JavaScript

Desenvolvimento de Software e Sistemas Móveis (DSSMV)
Licenciatura em Engenharia de Telecomunicações e Informática
LETI/ISEP

2025/26

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

Introduction to JavaScript

Introduction to JavaScript (I)

- JavaScript is the programming language of the web.
  - The overwhelming majority of websites use JavaScript, and all modern web browsers—on desktops, tablets, and phones—include JavaScript interpreters, making JavaScript the most-deployed programming language in history.
- Over the last decade, **Node.js** [1] has enabled JavaScript programming outside of web browsers, and the dramatic success of Node means that JavaScript is now also the most-used programming language among software developers.

```
//hello.js
console.log("Hello World!");
```

- `node hello.js`

[1] https://nodejs.org/en/

Introduction to JavaScript (II)

- JavaScript is a **case-sensitive** language.
- JavaScript supports two styles of **comments**:
    - Any text between a `//` and the end of a line.
    - Any text between the characters `/* ... */`.
- Identifiers
    - An identifier is simply a **name**.
        - Identifiers are used to name **constants, variables, properties, functions, and classes**
        - A identifier must begin with a letter, an underscore (_), or a dollar sign ($).
        - Subsequent characters can be letters, digits, underscores, or dollar signs.
        - Numbers are not allowed as the first character of an identifier

Introduction to JavaScript (III)

- Reserved Words

as	const	export	get	null	target	void
async	continue	extends	if	of	this	while
await	debugger	false	import	return	throw	with
break	default	finally	in	set	true	yield
case	delete	for	instanceof	static	try	
catch	do	from	let	super	typeof	
class	else	function	new	switch	var	

- Optional Semicolons
  - JavaScript uses the semicolon (`;`) to separate statements from one another.
  - In JavaScript, you can usually **omit** the semicolon (`;`) between two statements **if those statements are written on separate lines**.

Types, Values, and Variables

`null` and `undefined`

- `null` is a language keyword that evaluates to a special value that is usually used to indicate the absence of a value.
  - The `null` is a special assignment value, which can be assigned to a variable as a representation of no value
- JavaScript also has a second value that indicates absence of value, the `undefined` value represents a deeper kind of absence.
  - It is the value of variables that have not been initialized

0
Cookies
null
Cookies
undefined

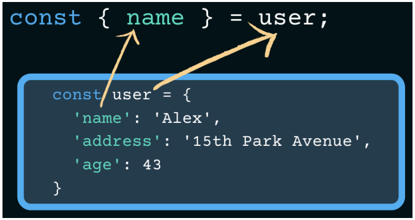
Declarations with `let` and `const`

- Variables are declared with the `let` keyword,
- To declare a constant instead of a variable, use `const` instead of `let`.

```
let message = "hello";
let i = 0, j = 0, k = 0;
let x = 2, y = x*x;
const H0 = 74; // Hubble constant (km/s/Mpc)
const C = 299792.458; // Speed of light in a vacuum (km/s)
const AU = 1.496E8; // Astronomical Unit: distance to the sun (km)
```

Destructuring Assignment (I)

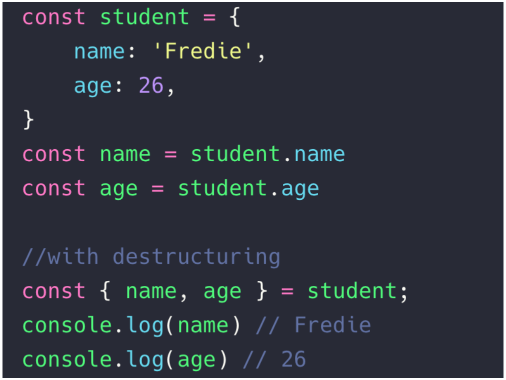
- ES6 implements a kind of compound declaration and assignment syntax known as **destructuring assignment**.
- In a destructuring assignment, the value on the righthand side of the equals sign is an array or object (a "structured" value), and the lefthand side specifies one or more variable names using a syntax that mimics array and object literal syntax.



```
const { name } = user;

const user = {
  'name': 'Alex',
  'address': '15th Park Avenue',
  'age': 43
}
```

