

# **FRONTEND DEVELOPMENT WINTERSEMESTER 2020**



# ES NEXT

# DESTRUCTURING





# DESTRUCTURING

## ARRAY DESTRUCTURING ASSIGNMENT

» makes it possible to unpack values from arrays

```
const [a, b] = [1, 2]  
console.log(a) // 1  
console.log(b) // 2
```

# DESTRUCTURING

## OBJECT DESTRUCTURING ASSIGNMENT

» makes it possible to unpack values from arrays

```
const { a, b } = { a: 1, b: 2 }  
console.log(a) // 1  
console.log(b) // 2
```

# DESTRUCTURING

## OBJECT DESTRUCTURING RENAMING

```
const { a: otherA, b: otherB } = { a: 1, b: 2 }  
//      ^^^^  
// rename value a to otherA
```

```
console.log(otherA) // 1  
console.log(otherB) // 2
```

# SPREAD OPERATOR

» adds the rest syntax to destructuring

```
const [a, b, ...rest] = [1, 2, 3, 4]  
console.log(rest) // [3, 4]
```

```
const { a, b, ...rest } = { a: 1, b: 2, c: 3, d: 4 }  
console.log(rest) // { c: 3, d: 4 }
```



# SPREAD OPERATOR COMPOSITION

```
const [, { b: otherB }, ...rest] = [{ a: 1 }, { b: 2 }, { c: 3 }]
// 1) ^^
// 2)      ^^^^^^^^^^^
// 2)              ^^^^^^^

// 1) ignore the first value
// 2) extract value b and rename to otherB
// 3) get all other elements
// recommendation don't overuse nested destructuring

console.log(otherB) // 2
console.log(rest)  // [{ c: 3 }]
```

A roller coaster track with a car at the bottom, overlaid with a red tint.

# DESTRUCTURING AND FUNCTIONS

# DESTRUCTURING AS NAMED ARGUMENTS

» can be used in functions for named arguments

```
const myFunction = ({ a, b }) => {  
  //  
  // destructure the parameters  
  return a + b  
}
```

```
myFunction({ a: 1, b: 2 })  
myFunction({ b: 2, a: 1 })  
//  
// order of arguments does not matter anymore
```

# DESTRUCTURING OF TUPLES

» can be used to destructure tuples as well

```
const myFunction = ([ a, b ]) => {  
  //      ^^^^  
  // assign variable names to each value  
  return a + b  
}
```

```
myFunction([ 1, 2 ])  
//      ^^^^  
// order of arguments matters
```



# DESTRUCTURING OF TUPLES

» I only use tuple destructuring with `Promise.all` <sup>6</sup>

```
Promise.all([
  fetchAsPromise(`/api/currentUser`),
  fetchAsPromise(`/api/weather`)
]).then(([ currentUser, weather ]) =>
//      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
//  destruct each value of the promise
  console.log(currentUser)
  console.log(weather)
})
```

<sup>6</sup> personal tip

# DESTRUCTURING OF TUPLES

```
const [ currentUser, weather ] = await Promise.all([  
  // 1)                                     ^^^^  
  // 2)      ^^^^^^^^^^^^^^^^^^^^^^^^^  
    fetchAsPromise( `/api/currentUser` ),  
    fetchAsPromise( `/api/weather` )  
  ])
```

```
// 1) await the promise  
// 2) assign result to variables
```



# OOP AND JS

# OOP AND JS

## CLASS BASED OOP

- » A class is like a blueprint – a description of the object to be created.
- » class: plan for a house
- » object: the actual house



# OOP AND JS

- » JS has a simple object based paradigm
- » An object is a collection of properties
- » A property is an association between a name and a value
- » Objects can be linked together, via prototypes

