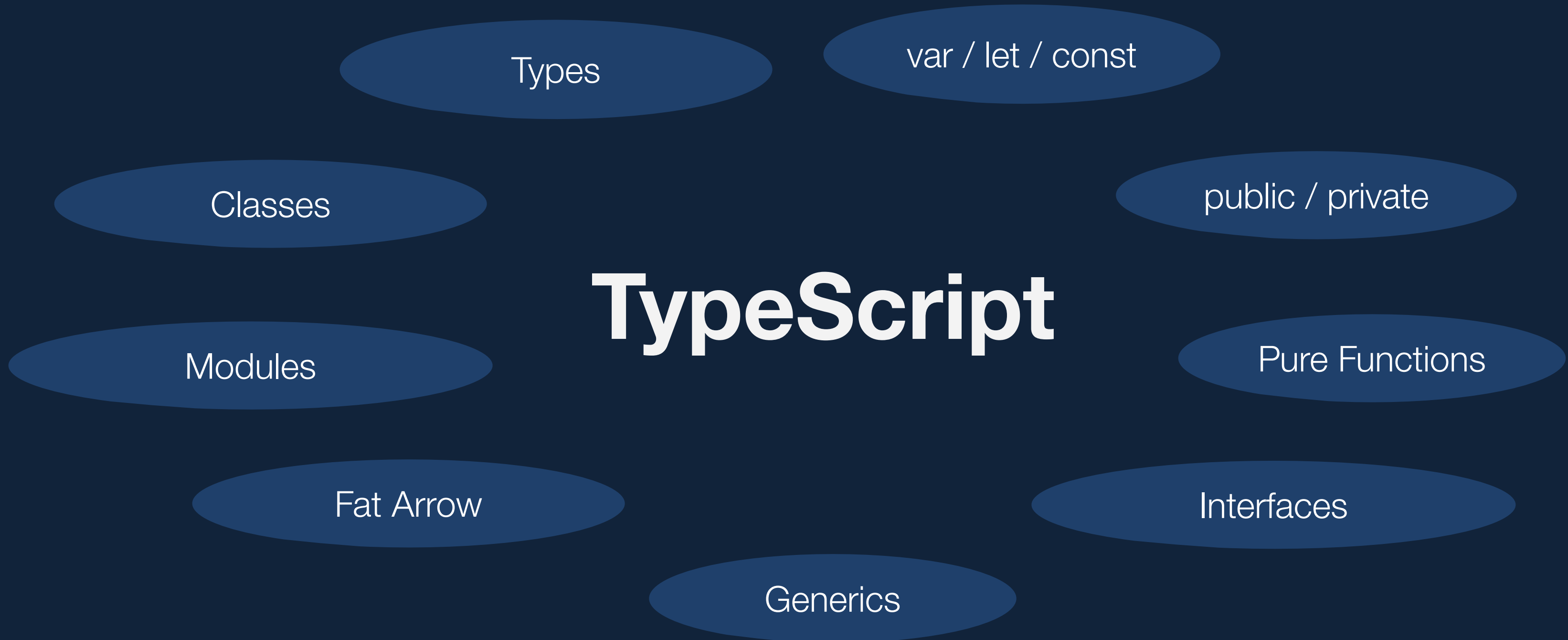
## Workshop

# TypeScript Introduction

# Task: Test your knowledge



# TypeScript

Optional static type-checking

TypeScript is a **typed superset** of JavaScript

that **compiles to plain JavaScript**.

# Why TypeScript

# Why **TypeScript**

- Statement completion and code refactoring
- Symbol-based navigation
- Avoid simple tests `(expect(service.get).toBeDefined)`
  - *It's better for the compiler to catch errors than to have things fail at runtime*.
- Types provide documentation
  - *The function signature is a theorem and the function body is the proof*.

**The result:** better maintenance for long-living projects

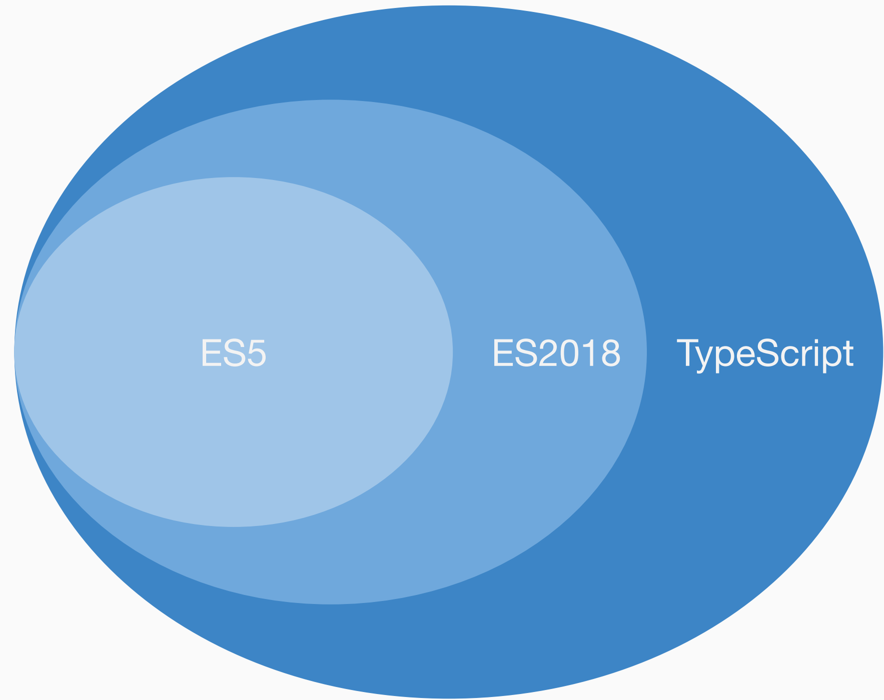
# Differences
# TypeScript vs EcmaScript

