JavaScript Spessore

A Thick Shot of Objects, Metaobjects, & Protocols

by Reginald "raganwald" Braithwaite

JavaScript Spessore

A thick shot of objects, metaobjects, & protocols

raganwald

Also By raganwald

Kestrels, Quirky Birds, and Hopeless Egocentricity

What I've Learned From Failure

How to Do What You Love & Earn What You're Worth as a Programmer

CoffeeScript Ristretto

JavaScript Allongé

Contents

Metaobjects

 [1]

In computer science, a metaobject is an object that manipulates, creates, describes, or implements other objects (including itself). The object that the metaobject is about is called the base object. Some information that a metaobject might store is the base object's type, interface, class, methods, attributes, parse tree, etc.–Wikipedia[2]

[1] Krups Machines (c) 2010 Shadow Becomes White, some rights reserved

[2] https://en.wikipedia.org/wiki/Metaobject

Templates and Mixins

The simplest possible metaobject in JavaScript is a *mixin*. Consider our naïve object:

```
var sam = {
  firstName: 'Sam',
  lastName: 'Lowry',
  fullName: function () {
    return this.firstName + " " + this.lastName;
  },
  rename: function (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
}
```

We can separate its domain properties from its behaviour:

```
var sam = {
  firstName: 'Sam',
  lastName: 'Lowry'
};

var person = {
  fullName: function () {
    return this.firstName + " " + this.lastName;
  },
  rename: function (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
};
```

And use `extend` to mix the behaviour in:

```
extend(sam, person);

sam.rename
  //=> [Function]
```

This allows us to separate the behaviour from the properties in our code. If we want to use the same behaviour with another object, we can do that:

```
var peck = {
  firstName: 'Sam',
  lastName: 'Peckinpah'
};

extend(peck, person);
```

Our `person` object is a *template*, it provides some functionality to be mixed into an object with a function like `extend`. Using templates does not require copying entire functions around, each object gets references to the functions in the template.

Things get even better: You can use more than one template with the same object:

```
var hasCareer = {
  career: function () {
    return this.chosenCareer;
  },
  setCareer: function (career) {
    this.chosenCareer = career;
    return this;
  }
};

extend(peck, hasCareer);
peck.setCareer('Director');
```

We say that there is a *many-to-many* relationship between objects and templates.

scope and coupling

Consider a design that has four kinds of templates, we'll call them `A`, `B`, `C`, and `D`. Objects in the system might mix in one, two, three, or all four templates. There are fifteen such "kinds" of objects, those that mix in `A`, `B`, `AB`, `C`, `AC`, `BC`, `ABC`, `D`, `AD`, `BD`, `ABD`, `CD`, `ACD`, `BCD`, and `ABCD`.

When you make a change to and one template, say `A`, you have to consider how that change will affect each of the eight kinds of objects that mixes `A` in. In only one of those, `A`, do you just consider `A`'s behaviour by itself. In `AB`, `ABC`, `ABD`, and `ABCD`, you have to consider how changes to `A` may interact with `B`, because they both share access to each object's private state. Same for `A` and `C`, and `A` and `D`, of course.

By itself this is not completely revelatory: When objects interact with each other in the code, there are going to be dependencies between them, and you have to manage those dependencies.

Encapsulation solves this problem by strictly limiting the scope of interaction between objects. If object `a` invokes a method `x()` on object `b`, we know that the scope of interaction between `a` and `b` is strictly limited to the method `x()`. We also know that any change in state it may create is strictly limited to the object `b`, because `x()` cannot reach back and touch `a`'s private state.

(There is some simplification going on here as we are ignoring parameters and/or the possibility that `a` is part of `b`'s private state)

However, two methods `x()` and `y()` on the same object are tightly coupled by default, because they both interact with all of the object's private state. When we write an object like this:

```
var counter = {
  _value: 0,
  value: function () {
    return this._value;
  },
  increment: function () {
    ++this._value;
    return this;
  },
  decrement: function () {
    --this._value;
    return this;
  }
}
```

We fully understand that `value()`, `increment()`, and `decrement()` are coupled, and they are all together in our code next to each other.

