JavaScript design patterns – Part 2: Adapter, decorator, and factory

You have come to Part 2 of this JavaScript Design Patterns series. It's been a little while since Part 1, so you might want to refresh yourself on the singleton, composite, and façade patterns. Also, Part 3 discusses another 3 design patterns: proxy, observer, and command.

This time around, you learn about the Adapter, Decorator, and Factory patterns.

Adapter

The adapter pattern is allows you to transform (or *adapt*) an interface to your needs. This is done by creating another object that has the interface you desire and connecting to the object that you're trying to change the interface of.

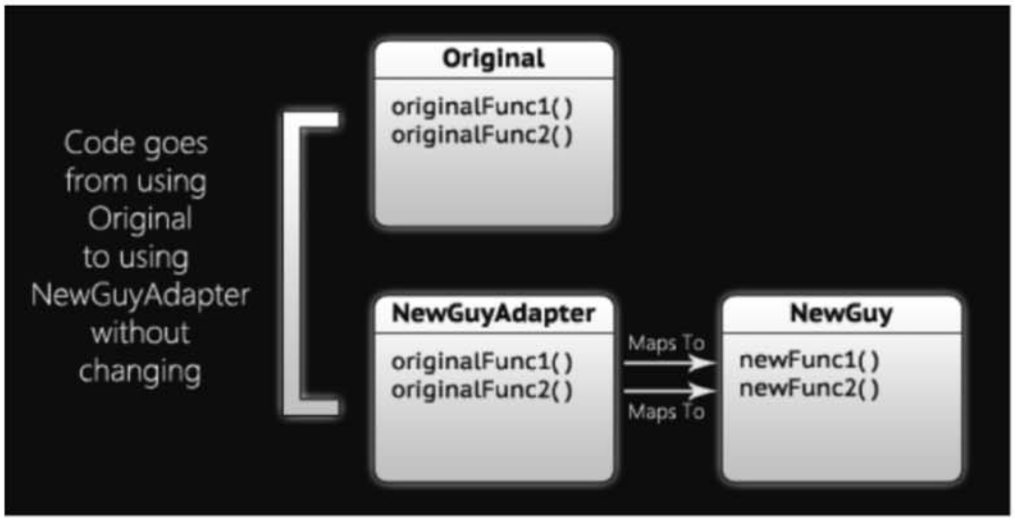

Figure 1. The structure of the adapter pattern

Why Do You Need the Adapter?

Often enough, you're working on an application and you decide that you need to replace a chunk of it, for example a library you're using for keeping logs or something of that nature. When you bring in a new library to replace it, it isn't likely to have the exact same interface. From here you have two options:

1. Go through all of your code and change everything that refers to the old library.
2. Create an adapter so that the new library can be used with the same interface as the old one.

Obviously in some cases, your application will be small enough or have few enough references to the old library that it feels more appropriate to just go back through the code and change it to match the new library rather than complicating your code with a new layer of abstraction. In most cases, though, it is more practical and time-saving to just create an adapter.

An Example of an Adapter

Why don't you just take the above hypothetical logger scenario and use that for our code example? Originally in your code you may use the console that is built into most browsers to make its logs, but there are a few problems with it: it's not built into all browsers (especially older browsers) and you can't see the logs that happen when other users use your application, so you don't get to see any of the problems that you didn't catch when you were testing it yourself. So you've decided that you want a logger than can use AJAX to ship those logs off to the server to handle. Here's what your new `AjaxLogger` library's API might look like:

```
AjaxLogger.sendLog(arguments);
AjaxLogger.sendInfo(arguments);
AjaxLogger.sendDebug(arguments);
etc...
```

Apparently, the author of this library didn't realize that you would be trying to use this to replace the console, so he or she felt it was neccesary to append "send" to the beginning of the name of each of the methods. Now you ask, "why don't I just edit the library and change the names of the methods?" There are a couple of good reasons not to do that. If you need to make an update to that library, your changes are be overwritten, so you need to go back and change them again. Also, if you pull your libraries down from a content delivery network, then you won't be able to edit them. So let's build an object that adapts this new library to the same interface as the console.

```
var AjaxLoggerAdapter = {
    log: function() {
        AjaxLogger.sendLog(arguments);
    },
    info: function() {
        AjaxLogger.sendInfo(arguments);
    },
    debug: function() {
        AjaxLogger.sendDebug(arguments);
    },
    ...
};
```

How Do You Use It?

