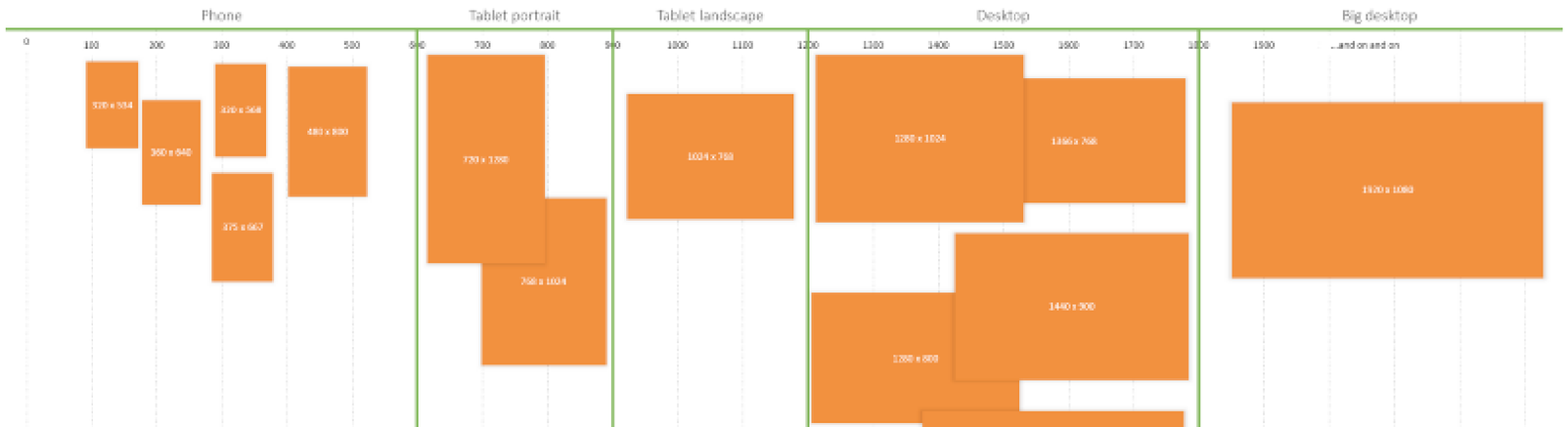
```
// Applied to both print mode
// and screen with width >= 320
@media print, screen
  and (min-width: 320px) {
    ...
  }
```

Viewport

Represents the currently viewed area of the page.

In CSS pixels. High resolution screens display multiple physical pixels per CSS pixels.

On mobile viewport not always equal to the size of the device, by default. It is wider than the screen and renders zoomed out.



Viewport Meta Tag

You can control the size and scale of the viewport by the meta tag

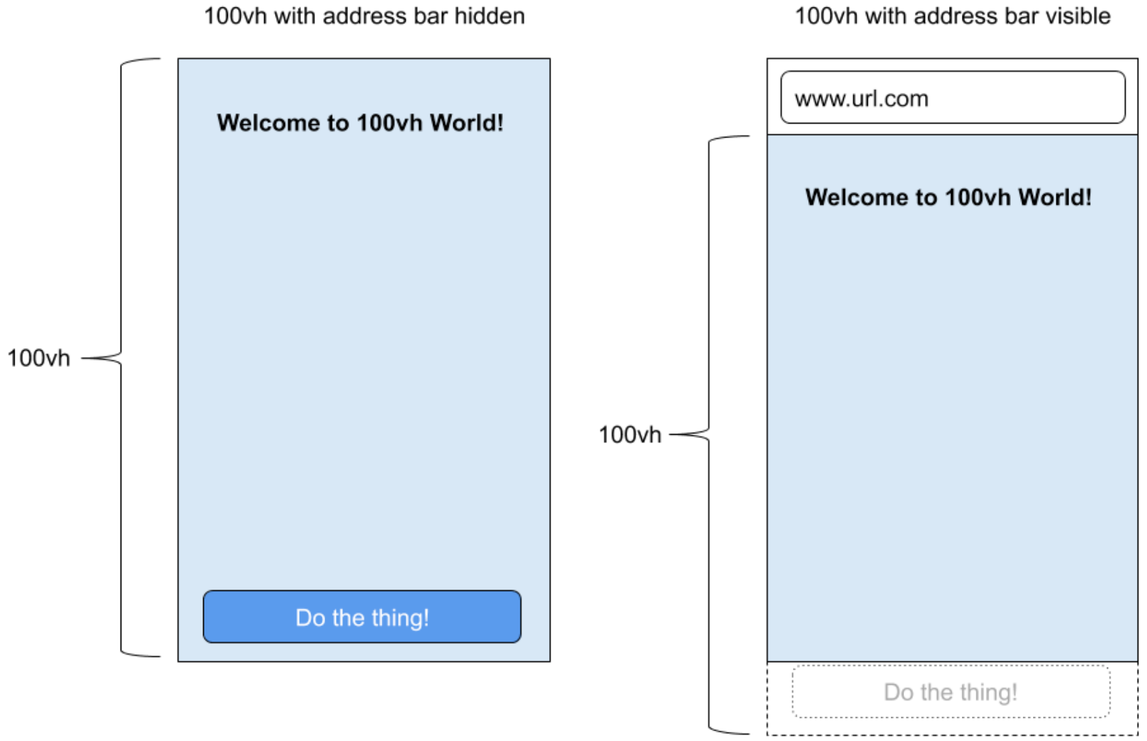
```
// Sets viewport size to the actual screen
// size of the device
<meta
  name="viewport"
  content="width=device-width"
>

// Controls the zoom of the page
<meta
  name="viewport"
  content="initial-scale=1, maximum-scale=2"
>
```

Viewport units

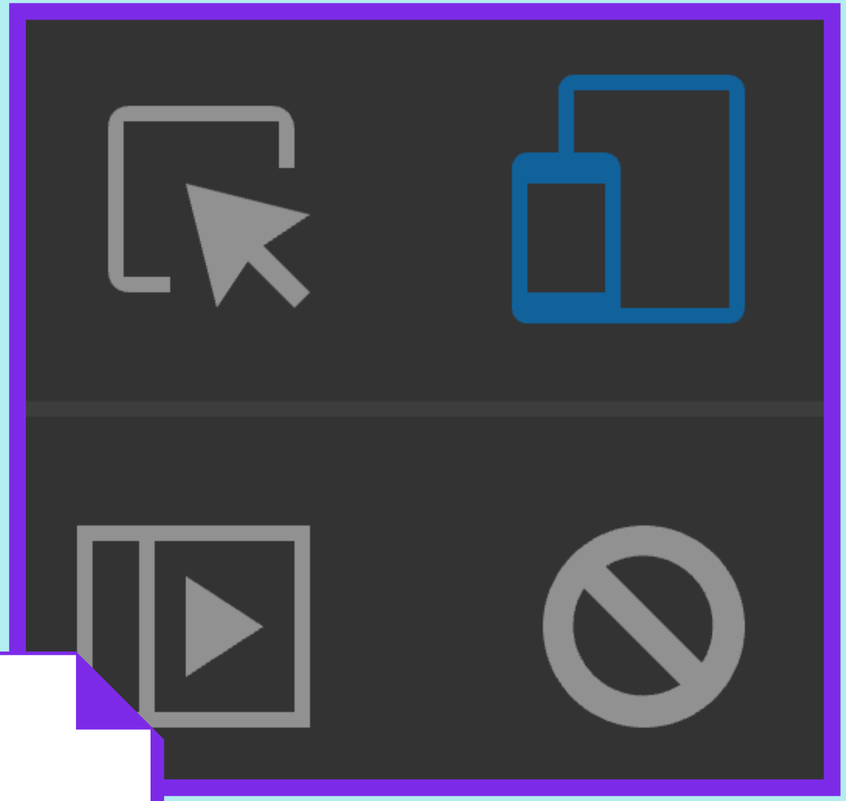
```
.box {
  // 10% of viewport width
  width: 10vw;
  // 10% of viewport height
  height: 10vh;
}
```

Warning, VH is counter-intuitive



https://chanind.github.io/javascript/2019/09/28/avoid-100vh-on-mobile-web.html

Dev Tools



Chrome DevTools has an instrument that allows to change the viewport

Device Mode

You can change the size of the viewport in Chrome device toolbar by choosing of the predefined devices or set the custom sizes

Note: its only change the viewport, not emulates the actual device

It also allows to change Orientation and Device Pixel Ratio

COMP6080
Web Front-End Programming

CSS Frameworks (Basic)

Basic CSS Frameworks

After a while, people started realising that very basic components (buttons, headers, cards) often had similar requirements for simple websites.

When you aren't building a boutique, high quality, or customer website, you just want something that is "good enough" and can get together quickly.

CSS Frameworks were developed and supported for this purpose.

Twitter Bootstrap

While there are many frameworks out there for use in vanilla websites, twitter bootstrap is one of the oldest and most common.

We will demonstrate it's use through a demo.

