

CHAPTER 13



The Flyweight Pattern

In this chapter, we examine another optimization pattern, the *flyweight*. It's most useful in situations where large numbers of similar objects are created, causing performance problems. This pattern is especially useful in JavaScript, where complex code can quickly use up all of the available browser memory. By converting many independent objects into a few shared objects, you can reduce the amount of resources needed to run web applications. The benefit of this can vary widely. For large applications that can potentially be used for days at a time without being reloaded, any technique that reduces the amount of memory used can have a very positive effect. For small pages that won't stay open in the browser for that long, memory conservation isn't as important.

The Structure of the Flyweight

It can be confusing at first to understand how the flyweight pattern works. Let's first take a high-level look at the structure. We will then explain each individual part in more detail.

The flyweight pattern is used to reduce the number of objects you need in your application. This is accomplished by dividing an object's internal state into two categories, *intrinsic data* and *extrinsic data*. Intrinsic data is the information that is required by the internal methods of a class; the class cannot function properly without this data. Extrinsic data is information that can be removed from a class and stored externally. We can take all of the objects that have the same intrinsic state and replace them with a single shared object, thus reducing the number of objects down to the number of unique intrinsic states you have.

Instead of using a normal constructor, a factory is used to create these shared objects. That way, you can track the objects that have already been instantiated and only create a new copy if the needed intrinsic state is different from an object you already have. A manager object is used to store the object's extrinsic state. When invoking any of the objects' methods, the manager will pass in these extrinsic states as arguments.

Let's drill down into each of these pieces.

Example: Car Registrations

Imagine that you need to create a system to represent all of the cars in a city. You need to store the details about each car (make, model, and year) and the details about each car's ownership (owner name, tag number, last registration date). Naturally, you choose to represent each car as an object:

```
/* Car class, un-optimized. */

var Car = function(make, model, year, owner, tag, renewDate) {
    this.make = make;
    this.model = model;
    this.year = year;
    this.owner = owner;
    this.tag = tag;
    this.renewDate = renewDate;
};

Car.prototype = {
    getMake: function() {
        return this.make;
    },
    getModel: function() {
        return this.model;
    },
    getYear: function() {
        return this.year;
    },
    transferOwnership: function(newOwner, newTag, newRenewDate) {
        this.owner = newOwner;
        this.tag = newTag;
        this.renewDate = newRenewDate;
    },
    renewRegistration: function(newRenewDate) {
        this.renewDate = newRenewDate;
    },
    isRegistrationCurrent: function() {
        var today = new Date();
        return today.getTime() < Date.parse(this.renewDate);
    }
};
```

This works well for a while, but as the population of your city increases, you notice that the system is running a little more slowly each day. Using hundreds of thousands of car objects has overwhelmed the available computing resources. To optimize this system, use the flyweight pattern to reduce the number of objects needed.

The first step is to separate the intrinsic state from the extrinsic state.

Intrinsic and Extrinsic State

Categorizing an object's data as intrinsic or extrinsic can be a bit arbitrary. You want to make as much of the data as possible extrinsic while still maintaining the modularity of each object. This distinction can be somewhat arbitrary. In this example, the physical car data (make, model, year) is intrinsic, and the owner data (owner name, tag number, last registration date) is extrinsic. This means that only one car object is needed for each combination of make, model, and

year. This is still a lot of objects, but it is several orders of magnitude fewer than before. The single instance of each make-model-year combination will be shared among all the owners of that type of car. Here is what the new Car class looks like. We explain later in the “Extrinsic State Encapsulated in a Manager” section where the extrinsic data goes.

```
/* Car class, optimized as a flyweight. */

var Car = function(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
};
Car.prototype = {
  getMake: function() {
    return this.make;
  },
  getModel: function() {
    return this.model;
  },
  getYear: function() {
    return this.year;
  }
};
```

All of the extrinsic data has been removed. All of the methods dealing with registration have been moved to a manager object (although you could have also left the methods in place and added arguments for all of the extrinsic data). Because the object’s data is split up, you must now use a factory to instantiate it.

