Chapter 28: Method Chaining

Section 28.1: Chainable object design and chaining

Chaining and Chainable is a design methodology used to design object behaviors so that calls to object functions return references to self, or another object, providing access to additional function calls allowing the calling statement to chain together many calls without the need to reference the variable holding the object/s.

Objects that can be chained are said to be chainable. If you call an object chainable, you should ensure that all returned objects / primitives are of the correct type. It only takes one time for your chainable object to not return the correct reference (easy to forget to add `return this`) and the person using your API will lose trust and avoid chaining. Chainable objects should be all or nothing (not a chainable object even if parts are). An object should not be called chainable if only some of its functions are.

Object designed to be chainable

```
function Vec(x = 0, y = 0){
    this.x = x;
    this.y = y;
    // the new keyword implicitly implies the return type
    // as this and thus is chainable by default.
}
Vec.prototype = {
    add : function(vec){
        this.x += vec.x;
        this.y += vec.y;
        return this; // return reference to self to allow chaining of function calls
    },
    scale : function(val){
        this.x *= val;
        this.y *= val;
        return this; //  return reference to self to allow chaining of function calls
    },
    log :function(val){
        console.log(this.x + ' : ' + this.y);
        return this;
    },
    clone : function(){
        return new Vec(this.x,this.y);
    }
}
```

Chaining example

```
var vec = new Vec();
vec.add({x:10,y:10})
    .add({x:10,y:10})
    .log()              // console output "20 : 20"
    .add({x:10,y:10})
    .scale(1/30)
    .log()              // console output "1 : 1"
    .clone()            // returns a new instance of the object
    .scale(2)           // from which you can continue chaining
    .log()
```

Don't create ambiguity in the return type

Not all function calls return a useful chainable type, nor do they always return a reference to self. This is where common sense use of naming is important. In the above example the function call `.clone()` is unambiguous. Other

examples are `.toString()` implies a string is returned.

An example of an ambiguous function name in a chainable object.

```
// line object represents a line
line.rotate(1)
   .vec();  // ambiguous you don't need to be looking up docs while writing.

line.rotate(1)
   .asVec()    // unambiguous implies the return type is the line as a vec (vector)
   .add({x:10,y:10)
// toVec is just as good as long as the programmer can use the naming
// to infer the return type
```

Syntax convention

There is no formal usage syntax when chaining. The convention is to either chain the calls on a single line if short or to chain on the new line indented one tab from the referenced object with the dot on the new line. Use of the semicolon is optional but does help by clearly denoting the end of the chain.

```
vec.scale(2).add({x:2,y:2}).log();  // for short chains

vec.scale(2)     // or alternate syntax
    .add({x:2,y:2})
    .log();  // semicolon makes it clear the chain ends here

// and sometimes though not necessary
vec.scale(2)
    .add({x:2,y:2})
    .clone()    // clone adds a new reference to the chain
         .log(); // indenting to signify the new reference

// for chains in chains
vec.scale(2)
    .add({x:2,y:2})
    .add(vec1.add({x:2,y:2})  // a chain as an argument
         .add({x:2,y:2})      // is indented
         .scale(2))
    .log();

// or sometimes
vec.scale(2)
    .add({x:2,y:2})
    .add(vec1.add({x:2,y:2})  // a chain as an argument
         .add({x:2,y:2})      // is indented
         .scale(2)
    ).log();   // the argument list is closed on the new line
```

A bad syntax

```
 vec          // new line before the first function call
    .scale()  // can make it unclear what the intention is
    .log();

 vec.          // the dot on the end of the line
    scale(2).  // is very difficult to see in a mass of code
    scale(1/2); // and will likely frustrate as can easily be missed
                // when trying to locate bugs
```

Left hand side of assignment

When you assign the results of a chain the last returning call or object reference is assigned.

```
var vec2 = vec.scale(2)
               .add(x:1,y:10)
               .clone();   // the last returned result is assigned
                           // vec2 is a clone of vec after the scale and add
```

In the above example `vec2` is assigned the value returned from the last call in the chain. In this case, that would be a copy of `vec` after the scale and add.

Summary

The advantage of changing is clearer more maintainable code. Some people prefer it and will make chainable a requirement when selecting an API. There is also a performance benefit as it allows you to avoid having to create variables to hold interim results. With the last word being that chainable objects can be used in a conventional way as well so you don't enforce chaining by making an object chainable.

Section 28.2: Method Chaining

Method chaining is a programming strategy that simplifies your code and beautifies it. Method chaining is done by ensuring that each method on an object returns the entire object, instead of returning a single element of that object. For example:

```
function Door() {
    this.height = '';
    this.width = '';
    this.status = 'closed';
}

Door.prototype.open = function() {
    this.status = 'opened';
    return this;
}

Door.prototype.close = function() {
    this.status = 'closed';
    return this;
}
Door.prototype.setParams = function(width,height) {
    this.width = width;
    this.height = height;
    return this;
}

Door.prototype.doorStatus = function() {
    console.log('The',this.width,'x',this.height,'Door is',this.status);
    return this;
}

var smallDoor = new Door();
smallDoor.setParams(20,100).open().doorStatus().close().doorStatus();
```

Note that each method in `Door.prototype` returns `this`, which refers to the entire instance of that `Door` object.