Lecture 3

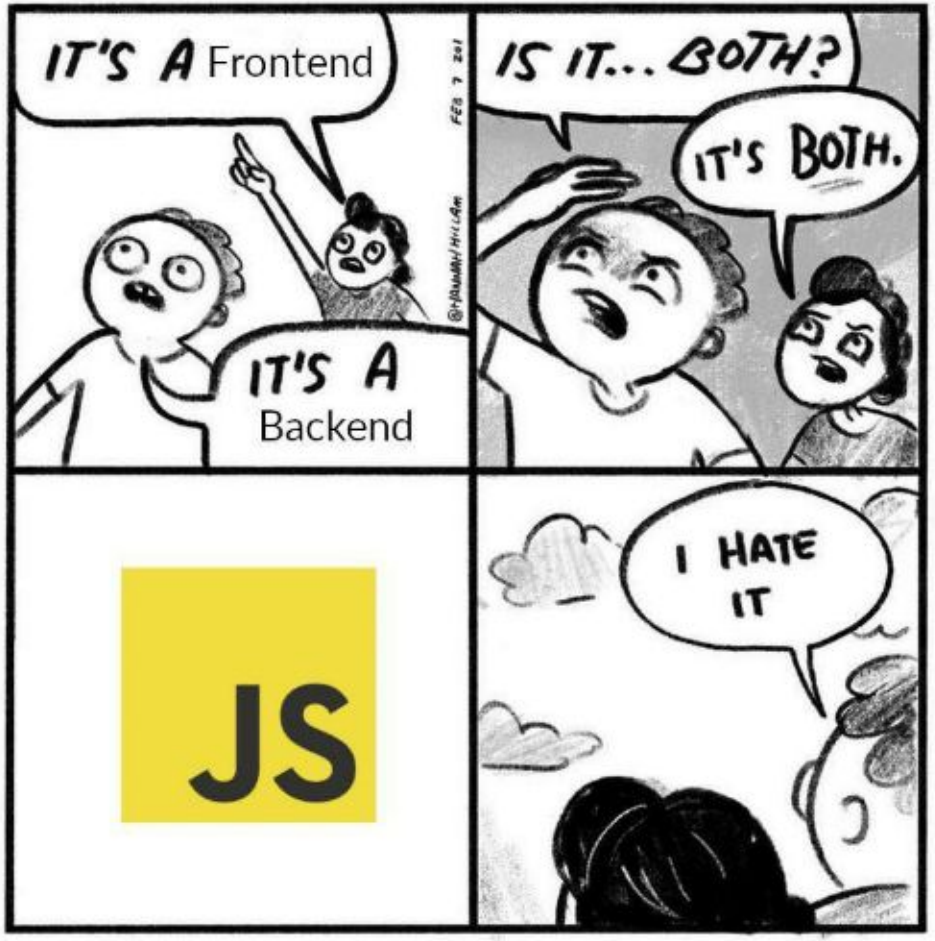
JavaScript Ecosystem + TypeScript

Frontend Web Development

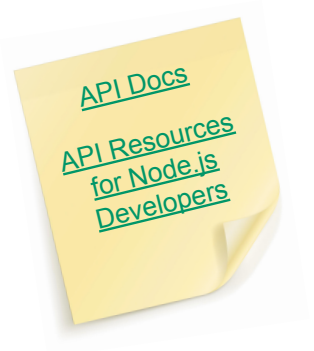
Today's plan

- NodeJS and npm
- Transpilation & Babel
- Bundlers (Webpack, Rollup, Snowpack, Vite)
- Tree Shaking and Minification (code optimizations)
- TypeScript

NodeJS



System APIs



command line utils

HTTP

files

net

path

cluster

OS

UDP

process

HTTPS

crypto

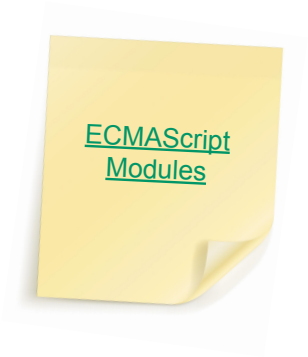
timers

URL

TLS/SSL

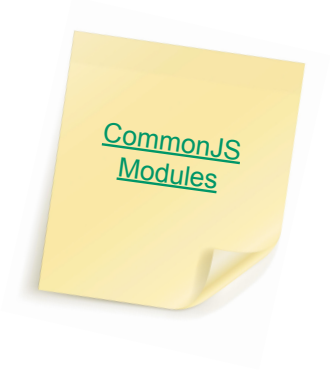
console

ECMAScript Modules (.mjs files)




```
// circle.js
const { PI } = Math;
export const area = (r) => PI * r ** 2;
export const circumference = (r) => 2 * PI * r;

// index.js
import { area } from './circle.js'
console.log(`The area of a circle of radius 4 is ${area(4)}`);
```

CommonJS Modules (.js files)




```
// circle.js
const { PI } = Math;
module.exports = {
    area: (r) => PI * r ** 2,
    circumference: (r) => 2 * PI * r
};

// index.js
const circle = require('./circle.js');
console.log(`The area of a circle of radius 4 is ${circle.area(4)}`);
```

Historical sidenote: Why "*ECMA*Script"?