Destructuring Assignment (II)

- When a destructuring assignment occurs, one or more values are extracted ("destructured") from the value on the right and stored into the variables named on the left.



```
const student = {
    name: 'Fredie',
    age: 26,
}
const name = student.name
const age = student.age

//with destructuring
const { name, age } = student;
console.log(name) // Fredie
console.log(age) // 26
```

Expressions, Operators, and Statements

Expression

- An expression is a phrase of JavaScript that **can be evaluated to produce a value**.
- A **variable name** is also a simple expression that evaluates to whatever value has been assigned to that variable.
- The most common way to build a complex expression out of simpler expressions is with an operator.
  - An operator combines the values of its operands (usually two of them) in some way and evaluates to a new value.

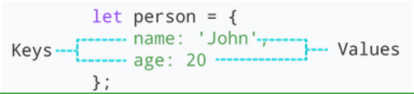
Object and Array Initializers

- An array initializer expression is a comma-separated list of expressions contained within square brackets.

```
let array=[]; // An empty array
let array2=[1+2,3+4];//A 2-element array. First element is 3, second is 7
let matrix = [[1,2,3], [4,5,6], [7,8,9]];
```

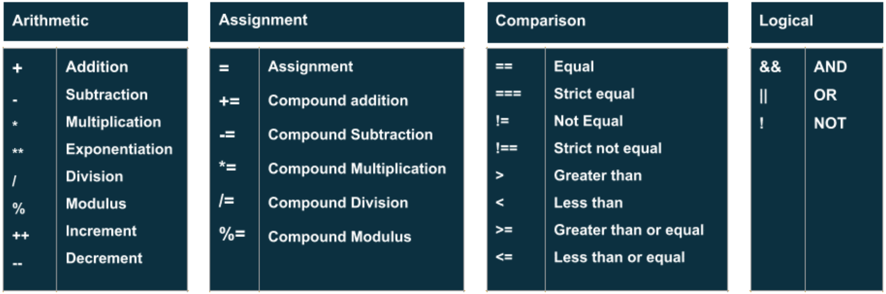
- Object initializer expressions are composed by a set of subexpression and each is prefixed with a property name and a colon.

```
   let p = { x: 2.3, y: -1.2 }; // An object with 2 properties
   let q = {};// An empty object with no properties
   q.x = 2.3; q.y = -1.2; // Now q has the same properties as p
   let rectangle = {
    upperLeft: { x: 2, y: 2 }, //property
    lowerRight: { x: 4, y: 5 }
   };
```



```
let person = {
  name: 'John',
  age: 20
};
```

Operators



Arithmetic	
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation
/	Division
%	Modulus
++	Increment
--	Decrement

Assignment	
=	Assignment
+=	Compound addition
-=	Compound Subtraction
*=	Compound Multiplication
/=	Compound Division
%=	Compound Modulus

Comparison	
==	Equal
===	Strict equal
!=	Not Equal
!==	Strict not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

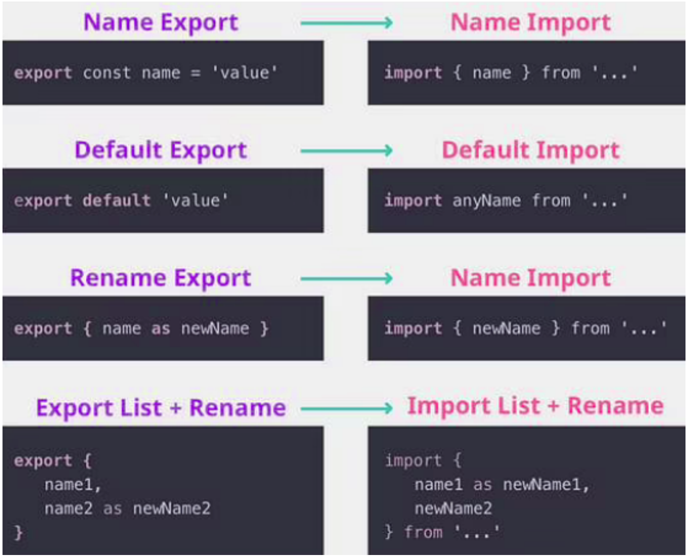
Logical	
&&	AND
||	OR
!	NOT

- The strict equality operator (`===`) checks whether its two operands are equal, returning a Boolean result.
  - In this operation, `A === B`, both `A` and `B` should be of the same type else the result will be always false since two values of different data types cannot be equal in any universe.