# What is **ECMAScript?**

- Standardardization of JavaScript
- Most modern browsers support most of **ES2017** now
- **ES2018** is standardized, browser support upcoming
- We compile TypeScript → ES5 (which was before ES2015)

# TypeScript is a superset

- Superset of EcmaScript
- Compiles to clean code
- Optional Types



# Variables

declaration and usage

# Basics

# Variables - Declaration

<code>

Declared with the keyword `var`, `let` or `const`

```
var value;

const pi = 3.1416;

let $value___123;
```

# Variables - var, let, const

| | `var` | `let` | `const` |
|---|---|---|---|
| scope | function | block | block |
| value changeable | ✔ | ✔ | ❌ |
| Standard | since ever | ES2015 / TS | ES2015 / TS |
| Cases to use | nearly never | ~30% | ~70% |

`let` is the new `var`, but most of the time you should use `const`

# Scoping

<code>

var is not block-scoped. Only functions get a new scope!

```
var foo = 123;

if (true) {
    var bar = ‘test’;
    console.log(foo, bar);
    // => 123 ‘test’
}

console.log(foo, bar);
// => ???
```

```
var foo = 123;

if (false) {
    var bar = ‘test’;
    console.log(foo, bar);
    // => 123 ‘test’
}

console.log(foo, bar);
// => ???
```

# Scoping

<code>

var is not block-scoped. Only functions get a new scope!

```
var foo = 123;

if (true) {
    var bar = 'test';
    console.log(foo, bar);
    // => 123 'test'
}

console.log(foo, bar);
// => 123 'test'
```

```
var foo = 123;

if (false) {
    var bar = 'test';
    console.log(foo, bar);
    // => 123 'test'
}

console.log(foo, bar);
// => 123 undefined
```

# Variables - Naming

- Almost all arbitrary names
- Exceptions:
  - **no** whitespace
  - **not** starting with a number
  - **no** dashes
  - **no** JS keywords (e.g. typeof etc.)

# Variables - Fun fact

<code>

UTF-8 characters are also allowed!

```
const π = Math.PI;

const ლ_ಠ益ಠ_ლ = 42;

const T̳̿Ȟ̳͠ = 'Zalgo';
```

# Variables

<code>

Hold the result of an expression

```
const helloWorld = 'Hello World';

const helloFunction = function() {};

const returnValue = getCurrentTime();
```

# Variables - Primitive types

<code>

Assign by value

```
let a = 'Hello World';
let b = a; // Only value is copied
a = 4;

console.log(b);
// => 'Hello World'
```

# Variables - Object types

<code>

## Assign by reference

```
const person = { firstName: 'John', lastName: 'Doe' };
const secondPerson = person;     // Copy the reference
secondPerson.firstName = 'Jane'; // Modify the object using the reference

console.log(secondPerson);
// => { firstName: 'Jane', lastName: 'Doe' }
console.log(person);
// => { firstName: 'Jane', lastName: 'Doe' }
```

# Variables - Object types (Array)

<code>

Call by reference

```
const a = [1, 2, 3];
const b = a; // Copy the reference
a[0] = 99;  // Modify the array using the reference

console.log(b);
// => [99, 2, 3]
```

# const

# Variables with `const`

<code>

Reassigning throws an error

```
const birthdate = new Date();

birthdate = new Date(); // TypeError: Cannot assign to read only property
```

# Objects with `const`

<code>

Only the reference immutable.

```
const myObj = {name: 'Florian'};

// You cannot change the reference
myObj = {name: 'Peter'}; // TypeError: Assignment to constant variable

// But the object is mutable!
myObj.name = 'Andreas';
```

# Types

# Types in TypeScript - Variables

<code>

Types exist for primitive and object types.

```
const isDone: boolean = true;

const size: number = 42;

const firstName: string = 'Lena';

const attendees: string[] = ['Elias', 'Anna'];
```

# Types - Any

<code>

any takes any type

```
let question: any = 'Can be a string';

question = 6 * 7;
question = false;
```

# Type inference

# Type inference

<code>

You don’t have to set the type of a variable if the compiler can infer it.

```
const firstName = 'Max'; // string
const age = 30; // number
const isEmployed = true; // boolean
const friends = ['Stefan', 'Frederike']; // string[]
const dayOfBirth = Date.parse('...'); // Date
```

# null and undefined

By default each data type in JavaScript (and so in TypeScript) can accept null and undefined.

# null and undefined

<code>

By default each data type in TypeScript can accept null and undefined.

```
let firstName: string  = null;
let age: number = undefined;
let isEmployed: boolean; // undefined
```

# Compiler Flag: strictNullChecks

<code>

You can activate strict null checks in the config file of your project.

```
// tsconfig.json
{
    ...
    'strict': true,
    ...
}
```

# Compiler Flag: strictNullChecks

<code>

In strict null checking mode, the null and undefined values are not in the domain of every type.

```
let firstName: string | null = null;
let age: number | undefined = undefined;
let isEmployed: boolean | undefined; // undefined
```

# ! - Non-Null Assertion Operator

<code>

Assert that its operand is non-null and non-undefined

```
let a = e.name;  // TS ERROR: e may be undefined.
let b = e!.name;  // OKAY. We are asserting that e is non-undefined.
```

# ! - Non-Null Assertion Operator

<code>

Assert that its operand is non-null and non-undefined

```
function setNextValue(node: ListNode, value: number) {
    if (node.next === undefined) {
        node.next = {data: value};
    }
}

function setNextValue(node: ListNode, value: number) {
    node.next!.data = value;
}
```

# Template Strings

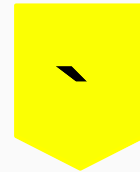
# Template Strings

**Template literals** are **string literals** allowing embedded expressions.

You can use **multi-line strings** and string interpolation features.

