# Javascript Design patterns

What is design pattern ?

The academic definition is a general, reusable solutionto a commonly-occurring problem within a given context in software design. In simple words, it's a way that has been defined as a proper approach to resolve common problems in code

functions as first-class citizens, what does that mean?

It means that when functions can be treated like a variable, meaning they can be passed as arguments to other functions as well or can be assignedas a value to a variable or even return in a function, it means the function is a first-class citizen and JavaScript does first-class citizen functions.

```
console.log("hello");
var calc = ()=>{
   return 4*3;
}
var aNumber = calc();
console.log("aNumber",aNumber);//here we are sending functiona variable to console, this is called
first class citizens
```

What is a callback and its role ?

In its simplest terms, a callback function is a function that is called inside of another function. In other words, whenever you pass a function in the arguments and run it inside this function you're doing the callback pattern.

```
console.log("hello");
var calc = ()=>{
   return 4*3;
}
var printCalc = (callback)=>{
   console.log(callback())
}
printCalc(calc);
```

**Categories of design patterns:**

1. **Creational - > create new things**
2. **Structural -> structure your code**
3. **Behavioral -> Use for behavior of code**

**Creational Patterns:**

1. Prototype/Class design pattern:

The prototype, or class pattern, allows us to define a blueprint for a specific type of item and then reuse it by creating a new object from that class.

```
class car{
   constructor(doors,engine,color){
      this.doors = doors;
      this.engine = engine;
      this.color = color;
   }
}
var civic = new car(4, "v6","grey");
console.log(civic);
```

2. Constructor pattern:

The constructor pattern is one step further from the class pattern, where we leverage a class created to initiate a new extended class from it.

```
class car{
   constructor(doors,engine,color){
      this.doors = doors;
      this.engine = engine;
      this.color = color;
   }
}
class suv extends car{
   constructor(doors,engine,color){
      super(doors,engine,color);
      this.wheels =4;
   }
}
var civic = new car(4, "v6","grey");
var cx5 = new suv(4,"v8","red");
console.log(cx5);
console.log(civic);
```

3. Singleton pattern:

it's simply preventing our class from creating more than one instance of the blueprint we've defined. In other words, we use the same principle we've used since the beginning of this chapter except we allow only one instance of the class to be created.

```
let instance = null;
class car{
  constructor(doors,engine,color){
    if(!instance){
      this.doors = doors;
      this.engine = engine;
      this.color = color;
      instance = this;
    }
    else{
      return instance;
    }

  }
}
var civic = new car(4, "v6","grey");//only one instance allowed
var honda = new car(4,"v8","red");
console.log(honda);//car {doors: 4, engine: "v6", color: "grey"}
console.log(civic);//car {doors: 4, engine: "v6", color: "grey"}
```

4.Factory Pattern:

The factory pattern is great when you want to create say a mechanism to create other objects. This can be useful when you want a factory to handle most of your classes and then simply use this factory method to create new objects

```
class car {
  constructor(doors, engine, color) {
    this.doors = doors;
    this.engine = engine;
    this.color = color;

  }
}
class carFactory{
  createCar(type){
    switch(type){
      case "civic":
```

```
            return new car(4,"v8","grey");
        case "honda":
            return new car(2,"v4","red");
      }
    }
}
const factory = new carFactory();
const honda = factory.createCar("honda");
console.log(honda);
```

5. Abstract Factory:

If you were to take the concept of factories further, the next step would be the abstract factory pattern where you abstract the factories and are able to create multiple factories, classes, et cetera.

```
class car {
  constructor(doors, engine, color) {
    this.doors = doors;
    this.engine = engine;
    this.color = color;

  }
}
class carFactory{
  createCar(type){
    switch(type){
      case "civic":
        return new car(4,"v8","grey");
      case "honda":
        return new car(2,"v4","red");
    }
  }
}

class suv {
  constructor(doors, engine, color) {
    this.doors = doors;
    this.engine = engine;
    this.color = color;

  }
}
class suvFactory{
  createSuv(type){
```

```
    switch(type){
      case "xs5":
        return new car(4,"v8","grey");
      case "xs6":
        return new car(2,"v4","red");
    }
  }
}
const carfactory = new carFactory();
const suvfactory = new suvFactory();
//below is abstract method we can use for any object
const automanufacturer = (type,model)=>{
  switch(type){
    case "car":
      return carfactory.createCar(model);
    case "suv":
      return suvfactory.createSuv(model);
  }
}

const xs5 = automanufacturer("suv","xs5");
console.log(xs5);
```

**Structural Patterns:**

1. Module Pattern:

The module pattern is used everywhere in our code especially if you use any frameworks. Whenever you encapsulate a block of code into a singular function or pure function as it is sometimes referred to, you're creating a module. The idea behind using module is to organize your code in pure functions so if you have code to debug, it is much easier to find where the error is.

