Advanced Constructs: Advanced Function Concepts-2

Relevel
by Unacademy

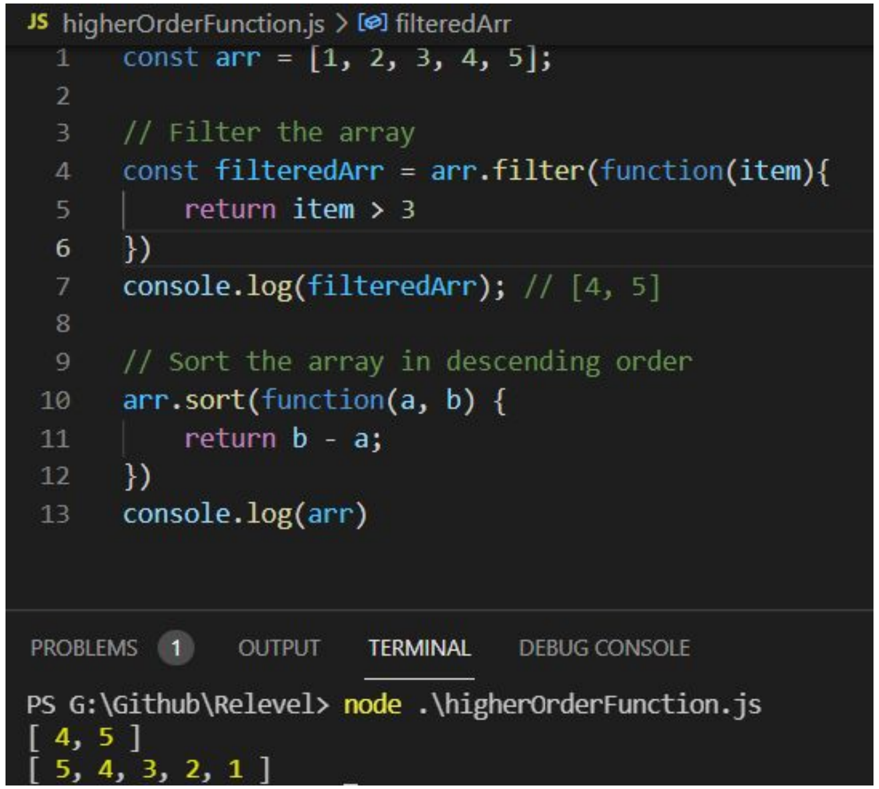
Topics Covered

- Higher Order Functions
- Composability
- Arrow Functions
- Why are functions in JS ‘first-class-citizen’
- IIFE
- Taking User input in JS

Higher Order Function

- Higher Order Function (HOF) are functions that takes other function as argument or return function as a result. I hope you are familiar with Arrays and their functions.
- Let me take you into deeper in this HOF concept. Arrays function such as map, filter, sort, reduce, forEach and so on are Higher Order function because these functions accept function as an argument and return desired output.

Programming Example


higherOrderFunction.js > filteredArr

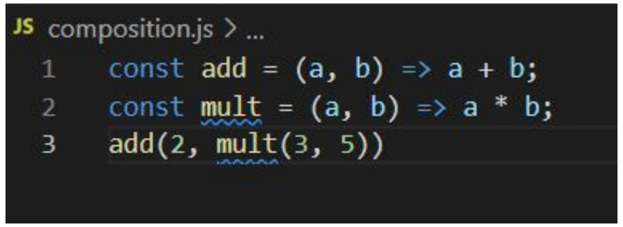
```javascript
const arr = [1, 2, 3, 4, 5];

// Filter the array
const filteredArr = arr.filter(function(item){
    return item > 3
})
console.log(filteredArr); // [4, 5]

// Sort the array in descending order
arr.sort(function(a, b) {
    return b - a;
})
console.log(arr)
```

```
PS G:\Github\Relevel> node .\higherOrderFunction.js
[ 4, 5 ]
[ 5, 4, 3, 2, 1 ]
```

- In this example, using the inbuilt array manipulation functions,
- Using filter function which accept function as an argument with one argument in the function and do the filtration process
- Using sort function to arrange the array in the descending order and that sort function will accept function as an argument with two argument and do the sorting.

Composability

- Function composition is a mechanism of combining multiple simple functions to build a more complicated one. The result of each function is passed to the next one.
- In mathematics, we often write something like: f(g(x)). So this is the result of g(x) that is passed to f. In programing we can achieved the composition by writing something similar.
- Let's take a quick example. Suppose I need to make some arithmetic by doing the following operation: 2 + 3 * 5. As you may know, the multiplication has the priority over the addition. So you start by calculating 3 * 5 and then when add 2 to the result.



