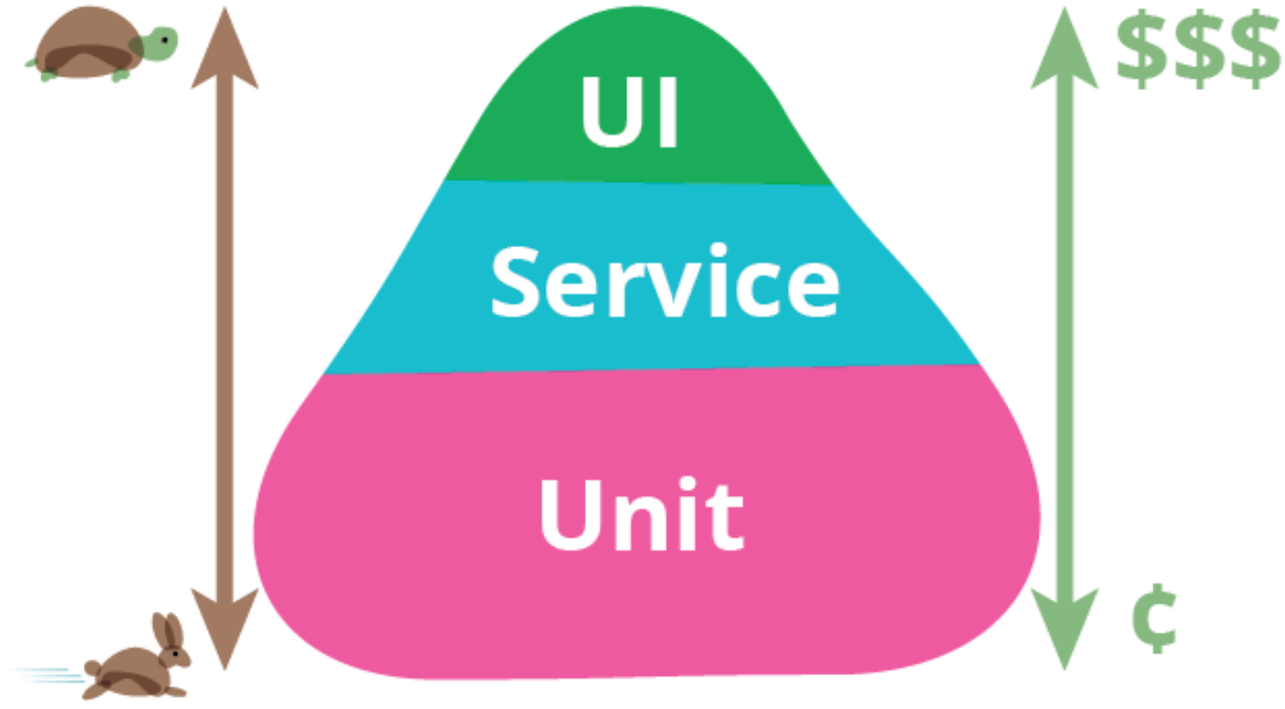
Unit tests

- You're already doing it
- Just make it reusable

Why automated tests?

- Increases confidence in the code
- Fearless refactoring
- Catch bugs early
- Deliver better software faster

The test pyramid



What should you test?

- Everything you write!
- In JS, this usually means every `export` should have tests
- You might consider `export`ing things just for testing
- You might change the design to make it more testable

What is a unit?

- The smallest testable part
- Only your code, not the network, or other external things
- In JS, a unit is usually a function

There are other kinds of tests

We'll look at them later

Unit testing frameworks

- There are several: Mocha, Qunit, Jasmine
- We're using Jest by Facebook
- Jest comes with batteries included

Writing a test

- *filename*.spec.js
- or __tests__/*.js

Writing a test

```
it('should add two numbers', () => {
  // perform test here
});
```

Writing a test

```
it('should add two numbers', () => {
  expect( add(1, 2) ).toEqual(3);
});
```

Writing a test

```
it('should calculate interest correctly', () => {
  // setup
  const interest = calculateInterest(1000, 10);

  // assertions
  expect(interest).toEqual(100);
});
```

Lab

- `day-1/4-unit-tests`
- Find further instructions in `README.md`
- Write code and tests

Try this

- Try writing the code first, then the test
- Try writing the test first, then the code