- **E**uropean **C**omputer **M**anufacturers **A**ssociation
- Purpose: Standardization of ICT systems (like ISO)
- JS naming history: Mocha → LiveScript → JavaScript → ECMAScript
- JS = ES + DOM + BOM
- Very irrelevant. Just use **JavaScript** :)

https://stackoverflow.com/a/30113184

npm

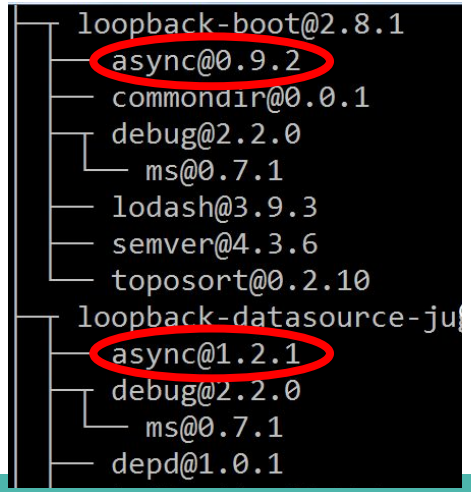
1. Node Package Manager (or **npm**) is the standard package manager of the Node.js ecosystem and a command-line interface tool used by developers to manage their Node.js projects
2. npm registry is the most extensive online package repository, containing over one-million packages

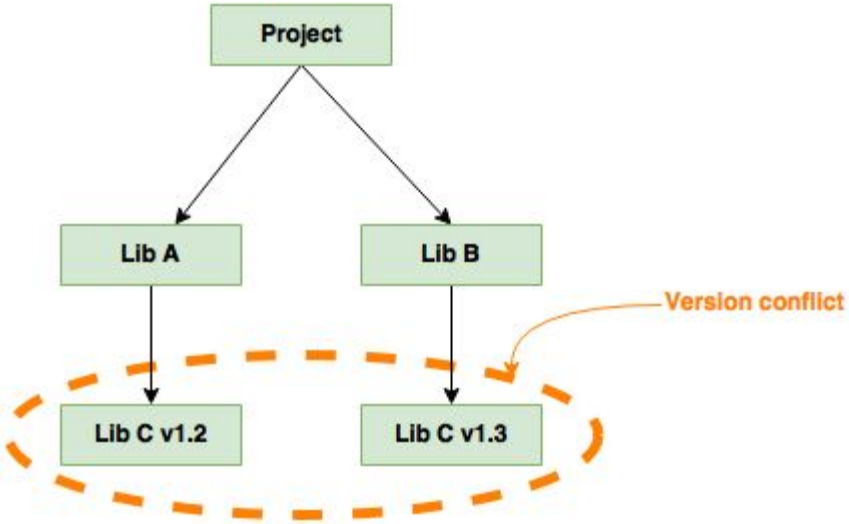
Semantic Versioning (semver)

- A “pinky swear” among developers about updates to the package
- Consists of 3 numbers:
  - Major version: breaking changes
  - Minor version: adding new features
  - Patch version: fixing a bug
- Separated by dots: `X.Y.Z`
- Example: `lodash: 4.17.21`

Dependency management

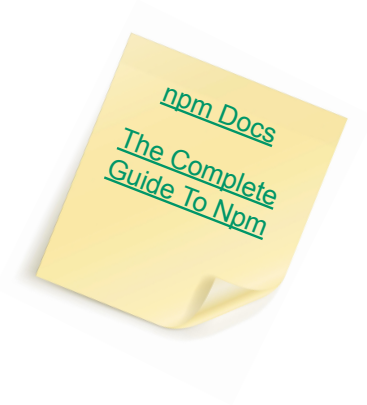
- Traditional package managers (e.g.: C++, Java, pip):
  - Only one version of a package at a time
  - Problematic for transitive dependencies with different versions
- Node:
  - Nesting (isolating) dependencies





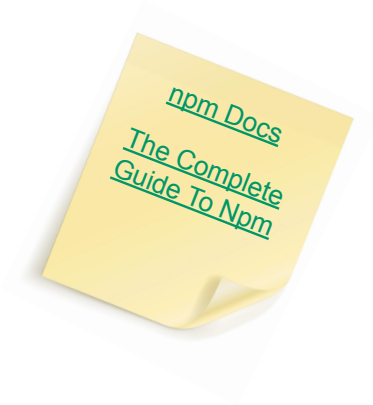
npm commands examples

```
# Initialize a project
npm init

# Install a package
npm install <package>

# Remove a package
npm uninstall <package>

# Update all packages
npm update

# Run any script
npm run <script>
```