Instantiation Using a Factory

The factory is fairly simple. It checks to see whether a car of this particular make-model-year combination has been created before. If so, it returns it. If not, it creates a new car and stores it for later use. This ensures that only a single copy of each unique intrinsic state is created:

```
/* CarFactory singleton. */

var CarFactory = (function() {

  var createdCars = {};

  return {
    createCar: function(make, model, year) {
      // Check to see if this particular combination has been created before.
      if(createdCars[make + '-' + model + '-' + year]) {
        return createdCars[make + '-' + model + '-' + year];
      }
      // Otherwise create a new instance and save it.
    }
  };
})();
```

```

    else {
        var car = new Car(make, model, year);
        createdCars[make + '-' + model + '-' + year] = car;
        return car;
    }
}
};
})();

```

Extrinsic State Encapsulated in a Manager

One more object is needed to finish this optimization. All of the data that was removed from the Car objects has to be stored somewhere; you use a singleton as a manager to encapsulate that data. Each of the old-style Car objects is now split into the extrinsic data and the reference to the shared car object that the data belongs to. The combination of a Car object and the owner data will be referred to as a *car record*. The manager stores both of those pieces. It also contains the methods removed from the old Car class:

```

/* CarRecordManager singleton. */

var CarRecordManager = (function() {

    var carRecordDatabase = {};

    return {
        // Add a new car record into the city's system.
        addCarRecord: function(make, model, year, owner, tag, renewDate) {
            var car = CarFactory.createCar(make, model, year);
            carRecordDatabase[tag] = {
                owner: owner,
                renewDate: renewDate,
                car: car
            };
        },

        // Methods previously contained in the Car class.
        transferOwnership: function(tag, newOwner, newTag, newRenewDate) {
            var record = carRecordDatabase[tag];
            record.owner = newOwner;
            record.tag = newTag;
            record.renewDate = newRenewDate;
        },
        renewRegistration: function(tag, newRenewDate) {
            carRecordDatabase[tag].renewDate = newRenewDate;
        },
        isRegistrationCurrent: function(tag) {
            var today = new Date();
            return today.getTime() < Date.parse(carRecordDatabase[tag].renewDate);
        }
    };
})();

```

```

    }
  };
})();

```

All of the data pulled out of the `Car` class is now stored in a private attribute of the `CarRecordManager` singleton, called `CarRecordDatabase`. This single `CarRecordDatabase` object is much more efficient than the huge number of objects used before. The methods dealing with ownership are now encapsulated here as well, since they all deal with extrinsic data.

As you can see, this optimization does come at a price in complexity. There is now one class and two singleton objects, where before there was only a single class. It is also a little confusing to have data for an object stored in two different places. But both of these concerns are small compared to the performance issues that have been addressed. When used in the appropriate situation, the flyweight pattern excels at making your program more efficient.

Managing Extrinsic State

There are many different ways to manage a flyweight object's extrinsic state. One common way is to use a manager object, which contains a centralized database of the extrinsic states and the flyweight object they belong to. This is the approach used in the `Car` example; it has the advantage of being simple and easy to maintain. It's also a fairly lightweight approach, since you are only using a single array or object literal to store the extrinsic data. We use this approach again later in the tooltip objects example.

Another way to manage extrinsic state is to use a *composite*. With the composite pattern, covered in Chapter 9, you can use the hierarchy of the object itself to store information, without the need for a centralized database. The leaf nodes can all be flyweight objects, allowing them to be shared among many locations in the composite's hierarchy. This can be extremely useful for large hierarchies, as the same data can be represented with far fewer unique objects.

Example: Web Calendar

To illustrate how a composite can be used to store extrinsic state, let's create a web calendar. First, the unoptimized, nonflyweight version is implemented. This is a large composite, starting with the composite object that represents the year. This encapsulates the month composite objects, which in turn encapsulate the day leaves. This is a simple example; it displays the days sequentially within each month, and the month sequentially within each year:

```

/* CalendarItem interface. */

var CalendarItem = new Interface('CalendarItem', ['display']);

/* CalendarYear class, a composite. */

var CalendarYear = function(year, parent) { // implements CalendarItem
  this.year = year;
  this.element = document.createElement('div');
  this.element.style.display = 'none';
  parent.appendChild(this.element);

