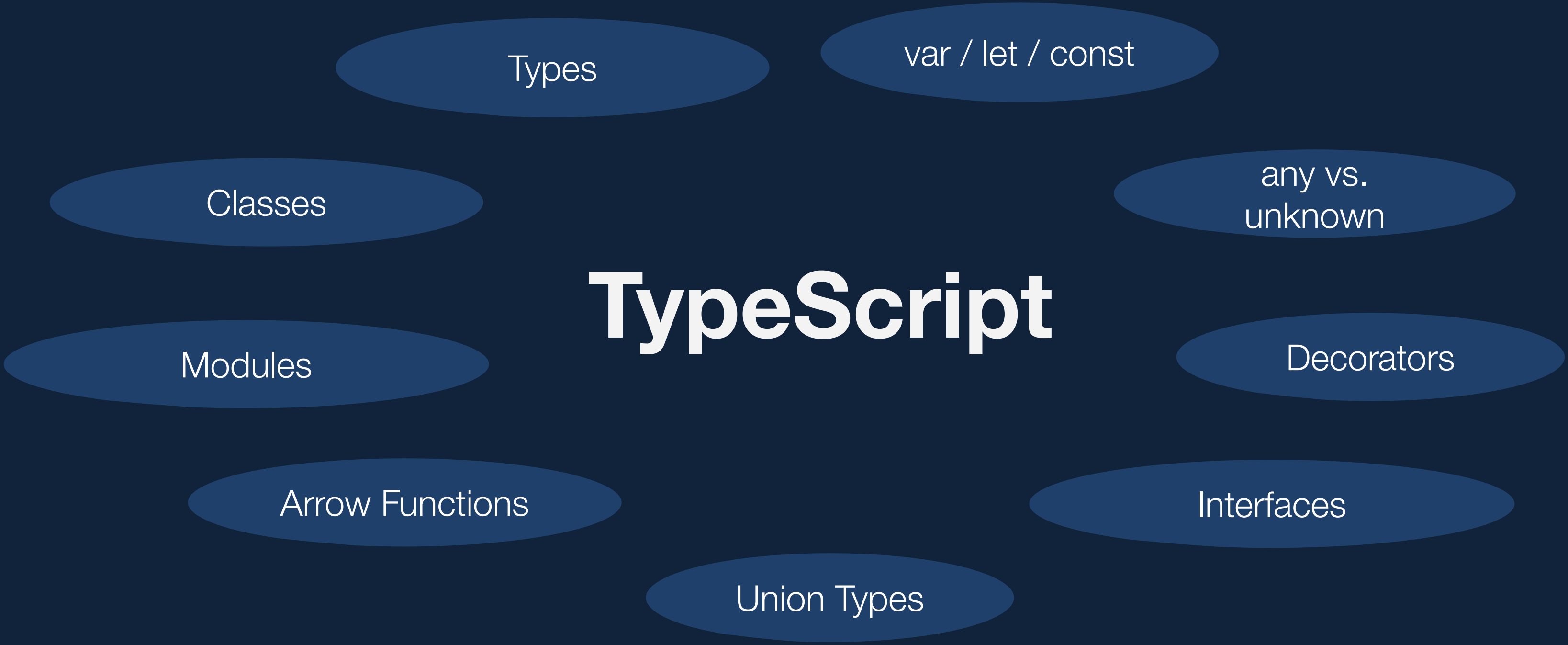
TS

Workshop

TypeScript Introduction

workshops.de

Task: Test your knowledge



TypeScript

Types

var / let / const

Classes

any vs. unknown

Modules

Decorators

Arrow Functions

Interfaces

Union Types

TypeScript

JavaScript with syntax for types

TypeScript is a **typed superset** of JavaScript

that **compiles to plain JavaScript**.