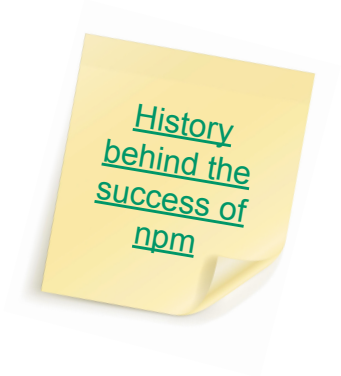
package.json

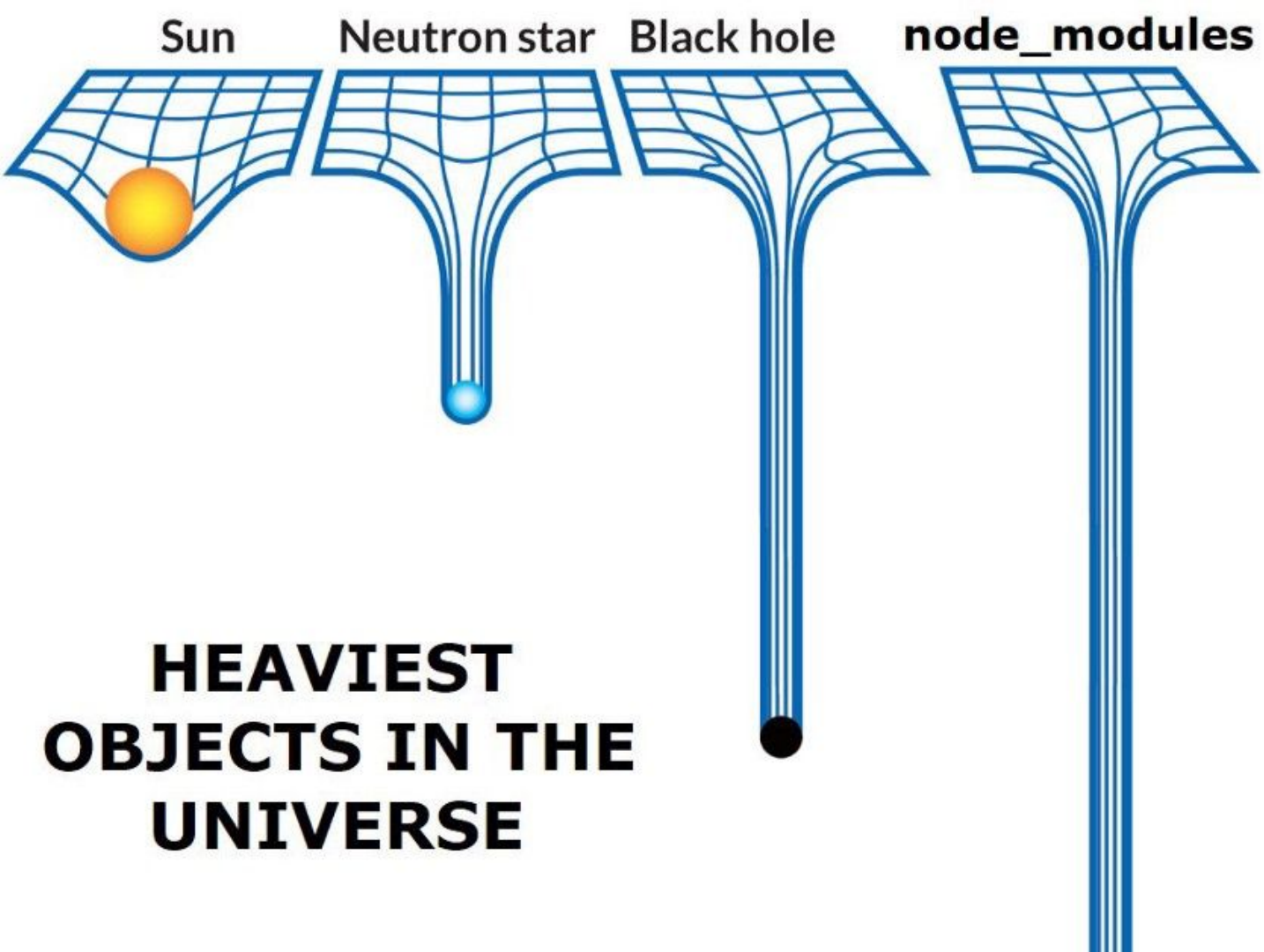
```
{
 "name": "react",
 "description": "React is a JavaScript library for building user interfaces.",
 "version": "17.0.3",
 "homepage": "https://reactjs.org/",
 "license": "MIT",
 "main": "index.js",
 "dependencies": { ... },
 "devDependencies": { ... }
}
```



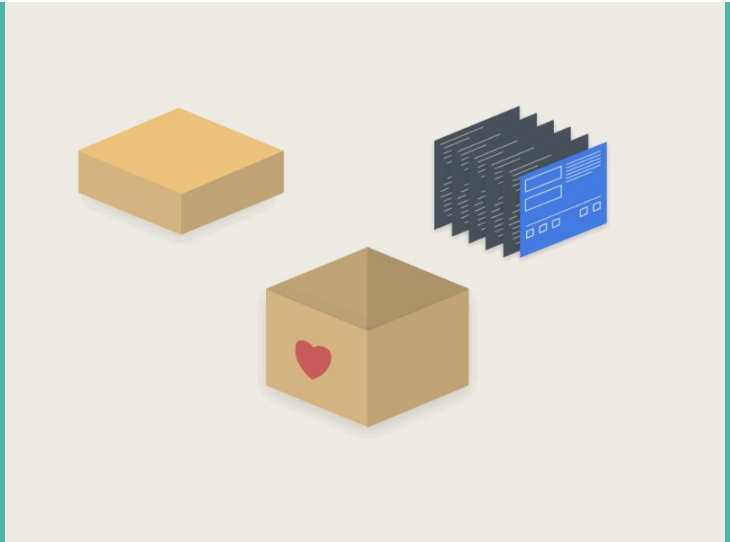
Node/npm on the frontend

- Same technology can be useful in frontend development as well
  - Publishing/importing reusable code
- Browsers don’t support it? No problem!
  - We’re in control over the development process
  - Can simply generate stuff the browser understands
- Everything is already JavaScript!