```

```
function isLeapYear(y) {
    return (y > 0) && !(y % 4) && ((y % 100) || !(y % 400));
}

this.months = [];
// The number of days in each month.
this.numDays = [31, isLeapYear(this.year) ? 29 : 28, 31, 30, 31, 30, 31, 31, 30,
    31, 30, 31];
for(var i = 0, len = 12; i < len; i++) {
    this.months[i] = new CalendarMonth(i, this.numDays[i], this.element);
}
);
CalendarYear.prototype = {
    display: function() {
        for(var i = 0, len = this.months.length; i < len; i++) {
            this.months[i].display(); // Pass the call down to the next level.
        }
        this.element.style.display = 'block';
    }
};

/* CalendarMonth class, a composite. */
var CalendarMonth = function(monthNum, numDays, parent) { // implements CalendarItem
    this.monthNum = monthNum;
    this.element = document.createElement('div');
    this.element.style.display = 'none';
    parent.appendChild(this.element);

    this.days = [];
    for(var i = 0, len = numDays; i < len; i++) {
        this.days[i] = new CalendarDay(i, this.element);
    }
};
CalendarMonth.prototype = {
    display: function() {
        for(var i = 0, len = this.days.length; i < len; i++) {
            this.days[i].display(); // Pass the call down to the next level.
        }
        this.element.style.display = 'block';
    }
};

/* CalendarDay class, a leaf. */
var CalendarDay = function(date, parent) { // implements CalendarItem
    this.date = date;
    this.element = document.createElement('div');
    this.element.style.display = 'none';
```

```

    parent.appendChild(this.element);
  };
  CalendarDay.prototype = {
    display: function() {
      this.element.style.display = 'block';
      this.element.innerHTML = this.date;
    }
  };
};

```

The problem with this code is that you have to create 365 `CalendarDay` objects for each year. To create a calendar that displays ten years, several thousand `CalendarDay` objects would be instantiated. Granted, these objects are not especially large, but so many objects of any type can stress the resources of a browser. It would be more efficient to use a single `CalendarDay` object for all days, no matter how many years you are displaying.

Converting the Day Objects to Flyweights

It is a simple process to convert the `CalendarDay` objects to flyweight objects. First, modify the `CalendarDay` class itself and remove all of the data that is stored within it. These pieces of data (the date and the parent element) will become extrinsic data:

```

/* CalendarDay class, a flyweight leaf. */

var CalendarDay = function() {}; // implements CalendarItem
CalendarDay.prototype = {
  display: function(date, parent) {
    var element = document.createElement('div');
    parent.appendChild(element);
    element.innerHTML = date;
  }
};

```

Next, create a single instance of the day object. This instance will be used in all `CalendarMonth` objects. A factory could be used here, as in the first example, but since you are only creating one instance of this class, you can simply instantiate it directly:

```

/* Single instance of CalendarDay. */

var calendarDay = new CalendarDay();

```

The extrinsic data is passed in as arguments to the `display` method, instead of as arguments to the class constructor. This is typically how flyweights work; because some (or all) of the data is stored outside of the object, it must be passed in to the methods in order to perform the same functions as before.

The last step is to modify the `CalendarMonth` class slightly. Remove the arguments to the `CalendarDay` class constructor and pass them instead to the `display` method. You also want to replace the `CalendarDay` class constructor with the `CalendarDay` object:

```
/* CalendarMonth class, a composite. */

var CalendarMonth = function(monthNum, numDays, parent) { // implements CalendarItem
    this.monthNum = monthNum;
    this.element = document.createElement('div');
    this.element.style.display = 'none';
    parent.appendChild(this.element);

    this.days = [];
    for(var i = 0, len = numDays; i < len; i++) {
        this.days[i] = calendarDay;
    }
};

CalendarMonth.prototype = {
    display: function() {
        for(var i = 0, len = this.days.length; i < len; i++) {
            this.days[i].display(i, this.element);
        }
        this.element.style.display = 'block';
    }
};
```

Where Do You Store the Extrinsic Data?

Unlike the previous example, a central database was not created to store all of the data pulled out of the flyweight objects. In fact, the other classes were barely modified at all; `CalendarYear` was completely untouched, and `CalendarMonth` only needed two lines changed. This is possible because the structure of the composite already contains all of the extrinsic data in the first place. The month object knows the date of each day because the day objects are stored sequentially in an array. Both pieces of data removed from the `CalendarDay` constructor are already stored in the `CalendarMonth` object.