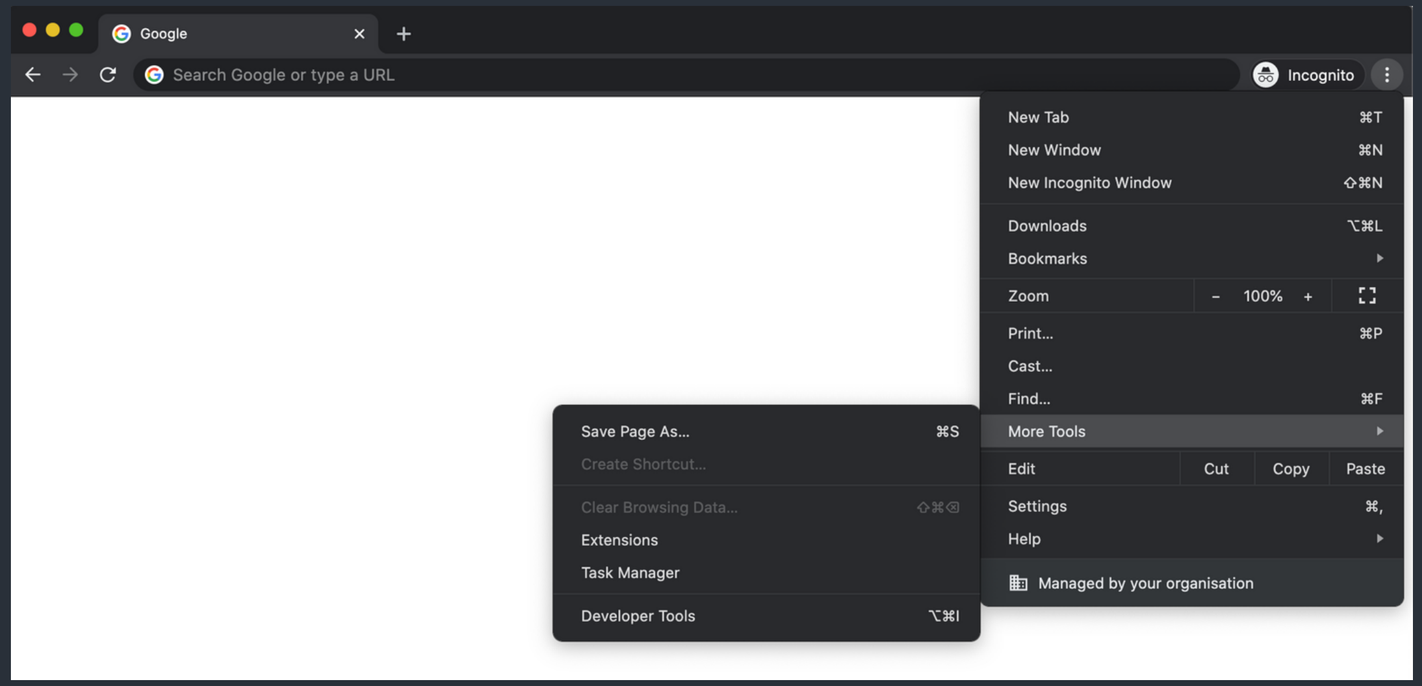
DevTools

Built-in tools in all modern web browsers

- Debugging
- Prototyping
- Analysing

Open DevTools

Main menu - More Tools - Developer Tools



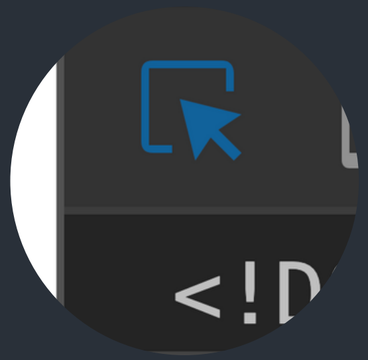
Shortcuts

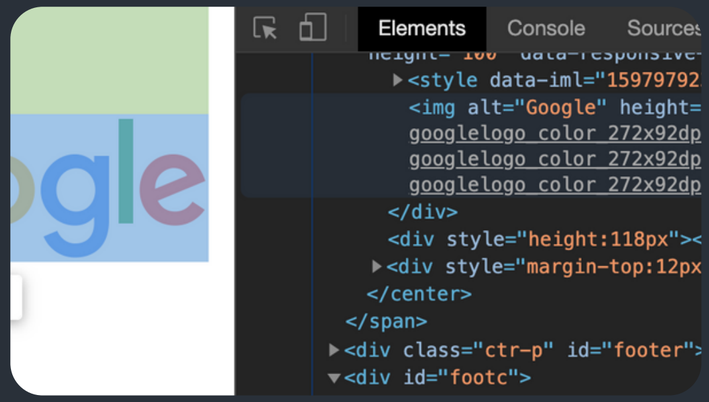
Mac: Command + Option + I

Windows: Control + Shift + I

Select Element

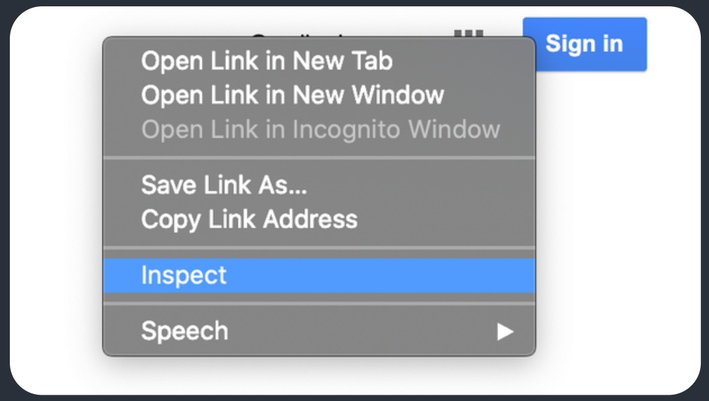
Select an element in the page



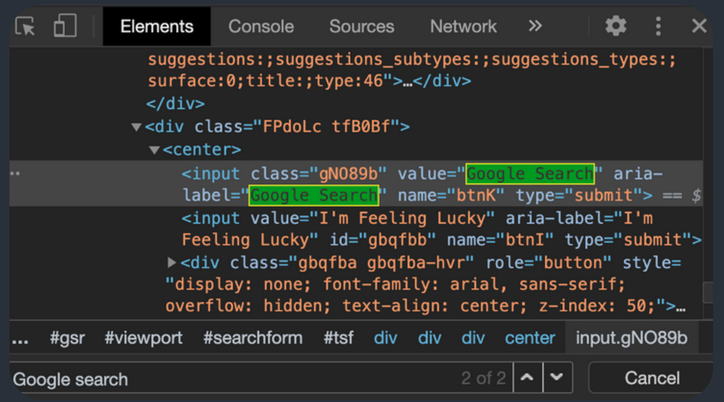


Elements panel

Context menu - Inspect

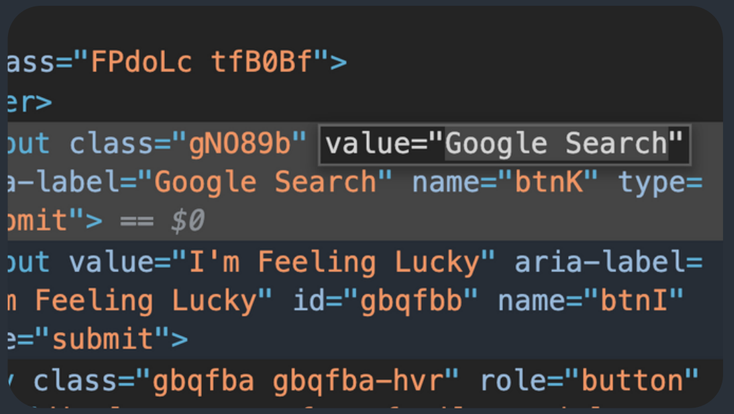


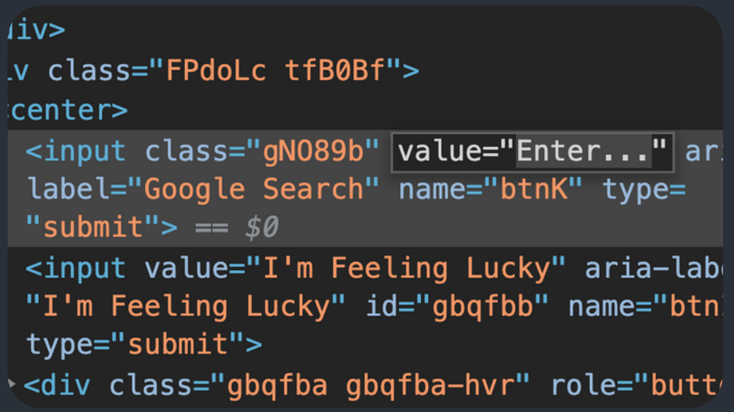
Search



Edit DOM

- Edit a node: **double click on tag/attribute/content**
  - Tag
  - Attributes
  - Content
- Delete a node: **select a node - press "Delete"**
- Reorder nodes: **select a node - drag**
- Hide a node: **select a node - press "H"**

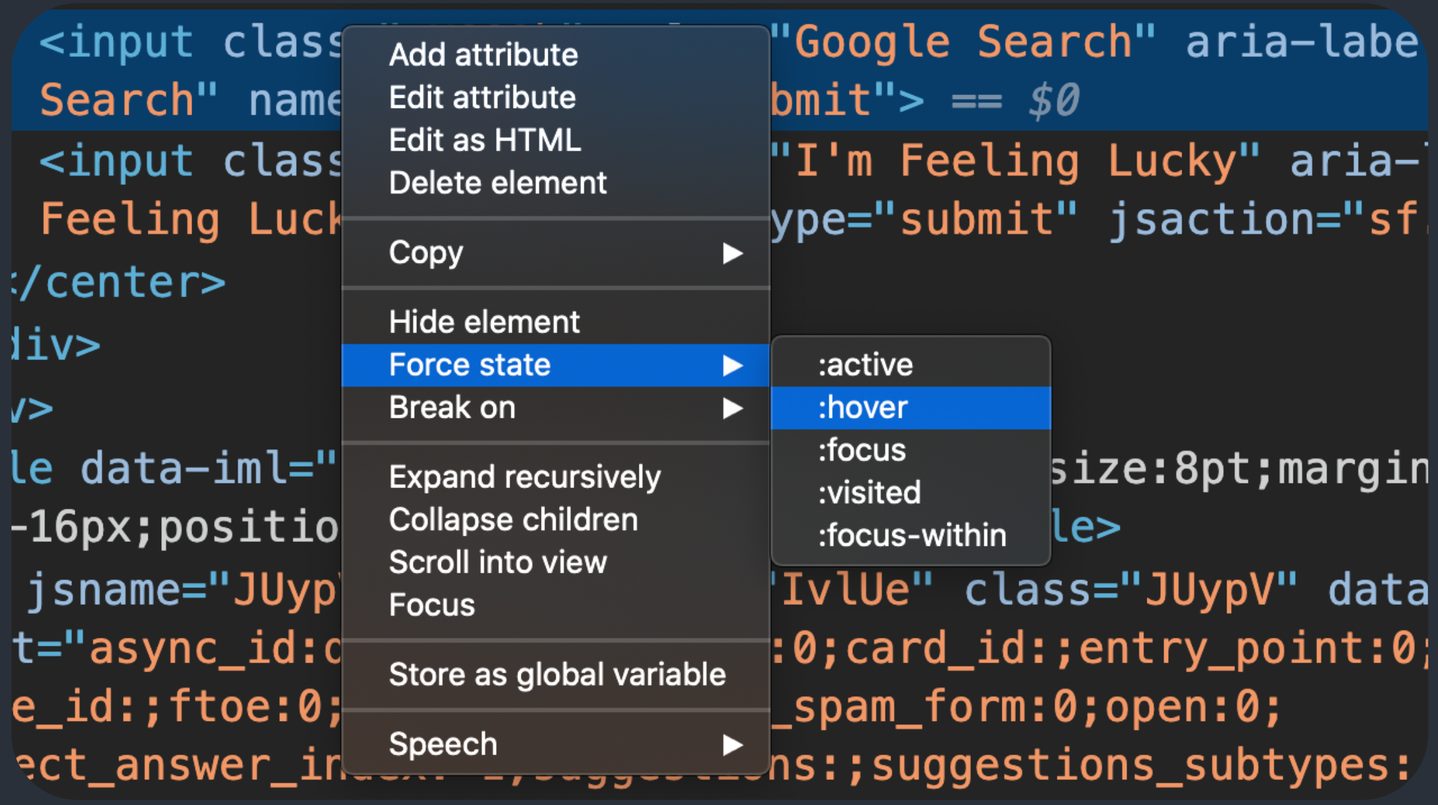




Force State

States

- active
- hover
- focus
- visited
- focus-within



Styles tab

A part of Elements panel which allows to manipulate with the styles of the selected element

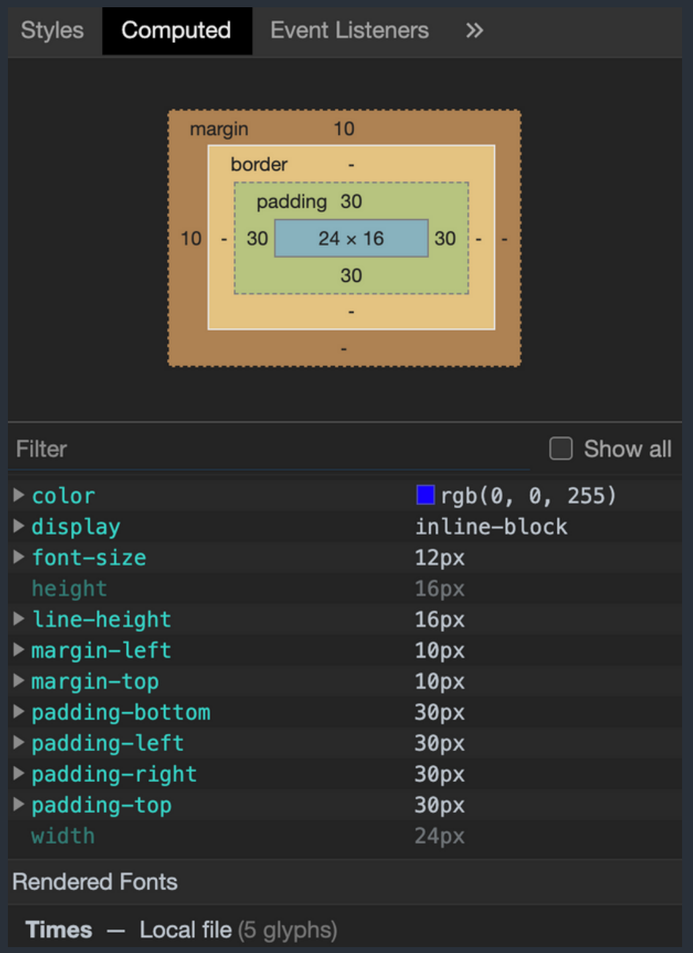
Allows to:

- See all the style rules applied to an element
- Add/delete properties
- Change the value of the property
- Add new style rules

Computed styles

Shows all the styles which are actually applied to the selected element

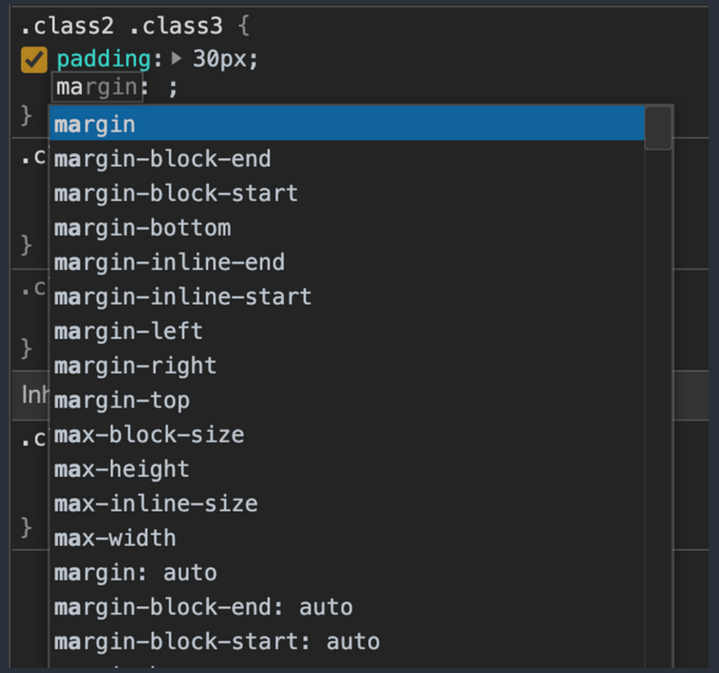
Shows the box model for the selected element