Cheat sheet

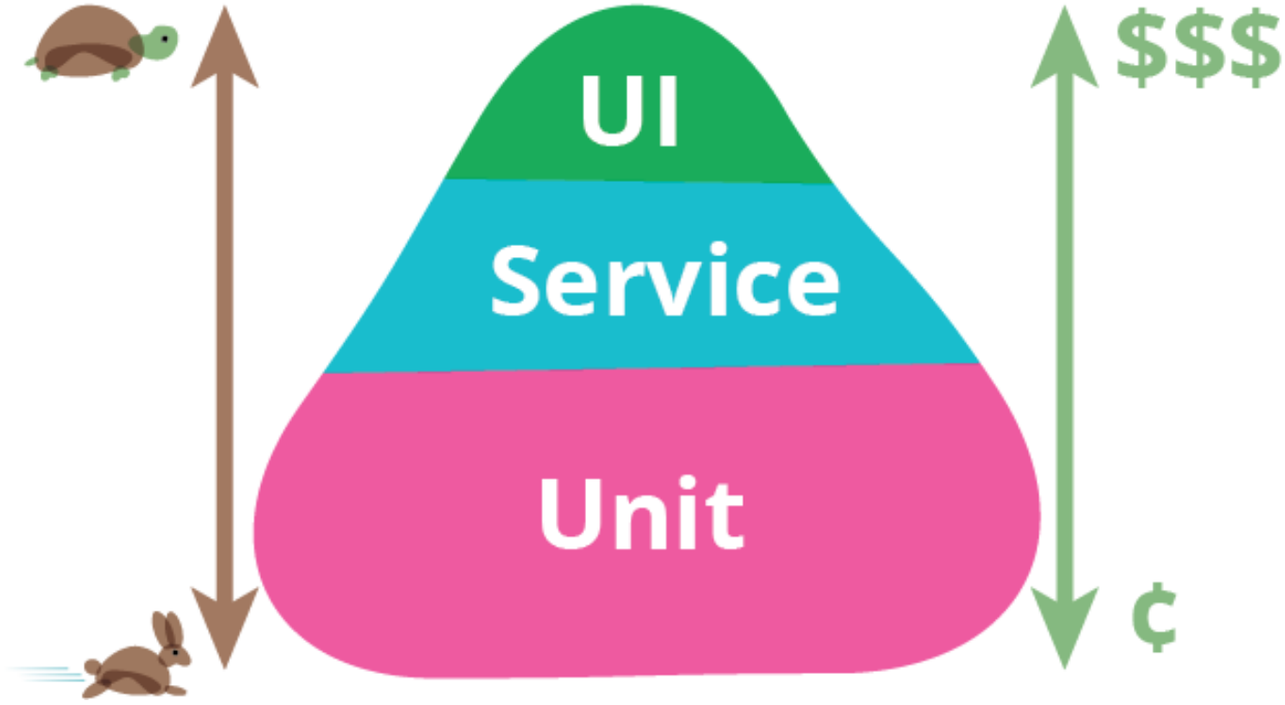
- Write a module to add, subtract, multiply and divide two numbers
- If any input is not a number, it should return `null`.
- You can use multiple `it` blocks and multiple assertions if you want
- Try writing tests first

```
it('should calculate interest', () => {
  // setup
  const interest = calcInterest(1000, 10

  // assertions
  expect(interest).toEqual(100);
});
```

Unit tests

The layers of testing


UI
Service
Unit
$$$
¢

Things we need to test

- Does our code do what's expected?
- Does our code interact correctly with the DB or API?
- Does the system as-a-whole behave correctly?
- Does the UI behave correctly?
- Does this match what our users want?

Async tests

Jest supports returning a promise

```
it('should test something', () => {
  return addAsync(1, 2)
    .then(value => expect(value).toEqual(3));
});
```

But a function that returns a promise is...
an `async` function!

Jest with `async/await`

```
it('should test something', async () => {
  const value = await addAsync(1, 2);
  expect(value).toEqual(3);
});
```

Refactor lab 2's tests

How do we test something with an external dependency?

`fs`? DB? API?

How do we test something with an external dependency?

`getUserById`

How do we test something with an external dependency?

Start the database first?

Problems with using a DB

- The test machine needs to have a running DB
- Tests lose repeatability because of DB persistence
- Tests run slower

How about using a throwaway in-memory DB for the tests?

mockgoose

- A real DB is 'real', but brings its problems.
- A fake DB solves many problems, but may not give confidence.