Why TypeScript

Why **TypeScript**

- Statement completion and code refactoring
- Symbol-based navigation
- Avoids simple tests `(expect(service.get).toBeDefined)`

The result: better maintenance for long-living projects

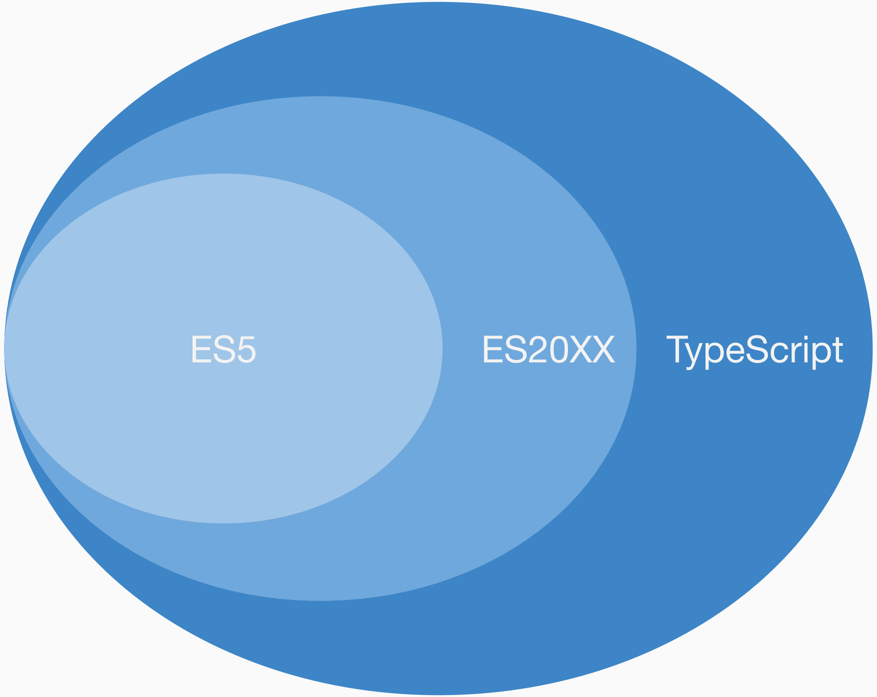
Differences
TypeScript vs ECMAScript

What is **ECMAScript?**

- Standardardization of JavaScript
- Most modern browsers support most of **ES2023** now
- **ES2024** is standardized, browser support upcoming
- We may transpile TypeScript → ES5 (which was before ES2015)

TypeScript is a superset

- → Superset of ECMAScript
- → Compiles to clean code
- → Optional Types



TypeScript or JavaScript?

Check the logo on each slide to know what we're talking about

- The line between JS/TS can seem blurry when you're just getting started
- In most cases TS only refers to static type annotations, everything else is JS
- Your first places to look something up
  - MDN JavaScript Docs
  - Official TypeScript Docs

Variables

declaration and usage

Variables - Declaration

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

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

```
var example = 1;

if (true) {
    var example = 2;
    console.log('Inside:  ' + example); // => Inside:  2
}

console.log('Outside: ' + example); // => Outside: 2
```

Scoping

let/const are block-scoped

```
const example = 1;

if (true) {
    const example = 2;
    console.log('Inside:  ' + example); // => Inside:  2
}

console.log('Outside: ' + example); // => Outside: 1
```

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

UTF-8 characters are also allowed!

```
const π = Math.PI;

const ლ_ಠ益ಠ_ლ = 42;

const T̿̿⃝̡̳H̳̃̌ = 'Zalgo';
```

Variables

Bind to the result of an expression

```
const helloWorld = 'Hello World';

const helloFunction = function() {};

const now = getCurrentTime();
```

Variables - Primitive types

Call by value

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

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

Variables - Object types

Call by reference

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

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

Variables with `const`

Reassigning throws an error

```
const birthdate = new Date();

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

Objects with `const`

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';
```

Type annotations

Specify explicit data types for variables, parameters and return types

Structured Types

Primitives

```
const isDone: boolean = true;

const size: number = 42;

const firstName: string = 'Lena';

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

Types - Any

any takes any type

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

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

Union Types

to combine multiple types into one

```
let question: string | string[];


type UnionPerson = Student | Professor;

let question: UnionPerson = ...
```

Union Types

For return types

```
function getPerson(n: number): Student | Professor {
    if (n === 1) {
        return new Student();
    } else {
        return new Professor();
    }
}
```

Template Strings

Strings - Template string

Variables in strings (multiline support!)