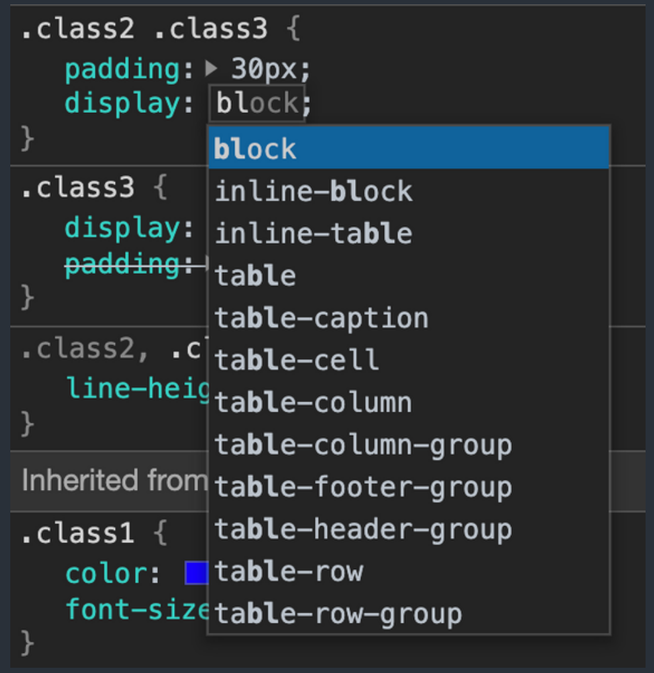
Name/value hints

DevTools will shows the hints when you start to type properties name/value

Property hint

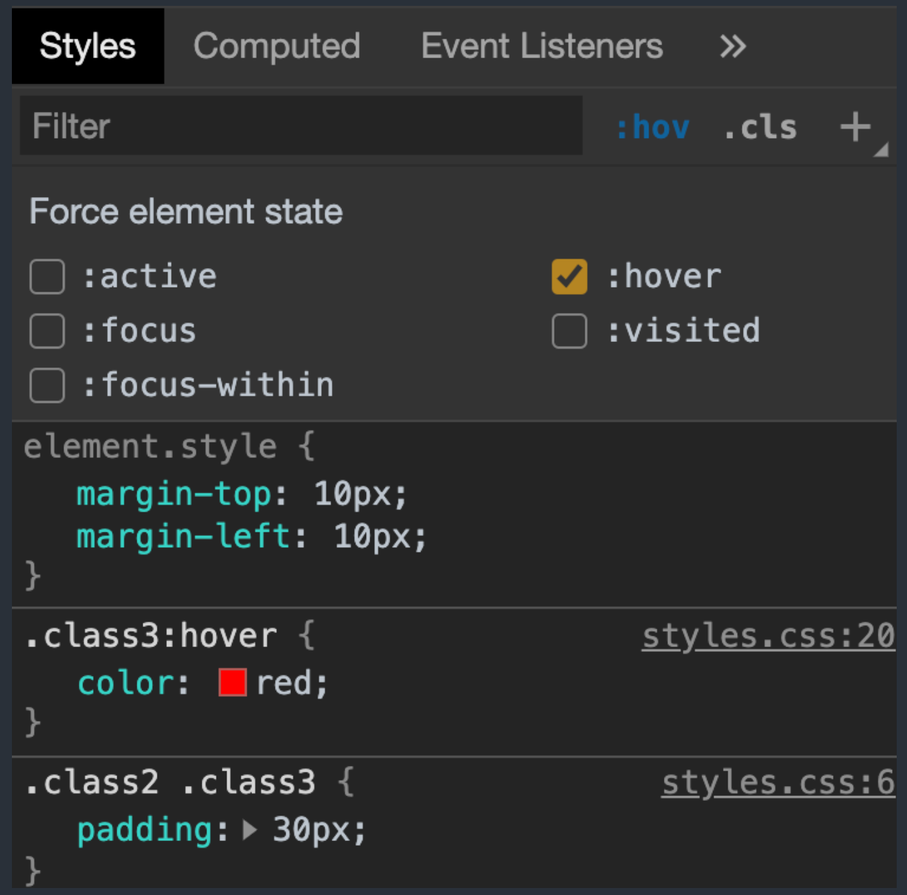


Value hint



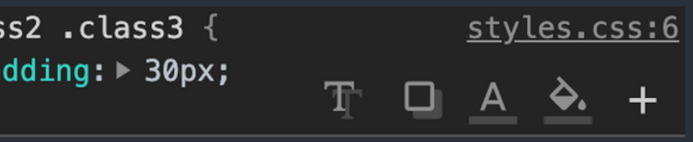
Element state

You can force the state of the selected element in the Style tab and see the style rules applied to this state

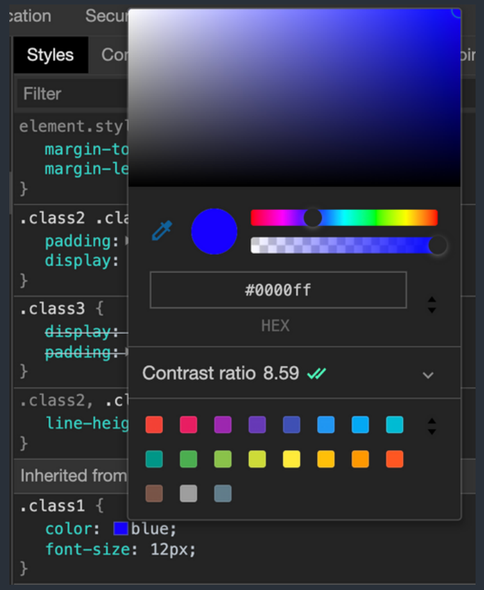


More style tools

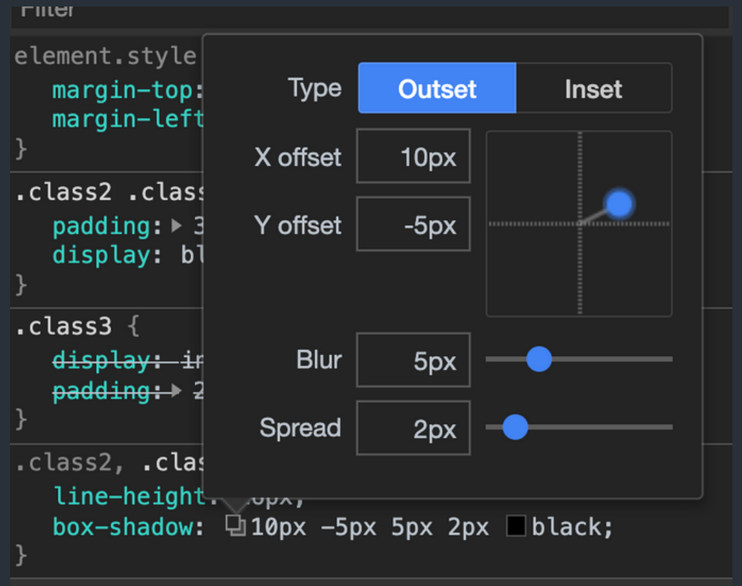
Chrome has more tools which helps to set the correct values of the properties



Color picker

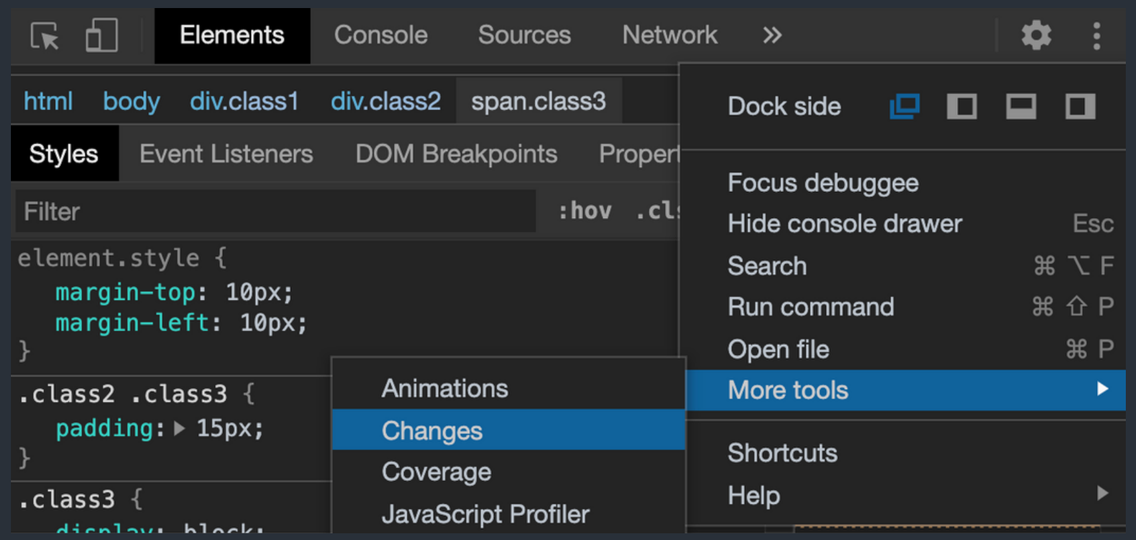


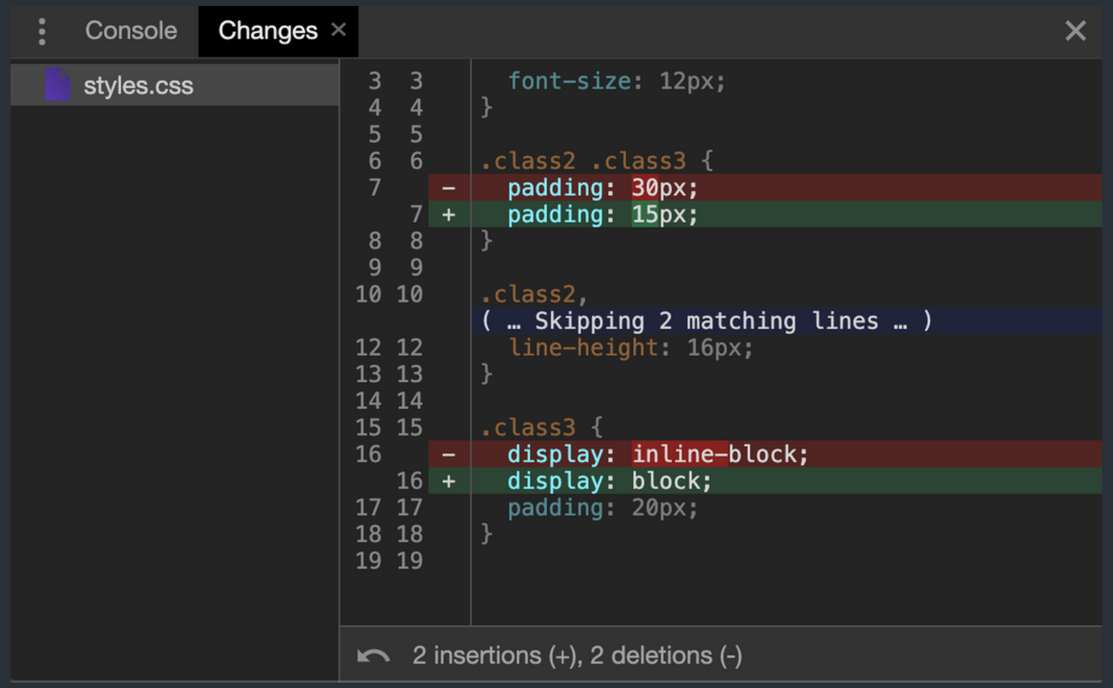
Box shadow



Local changes

In the settings of DevTools you can find more tools,
e.g. Changes which shows all the local changes you did