`import` and `export` (I)

- The `import` and `export` declarations **are used together to make values defined in one module of JavaScript code available in another module**.
- A module **is a file of JavaScript code with its own global namespace**, completely independent of all other modules.
- The only way that a value (such as function or class) defined in one module can be used in another module is if the defining module exports it with `export` and the using module imports it with `import`.

import and export (II)



```
export const name = 'value'
```

```
import { name } from '...'
```

```
export default 'value'
```

```
import anyName from '...'
```

```
export { name as newName }
```

```
import { newName } from '...'
```

```
export {
    name1,
    name2 as newName2
}
```

```
import {
    name1 as newName1,
    newName2
} from '...'
```

Objects

Objects

- Objects are JavaScript’s most fundamental datatype.
- An object **is a composite value**: it aggregates multiple values (primitive values or other objects) and allows you to store and retrieve those values by name.
- An object **is an unordered collection of properties**, each of which **has a name/key and a value**.
- JavaScript objects are dynamic–properties can usually be added and deleted–but they can be used to simulate the static objects and "structs" of statically typed languages.
- Objects **are mutable and manipulated by reference rather than by value**.

Serializing Objects

- Object serialization is the process of converting an object’s state to a string from which it can later be restored (deserialization).
- The functions `JSON.stringify()` and `JSON.parse()` serialize and restore JavaScript objects.

```
let o = {x: 1, y: {z: [false, null, ""]}}; // Define a test object
let s = JSON.stringify(o);
// s == '{"x":1,"y":{"z":[false,null,""]}}'
let p = JSON.parse(s);
// p == {x: 1, y: {z: [false, null, ""]}}
```

Creating Objects

- Objects can be created with object literals, with the `new` keyword, and with the `Object.create()` function.

```
let book = {
"main title": "JavaScript", //These property names include spaces,
"sub-title": "The Definitive Guide",//and hyphens, so use string literals.
author: {
 firstname: "David",
 surname: "Flanagan"
  }
};
```

```
let o = new Object(); // Create a empty object: same as {}.
let a = new Array(); // Create a empty array: same as [].
let d = new Date(); // Create a Date object represeting the current time
let r = new Map(); // Create a Map object for key/value mapping
```

```
let o1 = Object.create({x: 1, y: 2});
```

Spread Operator (I)

- The spread operator `...` **is used to expand or spread** an iterable object such as an array, a map, or a set.

```
onst arr1 = ['one', 'two'];
const arr2 = [...arr1, 'three', 'four', 'five'];
console.log(arr2); // ["one", "two", "three", "four", "five"]
```

- The spread operator merges two or more objects and creates a new object which contains all the properties of the merged objects.

```
const obj1 = { x : 1, y : 2 };
const obj2 = { z : 3 };

const obj3 = {...obj1, ...obj2}; // add members obj1 and obj2 to obj3

console.log(obj3); // {x: 1, y: 2, z: 3}
```

Spread Operator (II)

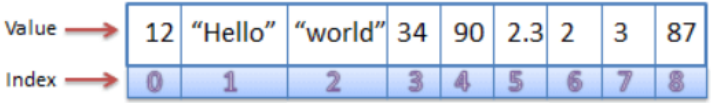
- Updating an object

```
const myObj = {
 name: "Brian",
 profession: "Rockstar"
};

function newObj(){
 return {
  ...myObj,
  profession: "Lion Tamer"
 }
}

console.log(newObj()); // { name: "Brian", profession: "Lion Tamer" }
```

Array

Array

- An array is an ordered collection of values.
- Each value is called an **element**, and each element has a numeric position in the array, known as its **index**.
- Arrays are **untyped**: an array element may be of any type, and different elements of the same array may be of different types.
  - Array elements may even be objects or other arrays, which allows you to create complex data structures, such as arrays of objects and arrays of arrays.



