Bachelor of Science (BSc) in Informatik
Modul Advanced Software Engineering 1 (ASE2)

LE 05 –TypeScript

Institut für Angewandte Informationstechnologie (InIT)

Walter Eich (eicw) / Matthias Bachmann (bacn)

https://www.zhaw.ch/de/engineering/institute-zentren/init/

Lernziele TypeScript

Der Studierende

- o Kennt die verschiedene ECMA Standards
- o Kann die Programmiersprache anwenden hinsichtlich von
  - Variablen
  - Typen
  - Template Strings
  - Objekten
  - Funktionen
  - Arrays und Iterables
  - Funktionen höherer Ordnung
  - Closure (Funktionsabschluss)
  - Klassen
  - Dekorierer
  - Zerlegen (Destructing)
  - Promise
  - Enum
  - RXJS

Agenda

TypeScript

- 01-Variablen
- 02-Typen
- 03-Template Strings
- 04-Objekten
- 05-Funktionen
- 06-this
- 07-Arrays und Iterables
- 08-Funktionen höherer Ordnung
- 09-Closure (Funktionsabschluss)
- 10-Klassen
- 11-Dekorierer
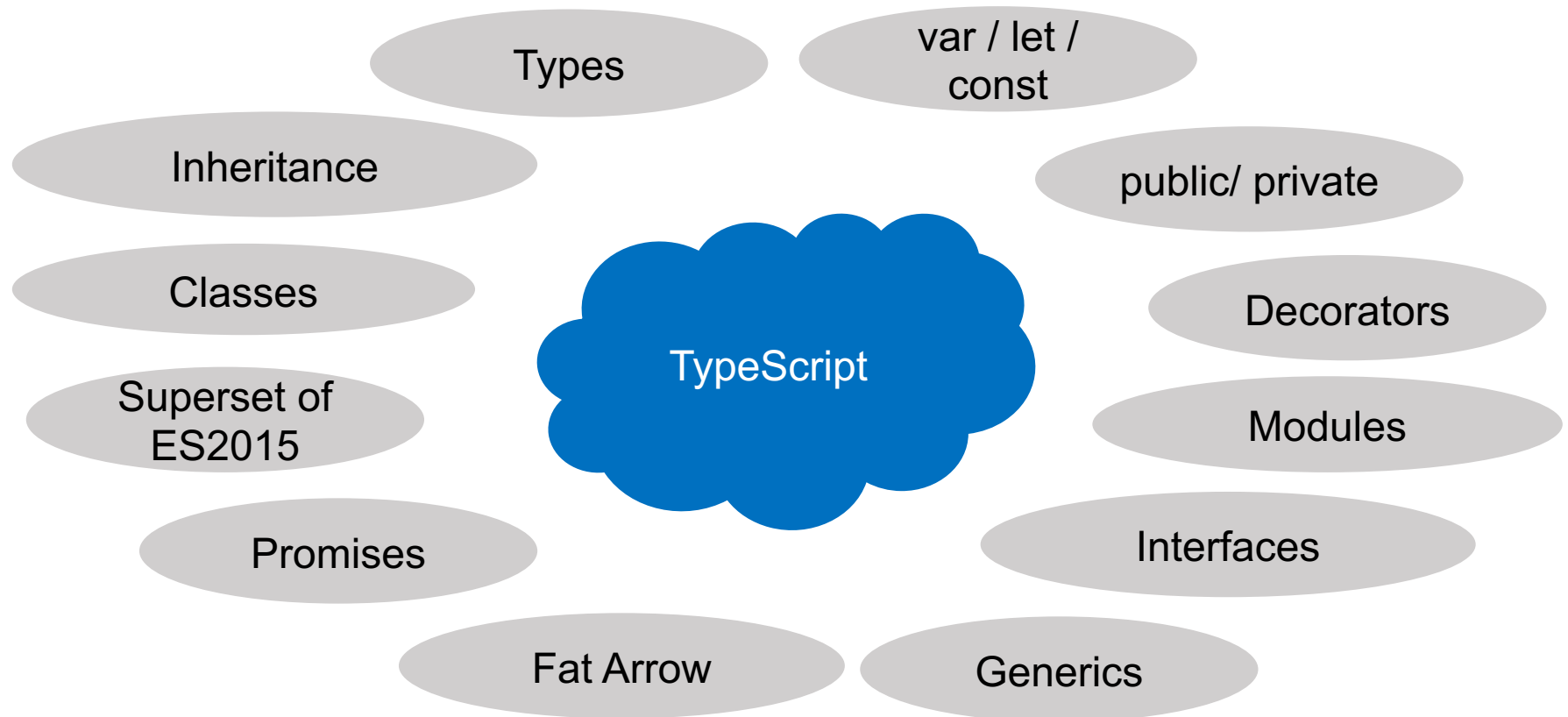- 12-Zerlegen (Destructing)
- 13-Promise
- 14-Enum
- 15-RSJS

Code Snippets:
https://github.engineering.zhaw.ch/bacn/ase2-typescript-01

Agenda Practices

Task #00   Preparation

Task #01   Object, Fat Arrow, Functions

Task #02   Date Object

Task #03   Higher Order Functions

Task #04   Promises

Task #05   RxJS

TypeScript

TypeScript in a Nutshell

Types

var / let / const

Inheritance

public/ private

Classes

Decorators

TypeScript

Superset of ES2015

Modules

Promises

Interfaces

Fat Arrow

Generics

https://www.typescriptlang.org/

TypeScript

- Writing **large applications** in JavaScript is difficult, not originally designed for large complex applications (mostly a scripting language, with functional programming constructs)

- Lacks structuring mechanisms like Class, Module, Interface

- TypeScript is a language for application scale JavaScript development.