Sun
Neutron star
Black hole
node_modules
HEAVIEST OBJECTS IN THE UNIVERSE

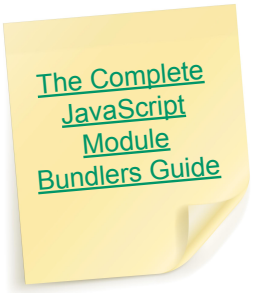
Bundlers



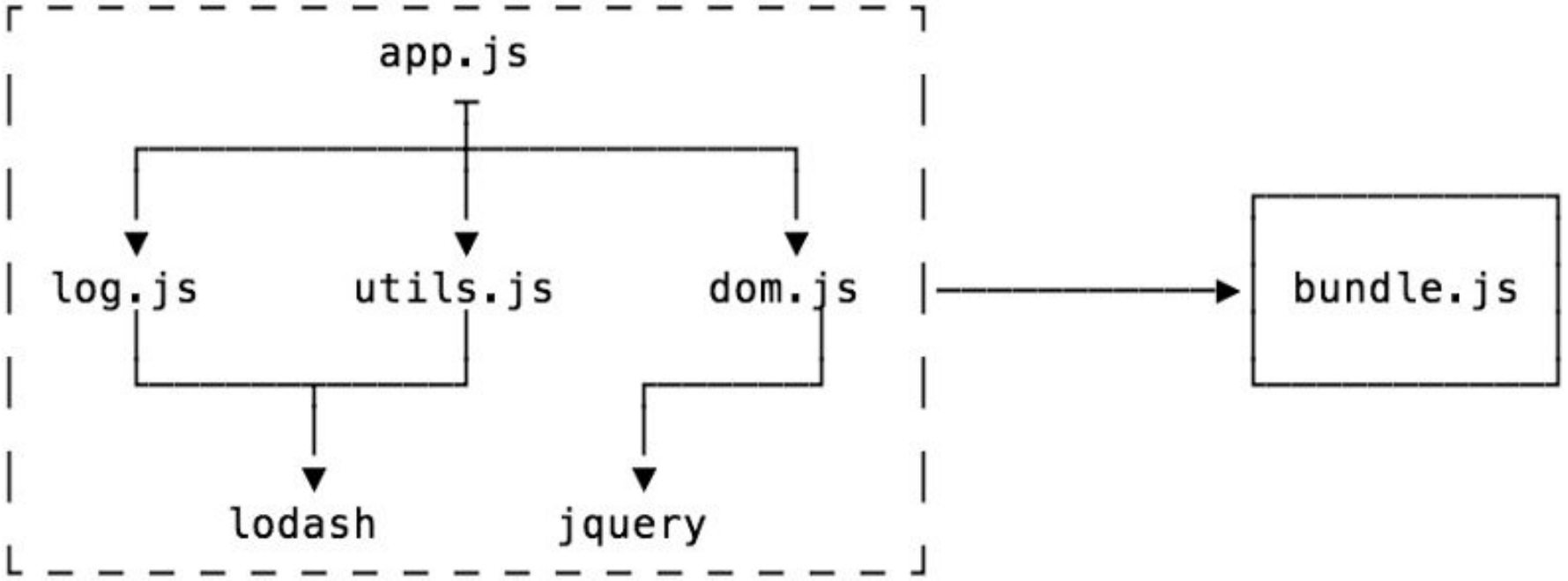
Bundler

A bundler is a development tool that combines many JavaScript code files into a single one that is production-ready loadable in the browser.

It generates a dependency graph as it traverses your first code files, and thus it keeps track of both your source files' dependencies and third-party dependencies.



Bundler



Why do we need it?

- Combine modules together in one production-ready file
- Use imports to manage dependencies
  - rather than order of `<script>`s
- Create more complicated build pipelines
  - e.g.: replace some string in the code with an environment variable
- Import other types of files (images, CSS, ...)
- Code splitting
- Output code in multiple different formats (ESM, CJS, ...)
  - Or standards (ES5, ES6, ES2022, ...)