This is why the composite pattern works so well with the flyweight pattern. A composite object will typically have large numbers of leaves and will also already be storing much of the data that could be made extrinsic. The leaves usually contain very little intrinsic data, so they can become a shared resource very easily.

Example: Tooltip Objects

The flyweight pattern is especially useful when your JavaScript objects need to create HTML. Having a large number of objects that each create a few DOM elements can quickly bog down your page by using too much memory. The flyweight pattern allows you to create only a few of these objects and share them across all of the places they are needed. A perfect example of this can be found in tooltips.

A *tooltip* is the hovering block of text you see when you hold your cursor over a tool in a desktop application. It usually gives you information about the tool, so that the user can know what it does without clicking on it first. This can be very useful in web apps as well, and it is fairly easy to implement in JavaScript.

The Unoptimized Tooltip Class

First, create a class that does not use the flyweight pattern. Here is a `Tooltip` class that will do the job:

```
/* Tooltip class, un-optimized. */

var Tooltip = function(targetElement, text) {
    this.target = targetElement;
    this.text = text;
    this.delayTimeout = null;
    this.delay = 1500; // in milliseconds.

    // Create the HTML.
    this.element = document.createElement('div');
    this.element.style.display = 'none';
    this.element.style.position = 'absolute';
    this.element.className = 'tooltip';
    document.getElementsByTagName('body')[0].appendChild(this.element);
    this.element.innerHTML = this.text;

    // Attach the events.
    var that = this; // Correcting the scope.
    addEvent(this.target, 'mouseover', function(e) { that.startDelay(e); });
    addEvent(this.target, 'mouseout', function(e) { that.hide(); });
};

Tooltip.prototype = {
    startDelay: function(e) {
        if(this.delayTimeout == null) {
            var that = this;
            var x = e.clientX;
            var y = e.clientY;
            this.delayTimeout = setTimeout(function() {
                that.show(x, y);
            }, this.delay);
        }
    },
    show: function(x, y) {
        clearTimeout(this.delayTimeout);
        this.delayTimeout = null;
        this.element.style.left = x + 'px';
        this.element.style.top = (y + 20) + 'px';
        this.element.style.display = 'block';
    },
    hide: function() {
        clearTimeout(this.delayTimeout);
        this.delayTimeout = null;
        this.element.style.display = 'none';
    }
};
```

In the constructor, attach event listeners to the `mouseover` and `mouseout` events. There is a problem here: these event listeners are normally executed in the scope of the HTML element that triggered them. That means the `this` keyword will refer to the element, not the `Tooltip` object, and the `startDelay` and `hide` methods will not be found. To fix this problem, you can use a trick that allows you to call methods even when the `this` keyword no longer points to the correct object. Declare a new variable, called `that`, and assign `this` to it. `that` is a normal variable, and it won't change depending on the scope of the listener, so you can use it to call `Tooltip` methods.

This class is very easy to use. Simply instantiate it and pass in a reference to an element on the page and the text you want to display. The `$` function is used here to get a reference to an element based on its ID:

```
/* Tooltip usage. */

var linkElement = $('#link-id');
var tt = new Tooltip(linkElement, 'Lorem ipsum...');
```

But what happens if this is used on a page that has hundreds of elements that need tooltips, or even thousands? It means there will be thousands of instances of the `Tooltip` class, each with its own attributes, DOM elements, and styles on the page. This is not very efficient.

Since only one tooltip can be displayed at a time, it doesn't make sense to recreate the HTML for each object. Implementing each `Tooltip` object as a flyweight means there will be only one instance of it, and that a manager object will pass in the text to be displayed as extrinsic data.

Tooltip As a Flyweight

To convert the `Tooltip` class to a flyweight, three things are needed: the modified `Tooltip` object with extrinsic data removed, a factory to control how `Tooltip` is instantiated, and a manager to store the extrinsic data. You can get a little creative in this example and use one singleton for both the factory and the manager. You can also store the extrinsic data as part of the event listener, so a central database isn't needed.

