Expanding JavaScript's Metaobject Protocol

A Project Report

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Computer Science

By

Tom Austin

December 2007

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Cay Horstmann

Dr. Mark Stamp

Dr. Chris Tseng

APPROVED FOR THE UNIVERSITY

ABSTRACT

EXPANDING JAVASCRIPT'S METAOBJECT PROTOCOL

by Tom Ausin

A metaobject protocol (MOP) can add a great deal of flexibility to a language. Because of JavaScript's prototype-based design and the small number of language constructs, it is possible to create a powerful MOP through relatively minimal changes to the language. In this project, I explored JavaScript and Ruby's existing metaprogramming features. I also created JOMP, the JavaScript One-metaclass Metaobject Protocol, which gives the language much of the same power that Ruby has. Finally, I built a web development framework with JSF and my modified version of Rhino JavaScript.

TABLE OF CONTENTS

TABLE OF FIGURES

1 Introduction

JavaScript has been a much maligned programming language. Browser incompatibilities, poor implementations, and some superficial flaws in its design have led to numerous headaches for developers, and for a long time, it was seen as an evil to be avoided.

All of this belies the fact that JavaScript is a very powerful language. It has support for closures, functional programming, and metaprogramming. In fact, it offers many of the same features that have made Ruby popular in recent years.

More importantly, JavaScript might be a better scripting language choice for Java programmers. Much of JavaScript's syntax and conventions follows those of Java. Furthermore, it boasts a strong, robust JVM implementation in Netscape/Mozilla's Rhino.

However, JavaScript has only a somewhat limited metaobject protocol (MOP). Expanding this could be a powerful addition to the language. This might also help to make JavaScript a viable server-side language. Ruby on Rails makes extensive use of some of these metaprogramming techniques, particularly in its ActiveRecord object-relational tool.

Metaprogramming and Metaobject Protocols are so closely tied together that I will slip back and forth between them throughout this paper. However, it is worthwhile to point out the differences between these two concepts.

Metaprogramming, simply put, is the writing of programs that can write and modify other programs. A metaobject protocol is a refinement of metaprogramming focused on objects within these languages. The authors of [10] use this definition:

> Metaobject protocols are interfaces to the language that give users the ability to incrementally modify the language's behavior and implementation, as well as the ability to write programs within the language.

In other words, a metaobject protocol allows us to modify the way that the constructs of the language behave. The Common Lisp Object System (CLOS) is the most famous example of a metaobject protocol, and is often cited as the archetype for these systems in general.

Metaobject protocols have numerous applications, including persistence [12,14], pre/post conditions [16], tool support [5], and security [23], among others. Although CLOS is the most renowned metaobject protocol, other systems exist for different languages. Smalltalk, Ruby, and Groovy all include at least partial metaobject protocols, and several models have been proposed for Java [18].

Traditionally, metaobject protocol research has been focused on class based

object-oriented systems. While class-based design is the more common approach, it is not the only one.

JavaScript instead relies on prototypes. Prototype-based object systems instead define a prototype object. New objects are created by cloning the prototype. This is an inherently more flexible system. It is easy to modify the behavior of a single object or a whole group of objects at runtime. In contrast, this is something that most class-based object-oriented languages cannot do. Interestingly, Ruby does have some metaprogramming features than can achieve some of the functionality usually reserved for languages with prototype-based object systems.

2 Ruby

Ruby has gained fame as a well designed, flexible, and powerful scripting language. It is usually described as a combination of Smalltalk and Perl, or Java and Perl for those without Smalltalk experience. The creator of Ruby is Yukihiro Matsumoto. In his own description of Ruby he attributes much of the design to Lisp as well [13]:

> Ruby is a language designed in the following steps:
> • take a simple lisp language (like one prior to CL).
> • remove macros, s-expression.
> • add simple object system (much simpler than CLOS).
> • add blocks, inspired by higher order functions.
> • add methods found in Smalltalk.
> • add functionality found in Perl (in OO way).