They were called **template strings** in prior editions of the ES2015 specification.

Template literals are enclosed by the **backtick** (




Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

# Template Strings

<code>

Simple template strings

```
const singleLine = `My first template string`; // single line
const multiLine = `<span>
    My first multiline template string!
  </span>`; // multiline
```

# Template Strings

<code>

Variables in template strings

```
const lastName = 'Eich';
const name = `Brendan ${lastName}`; // single line
// Brendan Eich

const multiLineString = `<span>
    My name is ${name}!
  </span>`; // multiline

// <span>
//    My name is Brendan Eich!
// </span>
```

# Template Strings

<code>

Expressions in template strings

```
const active = true;
const name = `Is active: ${active ? 'Yes' : 'No'}!`;
// Is active: Yes!

function isActive() {
    return 'No';
}
const name = `Is active: ${isActive()}!`;
// Is active: No!
```

# Objects

# Objects

An object is an **unordered** collection of **key-value pairs**.

# Objects

<code>

Object creation (equivalent behavior)

```
const a = {};

const b = new Object();
```

# Objects

<code>

## Object creation

```
let a = {};
let car = {
  make: 'Ford'
};
```

# Objects

<code>

Object property access

```
let car = {
  make: 'Ford'
};
car.model = 'Mustang';
car['full year'] = 1969;

// {
//   make: 'Ford',
//   model: Mustang',
//   'full year': 1969
// }
```

# Computed Property Names

<code>

Allows you to put an expression in brackets, that will be computed and used as the property name.

```
const param = 'size';
const config = {
  [param]: 12,
  ['mobile' + param.charAt(0).toUpperCase() + param.slice(1)]: 4
};

console.log(config); // {size: 12, mobileSize: 4}
```

# Spread Operator

<code>

Super useful to create shallow copies of arrays and objects.

```
// spread function arguments
myFunction(...iterableObj);

// create new arrays or objects
[...array, '4', 'five', 6];
{...obj, four: '5', six: SEVEN };

// create shallow copies
let objClone = { ...obj };
let arrayClone = [...array ];
```

# Task

## Add types and fix the errors

# Functions

# Functions - JavaScript

<code>

“First-class citizens”, functions are just expressions

```
const go = function() { alert('Hello JavaScript') };

http.get(url, function() {});
```

# Functions - JavaScript

<code>

Functions are also objects

```
const fn1 = function() {
    window.alert('Hello JavaScript');
};
fn1.foo = 'bar';
```

# Functions - Types

<code>

Add types to function arguments and return values.

```
function sayHi(firstName: string): void {
    console.log(firstName);
}
```

# Functions - Optional parameters

<code>

Parameters can be optional. Use a question mark.

```
function buildName(firstName: string, lastName?: string) {
  if (lastName) {
    return firstName + ' ' + lastName;
  } else {
    return firstName;
  }
}
```

# Functions - Default parameters

<code>

Function arguments can have defaults for arguments.

```
// type Inference: lastName is a string
function buildName(firstName: string, lastName: string = 'Bond') {
  return firstName + ' ' + lastName;
}
```

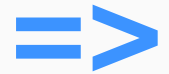
# Functions - Rest/Spread parameter

<code>

An arbitrary amount of parameters can be stored in an array.

```
function buildName(firstName: string, ...restOfNames: string[]) {

  const allNames = [firstName, ...restOfNames];
  // names = [firstName, restOfName[0], restOfName[1] ...]

  return allNames.join(' ');
}
```

# Fat Arrow



# Functions - Fat-Arrow-Function

<code>

## Concept

```
const square = (n) => { return n * n };

// const square = function(n) { return n * n; };
```

# Functions - Fat-Arrow-Function (Benefits)

<code>

Implicit return without a block

```
const square = (n) => n * n;

// const square = function (n) { return n * n; };
```

# Functions - Fat-Arrow-Function (Benefits)

<code>

Not need of braces around single parameters.

```
const square = n => n * n;

// const square = function (n) { return n * n; };
```

# Functions - Fat-Arrow-Function (Benefits)

<code>

Use braces around arguments if you have multiple parameters.

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

// const sum = function (a, b) { return a + b; };
```

# Functions - Fat-Arrow-Function

<code>

Use *curly braces* and *return* if you have multiple lines

```
const even = n => {
  const rest = n % 2;
  return rest === 0;
};
```

```
// const even = function(n) {
//   const rest = n % 2;
//   return rest === 0;
// };
```

# Functions - Fat-Arrow-Function

<code>

Use round *braces* and *curly braces* if you wanna return an object.

```
const person = () => ({
  firstName: 'John',
  lastName: 'Doe',
});
```

```
// const person = function() {
//   return {
//     firstName: 'John',
//     lastName: 'Doe',
//   };
// };
```

**this**

# Global context

# `this` - Global context

<code>

Outside of any function, `this` refers to the global object (*window*).

```
this.myTest = 42
console.log(window.myTest) // 42

this === window // true
```

# Function context

## Inside a function, the value of this depends on how the function is called.

# `this` - Arrow Functions

<code>

In arrow functions, `this` is set lexically, i.e. it's set to the value of the enclosing execution context's `this`.

```
const outerContext = this
const fatArrowFunction = () => this === outerContext

fatArrowFunction() // ==> true
```

# `this` - In objects

<code>

`this` is set to the object itself.

```
const myObject = {
    answer: 42,
    method: function () { return this.answer }
};

console.log(myObject.method()); // ==> 42
```

# `this` - In constructors

<code>

When a function is used as a constructor (with the `new` keyword), its `this` is bound to the new object being constructed.