I bet that anyone who uses the console calls on it directly by its reference, so how do you get every call to `console.xxx` to refer to the new adapter rather than to the console? If you had used an abstraction (like a factory) to retrieve the console, then you could just make a change in that abstraction layer, but as was already stated: everyone just refers directly to `console`. Well, JavaScript is a dynamic language, which allows us to change things at runtime, so why not just override console with the new `AjaxLoggerAdapter`?

```
window.console = AjaxLoggerAdapter;
```

That was easy, wasn't it? Be careful, though! If you do this in code that is meant to be used by others, the console no longer functions as the user expects. Also, don't be fooled by the simplicity of this example. In many cases, you won't have methods that easily map to each other (like `sendLog` to `log`). You may have to actually implement a bit of your own logic to convert the interface to be compatible with the new library.

Decorator

The decorator pattern is a very different beast from a lot of other patterns. The decorator pattern solves the problem of adding or changing functionality on a class without creating a subclass for every combination of functionality. For example, you have a car class with its default functionality. The car has several optional features that you can add on to the car (e.g. power locks, power windows, and air conditioning). If you tried taking care of this using subclassing, you would have a total of 8 classes to cover all of combinations:

- Car
- CarWithPowerLocks
- CarWithPowerWindows
- CarWithAc
- CarWithPowerLocksAndPowerWindows
- CarWithPowerLocksAndAc
- CarWithPowerWindowsAndAc
- CarWithPowerWindowsAndPowerLocksAndAc

This can very quickly get out of hand just by adding one more option which in turn would add 8 more subclasses. This can be solved using the decorator pattern, so that every time you add a new option, you create only one more class, rather than doubling the number of classes.

Structure of the Decorator

The decorator works by wrapping the base object (Car) with a decorator object that has the same interface as the base object. The decorator has a few options on how it can handle the methods:

1. It can completely override the methods of the object it wraps;

2. If the decorator doesn't affect the behavior of a method, it can just pass through to the wrapped object;
3. The decorator can add behavior before or after passing the call through to the wrapped object.

Decorators aren't limited to just wrapping the base object either; they can wrap other decorators, too, because they all implement the same interface. The general structure looks something like what is shown in figure 2.

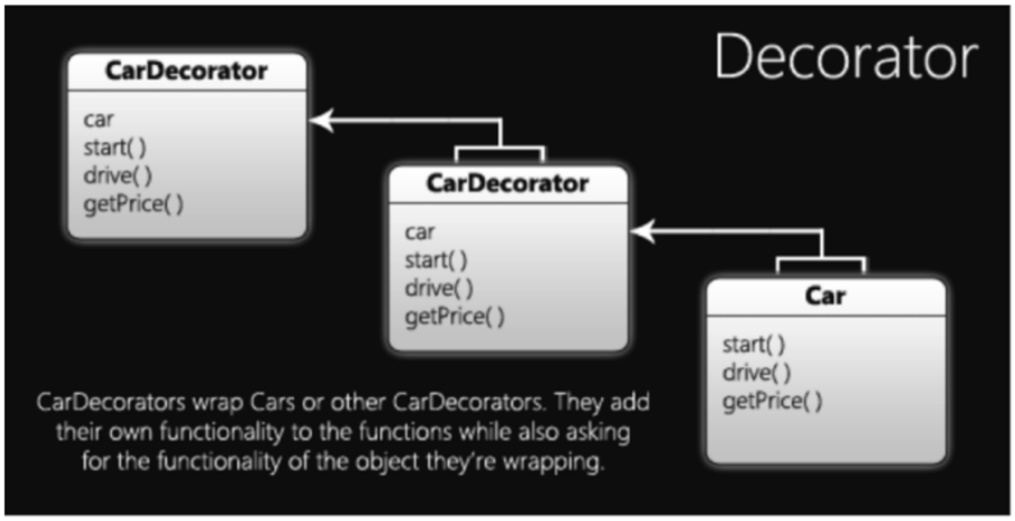
Decorator
CarDecorator
car
start( )
drive( )
getPrice( )
CarDecorator
car
start( )
drive( )
getPrice( )
Car
start( )
drive( )
getPrice( )
CarDecorators wrap Cars or other CarDecorators. They add their own functionality to the functions while also asking for the functionality of the object they're wrapping.