Local DJ

- Get GPS update when location changes
- Call service with location to find song
- Play the song through a media player
- Queue up the song if something's playing
- Never repeat a song immediately
- Some locations won't have songs

What is code?

Questions

Please write down your top takeaway of the day.

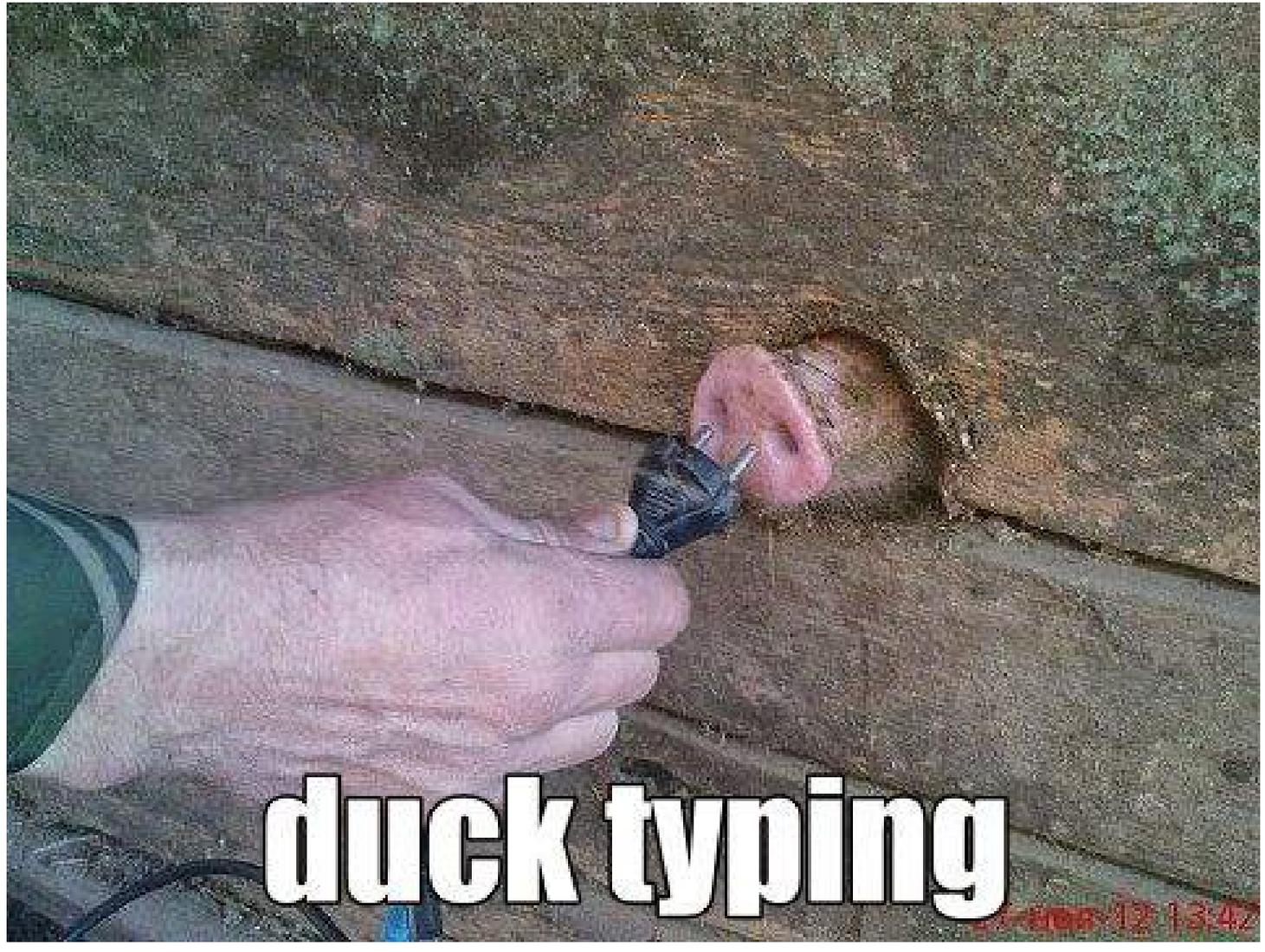
TypeScript

Why types?

JS already has types!

```
const a = "Hello world";
console.log(typeof a); // output: string

const b = 1; // number
const c = true; // boolean
const d = {}; // object
```


duck typing

What value does TypeScript add?

