Session 26

Mohamed Emary

July 4, 2024

1 Revision

1.1 Higher Order Functions

Suppose I have a collection of products:

1. To get the total cost of all products we can use the `reduce` function.
2. To get products with a price greater than 1000 we can use the `filter` function.
3. To get the first product with a price greater than 1000 we can use the `find` function.
4. To add 10% tax to all products we can use the `map` function.

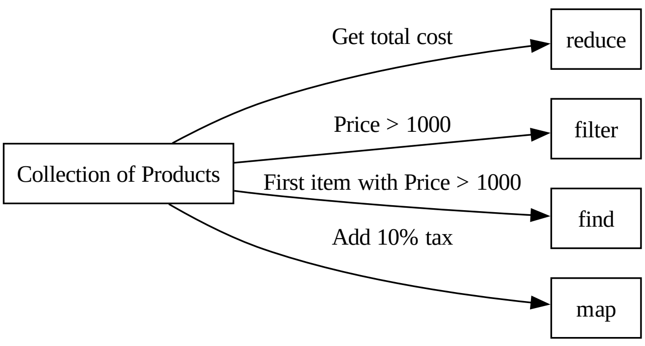

Figure 1: Higher Order Functions

1.2 Prototype

Objects in JS inherit properties from other objects. This is called prototypal inheritance. Every object has a prototype object, which acts as a template object that it inherits methods and properties from.

The prototype object can either be another object or `null`. If it is `null`, the object has no prototype and therefore does not inherit any properties or methods.

Array object inherits from `Array.prototype` and `Array.prototype` inherits from `Object.prototype` which is the root object and its prototype is `null`.

Figure 2: Prototype

The same with numbers, strings, and booleans.

2 Object Oriented Programming (OOP)

OOP is a programming paradigm based on objects. It makes your code more organized, easier to read, and maintain.

The most important advantage of OOP is that it simulates the real world. It allows you to break down your software into smaller parts, making it easier to solve complex problems.

For example to model a hospital management system, you can create classes for `Doctor`, `Patient`, `Nurse`, `Receptionist`, etc.

There are two main ways to implement OOP:

- **Class based OOP** (Most Programming Languages): In this paradigm, we use classes to define objects. A class is a blueprint for creating objects. An object is an instance of a class.
- **Prototype based OOP** (Used in JS): In this paradigm, we use prototypes to define objects. A prototype is a template object that an object inherits properties and methods from.

2.1 Class based OOP

In class based OOP, we create classes for the important entities in our application, and then we create objects from these classes.

For example, to model the hospital management system, we can create a class for `Doctor` and then create an object for each doctor, same with `Patient`, `Nurse`, etc.

Each doctor should have a `name`, `age`, and `salary`. We can define these properties in the `Doctor` class.

2.2 Prototype based OOP

Consider this function:

```
function Doctor(name, age, salary) {
  let doctor = {};
```

```
  doctor.name = name;
  doctor.age = age;
  doctor.salary = salary;
  return doctor;
}
let drAhmed = Doctor('Ahmed', 30, 5000);
```

In this example, `Doctor` is a function that creates a doctor object.

2.2.1 Constructor Functions

In JS, we use constructor functions to create objects. A constructor function is like a blueprint for creating objects.

```
function Doctor(name, age, salary) {
  this.name = name;
  this.age = age;
  this.salary = salary;
}
let drAhmed = new Doctor('Ahmed', 30, 5000);
```

This is the same as the previous example, but we use the `new` operator to create the object, and `this` to refer to the object being created.

To create functions that are shared between all objects created from a constructor function, we use the `prototype` property.

```
function Doctor(name, age, salary) {
  this.name = name;
  this.age = age;
  this.salary = salary;
}

Doctor.prototype.sayHi = function() {
  console.log('Hi, I am ' + this.name);
};

Doctor.prototype.getSalary = function() {
  console.log('My salary is ' + this.salary);
};

let drAhmed = new Doctor('Ahmed', 30, 5000);
let drAli = new Doctor('Ali', 35, 6000);

drAhmed.sayHi(); // Hi, I am Ahmed
drAli.sayHi(); // Hi, I am Ali
```

But why not to just add the functions directly to the object like what we did with the properties?

