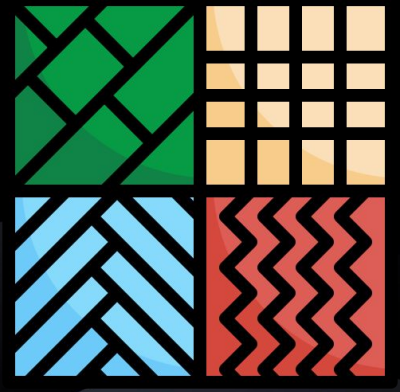
JavaScript

STUDIA PODYPLOMOWE
POLITECHNIKA BIAŁOSTOCKA


#3

Design Patterns 1

with JavaScript



What are design patterns?

- Solution for common problems
- Way of thinking with code
- Application of a style/pattern of coding
- You don’t have to use them all (and shouldn’t)
- But if you encounter a certain problem, you will know how to approach it thanks to design patterns!

Design pattern categories

Creational:

- Singleton
- Module
- Prototype
- Factory

Structural:

- Adapter
- Decorator
- Facade
- Proxy

Behavioral:

- Chain of responsibility
- Iterator
- Mediator
- Observer

IIFE recap

- Immediately Invoked Function Expression
- Declare and run functions in the same place
- Used mainly for its closure

iife.js

```
// IIFE

(function IIFEexample(){
    console.log('This is my IIFE example');
})();

```

Closure

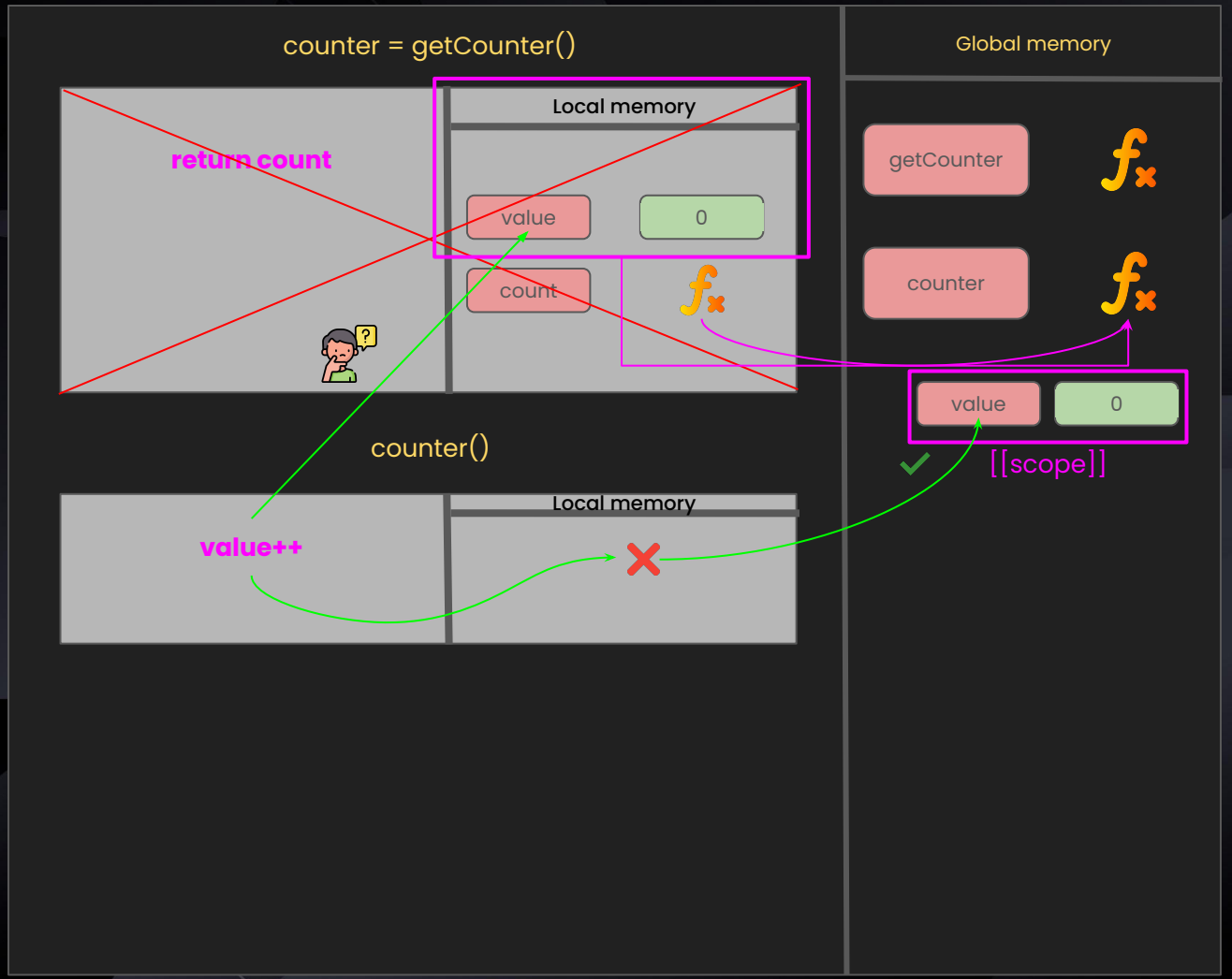
- Access to the scope of a function at the moment of its creation
- When you create a **function** inside another **function**, and return it (it can be a set of **function**)
- The returned **function** maintains access to the variables in the scope during the creation
- Although the context of the creating **function** is gone

