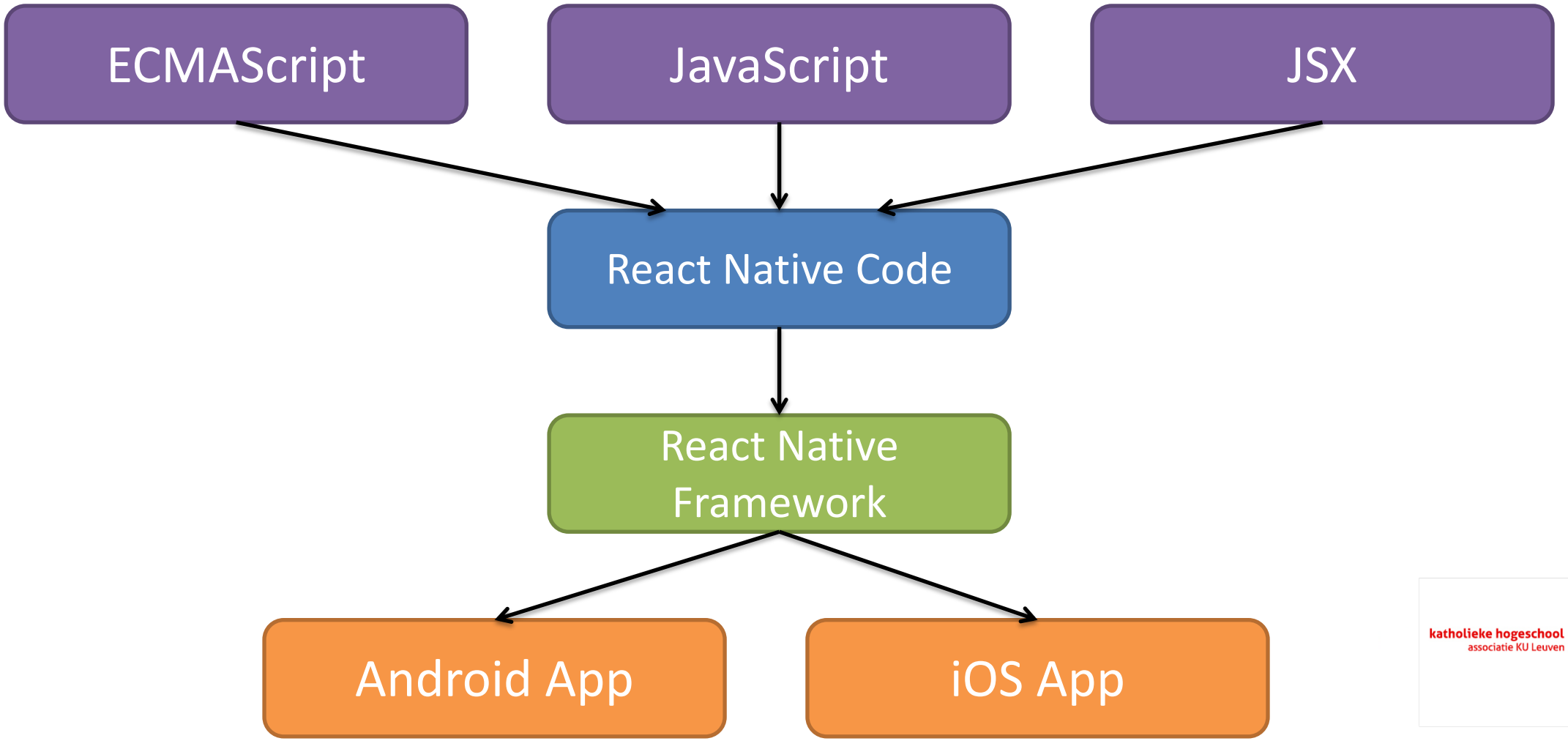
ES2015 (ES6)

Modern Javascript

katholieke hogeschool
associatie KU Leuven
vives

React Native, ES & JavaScript



ECMAscript

- Scripting language specification
  - Standaardiseren
- Javascript is implementatie van ECMAScript
- ECMAScript 5 = plain old JS
- Juni 2015: ES6 of ECMAScript 2015
  - Huidige standaard voor JavaScript
  - Aanzienlijke veranderingen tov ES5
- Babel: prepocessen van ES6
  - Backward compatible