Some bundlers

Top 5 JavaScript Module Bundlers for 2021

webpack

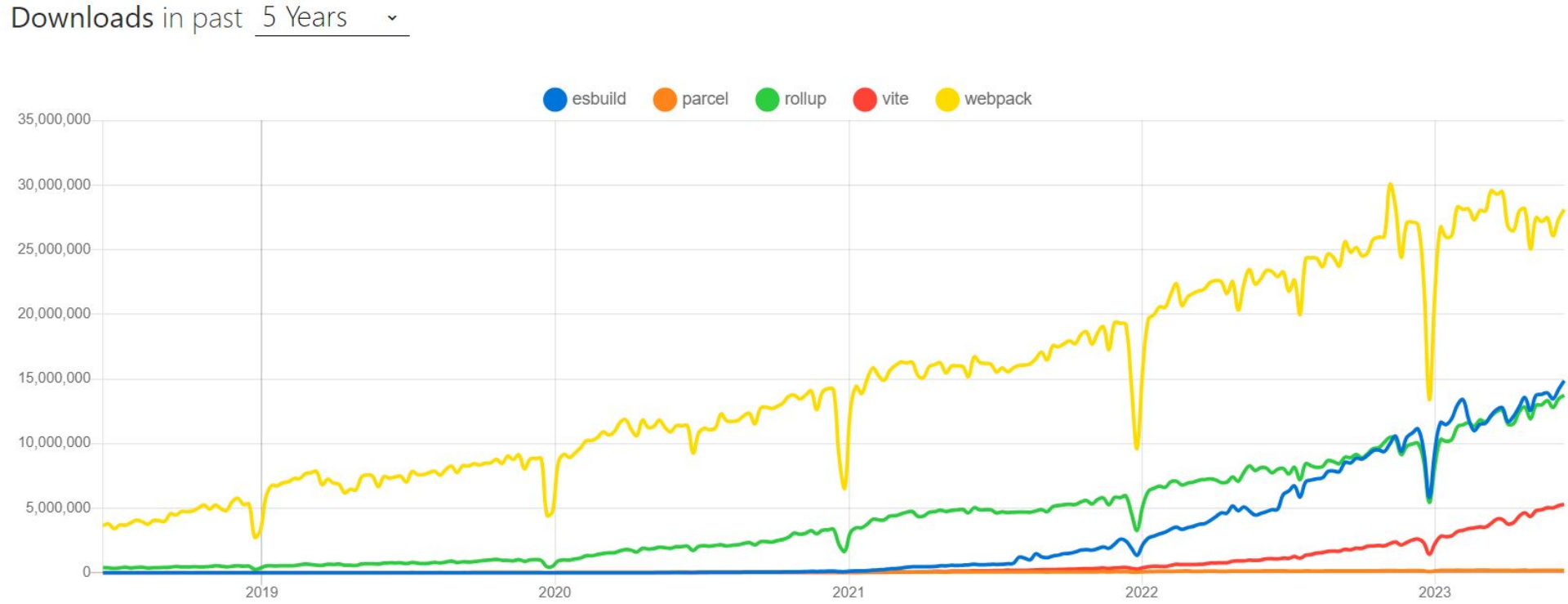
browserify

rollup.js

Snowpack

PARCEL

Comparison


Downloads in past 5 Years
esbuild
parcel
rollup
vite
webpack
35,000,000
30,000,000
25,000,000
20,000,000
15,000,000
10,000,000
5,000,000
0
2019
2020
2021
2022
2023

Tree Shaking

Dead Code Elimination

Removal of code that's never going to run no matter what you do.

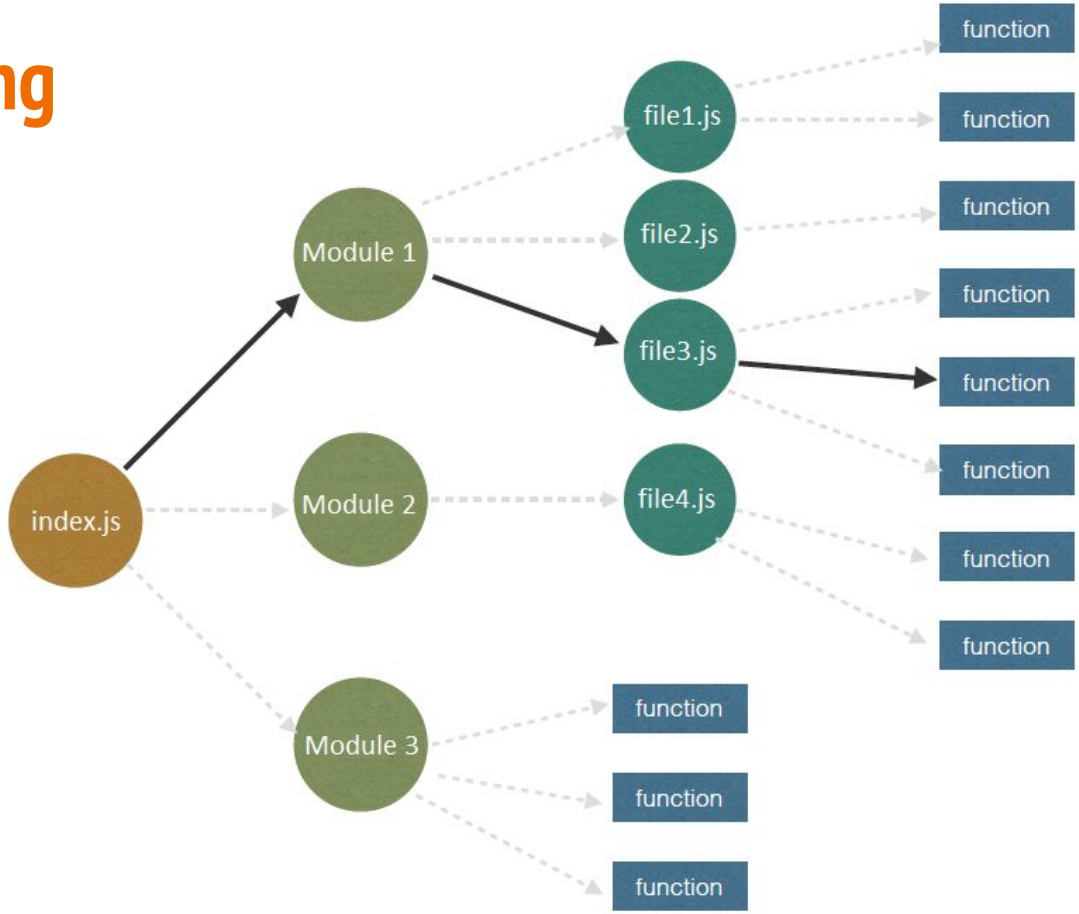
```
function answer() {
    return 42;
    console.log('Found it!');
}

if (false) { // for debugging
    console.log('Finished calculation');
}
```