Whereas, if we write:

```
function isanIncrementor (object) {
  object.increment = function () {
    ++this._value;
    return this;
  };
  return object;
}

// ...hundres of lines of code...

function isaDecrementor (object) {
  object.decrement = function () {
    --this._value;
    return this;
  };
  return object;
}
```

Our two templates are tightly coupled to each other, but not obviously so. They just 'happen' to use the same property. And they might never be both mixed into the same object. Or perhaps they might. Who knows?

The technical term for templates referring to an object's private properties is open recursion[3]. It is powerful and flexible, in exactly the same sense that having objects refer to each other's internal properties is powerful and flexible.

And just as objects can encapsulate their own private state, so can templates.

templates with private properties

Let's revisit our `hasCareer` template:

```
var hasCareer = {
  career: function () {
    return this.chosenCareer;
  },
  setCareer: function (career) {
    this.chosenCareer = career;
    return this;
  }
};
```

[3] https://en.wikipedia.org/wiki/Open_recursion#Open_recursion

`hasCareer` stores its private state in the object's `chosenCareer` property. As we've seen, that introduces coupling if any other method touches `chosenCareer`. What we'd like to do is make `chosenCareer` private. Specifically:

1. We wish to store a copy of `chosenCareer` for each object that uses the `hasCareer` template. Mark Twain is a writer, Sam Peckinpah is a director.
2. `chosenCareer` must not be a property of each person object, because we don't want other methods accessing it and becoming coupled.

We have a few options. The very simplest, and most "native" to JavaScript, is to use a closure.

privacy through closures

We'll write our own [functional mixin][fm]:

```
function HasPrivateCareer (obj) {
  var chosenCareer;

  obj.career = function () {
    return chosenCareer;
  };
  obj.setCareer = function (career) {
    chosenCareer = career;
    return this;
  };
  return obj;
}

HasPrivateCareer(peck);
```

`chosenCareer` is a variable within the scope of the `hasCareer`, so the `career` and `setCareer` methods can both access it through lexical scope, but no other method can or ever will.

This approach works well for simple cases. It only works for named variables. We can't, for example, write a function that iterates through all of the private properties of this kind of functional mixin, because they aren't properties, they're variables. In the end, we have privacy, but we achieve it by not using properties at all.

privacy through objects

Another way to achieve privacy in templates is to write them as methods that operate on `this`, but sneakily make `this` refer to a different object. Let's revisit our `extend` function:

```
function extendPrivately (consumer, template) {
  var methodName,
      privateProperty = Object.create(null);

  for (methodName in template) {
    if (template.hasOwnProperty(methodName)) {
      consumer[kemethodNamey] = template[methodName].bind(privateProperty);
    };
  };
  return consumer;
};
```

We don't need to embed variables and methods in our function, it creates one private variable (`privateProperty`), and then uses `.bind` to ensure that each method is bound to that variable instead of to whatever object is being extended with the template.

Now we can extend any object with any template, 'privately:'

```
extendPrivately(twain, hasCareer);
twain.setCareer('Author');
twain.career()
  //=> 'Author'
```

Has it modified `twain`'s properties?

```
twain.chosenCareer
  //=> undefined
```

No. twain has `.setCareer` and `.career` methods, but `.chosencareer` is a property of an object created when `twain` was privately extended, then bound to each method using `.bind`[4].