- TypeScript is a typed superset of JavaScript that compiles to plan JavaScript.

- TypeScript adds Static Typing and structuring (class, module) to JavaScript.

22.03.21

Why TypeScript

- Statement completion and code refactoring

- Symbol-based navigation (together with IDE)

- Types give guarantees

The result: Better maintenance for long-living projects

History of Typescript

Typescript was first made public in October 2012 (at version 0.8), after two years of internal development at Microsoft.

TypeScript 0.9, released in 2013, added support for generics

TypeScript 1.0 was released at Build 2014

TypeScript 2.0 was released 2016

TypeScript 3.0 was released 2018

TypeScript 4.0 was released 2020

Current Release 4.2.x (März 2021)

http://www.typescriptlang.org/play/

What is ECMAScript? (1)

- ○ Standardization of JavaScript
  ES1: June 1997—*ES2*: June 1998—*ES3*: Dec. 1999—*ES4*: Abandoned
  ES5: December 2009

- ○ Most modern browsers support most of **ES2015** (ES6) now

- ○ **ES2016** – **ES2020** (EcmaScript 7 - EcmaScript 11)

- ○ We transpile TypeScript to JavaScript Version:
  ES5 (2009) or ES6 (2015)

What is ECMAScript? (2)

https://www.ecma-international.org/
https://www.w3schools.com/js/js_versions.asp

- ECMAScript:
  - A language standardized by ECMA This term is usually used to refer to the standard itself.

- JavaScript:
  - The commonly used name for implementations of the ECMAScript standard. This term isn't tied to a particular version of the ECMAScript standard.

- ECMAScript 5 (ES5):
  - The 5th edition of ECMAScript, standardized in 2009. This standard has been implemented fairly completely in all modern browsers

- ECMAScript 6 (ES6) / ECMAScript 2015 (ES2015):
  - The 6th edition of ECMAScript, standardized in 2015. This standard has been partially implemented in most modern browsers.

- ECMAScript 2016/2017/2018/2019/2020:
  - The 7th, 8th, 9th, 10th, 11th edition of ECMAScript.

TypeScript is a superset

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

ES5 ES2020 TypeScript
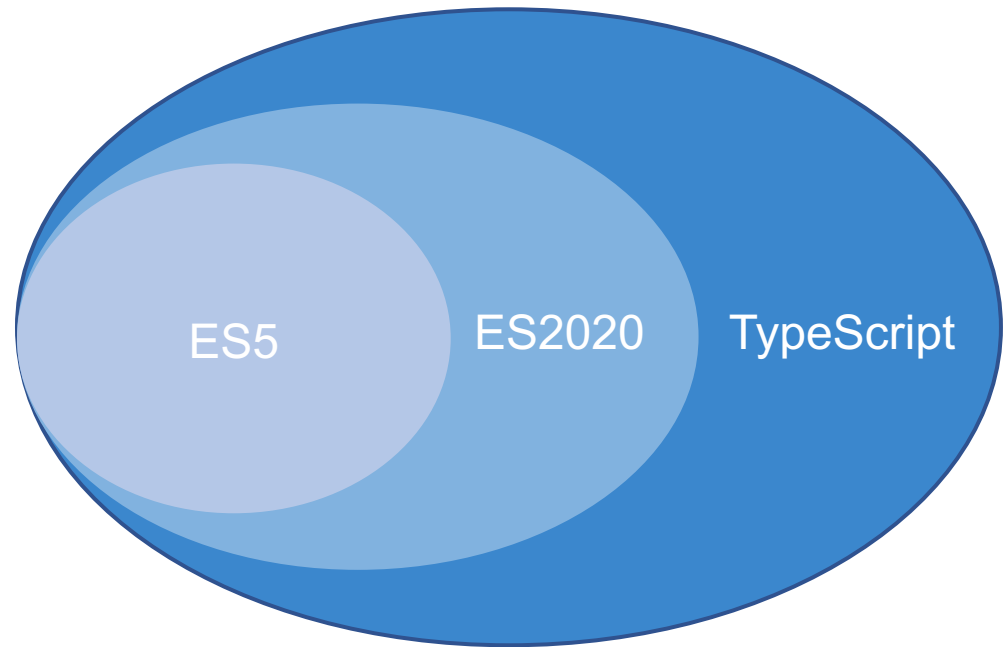
Task #00
Preparation

01 - Variables

declaration and usage

Variables - Declaration

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

```
var value;                          … forbidden in typescript


const pi = 3.1416;


let $value__123;
```

Variables - Naming

Almost all arbitrary names

Exceptions:

- **no** whitespace

- **not** starting with a number

- **no** dashes

- **no** JS keywors (e.g.typeof etc.)