First remove the extrinsic data from the `Tooltip` class:

```
/* Tooltip class, as a flyweight. */

var Tooltip = function() {
  this.delayTimeout = null;
  this.delay = 1500; // in milliseconds.

  // Create the HTML.
  this.element = document.createElement('div');
  this.element.style.display = 'none';
  this.element.style.position = 'absolute';
  this.element.className = 'tooltip';
  document.getElementsByTagName('body')[0].appendChild(this.element);
};
Tooltip.prototype = {
  startDelay: function(e, text) {
```

```

    if(this.delayTimeout == null) {
        var that = this;
        var x = e.clientX;
        var y = e.clientY;
        this.delayTimeout = setTimeout(function() {
            that.show(x, y, text);
        }, this.delay);
    }
},
show: function(x, y, text) {
    clearTimeout(this.delayTimeout);
    this.delayTimeout = null;
    this.element.innerHTML = text;
    this.element.style.left = x + 'px';
    this.element.style.top = (y + 20) + 'px';
    this.element.style.display = 'block';
},
hide: function() {
    clearTimeout(this.delayTimeout);
    this.delayTimeout = null;
    this.element.style.display = 'none';
}
};

```

As you can see, all arguments from the constructor and the event attachment code are removed. A new argument is also added to the startDelay and show methods. This makes it possible to pass in the text as extrinsic data.

Next comes the factory and manager singleton. The Tooltip declaration will be moved into the TooltipManager singleton so that it cannot be instantiated anywhere else:

```

/* TooltipManager singleton, a flyweight factory and manager. */

var TooltipManager = (function() {
    var storedInstance = null;

    /* Tooltip class, as a flyweight. */

    var Tooltip = function() {
        ...
    };
    Tooltip.prototype = {
        ...
    };

    return {
        addTooltip: function(targetElement, text) {
            // Get the tooltip object.
            var tt = this.getTooltip();

```

```

    // Attach the events.
    addEvent(targetElement, 'mouseover', function(e) { tt.startDelay(e, text); });
    addEvent(targetElement, 'mouseout', function(e) { tt.hide(); });
  },
  getTooltip: function() {
    if(storedInstance == null) {
      storedInstance = new Tooltip();
    }
    return storedInstance;
  }
};
})();

```

It has two methods, one for each of its two roles. `getTooltip` is the factory method. It is identical to the other flyweight creation method you have seen so far. The manager method is `addTooltip`. It fetches a `Tooltip` object and creates the `mouseover` and `mouseout` events using anonymous functions. You do not have to create a central database in this example because the closures created within the anonymous functions store the extrinsic data for you.

The code needed to create one of these tooltips looks a little different now. Instead of instantiating `Tooltip`, you call the `addTooltip` method:

```

/* Tooltip usage. */

TooltipManager.addTooltip($('link-id'), 'Lorem ipsum...');

```

What did you gain by making this conversion to a flyweight object? The number of DOM elements that need to be created is reduced to one. This is a big deal; if you add features like a drop shadow or an `iframe` shim to the tooltip, you could quickly have five to ten DOM elements per object. A few hundred or thousand tooltips would then completely kill the page if they aren't implemented as flyweights. Also, the amount of data stored within objects is reduced. In both cases, you can create as many tooltips as you want (within reason) without having to worry about having thousands of `Tooltip` instances floating around.

Storing Instances for Later Reuse

Another related situation well suited to the use of the flyweight pattern is modal dialog boxes. Like a tooltip, a dialog box object encapsulates both data and HTML. However, a dialog box contains many more DOM elements, so it is even more important to minimize the number of instances created. The problem is that it may be possible to have more than one dialog box on the page at a time. In fact, you can't know exactly how many you will need. How, then, can you know how many instances to allow?

Since the exact number of instances needed at run-time can't be determined at development time, you can't limit the number of instances created. Instead, you only create as many as you need and store them for later use. That way you won't have to incur the creation cost again, and you will only have as many instances as are absolutely needed.

The implementation details of the `DialogBox` object don't really matter in this example. You only need to know that it is resource-intensive and should be instantiated as infrequently as possible. Here is the interface and the skeleton for the class that the manager will be programmed to use:

```
/* DisplayModule interface. */  
  
var DisplayModule = new Interface('DisplayModule', ['show', 'hide', 'state']);  
  
/* DialogBox class. */  
  
var DialogBox = function() { // implements DisplayModule  
    ...  
};  
DialogBox.prototype = {  
    show: function(header, body, footer) { // Sets the content and shows the  
        ... // dialog box.  
    },  
    hide: function() { // Hides the dialog box.  
        ...  
    },  
    state: function() { // Returns 'visible' or 'hidden'.  
        ...  
    }  
};
```