```
const greeting = "Hi";

const name = 'Tom';

const introduction = `${greeting}, my name
                     is ${name}!`;

// => Hi, my name
//    is Tom!
```

Objects

An **object** is an unordered collection of

key-value pairs

Objects

Object creation (equivalent behavior)

```
// object literal, recommended for inline objects
const a = {};

// object constructor, only use for derived classes
const b = new Object();
```

Objects

Object properties

```
const car = {
  manufacturer: 'Ford'
};

car.model = 'Mustang';
car['year'] = 1964;
```

Functions

Functions

“First-class citizens”, functions are just expressions

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

const url = 'https://www.google.de';
http.get(url, function() {});
```

Functions

Functions are also objects

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

Functions - Type annotations

Add types to function parameters and return values.

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

Functions - Optional parameters

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

Function arguments can have defaults for arguments.

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

buildName()
```

Functions - Rest/Spread syntax

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

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

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

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

Arrow function expressions

Shorter alternative to traditional function expressions

Arrow function expressions

Implicit return without a block

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

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

Arrow function expressions

Use braces around arguments if you have multiple parameters.

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

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

Arrow function expressions

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;
// };
```

this

Keyword for referring to the current context

Global context

`this` - Global context

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

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

`this` is set to the object itself.

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

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

this - In constructors

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
```

Arrays

Arrays

Arrays are ordered - objects are not!

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

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

Arrays - Iterators

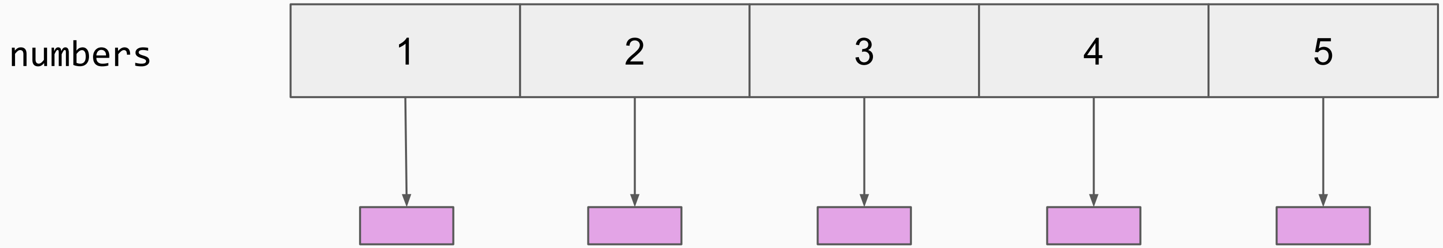
With a for and a for...of loop you have the opportunities to `break` or `continue` the loop and exit the surrounding function with `return`.

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

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

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

Arrays - Iterators

Array.forEach()

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

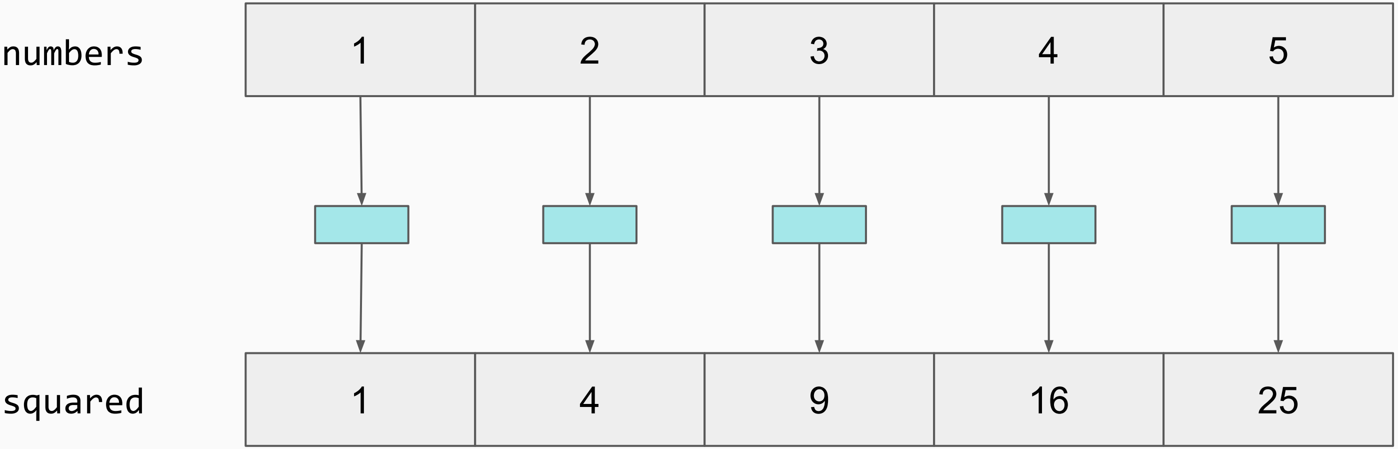


Arrays - Transformations

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



numbers	1	2	3	4	5
squared	1	4	9	16	25

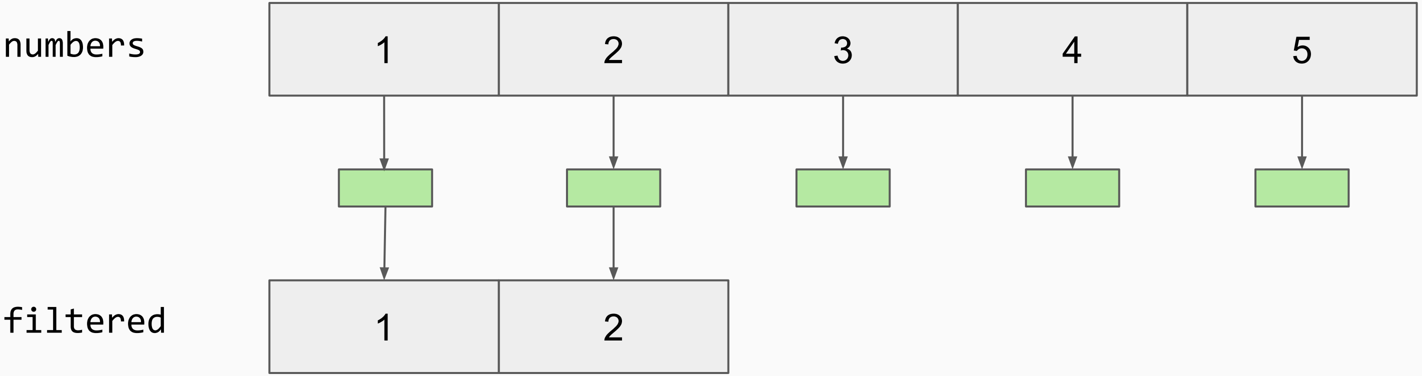
Arrays - Transformations

JS

Array.filter()

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

Filtering an array



Higher Order Functions

A famous concept in functional programming

Higher Order Functions

#1 Functions that accept a function as parameter

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

Higher Order Functions

#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

What happens with the variable after the function is terminated?

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

Closures

The result is?

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

    return () => localVar + 10;
};

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

Closures

Functions that “enclose” local variables

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

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

closure

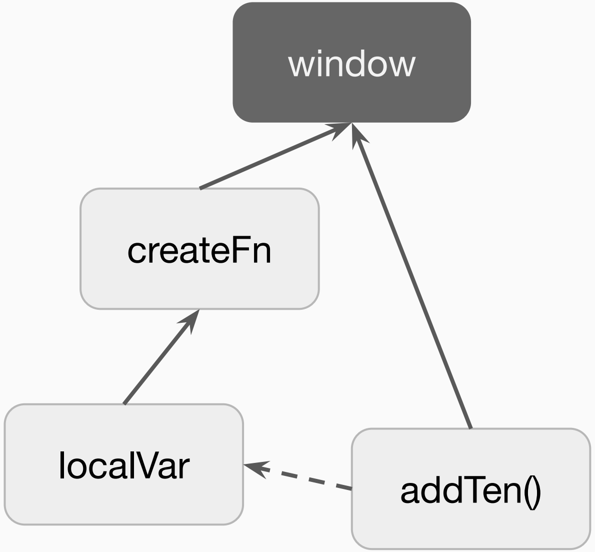
```
const addTen = createFunction();
addTen(); // 133
```

→ The inner function encloses *localVar* because it has read access to *localVar*.

→ The **inner anonymous function** is a so-called **closure**.

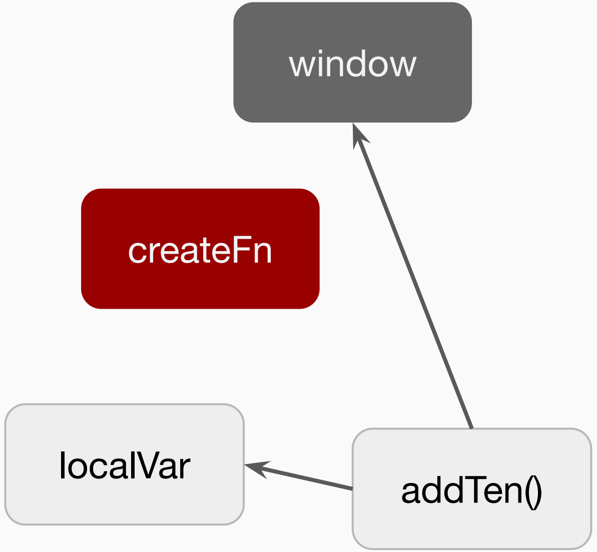
Garbage Collection