Variables – Fun fact

UTF-8 characters are also allowed!

```
let π = Math.PI;  // nicht gültig für ES5, gültig for ES2016

let ಠ_ಠ = Math.PI;



let ಲ_ಠ益ಠ_ಲ = 42;
```

Variables

Hold the result of an expression

```javascript
let helloWorld = 'Hello World';

let helloFunction = function() {};

let returnValue = getCurrentTime();
```

Call by value

```
let a = 'Hello World';

let b = a; // Only value is copied

c = 4;



console.log(b);

// => 'Hello World'
```

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]
```

02 - Types

first look on TypeScript

Types in TypeScript - Variables

Types exist for primitive and object types.

```typescript
let isDone: boolean = true;

let size: number = 42;

let firstName: string = 'Lena' ;

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

Types - Any

Any **type takes any type**

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


question = 6 * 7;

question = false;
```

Scoping

let is block-scoped

```javascript
let example = 1;

if (true) {

    let example = 2;

    console.log('Inside:  ' + example);

}


console.log('Outside:  ' + example);
// => Inside:  2
// => Outside: 1
```

Variable Scope

`let` is the new `var`

	var	**let**	**const**
Scope	function	block	block
Value changeable	✓	✓	✕
Standard	since ever	ES2015 / TS	ES2015 / TS

Variables with const

Reassigning throws an error

```
const dateOfBirth = new Date();

dateOfBirth = new Date(); // compile error!
```

const with objects

Only the reference immutable.

```
const myObject = {
  name: 'Florian' ,
  dateOfBirth: '1985-08-04'
};


// Object is mutable!
myObject.name = 'Andreas' ;


// but you cannot change the reference
myObject = {name: 'Peter'}; // this throws an error!
```

03 - Template Strings

Strings – Template string

Variables in strings (multiline support)

```
const name = 'Felix Muster';


const temp = `My name is ${name}`;    // Backtick


// => My name is Felix Muster
```

Objects

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

Objects

object creation (equivalent behavior)

```
let a = {};

let b = new Object();
```

Objects

Object properties

```javascript
let car = {

  make: 'Ford',

  model: 'none'

};


car.model = 'Mustang';

car['year'] = 1969;
```

05 - Functions

Functions - JavaScript

"First-class citizens", functions are just expressions

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

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

Functions - JavaScript

Functions are also objects

```javascript
let fn1= function() {

    window.alert('Hello JavaScript');

};

fn1['foo'] = 'bar';
```

Functions
Named and Anonymous

JavaScript has both, named and anonymous functions.

```
function namedFunction() {

    console.log("named function");

}
```

Named functions (function declarations) are usable even before the line they are declared. The declaration of the function is hoisted to the beginning of the function scope.

```
let anonymousFunction = function() {

    console.log("anonymous function.")

}
```

Anonymos functions (function expressions) normal expressions. There is no magic hoisting happening.

Functions - Types

Add types to functions arguments and return values.

```
function sayHi(firstName: string): void {

    console.log(firstName);

}
```

Functions – Optional parameters

Parameter can be optional. Use a question mark.

```typescript
function buildName(firstName: string, lastName?: string) {

  if (lastName) {

    return firstName + ' ' + lastName;

  } else {

    return firstName;

  }

}
```

Functions – Default parameters

Function arguments can have defaults for arguments.

```typescript
function buildName(firstName: string, lastName = 'Adams') {

  // type Inference: lastName is a string

return firstName + ' ' + lastName;

}
```

Functions – Rest/Spread parameter

An abitrary amout of parameters can be stored in an array.

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


  let allNames = [firstName, ...restOfNames];

  // names = [firstName, restOfName[0], restOfName[1] ...]


  return allNames.join(' ');

}
```

Fat Arrow

=>

Functions- Fat-Arrow-Function

Implicit return without a block

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

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

console.log(square(2));
```

Functions- Fat-Arrow-Function

Use braces around arguments if you have multiple parameters.

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

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

console.log(sum(2,3));
```

Functions- Fat-Arrow-Function

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

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

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

Task #02
Date

06 - this

Global context

this – Global context

In a global execution context (outside of any function), **this** refers to the global object. In browsers this is *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 object

this is set to the object itself.

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


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

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

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

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

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

https://www.w3schools.com/js/js_object_prototypes.asp

07 - Arrays and Iterables

Arrays

Arrays are ordered – objects not!

```javascript
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.

```javascript
var names = ['Hanni', 'Nanni'];


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

}


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

}
```
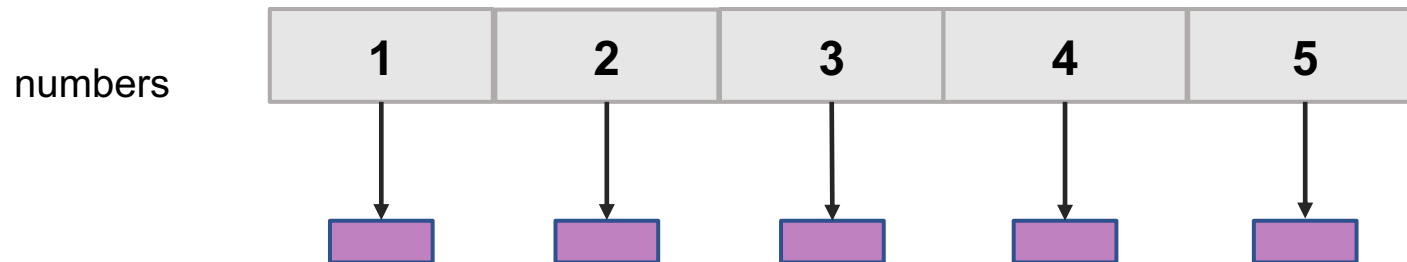
Arrays - Iterators

Array.forEach()