While Ruby and Lisp have very little superficial resemblance to one another, some of Ruby's features do illustrate the influence. One example is implicit returns; in Ruby, every statement is an expression. The return statement still exists, but with the exception of early returns, its use is mostly a matter of taste.

Ruby's alleged similarity to Lisp has been a highly contentious issue. Two blog posts in particular managed to stir up a heated debate: Eric Kidd's “Why Ruby is an acceptable LISP” and Steve Yegge's follow up “Lisp is not an acceptable Lisp”. The central point of both articles was that Ruby has much of the same flexibility and is much more practical for daily programming tasks. The comments on these articles ranged greatly in their opinions. Steve Yegge himself commented on this [21]:

> [Eric Kidd's article] got approximately 6.02e23 comments, ranging from "I agree!" through "I hate you!" to "I bred them together to create a monster!" Any time the comment thread becomes huge enough to exhibit emergent behavior, up to and including spawning new species of monsters, you know you've touched a nerve.

Regardless of Ruby's background, it has established a reputation as a cleanly designed and user-friendly scripting language. While it is not without its critics, its popularity is clearly on the rise.

In this section I will highlight some specific features of Ruby's design.

2.1 Object-oriented Design

In Ruby, everything is an object. Unlike Java (and JavaScript for that matter), there is no split between primitives and objects. As a result, `1.to_s()` is a valid statement. This leads to a simpler model, since programmers do not have to worry about this dichotomy between primitives and objects.

Ruby, like most object-oriented languages, uses a class-based system. It only supports single inheritance, but has the concept of "mix-ins". Mix-ins are modules that can be included in other classes in order to add functionality. Comparable and Enumerable are two examples of this. These serve in much the same role as interfaces do in Java, with the obvious benefit that they add actual functionality, instead of just obligations. (They do add in obligations as well -- the added methods typically make use of other methods that must be defined in the class. For example, Comparable requires that the <=> operator has been defined).

One notable distinction of Ruby's class system is that all classes are open. While this seems rife with possibilities for abuse by creative programmers, it does give a great degree of flexibility. Here is an example adding the car/cdr functions from Lisp to Ruby Arrays:

```
class Array
  # Returns the head element
  def car
    first
  end
  # Returns the tail
  def cdr
    slice(1,length)
  end
  def to_s
    s = "[ " + car.to_s
    self.cdr.each do |elem|
      s += ", " + elem.to_s
    end
    s += " ]"
  end
end
list = [1, 2, 3, 4]
puts list.car # prints 1
puts list.cdr.to_s # prints [ 2, 3, 4 ]
```

I will leave it to the reader to decide whether this is an example of why classes should be open or should not be open.

Both mix-ins and the open nature of Ruby's classes are important for metaprogramming,

so we will revisit these again later.

2.2 Type System

Ruby is dynamically typed, but not weakly typed. Although programmers do not need to specify the type of a new object, they may be required to convert it before some operations. For instance, here is an attempt to mix a String and an Integer in Ruby:

```
irb(main):002:0> "32" + 1
TypeError: can't convert Fixnum into String
        from (irb):2:in `+'
        from (irb):2
```

Instead, the type conversion must be manually specified. Either way will work:

```
irb(main):003:0> "32".to_i + 1
=> 33
irb(main):004:0> "32" + 1.to_s
=> "321"
```

In contrast, here is Rhino JavaScript:

```
js> 32 + "1"
321
js> "32" + 1
321
```

2.3 Ruby on Rails

It has been argued that every new language needs a popular application to bring it to the world's attention [19]. For Ruby, this has been the web development framework "Ruby on Rails". Rails has built-in facilities for testing, a clean division of the model/view/controller pieces, and a friendly object-relational tool named ActiveRecord. Rails advocates claim it offers a great boost in developer productivity.

