# JS.4

- ▼ What is a programming paradigm?

  Programming paradigm is an approach to solve problem using some programming language

- ▼ Is JavaScript multi-paradigm?

  yes

- ▼ What is OOP (Object Oriented Programming)?

  - Objects
  - Classes
  - Prototypes

  having objects as a programmer-provided data structure that can be manipulated with functions

- ▼ What is prototypal inheritance?

  it means that vanilla JavaScript has objects without classes

  In JavaScript, it creates a link when we create an object

  inheritance is flowing up the prototype chain of inheritance

- ▼ What is functional programming?

  - Closures
  - First class functions
  - Lambdas

  Functional programming basically means writing code that does something (declares what is done) but isn't specific about how to do it (imperative).

▼ Imperative vs. Declarative?

Declarative: You declare what you want to happen, not how it’s done.

Imperative: You declare how something is done. Procedural.

▼ Is functional programming declarative?

yes

▼ Is object oriented programming imperative?

yes

▼ What is imperative also called?

procedural

▼ What are examples of the imperative programming paradigm?

Object-oriented programming (OOP), procedural programming, and parallel processing

▼ What are examples of the declarative programming paradigm?

Functional programming, logic programming, and database processing

▼ What are some characteristics of declarative (functional) programming?

You declare what you want to happen, not how it’s done.

the data is often considered immutable values (not changeable)

▼ What are some characteristics of imperative programming?

- You specify exactly how to do something, not just the desired outcome.
- Variables, pointers, and stored procedures are commonplace, and data is often considered mutable variables (changeable)
- Inheritance is commonplace

▼ What are first class functions?

Functions are first-class objects in JavaScript, meaning they can be:

- stored in an array, variable, or object

- passed as one of the arguments to another function
- returned from another function

▼ What are lambda expressions (pre-ES6)?

In JavaScript, you can use any function as a lambda expression because functions are first class.

▼ What paradigm is used for React Hooks?

React Hooks are a functional programming approach for abstracting away complexity from managing the state (i.e. mutable variables) of an app.

▼ Rule of thumb for inheritance

use no more than 3 levels of inheritance

▼ What is BigInt (ES11)?

With BigInt we can go beyond this number in JS: 9007199254740991

```
let bigInt = BigInt(9007199254740992n) // this number and above are now represented
```

▼ What are Dynamic Imports (ES11)?

We can import modules dynamically through variables.

With that, the variables that receive the modules are able to encompass the namespaces of these modules in a global way

```
let Dmodule;

if ("module 1") {
  Dmodule = await import('./module1.js')
} else {
  Dmodule = await import('./module2.js')
}

/* It is possible to use Dmodule. (Methods)
throughout the file globally */
Dmodule.useMyModuleMethod()
```

▼ How can you export modules (ES11)?

```
export * as MyComponent from './Component.js'
```

▼ What is optional chaining (ES11)?

This functionality removes the need for conditionals before calling a variable or method enclosed in it.

```
// Without optional chaining
const number = user.address && user.address.number

// With optional chaining
const number = user.address?.number
```

▼ What is Nullish Coalescing Operator (ES11)?

The operator only allows undefined and null to be falsey values.

```
undefined ?? 'value'
// 'value'

null ?? 'value'
// 'value'

false ?? 'value'
// false

0 ?? 'value'
// 0

NaN ?? 'value'
// NaN
```

▼ What is Promise.AllSettled (ES11)?

The Promise.AllSettled attribute allows you to perform a conditional that observes whether all promises in an array have been resolved.

```
const myArrayOfPromises = [
    Promise.resolve(myPromise),
    Promise.reject(0),
    Promise.resolve(anotherPromise)
]

Promise.AllSettled(myArrayOfPromises).then ((result) => {
   // Do your stuff
})
```

▼ What is matchAll (ES11)?

The matchAll method is a feature that better details regex comparisons within a string.

Its result is an array that indicates the positions, as well as the string group and the source of the search.

```
const regex = /[0-5]/g
const year = '2059'
const match = year.matchAll(regex)
for (const match of year.matchAll(regex)) {
  console.log(match);
}

// example output
// ["2", index: 0, input: "2059", groups: undefined]
```

▼ What are design patterns?

they structure the code in an optimized manner to meet the problems we are seeking solutions to

▼ What is the constructor design pattern?

used to initialize the newly created objects

```
const object = new ConstructorObject();
```

▼ What is the prototype pattern?

it is based on prototypical inheritance whereby objects created to act as prototypes for other objects

```
const myCar= {
name:"Ford",
brake:function(){
console.log("Stop! I am applying brakes");
}
Panic : function (){
console.log ( "wait. how do you stop thuis thing?")
}
}
// use object create to instantiate a new car
const yourCar= object.create(myCar);
//You can now see that one is a prototype of the other
console.log (yourCar.name);]
```

▼ What is the module design pattern?