As long as the class implements the three methods defined in the `DisplayModule` interface (`show`, `hide`, and `save`), the specific implementation isn't important. The important part of this example is the manager that will control how many of these flyweight objects get created. The manager needs three components: a method to display a dialog box, a method to check how many dialog boxes are currently in use on the page, and a place to store instantiated dialog boxes. These components will be packaged in a singleton to ensure that only one manager exists at a time:

```
/* DialogBoxManager singleton. */  
  
var DialogBoxManager = (function() {  
    var created = []; // Stores created instances.  
  
    return {  
        displayDialogBox: function(header, body, footer) {  
            var inUse = this.numberInUse(); // Find the number currently in use.  
            if(inUse > created.length) {  
                created.push(this.createDialogBox()); // Augment it if need be.  
            }  
            created[inUse].show(header, body, footer); // Show the dialog box.  
        },  
        createDialogBox: function() { // Factory method.  
            var db = new DialogBox();  
            return db;  
        },  
        numberInUse: function() {  
            var inUse = 0;
```

```
        for(var i = 0, len = created.length; i < len; i++) {  
            if(created[i].state() === 'visible') {  
                inUse++;  
            }  
        }  
        return inUse;  
    }  
};  
})();
```

The array `created` stores the objects that are already instantiated so they can be reused. The `numberInUse` method returns the number of existing `DialogBox` objects that are in use by querying their state. This provides a number to be used as an index to the `created` array. The `displayDialogBox` method first checks to see if this index is greater than the length of the array; you will only create a new instance if you can't reuse an already existing instance.

This example is a bit more complex than the tooltip example, but the same principles are used in each. Reuse the resource intensive objects by pulling the extrinsic data out of them. Create a manager that limits how many objects are instantiated and stores the extrinsic data. Only create as many instances as are needed, and if instantiation is an expensive process, save these instances so that they can be reused later. This technique is similar to pooling SQL connections in server-side languages. A new connection is created only when all of the other existing connections are already in use.

When Should the Flyweight Pattern Be Used?

There are a few conditions that should be met before attempting to convert your objects to flyweights. Your page must use a large number of resource-intensive objects. This is the most important condition; it isn't worth performing this optimization if you only expect to use a few copies of the object in question. How many is a "large number"? Browser memory and CPU usage can both potentially limit the number of resources you can create. If you are instantiating enough objects to cause problems in those areas, it is certainly enough to qualify.

The next condition is that at least some of the data stored within each of these objects must be able to be made extrinsic. This means you must be able to move some internally stored data outside of the object and pass it into the methods as an argument. It should also be less resource-intensive to store this data externally, or you won't actually be improving performance. If an object contains a lot of infrastructure code and HTML, it will probably make a good candidate for this optimization. If it is nothing more than a container for data and methods for accessing that data, the results won't be quite so good.

The last condition is that once the extrinsic data is removed, you must be left with a relatively small number of unique objects. The best-case scenario is that you are left with a single unique object, as in the calendar and tooltip examples. It isn't always possible to reduce the number of instances down to one, but you should try to end up with as few unique instances of your object as possible. This is especially true if you need multiple copies of each of these unique objects, as in the dialog box example.

General Steps for Implementing the Flyweight Pattern

If all of these three conditions are met, your program is a good candidate for optimization using the flyweight pattern. Almost all implementations of flyweight use the same general steps:

1. Strip all extrinsic data from the target class. This is done by removing as many of the attributes from the class as possible; these should be the attributes that change from instance to instance. The same goes for arguments to the constructor. These arguments should instead be added to the class's methods. Instead of being stored within the class, this data will be passed in by the manager. The class should still be able to perform that same function as before. The only difference is that the data comes from a different place.
2. Create a factory to control how the class is instantiated. This factory needs to keep track of all the unique instances of the class that have been created. One way to do this is to keep a reference to each object in an object literal, indexed by the unique set of arguments used to create it. That way, when a request is made for an object, the factory can first check the object literal to see whether this particular request has been made before. If so, it can simply return the reference to the already existing object. If not, it will create a new instance, store a reference to it in the object literal, and return it.