```
function MyConstructor() { this.a = 42 }

const myInstance = new MyConstructor() // this is returned per default

console.log(myInstance.a) // ==> 42
```

# Task

## Setting this in functions

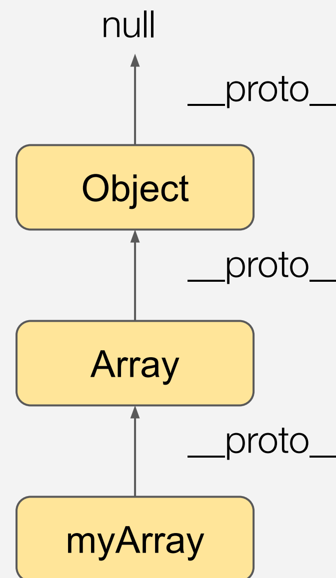
# Prototype

# Prototype

<code>

Every object has a prototype (`__proto__`).

```
const myArray = new Array();

myArray.__proto__.constructor === Array
// => true
```



# Prototype

<code>

You can inherit properties from objects using `__proto__`.

```
const hamster = {
    eat(food) {
        console.log(`${this.name}: "Oh ${food}! Nom nom..."`);
    }
};

const speedy = { name: 'speedy', __proto__: hamster };
const lazy   = { name: 'lazy',   __proto__: hamster };

speedy.eat('a battery'); // speedy: "Oh a battery! Nom nom..."
lazy.eat('a coconut');   // lazy: "Oh a coconut! Nom nom..."
```

# Task

## Use Fat Arrow Functions

# Arrays and Iterables

# Arrays

<code>

Arrays are ordered - objects are not!

```
const a = ['a','b'];

console.log(a[0]); // a
```

# Arrays - Iterators

<code>

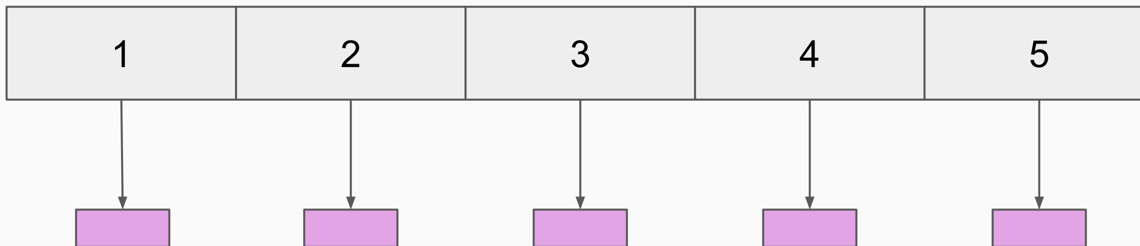
With a for and a for...of loop you have the opportunities to `break` or `continue` the loop and exit the surrounding function with `return`.

```
const names = ['Hanni', 'Nanni'];

for (let i = 0; i < names.length; i++) {
    console.log(names[i]);
}

for (let name of names) {
    console.log(name)
}
```

# Arrays - Iterators

Array.forEach()

```
const myArray = [1,2,3,4,5];
myArray.forEach(elem => console.log(elem));
```



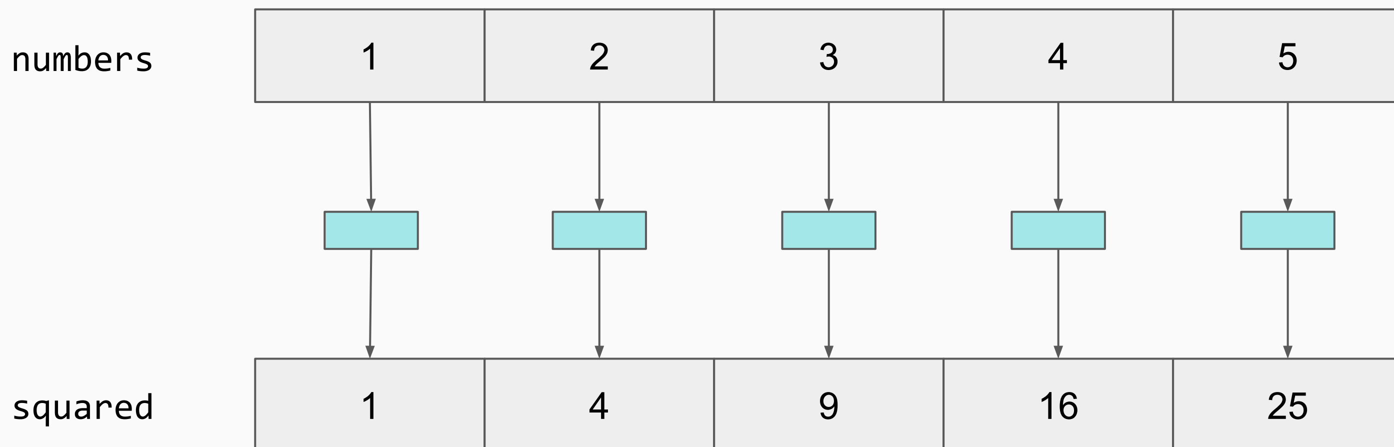

`forEach()` is slower than using a `for` loop!

# Arrays - Transformations

Array.map()