closure.js

```
// Closure

function getCounter() {
    let value = 0;

    function count() {
        value++;
        console.log(value);
    }

    return count;
}

const counter = getCounter();

counter(); // 1
counter(); // 2
counter(); // 3

```

closure.js

```
1  // Closure
2
3  function getCounter() {
4      let value = 0;
5
6      function count() {
7          value++;
8          console.log(value);
9      }
10
11     return count;
12 }
13
14 const counter = getCounter();
15
16 counter(); // 1
17 counter(); // 2
18 counter(); // 3
19
```



DEMO

Module pattern

- Separate certain logical pieces of code
- Allows encapsulation - only part of the module is public
- Avoid namespace collisions

module.js

```
// Module pattern

const nameModule = (function nameModuleIIFE() {
    let name = '';

    return {
        setName(inputName) {
            name = inputName;
        },
        logName() {
            console.log(name);
        },
    };
})();

nameModule.setName('Piotr');
nameModule.logName(); // Piotr

```

Revealing Module pattern

- The purpose of revealing module pattern is the same as the normal module pattern
- The difference is in the functions declarations location
- Easier changes in the privacy of fields

module.js

```
// Revealing Module pattern

const nameModule = (function nameModuleIIFE() {
    let name = '';

    function setName(inputName) {
        name = inputName;
    }

    function logName() {
        console.log(name);
    }

    return {
        setName,
        logName,
    };
})();

nameModule.setName('Piotr');
nameModule.logName(); // Piotr

```

ES6 Modules

- Makes the old modules patterns obsolete
- Good to know for better closure understanding

module.js

```
// ES6 Module

// nameModule.js

let name = '';

function setName(inputName) {
    name = inputName;
}

function logName() {
    console.log(name);
}

export { setName, logName };

// index.js

import * as nameModule from './nameModule.js';

nameModule.setName('Piotr');
nameModule.logName(); // Piotr

```

Singleton

- A piece of code that’s supposed to be run once
- Or a value that should be created only once and shared throughout the program

singleton.js

```
// Singleton

function getCurrentUser() {
    return {
        name: "Bob",
        role: "Admin",
    };
}

// index.js (imports omitted)

const currentUser = getCurrentUser();
currentUser.role = 'Moderator';

// someService.js (imports omitted)

const currentUser = getCurrentUser();
console.log(currentUser.role); // Admin
```

singleton.js

```
// Singleton

let currentUser;

function getCurrentUser() {
    if (!currentUser) {
        currentUser = {
            name: "Bob",
            role: "Admin",
        }
    }

    return currentUser;
}

// index.js (imports omitted)

const currentUser = getCurrentUser();
currentUser.role = 'Moderator';

// someService.js (imports omitted)

const currentUser = getCurrentUser();
console.log(currentUser.role); // Moderator
```

Wrapper (function)