New features

- Block Scoped Declarations (var, let, const)
- Arrow Functions
- Default arguments
- Spread and Rest Operator
- Destructuring
- Template Literal and Delimiters
- Modules
- Classes
- Working with arrays

Block scoped declarations

New variable type: let

```
let a = 20;
a = 50;
a = {};
```

Block scoped declarations

New variable type: let

The scope of a variable defined with var is function scope, or declared outside any function, global

Variables declared using the let or const keywords block-scoped, which means that they are available only in the block {} in which they were declared.

```
function varTest() {
  var x = 1;
  if (true) {
    var x = 2;  // same variable!
    console.log(x);  // 2
  }
  console.log(x);  // 2
}

function letTest() {
  let x = 1;
  if (true) {
    let x = 2;  // different variable
    console.log(x);  // 2
  }
  console.log(x);  // 1
}
```

Block scoped declarations

New variable type: const

Const creates a read-only reference to a value. It does not mean the value it holds is immutable, just that the variable identifier cannot be reassigned.

Variables declared using const keywords is also block-scoped.

```
const key = 'mykey';

key = 'newkey';
// error "key" is read only
```

Block scoped declarations

New variable type: const

```
const person = {
  name: 'Jennifer',
  age: 43,
  occupation: 'Dentist',
}

person.name = 'Jim';


console.log('person:', person);
```

```
{
  "age": 43,
  "name": "Jim",
  "occupation": "Dentist"
}
```

Block scoped declarations

New variable type: const

```
const people = ['Mark', 'Tim', 'Jennifer'];

people.push('Amanda');



console.log('people:', people);
```

```
    Array [
      "Mark",
      "Tim",
      "Jennifer",
      "Amanda"
    ]
```

Arrow functions

es5

```
var sayName = function() {
  console.log('I am Nader!');
};
```

es2015/ES6

```
const sayName = () => {
  console.log('I am Nader!');
};
```

Arrow functions

Arguments

```
const sayName = (name) => console.log('I am ' + name + ' !');

const getFullName = (first, last) => first + ' ' + last

const addThree = (a, b, c) => a + b + c;
```

Arrow functions

Implicit return vs explicit return

implicit

```
const getFullName = (first, last) => first + ' ' + last
```

explicit

```
const getFullName = (first, last) => {
  return first + ' ' + last;
};
```

```
const name = getFullName('nader', 'dabit');
console.log('name:', name);
```

```
"name:" "nader dabit"
```

Arrow functions

Implicit return vs explicit return

es5

```
var getFullName = function(first, last) {
  return first + ' ' + last
};
```

es2015/ES6

```
const getFullName = (first, last) => {
  return first + ' ' + last;
};
const getFullName = (first, last) => first + ' ' + last
```

Default arguments

Arrays

```
const append = (value, array = []) => {
  array.push(value);
  return array;
}

const arr1 = append(4);
const arr2 = append(4, [0, 1, 2, 3])

console.log('arr1:', arr1);
console.log('arr2:', arr2);
```

Console

```
"arr1:" Array [
  4
]
"arr2:" Array [
  0,
  1,
  2,
  3,
  4
]
```

Default arguments

Function

```
const getName = (name = myName()) => {
 return name;
}

const myName = () => {
  return 'nader';
}

console.log(getName());
console.log(getName('amanda'));
```

Console

```
"nader"
"amanda"
```

Spread and rest operators

When using **spread**, you are expanding a single variable into more:

```
const abc = ['a', 'b', 'c'];
const def = ['d', 'e', 'f'];
const alpha = [ ...abc, ...def ];
```

When using **rest** arguments, you are collapsing all remaining arguments of a function into one array:

```
const sum = ( first, ...others ) => {
    // others is now an array
    // so something with others
}
```

katholieke hogeschool associatie KU Leuven vives

Spread and rest operators

Array rest

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

```
const f = (...rest) => {
 console.log('rest:', rest);
}

f(1, 2, 3, 4)
    “rest:" Array [
      1,
      2,
      3,
      4
    ];
```