The different types of modifiers (both private and public) are set in the module pattern.

```
function AnimalContainer() {

const container = [];

function addAnimal (name) {
container.push(name);
}

function getAllAnimals() {
return container;
}

function removeAnimal(name) {
const index = container.indexOf(name);
if(index < 1) {
throw new Error('Animal not found in container');
}
container.splice(index, 1)
}

return {
add: addAnimal,
get: getAllAnimals,
remove: removeAnimal
}
}

const container = AnimalContainer();
container.add('Hen');
container.add('Goat');
container.add('Sheep');

console.log(container.get()) //Array(3) ["Hen", "Goat", "Sheep"]
container.remove('Sheep')
console.log(container.get()); //Array(2) ["Hen", "Goat"]
```

▼ What is singleton pattern or strict pattern?

It is essential in a scenario where only one instance needs to be created, for example, a database connection.

It is only possible to create an instance when the connection is closed or you make sure to close the open instance before opening a new one.

```
function DatabaseConnection () {

let databaseInstance = null;

// tracks the number of instances created at a certain time
let count = 0;

function init() {
console.log(`Opening database #${count + 1}`);
//now perform operation
}
function createIntance() {
if(databaseInstance == null) {
databaseInstance = init();
}
return databaseInstance;
}
function closeIntance() {
console.log('closing database');
databaseInstance = null;
}
return {
open: createIntance,
close: closeIntance
}
}

const database = DatabseConnection();
database.open(); //Open database #1
database.open(); //Open database #1
database.open(); //Open database #1
database.close(); //close database
```

▼ What is the factory pattern?

It is a creational concerned with the creation of objects without the need for a constructor.

Therefore, we only specify the object and the factory instantiates and returns it for us to use.

```
// Dealer A

DealerA = {};

DealerA.title = function title() {
return "Dealer A";
};

DealerA.pay = function pay(amount) {
console.log(
`set up configuration using username: ${this.username} and password: ${
this.password
}`
);
return `Payment for service ${amount} is successful using ${this.title()}`;
};

//Dealer B

DealerB = {};
DealerB.title = function title() {
return "Dealer B";
};

DealerB.pay = function pay(amount) {
console.log(
`set up configuration using username: ${this.username}
and password: ${this.password}`
);
return `Payment for service ${amount} is successful using ${this.title()}`;
};

//@param {*} DealerOption
//@param {*} config

function DealerFactory(DealerOption, config = {}) {
const dealer = Object.create(dealerOption);
Object.assign(dealer, config);
return dealer;
}

const dealerFactory = DealerFactory(DealerA, {
username: "user",
password: "pass"
});
console.log(dealerFactory.title());
console.log(dealerFactory.pay(12));

const dealerFactory2 = DealerFactory(DealerB, {
username: "user2",
password: "pass2"
});
console.log(dealerFactory2.title());
console.log(dealerFactory2.pay(50));
```

▼ What is the observer design pattern?

The observer design pattern is handy in a place where objects communicate with other sets of objects simultaneously.

the modules involved modify the current state of data

```
function Observer() {
this.observerContainer = [];
}

Observer.prototype.subscribe = function (element) {
this.observerContainer.push(element);
}

// the following removes an element from the container

Observer.prototype.unsubscribe = function (element) {

const elementIndex = this.observerContainer.indexOf(element);
if (elementIndex &gt; -1) {
this.observerContainer.splice(elementIndex, 1);
}
}

/**
* we notify elements added to the container by calling
* each subscribed components added to our container
*/
Observer.prototype.notifyAll = function (element) {
this.observerContainer.forEach(function (observerElement) {
observerElement(element);
});
}
```

▼ What is the command pattern?

it encapsulates method invocation, operations, or requests into a single object so that we can pass method calls at our discretion

```
(function(){

var carManager = {

//information requested
requestInfo: function( model, id ){
return "The information for " + model + " with ID " + id + " is foo bar";
},

// now purchase the car
buyVehicle: function( model, id ){
return "You have successfully purchased Item " + id + ", a " + model;
},

// now arrange a viewing
arrangeViewing: function( model, id ){
return "You have successfully booked a viewing of " + model + " ( " + id + " ) ";
}
};
})();
```