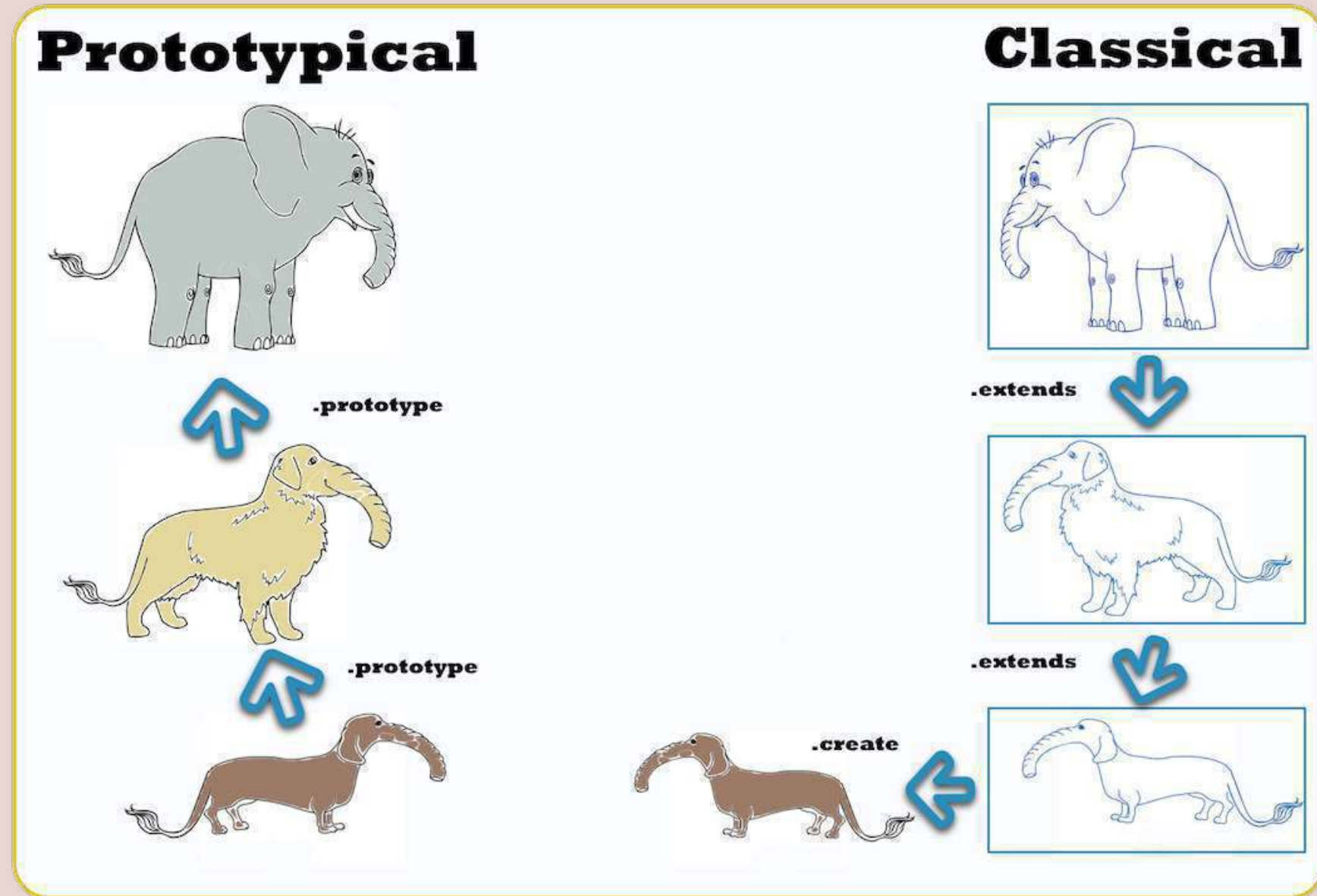
# OOP AND JS PROTOTYPAL INHERITANCE

“A prototype is a working object instance. Objects inherit directly from objects”