A major axiom of Ruby on Rails is "Don't Repeat Yourself", often simply referred to as the DRY principle. To achieve this, Rails makes heavy use of default settings. The philosophy of "convention over configuration" means that there is very little configuration in a typical Rails application. While Rails does provide the ability to override the defaults, this is generally done only for legacy applications.

ActiveRecord is arguably the core to Rails. It greatly eases interacting with the database, which is a key part of many web applications. It also makes use of "convention over configuration" more than any other single piece of the framework.

Here are two examples of ActiveRecord classes. The names of the database tables, the field to uniquely identify each record, and the foreign key to relate the objects is all determined by default values:

```
# In album.rb
class Album < ActiveRecord::Base
  belongs_to :artist
  has_many :songs
end
# In artist.rb
class Artist < ActiveRecord::Base
  has_many :albums
end
```

Setters and getters are added automatically to the language. As a result, the programmer could then write a script like the following:

```
mark_growden = Artist.new
mark_growden.name = "Mark Growden"
live_at_the_odeon = Album.new
live_at_the_odeon.artist = mark_growden
live_at_the_odeon.title = "Live at the Odeon"
live_at_the_odeon.save()
```

This would save both objects into the database, since ActiveRecord is aware of their relationship.

2.4 Metaprogramming

Ruby has many powerful tools for metaprogramming. Many of these also exist in JavaScript; some do not. These will be discussed in more detail later. The important point to note here is that Ruby's metaprogramming features are a key part of Ruby on Rails and ActiveRecord. Eric Kidd has argued that these offer nearly as much power as Lisp's macros do [11]:

> The real test of any macro-like functionality is how often it gets used to build mini-languages. And Ruby scores well here: In addition to Rails, there's Rake (for writing Makefiles), Needle (for connecting components), OptionParser (for parsing command-line options), DL (for talking to C APIs), and countless others. Ruby programmers write everything in Ruby.

3 JavaScript

JavaScript is a study in contrasts. It has many ugly, superficial quirks. At the same time, it has a surprisingly elegant core design. On the surface, it has a syntax that seems to be a deliberate clone of Java, but its prototype-based design and its first-class functions are alien concepts to the Java world. It has been regarded as a toy language, and yet it has powered many recent, beloved AJAX applications.

Douglas Crockford offers one of the most concise descriptions [4]:

> JavaScript is a sloppy language, but inside it there is an elegant, better language.

3.1 Rhino

Netscape/Mozilla's Rhino is one of the oldest JVM scripting languages. In addition to adding in tools to script Java, it also includes a number of additional functions that make up for shortcomings in the language's basic design.

As a result, developers have begun to bring JavaScript outside of the browser. Two notable applications that use Rhino are HttpUnit [9] and Phobos [15]. HttpUnit is a tool that can be combined with JUnit to facilitate testing page flow for web applications. Phobos is a Rails-inspired web development framework.

Furthermore, Sun and Google have contributed to the growth of JavaScript on the JVM. A version of Rhino is now included in Java 6, and Google is developing a "Rhino on Rails" web development framework [20].

3.2 Prototype-based Object Design

JavaScript is the most widely used prototype-based programming language. While this is an unfamiliar model to most programmers, it is a surprisingly flexible and powerful one. Also, every JavaScript object is a collection of properties. The combination of these two characteristics means that there are very few points that need to be considered when designing a metaobject protocol.

JavaScript borrowed much of it core design philosophy from Self. The designers of Self discussed the advantages of prototype-based object-oriented languages over the more traditional class-based approach [22]:

> Class-based systems are designed for situations where there are many objects with the same behavior. There is no linguistic support for an object to possess its own unique behavior, and it is awkward to create a class that is guaranteed to have only one instance. SELF [because of its prototype-based system] suffers from neither of these disadvantages.