Figure 2. The structure of the decorator pattern

An Example of the Decorator Pattern

Let's take the car illustration above and flesh it out into code. Construct the base `Car` class first.

```
var Car = function() {
    console.log('Assemble: build frame, add core parts');
};

// The decorators will also need to implement this interface
Car.prototype = {
    start: function() {
        console.log('The engine starts with roar!');
    },
    drive: function() {
        console.log('Away we go!');
    },
    getPrice: function() {
        return 11CCC.CC;
    }
};
```

Keep things simple by just sending phrases to the console to show what's happening. Obviously, you could add a lot more functionality to this because cars are very complex beasts, but for the sake of this demonstration I'm sure you won't mind keeping things simple.

Now create the `CarDecorator` class. This is meant to be an abstract class that isn't instantiated itself, but should be subclassed to create the real decorators.

```
var CarDecorator = function(car) {
    this.car = car;
};

// CarDecorator implements the same interface as Car
CarDecorator.prototype = {
    start: function() {
        this.car.start();
    },
    drive: function() {
        this.car.drive();
    },
    getPrice: function() {
        return this.car.getPrice();
    }
};
```

There are a few important points to note here. First, the `CarDecorator` constructor takes a car, or rather, it takes an object that implements the same interface as `Car`, which includes multiple `Car` and subclasses of `CarDecorator`. Also note that all of the methods are just passing the requests through to wrapped object. This is the default behavior that all of our decorators should use for methods that aren't being changed.

Now create all of the decorators. Since you're inheriting from `CarDecorator`, you only need to override the functionality that changes, otherwise `CarDecorator` works perfectly.

```
var PowerLocksDecorator = function(car) {
    // Call Parent Constructor
    CarDecorator.call(this, car);
    console.log('Assemble: add power locks');
};
PowerLocksDecorator.prototype = new CarDecorator();
PowerLocksDecorator.prototype.drive = function() {
    // You can either do this
    this.car.drive();
    // or you can call the parent's drive function:
    // CarDecorator.prototype.drive.call(this);
    console.log('The doors automatically lock');
};
PowerLocksDecorator.prototype.getPrice = function() {
    return this.car.getPrice() + 1CC;
};

var PowerWindowsDecorator = function(car) {
    CarDecorator.call(this, car);
    console.log('Assemble: add power windows');
};
PowerWindowsDecorator.prototype = new CarDecorator();
PowerWindowsDecorator.prototype.getPrice = function() {
    return this.car.getPrice() + 2CC;
};

var AcDecorator = function(car) {
    CarDecorator.call(this, car);
    console.log('Assemble: add A/C unit');
};
AcDecorator.prototype = new CarDecorator();
AcDecorator.prototype.start = function() {
    this.car.start();
    console.log('The cool air starts blowing.');
};
AcDecorator.prototype.getPrice = function() {
    return this.car.getPrice() + 6CC;
};
```

In this example, whenever I created a method that added functionality to the original, I just called `this.car.x()` rather than calling the method on the parent class. Generally it'd be wiser to call the parent's method, but since this is so simple, I figured it'd be simpler to just work with the car property directly. This could come back to hurt me if I need to change the default behavior built into `CarDecorator` to do something other than just pass requests through, but, considering this is only an example, I don't foresee that happening.

So, now that you have all of the elements you need, put them to action.

```
var car = new Car();                    // log "Assemble: build frame, add core
```

```
// give the car some power windows
car = new PowerWindowDecorator(car);     // log "Assemble: add power windows"

// now some power locks and A/C
car = new PowerLocksDecorator(car);      // log "Assemble: add power locks"
car = new AcDecorator(car);              // log "Assemble: add A/C unit"

// let's start this bad boy up and take a drive!
car.start(); // log 'The engine starts with roar!' and 'The cool air starts blo
car.drive(); // log 'Away we go!' and 'The doors automatically lock'
```

It's simple to create the car with the features that you want. Just create the car and the decorators you need, while adding the car with its current feature set to the decorators as you go. The creation code is pretty long though, especially compared to just instantiating one object like `CarWithPowerLocksAndPowerWindowsAndAc`. Don't worry though, I'll show you a much nicer way to create these decorated objects in the section about the factory pattern. In the meantime, if you'd like to read up a bit more on the decorator pattern you can read "JavaScript Design Patterns: Decorator" on my personal blog.

Factory

The factory gets its name from its intended purpose, simplifying the creation of objects. Simple factories abstract out all of those uses of the new keyword so that if a class name changes or is replaced by a different one, you only need to change it in one place. Plus it sets up a one-stop-shop for creating many different types of objects or a single type of object with different options. The standard factory is a little tougher to explain in so few words, so I bring that up later.

