@pravinvelusamy

THE ESSENTIAL JAVASCRIPT CONCEPTS THAT YOU SHOULD UNDERSTAND

PART-1

Core concepts

- Dynamic
- Reference VS Value
- Equality Operator (==) VS. Identity Operator(===)
- Scope
- Hoisting
- Closures
- IIEF
- Callbacks
- Higher-Order Functions

ES6+

- Spread Syntax
- Destructuring
- Rest Syntax
- Classes
- Promises
- Async Await
- Arrow Functions

@pravinvelusamy

Core Concepts

1. Dynamic

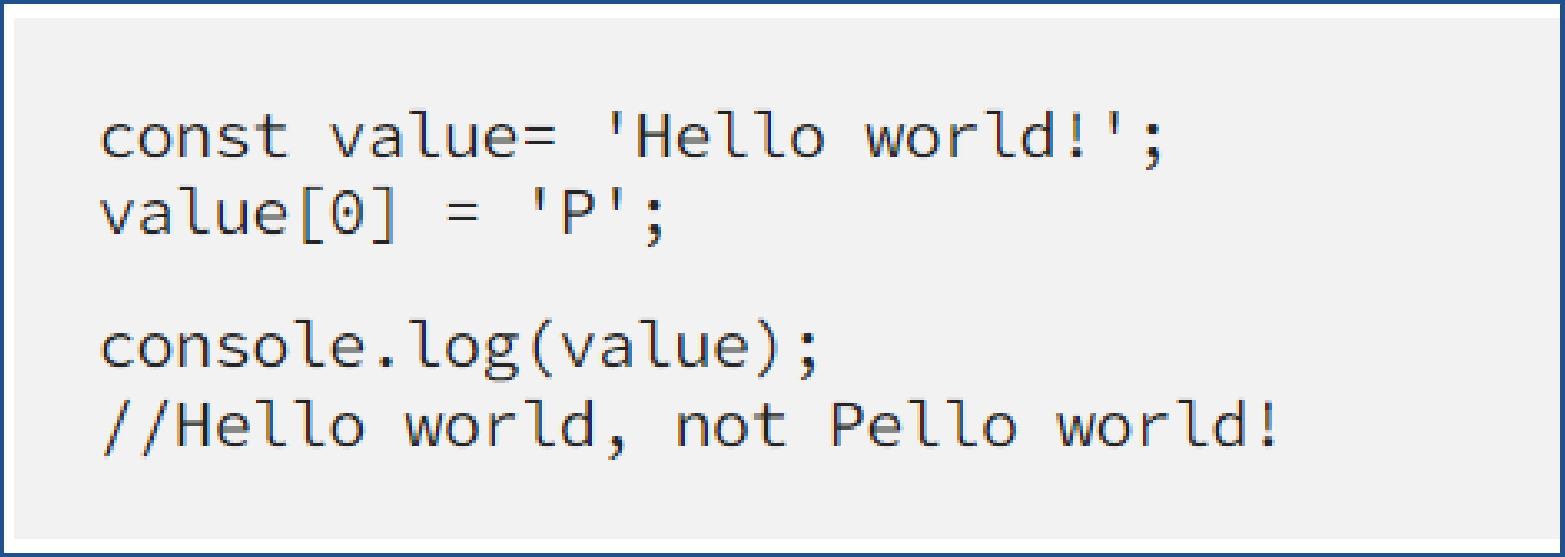
1. We consider JavaScript as a dynamic language because it has dynamic aspects.
2. In JavaScript, pretty much everything is dynamic, for example, all variables are dynamic, or even the code is dynamic.
3. Dynamic typing means that type of variables is determined at runtime and not require the explicit declaration of the variables before they're used.
4. In JavaScript also you can create new variables at runtime with the eval() function.
5. However, eval() is a dangerous function and is completely discouraged.

2. Reference Vs. Value

- In JS, We have Objects(Functions, Arrays) and primitive data types (string, number, boolean, null, and undefined). Primitive data types are passed by value always, and Objects are passed by reference.