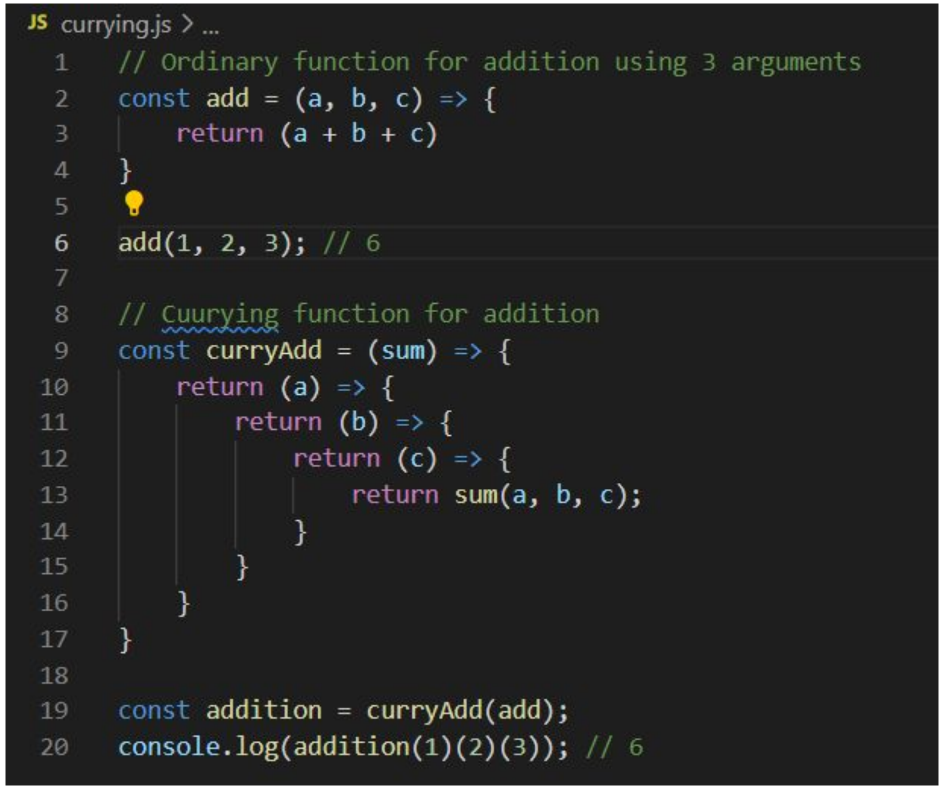
composition.js > ...

```javascript
const add = (a, b) => a + b;
const mult = (a, b) => a * b;
add(2, mult(3, 5))
```

When we use Composability

- Let me explain with real time example,
- Think of an industrial plant that produce bottles of cool drinks; first there is the operation (or function) f1f1 that puts the cool drinks inside the bottle, followed by the operation f2f2 that close the bottle with the cap.
- In the above example we need to follow certain series of action, in the similar way we need to implement certain functionality which needs to be followed one after another.

Currying



currying.js > ...

```js
// Ordinary function for addition using 3 arguments
const add = (a, b, c) => {
    return (a + b + c)
}

add(1, 2, 3); // 6

// Cuurying function for addition
const curryAdd = (sum) => {
    return (a) => {
        return (b) => {
            return (c) => {
                return sum(a, b, c);
            }
        }
    }
}

const addition = curryAdd(add);
console.log(addition(1)(2)(3)); // 6
```

- Currying is when you break down a function that takes multiple arguments into a series of function that each take only one argument.
- In the below example **curryAdd** function is returning a series of function and at the last function it is returning the value.

Currying

- Let me explain more simpler, If you want to buy a chocolate cake in shop.
- What is the process go out of the home -> take a bus -> find the shop -> check whether the chocolate cake is available or not -> if available then buy.
- In the above scenario, let us assume this as a task and we can split each and every in single function calling every function if every thing is good we will be getting the cake but any one of the task is failed we won't get cake and get proper reason and it is easy to find.
- Instead of that if we put all task in a single function and passing multiple argument that is hard to manage, that's why currying comes into picture.

Arrow Function

- Traditional function expressions are function [name]([param1[, param2[, ..., paramN]]]) {
  ```
  statements
  }
  ```

- The difference between named and unnamed functions are, If function name is omitted, it will be the variable name (implicit name). If function name is present, it will be the function name (explicit name).

