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 + " ) ";
}
};
})();
```

▼ What is transpiling?

the process of translating one language or version of a language to another

e.g. transpile ES6 to ES5 to get a better browser support

▼ What is the CommonJS (CJS) module format?

relies on importing and exporting modules with keywords require and exports

CJS is synchronous

CJS isn't natively understood by browsers; it requires either a loader library or some transpiling

```
// utils.js
  // we create a function
  function add(r){
    return r + r;
  }
  // export (expose) add to other modules
  exports.add = add;


  // index.js
  var utils = require('./utils.js');
  utils.add(4); // = 8
```

▼ What is the Asynchronous Module Definition (AMD) module format?

like CommonJS but it supports asynchronous module loading

```
// add.js
  define(function() {
    return add = function(r) {
      return r + r;
    }
  });


  // index.js
  define(function(require) {
    require('./add');
    add(4); // = 8
  }
```

▼ What is the Universal Module Definition (UMD) module format?

It is based on AMD but with some special cases included to handle CommonJS compatibility.

▼ What is the ES2015 Modules (ESM) module format (you use)?

This format is really simple to read and write and supports both synchronous and asynchronous modes of operation.

```
// add.js
  export function add(r) {
    return r + r;
  }


  // index.js
  import add from "./add";
  add(4); // = 8
```

▼ What is tree-shaking?

process that removes unused code from bundles

▼ Who created JavaScript?

Brendan Eich

▼ When was JavaScript created?

1995

▼ What is the Gorilla / Banana problem?

The problem with object-oriented languages is they've got all this implicit environment that they carry around with them.

You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

▼ What is the Fragile Base Class Problem?

Making small changes to a base class creates rippling side-effects that break things that should be completely unrelated.

▼ What is the Duplication by Necessity Problem?

creating new classes with subtle differences by changing up what inherits from what — but it's too tightly coupled to properly extract and refactor

When you find a bug, you don't fix it in one place. You fix it everywhere.

Avoid classical inheritance

▼ What is the DRY principle?

Don’t Repeat Yourself

▼ Should you use ES6 classes or classical inheritance?

No. See the Duplication by Necessity Problem

▼ How should you share behaviors?

functions and module imports

▼ How is {...variable} a form of prototypal inheritance?

You can copy/extend object properties using object spread syntax: {...a, ...b}.

Sources of clone properties are a specific kind of prototype called exemplar prototypes.

▼ What is cloning an exemplar prototype an example of?

concatenative inheritance

▼ Favor object composition over what?

class inheritance

▼ Like objects, closures are what?

a mechanism for containing state

▼ A closure is created when?

a function accesses a variable defined outside the immediate function scope

▼ What is prototypal OO (object-oriented)?

building new instances not from classes, but from a compilation of smaller object prototypes

▼ Why is functional programming is a natural fit for JavaScript?

JavaScript makes it easy to assign functions to variables, pass them into other functions, return functions from other functions, compose functions, and so on.

JavaScript offers immutable values for primitive types,

JavaScript offers features that make it easy to return new objects and arrays rather than manipulate properties of those that are passed in as arguments.

▼ What is idempotence?

Given the same inputs, a pure function will always return the same output, regardless of the number of times the function is called.

▼ Pure functions can be safely applied with no side-effects, meaning what?

they do not mutate any shared state or mutable arguments, and other than their return value, they don’t produce any observable output

▼ What is reactive programming?

Reactive programming uses functional utilities like map, filter, and reduce to create and process data flows which propogate changes through the system: hence, reactive.

When input x changes, output y updates automatically in response.

▼ When using reactive programming, you instead specify what?

data dependencies in a more declarative fashion

▼ What is a stream?

A stream is a list expressed over time.

▼ In Rx (reactive extensions) you create what?

you create observable streams and then process those streams using a set of functional utilities like the ones described above

▼ What is a promise in the context of a stream?

A promise is basically a stream that only emits a single value (or rejection).

▼ What are generators (ES6)?

A generator is a function that can stop midway and then continue from where it stopped.

In short, a generator appears to be a function but it behaves like an iterator.