```
const myArray = [1,2,3,4,5];

myArray.forEach(elem => console.log(elem));
```
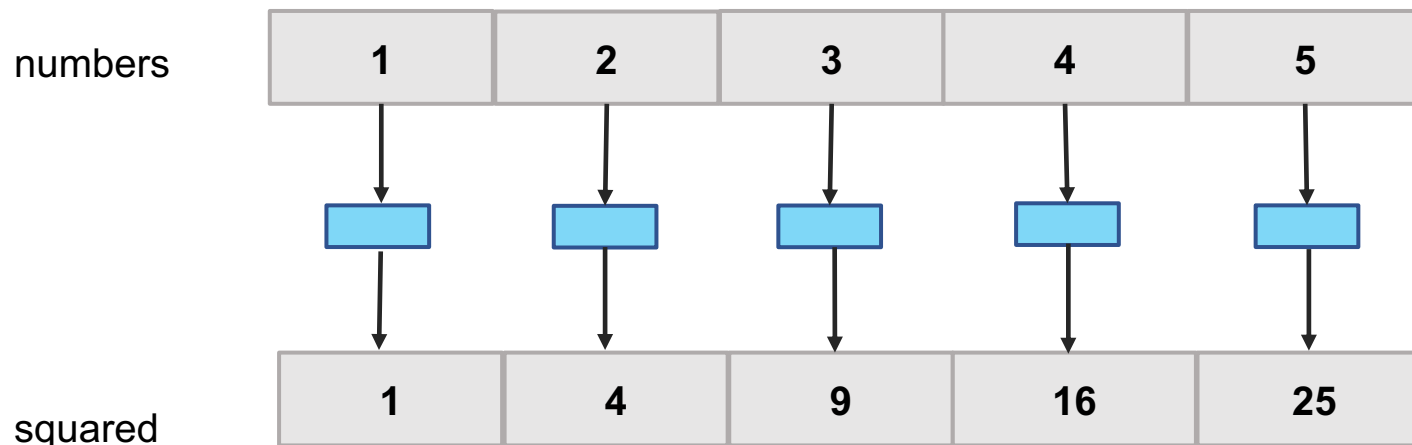
numbers

1	2	3	4	5

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]
```
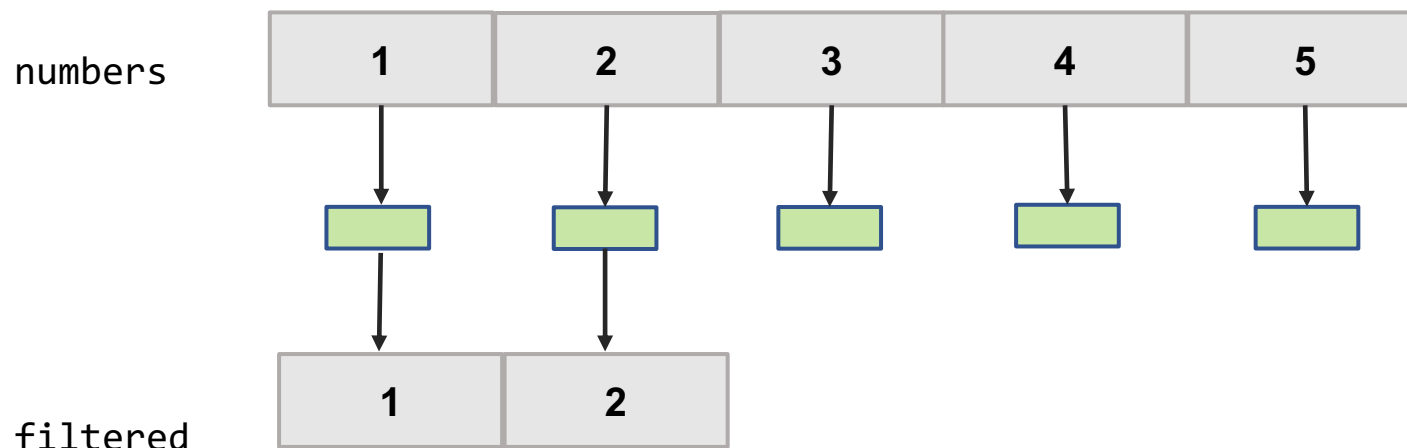
Transforming an array

Array.filter()

```
const numbers = [1, 2, 3, 4, 5];

const filtered = numbers.filter(num => num < 3);

// filterd is [1, 2]
```

Filtering an array

numbers

filtered

08 - 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!');

});
```

2. Function that return a function

```
let createAdder = function() {
    return function(a, b) {
        return a + b;
    };
};


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

09 - CLOSURES

Closures

What happens with the variable after the function is terminated?