# OOP AND JS PROTOTYPAL INHERITANCE

- » Prototypal inheritance is delegation
  - » You ask a friend for a pen
  - » Your friend does not have a pen but asks his neighbor
  - » This chain goes on until you either have a pen or none of your related friends have a pen
- » This could be seen as the prototype chain

# OOP AND JS<sup>5</sup>



<sup>5</sup> image from medium



# OOP AND JS

## CREATE OBJECT INSTANCES

```
function University(name) {  
  this.name = name  
  //^^^^  
  // define an instance variable  
}  
  
University.prototype.isBestUniversity = function() {  
  //          ^^^^^^^  
  // you need to use function here, as {} -> {} don't support `this`  
  return this.name === 'FHS'  
  //          ^^^^  
  // prototype is able to access instance variables  
}  
  
const fhs = new University('FHS')  
fhs.isBestUniversity() // true
```

# OOP AND JS WITH CLASS SYNTAX

» Emulates class based oop with prototypes

```
class University {  
  constructor(name) {  
    this.name = name  
    //^^^^  
    // define an instance variable  
  }  
  
  isBestUniversity() {  
    return this.name === 'FHS'  
  }  
}
```

```
const fhs = new University('FHS')  
fhs.isBestUniversity() // true
```

# OOP AND JS

## EXTENDING CLASSES

» Emulates class based oop with prototypes

```
class FHS extends University {  
  isBestUniversity() {  
    return true  
  }  
}
```

```
const fhs = new University('FHS')  
fhs.isBestUniversity() // true
```

# ARROW FUNCTIONS



# FUNCTIONS DECLARATION VS. FUNCTION EXPRESSIONS

» functions in JavaScript are values

» can be passed to other functions <sup>1</sup>

```
function myFunction() { console.log('Hallo') }
```

```
setTimeout(myFunction, 200)
```

```
//           ^^^^
```

```
// pass my function to setTimeout
```

```
// myFunction will be called after 200ms
```

<sup>1</sup> see callbacks from previous lecture

# FUNCTIONS DECLARATION VS. FUNCTION EXPRESSIONS

» functions can be defined like other values in JS

```
const myFunction1 = function myFunction1 () { console.log('Hallo') }  
// ^^^^^^^^^^^^^^^  
// functions can be assigned to a variable
```

```
const myFunction2 = function () { console.log('Hallo') }  
//                ^^^  
// name can be omitted here
```

```
setTimeout(myFunction1, 200)
```

# FUNCTIONS DECLARATION VS. FUNCTION EXPRESSIONS

// function declaration

```
function myFunction1 () { console.log('Hallo') }
```

// function expression

```
const myFunction2 = function () { console.log('Hallo') }
```

# ARROW FUNCTION VS. FUNCTION DECLARATION

- » compact alternative to function expressions
- » can't be used in all situations
- » no binding to this
- » no arguments keyword
- » can't be used as constructor

```
const myArrowFunction = () => { console.log('hallo') }
```

# ARROW FUNCTION

» arrow functions can have an implicit return value

```
const myFunction = () => 1 // returns 1
```

```
const myFunction = () => { 1 } // returns undefined
```

```
const myFunction = () => ({ test: 1 }) // returns { test: 1 }
```



# FUNCTION DECLARATIONS AND THIS

» JavaScript functions bind this when the new keyword is used

```
function Person() {  
  this.age = 0  
  setInterval(function() {  
    this.age++  
    //^^^^ references to window as the function was not created via `new`  
  }, 1000)  
}  
const myPerson = new Person()  
  
// wait a couple of seconds  
myPerson.age === 0  
window.age === NaN
```

# FUNCTION DECLARATIONS AND THIS

```
function Person() {  
  const that = this // save this as a variable so it can be used in setInterval  
  this.age = 0  
  setInterval(function() {  
    that.age++  
  }, 1000)  
}  
const myPerson = new Person()  
  
// wait a couple of seconds  
myPerson.age === 3  
window.age === undefined
```