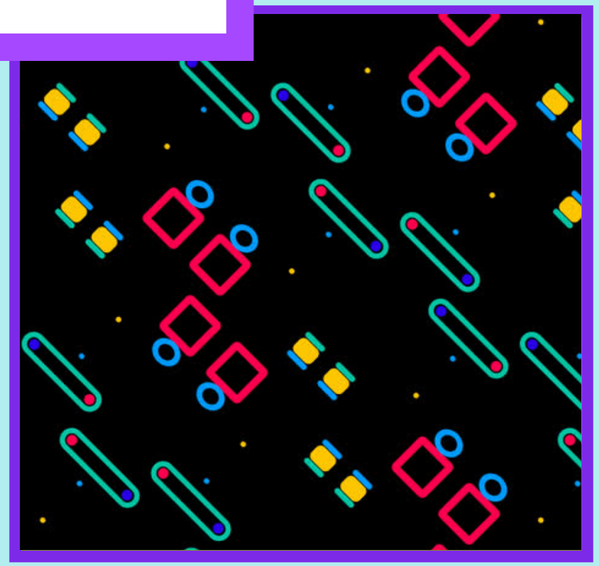


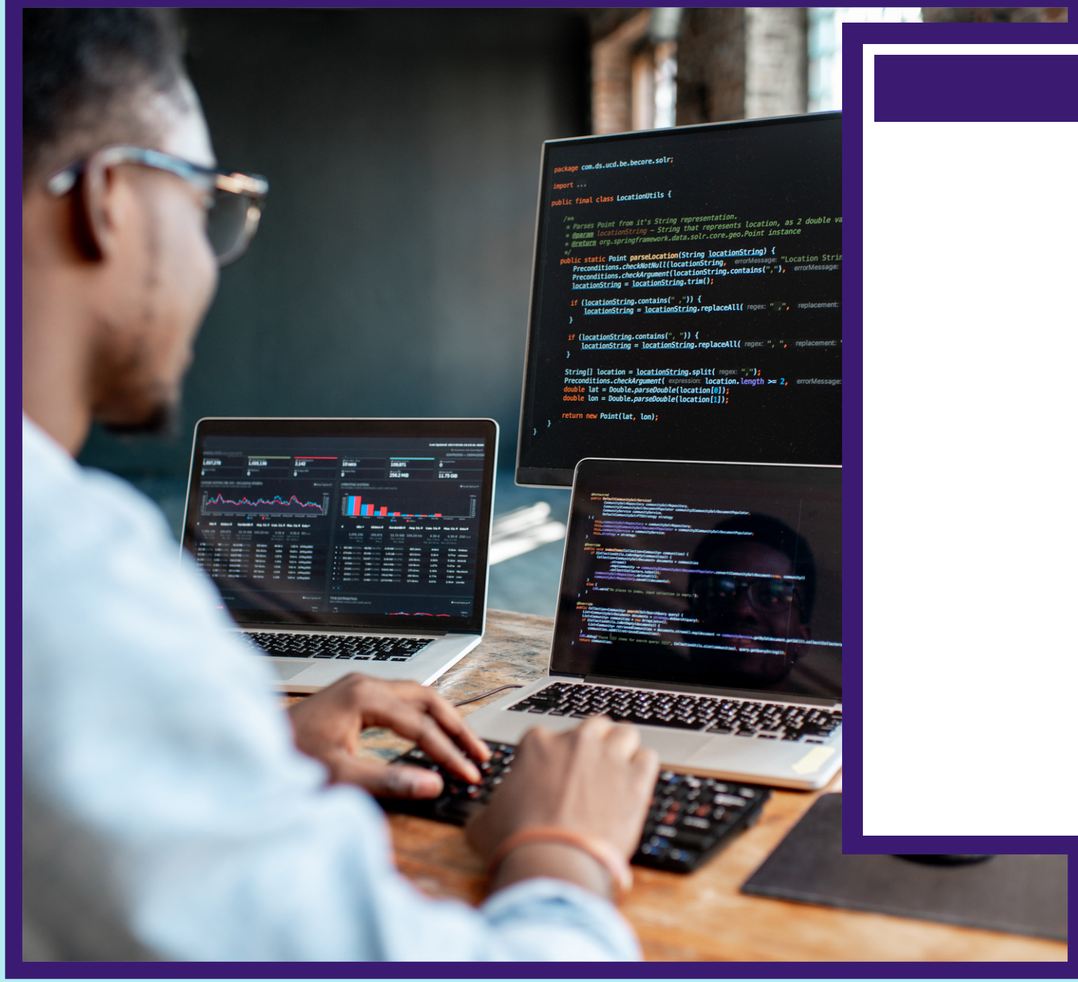


Javascript Syntax

Prepared by
Denis Tokarev

Canva





Basic JS syntax

The syntax, expressions, and operators

VARIABLES & CONSTANTS

You can define them using one of the keywords →

```
01 | // An immutable value available from the line
02 | // where the constant is defined
03 | const constValue = 'hey';
04 |
05 | // A mutable value available from the line
06 | // where the variable is defined
07 | let variableValue = 0;
08 |
09 | // A mutable value, old syntax. Hoists to the beginning
10 | // of the function scope or to the global scope,
11 | // whatever is closer. Not recommended for use.
12 | var anotherVariableValue = false;
13 |
14 |
15 |
16 |
17 |
18 |
19 |
20 |
21 |
22 |
23 |
24 |
25 |
```

PRINTING ARBITRARY VALUES TO CONSOLE

Just use **console.log** →

```
01 | // Let's say we have a constant storing a string
02 | const name = 'Bindi';
03 |
04 | // And a variable storing a number
05 | let age = 22;
06 |
07 | // And an object
08 | let wildlifeWarrior = {
09 |   name: name,
10 |   age: age,
11 |   gender: 'f',
12 | };
13 |
14 | // This will print 'Hello, world!' to console
15 | console.log('Hello, world!');
16 |
17 | // Additionally, you can print any value
18 | console.log(name); // Prints 'Bindi'
19 | console.log('name:', name); // Prints 'name: Bindi'
20 | console.log('age: ' + age); // Prints 'age: 22'
21 |
22 | // This will print an object as a string
23 | console.log('Person:', wildlifeWarrior);
24 | // Person: { name: 'Bindi', age: 22, gender: 'f' }
25 |
```

DATA TYPES

JavaScript is a dynamically typed language, meaning that the same variable can contain values of different types →

```
01 | let value;
02 |
03 | // A float or integer number
04 | value = 0.1;
05 |
06 | // A string, double quotes "" can also be used
07 | value = 'some-string';
08 |
09 | // In JavaScript, null is a separate type
10 | value = null;
11 |
12 | // This type represents no value and no variable
13 | value = undefined;
14 |
15 | // A boolean value, either true or false
16 | value = true;
17 |
18 | // A unique value
19 | value = Symbol();
20 |
21 | // An object, which is basically a dictionary
22 | // of dynamically typed values identified by their keys
23 | value = { key: 'hey', anotherKey: 10 };
24 |
25 |
```

DATA TYPES

Functions are first-class citizens, meaning that they can be assigned to variables and can be referenced →

```
let value;

// An anonymous function
value = (param1, param2) => {
  console.log(param1, param2);
};

// A normal function
value = function(param) {
  console.log('param =', param);
};

// Function can be defined without assigning it
// to a variable, in that case, it hoists
function foo(bar) {
  // If number, adds 1 to it, otherwise concatenates it
  return bar + 1;
}

// Calling a function
const result = foo(10);
value(result); // Prints 'param = 11'

// A big integer number
value = BigInt(9007199254740991);
```

DATA TYPES

You can always find out the type of the value that the varable currently holds, at runtime →

However, in some cases, you'd need a more complex check

```
01 | let value;
02 |
03 | value = 0.1;
04 | console.log(typeof value); // Prints 'number'
05 |
06 | value = 'some-string';
07 | console.log(typeof value); // Prints 'string'
08 |
09 | value = null;
10 | console.log(typeof value); // Prints 'object'
11 |
12 | value = undefined;
13 | console.log(typeof value); // Prints 'undefined'
14 |
15 | value = true;
16 | console.log(typeof value); // Prints 'boolean'
17 |
18 | value = Symbol();
19 | console.log(typeof value); // Prints 'symbol'
20 |
21 |
22 |
23 |
24 |
25 |
```

DATA TYPES

You can always find out the type of the value that the varable currently holds, at runtime →

However, in some cases, you'd need a more complex check

```
01 | let value;
02 |
03 | value = { key: 'hey', anotherKey: 10 };
04 | console.log(typeof value); // Prints 'object'
05 |
06 | value = function() { /* ... */ };
07 | console.log(typeof value); // Prints 'function'
08 |
09 | value = BigInt(9007199254740991);
10 | console.log(typeof value); // Prints 'bigint'
11 |
12 |
13 |
14 |
15 |
16 |
17 |
18 |
19 |
20 |
21 |
22 |
23 |
24 |
25 |
```

DATA TYPES

undefined is also the type of a variable that hasn't been assigned a value yet →

#justjavascriptythings

```
01 | let value;
02 |
03 | console.log(typeof value); // Prints 'undefined'
04 |
05 | // For more information on data types,
06 | // visit the MDN page:
07 | // http://mdn.io/Data_structures
08 |
09 | // I also recommend watching this old but gold talk
10 | // by Gary Bernhardt:
11 | // https://www.destroyallsoftware.com/talks/wat
12 | // This talk makes fun of some JS things like
13 | undefined == null
14 | // being true, and
15 | undefined === null
16 | // being false
17 |
18 |
19 |
20 |
21 |
22 |
23 |
24 |
25 |
```

TYPE CONVERSIONS

It's possible to convert values of one type into another, for some of the supported types →

```
01 | let value = 10;
02 |
03 | let strValue = value.toString();
04 | // or: let strValue = value + '';
05 |
06 | let numValue = parseInt(strValue, 10);
07 | // or: let numValue = parseFloat(strValue);
08 | // or: let numValue = +strValue;
09 |
10 | let bigIntValue = BigInt(numValue);
11 | // or: let bigIntValue = BigInt(strValue);
12 |
13 | let numValue = Number(bigIntValue);
14 | // or: let numValue = parseInt(bigIntValue, 10);
15 |
16 |
17 |
18 |
19 |
20 |
21 |
22 |
23 |
24 |
25 |
```

COMPARING VALUES

There is a strict equality check and a loose equality check, and the strict check is generally preferred whenever possible →

```
let value = 10;
let anotherValue = '10';

// With a loose equality check (aka '=='), two variables
// are considered as holding identical values
console.log(value == anotherValue); // Prints 'true'

// With a strict equality check (aka '==='), the result
// will indicate that the stored values are different
console.log(value === anotherValue); // Prints 'false'

// Comparing with an implicit type case can result
// in a pretty weird outcome and should be avoided
// when possible
const str = '[object Object]';
const obj = { catsSay: 'meow', dogsSay: 'woof' };
console.log(str == obj); // Prints 'true' ¯\_(ツ)_/¯
console.log(str === obj); // Prints 'false'

// For more information, please check out these pages:
// https://mdn.io/Equality_comparisons_and_sameness
// http://ecma-international.org/ecma-262/5.1/#sec-11.9.3


```

PRIMITIES

Object, Array, Map, Set

```
01 | // An object is a dictionary of dynamically typed values
02 | const obj = { key: 'some-value', anotherKey: 10.2 };
03 | console.log(obj.key); // Prints 'some-value'
04 | console.log(obj['key']); // Prints 'some-value'
05 |
06 | // An array is an indexed list of dynamically typed
07 | // values. Every Array is also an object.
08 | const arr = ['first value', 2, obj];
09 | console.log(arr[0]); // Prints 'first value'
10 | console.log(arr.length); // Prints '3'
11 | console.log(typeof arr); // Prints 'object'
12 |
13 | // A Map is an object that can use any value as a key
14 | // (not only strings) and preserves the original
15 | // element order.
16 | const map = new Map();
17 | map.set('one', 1);
18 | console.log(map.get('one')); // Prints '1'
19 |
20 | // A Set is an object that contains unique values
21 | const set = new Set();
22 | set.add('one');
23 | set.add('one'); // 'one' isn't added the second time
24 | console.log(set.has('one')); // Prints 'true'
25 |
```

CONDITIONALS

Conditionals look like their alternatives in other C-like languages

The value in parentheses automatically gets casted to a boolean value

```
01 | // One-liner if condition
02 | if (condition) doSomething(); // If condition is truthy
03 |
04 | // Multiline if
05 | if (condition) {
06 |   // If condition is truthy
07 | }
08 |
09 | // If with else
10 | if (confition) {
11 |   // If condition is truthy
12 | } else {
13 |   // If condition is falsy
14 | }
15 |
16 | // If with multiple branches
17 | if (condition1) {
18 |   // If condition1 is truthy
19 | } else if (condition2) {
20 |   // If condition2 is truthy
21 | } else {
22 |   // If both condition1 and condition2 are falsy
23 | }
24 |
25 |
```

CONDITIONALS

Using the **switch** statement often is more convenient and improves the code readability →

Don't forger about the **break** at the end of each case!

```
01 | const value = 5;
02 |
03 | // For mulitple branches, it's often more
04 | // convenient to use a switch-case block
05 | switch (value) {
06 |   case 0:
07 |     // If value equals to 0, the below code runs
08 |     console.log('No items in the bag!')
09 |     break;
10 |   case 1:
11 |     // If value equals to 1, the below code runs
12 |     console.log('One item is in the bag!');
13 |     break;
14 |   case 2:
15 |     // If value equals to 2, the below code runs
16 |     console.log('A couple of items is in the bag!');
17 |     break;
18 |   default:
19 |     // If value isn't 0, 1, or 2, the below code runs
20 |     console.log('Lots of items are in the bag');
21 | }
22 |
23 |
24 |
25 |
```

CONDITIONALS

Inline if, aka the Ternary operator, is a convenient way to write less code and make it easier to understand →

```
01 | // Inline if (aka the ternary operator)
02 | condition ? ifTrue() : ifFalse();
03 |
04 | // It is an operator, meaning that it returns a value
05 | const result = condition ? valIfTrue() : valIfFalse();
06 |
07 | // Additionally, you can use lazy || and && operators
08 | condition && ifTrue() || ifFalse();
09 |
10 | // The above also returns a value
11 | // (this is also known as rvalue in some languages)
12 | const result = condition && valIfTrue() || valIfFalse();
13 |
14 | // Thanks to the && operator, it's possible to write
15 | // a short version of a one-liner if:
16 | condition && runSomething();
17 | // which is a single line equivalent to
18 | if (condition) {
19 |   runSomething();
20 | }
21 |
22 |
23 |
24 |
25 |
```

CONDITIONALS

As said above, the condition inside parentheses gets auto-casted to a boolean value

However, to avoid accidental unexpected bugs, still, you must understand how the type casting works →

```
01 | if (true) {
02 |   console.log('Always prints');
03 | }
04 |
05 | if (false) {
06 |   console.log('Never prints');
07 | }
08 |
09 | if ('hey') {
10 |   console.log('Always prints');
11 | }
12 |
13 | if ('') {
14 |   console.log('Never prints');
15 | }
16 |
17 | if ('0') {
18 |   console.log('Always prints');
19 | }
20 |
21 | if (0) {
22 |   console.log('Never prints');
23 | }
24 |
25 |
```

CONDITIONALS

As said above, the condition inside parentheses gets auto-casted to a boolean value

However, to avoid accidental unexpected bugs, still, you must understand how the type casting works →

```
01 | if (1) {
02 |   console.log('Always prints');
03 | }
04 |
05 | if (null) {
06 |   console.log('Never prints');
07 | }
08 |
09 | if (undefined) {
10 |   console.log('Never prints');
11 | }
12 |
13 | if ({ x: 'test' }) {
14 |   console.log('Always prints');
15 | }
16 |
17 | if (Symbol()) {
18 |   console.log('Always prints');
19 | }
20 |
21 | if (function x() {}) {
22 |   console.log('Always prints');
23 | }
24 |
25 |
```

CONDITIONALS

And sometimes, it gets really tricky →

```
01 | if (BigInt(1020202)) {
02 |   console.log('Always prints');
03 | }
04 |
05 | if (BigInt(0)) {
06 |   console.log('Never prints');
07 | }
08 |
09 | if ('1' - 1) {
10 |   console.log('Never prints');
11 | }
12 |
13 | if (+'0') {
14 |   console.log('Never prints');
15 | }
16 |
17 | if (new String('')) {
18 |   console.log('Always prints');
19 | }
20 |
21 | if (new Number(0)) {
22 |   console.log('Always prints');
23 | }
24 |
25 |
```