Converting a JS file to TypeScript

Rename `.js` to `.ts`!

How it works

- You write JS + Types:
- The compiler checks if your types match correctly
- The compiler throws away type information, converts it to JS

How it works

- Types are only checked at build time
- No runtime type-checking!

const vs. let/var

```
const greeting = "Hello"; // Definitely always stri

let greeting2 = "Hello"; // String right now
let greeting2 = 1; // Should this be allowed?
```

const vs. let/var

```
let greeting1;
greeting1 = "Hello";
greeting1 = 1;  // valid TS

let greeting2 = "Hello";
let greeting2 = 1; // Error!
```

any

```
let greeting1;

// same as

let greeting1: any;  // 'any' type
```

string

```
let greeting1 = "Hello";

// same as

let greeting1: string = "Hello";
```

number

```
let greeting1 = 1;

// same as

let greeting1: number = 1;
```

Omit types where inference can do its job

```
let a = "Hello";
// same as
let a: string = "Hello";

let a;
// same as
let a: any;
```

Arrays

```
const a = [1, 2, 3];

// same as

const a: number[] = [1, 2, 3];

// or

const Array<number> = [1, 2, 3];
```

Functions

```ts
const increment = (x: number) => x + 1;

// same as

const increment = (x: number): number => x + 1;

// same as

const increment: (x:number) => number = x => x + 1;
```

Extra arguments

```
const increment: (x: number): number = x => x + 1;

increment(1, 2); // error
```

Too few arguments

```
const increment: (x:number): number = x => x + 1;

increment(); // error
```

Optional arguments

```ts
const increment = (x:number, by?:number) => {
  if(by === undefined) return x + 1;
  return x + by;
};

increment(1); // valid. by = 1
increment(1, 10); // valid. by = 10
increment(1, 2, 3); // error
```

Default arguments

```
const increment: (x:number, by?:number) => number =
  (x, by = 1) => x + by;

increment(10); // same as increment(10, 1);
```

Rest parameters

```
const sum = (...nums: number[]): number => {
  /* ... */
};

sum(1, 2, 3);
```

Array destructuring

```
const [ x, y ] = [ 1, 2 ];

// same as

const [ x, y ] : [number, number] = [ 1, 2 ];
```

Object destructuring

```ts
const { x, y } = { x: 1, y: 2 };

// same as

const { x, y } : { x: number, y: number } =
  { x: 1, y: 2 };
```

Object destructuring

```
const { x: renamed, y } = { x: 1, y: 2 };

// same as

const { x: renamed, y } : { x: number, y: number }
  { x: 1, y: 2 };
```

`null` and `undefined`

```
const u: undefined = undefined;
const n: null = null;
```

Subtype of all other types by default, but this can be changed

Lab

- `day-2/1-ts-basics`
- Ensure tests pass with `npm test`.
- Modify `tsconfig.json` as specified in `README.md`
- Add type signatures to make tests pass

Cheat sheet

```
// Type not always needed
const x = "Hello";
const y = 1;

// Destructuring
const [a: number, b: number] = [ 1, 2 ];

// Arrays
const num: number[] = [1, 2, 3];

// Functions
```

TypeScript

`never` type

```
const infiniteLoop = (): never => {
  while(true) {};
};
```

`void` type

```
const log = (thing): void => {
  console.log(thing);
};
```

Interfaces

Define the shape of your objects

Interfaces

```
interface NamedThing {
  name: string;
};

const printName = (x: NamedThing) =>
  console.log(x.name);

const person = { name: 'Alice', age: 42 };
printName(person);
```

Optional parameters

```
interface Person {
  name: string,
  age?: number
};

const printPerson = (person: Person) => {
  if(person.age) {
    return `${person.name} is ${person.age} yrs old.`;
  } else {
    return `This is ${person.name}.`;
  }
}

printPerson({ name: 'Alice' })
```

Excess property checks

```
interface Person {
  name?: string,
};

const printPerson = (person: Person) => // ...

printPerson({ named: 'Alice' }); // Error!
```

Use interfaces to avoid shape checks

```
interface Person {
  name: string,
};

const printPerson = (person: Person) => {
  // The following line isn't required
  if(typeof person.name !== 'string') throw ...;

  ...
}
```

Type Aliases

Similar to interfaces

Type Aliases

```ts
type NamedThing = {
  name: string
};

const printName = (x: NamedThing) =>
  console.log(x.name);

const person = { name: 'Alice', age: 42 };
printName(person);
```

Type Aliases with primitives

```
type Name = string;

type Person = {
  name: Name
};
```

Structural Typing (TS)
vs.
Nominal Typing (Java)

Structural Typing

```
type Person = {
  name: string
};

let person: Person;
person = { name: 'Alice' };
```

Nesting types

```
type Person = {
  name: string
};

type Ride = {
  commuters: Person[]
};
```

Lab

- `day-1/2-ts-objects`
- Verify that tests pass.
- Change `tsconfig.json` as shown in `README.md`
- Change `src/index.ts` to use `types`.