Spread and rest operators

Array spread

The spread syntax allows an expression to be expanded in places where multiple arguments (for function calls) or multiple elements (for array literals) or multiple variables (for destructuring assignment) are expected.

```
let fruits = ['banana'];
const moreFruits = ['apple', 'orange'];
fruits = [...fruits, ...moreFruits];
console.log(fruits);
```

```
Array [
  "banana",
  "apple",
  “orange"
]
```

katholieke hogeschool associatie KU Leuven vives

Spread and rest operators

Array spread

```
const people = ['Jason', 'Amanda'];
const allPeople = ['Nader', ...people, 'Chris', 'Jennifer'];

console.log('allPeople: ', allPeople);
```

```
"allPeople: " Array [
  "Nader",
  "Jason",
  "Amanda",
  "Chris",
  "Jennifer"
]
```

Spread and rest operators

Array spread

Items in array as function argument

```
const myFunction = (x, y, z) => {
  // do stuff with x, y, and z
  console.log('z:', z)
}

const args = [0, 1, 2];
myFunction(...args);
```

Console

```
"z:" 2
```

Spread and rest operators

Object spread

Overwriting object propeties

```
const person = { name: 'Jim', age: 22 };

const Amanda = { ...person, name: 'Amanda' };

console.log('Amanda:', Amanda);
```

```
"Amanda:" Object {
  "age": 22,
  "name": "Amanda"
}
```

Spread and rest operators

Object spread

Nested objects

```
const farmer = {
  info: {
    occupation: 'farmer'
  }
}

const person = {
  name: 'chris',
  ...farmer
}

console.log('person:', person);
```

```
person: { name: 'chris', info: { occupation: 'farmer' } }
```

Array destructuring

es5

```
const arr = [10, 20, 30, 40, 50];

const a = arr[0];
const b = arr[1];

console.log(a); // 10
console.log(b); // 20
```

es2015/ES6

```
const arr = [10, 20, 30, 40, 50];

const [a,b] = arr;


console.log(a); // 10
console.log(b); // 20
```

Array destructuring

Destructuring + rest

```
const arr = [10, 20, 30, 40, 50];

const [x, y, ...z] = arr;

console.log('z:', z);
```

```
"z:" Array [
  30,
  40,
  50
];
```

Object destructuring

es5

```
const person = {
  name: 'Chris',
  info: {
    hairColor: 'brown',
    height: "6'1",
  },
};

const name = person.name;
const info = person.info;
```

es2015/ES6

```
const person = {
  name: 'Chris',
  info: {
    hairColor: 'brown',
    height: "6'1",
  },
};

const { name, info } = person;
```

Object destructuring

Default values

```
const person = {
  info: {
    hairColor: 'brown',
    height: "6'1",
  },
};

const { name = 'Amanda' } = person;
```

```
"name:" "Amanda"
```

Object destructuring

Nested objects

```
const person = {
  name: 'Chris',
  info: {
    hairColor: 'brown',
    height: "6'1",
  },
};
const {name, info: personInfo, info: { hairColor }} = person;
```

```
"name:" "Chris"
"personInfo:" Object {
  "hairColor": "brown",
  "height": "6’1"
}
"hairColor:" "brown"
```

Template literals

es5

```
let name = 'Chris’;
let age = '36';
let str = 'Hello, my name is ' + name + 'and my age is ' + age;
```

es2015/ES6

```
let name = 'Chris';
let age = '36';
let str = `Hello, my name is ${name} and my age is ${age}`;
```

Template literals

Expressions / logic

```
const person = {
  name: 'Chris',
  age: 23,
  present: false
}

const getName = () => person.name

const info = `${getName()} is ${person.present ? 'present' : 'not present'}`;
```

Modules

By contrast to the old module.exports = {...}, we can now export multiple named values. Similarly, we can import multiple named values.

es5

```
//export
var myModule = {x: 1};
module.exports = myModule;

// import
var myModule = require('./myModule');
```

es2015/ES6

```
//export
const myModule = {x: 1};
export default myModule;

// import
import myModule from './myModule';
```

Modules

Default export / import

```
// name.js

const name = 'Chris';
export default name;
```

```
// in some other file

import name from './name';

console.log('name:', name);
"name:" "Chris"
```

Modules

Default export / import

There is one default export per file, and this exported value can be imported without refering to it by name. Every other import and export must be named.

```
// name.js
const name = 'Chris';
export default name;
```

```
// in some other file
import person from './name';

console.log(person:', person);
"person:" "Chris"
```

