Basic JavaScript Design Patterns

John Au-Yeung

Basic JavaScript Design Patterns

John Au-Yeung

Basic JavaScript Design Patterns

Basic JavaScript Design Patterns

JavaScript lets us do a lot of things. It’s sometimes too forgiving in its syntax.

To organize our code, we should use some basic design patterns. In this book, we’ll look at some basic object creation patterns that we may use.

Design Patterns

We should follow some design patterns so that we can have a good organization of our code.

If our code if well-organized, then working with them won’t be a pain now or in the future.

The Big 4 Object-Oriented Programming Building Blocks

Almost all good design patterns are based on 4 building blocks. They should have the following for them to be good.

Abstraction

Abstraction is one of the building blocks of a good design pattern.

It's the careful consideration of how we're going to handle problems.

Abstractions aren't a programming technique. It's a way to conceptualize a problem before applying object-oriented techniques.

It's a way for us to divide our program into natural segments.

If we don't divide our program into small parts, we'll have problems with managing the complexity.

We just got divide programs into small, reusable pieces so that we can deal with them.

In JavaScript, we can divide programs in different ways.

We can divide them into functions. So we can write:

```
const speak = () => {
  //...
}
```

Then we can call it wherever it's available.

We can also divide programs into classes. Classes serve as a mechanism to create new objects.

For instance, we can write:

```
class Animal {
  //...
}
```

Then we can use the `new` operator to invoke it as follows:

```
const animal = new Animal();
```

We can then access its members. Assuming that it has `speak` method, we can write: `animal.speak();`

Encapsulation

Encapsulation is wrapping methods and data into an object.

We want to do that so different objects won’t know too much about each other.

This way, we won’t have issues with items being accessed when they shouldn’t be.

We should hide as much as possible so that coupling between objects is as loose as possible.

If they’re loosely coupled, then when we change one object, the chance of it breaking the other is lower.

We can divide what we expose to the outside.

The parts that change the most can be hidden and we expose an interface to the outside that doesn’t change that much.

Polymorphism

Polymorphism allows us to write code with different object types and we can decide on which object we want it to be at runtime.

JavaScript doesn't have a polymorphism in the sense that we can cast them items to whatever we want explicitly.

But since JavaScript objects are dynamic, we can add or remove properties on the fly.

So we can still morph our objects in a way that we want.

We can perform actions according to the type that we set the object to.

We can add and remove properties on the fly.

For instance, we can write:

```
animal.run = () => {
  //...
}
```

to define a new method on an object.

We can also merge different objects with `Object.assign` or the spread syntax.

For example, we can write:

```
const obj = Object.assign({}, foo, bar);
```

or:

```
const obj = {...foo, ...bar };
```

Both lines of code merge the `foo` and `bar` objects together.

Because of JavaScript’s dynamic nature, we don’t cast objects to different types directly for polymorphism.

Inheritance

Inheritance is where our objects inherit the properties and methods of another object.

There are a few ways to do this with JavaScript.

We can use the `extends` keyword for classes or constructors:

```
class Animal {
  //...
}

class Dog extends Animal {
  //...
}
```

The `extends` keyword indicates that we inherit the members of `Animal` in the `Dog` class.

When we create a `Dog` instance, we'll see the `Animal` members in the `__proto__` property of the `Dog` instance.

If we create object literals, we can use the `Object.create` method as follows: `const parent = {
  foo: 1
};

const child = Object.create(parent);`

Once we run the code above, `child` would have `foo` in the `__proto__` property.

`__proto__` is the immediate prototype of the current object.

A prototype is a template object which other objects can inherit members from.

We can access the properties of an object's prototype by accessing them directly.

So we can write:

```
child.foo
```

to access the `foo` property of `child` .

Closed for Modification, Open for Extension

In the same vein, once we wrote some code, we shouldn’t be modifying them too often.

Instead, they should be open to extensions so that we can add capabilities to them later.