Cheat sheet

Add types to make object shapes explicit

```
type Person = {
  name: string,
  age?: number
};
```

Interfaces
Types

You can specify function interfaces

```
interface PersonsName {
  (p: Person): string
}

// or

type PersonsName = (p: Person): string
```

You can specify readonly properties

```
interface Person {
  readonly name: string
}

// or

type Person = {
  readonly name: string
}
```

Union Types

Union Types

```
const getName = personOrName => {
  if(typeof personOrName === 'string') {
    return personOrName;
  } else {
    return person.name;
  }
};

getName('Alice'); // output: 'Alice'
getName({ name: 'Alice' }); // output: 'Alice'
```

Union Types

```
const getName = (personOrName: any) => {
  if(typeof personOrName === 'string') {
    return personOrName;
  } else {
    return person.name;
  }
};

getName(true); // should be invalid, but isn't
```

Union Types

```
const getName = (personOrName: Person | string) =>
  if(typeof personOrName === 'string') {
    return personOrName;
  } else {
    return person.name;
  }
};

getName(true); // Compile error!
```

String Literal Types

String Literal Types

```
type LoadingState = 'loading' | 'success' | 'failure

type Request = {
  state: LoadingState,
  ...
};

const request: Request = { state: 'loading', ... };
```

Number Literal Types

Number Literal Types

```ts
type DieRolls = 1 | 2 | 3 | 4 | 5 | 6;

const rollDie = (): DieRolls => {
  // ...
};
```

Discriminated Unions

Tagged Unions
Algebraic Data Types

Discriminated Unions

```ts
type Square = { type: 'square', size: number };
type Circle = { type: 'circle', radius: number };
type Rectangle = {
  type: 'rectangle', width: number, height: number
};

type Shape = Square | Circle | Rectangle;
```

Discriminated Unions

```
const area = (shape: Shape): number => {
  switch(shape.type) {
    'square': return shape.size * shape.size;
    'circle': return Math.PI * shape.radius ** 2;
    'rectangle': return shape.width * shape.height;
    default: const n: never = shape;
  }
};
```

Exhaustiveness checking

```
type Shape = Square | Rectangle | Circle | Triangle

const area = (shape: Shape): number => {
  switch(shape.type) {
    'square': return shape.size * shape.size;
    'circle': return Math.PI * shape.radius ** 2;
    'rectangle': return shape.width * shape.height;
    default: const n: never = shape; // Compile err
  }
};
```

Discriminated Unions are everywhere

```
type PaymentMethod = CreditCard | PayPal | Cash;

const processPayment = (method: PaymentMethod) => {
  switch(method.type) {
    case 'credit-card': return `CC: ${method.cardNo}
    case 'paypal': return `PayPal: ${method.email}`;
    case 'cash': return 'Cash';
    case default: const n: never = method;
  }
};
```

Discriminated Unions are everywhere

```
type AuthMethod = Cookies | Headers;

type Packet = Ping | Odometry | GPSLocation;

type ProductTypes = Shirts | Shoes | Wallets;
```

Lab

- `day-1/3-ts-unions`
- Verify that tests pass.
- Change `tsconfig.json` as shown in `README.md`
- Change `src/index.ts` to use union types.

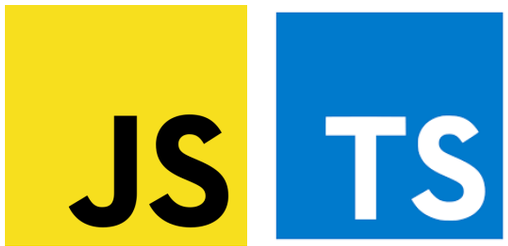
Cheat sheet

- Modify `src/index.ts` to use union types
- Bonus: Add a PayPal payment type

```
type Square = { kind: 'square', size: number };
type Circle = { kind: 'circle', radius: number

type Shape = Square | Circle;
```

Union Types



Classes

> *Actually, I invented the term Object-Oriented, and I can tell you I did not have C++ in mind.*
>
> *— Alan Kay*

> *The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana... and the entire jungle.*
>
> *— Joe Armstrong, creator, Erlang*

What is Object Oriented Programming?