```
function * generatorFunction() { // Line 1
  console.log('This will be executed first.');
  yield 'Hello, ';   // Line 2
  console.log('I will be printed after the pause');
  yield 'World!';
}
const generatorObject = generatorFunction(); // Line 3
console.log(generatorObject.next().value); // Line 4
console.log(generatorObject.next().value); // Line 5
console.log(generatorObject.next().value); // Line 6
// This will be executed first.
// Hello,
// I will be printed after the pause
// World!
// undefined
```

▼ What is an iterator?

In JavaScript an iterator is an object which defines a sequence and potentially a return value upon its termination.

```
function makeRangeIterator(start = 0, end = Infinity, step = 1) {
    let nextIndex = start;
    let iterationCount = 0;

    const rangeIterator = {
       next: function() {
           let result;
           if (nextIndex < end) {
               result = { value: nextIndex, done: false }
               nextIndex += step;
               iterationCount++;
               return result;
           }
           return { value: iterationCount, done: true }
       }
    };
    return rangeIterator;
}

const it = makeRangeIterator(1, 10, 2);

let result = it.next();
while (!result.done) {
 console.log(result.value); // 1 3 5 7 9
```

```
 result = it.next();
}

console.log("Iterated over sequence of size: ", result.value); // [5 numbers returned, that took interval in between: 0 to 10]
```

▼ Where are closures used in JavaScript?

for object data privacy, in event handlers and callback functions, and in partial applications, currying, and other functional programming patterns

▼ What is a closure?

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment)

▼ What is the lexical environment equivalent to?

surrounding state

▼ A closure gives you access to what?

an outer function’s scope from an inner function

▼ Closures are created when?

every time a function is created, at function creation time

▼ To use a closure, define what?

a function inside another function and expose it

▼ To expose a function you have to do what?

return it or pass it to another function

▼ How can closures enable data privacy?

When you use closures for data privacy, the enclosed variables are only in scope within the containing (outer) function.

You can’t get at the data from an outside scope except through the object’s privileged methods.

▼ What is Application?

The process of applying a function to its arguments in order to produce a return value.

▼ What is Partial Application?

The process of applying a function to some of its arguments.

The partially applied function gets returned for later use.

▼ Partial application fixes (partially applies the function to) what?

one or more arguments inside the returned function, and the returned function **takes the remaining parameters as arguments in order to complete the function application**.

```
partialApply(targetFunction: Function, ...fixedArgs: Any[]) =>
  functionWithFewerParams(...remainingArgs: Any[])
```

▼ Partial application takes advantage of what in order to fix parameters?

closure scope

▼ Partial Application example

You want a function that adds 10 to any number.

We'll call it `add10()`.

The result of `add10(5)` should be `15`.

key: takes the remaining parameters as arguments in order to complete the function application

In this example, the argument, `10` becomes a fixed parameter remembered inside the `add10()` closure scope.

```
const add = (a, b) => a + b;

const add10 = partialApply(add, 10);
add10(5);
```

▼ What is a Promise?

A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it's not resolved.

▼ A promise may be in one of 3 possible states

fulfilled, rejected, or pending

▼ Promise users can attach what to handle the fulfilled value or the reason for rejection?

callbacks

▼ A promise is an object which can be returned synchronously from what?

an asynchronous function

▼ A promise is settled if what?

it's not pending (it has been resolved or rejected)

▼ Every promise must supply a .then() method with the following signature:

```
promise.then(
  onFulfilled?: Function,
  onRejected?: Function
) => Promise
```

▼ Error handling for a promise?