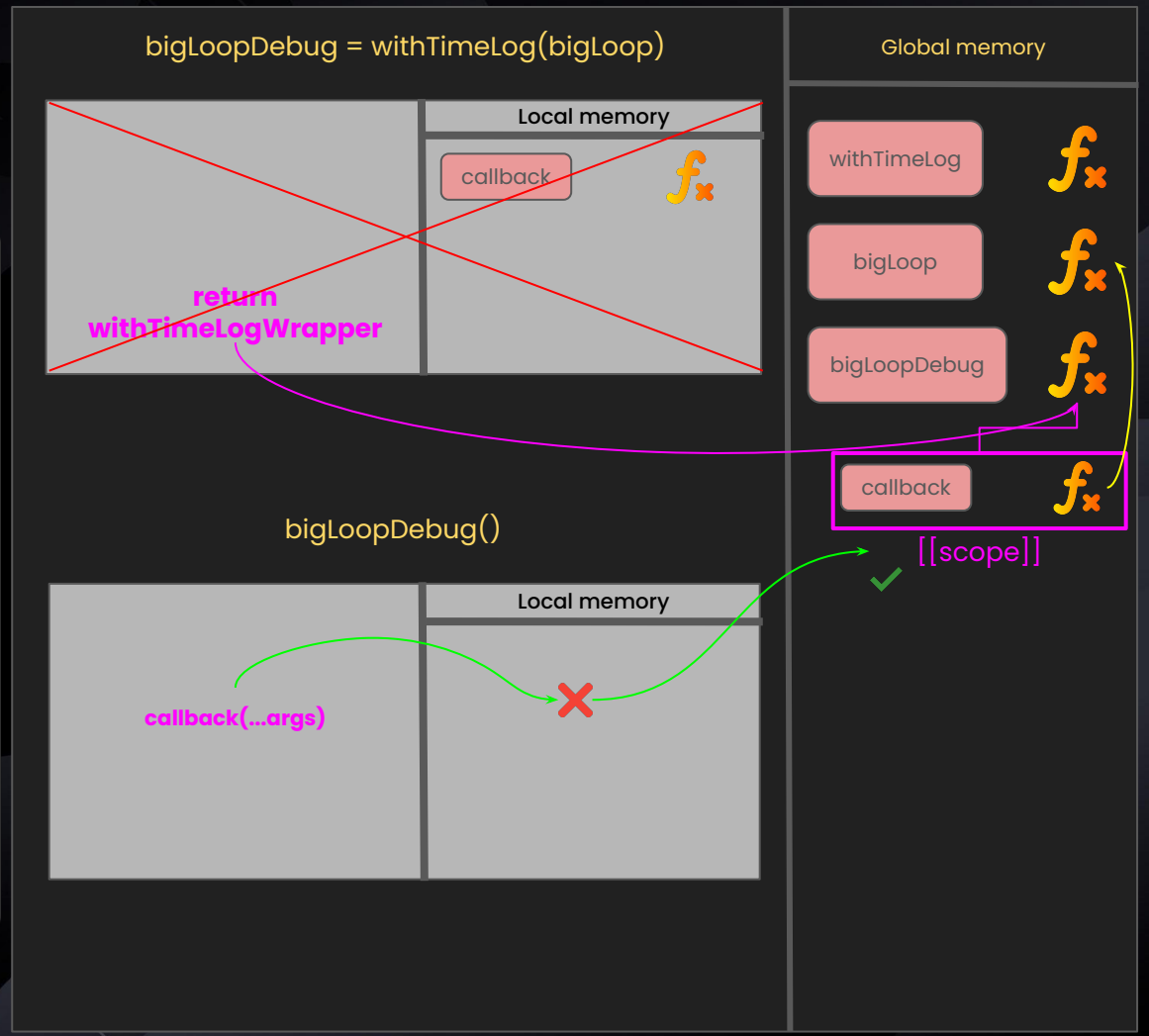
- Function wrapping another function
- Allows to add additional functionality during the function's run

wrapper.js

```
// Wrapper

function withTimeLog(callback) {
    return function withTimeLogWrapper(...args) {
        console.time(callback.name);
        callback(...args);
        console.timeEnd(callback.name);
    };
}

function bigLoop() {
    let count = 0;

    for (let i = 0; i < 1_000_000_000; i++) {
        count++;
    }
}

const bigLoopDebug = withTimeLog(bigLoop);
bigLoopDebug();
```

wrapper.js

```
1  // Wrapper
2
3  function withTimeLog(callback) {
4      return function withTimeLogWrapper(...args) {
5          console.time(callback.name);
6          callback(...args);
7          console.timeEnd(callback.name);
8      };
9  }
10
11 function bigLoop() {
12     let count = 0;
13
14     for (let i = 0; i < 1_000_000_000; i++) {
15         count++;
16     }
17 }
18
19 const bigLoopDebug = withTimeLog(bigLoop);
20 bigLoopDebug();
```