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

The result is?

```
let createFunction = function() {
    let localVar = 123;

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


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

Closures

Functions that «enclose» local variables

```
let createFunction = function() {

    let localVar = 123;


    return function() {

        return localVar + 10;

    };

};                      closure


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

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

- The **inner anonymous function** is a so-calles **closure**.

Task #03

Higher order functions

10 - Classes

Code is more readable

Syntactic sugar over prototype-based inheritance

Not introducing a new object-oriented inhenritance model

Classes in TypeScript

Class can have a constructor, attributes and methods.

```typescript
class Person {
  bornOn: Date;

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

  shout(): void {}
}
```

Classes in TypeScript

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

```typescript
class Person {
  bornOn: Date; // public by default
  public name: string;
  private weight: number;
}
```

Classes in TypeScript - Instances

Create new instances with the *new* keyword.

```typescript
class Person {…}

const john = new Person('John');

john.bornOn; // => a Date object

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

Classes in TypeScript - Inheritance

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

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


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

11 - ES2017/TS Decorators

Just a higher-order function
for classes, methods, attributes and parameter

What is a Decorator?

A Decorator is a special kind of declaration that can be attached to

a class declaration,

method,

accessor,

property, or parameter.

Decorators use the form @expression, where expression must evaluate to a function that will be called at runtime with information about

How to decorate in ES5

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

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


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


Number5['isRobot'] // ==> true
```

How to decorate a ES2015/TS class

The constructor function is can be notated as class

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


class Number5() {…}
Robot(Number5);


Number5['isRobot'] // true
```

But the isRobot **call belongs directly to** Number5

The constructor function is can be notated as class

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


@Robot()
class Number5() {…}


Number5['isRobot'] // ==> true
```

To decorate a class just add a "@" in front of the decorator function above a class definition.

How to decorate in ES2016/TS

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

```javascript
function MyRobot(roboName) {

    return function(target) {

        target.roboName = roboName;

        target.isRobot = true;

    }

}
@MyRobot('Bender')
class Number6 {…}


console.log(Number6['isRobot']); // ==> true
console.log(Number6['roboName']); // ==> Bender
```

12 - Destructing

Destructuring - Objects

Get multiple local variables from an object with destructuring.

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

let {x, y} = circle;

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

Destructuring - Arrays

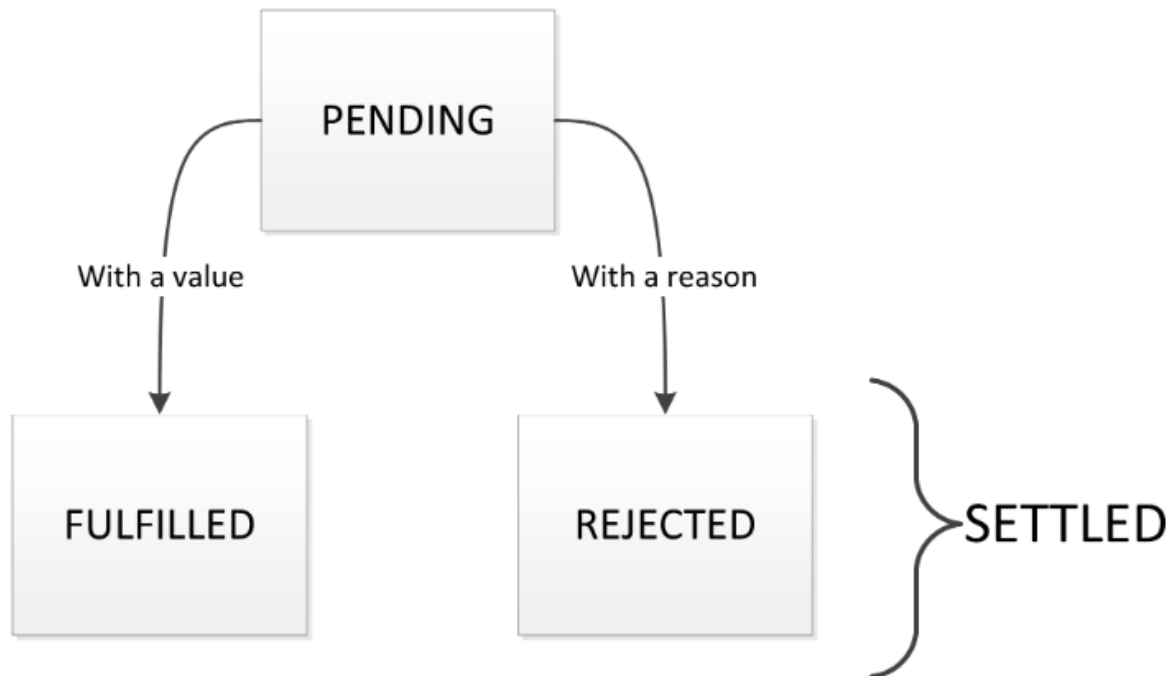**Get multiple local variables from an object with destructuring.**

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

let [lat, lng] = coords;

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

13 - Promises

Why promises

The main motivation for promises is to bring synchronous style error handling to Async / Callback style code.

A promise can be either pending or fulfilled or rejected.