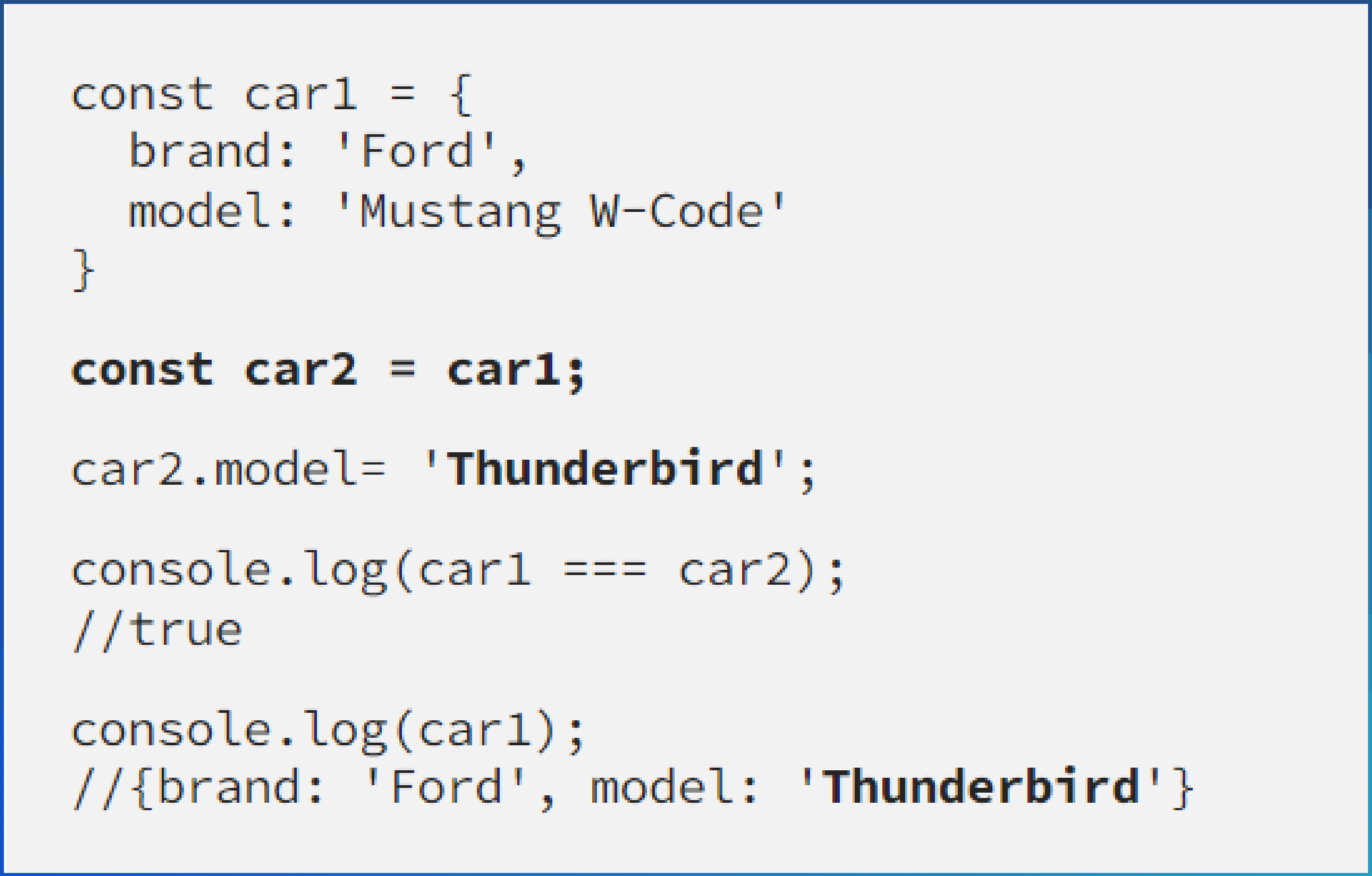
- The primitive values are immutable, which means that once created, they cannot be modified, we create a copy of the original object. Instead, by reference, we create a link to the original.

By value:



```
const value= 'Hello world!';
value[0] = 'P';

console.log(value);
//Hello world, not Pello world!
```

By reference:



```
const car1 = {
  brand: 'Ford',
  model: 'Mustang W-Code'
}

const car2 = car1;

car2.model= 'Thunderbird';

console.log(car1 === car2);
//true

console.log(car1);
//{brand: 'Ford', model: 'Thunderbird'}
```

@pravinvelusamy

3. Equality Operator (==) Vs Identity Operator(===)

The main difference among “==” and “===” operator is that “==” compares variable by making type coercion and === compares variable using strict equality and both the type and the value we are comparing have to be identical.

```
10 === 10
//true

10 == "10"
//true (Although 10 is a number and the "10" a string)
```

@pravinvelusamy

4. Scale

- This concept is one of the most challenging concepts to understand for many new developers.
- One of the most fundamental paradigms of almost all programming languages is the capacity to store values in variables.
- But where do those variables live? Where are they stored? How our program gets them? And finally, what is Scope?
- “Scope” is the area of our code over which an identifier is valid.