- Every array has a `length` property

```
let vec = [1,3,4,5,2,7];
for(let i = 0 ; i < vec.length; i++) {
  // loop body remains the same
}
```

Creating Arrays

- Array literals

```
let misc = [ 1.1, true, "a", ]; // 3 elements of various types + trailing
     comma
```

- The `...` spread operator on an iterable object

```
  let a = [1, 2, 3];
  let b = [0, ...a, 4]; // b == [0, 1, 2, 3, 4]
```

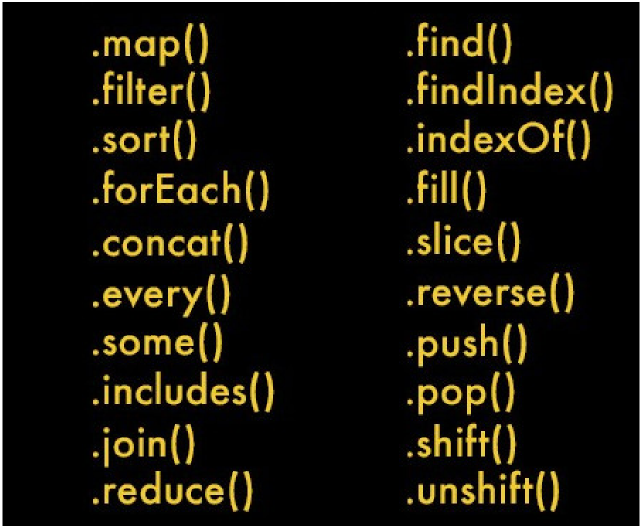
- The `Array()` constructor

```
  let a = new Array(5, 4, 3, 2, 1, "testing, testing");
```

- The `Array.of()` and `Array.from()` factory methods

```
  let a = Array.of() // => []; returns empty array with no arguments
  let b = Array.of(10)// => [10]; can create arrays with a single numeric
       argument
  let c= Array.of(1,2,3)// => [1, 2, 3]

  let copy = Array.from(original);
```

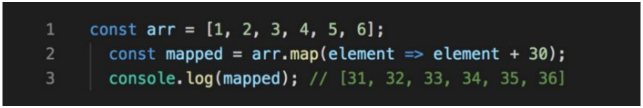
Array Methods (I)



.map()
.filter()
.sort()
.forEach()
.concat()
.every()
.some()
.includes()
.join()
.reduce()
.find()
.findIndex()
.indexOf()
.fill()
.slice()
.reverse()
.push()
.pop()
.shift()
.unshift()

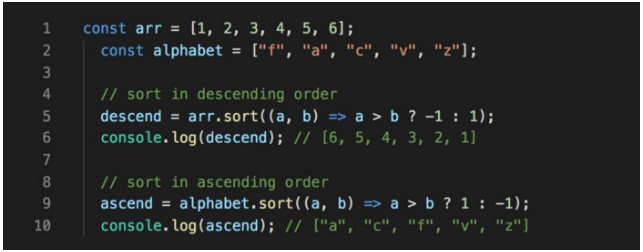
Array Methods (II)

- `map()`
  - This method creates a new array with the results of calling a provided function on every element in this array.



```
const arr = [1, 2, 3, 4, 5, 6];
  const mapped = arr.map(element => element + 30);
  console.log(mapped); // [31, 32, 33, 34, 35, 36]
```

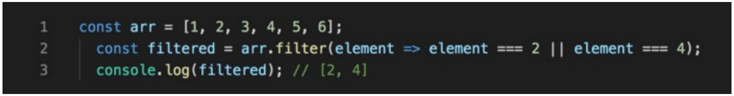
- `sort()`
  - This method is used to arrange/sort array's elements either in ascending or descending order.



```
const arr = [1, 2, 3, 4, 5, 6];
  const alphabet = ["f", "a", "c", "v", "z"];

  // sort in descending order
  descend = arr.sort((a, b) => a > b ? -1 : 1);
  console.log(descend); // [6, 5, 4, 3, 2, 1]

  // sort in ascending order
  ascend = alphabet.sort((a, b) => a > b ? 1 : -1);
  console.log(ascend); // ["a", "c", "f", "v", "z"]
```