Curry

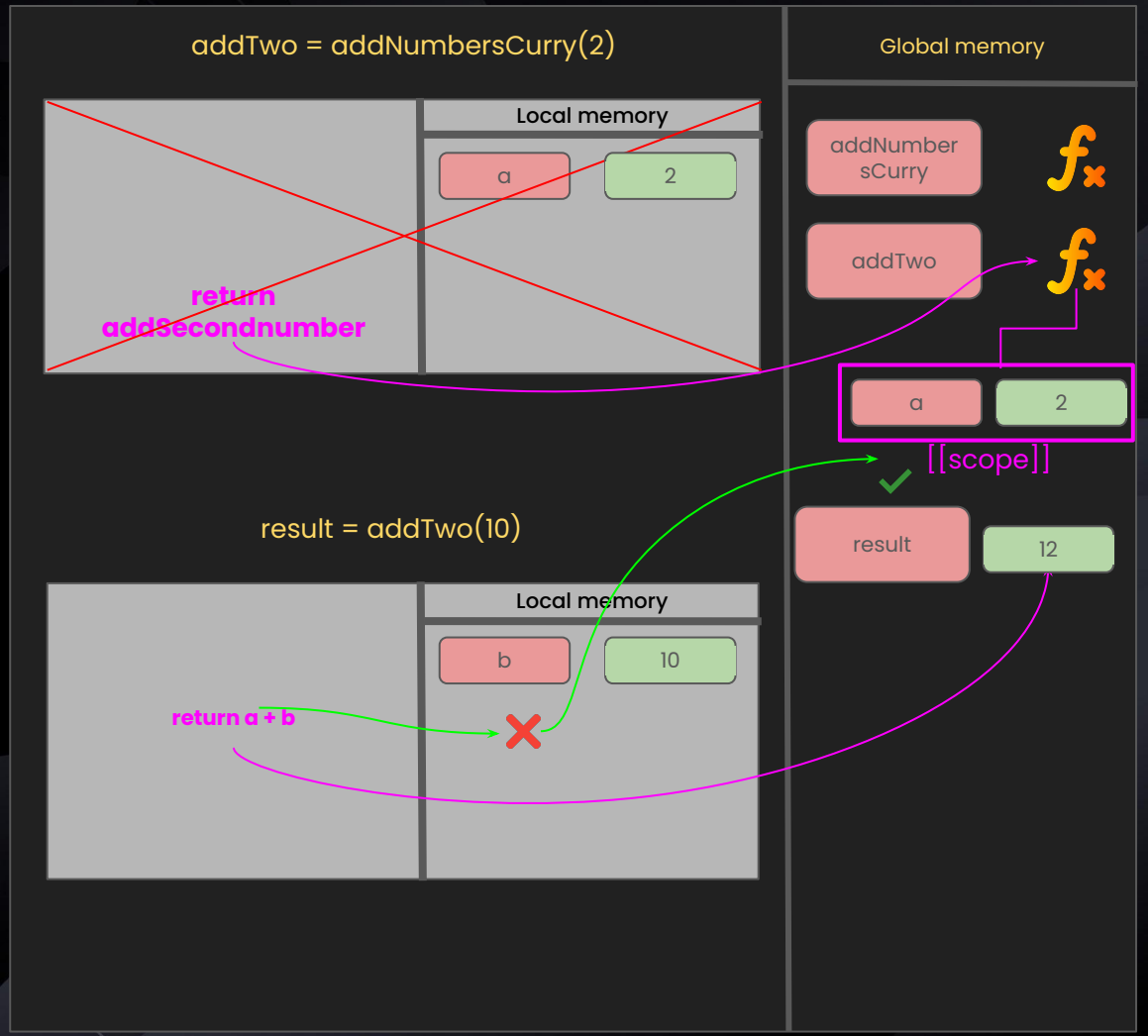
- Functional programming's bread and butter
- Unary function returning another unary function
- Unary = one parameter
- A way of creating a more specialized function out of a less specialized one
- Relies on closure

curry.js

```
// Curry

function addNumbers(a, b) {
    return a + b;
}

// Curried verison

function addNumbersCurry(a) {
    return function addSecondNumber(b) {
        return a + b;
    };
}

const addTwo = addNumbersCurry(2);

const result = addTwo(10);
console.log(result); // 12