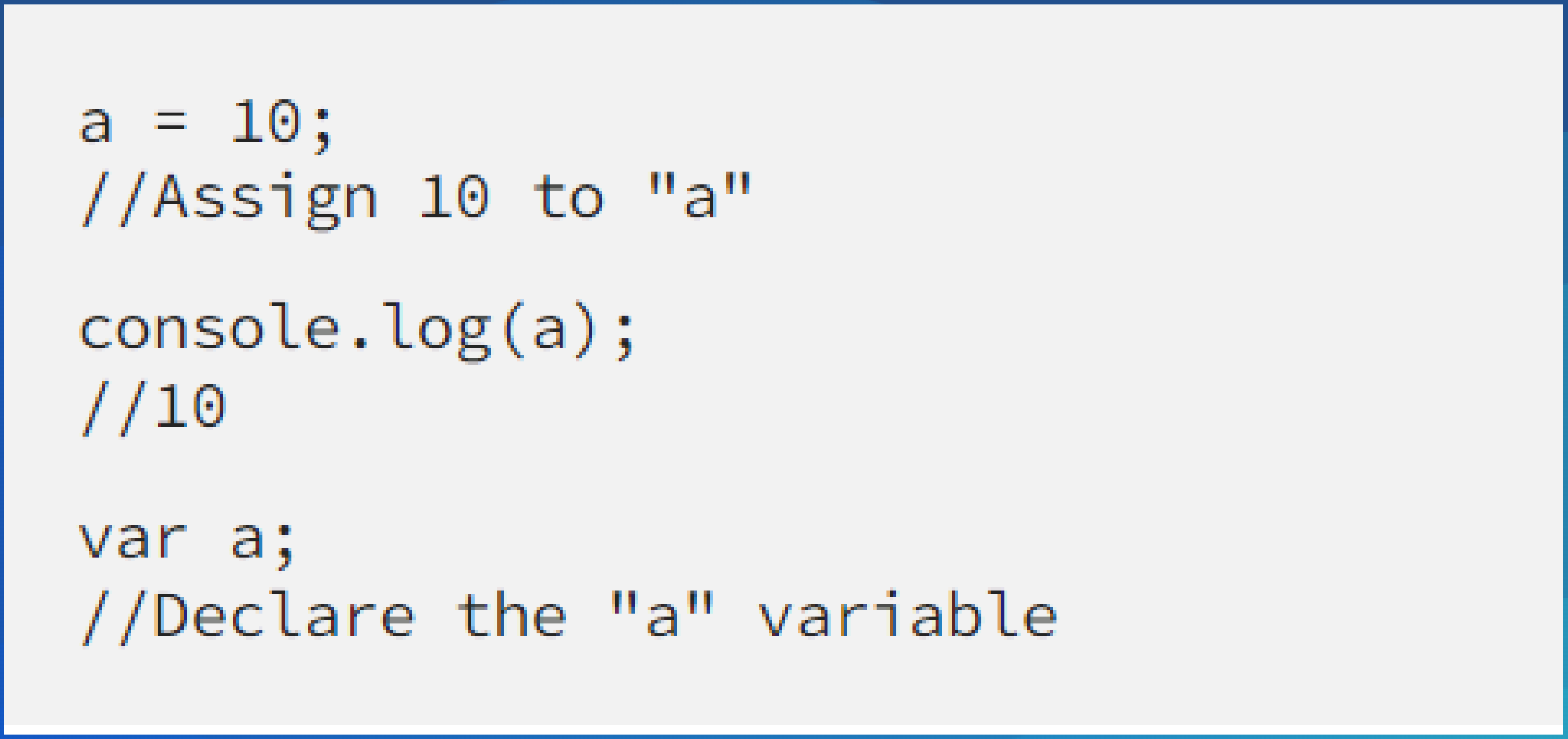
- In JavaScript, we have four types of ScopeScope:

1. **Global ScopeScope:** visible by everything (var)
2. **Function Scope:** visible within a function (var)
3. **Block Scope(ES6+):** visible within a block (let, const)
4. **Module(ES6+):** visible within a module

@pravinvelusamy

5. Hoisting

Hoisting is a mechanism where variables and function declarations are moved to the top of their "Scope" before code execution, and in consequence, variables might actually be available before their declaration.