3.3 First-class Functions

JavaScript functions are first class citizens. They can be passed as arguments, returned from other functions, or stored as properties. Functions are also closures. David Flanagan discusses this in his authoritative reference book on JavaScript [8]:

> The fact that JavaScript allows nested functions, allows functions to be used as data, and uses lexical scoping interact to create surprising and powerful effects.

Throughout his book, Flanagan demonstrates multiple uses for this feature of the language. It can be used to create private namespaces, set breakpoints, and create unique number generators.

When Brendan Eich created JavaScript, he originally wanted to create a dialect of Scheme [3]. Though it superficially more resembles Java and C, its first class functions and simple, elegant design show these roots.

3.4 Properties

JavaScript also borrowed its handling of properties from Self. In Self, they are called 'slots' and can hold any value, including functions [22]. Partially as a result of this design, JavaScript can easily mimic many of Ruby's metaprogramming features.

However, properties are intrinsically public. This is often undesirable, and it makes it difficult to intercept calls to set or get properties. While nested functions can be used to create getters and setters for private data, this is not the JavaScript way. It breaks with the conventions of the language and loses much of the power and flexibility that JavaScript's design offers. This will be one major issue that will be addressed with the proposed extensions.

4 Metaprogramming: Ruby vs. JavaScript

This section will focus on the metaprogramming features within Ruby and the equivalent features within JavaScript. David Black's "Ruby For Rails" covers most of these features in great detail [1]. Outside of digging through the source code for Rails, this was the primary reference for this section.

4.1 Singleton Classes

Singleton classes are used to add methods or attributes to individual objects rather than to classes. Ruby's syntax allows the programmer to either define individual methods of the singleton class, or to open the singleton class and add methods or variables that way. I will show an example of the former first, since its syntax is easier to follow:

```
greeting = "Hello"
bob = "Bob"
def greeting.say_twice
  puts self
  puts self
end
greeting.say_twice # This will print "Hello" twice
bob.say_twice # This will throw a NoMethodError
```

Rails uses this technique in its DRb server setup for ActionController. (DRb stands for Distributed Ruby, which is one of the several options for storing session information). With this technique, access to the `session_hash` is synchronized. They use the alternate syntax of `class <<obj` since they are adding several methods to the class at once. Here is an excerpt:

```
session_hash.instance_eval { @mutex = Mutex.new }
class <<session_hash
  def []=(key, value)
    @mutex.synchronize do
      super(key, value)
    end
    # More methods omitted
```

```
  end
end
```

For JavaScript, this is nothing special. JavaScript's prototype-based design inherently provides the same functionality. For instance, the JavaScript equivalent of the say_twice method would be the following:

```
var greeting = new String("Hello");
var bob = new String("Bob");
greeting.sayTwice = function() {
  print(this);
  print(this);
}
greeting.sayTwice(); // This will print "Hello" twice
bob.sayTwice(); // This will throw an Exception
```

The code is not any shorter, but its syntax is arguably cleaner. Ruby's singleton classes seem like a bolted-on measure to emulate prototypes.

4.2 Eval Methods

This is one of the most powerful metaprogramming features in Ruby. It allows the execution of arbitrary strings as Ruby commands. There are 4 different eval functions:

- eval
- instance_eval
- class_eval
- module_eval

Eval is the most basic and most powerful. Also, it is the most dangerous. Probably for this reason, it does not seem to be used much in Rails.

The other three eval methods are more often used. They differ from the basic eval in that they can also accept blocks of code, meaning that they can be used with much less risk.

The main purpose for `instance_eval` is to gain access to the private members of another class. The `class_eval` and `module_eval` methods are designed to add to the functionality of a class or module and to include variables from the current scope. Together, all 3 of these serve to allow the programmer to inject functionality into another class.

JavaScript has the same basic `eval` function. The `apply` and `call` methods of Function generally fill the same role as the other versions. Because of the elegance of JavaScript's prototype design, fewer MOP tools are needed. This proves to be a recurring theme when comparing metaprogramming in these two languages.