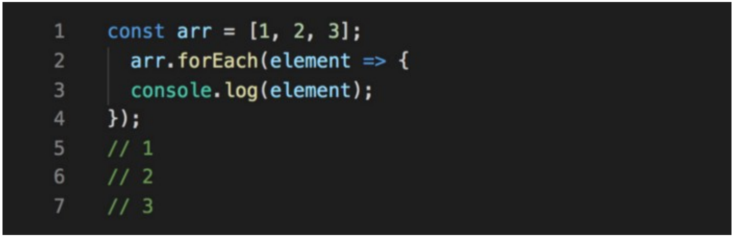
Array Methods (III)

- `filter()`
  - This method creates a new array with only elements that passes the condition inside the provided function.



```
const arr = [1, 2, 3, 4, 5, 6];
  const filtered = arr.filter(element => element === 2 || element === 4);
  console.log(filtered); // [2, 4]
```

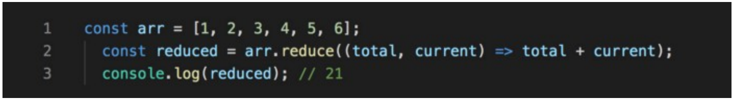
- `forEach()`
  - This method helps to loop over array by executing a provided callback function for each element in an array.



```
const arr = [1, 2, 3];
  arr.forEach(element => {
  console.log(element);
});
// 1
// 2
// 3
```

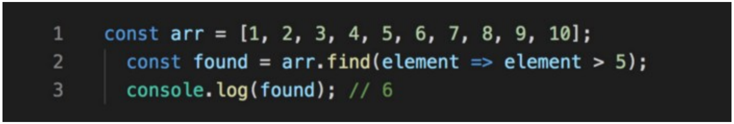
Array Methods (IV)

- `reduce()`
  - This method applies a function against an accumulator and each element in the array to reduce it to a single value.



```
const arr = [1, 2, 3, 4, 5, 6];
  const reduced = arr.reduce((total, current) => total + current);
  console.log(reduced); // 21
```

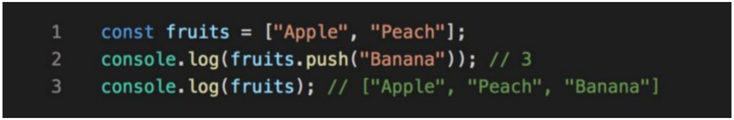
- `find()`
  - This method returns the value of the first element in an array that pass the test in a testing function.



```
const arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
  const found = arr.find(element => element > 5);
  console.log(found); // 6
```

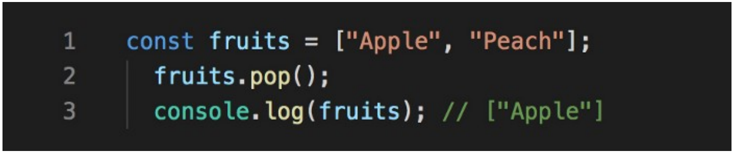
Array Methods (V)

- `push()`
  - This method adds one or more elements to the end of array and returns the new length of the array.



```
const fruits = ["Apple", "Peach"];
console.log(fruits.push("Banana")); // 3
console.log(fruits); // ["Apple", "Peach", "Banana"]
```

- `pop()`
  - This method removes the last element from the end of array and returns that element.