Tree Shaking

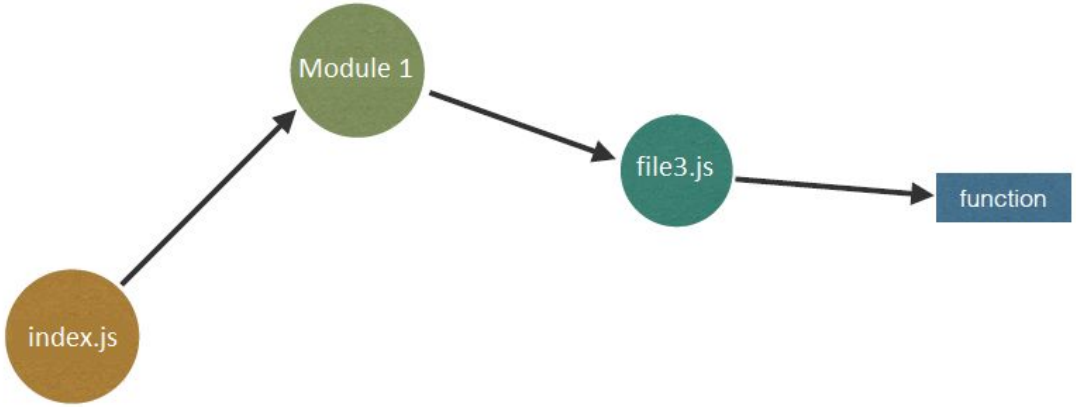
Subcategory of dead code elimination, but based on modules and their usages.

A module can export multiple features, but only ones that are imported (or used transitively) will remain in the bundle.

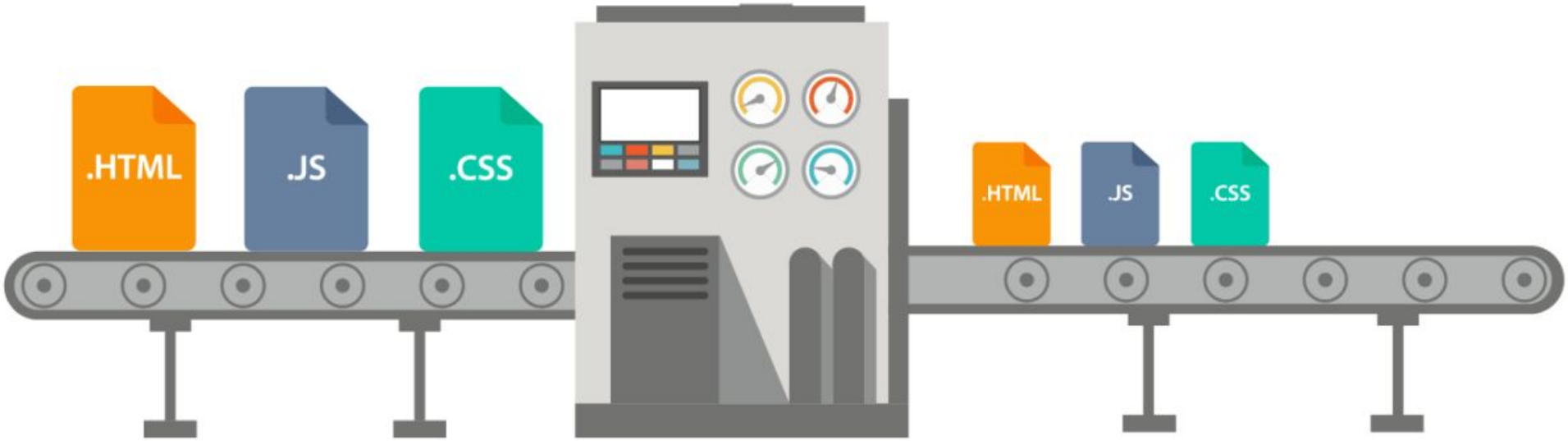
Tree Shaking



Tree Shaking



Minification



Minification

Minimizes the amount of code shipped as much as possible by:

- Removing redundant whitespace
- Shortening variable/function names
- Removing comments
- Replacing code constructs with other functionally-equivalent but shorter ones

```
// This is a comment that will be removed
const array = [];
for (var i = 0; i < 20; i++) {
  array[i] = i;
}
```

```
for(var a=[i=0];i<20;a[i]=i++);
```

Benefits of minification & tree-shaking

1. Faster load times
2. Lower business costs
    a. less data is transmitted over the network
    b. lower resource usage since less data needs to be processed for each request
    c. generated once, use for an unlimited number of requests.
3. Better development experience

Before Minification

```
<html>
<head>
    <style>
        #myContent { font-family: Arial }
        #myContent { font-size: 90% }
    </style>
</head>
<body>
<!-- start myContent -->
<div id="myContent">
    <p>Hello world!</p>
</div>
<!-- end myContent -->
</body>
</html>
```

Total: 250 bytes

After Minification

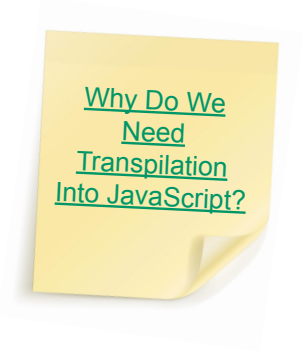
```
<style>#d{font-family:Arial;font-size:90%}</style><div
id=d><p>Hello world!</div>
```

Total: 81 bytes

Transpilation & Babel

Transpilation

Transpilation is the process of transforming the program written in language **X** into the equivalent program in language **Y**. In contrast to compilation, languages **X** and **Y** have roughly the same level of abstraction.