Another technique, *pooling*, uses an array to keep references to the instantiated objects. This is useful if the number of available objects is what is important, not the uniquely configured instances. Pooling can be used to keep the number of instantiated objects down to a minimum. The factory handles all aspects of creating the objects, based on the intrinsic data.

3. Create a manager to store the extrinsic data. The manager object controls all aspects dealing with the extrinsic data. Before implementing the optimization, you created new instances of the target class each time you needed it, passing in all data. Now, any time you need an instance, you will call a method of the manager and pass all the data to it instead. This method determines what is intrinsic data and what is extrinsic. The intrinsic data is passed on to the factory object so that an object can be created (or reused, if one already exists). The extrinsic data gets stored in a data structure within the manager. The manager then passes this data, as needed, to the methods of the shared objects, thus achieving the same result as if the class had many instances.

Benefits of the Flyweight Pattern

The flyweight pattern can reduce your page's resource load by several orders of magnitude. In the example on tooltips, the number of `ToolTip` objects (and the HTML elements that it creates) was cut down to a single instance. If the page uses hundreds or thousands of tooltips, which is typical for a large desktop-style app, the potential savings is enormous. Even if you aren't able to reduce the number of instances down to one, it is still possible to get very significant savings out of the flyweight pattern.

It doesn't require huge changes to your code to get these savings. Once you have created the manager, the factory, and the flyweight, the only change you must make to your code is to call a method of the manager object instead of instantiating the class directly. If you are creating the flyweight for other programmers to use as an API, they need only slightly alter the way they call it to get the benefits. This is where the pattern really shines; if you make this optimization to your API once, it will be much more efficient for everyone else who uses it. When using this optimization for a library that is used over an entire site, your users may well notice a huge improvement in speed.

Drawbacks of the Flyweight Pattern

This is only an optimization pattern. It does nothing other than improve the efficiency of your code under a strict set of conditions. It can't, and shouldn't, be used everywhere; it can actually make your code less efficient if used unnecessarily. In order to optimize your code, this pattern adds complexity, which makes it harder to debug and maintain.

It's harder to debug because there are now three places where an error could occur: the manager, the factory, and the flyweight. Before, there was only a single object to worry about. It is also very tricky to track down data problems because it isn't always clear where a particular piece of data is coming from. If a tooltip is displaying the wrong text, is that because the wrong text was passed in, or because it is a shared resource and it forgot to clear out the text from its last use? These types of errors can be costly.

Maintenance can also be more difficult because of this optimization. Instead of having a clean architecture of objects encapsulating data, you now have a fragmented mess with data being stored in at least two places. It is important to document why a particular piece of data is intrinsic or extrinsic, as such a distinction may be lost on those who maintain your code after you.

These drawbacks are not deal breakers; they simply mean that this optimization should only be done when needed. Trade-offs must always be made between run-time efficiency and maintainability, but such trade-offs are the essence of engineering. In this case, if you are unsure whether a flyweight is needed, it probably isn't. The flyweight pattern is for situations where system resources are almost entirely utilized, and where it is obvious that some sort of optimization must be done. It is then that the benefits outweigh the costs.

Summary

In this chapter we discussed the structure, usage, and benefits of the flyweight pattern. It is solely an optimization pattern, used to improve performance and make your code more efficient, especially in its use of memory. It is implemented by taking an existing class and stripping it of all data that can be stored externally. Each unique instance of this class then becomes a resource shared among many locations. A single flyweight object takes the place of many of the original objects.

For the flyweight object to be shared like this, several new objects must be added. A factory object is needed to control how the class gets instantiated and to limit the number of instances created to the absolute minimum. It should also store previously created instances, to reuse them if a similar object is needed later. A manager object is needed to store the extrinsic data and pass it in to the flyweight's methods. In this manner, the original function of the class can be preserved while greatly reducing the number of copies needed.

When used properly, the flyweight pattern can improve performance and reduce needed resources significantly. When used improperly, however, it can make your code more complicated, harder to debug, and harder to maintain, with few performance benefits to make up for it. Before using this pattern, ensure that your program meets the required conditions and that the performance gains will outweigh the code complexity costs.

This pattern is especially useful to JavaScript programmers because it can be used to reduce the number of memory-intensive DOM elements that you need to manipulate on a page. By using it in conjunction with organizational patterns, such as composites, it is possible to create complex, full-featured web applications that still run smoothly in any modern JavaScript environment.