4.3 Aliasing a Method

This is heavily used in ActiveRecord, and seems to be one of the core pieces of the

design in Rails. The 2 methods used primarily in this are `alias_method` and (to a lesser extent) `define_method`. These are used in tandem to create a wrapper around methods.

The method is aliased to a new name, and the original method name is overridden by the wrapper method. In Rails, this is often used to change the functionality of a method. For example, ActionController uses these methods to change what happens when `page.render` is called.

This is nothing exciting for JavaScript. Moving around methods is easy since they are just functions stored as properties. We will take heavy advantage of this fact when designing the new metaobject protocol for JavaScript.

4.4 Callable Objects

Proc, block, and lambda are collectively referred to as 'callable objects'. All three are variations of the same idea -- they are ways to define temporary pieces of executable code. Javascript can already create anonymous functions, so there is little that it is missing.

Ruby has `method`, which returns a reference to the named method. This is mostly needed because of the blurred line between properties and method calls in Ruby. JavaScript does not have this issue. `music.method(:play)` in Ruby would translate to just `music.play` in JavaScript.

Often used along with `method` are `bind` and `unbind`. Together, these can be used to allow method references to be moved around between objects. The need for this is unclear, and Rails seems to make little use of this feature. In fact, in his discussion on the subject, David Black suggests that if you are using this, you most likely have a problem in your design [1]:

> This is an example of a Ruby technique with a paradoxical status: It's within the real of things you should understand, as someone gaining mastery of Ruby's dynamics; but it's outside the realm of anything you should probably be doing.

JavaScript does all of this already. Its functions seem to be more powerful and flexible. They can have properties of their own (which is not true for Ruby methods), they can be passed as arguments, and they can be bound and unbound at will. Ruby's methods are close, but they are not quite as flexible, which seems to require this extra complexity to achieve the same results.

4.5 Mix-ins

As discussed before, mix-ins are used in Ruby in place of multiple inheritance. They are ways of adding a chunk of functionality to another class. JavaScript has no built in function to do this, though it is easily mimicked. In section 9.6 of his book, Flanagan provides a 6-line method to achieve this [8]. Again, the combination of properties and

first class functions provide JavaScript with the power that it needs.

4.6 Callbacks and Hooks

Ruby has several different points where a programmer can hook in to the application. They are:

- Module#method_missing
- Module#included
- Class#inherited
- Module#const_missing

Of these, const_missing is used the least. It does not seem to be particularly important. David Black suggests that it could be useful for giving default values to uninitialized constants, but why constants would need default values is a little unclear.

In contrast, method_missing is used frequently. It helps to create shortcuts and more intuitive APIs. ActiveRecord uses this to allow calls like `Employee.find_by_last_name("Austin")`. Behind the scenes, method_missing converts this to `Employee.find(:first, :last_name => "Austin")`.

We could use method_missing to extend the earlier Lisp-like additions to the Array class:

```
class Array
  # This will give more advanced list functions, like cadar or caar.
  # However, unlike in Lisp, there will be no limit to the available
  # methods.
    meth_name = method_called.to_s
    if meth_name =~ /^c(a|d)+r$/
      list = self
      meth_name.reverse.scan(/./).each do |op|
        if op == 'a'
          list = list.car
        elsif op == 'd'
          list = list.cdr
        end
      end
      return list
    else
      super(method_called, *args, &block)
    end
  end
end
list = [[0, [1, 2], 3], 4]
puts list.caadar  #prints 1
```

While method_missing can create friendlier APIs, it does not seem to offer any extra programming power in Ruby. However, when combined with JavaScript's prototype-based object design, it does suggest some interesting possibilities. For one, this might be a technique for creating multiple inheritance. If a method did not exist in

one prototype chain, a second prototype chain could be searched.