- Unnamed functions are called as anonymous function.

- Arrow function is a different form of writing function compare to traditional function and it was introduced in the year 2015 ES6 (ECMAScript6) edition. They are less verbose than traditional function expression.

- Let's have a quick example and comparison of Arrow function with traditional function.

- Arrow functions are new way to write an anonymous function and are similar to **Lamda** function in other programming languages.

- Syntax: (argument) => { ... Logic}

Difference between Arrow and Regular Function

	Regular Function	Arrow Function

Constructor	function Car(color) {
	  this.color = color;
	}
	const redCar = new Car('red');
	redCar instanceof Car; // => true
	
	We can create an instance for Car
		const Car = (color) => {
		  this.color = color;
		};
		const redCar = new Car('red'); // TypeError: Car is not a constructor
		
		We cannot create an instance

Argument Object	let user = {
	    show(){
	        console.log(arguments); // 1, 2, 3
	    }
	};
	user.show(1, 2, 3);
	Argument object are available
		let user = {
		        show_ar : () => {
		        console.log(...arguments); // error
		    }
		};
		user.show_ar(1, 2, 3);
		Argument object are not available

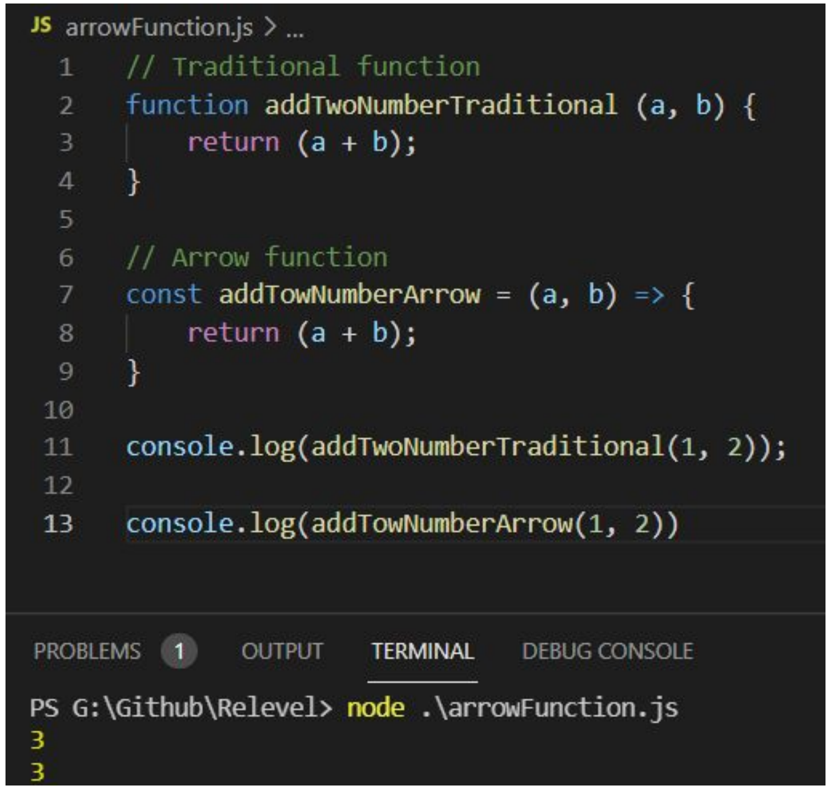
Difference between Arrow and Regular Function

	Regular Function	**Arrow Function**

this Keyword	let user = {
	    name: "Relevel",
	    regularfn(){
	        console.log("hello " + this.name); // 'this' binding here
	    }
	};
	user.regularfn();
	this binding here
		let user = {
		    name: "Relevel",
		    arrowfn:()=> {
		        console.log("hello " + this.name); // no 'this' binding here
		    }
		 };
		user.arrowfn();
		no this binding here

Implicit return	normalfn () {
	    12;
	    return;
	}
	normalfn(); // undefined
		const arrowfn = () => 44
		