Changing code always introduces risks. The more changes we make, the higher the chance that we break things.

Therefore, we should just extend things as much as possible so that the existing code stays untouched.

Singleton

The singleton pattern is where we create a single instance of a class.

Since we can create object literals without a constructor in JavaScript, we can create an object easily as follows: `const obj = { foo: 'bar' };`

`obj` will also be a standalone object, so defining an object literal follows the singleton pattern.

If we create a new object that has the same structure, we'll get a different object.

For instance, if we create a new object:

```
const obj2 = {
  foo: 'bar'
};
```

Then when we write:

```
obj2 === obj
```

to compare 2 objects, then that'll return `false` since they're different references.

Using new

We can also create a singleton object even if we create them from constructors or classes.

For instance, we can write:

```
let instance;
class Foo {
  constructor() {
    if (!instance) {
      instance = {
        foo: 'bar'
      };
    }
    return instance;
  }
}
```

Then if we create 2 `Foo` instances:

```
const foo1 = new Foo();
const foo2 = new Foo();
```

Then when we compare them:

```
console.log(foo1 === foo2);
```

We see `true` logged.

We check if the `instance` is created and then return it as is if it is. Otherwise, we set it to an object.

Since `instance` won't change after it's set, all instances are the same.

We can make the `instance` variable private by putting it in an IIFE.

For instance, we can write:

```
const Foo = (() => {
  let instance;
  return class {
    constructor() {
      if (!instance) {
        instance = {
          foo: 'bar'
        };
      }
      return instance;
    }
  }
})()
```

Now we can only access the returned instance rather than looking at the constructor.

An Instance in a Static Property

Alternatively, we can put the instance of the singleton in a static property of the constructor or class.

For instance, we can write:

```
class Foo {
  constructor() {
    if (!Foo.instance) {
      Foo.instance = {
        foo: 'bar'
      };
    }
    return Foo.instance;
  }
}
```

This way, we set the public `instance` property of the class `Foo` to return the instance if it exists. Otherwise, it'll create it first.

Now if we write:

```
const foo1 = new Foo();
const foo2 = new Foo();

console.log(foo1 === foo2);
```

We get the same result as before.

Instance in a Closure

We can also put the singleton instance in a closure.

For instance, we can write:

```
function Foo() {
  const instance = this;
  this.foo = 'bar';
  Foo = function() {
    return instance;
  }
}
```

We can write a constructor function that gets reassigned to itself at the end of the function.

This will let us return the `instance` while using `this` to assigning instant variables.

This won’t work with the class syntax since it’s created as a constant, so we can’t reassign it to a new value.

Factory

We can create a factory function to create an object.

A factory function is just a regular function that we can use to create objects.

It's useful for performing repeated operations when setting up similar objects.

It also offers a way for users of the factory to create objects without the specific class at compile time.

For instance, we can create our own factory function as follows: `class Animal {}`

```
class Dog extends Animal {}
class Cat extends Animal {}
class Bird extends Animal {}

const AnimalMaker = type => {
  if (type === 'dog') {
    return new Dog()
  } else if (type === 'cat') {
    return new Dog()
  } else if (type === 'bird') {
    return new Dog()
  }
}
```

The code above has the `AnimalMaker` factory function.

It takes the `type` parameter which allows us to create different subclasses given the `type` .

So we can call it as follows:

```
const animal = AnimalMaker('dog');
```

to create a `Dog` instance.

We can also add methods as we wish into the classes: `class`

```
Animal {
  walk() {}
}
class Dog extends Animal {
  bark() {}
}
class Cat extends Animal {}
class Bird extends Animal {}

const AnimalMaker = type => {
  if (type === 'dog') {
    return new Dog()
  } else if (type === 'cat') {
    return new Dog()
  } else if (type === 'bird') {
    return new Dog()
  }
}
```

This will also work.