# FUNCTION DECLARATIONS AND THIS

```
function Person() {  
  this.age = 0  
  setInterval(() => {  
    this.age++ // no need to use that hack  
  }, 1000)  
}  
const myPerson = new Person()  
  
// wait a couple of seconds  
myPerson.age === 3  
window.age === undefined
```

# FUNCTIONS IN JS

// function declaration

```
function myFunction { console.log('hallo') }
```

// function expression

```
const myArrowFunction = function () { console.log('hallo') }
```

// arrow function

```
const myFunction = () => { console.log('hallo') }
```

# FUNCTION DEFAULT VALUES

» since es6 functions accept default values

```
function myFunction (a = 1) {  
    //          ^^^  
    // define a default value for your function  
    console.log(a)  
}  
myFunction() // 1  
myFunction(2) // 2
```



# FUNCTION DEFAULT WITH NAMED ARGUMENTS

```
function myFunction ({ a = 1, b = 2 }) {  
    //  
    // define a default value for your function  
    console.log(a + b)  
}  
myFunction() // 3  
myFunction({ a: 2 }) // 4  
myFunction({ b: 3 }) // 4  
myFunction({ a: 2, b: 3 }) // 5
```

# REST PARAMETERS

» The rest parameter syntax allows us to represent an indefinite number of arguments as an array.

```
function myFunction (...values) {  
    //          ^^^  
    // all arguments will be available as values  
    console.log(values)  
}  
myFunction() // []  
myFunction(1) // [1]  
myFunction(1, 2, 3, 4, 5, 6) // [1, 2, 3, 4, 5, 6]
```

# TEMPLATE LITERALS

# ARRAY METHODS

# ARRAY METHODS

## ARRAY.PROTOTYPE.FOREACH

» calls given callback for each element inside the array

```
const myArray = [1, 2, 3, 4, 5]
const result = myArray.forEach((item) => { console.log(item * 2) })
// logs 1
// logs 2
// ...
// result === undefined
```



# ARRAY METHODS

## ARRAY.PROTOTYPE.MAP

» creates a new array populated with the result of the provided function

```
const myArray = [1,2,3,4,5]
const result = myArray.map((item) => item * 2)
// result will be [2, 4, 6, 8, 10]
```

# ARRAY METHODS

## ARRAY.PROTOTYPE.FILTER

» creates a new array with all elements that pass the given function

```
const myArray = [1,2,3,4,5]
const result = myArray.filter((item) => (item % 2) === 0)
// result will be [2, 4]
```

# ARRAY METHODS

## ARRAY.PROTOTYPE.REDUCE

- » executes a reducer function on each element of the array
- » results in a single value

```
const myArray = [1,2,3,4,5]
const sumOfArray = myArray.reduce((result, item) => {
  return result + item
}, 0)
// sum of array 15
```

# ARRAY METHODS

## ARRAY.PROTOTYPE.REDUCE

```
const myArray = [1,2,3,4,5]
const sumOfArray = myArray.reduce((accumulator, item) => {
  // 1) ^^^^^^^^^^^^^^^^^^^^^
  // 2) ^^^^^
  // 3)          ^^^^
  // 1) reducer function
  // 2) accumulated value of previous iterations
  // 3) the current value of the iteration (1, 2, 3, ...)

  return accumulator + item
  // 4) ^^^^^^^^^^^^^^^^^^^^^
  // 4) return the result for the next iteration
}, 0)
// ^
// define initial value
```

# ARRAY METHODS

## ARRAY METHODS CAN BE COMBINED

```
const makeSmoothie = (ingredients) => {  
  return ingredients  
    .filter((ingredient) => ingredient.rotten === false)  
    .map((ingredient) => ingredient.slice())  
    .reduce((smoothie, ingredient) => smoothie.add(ingredient), new Smoothie())  
}
```



# ARRAY METHODS

## ARRAY.PROTOTYPE.FIND

» finds the first matching element in an array

```
const myArray = [1,2,3,4,5]
const result = myArray.find((item) => (item % 2) === 0)
// result will be 2
```