```
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

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!
```

Classes

Classes in JavaScript

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

Classes in TypeScript

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

```
class Person {
  birthDate: Date;

  constructor(birthDate: Date) {
    this.birthDate = birthDate
  }

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

Classes in TypeScript

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

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

  public name: string;

  protected bornOn: Date;

  private weight: number;
}
```

Classes in TypeScript

Declare a class property from a constructor parameter.

```
class Person {
  constructor(public birthDate: Date) {
  }

  shout(): void { alert(this.birthDate); }
}
```

Classes in TypeScript - Instances

Create new instances with the *new* keyword.

```
class Person {…}

const john = new Person(new Date());

john.birthDate; // => a Date object

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

Classes in TypeScript - Inheritance

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);
    // ...
  }
}
```

Interfaces

Interfaces

- Type-checking of the shape of objects
- Interfaces give a type to these shapes
- Only exist during development, can be violated at runtime

Interfaces - Without an interface

You can generate interfaces on the fly.

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

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

Interfaces - With an interface

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

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

let book: Book;

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

Interfaces - Optional properties

Properties can be optional.

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

Interfaces - Class types

Forgetting to implement ngOnInit throws a compile error.

```
interface OnInit {
  ngOnInit(): void;
}

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

Decorators

How to decorate in ES5

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

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 ES2015/TS

The constructor function can be notated as class

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

@Robot
class Number5 {…}

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

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

How to decorate in ES2015/TS

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'
```

Destructuring

Destructuring - Objects

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 - Arrays

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
```

Modules

Modules - General

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

Modules

Imports and exports

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

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

Destructuring!

JavaScript Runtime

Execution Model

- **Single Threaded**

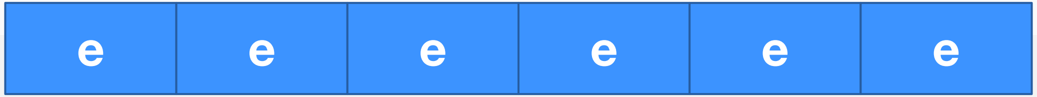
  A program is executed in *only one thread*.

  (Exception: Web Worker)

- **Run-To-Completion**

  A program can’t get interrupted.