		arrowfn(); // 44

Programming Example



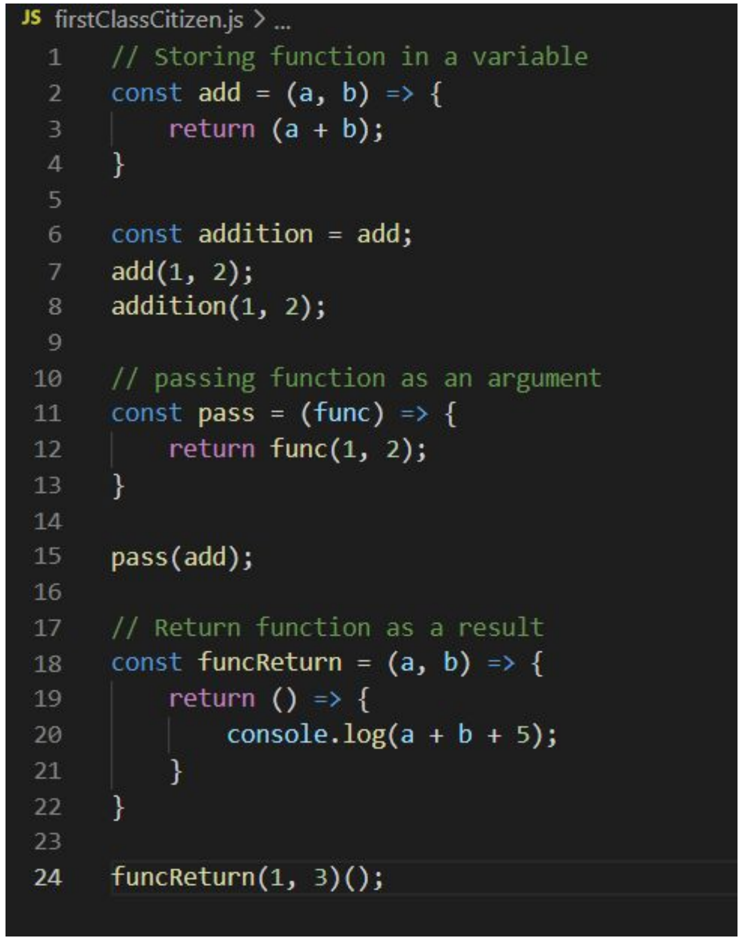
```
JS arrowFunction.js > ...
// Traditional function
function addTwoNumberTraditional (a, b) {
    return (a + b);
}

// Arrow function
const addTowNumberArrow = (a, b) => {
    return (a + b);
}

console.log(addTwoNumberTraditional(1, 2));

console.log(addTowNumberArrow(1, 2))
```

```
PS G:\Github\Relevel> node .\arrowFunction.js
3
3
```

In this program, function **addTwoNumberTraditional** is the function expression we have seen in the class which is the traditional way of declaring the function.

The function **addTwoNumberArrow** is called as Arrow function because in the expression we are using => and this function is assigning to a variable using **const** keyword, so hoisting will consider this as a variable.

Function - A first class citizen



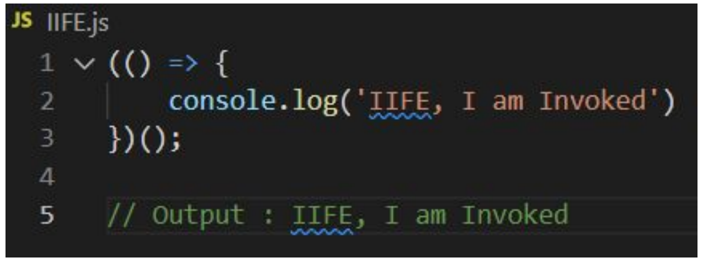
JS firstClassCitizen.js > ...

```javascript
// Storing function in a variable
const add = (a, b) => {
    return (a + b);
}

const addition = add;
add(1, 2);
addition(1, 2);

// passing function as an argument
const pass = (func) => {
    return func(1, 2);
}

pass(add);

// Return function as a result
const funcReturn = (a, b) => {
    return () => {
        console.log(a + b + 5);
    }
}

funcReturn(1, 3)();
```

Function in javascript are first class citizen which means you can store function in a variable, pass function as an argument, return function as a result.

IIFE (Immediately Invoked Function Expression)

- IIFE is a function that runs as soon as it is declared. Example will help you to understand what is IIFE.
- This is similar to declaring the function and invoking the function but only difference here is it will invoke as soon as it is declared.



```
(() => {
    console.log('IIFE, I am Invoked')
})();

// Output : IIFE, I am Invoked
```

Advantages

Secure Variables Scope

As you know var keyword scope is global so to secure the reference we can use IIF

```
(function () {
  var greeting = 'Good morning! How are you today?';
  console.log(greeting); // Good morning! How are you today?
})();console.log(greeting); // error: Uncaught ReferenceError: greeting is not definedAs you can see in the example
```

above, what happens in the IIFE scope, stays in the IIFE scope. You can't use the variable defined inside IIFE from the outside.

Avoid Naming Conflict

Using many JavaScript libraries can cause conflicts because some of them might export an object with the same name.