```
save().then(
  handleSuccess,
  handleError
);
```

▼ What is Promise.reject()?

it returns a rejected promise

▼ What is Promise.resolve()?

it returns a resolved promise

▼ What is Promise.race()?

it takes an array (or any iterable) and returns a promise that resolves with the value of the **first** resolved promise in the iterable,

or rejects with the reason of the first promise that rejects

▼ What is Promise.all()?

it takes an array (or any iterable) and returns a promise that resolves when **all** of the promises in the iterable argument have resolved,

or rejects with the reason of the first passed promise that rejects.

▼ What is a function?

A function is a process which takes some input, called arguments, and produces some output called a return value.

▼ What is mapping?

A function maps input values to output values.

▼ What is a procedure?

a sequence of steps

▼ What is a pure function (think math)?

- Given the same input, will always return the same output.
- Produces no side effects which means that it can't alter any external state.
- Is completely independent of outside state

▼ An example of an impure function?

a function where it makes sense it call it without a return value

▼ Pure functions are the foundation of what?

functional programming

▼ What does JS is a single threaded language mean?

This means it has one call stack and one memory heap.

As expected, it executes code in order and must finish executing a piece code before moving onto the next.

▼ What is concurrency?

the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or at the same time simultaneously

▼ What are examples of concurrency in JS?

API I/O, event listeners, web workers, iframes, and timeouts

▼ What is a a deterministic algorithm?

an algorithm that, given a particular input, will always produce the same output aka pure function

▼ Functions that rely on information other than the arguments you pass in to produce results are what?

impure functions

▼ Functions that take objects as arguments are references. Pure functions must not

mutate those references. Make a copy of those references

```
const copy = {...someObject}; // shallow copy. doesn't copy nested objects
```

▼ JavaScript switch statement

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

▼ JavaScript parseInt()

```
parseInt(string, radix) // radix aka base. it does not default to 10.

parseInt("2.5") // this will be converted to a Number not a float if no radix.
parseInt("2", 10)
```

▼ With object oriented programming, application state is what?

shared and colocated with methods in objects

▼ Functional programming (often abbreviated FP) is what?

the process of building software by composing pure functions, avoiding shared state, mutable data, and side-effects

▼ Functional programming is not imperative but what?

declarative

▼ How does application state flow with functional programming?

through pure functions

▼ Functional programming is a programming what?

paradigm

▼ What is a programming paradigm?

it is a way of thinking about software construction based on some fundamental, defining principles

▼ What are pros of functional code?

it tends to be more concise, more predictable, and easier to test

▼ A pure function is a function which what?

- Given the same inputs, always returns the same output, and
- Has no side-effects

▼ Pure functions have referential transparency. What is it?

you can replace a function call with its resulting value without changing the meaning of the program

▼ What is function composition?

the process of combining two or more functions in order to produce a new function or perform some computation.

For example, the composition f . g (the dot means “composed with”) is equivalent to f(g(x)) in JavaScript.

▼ What is shared state?

any variable, object, or memory space that exists in a shared scope, or as the property of an object being passed between scopes

▼ What is a shared scope?

it can include global scope or closure scopes

▼ How does functional programming avoid shared state?

it relys on immutable data structures and pure calculations to derive new data from existing data

▼ What is the problem with shared state?

in order to understand the effects of a function, you have to know the entire history of every shared variable that the function uses or affects

▼ What is a race condition?

When a program's behavior is dependent on the sequence or timing of other uncontrollable events it can lead to a bug when one or more of the possible behaviors is undesirable

▼ When you avoid shared state, what do you not have to worry about?

the timing and order of function calls don’t change the result of calling the function

▼ What is an immutable object?

an object that can’t be modified after it’s created

▼ const does not create immutable objects but instead it...

creates a variable name binding which can’t be reassigned after creation

▼ What are tries, pronounced "trees", and why does Immutable.js use them?

trie data structures (pronounced “tree”) which are effectively deep frozen — meaning that no property can change, regardless of the level of the property in the object hierarchy

▼ What is a side effect?

any application state change that is observable outside the called function other than its return value

▼ What do side effects include? (examples)

- Modifying any external variable or object property (e.g., a global variable, or a variable in the parent function scope chain)
- Logging to the console
- Writing to the screen
- Writing to a file

- Writing to the network
- Triggering any external process
- Calling any other functions with side-effects

▼ Side-effect actions need to be isolated from what?

the rest of your program logic

▼ What is a higher-order function?

any function which takes a function as an argument, returns a function, or both

▼ What is a functor?

A functor is something that can be mapped over.

In other words, it's a container which has an interface which can be used to apply a function to the values inside it.

▼ A list expressed over time is what?

a stream

▼ Imperative?

the flow control: How to do things

▼ Declarative?

the data flow: What to do

▼ Imperative code uses what?

statements.

A statement is a piece of code which performs some action.

▼ Declarative code uses what?

expressions.

An expression is a piece of code which evaluates to some value. Expressions are usually some combination of function calls, values, and operators which are evaluated to produce the resulting value.

▼ What is function composition?

the process of combining two or more functions to produce a new function

▼ Composing functions together is like what?

snapping together a series of pipes for our data to flow through

▼ A composition of functions is evaluated how?

from right to left

given f(g(x))

x evaluated then g then f

▼ How can we compose functions in JavaScript (example)? - right to left evaluation