Built-in Object Factory

JavaScript's standard library comes with several factory methods.

They include `Object` , `Boolean` , `Number` , `String` , and `Array` .

For instance, we can use the `Boolean` factory function as follows: `const bool = Boolean(1);`

Then `bool` is `true` since 1 is truthy.

All they do is return different objects or values according to what we pass in.

Many of them are useful. However, the `Object` constructor isn't all that useful since we can create objects with object literals.

Iterator

JavaScript has iterators since ES6.

Therefore, we don't have to write anything from scratch to implement this pattern We just have to use a generator function to return the values we're looking for sequentially.

Then we can call the built-in `next` function to get the value returned after the generator is returned from the generator function.

For instance, we can write:

```
const numGen = function*() {
  while (true) {
    yield (Math.random() * 100).toFixed(1);
  }
}
```

to create a generator function.

A generator function is denoted by the `function*` keyword. There's an asterisk after the `function` keyword.

The `yield` keyword will return the next value sequentially.

Then we can use it to return a random number as follows:

```
const nums = numGen();
console.log(nums.next().value);
console.log(nums.next().value);
console.log(nums.next().value);
```

We first call `numGen` to return an iterator.

Then we call `nums.next().value` to get the values we want.

We can also have a generator function that returns a finite number of values.

For instance, we can write:

```
const numGen = function*() {
  const arr = [1, 2, 3];
  for (const a of arr) {
    yield a;
  }
}

const nums = numGen();
let value = nums.next().value;
while (value) {
  console.log(value);
  value = nums.next().value;
}
```

In the code above, we changed the `numGen` function to return an array's entries in sequence.

This way, we can call the `next` method in a loop to get the entries in sequence with the `next` method.

In addition to `value` property, the object returned by the `next` method also has the `done` property.

So we can write:

```
const nums = numGen();
let {
  done,
  value
} = nums.next();

while (!done) {
  console.log(value);
  const next = nums.next();
  done = next.done;
  value = next.value;
}
```

Once all the values are returned, `done` will be set to `true`.

Additional Functionality

If we want more functionality, then we've to implement our own iterator.

For instance, if we want rewind functionality, we can create our own iterator as follows: `const iterator = (() => {`

```
  const arr = [1, 2, 3];
  let index = 0;
  return {
    next() {
      index++;
      return arr[index];
    },
    rewind() {
      index--;
      return arr[index];
    }
  }
})();
```

We just added an array with the `next` method to move to the next `arr` entry and `rewind` to move to the previous `arr` entry.

Then we can use `iterator` as follows:

```
console.log(iterator.next());
console.log(iterator.rewind());
```

And we see:

```
2
1
```

from the console log output. # Decorator

Decorators let us add functionality to an object on the fly.

JavaScript has decorators now so that we can use them to create our own decorator and mutate objects on the fly.

For instance, we can create one as follows:

```
const foo = () => {
  return (target, name, descriptor) => {
    descriptor.value = "foo";
    return descriptor;
  };
};

class Foo {
  @foo()
  bar() {
    return "baz";
  }
}

const f = new Foo();
console.log(f.bar);
```

A decorator in JavaScript, it's just a function that returns a function with the parameters `target` , `name` and `descriptor` .

`descriptor` is the property descriptor, so we can set the `value` , `writable` , and `enumerable` properties for it.

Our `foo` decorator transformed the `value` property of it to the string `'foo'` .

So we get that `Foo` instance's `bar` property becomes the string `'foo'` instead of a method.

JavaScript decorators are still experimental syntax, so we need Babel to use it.

Facade

The facade is a simple pattern that lets us provide an alternative interface to an object.

We can use it to hide the complex implementation behind a facade.

This is good because we don’t want users of our code to know everything.

The facade is very simple and we don’t have to do much.

We can also use it to combine multiple operations that are called together into one.

For instance, if we have multiple methods that are called together, then they can be put together into one method.