- Because if we have many objects, each object will have its own copy of the function which is a waste of memory.
- With properties it's fine because each object has its own values for the properties, for example the `name` is different for each object, but with functions it's the same implementation

for all objects, so no need to have a copy for each object.

2.2.2 Sugar Syntax

In ES6, we have a sugar syntax for creating classes, but under the hood, it's the same as the constructor functions.

```
class Doctor {
  constructor(name, age, salary) {
    this.name = name;
    this.age = age;
    this.salary = salary;
  }

  sayHi() {
    console.log('Hi, I am ' + this.name);
  }

  getSalary() {
    console.log('My salary is ' + this.salary);
  }
}

let drAhmed = new Doctor('Ahmed', 30, 5000);
let drAli = new Doctor('Ali', 35, 6000);

drAhmed.sayHi(); // Hi, I am Ahmed
drAli.sayHi(); // Hi, I am Ali
```

This is the same as the previous example, but with a different syntax.

Important things to notice:

- The constructor function is called `constructor` in the class.
- We don't use the `function` keyword before the functions.
- If we want to use arrow functions, we can use for example `sayHi = () => { ... }`.
- When creating an object from a class, we use the `new` keyword.
- The first thing to get executed when creating an object from a class is the `constructor`.

3 Main OOP Concepts

3.1 Inheritance

Inheritance is a mechanism that allows you to eliminate redundant code by reusing existing classes. It allows you to create a new class that is based on an existing class.

For example, we can create a class for `Employee` and then create a class for `Doctor` that inherits from `Employee`.

```
class Employee {
  constructor(name, age) {
    this.name = name;
```

```javascript
    this.age = age;
  }

  sayHi() {
    console.log('Hi, I am ' + this.name);
  }
}

class Doctor extends Employee {
  constructor(name, a   ge, salary) {
    super(name, age);
    this.salary = salary;
  }

  getSalary() {
    console.log('My salary is ' + this.salary);
  }
}

let drAhmed = new Doctor('Ahmed', 30, 5000);
drAhmed.sayHi(); // Hi, I am Ahmed
drAhmed.getSalary(); // My salary is 5000
```

In this example, `Doctor` inherits from `Employee`. The `Doctor` class has access to the `sayHi` function from the `Employee` class.

Notice that `Doctor` has its own constructor function, but it calls the `Employee` constructor function using `super`.

If you log the `drAhmed` object, you will see that it has all the properties of `Employee` and `Doctor` directly in the object. You will also notice that the prototype of `Doctor` has both functions `sayHi` and `getSalary`.

Multiple Inheritance

You can't inherit from multiple classes in JS, however you can make a class inherit from another class that inherits from another class.

---

for example if you have `Employee` and `Doctor` and you want to create a class `Surgeon` that inherits from both, you can make `Doctor` inherit from `Employee` and then make `Surgeon` inherit from `Doctor`.

```javascript
class Surgeon extends Doctor, Employee { // Syntax Error
```

If we have the same variable or function in both the parent and child class, the variable or function in the child class will override the variable or function in the parent class (Polymorphism).

3.2 Polymorphism

Polymorphism is a feature that allows you to use a single interface to represent different data types.

For example, we can have a `sayHi` function in the `Employee` class and a `sayHi` function in the `Doctor` class. When we call `sayHi` on a `Doctor` object, it will call the `sayHi` function in the

`Doctor` class, not the one in the `Employee` class.

```
class Employee {
  sayHi() {
    console.log('Hi, I am an employee');
  }
}

class Doctor extends Employee {
  sayHi() {
    console.log('Hi, I am a doctor');
  }
}

let drAhmed = new Doctor();
drAhmed.sayHi(); // Hi, I am a doctor
```

Polymorphism has two types:

- **Overloading**: Same function name with different parameters.
- **Overriding**: Same function name with the same parameters. (In JS we only have overriding).