```
const numbers = [1, 2, 3, 4, 5];
const squared = numbers.map(num => num * num);
// squared is [1, 4, 9, 16, 25]
```

**Transforming an array**

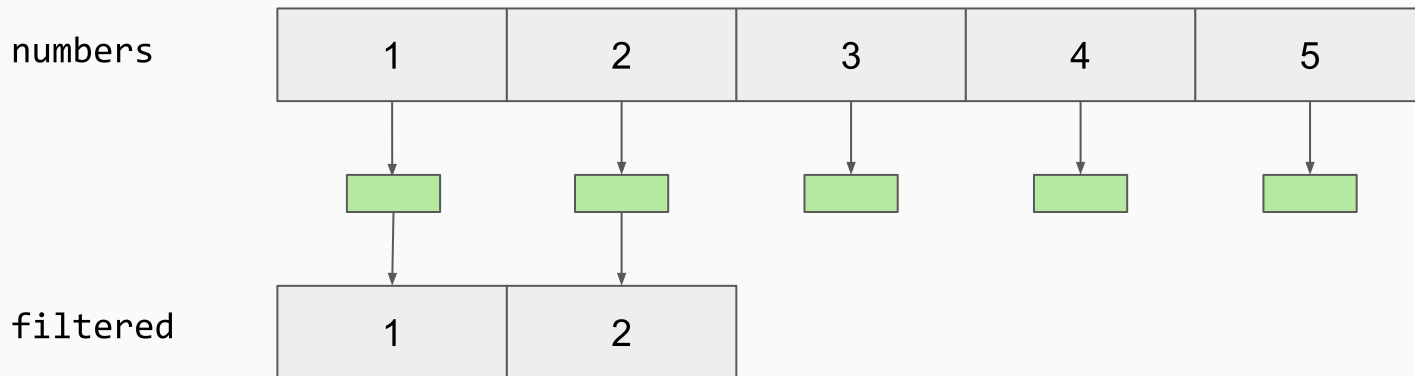


# Arrays - Transformations

Array.filter()

```
const numbers = [1, 2, 3, 4, 5];
const filtered = numbers.filter(num => num < 3);
// filtered is [1, 2]
```

**Filtering an array**



# Destructuring

# Destructuring - Objects

<code>

Get multiple local variables from an object with destructuring.

```
const circle = {radius: 10, x: 140, y: 70};

const {x, y} = circle;
// const x = circle.x;
// const y = circle.y;

console.log(x, y)
// => 140, 70
```

# Destructuring - Nested Objects

<code>

Access properties from nested objects

```
const circle = {x: 140, y: 70, style: {border: 'dashed'}};

const {x, y, style: {border}} = circle;
// const x = circle.x;
// const y = circle.y;
// const border = circle.style.border;

console.log(border)
// => 'dashed'
```

# Destructuring - Aliasing

<code>

Avoid potential naming collision by aliasing destructured properties

```
const x = ‘unrelated x’;
const y = ‘unrelated y’;

const circle = {x: 140, y: 70};

const {x: xAxis, y: yAxis} = circle;
// const xAxis = circle.x;
// const yAxis = circle.y;
```

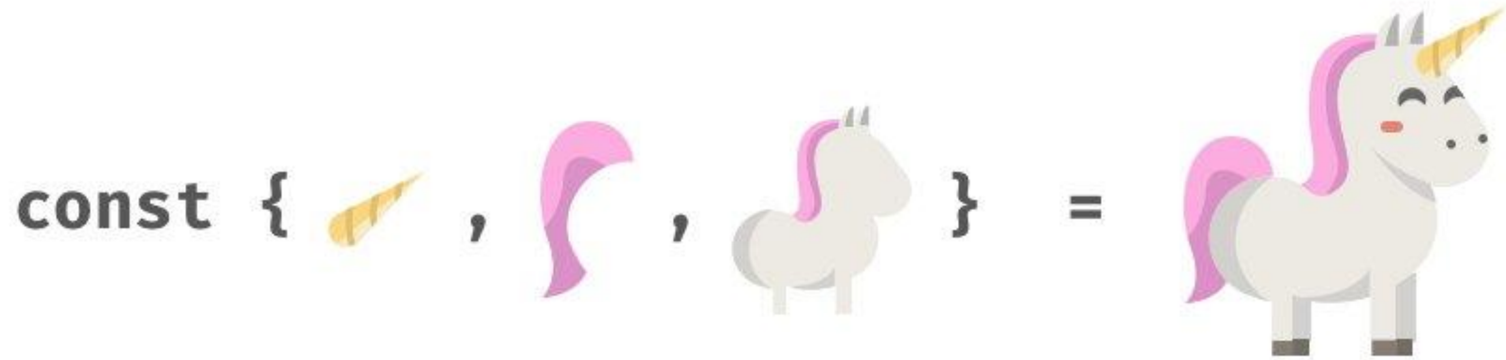
# Destructuring - Arrays

<code>

Get multiple local variables from an object with destructuring.

```
const coords = [51, 6];

const [lat, lng] = coords;
// const lat = coords[0];
// const lng = coords[1];

console.log(lat, lng)
// => 51, 6
```

# Destructuring



# Task

## Use Arrays and Destructuring

# HIGHER ORDER FUNCTIONS

A famous concept in functional programming

# Higher Order Functions

<code>

**#1** Functions that accept a function as parameter

```
http(url, () => {
    console.log('Ready!');
});
```

# Higher Order Functions

<code>

**#2** Functions that return a function

```
const createAdder = () => {
    return (a, b) => {
        return a + b;
    };
};

createAdder()(2, 3);

const myAdder = createAdder();
myAdder(2, 3);
```

# Not interesting without closures

# CLOSURES

# Closures

<code>

What happens with the variable after the function is terminated?

```
function getNumber() {
    const myNumber = 13;
    return myNumber;
}
getNumber();
```

# Closures

<code>

The result is?