What can OOP do that others can't?

Objects are (usually) containers for mutable state

State is bad!

We'll talk more about state in the next section

Design Patterns

Design Patterns are ways to overcome limitations in a programming language

Design Patterns from Java, C++, etc. don't apply to languages like JavaScript

OO pattern/principle	FP pattern/principle
Single Responsibility Principle	Functions
Open/Closed principle	Functions
Dependency Inversion Principle	Functions, also
Interface Segregation Principle	Functions
Factory pattern	Yes, functions
Strategy pattern	Oh my, functions again!
Decorator pattern	Functions
Visitor pattern	Functions

Scott Wlaschin

JavaScript has first-class functions
This makes it more powerful than classical OO languages

Most OO patterns are redundant in JS

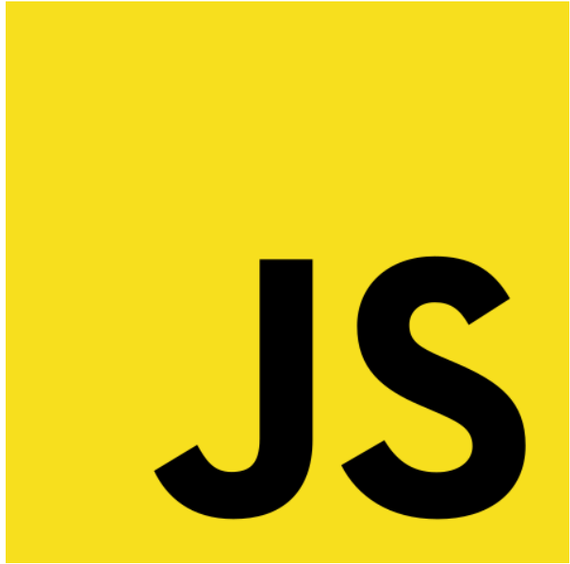
The complexity of OO

- Classes
- Inheritance
- Polymorphism
- Public/private/protected members
- Getters and setters
- `this`

Temporal Complexity

OO in JS is unfamiliar

- Prototypal inheritance
- Dynamically scoped `this`



OO in JS

```
let Greeter = (function () {
  function Greeter(message) {
    this.greeting = message;
  }
  Greeter.prototype.greet = function () {
    return "Hello, " + this.greeting;
  };
  return Greeter;
})();
```

`class` syntax sugar

```
class Greeter {
  constructor(message) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}
```