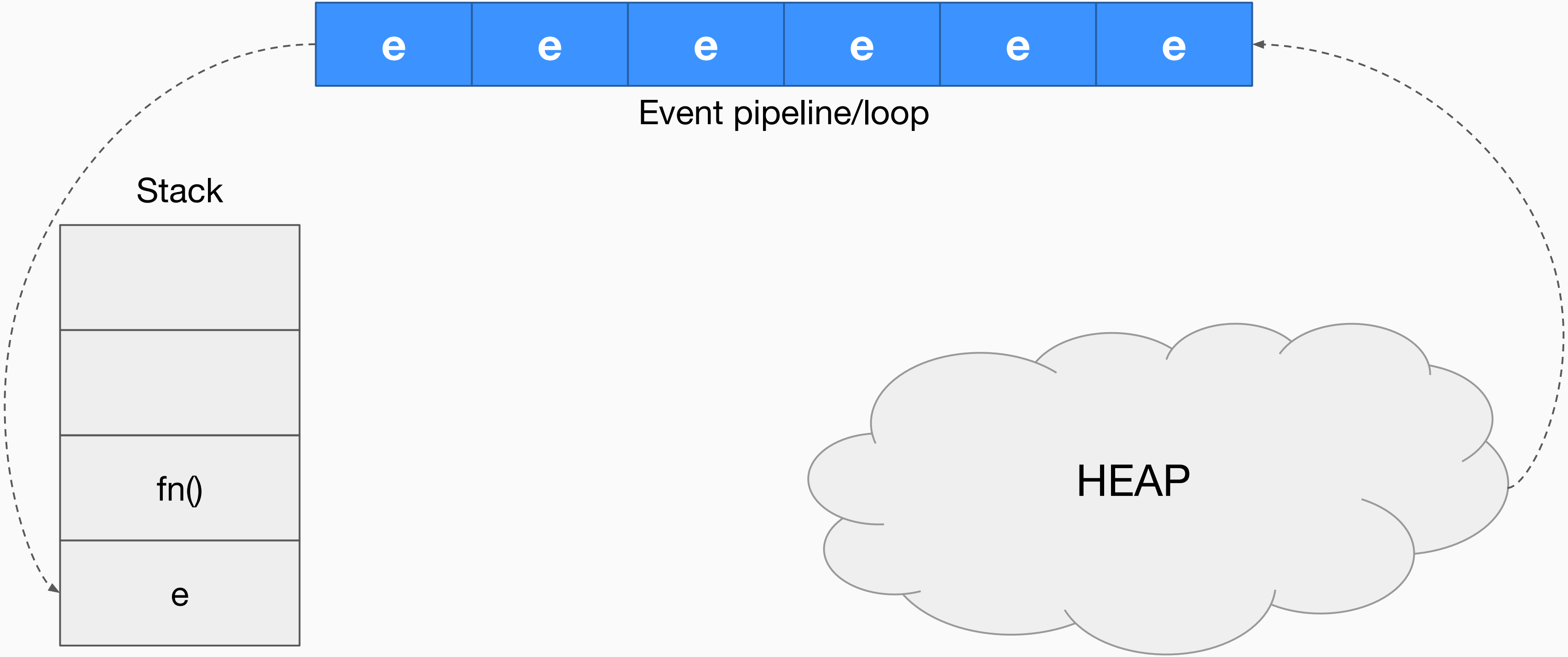
Execution Model



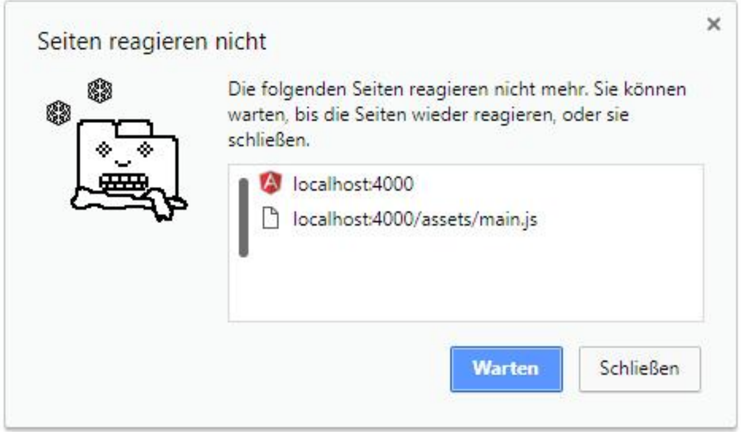
e e e e e e

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

Execution Model



Long running tasks



Seiten reagieren nicht

Die folgenden Seiten reagieren nicht mehr. Sie können warten, bis die Seiten wieder reagieren, oder sie schließen.

- localhost:4000
- localhost:4000/assets/main.js

after ~10 seconds

symetics