```

curry.js

```
1  // Curry
2
3  function addNumbers(a, b) {
4      return a + b;
5  }
6
7  // Curried verison
8
9  function addNumbersCurry(a) {
10     return function addSecondNumber(b) {
11         return a + b;
12     };
13 }
14
15 const addTwo = addNumbersCurry(2);
16
17 const result = addTwo(10);
18 console.log(result); // 12
19
```



curry.js

```
// Curry

// Async omitted

function getFromApi(endpoint, options, callback) {
    // get data from endpoint based on options
    // and runs callback with the data
}

// Curried Version

function getFromApiCurry(endpoint) {
    return function getFromEndpoint(options) {
        return function runResultWith(callback) {
            // get data from endpoint based on options
            // and runs callback with the data
        }
    }
}

getFromApi('/users', { id: 100 }, console.log);

getFromApiCurry('/users')({ id: 100 })(console.log);

const getUser = getFromApiCurry('/users');
const getCurrentUser = getUser({ id: 100 });
const logCurrentUser = getCurrentUser(console.log);

```

Composition (functional)

- Process of composing small units into bigger units
- Input of one function comes from the output of previous one

composition.js

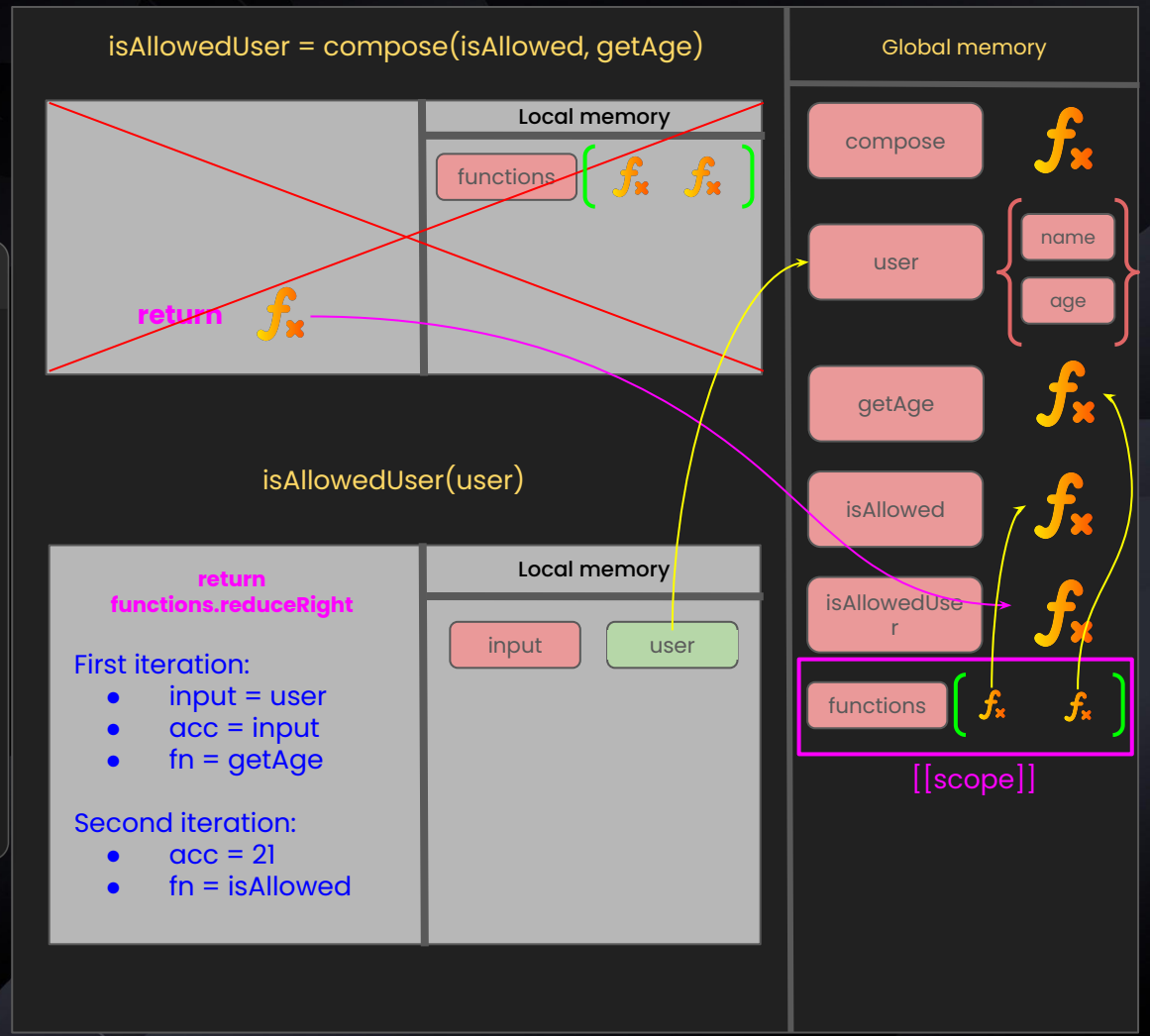
```
// Function declaration

function compose(...functions) {
  return function (input) {
    return functions.reduceRight(function (acc, fn) {
      return fn(acc);
    }, input);
  };
}

// Arrow example

const compose =
  (...functions) =>
    (input) =>
      functions.reduceRight((acc, fn) => fn(acc), input);
```