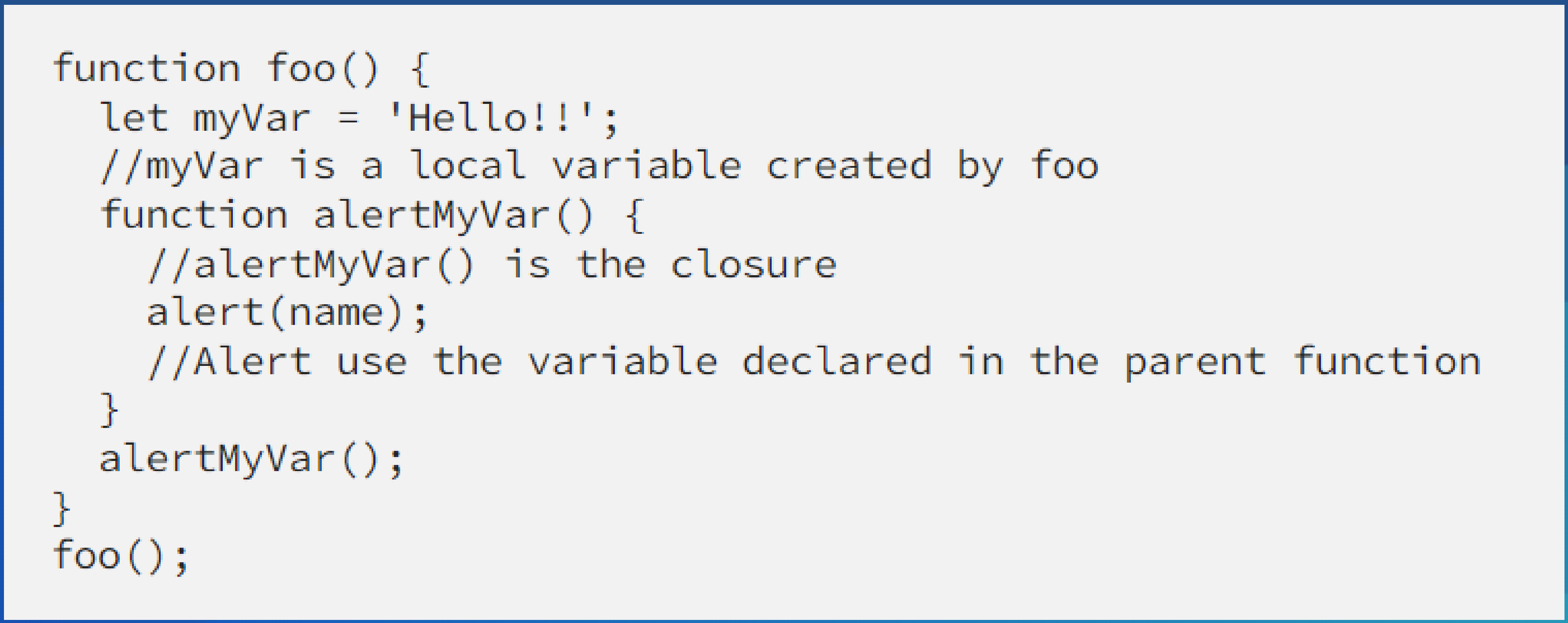


```
a = 10;
//Assign 10 to "a"

console.log(a);
//10

var a;
//Declare the "a" variable
```

@pravinvelusamy

6. Closures

A closure is a mix of a function bundled together (enclosed) with references to its surrounding state that gives you access to an outer function’s Scope from the inner function.



```
function foo() {
  let myVar = 'Hello!!';
  //myVar is a local variable created by foo
  function alertMyVar() {
    //alertMyVar() is the closure
    alert(name);
    //Alert use the variable declared in the parent function
  }
  alertMyVar();
}
foo();
```

@pravinvelusamy

The foo() function creates a local variable called myVar and a function called alertMyVar(). The alertMyVar() function is an inner function that is defined inside the foo() function and is available only within the body of the foo() function. alertMyVar() has access to the variables of outer functions, so alertMyVar() can access to the variable myVar declared in the parent function.