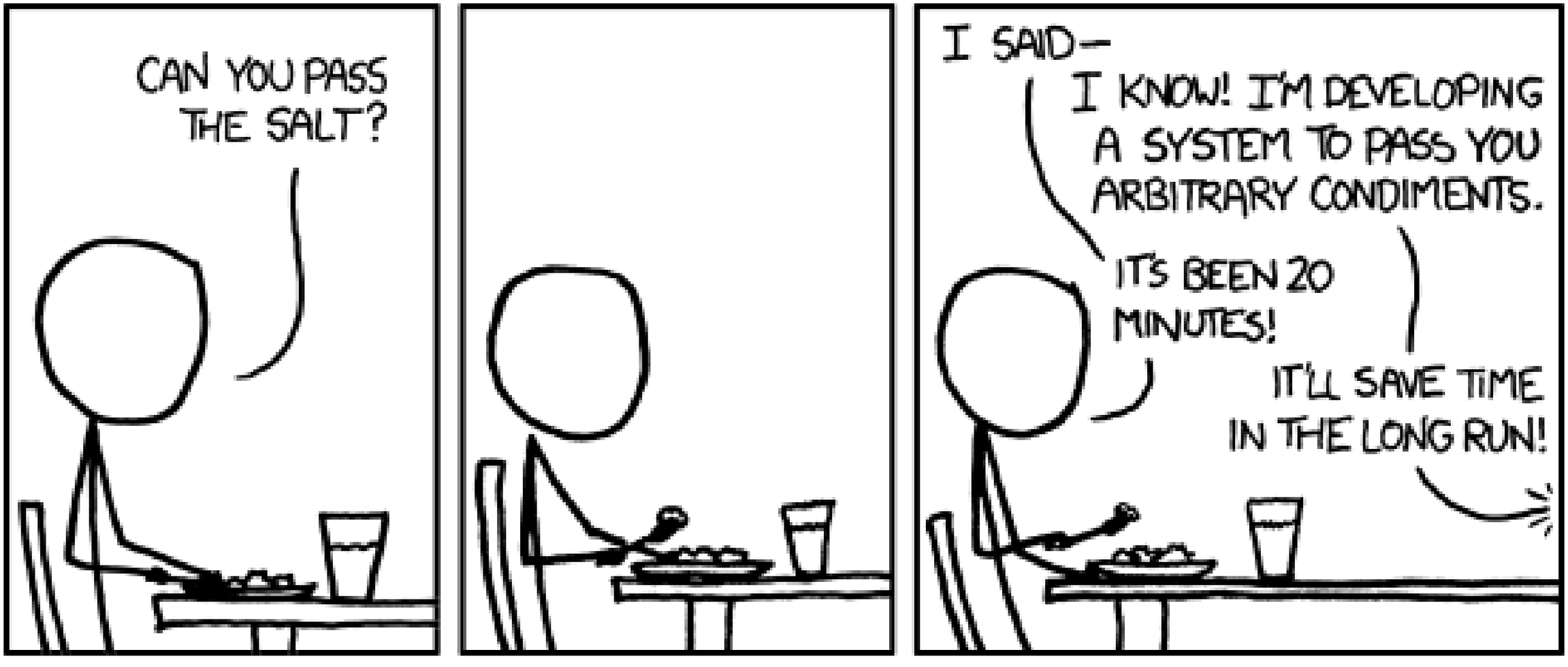
`class` with types

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}
```

`AbstractAnnotationConfigDispatcherServletIniti`

springframework.web.servlet

This doesn't make your code better!


CAN YOU PASS THE SALT?
I SAID—
I KNOW! I'M DEVELOPING A SYSTEM TO PASS YOU ARBITRARY CONDIMENTS.
IT'S BEEN 20 MINUTES!
IT'LL SAVE TIME IN THE LONG RUN!

> *Complexity is your enemy. Any fool can make something complicated. It is hard to keep things simple.*
> *— Richard Branson*

Avoid classes

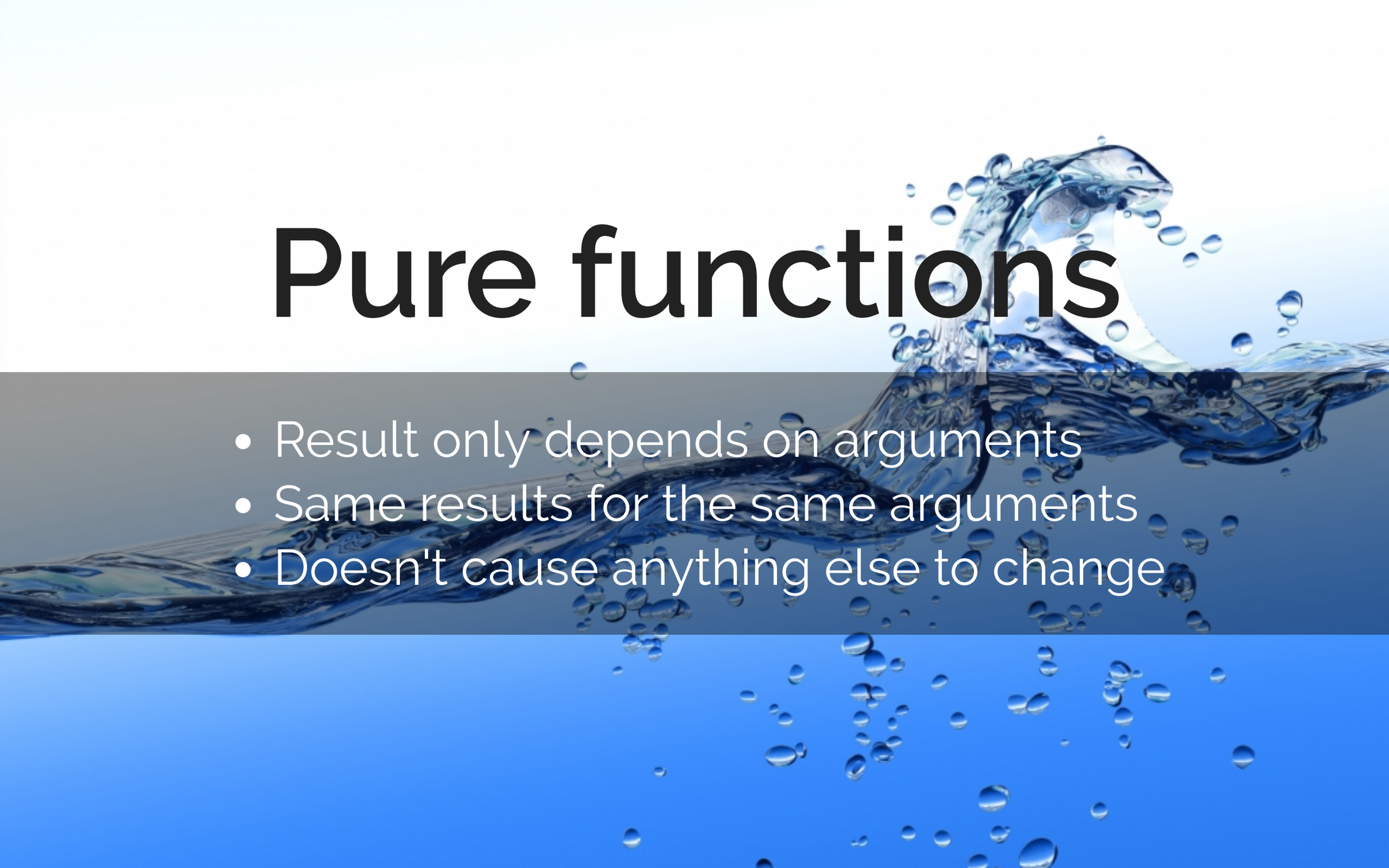
Too much complexity, too little gain

Functional Programming

Functions in maths

$$f(x) = x + 1$$

Result only depends on its argument



Pure functions

- Result only depends on arguments
- Same results for the same arguments
- Doesn't cause anything else to change

Pure functions