```
const createFunction = () => {
    const localVar = 123;
    const data = [1,2,3,4,5];

    return () => localVar + 10;
};

const addTen = createFunction();
addTen(); // ???
```

# Closures

<code>

Functions that “enclose” local variables

```
const createFunction = () => {
    const localVar = 123;
    const data = [1,2,3,4,5];

    return () => {
        return localVar + 10;
    };
};
```

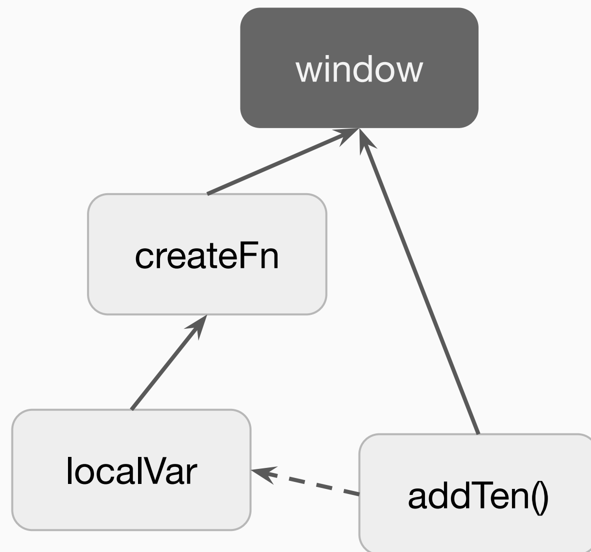
closure

```
const addTen = createFunction();
addTen(); // 133
```

- The inner function encloses *localVar* because it has read access to *localVar*.
- The **inner anonymous function** is a so-called **closure**.

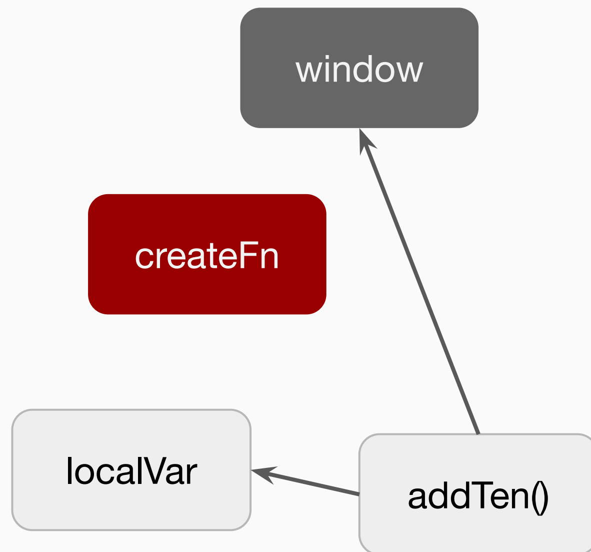
# Garbage Collection

```javascript
const createFunction = function() {
    const localVar = 123;

    return function() {
        return localVar + 10;
    };
};

const addTen = createFunction();
addTen(); // 133
```



# Garbage Collection

- mark & sweep
- reference counting
- elements without refs are garbage collected



# Closures

<code>

higher order functions and closures in combination

```
const createLogger = function(loggerName) {
    return function(msg) {
        console.log('[' + loggerName + '] ' + msg);
    };
};

const info = createLogger('INFO');

info('User successfully logged in!');
// [INFO] User successfully logged in!
```

# Task

## Closures in Action

# Pure Functions

A **pure function** doesn’t depend on and doesn’t modify the states of variables out of its scope.

# Why Pure Functions?

# Why **Pure Functions**

- Given the same input, will always return the same output
- Doesn’t have any side effect
- Easy to unit test
- Simpler, more flexible code

# Why **Pure Functions**

"when you call a pure function, you just need to focus on the return value as you know you didn't break anything elsewhere doing so."

https://www.nicoespeon.com/en/2015/01/pure-functions-javascript/

# Examples

# Impure Function - Example #1

<code>

The function modifies something outside of its own scope

```
const values = { a: 1 }

function impureFunction(items) {
  const b = 1

  items.a = items.a * b + 2

  return items.a
}

const c = impureFunction(values)
// Now `values.a` is 3, the impure function modifies it.
```

https://www.nicoespeon.com/en/2015/01/pure-functions-javascript/

# Impure Function - Example #1

<code>

The function modifies something outside of its own scope

```
const values = { a: 1 }

function impureFunction(items) {
  const b = 1

  items.a = items.a * b + 2

  return items.a
}

const c = impureFunction(values)
// Now `values.a` is 3, the impure function modifies it.
```

https://www.nicoespeon.com/en/2015/01/pure-functions-javascript/

# Pure Function - Example #1

<code>

The function doesn’t modifies anything outside of its own scope

```
const values = { a: 1 }

function pureFunction(a) {
  const b = 1

  a = a * b + 2

  return a
}

const c = pureFunction(values.a)
// `values.a` has not been modified, it's still 1
```

https://www.nicoespeon.com/en/2015/01/pure-functions-javascript/

# Impure Function - Example #2

<code>

The function depends on something outside of its own scope

```
const values = { a: 1 }
let b = 1

function impureFunction(a) {
  a = a * b + 2

  return a
}

const c = impureFunction(values.a)
// Actually, the value of `c` will depend on the value of `b`.
```

https://www.nicoespeon.com/en/2015/01/pure-functions-javascript/

# Impure Function - Example #2

<code>

The function depends on something outside of its own scope

```
const values = { a: 1 }
let b = 1

function impureFunction(a) {
  a = a * b + 2

  return a
}

const c = impureFunction(values.a)
// Actually, the value of `c` will depend on the value of `b`.
```

https://www.nicoespeon.com/en/2015/01/pure-functions-javascript/

# Pure Function - Example #2

<code>

The function doesn’t depend on anything outside of its own scope

```
const values = { a: 1 }
let b = 1

function pureFunction(a, c) {
  a = a * c + 2

  return a
}

const c = pureFunction(values.a, b)
// Here it's made clear that the value of `c` will depend on
// the value of `b`. No sneaky surprise behind your back.
```

https://www.nicoespeon.com/en/2015/01/pure-functions-javascript/

# Classes

# Classes in TypeScript

- Code is more readable
- Syntactic sugar over prototype-based inheritance
- Not introducing a new object-oriented inheritance model

# Classes in TypeScript

<code>

Class can have a `constructor`, `attributes` and `methods`.

```
class Person {
  bornOn: Date;

  constructor(name: string) {
    this.bornOn = new Date();
  }

  shout(): void { alert('Hello TypeScript!'); }
}
```

# Classes in TypeScript

<code>

Class *attributes* and *methods* can be public or private.