Modules

Default export / import: functions

```
// getName.js

const person = {
  name: 'Chris',
  age: 22,
};

const getName = () => person.name;

export default getName;
```

```
// in some other file

import getName from './getName';

console.log(getName());

"name:" "Chris"
```

Modules

Default export / import: objects

```
// person.js

const person = {
  name: 'Amanda',
  age: 33,
};

export default person;
```

```
// in some other file

import person from './person';

console.log('age: ', person.age);

"age: " 33
```

Modules

Named export / import

```
// in constants.js

export const IS_LOADING = 'IS_LOADING';
export const IS_LOADED = 'IS_LOADED';
```

```
// in some other file

import {
  IS_LOADING,
  IS_LOADED,
} from './constants';

console.log('IS_LOADED:', IS_LOADED);
```

```
"IS_LOADED:" "IS_LOADED"
```

Modules

Multiple types

```
// in person.js

const person = {
  name: 'Jason',
  occupation: 'Realtor',
  age: 22,
};

const getAge = () => person.age;

export { getAge, person as default };
```

```
// in some other file

import person, { getAge } from './person';
```

Modules

Import multiple types – React Native

```
// import the default export
import React from 'react-native'

// import other named exports
import {View, Text, Image} from 'react-native'

// import default and others simultaneously
import React, {View, Text, Image} from 'react-native'
```

Classes

es5.js

```es5.js
"use strict";

//Parent class
var Shape = function(id, x, y) {
  this.id = id;
  this.location(x, y);
};

Shape.prototype.location = function(x, y) {
  this.x = x;
  this.y = y;
};

Shape.prototype.toString = function(x, y) {
  return "Shape(" + this.id + ")"
};

Shape.prototype.getLocation = function() {
  return {
    x: this.x,
    y: this.y
  }
};



//Child class
var Circle = function(id, x, y, radius) {
  Shape.call(this, id, x, y);
  this.radius = radius;
};
Circle.prototype = Object.create(Shape.prototype);
Circle.prototype.constructor = Circle;


Circle.defaultCircle = function() {//static function
  return new Circle("default", 0, 0, 100);
};

Circle.prototype.toString = function() {
  return "Circle > " + Shape.prototype.toString.call(this);
};



var defaultCircle = Circle.defaultCircle(); //call static function

var myCircle = new Circle('123', '5px', '10px', 5); // create new instance
console.log(myCircle.toString()); // Circle > Shape(id = 123)
console.log(myCircle.getLocation()); // { x: '5px', y: '10px' }
```

es6.js

```es6.js
"use strict";

//Parent class
class Shape {
  constructor(id, x, y) { // constuctor syntactic sugar
    this.id = id
    this.location(x, y)
  }

  location(x, y) { //<-- prototype function
    this.x = x
    this.y = y
  }

  toString() { //<-- prototype function
    return `Shape(id = ${this.id})`
  }

  getLocation() { //<-- prototype function
    return {
      x: this.x,
      y: this.y
    };
  }
}



//Child class
class Circle extends Shape {
  constructor(id, x, y, radius) {
    super(id, x, y) // Call Shape's constructor (via super)
    this.radius = radius
  }

  static defaultCircle() { // Static function
    return new Circle("default", 0, 0, 100)
  }

  toString() { //<-- override toString of Shape
    return "Circle > " + super.toString() //call "super" instead of "this" to access parent
  }
}



var defaultCircle = Circle.defaultCircle(); //call static function

var myCircle = new Circle('123', '5px', '10px', 5); // create new instance
console.log(myCircle.toString()); // Circle > Shape(id = 123)
console.log(myCircle.getLocation()); // { x: '5px', y: '10px' }
```

One thing to note here is that ES6 class is not a new object-oriented inheritance model. They just serve as a syntactical sugar over JavaScript's existing prototype-based inheritance

Classes

Class

`class` gives us built in instance functions, static functions, and inheritance. `constructor` is a special function that is called automatically every time a class instance is created. We can use the `static` keyword to declare static class functions.