Creating a promise

It's a simple matter of calling **new** on **Promise** (the promise constructor). The promise constructor is
passed **resolve** and **reject** functions for settling the promise state:

```typescript
const promise = new Promise((resolve, reject) => {
    // the resolve / reject functions control the fate of the promise

});
```

TypeScriptt

Subscribing a Promise

The promise can be subscribed to using **.then** (if resolved) or **.catch** (if rejected). Catch is never called.

```typescript
const promise = new Promise((resolve, reject) => {
    resolve(123);
});


promise.then((res) => {
    console.log('I get called:', res === 123); // I get called: true
});


promise.catch((err) => {
    // This is never called
});
```

Subscribing a Promise

The promise can be subscribed to using **.then** (if resolved) or **.catch** (if rejected). Then is never called.

```typescript
const promise = new Promise((resolve, reject) => {
    reject(new Error("Something awful happened" ));
});


promise.then((res) => {
    // This is never called
});


promise.catch((err) => {
    console.log('I get called:', err.message);
    // I get called: 'Something awful happened'
}
```

TypeScriptt

Chain-ability of Promises

The chain-ability of promises **is the heart of the benefit that promises provide**. Once you have a promise, from that point on, you use the **then** function to create a chain of promises.

```
Promise.resolve(123)
    .then((res) => {
        console.log(res); // 123
        return 456;
    })
    .then((res) => {
        console.log(res); // 456
        return Promise.resolve(123); // Notice that we are returning a Promise
    })
    .then((res) => {
        console.log(res);
        // 123 : Notice that this `then` is called with the resolved value
        return 123;
    })
```

TypeScriptt

Chain-ability of Promises

Only the relevant (nearest tailing) catch is called for a given error (as the catch starts a new promise chain).

```
Promise.resolve(123)
    .then((res) => {
        throw new Error('something bad happened');
        // throw a synchronous error
        return 456;
    })
    .catch((err) => {
        console.log('first catch: ' + err.message); // something bad happened
        return 123;
    })
    .then((res) => {
        console.log(res); // 123
        return Promise.resolve(789);
    })
    .catch((err) => {
        console.log('second catch: ' + err.message); // never called
    })
```

Use async and await to wait til the Promise finished

```
const runAsync = async () => {

    await getRandomPromise().then …

    console.log ('Promise finished');

}
```

Parallel Promises

you might potentially want to run a series of async tasks and then do something with the results of all of these
tasks. **Promise** provides a static **Promise.all** function that you can use to wait for **n** number of promises to complete.

```typescript
function loadItem(id: number): Promise<{ id: number }> {

    return new Promise<number>((resolve) => { …



Promise.all([loadItem(1), loadItem(2)])
    .then((res) => {
        [item1, item2] = res;
        console.log('done');
    }
```

Promise Race (wait for the first Promise)

Sometimes, you want to run a series of async tasks, but you get all you need as long as any one of these tasks is settled. **Promise** provides a static **Promise.race** function for this scenario.

```typescript
let task1 = new Promise(function(resolve, reject) {
    setTimeout(resolve, 1000, 'one');
});


let task2 = new Promise(function(resolve, reject) {
    setTimeout(resolve, 2000, 'two');
});

Promise.race([task1, task2]).then(function(value) {
    console.log(value); // "one"
    // Both resolve, but task1 resolves faster
});
```

TypeScriptt

Task #04
Promise

14 - Enum

Enum

Enum is a way of giving more friendly names to sets of numeric values. By default, enum begins numbering at 0. You can change this by manually setting the value of one of ist member.

```
enum Color {
    red, green, blue
}



enum Color {
    red= 100, green = 101, blue=102
}


let blue = Color.blue;
```

15 - rxjs

Observable

An Observable object can contain 0 to unlimited number of values beween now and the end of time

```typescript
let observable = new Observable((observer:Observer<string>) => {});


function pipeToLowerCase(observable: Observable<string>) {
    observable.pipe(
        map((s: string) => s.toLowerCase())
    ).subscribe(s => console.log(s));
}
```

22.03.21

Operators

Methods that perform calculations on the values

There many RxJS operators such as:

tap(),

map(),

filter(),

concat(),

share(),

retry(),

catchError(),
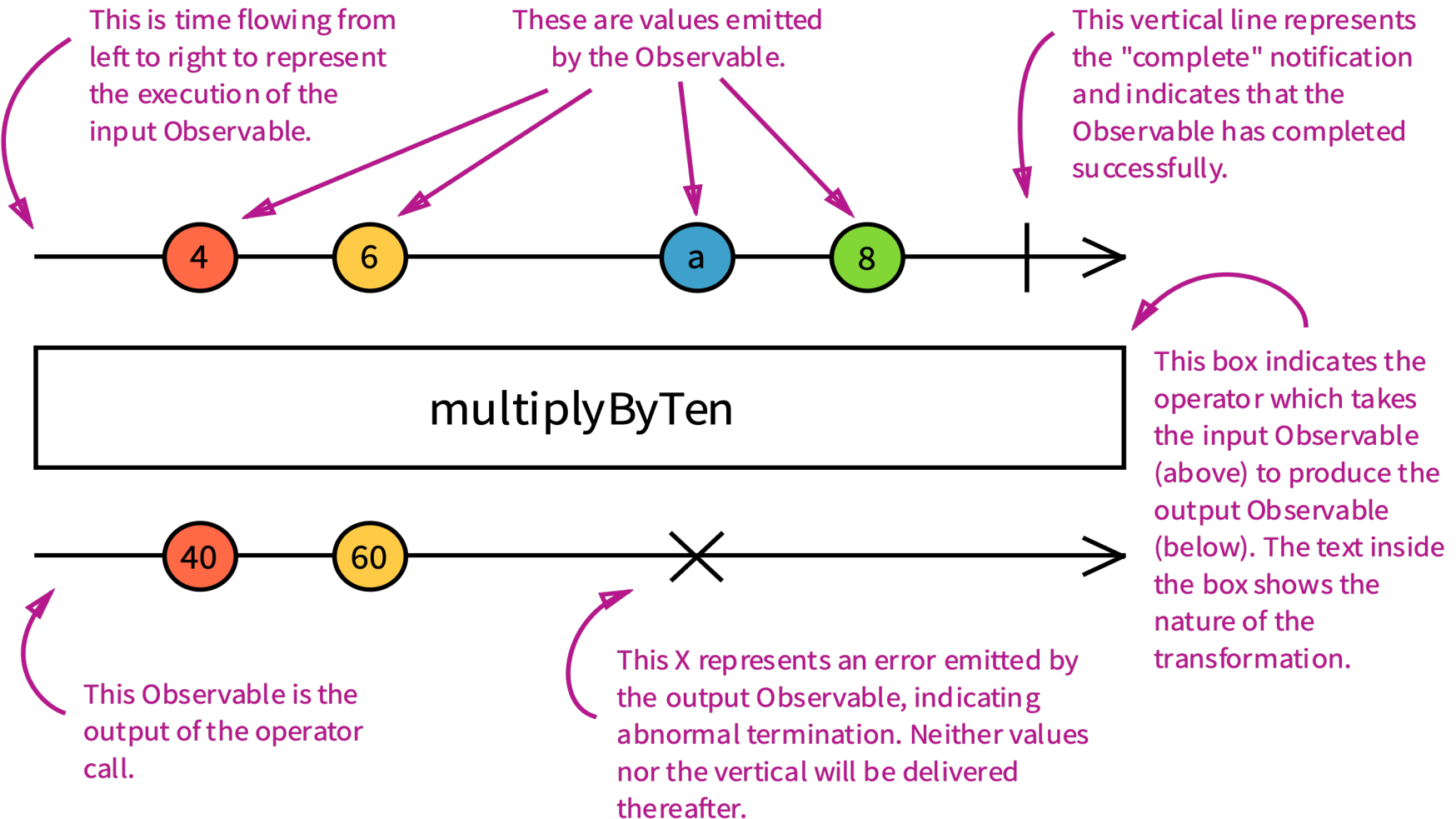
switchMap(),

and flatMap() etc.
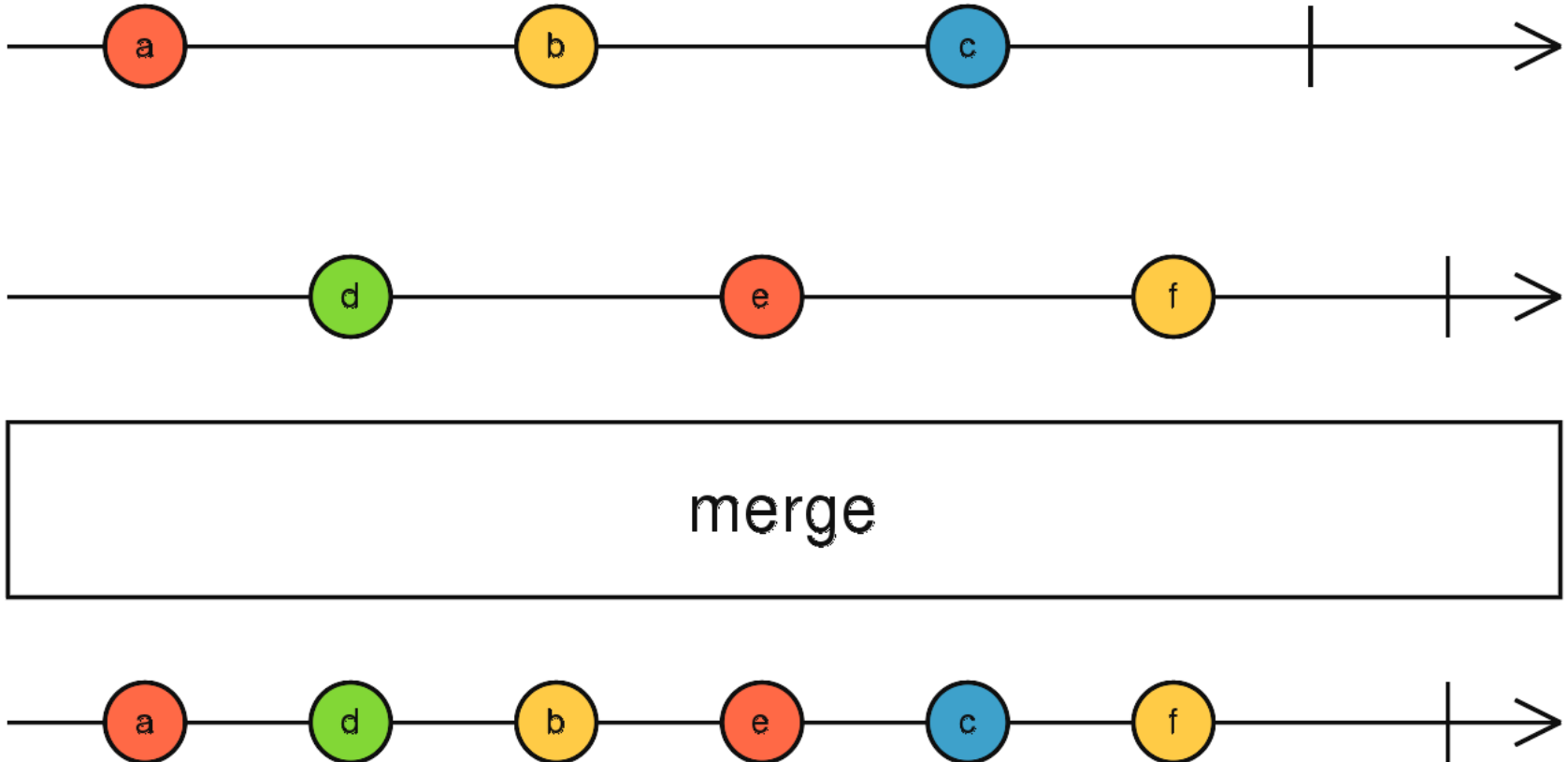
Categories

- o Creation
- o Transformation
- o Filtering
- o Combination
- o Multicasting
- o Error Handling
- o Utility
- o Conditional & Boolean
- o Mathematical and Aggregate

Understanding Marble Diagrams

The official documentation uses marble diagrams to help you understand how a given operator modifies an observable.

This is time flowing from left to right to represent the execution of the input Observable.

These are values emitted by the Observable.

This vertical line represents the "complete" notification and indicates that the Observable has completed successfully.

4   6   a   8 |

multiplyByTen

This box indicates the operator which takes the input Observable (above) to produce the output Observable (below). The text inside the box shows the nature of the transformation.

40   60   ✕

This Observable is the output of the operator call.

This X represents an error emitted by the output Observable, indicating abnormal termination. Neither values nor the vertical will be delivered thereafter.

Example Merge

TypeScript

Example Merge

```
import { Observable } from "rxjs/Observable";
import { merge } from 'rxjs/observable/merge';

let observable = Observable.create((observer:any) => {
    observer.next('Hey guys!')
})

let observable2 = Observable.create((observer:any) => {
    observer.next('How is it going?')
})

// We're using our merge operator here:
let newObs = merge(observable, observable2);

newObs.subscribe((x:any) => addItem(x));

// Our handy function for showing the values:
function addItem(val:any) {
    let node = document.createElement("li");
    let textnode = document.createTextNode(val);
    node.appendChild(textnode);
    document.getElementById("output").appendChild(node);
}
```

RxJS Marbles