Method_missing has proven to be particularly popular, and it has been copied by other languages. Most importantly, the latest version of Rhino JavaScript has added a `__noSuchMethod__` function that operates exactly like method_missing, though this is not part of the ECMAScript specification. However, since property references in JavaScript are not the same as method calls, this does not offer the full power of Ruby's method_missing.

The included and inherited methods seem to be the core of Ruby metaprogramming, at least for how it is applied in Rails. This is used heavily in ActiveRecord and even more so in ActionController. Here is an example from the base ActionController class:

```
module Layout
  def self.included(base)
    base.extend(ClassMethods)
    base.class_eval do
      alias_method :render_with_no_layout, :render
      alias_method :render, :render_with_a_layout
      class << self
        alias_method :inherited_without_layout, :inherited
        alias_method :inherited, :inherited_with_layout
      end
    end
  end
  # ... Rest omitted
```

When the Layout module is included, it rewires the render method of the host object so that it will use the layout. It also changes the behavior of the inherited method.

JavaScript does not seem able to compete here. It has no real equivalent to the included/inherited methods, and no standard equivalent to method_missing. Fortunately, JavaScript's design makes it easy to cover all of these by intercepting calls to the object.

Setting new properties in JavaScript covers both inclusion of other modules and inheritance (via the prototype chains). By intercepting the getting of properties from an object, method_missing and const_missing could both be mimicked as well. If a mechanism can be created for intercepting the setting and getting of properties, JavaScript's metaprogramming features can become every bit as powerful as those of Ruby.

5 JavaScript Metaobject Protocol Proposal

JavaScript's power can be greatly increased by adding callbacks and hooks to the language. Fortunately, since JavaScript makes heavy use of properties, we can add most of our hooks at a single point.

Because JavaScript has no classes, we really only need to consider objects and functions. In contrast, Ruby has Object, Method, Class, and Module metaclasses to deal with among others.

As it turns out, we can add the additional power we need with Object alone. All functions are properties of some object. Therefore, we can create a wrapper function and return that whenever a function is requested. Even top-level functions are properties of the global object [6].

In this section, I will outline my proposal for a new metaobject protocol for JavaScript. I have named it JOMP – the JavaScript One-metaclass Metaobject Protocol.

5.1 Mix-ins

JavaScript can mimic this already, though it is not built in to the language. We can fix this by adding these methods to Object:

- `addMixIn(mixIn)`
- `mixedIn(recipient)` – not automatically added, but reserved by convention.

The addMixIn method is just a modification of David Flanagan's version. It is done in a more object-oriented manner and with a callback mechanism added:

```
Object.prototype.addMixIn = function(mixIn) {
   var from = mixIn;
   var to = this.prototype;

   for (method in from) {
      if (from.hasOwnProperty(method)) {
         if (typeof from[method] !== "function") continue;
         if (method === "addMixIn" ||method === "mixedIn") continue;
         to[method] = from[method];
      }
   }
   // If the mix-in object has a mixedIn method, it will be called.
   //  This emulates Ruby's Module#Included callback method.
   if (mixIn.mixedIn) {
      mixIn.mixedIn(this);
   }
}
```

Whenever a mix-in is added to another module, the recipient checks the mix-in for a mixedIn() method. If it finds one, it calls that method and passes itself as the object. This also illustrates how we could track clones of a prototype, although we will need a mechanism to track their creation.

Here is an example mix-in. In this case, we are again adding car/cdr functionality to Arrays, but we are doing it as a mix-in instead:

```
function LispListMixIn() {
   this.mixedIn = function(receiver) {
      var recvMatch = receiver.toString().match(/function (.*?)\(/);
      var recvName = recvMatch ? recvMatch[1] : "primitive";
      print("Adding Lisp functionality to " + recvName);
   }
   this.car = function() {
      return this[0];
   }
   this.cdr = function() {
      return this.slice(1);
   }
}
Array.addMixIn(new LispListMixIn());

var numbers = [1,2,3];
print(numbers.cdr().car()); //This will print 2
```

5.2 The __metaobject__ Property

With JOMP, every object in the language may have a `__metaobject__` property. If this does not exist, the object will behave normally. However, if this property is specified, its methods may alter the behavior of the object.