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  static beProud() {
    console.log('I AM AN ANIMAL’);
  }

  printName() {
    console.log(this.name);
  }
}

const animal = new Animal('Cat')
animal.printName() // Cat
Animal.beProud() // I AM AN ANIMAL
```

Classes

Inheritance

The `class` gives us simple inheritance with the keyword `extends`. In classes that inherit from parents, we have access to a function `super()`. Within an inherited function in that child class, `super` will invoke the parent class's version of that function.

```
class Cat extends Animal {

  printName() {
    super.printName();
    console.log(`My name is ${this.name}`);
  }
}
```

Working with arrays

```
const emptyArray = [];
const fruits = ['apple', 'orange’];
const mixed = ['apple’, 13, true];

const aantalElementen = mixed.length;

const colors = ['green', 'blue', 'red’];
console.log(colors[0]);          // ‘green’
console.log(colors[1]);          // ‘blue’
console.log(colors[2]);          // ‘red’
console.log(colors[0]);          // undefined
```

katholieke hogeschool associatie KU Leuven vives

Working with arrays

ES5 Javascript functies

- pop()
- push(e)
- unshift(e)
- shift()
- slice(i1, i2)

Working with arrays

ES5 Native array functions – forEach()

Executes the provided function for each element of the array, passing the array element as an argument.

```
const colors = ['banana’, ‘apple’, ‘orange'];
colors.forEach(color => {
    console.log(color);
});
```

```
"banana",
"apple",
“orange"
```

Working with arrays

ES5 Native array functions – filter()

Creates a new array containing a subset of the original array. The result has these elements that pass the test implemented by the provided function, which should return true or false

```
const values = [1, 60, 34, 30, 20, 5];
const lessThan20 = values.filter(v => {
    return v < 20;
});

console.log(lessThan20);
[1, 5]
```

Working with arrays

ES5 Native array functions – map()

Creates a new array containing the same number of elements, but output elements are created by the provided function. It just converts each array element to something else

```
const colors = ['banana’, ‘apple’, ‘orange'];
const capColors = colors.map(color => {
    return color.toUpperCase();
});

console.log(capColors);
[”BANANA", "APPLE", “ORANGE"]
```

Working with arrays

ES5 Native array functions – some()

Checks if **any** element of the array passes the test implemented by the provided function, which should return true or false.

```
const people = [
    {name: 'Jennifer', age: 43},
    {name: 'Jim', age: 9},
    {name: 'John', age: 16},
    {name: 'Ann', age: 19},
];

const thereAreTeenagers = people.some(person => {
    return person.age > 10 && person.age < 20;
});
// thereAreTeenagers = true
```

Working with arrays

ES5 Native array functions – every()

Checks if **every** element of the array passes the test implemented by the provided function, which should return true or false.

```
const people = [
    {name: 'Jennifer', age: 43},
    {name: 'Jim', age: 9},
    {name: 'John', age: 16},
    {name: 'Ann', age: 19},
];

const thereAreTeenagers = people.every(person => {
    return person.age > 10 && person.age < 20;
});
// thereAreTeenagers = false
```

Working with arrays

ES5 Native array functions – reduce()

Applies a function passed as the first parameter against an accumulator and each element in the array (from left to right), thus reducing it to a single value. The initial value of the accumulator should be provided as the second parameter of the reduce function.

```
const values = [1, 2, 3, 4, 5];
const s1 = values.reduce((acc,value) => {
    return acc + value;
})
// s1 = 15

const s2 = values.reduce((acc,value) => {
    return acc + value;
}, 5)
// s2 = 20
```

Working with arrays

ES6 find()

The find() method returns the **value** of the **first element** in the array that satisfies the provided testing function. Otherwise undefined is returned.

```
const values = [1, 2, 3, 4, 5];
const oddNumber = values.find(num => {
    return num % 2 == 1;
})
// oddNumber = 1
```

Working with arrays

ES6 for … of …

For-of is a new loop in ES6 that we can use to loop over iterable objects like arrays, strings, maps, and sets

```
const colors = ['banana’, ‘apple’, ‘orange'];
for(let color of colors) {
    console.log(color);
});
```

```
"banana",
"apple",
“orange"
```

Time to work

- Toledo
- GitHub Classroom link
- Account linken met naam
- Private repository wordt voor jou aangemaakt
- Clone private repository => local repository
- Doorlopen hoofdstukken online
- Maak opdrachten in VS Code
- Commit & push