LOOPS

JavaScript supports the same syntax for the loops that many curly bracket syntax languages like C do

```
01 | // Good old for loop
02 | for (let i = 0; i < 100; i++) {
03 |   console.log(i);
04 | }
05 |
06 | // An equivalent while loop
07 | let i = 0;
08 | while (i < 100) {
09 |   console.log(i);
10 |   i++;
11 | }
12 |
13 | // Or equivalent do...while loop
14 | let i = 0;
15 | do {
16 |   ++i;
17 |   if (i >= 100) {
18 |     break;
19 |   }
20 |
21 |   console.log(i);
22 | } while (true);
23 |
24 |
25 |
```

CLASSES

Modern JavaScript also supports classes, allowing for a convenient way to implement abstractions and ship the data along with the functions that can process it →

```
01 | // Defining a new class
02 | class Snack {
03 |   // A consuctor that will be auto-called
04 |   constructor(calories, name) {
05 |     this.caloriesRemaining = calories;
06 |     this.name = name;
07 |   }
08 |
09 |   // One of attached functions, usually called 'methods'
10 |   chew() {
11 |     this.caloriesRemaining -= 100;
12 |   }
13 | }
14 |
15 |
16 |
17 |
18 |
19 |
20 |
21 |
22 |
23 |
24 |
25 |
```

CLASSES

A JavaScript class can be extended and can have static members

```
01 | class Snack {
02 |   constructor(calories, name) {
03 |     this.caloriesRemaining = calories;
04 |     this.name = name;
05 |   }
06 |   chew() {
07 |     this.caloriesRemaining -= 100;
08 |   }
09 | }
10 |
11 | class Pizza extends Snack {
12 |   static caloriesPerGram = 2.66;
13 |
14 |   constructor(weightInGrams) {
15 |     super(Pizza.caloriesPerGram * weightInGrams, 'XL');
16 |   }
17 | }
18 |
19 | class Crisps extends Snack {
20 |   static caloriesPerGram = 5.36;
21 |
22 |   constructor(weightInGrams) {
23 |     super(Crisps.caloriesPerGram * weightInGrams, 'Thins');
24 |   }
25 | }
```

CLASSES

With classes, you can use all features and patterns established in object-oriented programming

However, internally, each class is just a constructor function, because JavaScript OOP is based on 'prototyping'

```
01 | class Meal {
02 |   constructor(snacks) {
03 |     this.snacks = snacks;
04 |   }
05 |
06 |   eatSome() {
07 |     for (let i = 0; i < this.snacks.length; i++) {
08 |       const snack = this.snacks[i];
09 |       console.log('Chewing', snack.name);
10 |       snack.chew();
11 |     }
12 |   }
13 |
14 |    logCalories() {
15 |     for (let i = 0; i < this.snacks.length; i++) {
16 |       const snack = this.snacks[i];
17 |       console.log(snack.name, ': ', snack.caloriesRemaining);
18 |     }
19 |    }
20 | }
21 |
22 | console.log(typeof Meal); // Prints 'function'
23 |
24 |
25 |
```

CLASSES

A class is just a constructor function, and a class instance is an object

```
const snacks = [
  new Pizza(800),
  new Crisps(150),
];

const meal = new Meal(snacks);
meal.logCalories();
meal.eatSome();
meal.logCalories();

console.log(typeof Meal); // Prints 'function'
console.log(typeof meal); // Prints 'object'
```

SIMPLE FIBONACCI SEQUENCE GENERATOR IN JAVASCRIPT

So hopefully, this example's got a bit easier to understand →

```
01 | const fib = (n) => {
02 |   if (n === 0) {
03 |     return 0;
04 |   }
05 |
06 |   if (n === 1) {
07 |     return 1;
08 |   }
09 |
10 |   let current = 0;
11 |   let previous = 1;
12 |   let prePrevious = 0;
13 |   // Iterative calculation
14 |   for (let i = 0; i < n; i++) {
15 |     prePrevious = previous;
16 |     previous = current;
17 |     current = prePrevious + previous;
18 |   }
19 |
20 |   return current;
21 | };
22 |
23 | console.log(fib(5));
24 |
25 |
```

COMP6080
Web Front-End Programming

Week 2
The Javascript Ecosystem

What even is "Javascript"?

- Is it what Google Chrome has?
- Is it what NodeJS is?
- Is ReactJS different from Javascript?
- What version am I using?

Let's take a step back....

Language V Compiler/Interpreter

Language Definition (Standards)

- Describes how a language should function (rules, syntax)
- Typically defined as a globally recognised standard
- New features to a language mean new versions of the language

Compiler or Interpreter

- A program that takes source code (plain text) from you, and, following language definition rules, produces runnable code for execution
- E.G. Python3, Node, Gcc

Source Code (plain text)

- Programs that you write in .py, .js, .c, .cpp, .java files.
- Fundamentally just plain text (ascii) that compilers interpret based on a language definition

Language V Compiler (Interpreter)

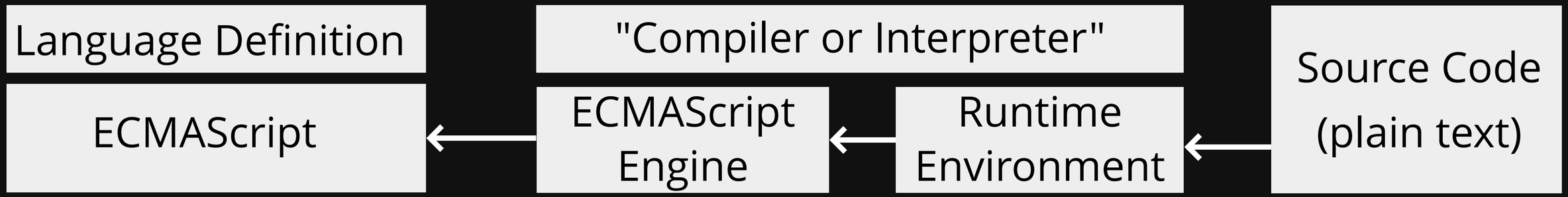
Language Definition (Standards)

Compiler or Interpreter

Source Code (plain text)

Compilers/Interpreters take source code (plain text) and produce executable programs. The way to interpret the source code into executable programs is provided in the language definition.

"Javascript"



- ECMAscript (ES)
- First appeared 1997
- Major releases are:
  - ES5 (ECMAScript 2009)
  - ES6 (ECMAScript 2015)
  - ECMAScript 2016
  - ... etc
  - ECMAScript 2019

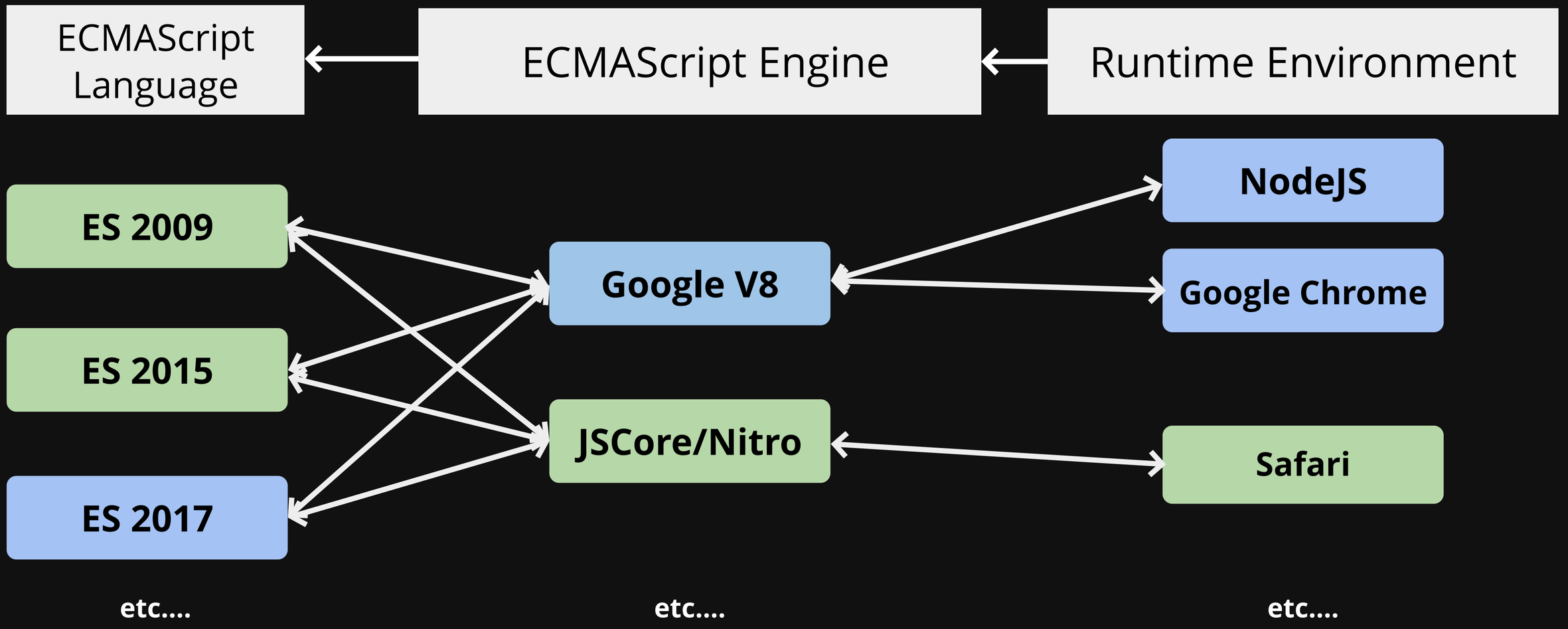
An article about language features

- Javascript compilers or interpreters are known as **runtime environments.**
  - Examples of runtime environments include NodeJS, Google Chrome
- Runtime environments are built on top of **ECMAScript Engines** which are the engines that interpret the ES language and produce runnable code. They do not have I/O nor do they have APIs
  - Examples of engines include V8, Nitro
- Let's chat about this more...

- .js files you write

Javascript refers to a runtime environment that is built on top of an ECMAScript engine

"Javascript"



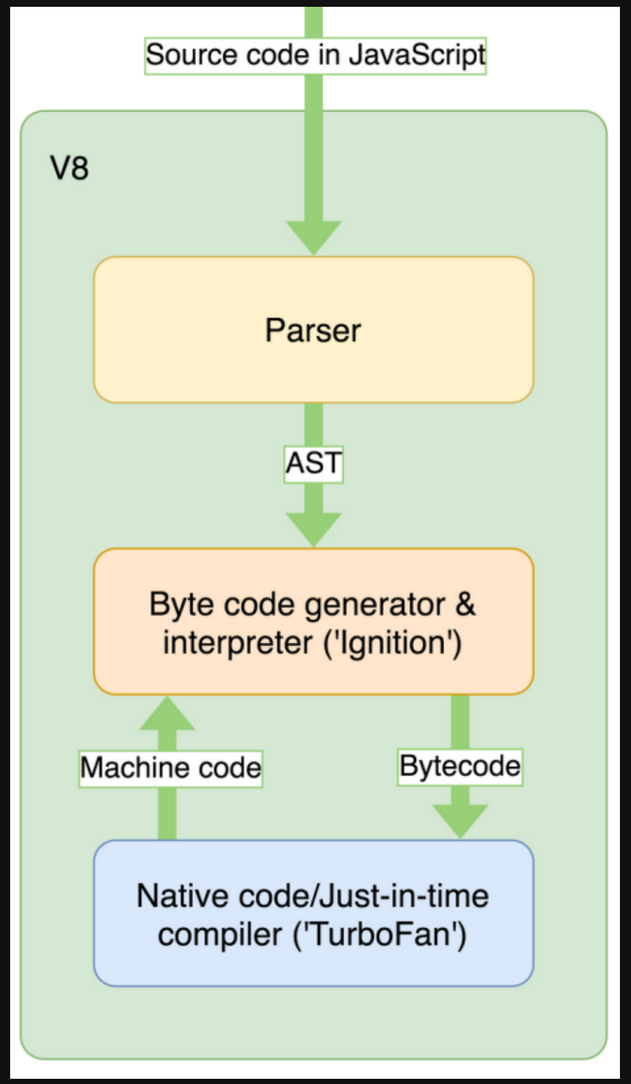
Each version of a runtime environment is built off a particular version of an ECMAScript engine.

Each version of an ECMAScript engine is built to a particular version of ECMAScript

Google V8 Engine

Google's V8 Engine is an open-source Javascript execution engine, a part of the Chromium project. It can run standalone, or can be embedded and extended into any C++ application as a library. V8 is just a compiler+vm toolset, it does not have I/O and APIs built in.

V8 parses, interprets, executes and compiles Javascript code. It is shipped ONLY with the APIs that the ECMAScript Standard specifies.