3.2.1 Access Modifiers

Access modifiers are keywords that set the accessibility of properties and methods in a class.

- **Public**: Accessible from anywhere. (Default).
- **Private**: Not accessible from outside the class. (use `#` before the property or method name ex: `this.#name`).
- **Protected**: Accessible within the class and its subclasses. (Not available in JS).

```
class Employee {
  #name; // Private property

  constructor(name, age) {
    this.#name = name;
    this.age = age;
  }

  #sayHi() { // Private method
    console.log('Hi, I am ' + this.#name);
  }

  sayHi() {
    this.#sayHi();
  }
}

let emp = new Employee('Ahmed', 30);
console.log(emp.age); // 30
```

```
20 // console.log(emp.#name); // Syntax Error
21 emp.sayHi(); // Hi, I am Ahmed
```

Notice that we can't access the private property `#name` from outside the class.

4 Modules

Modules are a way to split your code into multiple files. Each file is a module that *exports* some functions or variables that can be *imported* in other files.

To use modules in JS, we use the `export` and `import` keywords, we also use the `type="module"` attribute in the script tag.

To be able to use modules in the browser, you need to use a live server.

```
1 // variables.js
2 export let name = 'Ahmed';
3 export let age = 30;
4
5 // script.js
6 import { name, age } from './variables.js';
7 console.log(name); // Ahmed
8 console.log(age); // 30
```

In `index.html`:

```
1 <script type="module" src="script.js"></script>
```

The same can be done with functions and classes, just add `export` before the function or class, and `import` it in the other file.

When exporting multiple things, you can use `export { name, age }` in the end of the file, and when importing you can use `import * as vars from './variables.js'` to import all the exported things in an object called `vars`.

```
1 // info.js
2 let name = 'Ahmed';
3 let age = 30;
4 let sayHi = function() {
5   console.log('Hi, I am ' + name);
6 };
7
8 export { name, age, sayHi };
9
10 // script.js
11 import * as info from './info.js';
12 console.log(info.name); // Ahmed
13 console.log(info.age); // 30
14 info.sayHi(); // Hi, I am Ahmed
```

We can use `export default` to export something as the default export, and when importing we can import it without the curly braces and with any name we want.

```
1 // info.js
2 let name = 'Ahmed';
```

```
let age = 30;
let sayHi = function() {
  console.log('Hi, I am ' + name);
};

export { name, age, sayHi };
export default sayHi;

// script.js
import greet from './info.js'; // We can use any name to import the
  ↪ default export
greet(); // Hi, I am Ahmed
```

To import the other exports along with the default export, you can use `import greet, { name, age } from './info.js';`. Notice that `name`, `age` are in curly braces and have the same name as the exported variables.

5 Summary

- Higher order functions are functions that take other functions as arguments or return functions.
- In JS, objects inherit properties and methods from other objects using prototypal inheritance.
- OOP is a programming paradigm based on objects. It simulates the real world and makes your code more organized.
- There are two main ways to implement OOP: class based OOP and prototype based OOP.
- In class based OOP, we use classes to define objects. A class is a blueprint for creating objects.
- In prototype based OOP, we use prototypes to define objects. A prototype is a template object that an object inherits properties and methods from.
- Inheritance is a mechanism that allows you to eliminate redundant code by reusing existing classes.
- Polymorphism is a feature that allows you to use a single interface to represent different data types.
- Access modifiers are keywords that set the accessibility of properties and methods in a class.
- Modules are a way to split your code into multiple files. Each file is a module that exports some functions or variables that can be imported in other files.
- To use modules in JS, use the `export` and `import` keywords, and use the `type="module"` attribute in the script tag.
- You can export multiple things using `export { name, age }` and import them using `import * as vars from './variables.js'`.
- You can export something as the default export using `export default` and import it without the curly braces and with any name you want.