```
// since a composition of functions is evaluated right to left, we can use
// a reducer that reduces to a single value each time, starting on the right

const compose = (...fns) => x => fns.reduceRight((v, f) => f(v), x);

// example
const toSlug = compose(
  encodeURIComponent,
  join('-'),
  map(toLowerCase),
  split(' ')
);

console.log(toSlug('JS Cheerleader')); // 'js-cheerleader'
```

▼ How can we compose functions in JavaScript (example)? - left to right evaluation - easier

```
// a reducer that reduces to a single value each time, starting on the left

const pipe = (...fns) => x => fns.reduce((v, f) => f(v), x);

// example
const toSlug = pipe(
  split(' '),
  map(toLowerCase),
  join('-'),
  encodeURIComponent
);

console.log(toSlug('JS Cheerleader')); // 'js-cheerleader'
```

▼ What is points-free style? (example)

The `pipe()` implementation above is written in a points-free style, which means that it does not identify the arguments on which it operates at all.

```
const pipe = (...fns) => x => fns.reduce((v, f) => f(v), x);
```

▼ JavaScript's object system is based on classes not what?

prototypes

▼ What is a class?

a description of an object to be created

▼ Classes create subclass relationships which create what?

hierarchical class taxonomies

▼ JavaScript's class inheritance uses the prototype chain to what?

wire the child `Constructor.prototype` to the parent `Constructor.prototype` , the tightest coupling available in OO design.

▼ What is Prototypal Inheritance?

A prototype is a working object instance.

▼ Objects inherit directly from what?

other objects

▼ What is the fragile base class problem?

The fragile base class problem is a fundamental architectural problem of object-oriented programming systems where base classes are considered "fragile" because seemingly safe modifications to a base class, when inherited by the derived classes, may cause the derived classes to malfunction.

▼ What is the inflexible hierarchy problem?

eventually, all hierarchies are wrong for new uses

▼ What is the duplication by necessity problem?

due to inflexible hierarchies, new use cases are often shoe-horned in by duplicating, rather than adapting existing code

▼ What is the Gorilla/banana problem?

What you wanted was a banana, but what you got was a gorilla holding the banana, and the entire jungle

▼ Favor what over class inheritance?

object composition

▼ What are three different kinds of prototypal OO?

Concatenative inheritance

Prototype delegation

Functional inheritance

▼ What is Concatenative inheritance?

The process of inheriting features directly from one object to another by copying the source objects properties.

e.g. Object.assign()

▼ What is Prototype delegation?

an object may have a link to a prototype for delegation

If a property is not found on the object, the lookup is delegated to the delegate prototype, which may have a link to its own delegate prototype, and so on up the chain until you arrive at `Object.prototype`, which is the root delegate.

▼ What is Functional inheritance?

works by producing an object from a factory, and extending the produced object by assigning properties to it directly (using concatenative inheritance)

▼ What is a factory function?

When a function is not a constructor (or `class`)

▼ With object composition you have what instead of classes?

features

```
const C = compose(feat1, feat3);
const D = compose(feat1, feat2, feat4);
```

▼