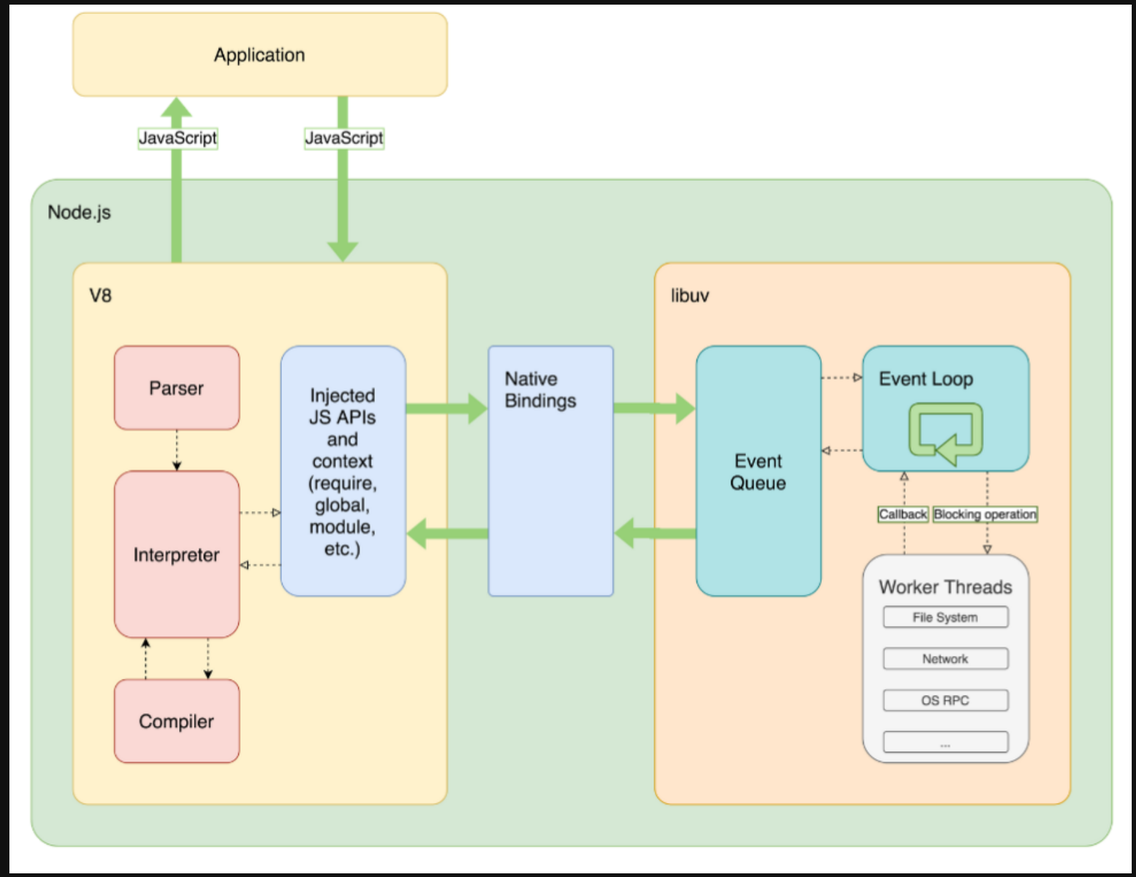


Node.JS

NodeJS is a javascript runtime, with easy to use **command-line** capabilities, that is built on Chrome's V8 Javascript engine.

V8 only provide the core parsing and compiling, but features such as the async event loop/queue are built on top as part of NodeJS.

NodeJS also ships with I/O APIs for network, file system operations, and the concept of modules.

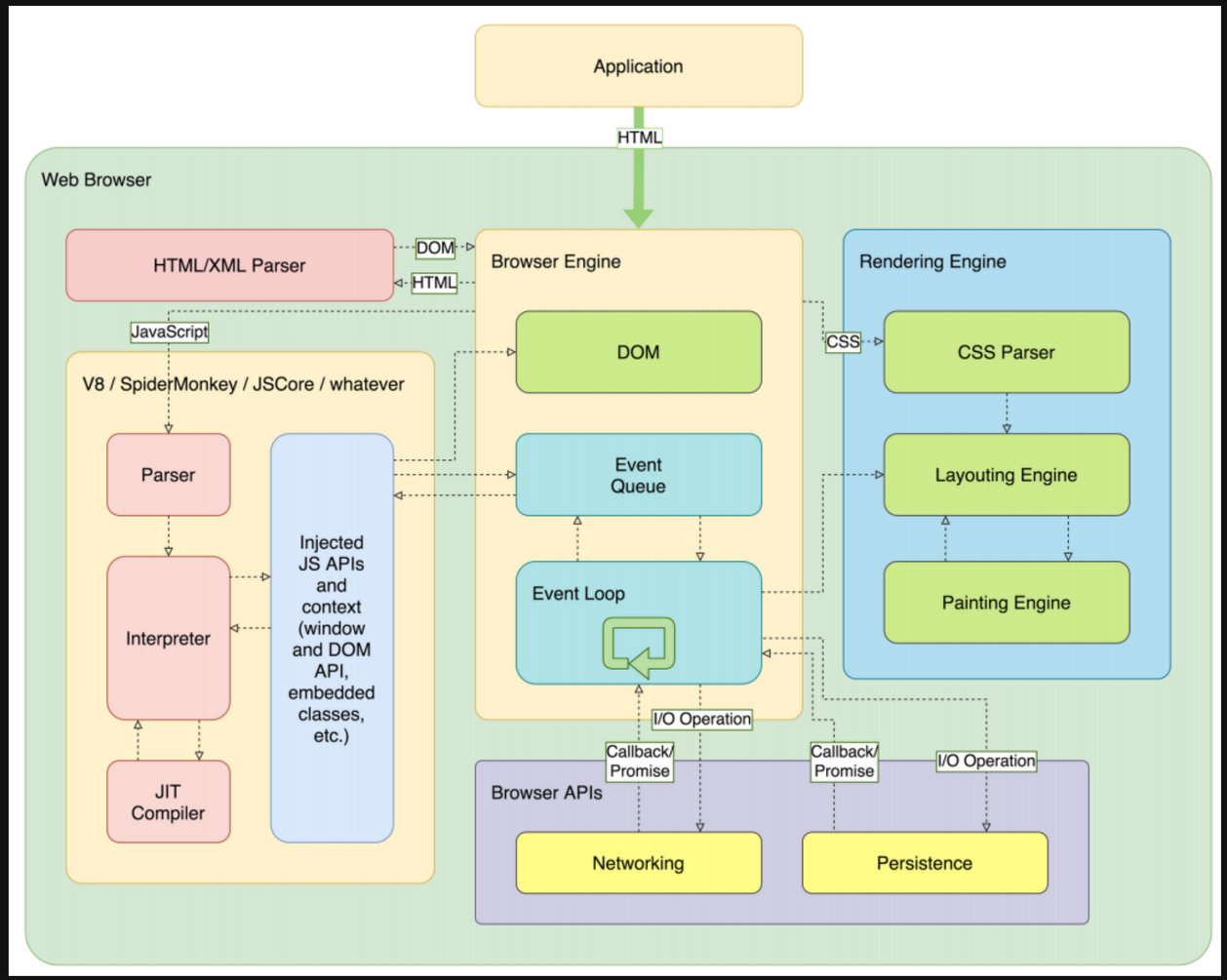


Google Chrome (any web browser)

A web browser is a HTML & CSS document renderer for client-side user interfaces.

The Javascript V8 engine is a small but critical part of a web browser that allows for the execution of Javascript. The primary purpose of Javascript execution in web browsers is to:

- Mutate the DOM
- Make network requests
- Persist data client-side



Feedback

COMP1531

🛠️ Development - Advanced Functions

Lecture 4.1

Author(s): Hayden Smith

(Download as PDF)

In This Lecture

- **Why? 🤔**
  - Higher level languages have many powerful methods of how we can use functions

- **What? 📰**
  - Function Syntax
  - First-class Functions
  - Higher Order Functions (HOCs)
  - Callbacks

🔠 Function Syntax

In Javascript there are three equivalent ways to define functions.

Method 1

```
1 function sum(a, b) {
2   return a + b;
3 }
```

4.1_fn_syntax_1.js

Method 2

```
1 const sum = function(a, b) {
2   return a + b;
3 };
```

4.1_fn_syntax_2.js

Method 3

```
1 const sum = (a, b) => {
2   return a + b;
3 };
```

4.1_fn_syntax_3.js

- Method 1 is the old school method. Though it's fine :)
- Method 2 and 3 are similar, treating functions like variables.
- Method 3 is more modern and has other advantages.

🔠 Function Syntax

In Javascript there are three equivalent ways to define functions.

Method 1

```ts
function sum(a: number, b: number) {
  return a + b;
}
```

4.1_fn_syntax_1.ts

Method 2

```ts
const sum = function(a: number, b: number) {
  return a + b;
};
```

4.1_fn_syntax_2.ts

Method 3

```ts
const sum = (a: number, b: number) => {
  return a + b;
};
```

4.1_fn_syntax_3.ts

- Method 1 is the old school method. Though it's fine :)
- Method 2 and 3 are similar, treating functions like variables.
- Method 3 is more modern and has other advantages.

🔠 Function Syntax

Method 3 also has another advantage. IFF (If and only if) the function body is a single line body that simply returns a value, the braces `{ }` and `return` keyword can be omitted.

Method 3

```ts
const sum = (a: number, b: number) => {
  return a + b;
};
```

4.1_fn_syntax_3.ts

Method 3 Compact

```ts
const sum = (a: number, b: number) => a + b;

// so short!!
```

4.1_fn_syntax_3_compact.ts

🔠 Function Syntax

Another example: You might have previously written this function like this:

Many String

```ts
function manyString(repeat: number, str: string) {
  let outString = '';
  for (let i = 0; i < repeat; i++) {
    outString += str;
  }
  return outString;
}
console.log(manyString(5, 'hello '));
```

4.1_many_string_1.ts

🔠 Function Syntax

But we can also write it like this

Many String

```ts
function manyString(repeat: number, str: string) {
  let outString = '';
  for (let i = 0; i < repeat; i++) {
    outString += str;
  }
  return outString;
}
console.log(manyString(5, 'hello '));
```

4.1_many_string_1.ts

Many String Update 1

```ts
const manyString = function(repeat: number, str: string) {
  let outString = '';
  for (let i = 0; i < repeat; i++) {
    outString += str;
  }
  return outString;
};
console.log(manyString(5, 'hello '));
```

4.1_many_string_2.ts

🔠 Function Syntax

Taking it a step further!

```ts
const manyString = function(repeat: number, str: string) {
  let outString = '';
  for (let i = 0; i < repeat; i++) {
    outString += str;
  }
  return outString;
};
console.log(manyString(5, 'hello '));
```

4.1_many_string_2.ts

```ts
const manyString = (repeat: number, str: string) => {
  let outString = '';
  for (let i = 0; i < repeat; i++) {
    outString += str;
  }
  return outString;
};
console.log(manyString(5, 'hello '));
```

4.1_many_string_3.ts

 First-Class Functions

Variable Definition

```
1 const name = 'Hayden';
2 console.log(name);
```

Function Definition

```
1 const getName = () => {
2   return 'Hayden';
3 };
4 console.log(getName);
```

What do these different function syntaxes teach us about functions in Javascript?

👭 First-Class Functions

Variable Definition

```
1 const name = 'Hayden';
2 console.log(name);
```

Function Definition

```
1 const getName = () => {
2   return 'Hayden';
3 };
4 console.log(getName);
```

What do these different function syntaxes teach us about functions in Javascript?

That we can treat functions similar to variables! E.G. What we're doing with the `console.log`

👭 First-Class Functions

Variable Definition

```
1 const name = 'Hayden';
2 console.log(name);
```

Function Definition

```
1 const getName = () => {
2   return 'Hayden';
3 };
4 console.log(getName);
```

What do these different function syntaxes teach us about functions in Javascript?

That we can treat functions similar to variables! E.G. What we're doing with the `console.log`

Therefore in Javascript, we say the language has **first-class functions**

👭 First Class Functions

A language has **first-class functions** when functions are treated just like any other variable.

Most notably, functions can be passed into functions just like variables can.

👭 First Class Functions

You're already used to the idea that you can write a function that produces a different output based on the variable inputted.

```
1 const sayHi = (name: string) => {
2   return `Hello ${name}!`;
3 };
4 console.log(sayHi('Hayden'));
```

4.1_fcf_var.ts

👭 First-Class Functions

So let's expand that idea and say we can write a function that produces a different output based on a **function** that is inputted. We essentially pass the function object in, and then **call** the function inside the body.

```
1  type Fmtr = (str: string) => string;
2
3  function brackets(str: string) {
4    return `(${str})`;
5  }
6
7  function fullstop(str: string) {
8    return `${str}.`;
9  }
10
11 function sayHi(name: string, format: Fmtr) {
12   return `Hello, ${format(name)}!`;
13 }
14
15 const result = sayHi('Hayden', brackets) +
16               ' -- ' +
17               sayHi('Hayden', fullstop);
18
19 console.log(result);
```

4.1_fcf_format_atomic.ts

👫 First-Class Functions

Now let's use an alternative syntax!

```ts
type Fmtr = (str: string) => string;

function brackets(str: string) {
  return `(${str})`;
}

function fullstop(str: string) {
  return `${str}.`;
}

function sayHi(name: string, format: Fmtr) {
  return `Hello, ${format(name)}!`;
}

const result = sayHi('Hayden', brackets) +
               ' -- ' +
               sayHi('Hayden', fullstop);

console.log(result);
```

4.1_fcf_format_atomic.ts

```ts
type Fmtr = (str: string) => string;

const brackets = (str: string) => `(${str})`;
const fullstop = (str: string) => `${str}.`;
const sayHi = (name: string, format: Fmtr) => `Hello, ${format(name)}

const result = sayHi('Hayden', brackets) +
               ' -- ' +
               sayHi('Hayden', fullstop);

console.log(result);
```

4.1_fcf_format_atomic_alt.ts

👭 First-Class Functions

For something more complicated: Now let's use functions as arguments, but apply it within a loop.

```ts
type Fmtr = (str: string) => string;

function brackets(str: string) {
  return `(${str})`;
}

const names = ['Hayden', 'Giuliana', 'Tam'];

function formatNames(list: string[], format: Fmtr) {
  const newList = [];
  for (const name of list) {
    newList.push(format(name));
  }
  return newList;
}

const newNames = formatNames(names, brackets);
console.log(newNames);
```