We often use modules too with the keyword, import or export, so when we compile our code.

2. Mixins Pattern:

Mixins are a great way to mix functions and instances of a class after they have been created.

```
class car {
    constructor(doors, engine, color) {
        this.doors = doors;
        this.engine = engine;
        this.color = color;
```

```
    }
}
class carFactory{
    createCar(type){
        switch(type){
            case "civic":
                return new car(4,"v8","grey");
            case "honda":
                return new car(2,"v4","red");
        }
    }
}
let carMixin = {
    revEngine(){
        console.log(`the ${this.engine} is vroom vroom`);//the v4 is vroom vroom
    }
}
const carfactory = new carFactory();
const automanufacturer = (type,model)=>{
    switch(type){
        case "car":
            return carfactory.createCar(model);
        case "suv":
            return suvfactory.createSuv(model);
    }
}
Object.assign(car.prototype,carMixin);

const honda = automanufacturer("car","honda");
honda.revEngine();
//So basically what we did is use mixins to add a new function with our car class
and then made sure that we created a new instance of that car which is a honda
and then we passed the engine of the honda inside of the function created with
the mixin and was able to print The v4 is doing Vroom Vroom.
```

3. Façade Pattern:

What is a facade? It is basically the pattern of hiding away complexity by creating a facade for the complex code. So if you're thinking, "What?" That's absolutely normal. If you're a react developer or building components, you have been using

facades every day. When you are building a component in any framework, you code the complexity of this component into a module or file and then leverage a simple line to render this component into your code. Well, this is what the facade pattern is.

4. Fly weight Pattern:

The Flyweight pattern is a method to minimize recreating the same items twice, and therefore minimize the memory impact in our systems. You have to understand that whenever we create new items with our applications, we stack these items into the memory of a browser. Your browsers use the Flyweight pattern to save images, for example, in memory, so they don't load twice. And guess what, the Flyweight pattern uses a similar approach to the Singleton.

5. Decorator Pattern:

The decorator pattern is very similar to mixins, where we decorate code with classes or code that came from another area.There is actual syntax on the most recent versions of JavaScript. And has been used for a while on TypeScript and in heavy use in Angular code. So, the purpose of a decorator pattern, like a mixin, is to take from example, a class and extend it with other code.

6. Model-view-controller (MVC) pattern:

This pattern basically defines how an application should be split and often reflects how your modules are organized within three simple categories, models, views and controllers. The model is where your data resides where you define your schemas and the models for your data. The views is where you have your views and in most case, the pure HTML of your application, where the visuals are, and finally, the controllers are where you have your logic of your application, the functions that makes your application run.

7. Model-view-presenter (MVP) Pattern:

the Model-View-Presenter pattern, which is loosely based on MVC and almost the same. Looking at both will help define the major differences in between the two, but they offer similar approaches but architectured differently. In an MVC, or Model-View-Controller, pattern, we have our application organized in models, views and controllers. Typically, the organization will have views pull data from controllers or models directly. And if there are any other logic or functions needed for the view, the controllers will supply them.

So in other words, the views have access to both models and controllers in an MVC model. Where MVP differs is the view doesn't have access to the model. It has to get it from the presenter. And the presenter serves as the logic and supplier of data. In this pattern, the view passes through the presenter to get the data through functions. And

the presenter pulls from the model. It is the major difference. The MVP pattern is seen in several frameworks, such as Backbone, but is quite popular in Android development.

8. Model-view-viewModel (MVVM):

The MVVM, or the Model-View View Model pattern, is similar than the other two we already explored and is different only in implementation again. It is also sometimes referred to as MVVC, or Model-View View Controller, but in both cases, it serves the same purpose. The first view is your view which doesn't have any data or logic. It is simply a dumb component, or component without any logic or data, which is the view. Then you have the second view model, or view controller in MVVC, which holds the logic and the state of the data.

And this view model connects to a model. So, if you'd like to see examples of this MVVM pattern, simply develop an architecture applications with react and angular, and you'll see this MVVM approach in action. For example, in react, your application is architectural in stateless components, which are views. Stateful components which hold data and logic, therefore the view model, and then, finally, the model, is where react typically connects to a back end to process data, where your models are defined.

**Behavioral Patterns:**

1. Observer Pattern:

The observer pattern is one where we maintain a list of objects, based on events, and is typically done with updating data based on events. It is implemented for example with the subscribe and publish methods in MeteorJS.