The web page RsJS Marbles contains interactive diagrams:

https://rxmarbles.com/

CREATION OBSERVABLES

CONDITIONAL OPERATORS

COMBINATION OPERATORS

FILTERING OPERATORS

MATHEMATICAL OPERATORS

TRANSFORMATION OPERATORS

UTILITY OPERATORS

TypeScriptt

RxJS provides two versions of the **pipe()** function: A standalone function and a method on the **Observable** interface.

```
import { filter, map } from 'rxjs/operators';


const squareOf2 = of(1, 2, 3, 4, 5,6)

 .pipe(

     filter(num => num % 2 === 0),

     map(num => num * num)

  );

  squareOf2.subscribe( (num) => console.log(num));
```

22.03.21

Custom pipeable operators

Easily create your own operator

```typescript
const pow = (p: number) =>
    (source: Observable<number>) =>
        source.pipe(map(n => n ** p));
```

22.03.21

Custom pipeable operators

Just call the custom pipe operator like any other

```typescript
const pow = (p: number) =>
    (source: Observable<number>) =>
        source.pipe(map(n => n ** p));




source$.pipe(
  filter(x => x > 100),
  pow(3),
).subscribe(x => console.log(x));
```

Importing in v6

Import from previous versions has changed

```
import { interval, of } from 'rxjs';
import { filter, mergeMap, scan } from 'rxjs/operators';

interval(1000).pipe(
  filter(x => x % 2 === 0),
  mergeMap(x => of(x + 1, x + 2, x + 3)),
  scan((s, x) => s + x, 0),
).subscribe(x = console.log(x));
```

An exhaustive list of v6 import sites

rxjs

rxjs/operators

rxjs/testing

rxjs/webSocket

rxjs/ajax

The two you care about

- **rxjs**
  - **Types**: Observable, Subject, BehaviorSubject, etc.
  - **Creation methods**: fromEvent, timer, interval, delay, concat, etc.
  - **Schedulers**: asapScheduler, asyncScheduler, etc.
  - **Helpers**: pipe, noop, identity, etc

- **rxjs/operators**
  - **All operators**: map, mergeMap, takeUntil, scan, and so one.

Install the library from npmjs

```
> yarn add rxjs
> yarn add rxjs-compat


or with npm

> npm install rxjs --save
```

Task #05
RxJS