```
class Person {
  bornOn: Date; // public by default

  public name: string;

  private weight: number;
}
```

# Classes in TypeScript

<code>

Declare a class property from a constructor parameter.

```
class Person {
  bornOn: Date;

  constructor(public name: string) {
    this.bornOn = new Date();
  }

  shout(): void { alert('Hello TypeScript!'); }
}
```

# Classes in TypeScript - Instances

<code>

Create new instances with the *new* keyword.

```
class Person {…}

const john = new Person('John');

john.bornOn; // => a Date object

john.shout(); // => nothing but alerts
```

# Classes in TypeScript - Inheritance

<code>

You can inherit from another class. Use super to call the constructor.

```
class Person {
  constructor(public name: string) {…}
}

class Employee extends Person {
  constructor(name: string, public salary: number) {
    super(name);
    // …
  }
}
```

https://www.typescriptlang.org/docs/handbook/classes.html

# Decorators

# How to decorate in ES5

<code>

Decorators, or higher order functions for classes in ES5 are simple

```
function Robot(target) {
    target.isRobot = true;
}

function Number5() {...}
Robot(Number5);

Number5.isRobot; // ==> true
```

# How to decorate a ES2015/TS class

<code>

The constructor function can be notated as class

```
function Robot(target) {
    target.isRobot = true;
}

class Number5 {…}
Robot(Number5);

Number5.isRobot; // ==> true
```

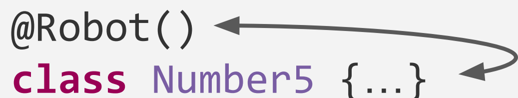
But the isRobot call belongs directly to Number5

# How to decorate in ES2018/TS

<code>

The constructor function can be notated as class

```
function Robot(target) {
    target.isRobot = true;
}
```




```
Number5.isRobot; // ==> true
```

To decorate a class just add a "@" decorator function above a class definition.

# How to decorate in ES2018/TS

<code>

Since the decorator function is just a function, it can be a Higher Order Function to get configuration parameters.

```
function Robot(roboName) {
    return function(target) {
        target.roboName = roboName;
    }
}

@Robot('Johnny 5')
class Number5 {…}
Number5.roboName; // ==> 'Johnny 5'
```

# Modules in JavaScript

# Modules - General

- organize code
- split the application into multiple files
- solve a specific problem/deal with a specific topic
- share functionalities between modules

# Modules

<code>

Imports and exports

```
// book.ts
export class Book {...}

// bookshelf.ts
import {Book} from "./book";
```

Destructuring!

# Named Export

<code>

You can export functions, objects, or primitive values from the module so they can be used by other programs with the import statement.

```
// foo.js
export myFunction;
export const foo = Math.sqrt(2);
export const MY_CONSTANT = 'MY_CONSTANT';

// bar.js
import { myFunction, foo } from ‘foo’; // names must match!
```

# Default Export

<code>

You can have multiple named exports per module but **only one default export**.

```
// foo1.js
export default class { /* ... */ };

// foo2.js
class MyClass { /* ... */ };
export default MyClass;

// bar.js
import ImportWithAnyName from ‘foo1’;
import ImportWithAnyOtherName from ‘foo2’;
```

# Renaming

<code>

You can rename an export when importing it.

```
// foo.js
export myValue = 123;

// bar.js
import {myValue as differentName} from ‘foo’;
```

# Import entire module's contents

<code>

This inserts myModule into the current scope, containing all the exports from the module in the file my-module.js.

```
import * as myModule from 'my-module.js';

myModule.importedFunction();
console.log(myModule.IMPORTED_CONSTANT);
```

# Import entire module's contents

<code>

You can import default and named exports in one statement.

```
import React, { Component, useState } from 'react';
```

# JavaScript Runtime

# Execution Model

- **Single Threaded**

  A program is executed in *only one thread*.

  (Exception: Web Worker)

- **Run-To-Completion**

  A program can't get interrupted.

# Long running tasks

after ~10 seconds

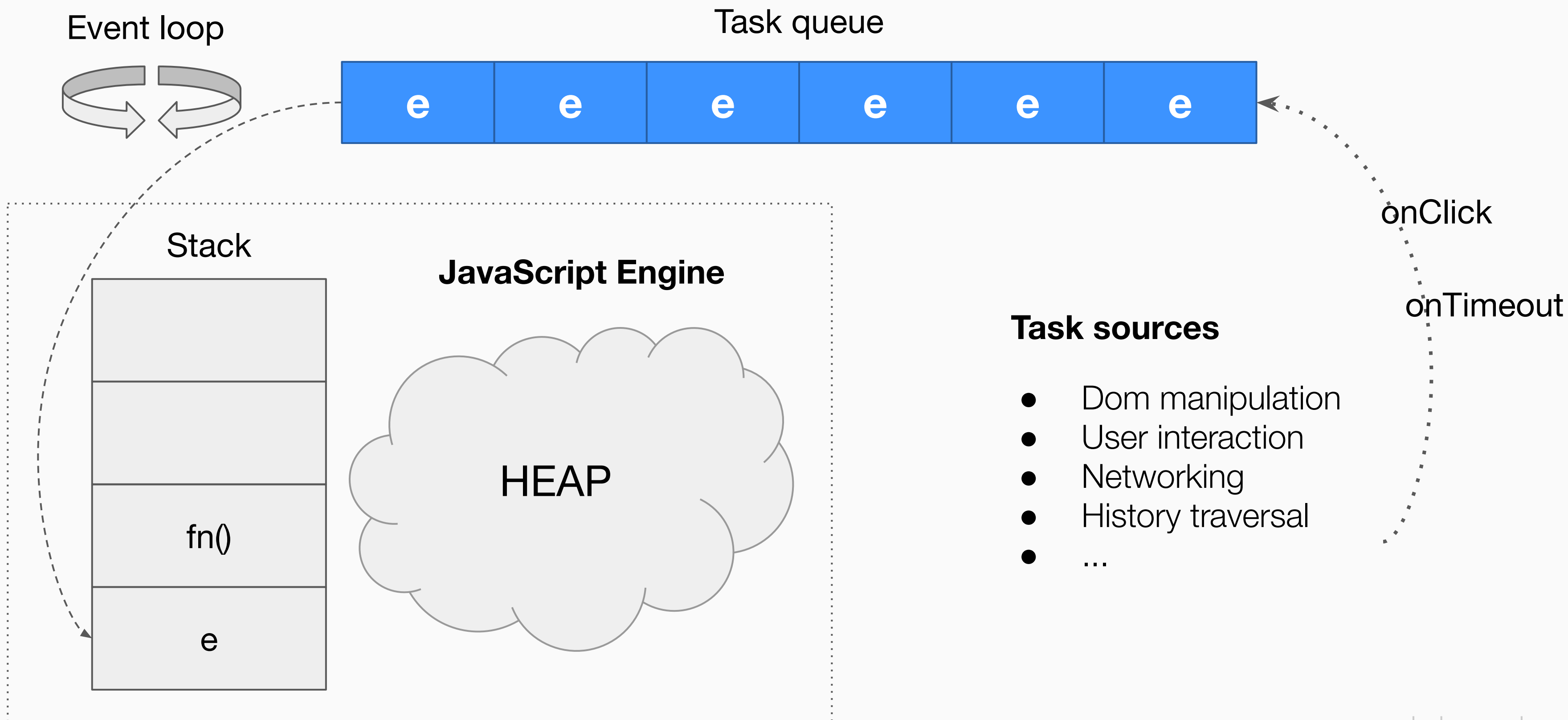
# Execution Model