Let's say you're using jQuery. We all know it export $ as its main object. So, if there's any library in your dependencies using $ as its exported object as well, a conflict will occur.

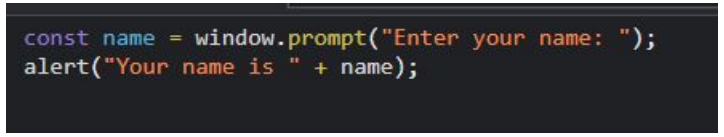
Fortunately, you can use IIFEs to solve this problem by applying the aliasing technique:

```
(function ($) {
  // You're safe to use jQuery here
})(jQuery);
```

By wrapping your code inside an IIFE that takes jQuery as an argument, we will make sure that the $ symbol now refers to jQuery, not other libraries.

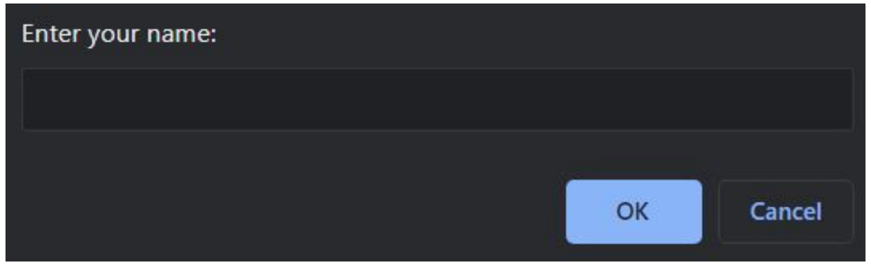
Taking user input in JS

- We can take user input in various way using javascript, but as a beginner we should be aware of the prompt which will get user input and do the logic as per the next process.
- JavaScript has a few window object methods that you can use to interact with your users. The prompt() method lets you open a client-side window and take input from a user.

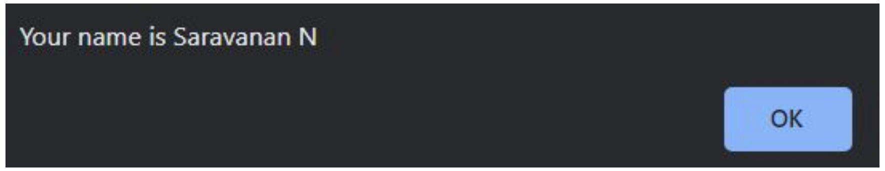


```
const name = window.prompt("Enter your name: ");
alert("Your name is " + name);
```

Window object will prompt for user input and asking the user to enter name



Enter your name:

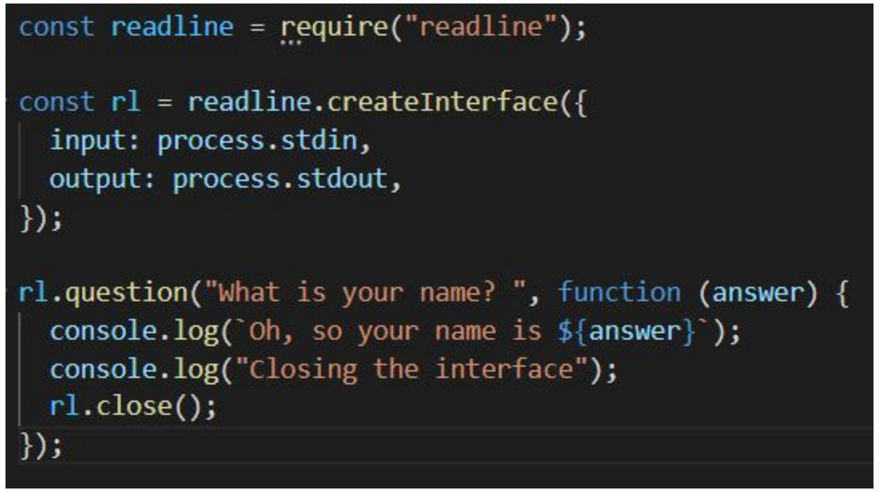


Your name is Saravanan N

As per the 2nd line code the will show the name which user had given as an alert.

Get input using Node

- To access input from user, you need to create an Interface instance that is connected to an input stream.
- You create the Interface using readline.createInterface() method, while passing the input and output options as an object argument.



```
const readline = require("readline");

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

rl.question("What is your name? ", function (answer) {
  console.log(`Oh, so your name is ${answer}`);
  console.log("Closing the interface");
  rl.close();
});
```

Thank You