# ARRAY METHODS

## ARRAY.PROTOTYPE.FLAT

» The `flat()` method converts nested objects into a flat list

```
const myArray = [1, [2, [3], 4], 5]
myArray.flat() // [1, 2, [3], 4, 5]
myArray.flat(2) // [1, 2, 3, 4, 5]
//           ^^^
// amount of levels to flatten
```

# EXERCISE TIME



# EXERCISE TIME

- » You have a list of students:
  - » create a function `countStudentLength` which
    - » gets a string as argument
    - » filter students by given string
    - » sum the length of the students names

# EXERCISE TIME

```
const students = [  
  { name: "Hans" },  
  { name: "Mike" },  
  { name: "Fabian" },  
  { name: "Anna" }  
]  
// todo: implement me
```

# TEMPLATE LITERALS

- » es6 enhances strings with a completely new syntax
  - » called template literals
- » they make it possible to
  - » interpolate strings
  - » multiline strings
  - » embed expressions

<sup>2</sup> see <https://developers.google.com/web/updates/2015/01/ES6-Template-Strings> for more info

# TEMPLATE LITERALS

## STRING INTERPOLATION

```
const university = 'FHS'  
const myString = `My University is ${university}`  
//  
// template literals are using back-ticks ``
```



# TEMPLATE LITERALS

## EMBEDDED EXPRESSIONS

```
const myUniversity = () => 'FHS'  
const myString = `My University is ${myUniversity()}`  
//                                     ^  
//                                     ^  
// functions and methods can be called ``
```

# TEMPLATE LITERALS

```
const myUniversity = () => 'FHS'  
const myString = `My University is ${myUniversity()}`  
//                                     ^  
// functions and methods can be called ``
```

# TEMPLATE LITERALS

## MULTI LINE STRINGS

```
const greeting1 = "Hello \
World";
//                      ^
// use backslash \ to start a new line
```

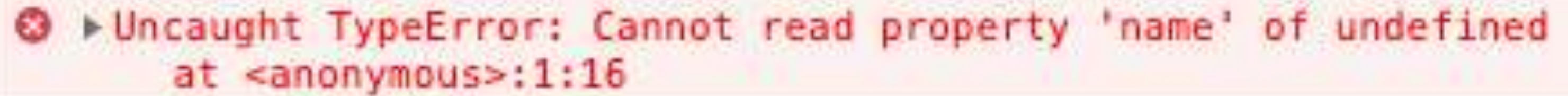
```
const greeting2 = "Hello " +
"World";
//                      ^
// use backslash + to concat 2 strings
```

```
const greeting3 = `Hello
World`;
//                      ^
// with template literals new lines
// will be put into one line
```



# OPTIONAL CHAINING

# OPTIONAL CHAINING



Uncaught TypeError: Cannot read property 'name' of undefined  
at <anonymous>:1:16

# OPTIONAL CHAINING

## OBJECT VALUES<sup>3</sup>

» Allows to read values deep within an object chain

» When value is null or undefined returns null

```
const adventurer = {
  cat: { name: 'Dinah' }
}

const dogName = adventurer.dog?.name;
//                ^^^^^
// dogName will be undefined and no error

const catName = adventurer.cat?.name;
//                ^^^^^
// catName will be 'Dinah' as cat is defined
```

<sup>3</sup> Compiled Source

# OPTIONAL CHAINING NESTED FUNCTIONS<sup>4</sup>

```
const adventurer = {  
  name: 'Alice',  
  dogName: () => 'Dinah'  
}  
adventurer.dogName.?.() // undefined  
adventurer.catName.?.() // 'Dinah'
```

<sup>4</sup> Compiled Source





**NO HOMEWORK TILL NEXT TIME**



# FEEDBACK

» Questions: [tmayrhofer.lba@fh-salzburg.ac.at](mailto:tmayrhofer.lba@fh-salzburg.ac.at)

» [Feedback Link](#)