```
while(true) {
  e = getNextEvent();
  executeEvent(e);
}
```

# Execution Model

workshops.de

# Execution Model



# Interfaces

# Interfaces

- Type-checking of the shape of values
- Interfaces give a type to these shapes

# Interfaces - Without an interface

<code>

You can generate interfaces on the fly.

```
let book: { isbn: string, title: string };

book = {
  isbn: '978-1593272821',
  title: 'Eloquent JavaScript'
};
```

# Interfaces - With an interface

<code>

Give an interface a name and use it as a type for variables.

```
interface Book {
  isbn: string;
  title: string;
}

const book: Book;

book = {
  isbn: '978-1593272821',
  title: 'Eloquent JavaScript'
};
```

# Interfaces - Optional properties

<code>

Properties can be optional.

```
interface Book {
    isbn: string;
    title: string;
    pages?: number;
}
```

# Interfaces - Nesting Interfaces

<code>

Interfaces can be nested

```
interface Profile {
  id: number;
  gender: string;
  name: string;
  pictureUrl?: string;
  address: {
    street: string,
    zipCode: string,
  }
  city: string
}
```

# Interfaces - Class types

<code>

Forgetting to implement ngOnInit throws a compile error.

```
interface OnInit {
  ngOnInit();
}

class BookListComponent implements OnInit {
  ngOnInit() {
  }
}
```

# TypeScript Generics

Generics allows us to define type dependencies in classes, interfaces and functions by enabling *types* to be parameters.

# Why we need generics?

- Stronger type checking at compile time
- Create (more) reusable code
- Allow other users of your methods to define their types

# TypeScript Generics

<code>

Problem: our generic data structure should push and pop items of the same type

```
class Queue {
  private data = [];
  push(item) { this.data.push(item); }
  pop() { return this.data.shift(); }
}

const queue = new Queue();
queue.push(0);
queue.push("1");
...
queue.pop().toPrecision() // Oops a mistake we'll only notice at runtime
```

# TypeScript Generics

<code>

We could add an implementation for every type we want to support

```
class QueueNumber extends Queue {
  push(item: number) { this.data.push(item); }
  pop(): number { return this.data.shift(); }
}

const queue = new Queue();
queue.push(0);
queue.push("1"); // ERROR : cannot push a string. Only numbers allowed
```

We want to express the dependence between push and pop; whatever type we push onto the queue it should be the same type that is popped.

# How to use a Generic in TypeScript?

# TypeScript Generics

<code>

Luckily we can make the type a parameter of our class definition

```
class Queue<T> {
  push(item: T) { this.data.push(item); }
  pop(): T { return this.data.shift(); }
}

const queue = new Queue<number>();
queue.push(0);
queue.push("1"); // ERROR : cannot push a string. Only numbers allowed
```

# TypeScript Generics: Constraints

<code>

Constraints can be added to generics via ‘extends’

```
class Queue<T extends Pushable> {
  push(item: T) { this.data.push(item); }
  pop(): T { return this.data.shift(); }
}
```

# How to define a generic function?

# Generic Functions

<code>

Reverse a list: what gets passed into the function should be of the same type of what gets returned

```
function reverse<T>(items: T[]): T[] {
  let result = [];
  for (let i = items.length - 1; i >= 0; i--) {
    result.push(items[i]);
  }
  return result;
}

const reversed = reverse([3, 2, 1]) // [1, 2, 3]
// Safety!
reversed[0] = '1'; // Error!
```

# Stronger type checking at compile time

# Generic Interfaces

<code>

```
interface KeyPair<T, U> {
  key: T;
  value: U;
}

let kv1: KeyPair<number, string> = { key: 1, value: ”Steve” }; // OK
let kv2: KeyPair<number, string> = { key: 1, value: 1234 }; // Error
```

# Task

## Create an Interface and use a generic function

https://www.typescriptlang.org/docs/handbook/advanced-types.html

# We teach.

workshops.de