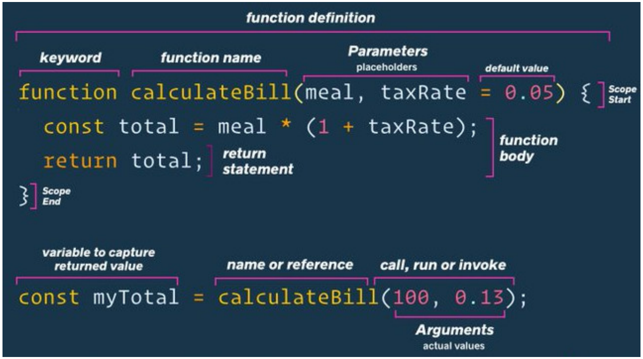


```
const fruits = ["Apple", "Peach"];
  fruits.pop();
  console.log(fruits); // ["Apple"]
```

Functions

Functions

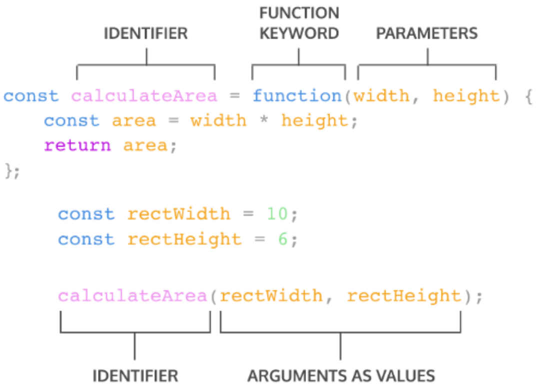
- A **function** is a block of JavaScript code that **is defined once** but may be **executed/invoked any number of times**.
- A function definition expression typically consists:
  - Of the keyword `function` followed by a comma-separated list of zero or more identifiers (**the parameter names**) in parentheses and a block of JavaScript code (**the function body**) in curly braces.



```
function calculateBill(meal, taxRate = 0.05) {
  const total = meal * (1 + taxRate);
  return total;
}

const myTotal = calculateBill(100, 0.13);
```

Function Expression (I)

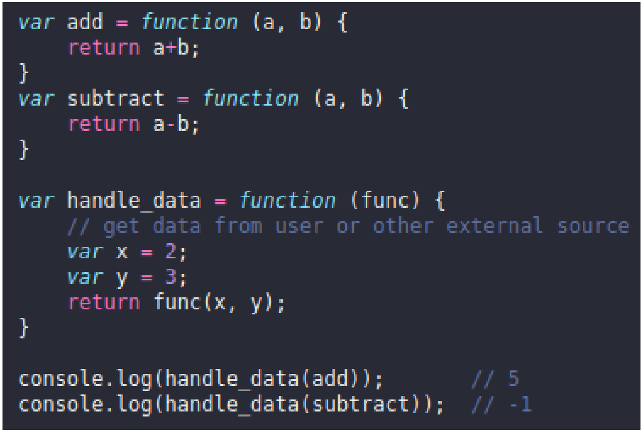
- To define a function inside **an expression**, we can use the `function` keyword and **the function name is usually omitted**.
- A function with no name is called **an anonymous function**.
- A function expression **is often stored in a variable in order to refer to it**.



```
const calculateArea = function(width, height) {
    const area = width * height;
    return area;
};

      const rectWidth = 10;
      const rectHeight = 6;

      calculateArea(rectWidth, rectHeight);
```

Function Expression (II)

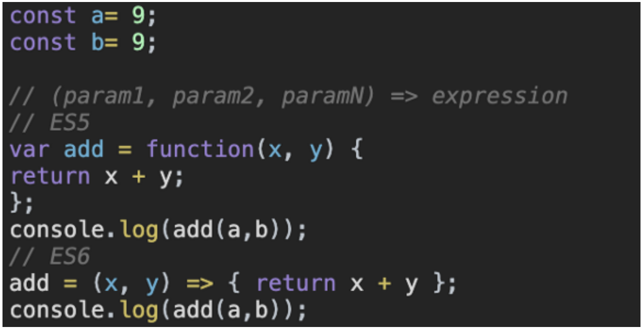
- In JavaScript you **can assign functions to variables and pass them to other functions**



```
var add = function (a, b) {
    return a+b;
}
var subtract = function (a, b) {
    return a-b;
}

var handle_data = function (func) {
    // get data from user or other external source
    var x = 2;
    var y = 3;
    return func(x, y);
}

console.log(handle_data(add));       // 5
console.log(handle_data(subtract));  // -1
```