4.1_fcf_format_loop.ts

👭 First-Class Functions

Comparing with new function syntax.

```ts
type Fmtr = (str: string) => string;

function brackets(str: string) {
  return `(${str})`;
}

const names = ['Hayden', 'Giuliana', 'Tam'];

function formatNames(list: string[], format: Fmtr) {
  const newList = [];
  for (const name of list) {
    newList.push(format(name));
  }
  return newList;
}

const newNames = formatNames(names, brackets);
console.log(newNames);
```

4.1_fcf_format_loop.ts

```ts
type Fmtr = (str: string) => string;

const brackets = (str: string) => {
  return `(${str})`;
};

const names = ['Hayden', 'Giuliana', 'Tam'];

function formatNames(list: string[], format: Fmtr) {
  const newList: string[] = [];
  for (const name of list) {
    newList.push(format(name));
  }
  return newList;
}

const newNames = formatNames(names, brackets);
console.log(newNames);
```

4.1_fcf_format_loop_new.ts

👭 First-Class Functions

Anonymous Functions

If we only intend to use a function once, we can pass it directly into a function. Because this function doesn't have a **name**, we call it an anonymous function.

```ts
type Fmtr = (str: string) => string;

const brackets = (str: string) => {
  return `(${str})`;
};

const names = ['Hayden', 'Giuliana', 'Tam'];

function formatNames(list: string[], format: Fmtr) {
  const newList: string[] = [];
  for (const name of list) {
    newList.push(format(name));
  }
  return newList;
}

const newNames = formatNames(names, brackets);
console.log(newNames);
```

4.1_fcf_format_loop_new.ts

```ts
type Fmtr = (str: string) => string;

const names = ['Hayden', 'Giuliana', 'Tam'];

function formatNames(list: string[], format: Fmtr) {
  const newList = [];
  for (const name of list) {
    newList.push(format(name));
  }
  return newList;
}

const newNames = formatNames(names, (str) => {
  return `(${str})`;
});

console.log(newNames);
```

4.1_fcf_format_loop_anon.ts

👭 First-Class Functions

In Summary: What?

First-class functions are predominately used in terms of letting **functions take in other functions as arguments**

In Summary: Why?

Allows us to create more concise and clear code. In particular, the use of anonymous functions for one-off usage make code both more compact and more readable. It's also the fundamental part of understanding callbacks.

👭 First-Class Functions

In Summary: What?

First-class functions are predominately used in terms of letting **functions take in other functions as arguments**

In Summary: Why?

Allows us to create more concise and clear code. In particular, the use of anonymous functions for one-off usage make code both more compact and more readable. It's also the fundamental part of understanding callbacks.

Wait, what the %@# is a callback!?!?

🧪 Map, Reduce, Filter

Map, reduce, filter are functions that act on array objects that help us accomplish basic iterative tasks without the overhead of a loop setup. They are based on ideas of **first-class functions** and **anonymous functions**

- **Map**: Modify an array
- **Filter**: Select from an array
- **Reduce**: Summarise an array

🧪 Map, Reduce, Filter

Map

Takes an array of size N, and produces a new array of size N having modified each element according to a function passed in.

Same array size. Modified elements.

🧪 Map, Reduce, Filter

Map

Classic

```ts
const tutors = [
  'Simon',
  'Teresa',
  'Kaiqi',
  'Michelle',
];

const shout = function(str: string) {
  return `${str.toUpperCase()}!!!`;
};

const newList = [];
for (const tutor of tutors) {
  const newTutor = shout(tutor);
  newList.push(newTutor);
}
console.log(newList);
```

4.1_map_old.ts

Modern

```ts
const tutors = [
  'Simon',
  'Teresa',
  'Kaiqi',
  'Michelle',
];

const shout = function(str: string) {
  return `${str.toUpperCase()}!!!`;
};

const newList = tutors.map(shout);
console.log(newList);
```

4.1_map.ts

Same array size. Modified elements.

🧪 Map, Reduce, Filter

Map

Classic

```ts
const tutors = [
  'Simon',
  'Teresa',
  'Kaiqi',
  'Michelle',
];

const shout = function(str: string) {
  return `${str.toUpperCase()}!!!`;
};

const newList = [];
for (const tutor of tutors) {
  const newTutor = shout(tutor);
  newList.push(newTutor);
}
console.log(newList);
```

4.1_map_old.ts

Modern

```ts
const tutors = [
  'Simon',
  'Teresa',
  'Kaiqi',
  'Michelle',
];

const shout = function(str: string) {
  return `${str.toUpperCase()}!!!`;
};

const newList = tutors.map(shout);
console.log(newList);
```

4.1_map.ts

Same array size. Modified elements.

🧪 Map, Reduce, Filter

Filter

Takes an array of size N, and produces a new array of size 0..N without modifying any of the data.

Possibly smaller array size. Elements unchanged.

🧪 Filter, Reduce, Filter

Filter

Classic

```ts
const marks = [65, 72, 81, 40, 56];

const isPass = function(mark: number) {
  return mark >= 50;
};

const newList = [];
for (const mark of marks) {
  if (isPass(mark)) {
    newList.push(mark);
  }
}
console.log(newList);
```

4.1_filter_old.ts

Modern

```ts
const marks = [65, 72, 81, 40, 56];

const isPass = function(mark: number) {
  return mark >= 50;
};

const newList = marks.filter(isPass);
console.log(newList);
```

4.1_filter.ts

Possibly smaller array size. Elements unchanged.

🧪 Filter, Reduce, Filter

Filter

Classic

```ts
const marks = [65, 72, 81, 40, 56];

const isPass = function(mark: number) {
  return mark >= 50;
};

const newList = [];
for (const mark of marks) {
  if (isPass(mark)) {
    newList.push(mark);
  }
}
console.log(newList);
```

4.1_filter_old.ts

Modern

```ts
const marks = [65, 72, 81, 40, 56];

const isPass = function(mark: number) {
  return mark >= 50;
};

const newList = marks.filter(isPass);
console.log(newList);
```

4.1_filter.ts

Possibly smaller array size. Elements unchanged.

🧪 Map, Reduce, Filter

Reduce

Executes a reducer function (that you provide) on each member of the array resulting in a single output value.

Array turned into atomic value.

🧪 Filter, Reduce, Filter

Reduce

Classic

```ts
const students = [
  { name: 'Amy', mark: 55 },
  { name: 'Bob', mark: 43 },
  { name: 'Cap', mark: 34 },
  { name: 'Dex', mark: 23 },
];

let single = 0;
for (const student of students) {
  single += student.mark;
}
console.log(single);
```

4.1_reduce_old.ts

Modern

```ts
type Student = {
  name: string;
  mark: number;
}

const students: Student[] = [
  { name: 'Amy', mark: 55 },
  { name: 'Bob', mark: 43 },
  { name: 'Cap', mark: 34 },
  { name: 'Dex', mark: 23 },
];

const sum = (prev: number, curr: Student) => {
  return prev + curr.mark;
};

const single = students.reduce(sum, 0);
console.log(single);
```

4.1_reduce.ts

Array turned into atomic value.

🧪 Map, Reduce, Filter

Anonymous Functions

We can use anonymous functions to further streamline our previous code.

```
1 const tutors = ['Simon', 'Teresa', 'Kaiqi', 'Michelle'];
2 const newList = tutors.map((string) => `${string.toUpperCase()}!!!`);
3 console.log(newList);
```

4.1_map_stream.ts

```
1 const marks = [65, 72, 81, 40, 56];
2 const newList = marks.filter((mark) => mark >= 50);
3 console.log(newList);
```

4.1_filter_stream.ts

```
1 const students = [
2   { name: 'Amy', mark: 55 },
3   { name: 'Bob', mark: 43 },
4   { name: 'Cap', mark: 34 },
5   { name: 'Dex', mark: 23 },
6 ];
7 const single = students.reduce((prev, curr) => prev + curr.mark, 0);
8 console.log(single);
```

4.1_reduce_stream.ts

🧪 Map, Reduce, Filter Combined

We can combine these methods to create some very clean code.

Print the average mark of those student's who passed the 60 mark exam. Express the average as a %.

```
1 const marks = [39, 43.2, 48.6, 24, 33.6];
2 const normalisedMarks = marks.map(m => 100 * m / 60);
3 const passingMarks = normalisedMarks.filter(m => m >= 50);
4 const total = passingMarks.reduce((a, b) => a + b, 0);
5 const average = total / passingMarks.length;
6 console.log(average);
```

4.1_mapfilterreduce.ts

🙏🏾 Higher Order Functions

Higher-order Functions are essentially functions that return functions. Think of them like mini function factories.

Let's go through a code cleanup journey.

```
1  function congratMarkPS(name: string) {
2    return `Congratulations ${name} on your pass`;
3  }
4  function congratMarkCR(name: string) {
5    return `Congratulations ${name} on your credit`;
6  }
7  function congratMarkDN(name: string) {
8    return `Congratulations ${name} on your distinction`;
9  }
10 console.log(congratMarkCR('Hayden'));
```

4.1_hoc_1.ts

🙏 Higher Order Functions

```ts
function congratMarkPS(name: string) {
  return `Congratulations ${name} on your pass`;
}
function congratMarkCR(name: string) {
  return `Congratulations ${name} on your credit`;
}
function congratMarkDN(name: string) {
  return `Congratulations ${name} on your distinction`;
}
console.log(congratMarkCR('Hayden'));
```

4.1_hoc_1.ts

```ts
function congratWrapper(markstr: string, name: string) {
  return `Congratulations ${name} on your ${markstr}`;
}
function congratMarkPS(name: string) {
  return congratWrapper('pass', name);
}
function congratMarkCR(name: string) {
  return congratWrapper('credit', name);
}
function congratMarkDN(name: string) {
  return congratWrapper('distinction', name);
}
console.log(congratMarkCR('Hayden'));
```

4.1_hoc_2.ts

🙏🏾 Higher Order Functions

```ts
function congratWrapper(markstr: string, name: string) {
  return `Congratulations ${name} on your ${markstr}`;
}
function congratMarkPS(name: string) {
  return congratWrapper('pass', name);
}
function congratMarkCR(name: string) {
  return congratWrapper('credit', name);
}
function congratMarkDN(name: string) {
  return congratWrapper('distinction', name);
}
console.log(congratMarkCR('Hayden'));
```

4.1_hoc_2.ts

```ts
const congratWrapper = (markstr: string, name: string) => {
  return `Congratulations ${name} on your ${markstr}`;
};
const congratMarkPS = (name: string) => {
  return congratWrapper('pass', name);
};
const congratMarkCR = (name: string) => {
  return congratWrapper('credit', name);
};
const congratMarkDN = (name: string) => {
  return congratWrapper('distinction', name);
};
console.log(congratMarkCR('Hayden'));
```

4.1_hoc_3.ts

🙏🏾 Higher Order Functions

```ts
const congratWrapper = (markstr: string, name: string) => {
  return `Congratulations ${name} on your ${markstr}`;
};
const congratMarkPS = (name: string) => {
  return congratWrapper('pass', name);
};
const congratMarkCR = (name: string) => {
  return congratWrapper('credit', name);
};
const congratMarkDN = (name: string) => {
  return congratWrapper('distinction', name);
};
console.log(congratMarkCR('Hayden'));
```

4.1_hoc_3.ts

```ts
function genCongratMark(markstr: string) {
  const ret = function(name: string) {
    return `Congratulations ${name} on your ${markstr}`;
  };
  return ret;
}
const congratMarkPS = genCongratMark('pass');
const congratMarkCR = genCongratMark('credit');
const congratMarkDN = genCongratMark('distinction');

console.log(congratMarkCR('Hayden'));
```

4.1_hoc_4.ts

🙏 Higher Order Functions

```
function genCongratMark(markstr: string) {
  const ret = function(name: string) {
    return `Congratulations ${name} on your ${markstr}`;
  };
  return ret;
}
const congratMarkPS = genCongratMark('pass');
const congratMarkCR = genCongratMark('credit');
const congratMarkDN = genCongratMark('distinction');

console.log(congratMarkCR('Hayden'));
```

4.1_hoc_4.ts

```
const genCongratMark = (markstr: string) => {
  const ret = (name: string) => {
    return `Congratulations ${name} on your ${markstr}`;
  };
  return ret;
};

const congratMarkPS = genCongratMark('pass');
const congratMarkCR = genCongratMark('credit');
const congratMarkDN = genCongratMark('distinction');

console.log(congratMarkCR('Hayden'));
```

4.1_hoc_5.ts

🙏🏾 Higher Order Functions

```ts
const genCongratMark = (markstr: string) => {
  const ret = (name: string) => {
    return `Congratulations ${name} on your ${markstr}`;
  };
  return ret;
};

const congratMarkPS = genCongratMark('pass');
const congratMarkCR = genCongratMark('credit');
const congratMarkDN = genCongratMark('distinction');

console.log(congratMarkCR('Hayden'));
```

4.1_hoc_5.ts

```ts
const genCongratMark = (markstr: string) => {
  return (name: string) => {
    return `Congratulations ${name} on your ${markstr}`;
  };
};

const congratMarkPS = genCongratMark('pass');
const congratMarkCR = genCongratMark('credit');
const congratMarkDN = genCongratMark('distinction');

console.log(congratMarkCR('Hayden'));
```

4.1_hoc_6.ts

🙏🏾 Higher Order Functions

```ts
const genCongratMark = (markstr: string) => {
  return (name: string) => {
    return `Congratulations ${name} on your ${markstr}`;
  };
};

const congratMarkPS = genCongratMark('pass');
const congratMarkCR = genCongratMark('credit');
const congratMarkDN = genCongratMark('distinction');

console.log(congratMarkCR('Hayden'));
```

4.1_hoc_6.ts

```ts
const genCongratMark = (markstr: string) => (name: string) =>
  `Congratulations ${name} on your ${markstr}`;

const congratMarkPS = genCongratMark('pass');
const congratMarkCR = genCongratMark('credit');
const congratMarkDN = genCongratMark('distinction');

console.log(congratMarkCR('Hayden'));
```

4.1_hoc_7.ts

🙏🏾 Higher Order Functions

Summary

Higher order functions are syntactically sleek ways to generalise function definitions.