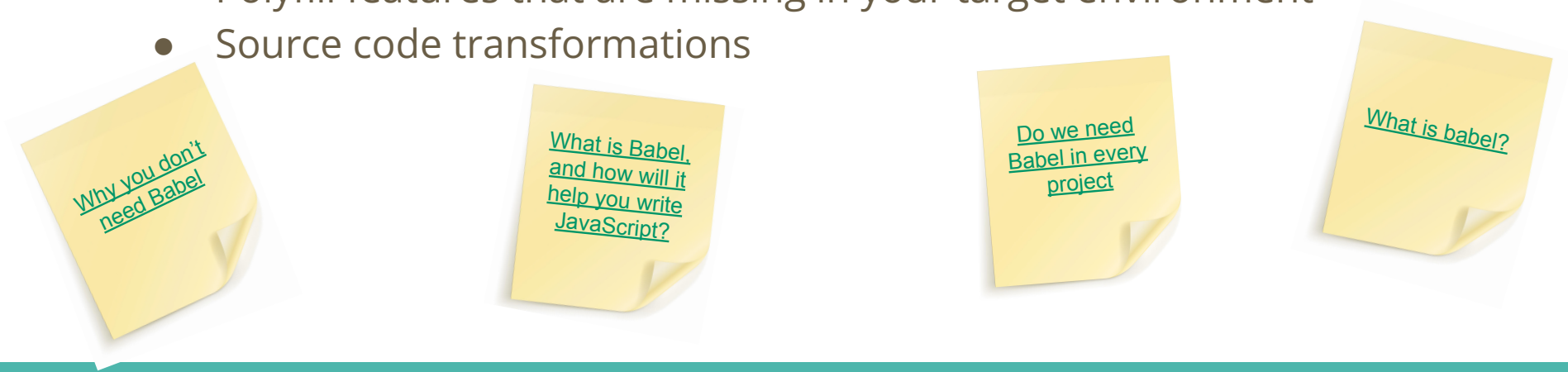
Why do we need it?

1. Migration between different versions of the same language
2. Translation from one programming language into another based on the runtime system requirements and/or developers' wishes

Babel

Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments. Here are the main things Babel can do for you:

- Transform syntax
- Polyfill features that are missing in your target environment
- Source code transformations



Example: JS

```
// Babel Input: ES2015 (ES6) arrow function
[1, 2, 3].map(n => n + 1);

// Babel Output: ES5 equivalent
[1, 2, 3].map(function(n) {
    return n + 1;
});
```

Example: React

```
// JSX Syntax
const Component = () => {
    return (
        <div style={{color: '#fff'}}>
            Convert JSX to JS
        </div>
    )
};

// Pure JS
const Component = () => {
    return React.createElement('div', {
        style: {color: '#fff'}
    }, 'Convert JSX to JS')
};
```

Polyfills

“A piece of code (usually JavaScript on the Web) used to provide modern functionality on older browsers that do not natively support it.” - MDN

```
// check if the method `startsWith` exists on the standard built-in `String`
if (!String.prototype.startsWith) {
    // if not we add our own version of the native method newer browsers provide
    String.prototype.startsWith = function (searchString, position) {
        position = position || 0;
        return this.substr(position, searchString.length) === searchString;
    };
}
```

Polyfills and transpilers

Introduction To Polyfills & Their Usage

Recap

TypeScript

Why?

```
<body>
  <input id="input" type="number" />
  <p id="result">Can you guess the 2 bugs here?</p>
  <script>
    const p = document.getElementById('result');
    const input = document.getElementById('result');
    function sum(a, b) { return a + b; }
    p.textContent = sum(input.value, 10);
  </script>
</body>
```

It works =/= It's correct!

Typos

```
const announcement = "Hello World!";

// How quickly can you spot the typos?
announcement.toLocaleLowercase();
announcement.toLocalLowerCase();

// We probably meant to write this...
announcement.toLocaleLowerCase();
```

Uncalled functions

```
function flipCoin() {
  // Meant to use Math.random()
  return Math.random < 0.5;
  Operator '<' cannot be applied to types '() => number' and 'number'
}
```

Basic logic errors

```
const value = Math.random() < 0.5 ? "a" : "b";
if (value !== "a") {
  // ...
} else if (value === "b") {
  // Oops, unreachable
}
```

This condition will always return 'false' since the types '"a"' and '"b"' have no overlap


DYNAMIC TYPING
STATIC TYPING
DONE!

What is TypeScript?