The advantage of this approach over closures is that the template and the mechanism for mixing it in are separate: You just write the template's methods, you don't have to carefully ensure that they access private state through variables in a closure.

another way to achieve privacy through objects

In our scheme above, we used `.bind` to create methods bound to a private object before mixing references to them into our object. There is another way to do it:

[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

```
function forward (consumer, methods, to) {
    for (methodName in methods) {
      if (to.hasOwnProperty(methodName)) {
        consumer[methodName] = function () {
          return to[methodName].apply(to, arguments);
        }
      };
    }
  });

  return consumer;
};
```

This function *forwards* methods to another object. Any other object, it could be a metaobject specifically designed to define behaviour, or it could be a domain object that has other responsibilities.

Dispensing with a lot of mixins, here is a very simple example example. We start with some kind of investment portfolio object that has a `netWorth` method:

```
var portfolio = {
  investments: [],
  addInvestment: function (investment) {
    investments.push(investment);
    return this;
  },
  netWorth: function () {
    return investments.reduce(
      function (acc, investment) {
        return acc + investment.value;
      },
      0
    );
  }
};
```

And next we create an investor who has this portfolio of investments:

```
var investor = {
  //...
  investments: portfolio
}
```

What if we want to make investments and to know an investor's net worth?

```
forward(investor, ['addInvestment', 'netWorth'], investor.portfolio);
```

We're saying "Forward all requests for `addInvestment` and `netWorth` to the object representing the investor's portfolio property."

forwarding

Forwarding is a relationship between a consumer and a provider object. They may be peers. The provider may be contained by the consumer. Or perhaps the provider is a metaobject.

When forwarding, the provider object has its own state. There is no special binding of function contexts, instead the consumer object has its own methods that forward to the provider and return the result. Our `forward` function above handles all of that, iterating over the provider's properties and making forwarding methods in the consumer.

The key idea is that when forwarding, the provider object handles each method *in its own context*. This is very similar to the effect of our solution with `.bind` above, but not identical.

Because there is a forwarding method in the consumer object and a handling method in the provider, the two can be varied independently. Here's a snippet of our `forward` function from above:

```
consumer[methodName] = function () {
  return to[methodName].apply(to, arguments);
}
```

Each forwarding function invokes the method in the provider *by name*. So we can do this:

```
portfolio.netWorth = function () {
  return "I'm actually bankrupt!";
}
```

We're overwriting the method in the `portfolio` object, but not the forwarding function. So now, our `investor` object will forward invocations of `netWorth` to the new function, not the original. This is not how out `.bind` system worked above.

That makes sense from a "metaphor" perspective. With our `extendPrivately` function above, we are creating an object as a way of making private state, but we don't think of it as really being a first-class entity unto itself. We're mixing those specific methods into a consumer.

Another way to say this is that mixing in is "early bound," while forwarding is "late bound:" We'll look up the method when it's invoked.

summarizing what we know so far

So now we have three things: Mixing in a template; mixing in a template with private state for its methods ("Private Mixin"); and forwarding to a first-class object. And we've talked all around two questions:

1. Is the mixed-in method being early-bound? Or late-bound?
2. When a method is invoked on a receiving object, is it evaluated in the receiver's context? Or in the metaobject's state's context?

If we make a little table, each of those three things gets its own spot:

	Early-bound	***Late-bound***
Receiver's context	Mixin	
Metaobject's context	Private Mixin	Forwarding

So... What goes in the missing spot? What is late-bound, but evaluated in the receiver's context?

```
var careerHelper = extend({}, hasCareer);
forward(twain, ['career', 'setCareer'], careerHelper);
```

This is fantastic in one way, but why should the code that extends `twain` with `hasCareer` know about private extension? Why should it know whether `hasCareer` was carefully written to be completely self-contained and not use any properties that other mixins may read and write?