7. IIFE

An Immediately-invoked Function Expression (IIFE) is a process to run functions as soon as they are created. IIFEs are very useful because they protect against polluting the global environment allowing public access to methods while maintaining privacy for variables defined inside the function.

```
(function() {
   let name= "Hello world!";
   alert(name);
  }
)();

//Hello world!
```

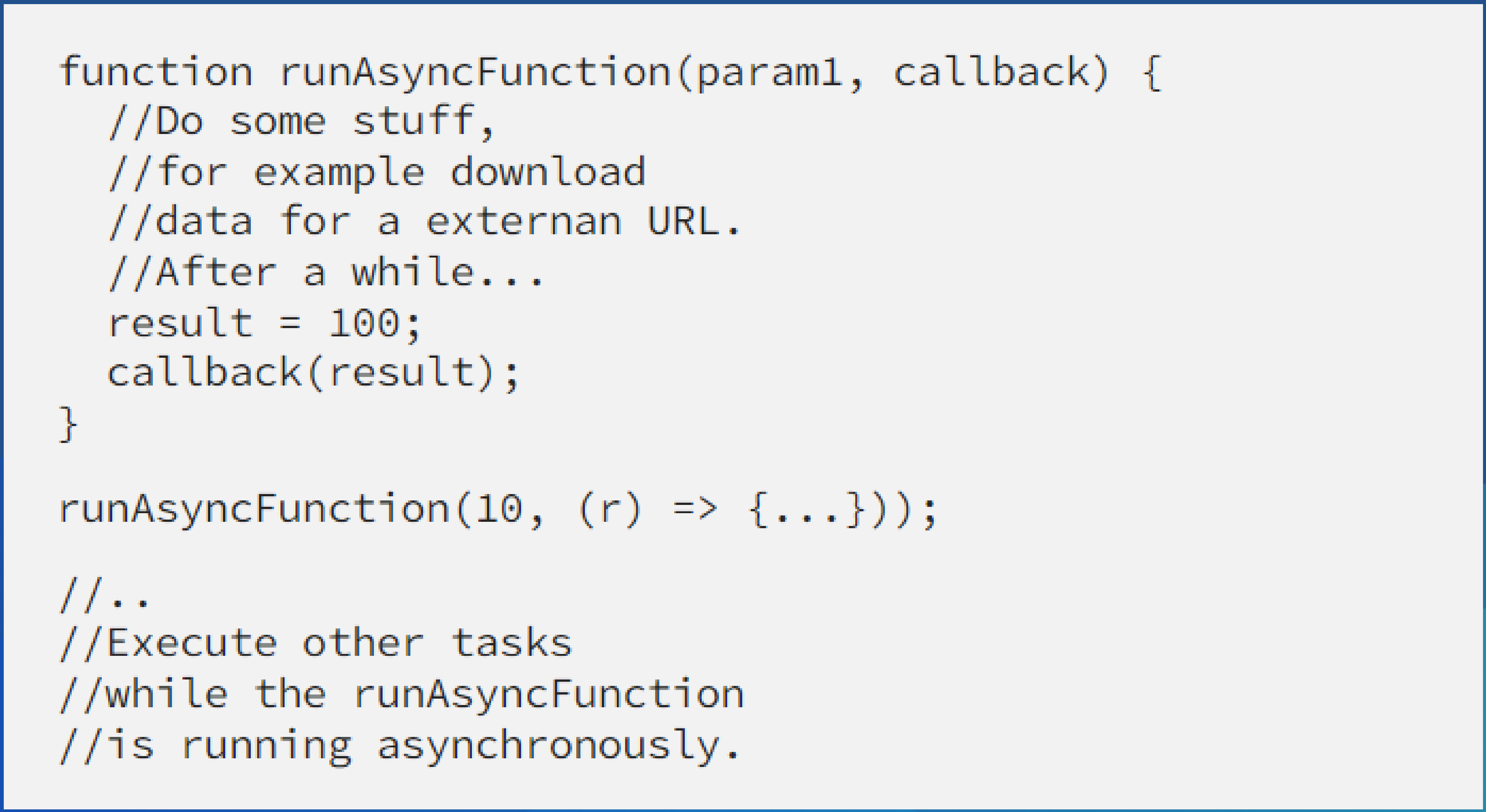
In summary, we use IIFE mainly because of privacy. Any variables declared inside the IIFE function are not available from the outside world.