composition.js

```
// Functional composition

function compose(...functions) {
  return function (input) {
    return functions.reduceRight(function (acc, fn) {
      return fn(acc);
    }, input);
  };
}

const user = { name: 'Bob', age: 21 };

const getAge = (user) => user.age;
const isAllowed = (age) => age >= 18;

const isAllowedUser = compose(isAllowed, getAge);

console.log(isAllowedUser(user)); // true
```

composition.js

```js
// Functional composition

function compose(...functions) {
  return function (input) {
    return functions.reduceRight(function (acc, fn) {
      return fn(acc);
    }, input);
  };
}

const user = { name: 'Bob', age: 21 };

const getAge = (user) => user.age;
const isAllowed = (age) => age >= 18;

const isAllowedUser = compose(isAllowed, getAge);

console.log(isAllowedUser(user)); // true
```



Decorator

- Altering or augmenting and object/class/method/parameter
- Usually seen in TypeScript with @
- ‘Intercept’ calls to the methods and alter them
- Add functionality to a class
- Change some configuration
- etc.


decorator.js
// Decorator
function addLogMe(object) {
object.logMe = function objectLogger() {
console.log(this.name);
};
}
class Car {
constructor(name) {
this.name = name;
}
}
const myCar = new Car('Toyota');
addLogMe(myCar);
myCar.logMe(); // Toyota

decorator.js

```
// Decorator

function addLogMe(inputClass) {
    inputClass.prototype.logMe = function objectLogger() {
        console.log(this.name);
    };
}

class Car {
    constructor(name) {
        this.name = name;
    }
}

addLogMe(Car);

const myCar = new Car('Toyota');
myCar.logMe(); // Toyota

```

decorator.js

```
// Decorator Typescript

@Controller('banks')
@UseGuards(JwtCookieGuard, RolesGuard)
export class BanksController {
    constructor(private readonly bankService: BanksService) { }

    @Post()
    @Roles(Role.admin)
    @HttpCode(HttpStatus.CREATED)
    create(@Body() createBankDto: CreateBankDto) {
        return this.bankService.create(createBankDto);
    }

    @Get()
    @Roles(Role.admin, Role.moderator)
    findAll() {
        return this.bankService.findAll();
    }

    @Get(':id')
    @Roles(Role.admin)
    async findOne(@Param('id') id: string) {
        const foundBank = await this.bankService.findOne(id);

        if (!foundBank) {
            throw new HttpException('No such bank', HttpStatus.NOT_FOUND);
        }

        return foundBank;
    }
}

```

Mixin

- Adding functionality to an object by mixing it with another object

mixin.js

```
// Mixin

const speakMixin = {
    speak() {
        console.log(this.message);
    },
};

const obj = {
    message: 'Hello, world!',
};

Object.assign(obj, speakMixin);

obj.speak(); // Hello, world!

```


THE
END