We can solve this by coupling the template to the code that performs the extension. For example an [allong.es] combinator will do the trick:

```
var rewuire = require('allong.es');
var callRight = allong.es.callRight;

var hasCareerMixin = callRight(extendPrivately, hasCareer);

hasCareerMixin(twain);
```

We could also write things out the long way with a closure:

```
function makePrivateMixin (template) {
  return function (consumer) {
    var key,
        privateProperty = Object.create(null);

    for (key in template) {
      if (template.hasOwnProperty(key)) {
        consumer[key] = template[key].bind(privateProperty);
      };
    };
    return consumer;
  };
}

var hasCareerMixin = makePrivateMixin(hasCareerMixin);

hasCareerMixin(twain);
```

A combination of template and the function that does the extending is called a functional mixin[5]. Functional mixins allow us to encapsulate the mechanism of extending an object. Objects can also be mxins, when they provide a method for mixing a template into an entity: The key idea of a "mixin" is that it bundles both the mxing functionality and the template to be mixed in.

scope and coupling

In Templates and Mixins, we saw how we can separate an object's properties form its behaviour in our code, and how to bundle the behaviour into mixins. We saw that you can have a 1-1, a 1-many, a many-1, or even many-many relationship between objects and mixins.

Our simplest mixin copied references to functions into an object. These became the object's methods, and used `this` to access the object's private state. We also saw a pattern where a mixin could bind copies of the functions to a separate object, thus guaranteeing that there would be a strict separation between the mixed-in methods and the object's private state.

This last point matters greatly. The principle value cherished by object-orientated design is *encapsulation*: Objects communicate with each other by invoking methods, not by directly manipulating each other's properties. Although JavaScript does not prevent objects from manipulating each other's properties, it is quite easy to refrain from doing so by convention.

Consider a design that has four kinds of mixins, we'll call them `A`, `B`, `C`, and `D`. Objects in the system might mix in one, two, three, or all four mixins. There are fifteen such "kinds" of objects, those that mix in `A`, `B`, `AB`, `C`, `AC`, `BC`, `ABC`, `D`, `AD`, `BD`, `ABD`, `CD`, `ACD`, `BCD`, and `ABCD`.

[5] https://javascriptweblog.wordpress.com/2011/05/31/a-fresh-look-at-javascript-mixins/

When you make a change to and one mixin, say `A`, you have to consider how that change will affect each of the eight kinds of objects that mixes `A` in. In only one of those, `A`, do you just consider `A`'s behaviour by itself. In `AB`, `ABC`, `ABD`, and `ABCD`, you have to consider how changes to `A` may interact with `B`, because they both share access to each object's private state. Same for `A` and `C`, and `A` and `D`, of course.

By itself this is not completely revelatory: When objects interact with each other in the code, there are going to be dependencies between them, and you have to manage those dependencies.

Encapsulation solves this problem by strictly limiting the scope of interaction between objects. If object `a` invokes a method `x()` on object `b`, we know that the scope of interaction between `a` and `b` is strictly limited to the method `x()`. We also know that any change in state it may create is strictly limited to the object `b`, because `x()` cannot reach back and touch `a`'s private state.

(There is some simplification going on here as we are ignoring parameters and/or the possibility that `a` is part of `b`'s private state)

However, two methods `x()` and `y()` on the same object are tightly coupled by default, because they both interact with all of the object's private state. When we write an object like this:

```
var counter = {
  _value: 0,
  value: function () {
    return this._value;
  },
  increment: function () {
    ++this._value;
    return this;
  },
  decrement: function () {
    --this._value;
    return this;
  }
}
```

We fully understand that `value()`, `increment()`, and `decrement()` are coupled, and they are all together in our code next to each other.

Whereas, if we write:

```
function isanIncrementor (object) {
  object.increment = function () {
    ++this._value;
    return this;
  };
  return object;
}

// ...hundres of lines of code...

function isaDecrementor (object) {
  object.decrement = function () {
    --this._value;
    return this;
  };
  return object;
}
```

Our two mixins are tightly coupled to each other, but not obviously so. They just ‘happen’ to use the same property. And they might never be both mixed into the same object. Or perhaps they might. Who knows?

WhThe whole point of objects is to encapsulate private data so that