We can implement the facade pattern as follows:

```
const adder = {
  add() {
    //...
  },
}

const subtractor = {
  subtract() {
    //...
  },
}
const facade = {
  calc() {
    adder.add();
    subtractor.subtract();
  }
}
```

We created a `facade` object that combines the `add` and `subtract` methods.

The facade pattern is used to create an interface in front of multiple objects.

Also, we can use it to handle the intricacies of browsers.

Some browsers may not have the `preventDefault` or `stopPropgation` methods, so we may want to check that they exist before calling them: `const facade = {`

```
  stop(e) {
    if (typeof e.preventDefault === "function") {
      e.preventDefault();
    }
    if (typeof e.stopPropagation === "function") {
      e.stopPropagation();
    }
    //...
  }
}
```

Now we call always call `stop` and don't have to worry about their existence since we check that they exist before calling them.

Proxy

The proxy pattern is where one object acts as an interface to another object.

The difference between proxy and facade patterns are that we can proxies create interfaces for one object.

But facades can be interfaces for anything.

We can create a proxy as follows:

```
const obj = {
  foo() {
    //...
  },

  bar() {
    //...
  },
}

const proxy = {
  call() {
    obj.foo();
    obj.bar();
  }
}
```

We use the `proxy` to call both `foo` and `bar` in `obj` .

This pattern is good for lazy initialization.

This is where we only initialize something when it’s used.

Instead of always running the initialization code, we run the initialization code only when needed.

Proxy As a Cache

Proxies can act as a cache. We can check the cache for the results before returning the result.

If we have it, then we return the result from the cache.

Otherwise, we generate the result, put it in the cache, and then return it.

For instance, we can write: `const obj = {`

```
  add() {
    //...
  }
}

const proxy = {
  cache: {},
  add(result) {
    if (!cache[result]) {
      cache[result] = obj.add();
      //...
    }
    return cache[result];
  }
}
```

Now we can do expensive operations only when needed instead of always doing it.

Adapter

The adapter pattern is useful for creating a consistent interface for another object.

The object we're creating an interface for doesn't fit with how it's used.

Therefore, we create an adapter so that we can use them easily.

It also hides the implementation so that we can use it easily and with low coupling.

For instance, we can create an adapter object as follows:

```
const backEnd = {
  setAttribute(key, value) {
    //.,,
  },
  getAttribute() {
    //...
  }
}

const personAdapter = {
  setFirstName(name) {
    backEnd.setAttribute('firstName', name);
  },
  setLastName(name) {
    backEnd.setAttribute('lastName', name);
  },
  getFirstName() {
    backEnd.getAttribute('firstName');
  },
  getLastName() {
    backEnd.getAttribute('lastName');
  }
}
```

We created an adapter object called `personAdapter` so that we can use `backEnd` to set all the attributes.

The adapter has an interface that people can understand more easily since the method names are explicit.

Other objects or classes can use the object to manipulate the attributes.

We can use adapters to create interfaces that we want to expose to the public any way we want.

Inheriting Objects with Adapters

With adapters, we can also create an interface for multiple classes or objects.

For instance, we can write:

```
const backEnd = {
  setAttribute(key, value) {
    //.,,
  },
  getAttribute() {
    //...
  }
}

const parentAdapter = {
  //...
  getAdapterName() {
    //...
  }
}

const personAdapter = Object.create(parentAdapter);

personAdapter.setFirstName = (name) => {
  backEnd.setAttribute('firstName', name);
};

personAdapter.setLastName = (name) => {
  backEnd.setAttribute('lastName', name);
};

personAdapter.getFirstName = () => {
  backEnd.getAttribute('firstName');
};

personAdapter.getLastName = () => {
```

```
  backEnd.getAttribute('lastName');
}
```

In the code above, we created `personAdapter` with `Object.create` , which allows us to inherit from another parent object.