Arrow Functions (I)

- This syntax is reminiscent of mathematical notation and uses an `=>` “arrow” **to separate the function parameters from the function body**.
- The `function` keyword is not used, and, since arrow functions are expressions instead of statements, there is no need for a function name, either.



```
const a= 9;
const b= 9;

// (param1, param2, paramN) => expression
// ES5
var add = function(x, y) {
return x + y;
};
console.log(add(a,b));
// ES6
add = (x, y) => { return x + y };
console.log(add(a,b));
```

Arrow Functions (II)

- **No param** requires parentheses. With simple expression (only one statement) return is not needed:

```
  () => expression
```

- **One param**, parentheses are not required:

```
param => expression
```

- **Multiple params** require parentheses:

```
(param1, paramN) => expression
```

- Expression with **multiline statements require body braces and return**:

```
param => {
  let a = 1; //statement
  return a + param; //statement
}
```

Arrow Functions (III)

- If you use the following syntax to return an object literal from an arrow function, you will get an error.

```
p => {object:literal}
```

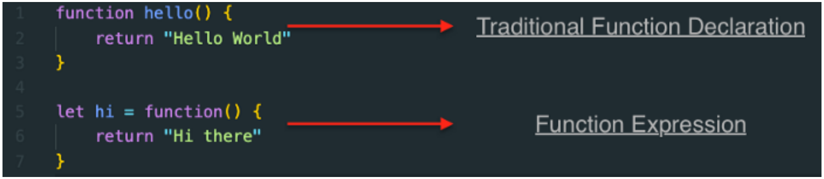
- For example, the following code causes an error.

```
let setColor = color => {value: color };
```

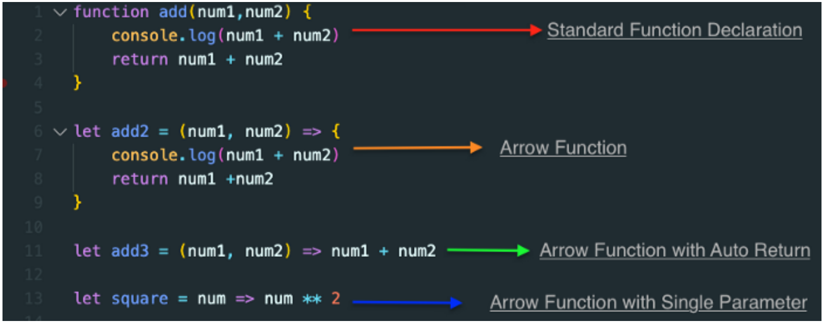
- Since both block and object literal use curly brackets, the JavasScript engine cannot distinguish between a block and an object.
- To fix this, you need to wrap the object literal in parentheses as follows:

```
let setColor = color => ({value: color });
```

Regular, Expression, and Arrow Functions (III)



```
function hello() {
    return "Hello World"
}

let hi = function() {
    return "Hi there"
}
```



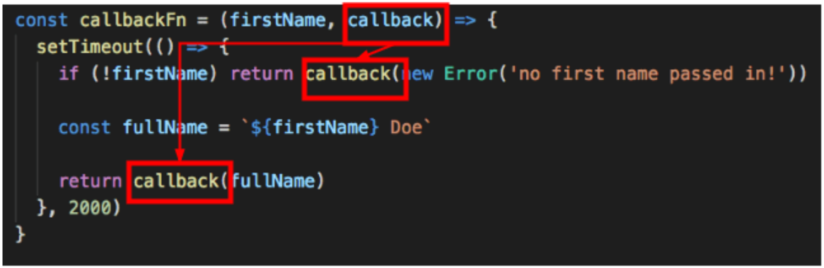
```
function add(num1,num2) {
    console.log(num1 + num2)
    return num1 + num2
}

let add2 = (num1, num2) => {
    console.log(num1 + num2)
    return num1 +num2
}

let add3 = (num1, num2) => num1 + num2

let square = num => num ** 2
```