👂 Feedback

Or go to the form here.

COMP1531

🌏 Projects - Package Management

Lecture 2.1

Author(s): Hayden Smith

(Download as PDF)

In This Lecture

- **Why? 🤔**
  - To utilise javascript fully, we need to understand how to install other modules that aren't on our system
  - We need to know how to manage our installations on multi-user projects

- **What? 📰**
  - Problems with packages
  - NPM and how it works
  - How to manage packages
  - Custom scripts

🌳 Disclaimer: Environment Change

Beginning from lecture 2.1, we will be working inside the `env1` folder with the lecture code. To "run" code from lectures slides further on you will need to ensure you have a similar environment.

Don't stress, though! For your labs in week 3 + iteration 1 we have setup your project to contain everything you need.

😳 Problems That We Face

Sometimes we might want to use a library in NodeJS, but this library wasn't built-in by us. Someone else on the internet wrote it.

An example of this might be you googling "javascript how do I check if a date is valid" and coming across some code. And we find this snippet that looks good... so we try it out!

```
1 import { isValid } from 'date-fns';
2
3 console.log(isValid(new Date('2021, 02, 30')));
```

😳 Problems That We Face

So now we turn it into a function that looks good for our purposes.

```
1 import { isValid } from 'date-fns';
2
3 function dateIsValid(year, month, day) {
4   return isValid(new Date(year, month, day));
5 }
6
7 console.log(dateIsValid('2022', '14', '02'));
```

😳 Problems That We Face

So now we turn it into a function that looks good for our purposes.

```
1 import { isValid } from 'date-fns';
2
3 function dateIsValid(year, month, day) {
4   return isValid(new Date(year, month, day));
5 }
6
7 console.log(dateIsValid('2022', '14', '02'));
```

But when we run it we get this error...

😳 Problems That We Face

So now we turn it into a function that looks good for our purposes.

```
1 import { isValid } from 'date-fns';
2
3 function dateIsValid(year, month, day) {
4   return isValid(new Date(year, month, day));
5 }
6
7 console.log(dateIsValid('2022', '14', '02'));
```

But when we run it we get this error...

```
1 SyntaxError: Cannot use import statement outside a module'
```

😳 Problems That We Face

So now we turn it into a function that looks good for our purposes.

```
1 import { isValid } from 'date-fns';
2
3 function dateIsValid(year, month, day) {
4   return isValid(new Date(year, month, day));
5 }
6
7 console.log(dateIsValid('2022', '14', '02'));
```

But when we run it we get this error...

```
1 SyntaxError: Cannot use import statement outside a module'
```

We need **tools** to solve this...

🧩 NPM: Node Package Manager

NPM (Node Package Manager) is a tool that is automatically installed alongside NodeJS to manage dependencies/modules/libraries (all the same thing) for NodeJS (Javascript) projects.

It's command on terminal is:

`npm`

NPM: Node Package Manager

The most common usage of NPM is to allow you to download external libraries.

The external libraries that you're able to download with NPM are found on the npmjs website.

Let's have a look for our `date-fns` library!

NPM: Node Package Manager

To setup a code repository to use `npm`, all we need to do is run `npm init` (if it hasn't already been run). It will ask you a few questions (don't stress about getting them right, you can change them later).

Once this is done you should now see a `package.json` in your repository. This is essentailly your projects NPM configuration file.

```
1  {
2    "name": "example",
3    "version": "1.0.0",
4    "description": "",
5    "type": "module",
6    "main": "index.js",
7    "scripts": {
8      "test": "echo \"Error: no test specified\" && exit 1"
9    },
10   "author": "",
11   "license": "ISC"
12 }
```

2.1_package_simple.json

Sometimes we need to configure further. For example for 1531 reasons we added `"type": "module"` too.

Managing Packages

We can install dependencies with

```
npm install [dependency]
```

For example:

```
npm install date-fns
```

You will see that this command automatically adds the most recent stable version of `date-fns` to our `package.json`.

Let's inspect `package.json`.

Take note of the ~ and ^ symbols.

Managing Packages

Our code will now run successfully.

```
1 import { isValid } from 'date-fns';
2
3 function dateIsValid(year, month, day) {
4   return isValid(new Date(year, month, day));
5 }
6
7 console.log(dateIsValid('2022', '14', '02'));
```

2.1_date_fns.js

Anatomy Of NPM

The structure of NPM involves a few key files and folders.

package.json

Where we store meta data about our project including a list of dependencies to install

package-lock.json

Where we store versioning information about dependencies to ensure everyone has the right versions (oversimplification)

node_modules/

Where the dependencies are installed locally.

 Anatomy Of NPM

The structure of NPM involves a few key files and folders.

package.json

Changes are always commited to git.

package-lock.json

Changes are always commited to git.

node_modules/

Changes are never commited to git.

 Anatomy Of NPM

By committing this information we ensure that our environment is completely reproducable on others' machines + our own.

If you fresh clone the repo, the `node_modules` folder won't exist which means the dependencies won't work.

However, if you run `npm install` it will read the `package.json` and `package-lock.json` file to install the appropriate dependencies.

📦 Custom Scripts

A useful feature of `npm` that we will explore in another lecture is the ability to add **scripts** to the `package.json`.

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
};
```

📦 Custom Scripts

We can add our own command and run it with:

`npm run hayden`.

```
1 "scripts": {
2   "test": "echo \"Error: no test specified\" && exit 1",
3   "hayden": "echo 'Hi Hayden!'"
4 };
```

Scripts use a mixture of `bash` and `json`. We will cover these more later in the course.

For now we'll tell you exactly how to modify this stuff.

👂 Feedback

Or go to the form here.

More NPM

Presented by
Tom Isles

Canva

```
rt_tree = require('../util/convert-tree
til = require('../util/time-util.js');
nst = require('../util/app-const.js');

exports.getRender = f(req, res){
  if(req.query.rootId){
    res.render('tree', {
      title: req.i18n.__("gnr.title"),
      rootId: req.query.rootId,
      currentLocale: req.i18n.getLocale()
    });
  } else {
    res.render('tree', {
      title: req.i18n.__("gnr.title"),
      rootId: null,
      currentLocale: req.i18n.getLocale()
    });
  }
};
```

Recap

- NPM is a package management system for javascript applications. It allows you to install dependencies / libraries from the web.
- You can install new dependencies using *npm install*, start applications using *npm start*, and so on.
- *package.json* is a file that contains metadata about your application and exists in the root of your directory.
- Yarn is an alternative to NPM that was released by facebook and has a few advantages.

package.json

the package.json construct is tied to NPM, or more specifically, to **npmjs.org**, which is where javascript packages are hosted.

Many fields are related to the publication of packages.

Other fields relate to dependency management, browser support, and custom scripts.

For the purposes of this lecture we will talk about NPM, but these can be used interchangeably.

```
{
  "name": "test-app",
  "author": "Tom Isles <my_email@example.com> (https://example.com)",
  "contributors": [
    "Other Contributor <other@example.com> (https://other.com)"
  ],
  "bugs": "https://github.com/repository/package/issues",
  "homepage": "example.com/test-app",
  "version": "0.1.0",
  "license": "MIT",
  "keywords": [
    "demo",
    "unsw"
  ],
  "description": "A package to demonstrate proper package.json",
  "repository": "github:example/example",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^4.2.4",
    "@testing-library/react": "^9.3.2",
    "@testing-library/user-event": "^7.1.2",
    "react": "^16.13.1",
    "react-dom": "^16.13.1",
    "react-scripts": "3.4.3"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": "react-app"
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

package.json

Many fields in package.json are not related to your application per se, but define metadata related to it.

If you publish a package on *npmjs.org,* fields like *name, author, contributors, bugs, homepage, license, keywords, description* and *repository* are all used to generate your package's page.

Example Package:

https://www.npmjs.com/package/react

License is a special case here. While it is displayed on the NPM site it also determines how your code is licensed and has legal ramifications depending on the license.

```
{
  "name": "test-app",
  "author": "Tom Isles <my_email@example.com>
(https://example.com)",
  "contributors": [
    "Other Contributor <other@example.com> (https://other.com)"
  ],
  "bugs": "https://github.com/repository/package/issues",
  "homepage": "example.com/test-app",
  "license": "MIT",
  "keywords": [
    "demo",
    "unsw"
  ],
  "description": "A package to demonstrate proper
package.json",
  "repository": "github:example/example",
```

package.json

***version* is a specification of your package's current version. If you make a change to a hosted package you should also change the version.**

Versions are used by other developers who wish to depend on your package.

Versions use semantic versioning*:

MAJOR.MINOR.PATCH, where:

A change in MAJOR version means you changed the API in a breaking way.

A change in MINOR version means you added functionality in a backwards compatible manner.

A change in PATCH version means you made backwards compatible bug fixes.

**semver.org*

```
"version": "0.1.0",
```

package.json

***private* prevents accidental publication of packages.** If you have a package that you never wish to upload to NPM, then private should be set to true.

```
"private": true,
```

package.json

Dependencies list the packages that your package relies on.

When specifying a dependency, we also specify a corresponding *version*. Like in our version field, we use semantic versioning to specify which version of the package we require.

Dependency versions do not have to be straight versions, they can include ***semver ranges***.

Some examples:

>1.2.3 Any package with a more recent version than 1.2.3, including major releases.

^1.2.3 Any version of the package with a minor version greater than 2 and a patch version greater than 3. Cannot exceed major version **1**. Get most updated major version of package.

~ 1.2.3 Any version with a patch version greater than 3. Cannot exceed major version **1** or minor version **2.** Get most updated minor version of package.

https://nodesource.com/blog/semver-a-primer/

```
"dependencies": {
  "@testing-library/jest-dom": "^4.2.4",
  "@testing-library/react": "^9.3.2",
  "@testing-library/user-event": "^7.1.2",
  "react": "^16.13.1",
  "react-dom": "^16.13.1",
  "react-scripts": "3.4.3"
},
```

package.json

package.json has numerous dependency types:

- dependencies
- devDependencies
- peerDependencies
- optionalDependencies

```
"dependencies": {
  "@testing-library/jest-dom": "^4.2.4",
  "@testing-library/react": "^9.3.2",
  "@testing-library/user-event": "^7.1.2",
  "react": "^16.13.1",
  "react-dom": "^16.13.1",
  "react-scripts": "3.4.3"
},
```

Dependency Types

dependencies
Modules required by the application during runtime.
Common examples: *react*

devDependencies
Modules required during development of the application.
Common examples: *babel, eslint, jest (testing library)*

Dependency Types

peerDependencies
If a package has a peer dependency, any application that consumes it must also install the peer dependency. The peer dependency won't be installed transitively.

An example of this is *react-dom*. r*eact* lists it as a peer dependency and so react applications must depend both on *react* and *react-dom.*

optionalDependencies
Any dependency where, if it fails to install, npm / yarn will say installation was still successful.

package.json

scripts allows you to specify command line scripts to run with npm.

You have already seen commands like *npm install*. NPM and Yarn both have a set of reserved keywords which link to inbuilt commands - like *install* - but any other keyword is fair game.

The scripts object is a simple key value object. The key is your command, which you run with npm, ie *npm <key>*. The value is the command that will be run when you call out to it. The command should be written in a syntax parseable by your default shell.

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
```

Demo

package.json

package.json is a JSON file, so it can be arbitrarily extended.

In this case, an *eslintConfig* field has been added. It's not part of a standard package.json specification, but is used by eslint to configure linting for your application.

```
"eslintConfig": {
  "extends": "react-app"
},
```

package.json

browserslist is another example of an option that is referenced by certain tools and is specified by the browserslist plugin (npmjs.com/package/browserslist).

It specifies what browser types are supported by the application.

```
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

Yarn vs NPM

What's the difference between Yarn and NPM?

Yarn was created at Facebook to solve their problems with NPM at scale. At time of release, Yarn was:

- Significantly faster.
- Used a deterministic install algorithm. At the time, NPM could install dependencies in a different order leading to different *node_modules* folder structures given the same *package.json* between developers.
- Used lockfiles.

Performance

Performance Improvements

In Yarn, packages that are not dependent on each other are installed in separate threads, meaning install times are cut down compared to NPM.

NPM version 5 achieved significant performance improvements which have closed the performance gap between the two tools

Lockfiles

When specifying dependencies in *package.json,* version ranges can be specified using the semver syntax (^, >, *, etc).

This means that different developers developing the same application can end up using different versions of their packages. This leads to 'works on my computer' problems.

A **lockfile** records the exact versions of each dependency that has been installed.

Lockfiles

We commit a **lockfile** into version control and this ensures that the next time the repository is checked out and installed, the same versions are installed.

In yarn, **yarn.lock** is the lockfile and is generated each time dependencies are installed.

In NPM, the **npm shrinkwrap** command creates an **npm-shrinkwrap.json** file which performs the same function.

Good Work!