So we get the `getAdapterName` method from the `parentAdapter` prototype object.

Then we add our own methods to `personAdapter` .

Now we can get inherited methods from another object plus the methods in our own adapter object in the adapter.

Likewise, we can do the same with the class syntax.

For instance, we can create an adapter class instead of an object.

We can write:

```
const backEnd = {
  setAttribute(key, value) {
    //.,,
  },
  getAttribute() {
    //...
  }
}

class ParentAdapter {
  //...
  getAdapterName() {
    //...
  }
}

class PersonAdapter extends ParentAdapter {
  setFirstName(name) {
    backEnd.setAttribute('firstName', name);
```

```
  }

  setLastName(name) {
    backEnd.setAttribute('lastName', name);
  }

  getFirstName() {
    backEnd.getAttribute('firstName');
  }

  getLastName() {
    backEnd.getAttribute('lastName');
  }

}

const personAdapter = new PersonAdapter();
console.log(personAdapter);
```

The code above has a `PersonAdapter` and `ParentAdapter` classes instead of objects.

We used the `extends` keyword to inherit members from `ParentAdapter` .

This is another way we can inherit members from a parent entity.

Creating Algorithms

Once we decided on the patterns to use, then we've to devise our algorithm for our program.

This should be easy once we created some basic designs. Implementation is all that's needed in most cases.

We can use existing libraries to make our lives easier, which is what we should do in most cases.

If we have an is-a relationship between classes or objects, then we need to think about which pieces of code are shared and which ones are unique to classes or objects.

If we use a has-a model instead, then we can create algorithms in different places and use them as we wish to.

We should think about making our program configurable so that we don't have to change code to have slightly different functionality.

This way, we make our lives easier since code change always has some risks.

To reduce that, we should make things that change frequently be configurable so that we don't have to deal with them.

For instance, we can read settings from a file or database so an algorithm can be selected according to settings that are set in those places.

Strategy Pattern

The strategy design pattern is a design pattern that lets us select from a family of algorithms during run time.

The algorithms should be substitutable with each other.

This pattern's main idea is to let us create multiple algorithms for solving the same problem.

We have one object that does things one way and another that does things another way.

There may be more than one way to do the same thing.

For instance, we may have one function that calls a function depending on what we want to do.

We may write:

```
const smartStrategy = () => {
  //..
}

const smarterStrategy = () => {
  //..
}

const dumbStrategy = () => {
  //..
}

const doTask = () => {
  if (shouldBeSmart) {
    smartStrategy()
  } else if (shouldBeSmarter) {
    smarterStrategy()
  } else if (shouldBeDumb) {
```

```
    dumbStrategy()
  }
}
```

We have the `doTask` function that runs the functions that have the strategies that we want to use to accomplish a task.

This way, we can pick the algorithm that suits the task the most.

This can be from a user setting or it can be chosen automatically.

The whole idea is to define a family of algorithms, encapsulate each one, and make them interchangeable.

Making them interchangeable is important since we want them to accomplish the same results at the end.

The strategy pattern is good for separating volatile code from a part of the program so that the part that changes are separate from the stable code.

Also, using the strategy pattern, we don’t have to split the implementation code over several inherited classes as often.

Using this pattern reduces the chance of having many child classes for one parent class.

This is a simple pattern that let us choose different ways to do the same thing and reduce the risk of volatile code that break things by separating them out.

Algorithms are encapsulated by one interface that doesn’t change so that outside code can just use that instead of changing the inner workings of the code all the time.

Observer

The observer pattern is where we have multiple objects that listen to the one observable object.

This way, the observable object, which is also called the subject, can notify the objects that subscribed to the observable object with data that are emitted.