A `__metaobject__` can specify any or all of these methods:

- has(thisObj,property)
- get(thisObj,property)
- set(thisObj,property,value)
- remove(thisObj,property)
- getIds(thisObj)
- hasInstanceOf(thisObj,instance)

The first argument of all of these methods is the object itself. The second argument for `has`, `get`, `set`, and `remove` is the name of the property. For the `set` method, the last value is the value being given to the specified property.

Each of these methods corresponds to a different action; `has` is called when testing for the existence of a property, `get` is called when attempting to retrieve the value for that property, `set` is called when attempting to set it, and `remove` is called when the delete command is used on a property.

The return value for these actions, if there is one, will be the return value for the method call. For instance, if `foo.bar` is called, the value will be the result of calling `foo.__metaobject__.get(foo,'bar')`. The other methods follow the same pattern.

The `hasInstanceOf` method works differently than the others in that it is usually part of the prototype's metaobject. This is called whenever the `instanceof` operator is used.

The first argument is the prototype and the second is the instance. So, `joe instanceof Employee` will result in a call to `Employee.__metaobject__.hasInstanceOf(joe)`, if the Employee's `__metaobject__` property contains that method.

If the `__metaobject__` property does not define any of these methods, the object's corresponding behavior will not be altered.

5.2.1 Looking Up the Metaobject in the Prototype Chain.

The `__metaobject__` does not have to be part of the object in question. It can be looked up in the prototype chain just like any other property.

This is a key point. Because of this feature, modifying the behavior of objects can be as granular as needed. A single object can be given its own behavior, or `Object.prototype.__metaobject__` can be set, in which case the behavior of every object will be changed.

5.2.2 Creating a Separate Metaobject

One unusual aspect of this design is that a separate `__metaobject__` is defined. A different and perhaps more obvious approach would have been to add `__has__`, `__get__`, `__set__`, and `__remove__` properties to the Object prototype. This is, in fact, the approach that Ruby has taken in the design of its MOP.

However, the advantage to JOMP's design is that the behavioral rules can be contained in a single object. For instance, we could create a tracingMO object that simply printed whenever any of its methods were called. Tracing an object would then simply become a matter of setting its `__metaobject__` property to tracingMO. This also allows us to more easily add logic in order to combine effects of different sets of behavioral rules. Later we will show an example of a tracing metaobject that is designed to be layered over an object's existing `__metaobject__` property.

This could still be achieved with separate methods, but it becomes more complicated. The `__metaobject__` property approach gives an easy way to contain the behavioral rules in s single package.

5.2.3 One Metaclass

One noticeable difference in the design of JOMP is that it has no real metaclasses. In most MOPs, metaclasses are the principal means of organizing the different metaobjects. It would seem odd to have metaclasses in a language without classes, but that was not the reason for the omission.

As JOMP's name indicates, we only needed a metaclass for objects. With only one construct, the concept of a metaclass is not a particularly useful one.

If JOMP were extended to add MOP features that were specific to functions, or to include support for primitives and operators, metaclasses might become necessary. However, this would probably need a metaprototype, or some other construct more fitting with the prototype design philosophy.

5.3 Applications of the JOMP

The new extensions allow JavaScript to do many things that have not been possible before. In this section, we will cover a few examples.

5.3.1 Getter and Setter Basics

In Java and other languages, you intercept properties by using a setter and getter. However, the key difference here is that we may decide to change the behavior at runtime, something that many languages cannot do easily.