- A statically-typed superset of JavaScript
  - Can be progressively adopted
  - Inter-operates with existing JS code
  - Structurally-typed
- Just a compiler, no runtime
  - Transpiles to JavaScript
- Developed by Microsoft in 2012
- Open-source

Advantages

- *Optional* static typing
- Better code readability
- IDE support (auto-completion)
- Easier refactoring
- Integration of newer ES standards
  - built-in transpiling (downleveling) to older versions
- Massive community

Basics of typing

```
let myName: string = "Alice";

function greet(person: string, date: Date): void {
   console.log(`Hello ${person}, today is ${date.toDateString()}!`);
}

greet("Brendan");
```

```
Expected 2 arguments, but got 1
```

```
greet("Maddison", Date());
```

```
Argument of type 'string' is not assignable to parameter of type 'Date'
```

Basics of typing: Type erasure

Compiled output:

```
"use strict";
function greet(person, date) {
    console.log("Hello " + person + ", today is " +
date.toDateString() + "!");
}
greet("Maddison", new Date());
```

Basics of typing: Built-in types

- All primitives: `string`, `number`, `boolean`, …
  - Note: never use the capital (`String`, `Number`, `Boolean`) alternatives!
- Arrays: `number[]`, or `Array<number>`
- Tuples: `[x: number, y: number]`
  - Actually just arrays under the hood
- `any`: turns off type-checking entirely
- `unknown`: as the name implies 🙂

Interfaces

```
interface PaintOptions {
    shape: string;
    xPos?: number;
    yPos?: number;
}

function paintShape(opts: PaintOptions) {
    // ...
}
```

Extending Interfaces

```
interface Colorful {
   color: string;
}

interface Circle {
   radius: number;
}

interface ColorfulCircle extends Colorful, Circle {}

const cc: ColorfulCircle = {
   color: "red",
   radius: 42,
};
```

Implementing Interfaces

```
interface Pingable {
    ping(): void;
}

class Sonar implements Pingable {
    ping() {
        console.log("ping!");
    }
}
```

Type Inference

```
let msg = "hello there!";
```

```
let msg: string
```

```
let x = [0, 1, null];
```

```
let x: (number | null)[]
```

Type Inference: Contextual typing

```
const names = ["Alice", "Bob", "Eve"];


names.forEach(function (s) {
  console.log(s.toUppercase());
});
```

```
Property 'toUppercase' does not exist on type 'string'. Did you mean 'toUpperCase'?
```

Generics

```
// Only numbers
function identity(arg: number): number {
    return arg;  // return 5;
}

// Any type for input, and any type for output
function identity(arg: any): any {
    return arg; // return { hello: "world" };
}

// Generic type
function identity<Type>(arg: Type): Type {
    return arg; // the only possible implementation

}
```

Type assertion (casting)

```
const myCanvas = document.getElementById("main_canvas") as
HTMLCanvasElement;
```

Only allows more specific or less specific type casts (cannot be unrelated)

```
const x = "hello" as number;
```

```
Conversion of type 'string' to type 'number' may be a mistake because neither type
sufficiently overlaps with the other
```

```
const y = ("hello" as any) as number;
```

Type aliases

```
type Point = { x: number; y: number };

type UserInputSanitizedString = string;

function sanitizeInput(str: string): UserInputSanitizedString {
 return sanitize(str);
}

// Create a sanitized input
let userInput = sanitizeInput(getInput());

// Can still be re-assigned with a normal string though
userInput = "new input";
```

Function types

```
type Logger = (msg: string) => void;

let toFixed: (digits: number | undefined) => string;


function add(a: string, b: string): string;
function add(a: number, b: number): number;
function add(a: any,    b: any):    any {
  return a + b;
}
```

Union types & type narrowing

```ts
type ID = number | string;

function printId(id: number | string) {
  if (typeof id === "string") {
    // In this branch, id is of type 'string'
    console.log(id.toUpperCase());
  } else {
    // Here, id is of type 'number'
    console.log(id);
  }
}
```

Dependent/Literal types

Values can be used as types.

```
type Alignment = 'left' | 'right' | 'center';

function printText(s: string, alignment: Alignment) { /**/ }


printText("Hello, world", "left");

printText("Top of the mornin' to ya", "centre");
```

```
Argument of type '"centre"' is not assignable to parameter of type '"left" | "right" | "center"'
```

Discriminated unions

```
interface Shape {
  kind: "circle" | "square";
  radius?: number;
  sideLength?: number;
}
function getArea(shape: Shape) {
  if (shape.kind === "circle") {
    return Math.PI * shape.radius ** 2;
                  Object is possibly 'undefined'.
  }
}
```

Discriminated unions

```ts
interface Circle {
 kind: "circle";
 radius: number;
}

interface Square {
 kind: "square";
 sideLength: number;
}

type Shape = Circle | Square;
```

```ts
function getArea(shape: Shape) {
  return Math.PI * shape.radius ** 2;
}
```

```
Property 'radius' does not exist on type 'Shape'
 Property 'radius' does not exist on type 'Square'
```

```ts
function getArea(shape: Shape) {
  if (shape.kind === "circle") {
    return Math.PI * shape.radius ** 2;
  }
}
```

```
(parameter) shape: Circle
```

Remember: Entities specific to TypeScript (like interfaces, type aliases, enums, etc.) do not exist in runtime and are either only used in compilation process (e.g. interfaces) or compiled into pure plain JavaScript (e.g. enums).


100 SECONDS OF
JS
TS
TypeScript

That’s all for today!

Resources

- TypeScript Playground (official docs)
  - Structural Typing
- The TypeScript Handbook (official docs)
- TypeScript in 5 minutes (official docs)
- The concise TypeScript book (GitHub)
- 5 Inconvenient Truths about TypeScript (short article)