For instance, we can write:

```
const observable = {
  observers: {},
  subscribe(obj) {
    const id = Object.keys(this.observers).length + 1;
    this.observers[id] = obj;
    return id;
  },

  notify(data) {
    for (const o of Object.keys(this.observers)) {
      this.observers[o].listen(data);
    }
  }
}

const observer = {
  listen(data) {
    console.log(data);
  }
}
observable.subscribe(observer);
observable.notify({
  a: 1
});
```

We have the `observable` object that lets observer objects, which have a `listen` method to listen to data, to subscribe to notifications from the `observable` object.

When we can `observable.notify` with something as we did above, the `listen` method of the observer objects is run.

This way, the communication is all done by communicating via the `notify` method and nowhere else.

No implementations are exposed and therefore no tight coupling.

As long as the methods do the same thing, we shouldn't have to worry about breaking the observer objects that subscribe to the `observable` .

We can also add an `unsubscribe` method to `observable` which uses the returned `id` from `subscribe` to remove an observer object from the `observers` object.

For instance, we can write:

```
const observable = {
  observers: {},
  subscribe(obj) {
    const id = Object.keys(this.observers).length + 1;
    this.observers[id] = obj;
    return id;
  },

  notify(data) {
    for (const o of Object.keys(this.observers)) {
      this.observers[o].listen(data);
    }
  },

  unsubscribe(id) {
    delete this.observers[id];
  }
}

const observer = {
  listen(data) {
    console.log(data);
```

```
  }
}
const subscriberId = observable.subscribe(observer);
observable.notify({
  a: 1
});

observable.unsubscribe(subscriberId);
```

We added an `unsubscribe` method so that we can remove the observer object from the `this.observers` list.

Once we unsubscribed, we won't get notifications any more with the unsubscribed observer if we call `notify` again.

Examples of the observer pattern are used in many places.

Another good thing about the observer pattern is that we don't communicate anything through the classes.

Loose coupling should always be preferred to tight coupling.

We only communicate through one channel in the example above so the observer pattern is as loose as coupling can get.

They include message queues and event handlers for GUI events like mouse clicks and key presses.

Chain of Responsibility Pattern

The chain of responsibility is similar to the observer pattern, except that it sends the notification to one object, and then that object sends the notification to another object, and so on.

In the observer pattern, the notification is sent to all the observers at the same time.

For instance, if we send something to one object and then that object picks that up and does something with and send that to another and so on, then that implements the chain of responsibility patterns.

We can implement that as follows:

```
const backend = {
  receive(data) {
    // do something with data
  }
}

const middleLayer = {
  notify(data) {
    backend.receive(data);
  }
}

const frontEnd = {
  notify(data) {
    middleLayer.notify(data);
  }
}
```

In the code above, we have the `frontEnd` , which calls the `notify` method of the `middleLayer` object, which in turn called the `backend` ‘s `receive` method.

This way, we can pass data between them with one method seamlessly.

As long as we don’t expose any other methods that do communication between both objects, we have a clean way of sending data between `frontEnd` and `middleLayer` and `backEnd` .

Flyweight Pattern

The flyweight pattern is where we restrict object creation, We create a set of small objects that each consume a smaller amount of resources.

These objects sit behind a flyweight object, which is used to interact with these objects, Those objects also interact with our code.

This way, we get the benefits of a big object, while we interact with smaller objects that are more manageable.

For instance, we can implement it as follows:

```
class Student {
 //..
}

const studentIdentity = {
 getIdentity(){
   const student = new Student();
    //..
    return {
    //...
    }
  }
}

const studentScore = {
 getScore(){
   const student = new Student();
    //..
    return {
    //...
    }
  }
}
```

We have a `Student` class that represents a student's data.

Then in the `studentIdentity` and `studentScore` objects, we have methods to get his or her identity or score respectively.

This way, we don't have to put all the methods into the `Student` class.

Instead, we have methods outside of it that are smaller that we can use to deal with specific kinds of student data.

Conclusion

There are many design patterns we can adopt to better organize our code.

These are tested with time and many projects, so we should consider adopting them in our own JavaScript projects.