```
const add = (x, y) => x + y;

String.prototype.toUpperCase

Math.floor
```

Side effects

Side effects

```
let currentUserId = 3; // state

const setCurrentUserId = userId => {
  currentUserId = userId; // side-effect
  return id;
};

setCurrentUserId(4);
```


State is a slice of time

> *In the presence of side effects, a program's behaviour may depend on history; that is, the order of evaluation matters.*
>
> *Understanding and debugging a function with side effects requires knowledge about the context and its possible histories.*
>
> *— Side effect, Wikipedia*

Mutable state is very complex

Side effects

```
const addToArray = (arr, item) => {
  arr.push(item); // Mutated argument!
  return arr;
};
```

Side effects

```
// Other examples of side effects
console.log('Hello world');
Math.random();
Date.now();
fetch(url);
```

It is not possible to avoid side-effects

But we can at least externalize it

Make the impure pure

Make the impure pure

```
const currentId = 0;
const getNextId = () => {
  currentId++;
  return currentId;
};

// becomes

const getNextId = currentId => currentId + 1;
getNextId(0);
```

Make the impure pure

```
const addToArray = (arr, item) => {
  arr.push(item);
  return arr;
};

// becomes

const addToArray = (arr, item) => ([ ...arr, item ]
```

Make the impure pure

```
const addToObject = (obj, key, value) => {
  obj[key] = value;
  return obj;
};

// becomes

const addToObject = (obj, key, value) =>
  ({ ...obj, [key]: value });
```

Why pure functions?

Cacheability

```
const add2 = x => x + 2;

add2(4); // output: 6
add2(4); // output: still 6
add2(4); // output: it isn't going to change
```

Testable

Testing is easy, since setup is minimal.

Easy to reason about

No external dependencies

Referential transparency

Referential Transparency

```
const increment = x => x + 1;

const afterZero = increment(0);
```

Parallelisation

Functional Programming

First Class Functions

Functions are types

Functions are types

```
const x = () => {};

console.log(typeof x); // output: 'function'
```

Functions are types

```
const foo = {
  bar: () => {}
};

const foo = [
  () => {},
  () => {}
];
```

Functions are types

```
fs.readFile('/path/to/file', (err, data) => {
  // ...
});
```

Functions are types

```
new Promise((resolve, reject) => {
  resolve();
});
```

Currying