A Simple JavaScript Factory

This example is a simple factory using the conundrum we encountered with the decorator example above: the actual code to create a car with features is way too long. Using a factory, you cut this all down to a single function call. Start off with an object literal containing a single function. In JavaScript, an object literal/singleton is how a simple factory is built. In classical object-oriented programming languages, this would be a static class.

```
var CarFactory = {
    makeCar: function(features) {
        var car = new Car();

        // If they specified some features then add them
        if (features && features.length) {
            var i = 0,
                l = features.length;

            // iterate over all the features and add them
            for (; i < l; i++) {
                var feature = features[i];

                switch(feature) {
                    case 'powerwindows':
                        car = new PowerWindowsDecorator(car);
                        break;
                    case 'powerlocks':
                        car = new PowerLocksDecorator(car);
                        break;
                    case 'ac':
                        car = new ACDecorator(car);
                        break;
                }
            }
        }

        return car;
    }
}
```

The one function that this factory has is `makeCar`, which does a lot of heavy lifting. First of all, the one argument it accepts is an array of strings, which map to the different decorator classes. `makeCar` creates a plain `Car` object, then iterates through the features and decorates the car. Now, instead of at least four lines of code to create a car with all of the features, we just need one:

```
Var myCar = CarFactory.makeCar(['powerwindows', 'powerlocks', 'ac']);
```

You can even adjust the `makeCar` function so that it makes sure only one of any type of decorator is used and that they are attached in a specific order (like if they were actually being made in a factory). If you'd like to see an example of that and a few other great ways to use the factory pattern, you can take a look at JavaScript Design Patterns: Factory on my personal blog.

Standard Factory

The standard factory pattern is quite different from the simple factory, but of course, still has the role of creating objects. Instead of using a singleton, though, just use an abstract method on a class. For example, let's pretend there are several different car manufacturers out there and they all have their own shops. For the most part all of the shops utilize the same methods for selling cars, except they manufacture different cars. So they all inherit their methods from the same prototype, but they implement their own manufacturing process.

Let's turn this example into code to help see what I'm talking about. First create a car shop that is only meant to be subclassed and has one method – `manufactureCar` – that is a stub. It throws an error unless it is overwritten by subclass.

```
/* Abstract CarShop "class" */
var CarShop = function(){};
CarShop.prototype = {
    sellCar: function (type, features) {
        var car = this.manufactureCar(type, features);

        getMoney(); // make-believe function

        return car;
    },
    decorateCar: function (car, features) {
        /*
        Decorate the car with features using the same
        technique laid out in the simple factory
        */
    },
    manufactureCar: function (type, features) {
        throw new Error("manufactureCar must be implemented by a subclass");
    }
};
```

Notice that `sellCar` calls `manufactureCar`. That means `manufactureCar` needs to be implemented by a subclass in order for you to sell a car. So create a couple car shop to see how they implement it.

```
/* Subclass CarShop and create factory method */
var JoeCarShop = function() {};
JoeCarShop.prototype = new CarShop();
JoeCarShop.prototype.manufactureCar = function (type, features) {
    var car;

    // Create a different car depending on what type the user specified
    switch(type) {
        case 'sedan':
            car = new JoeSedanCar();
            break;
        case 'hatchback':
            car = new JoeHatchbackCar();
            break;
        case 'coupe':
        default:
            car = new JoeCoupeCar();
    }

    // Decorate the car with the specified features
    return this.decorateCar(car, features);
};


/* Another CarShop and with factory method */
var ZimCarShop = function() {};
ZimCarShop.prototype = new CarShop();
ZimCarShop.prototype.manufactureCar = function (type, features) {
    var car;

    // Create a different car depending on what type the user specified
    // These are all Zim brand
    switch(type) {
        case 'sedan':
            car = new ZimSedanCar();
            break;
        case 'hatchback':
            car = new ZimHatchbackCar();
            break;
        case 'coupe':
        default:
            car = new ZimCoupeCar();
    }

    // Decorate the car with the specified features
    return this.decorateCar(car, features);
};
```

The methods are essentially identical in how they work, except that they each create different cars (for conciseness I left out the implementation for the car classes because they weren't necessary). The point is that the `manufactureCar` method is the factory method. A factory method is abstract on the parent class, implemented by the subclasses, and its job is to create objects (each factory method creates objects with the same interface, too).

Finishing Touches

That covers the basics behind the factory pattern. If you're looking to read up a little more on the factory pattern, you can visit JavaScript Design Patterns: Factory (simple factory) and JavaScript Design Patterns: Factory Part 2 (standard factory) on my personal blog.