Standard Function Declaration
Arrow Function
Arrow Function with Auto Return
Arrow Function with Single Parameter

Asynchronous Programming

Asynchronous programming

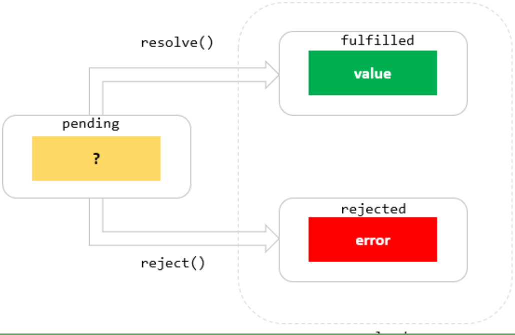
- Asynchronous programming in JavaScript is done with **callbacks**.
- A **callback is a function** that you write and then **pass to some other function**.
- That other function then invokes (“calls back”) your function when some condition is met or some (asynchronous) event occurs.
    - The invocation of the callback function you provide notifies you of the condition or event, and sometimes, the invocation will include function arguments that provide additional details.



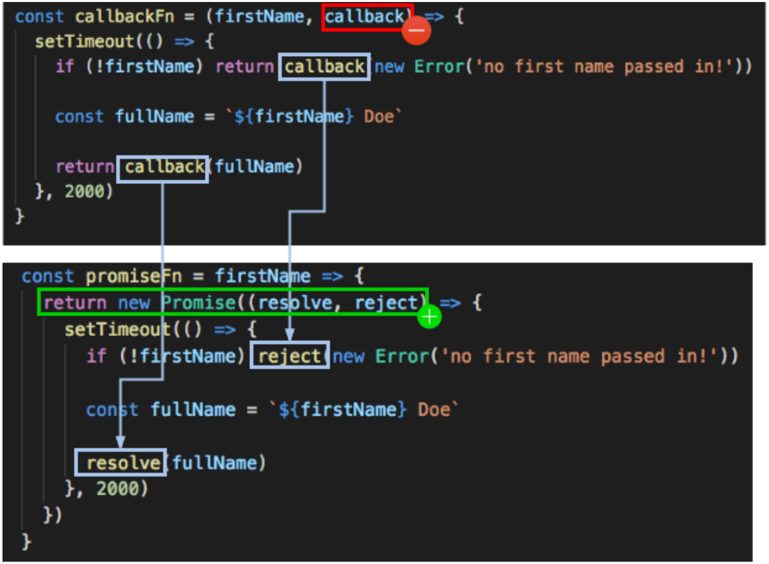
```
const callbackFn = (firstName, callback) => {
  setTimeout(() => {
    if (!firstName) return callback(new Error('no first name passed in!'))

    const fullName = `${firstName} Doe`

    return callback(fullName)
  }, 2000)
}
```

What is promise? (I)

- A native Promise is an object, from which **asynchronous operation messages can be obtained**.
- Promise means **it promises that I will give you a result after a period of time (usually an asynchronous operation)**.
- A promise has three states **pending**(in progress), **fulfilled**(successful) and **rejected**(failed).



What is promise? (II)



```
const callbackFn = (firstName, callback) => {
  setTimeout(() => {
    if (!firstName) return callback(new Error('no first name passed in!'))

    const fullName = `${firstName} Doe`

    return callback(fullName)
  }, 2000)
}
```

```
const promiseFn = firstName => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (!firstName) reject(new Error('no first name passed in!'))

      const fullName = `${firstName} Doe`

      resolve(fullName)
    }, 2000)
  })
}
```

Bibliography

Resources

- David Flanagan, "JavaScript: The Definitive Guide", O'Reilly Media, Inc., 2020
- Adam Boduch and Roy Derks, "React and React Native, Third Edition,A complete hands-on guide to modern web and mobile development with React.js"
- Dan Ward, "React Native Cookbook Second Edition Step-by-step recipes for solving common React Native development problems"
- `https://reactnative.dev/`
- `https://reactjs.org/`
- `https://reactnavigation.org/`