@pravinvelusamy

8. CallBacks

- In JavaScript, a CallBack is a function that is passed as an argument to other functions, which is then invoked inside the outer function.

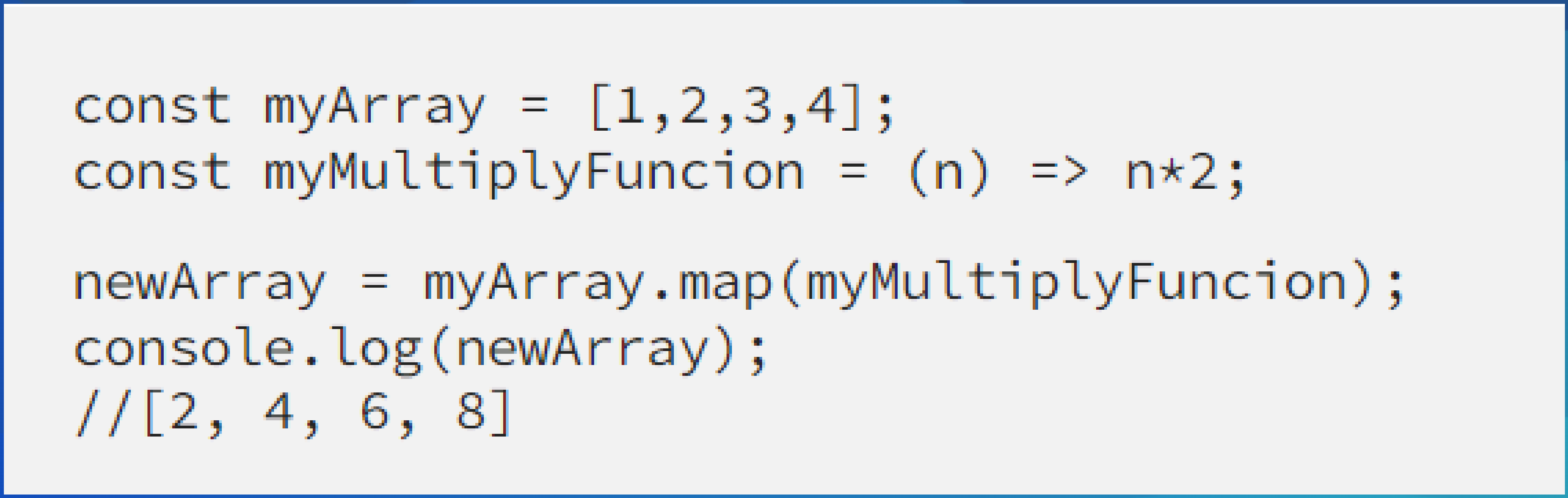
- Callbacks are also closures, and the function that is passed to it is a function that is executed inside the other function as if the Callback was defined in containing function.



```
function runAsyncFunction(param1, callback) {
  //Do some stuff,
  //for example download
  //data for a externan URL.
  //After a while...
  result = 100;
  callback(result);
}

runAsyncFunction(10, (r) => {...}));

//..
//Execute other tasks
//while the runAsyncFunction
//is running asynchronously.
```

8. Higher-Order Functions

- Higher-order functions are functions that receive other functions as arguments or return functions as their results. They are extensively used in JavaScript like in .filter(), .map(), .reduce() or .forEach() functions.



```
const myArray = [1,2,3,4];
const myMultiplyFuncion = (n) => n*2;

newArray = myArray.map(myMultiplyFuncion);
console.log(newArray);
//[2, 4, 6, 8]
```

@pravinvelusamy

- The higher function here is the .map() function that takes myMultiplyFuncion and returns a new array with the values multiplied by two.

- Also, JavaScript allows functions to return other functions as a result. Remember that functions are simply objects, and they can be returned the same as any additional value.

- Although this example is a bit silly because you can use str.toUpperCase() directly, it serves to show how one function can return another:

```
const myToUpperCasse = function(str) {
  return str.toUpperCase();
};

console.log(myToUpperCasse("hello world!"));
//HELLO WORLD!
```

@pravinvelusamy

PART-2

Soon

@pravinvelusamy

Thanks
for
Reading....

Stay Connected :

Pravin Velusamy

@pravinvelusamy