For a simple example, let's create a new employee:

```
function Employee(firstName, lastName, salary) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.salary = salary;
}

var t = new Employee('Tom', 'Austin', 1000000);
print(t.firstName + " " + t.lastName + " $" + t.salary);
```

After creating this employee, we may want to prevent the salary field from being changed accidentally. To do this, we can change the rules for setting the salary property:

```
//Now we want to make salary read only
var mop = {};
mop.set = function(thisObj, prop, value) {
  if (prop == 'salary') {
    throw new Error('Warning: Salary is a read-only property');
  }
  thisObj[prop]=value;
}
t.__metaobject__ = mop;
```

After this, any attempt to change the salary will not work.

```
//This will print an error and the salary will not be changed.
try {
  t.salary = 999999;
}
catch (e) {
  print(e);
}
```

Over time, the definition for a field might change. For example, salary could include a bonus, but you might still want salary to refer to the total salary. With a change to the object's behavior, this is easily done:

```
//Change salary to use baseSalary and bonusPay
t.baseSalary = 1000000;
t.bonusPay = 500;
t.__metaobject__.get = function(thisObj, prop) {
  if (prop == 'salary') {
    return thisObj.baseSalary + thisObj.bonusPay;
  }
  else return thisObj[prop];
}
```

Although we have not used JOMP for anything greatly original so far, these examples do show how some basic changes to the language can be useful.

5.3.2 Tracing

Logging is a common use-case given for metaobject protocols. Often you would like to trace an object's behavior for troubleshooting. One common method is to insert print statements, but this clutters up the code. More importantly, it might clutter up the logs as well, making it harder for you to spot the problem.

Metaobject protocols offer a good solution to this. The code to an object can be left unchanged, but you can modify its behavior to report back detailed messages.

An important point here is that the object's behavior can be changed on the fly, so you can limit the verbose logging to only a portion of the code. Also, you can alter the behavior of only a given object or a whole group of objects just as easily.

Here is an example function that will trace an object's behavior:

```
function traceObject(o, objName) {
  var oldMo = o.__metaobject__;
  var tracingMO = {};

  // This function can be used to disable a tracing routine.
  tracingMO.stopTrace = function() {
    o.__metaobject__ = oldMo;
  }

  // Logs the getting of properties.  Functions returned
  // will print their property
  tracingMO.get = function(thisObj, prop) {
    logMessage("***Getting " + prop + " from " + objName);

    var returnVal = thisObj[prop];
    if (oldMo) returnVal = oldMo.get(thisObj, prop);
```

```
    //We will wrap functions so that we know when they are called.
    if ((typeof returnVal) == "function") {
      var wrapFunct = function() {
        var msg = "***Calling " + prop + " with args:";
        for (var i=0; i<arguments.length; i++) {
          msg += " " + arguments[i];
        }
        logMessage(msg);
        returnVal.apply(thisObj, arguments);
      }
      return wrapFunct;
    }
    else return returnVal;
  }

  // Logs the setting of properties
  tracingMO.set = function(thisObj, prop, value) {
    logMessage("***Setting " + objName + "'s " + prop
         + " to '" + value + "'");

    if (oldMo) oldMo.set(thisObj, prop, value);
    else thisObj[prop] = value;
  }
  o.__metaobject__ = tracingMO;
}
```

There are a few key points to note in this example. First of all, the original object might have its own `__metaobject__`. We do not want to lose that, so we must wrap the tracing functions around the original. Also, since the original might not have a `__metaobject__` specified, we have to consider that case as well.

We want to be able to track when a function is called and with what arguments. To do this, we can wrap the original function in a new one and return that on the fly.

This highlights a couple of the downsides to not having a `__metafunction__` property as well. First of all, constructing the new functions on the fly can be expensive. For troubleshooting, that is probably acceptable.

Another, more subtle problem is that the new function can be treated as an object. It is possible that it might be passed as an argument to another function, stored as a property for another object, etc. At that point, the function is no longer under the control of the tracing metaobject. Turning off the tracing behavior will not affect the new function.

Here is an example using the earlier function:

```
var rincewind = {};
traceObject(rincewind, "Rincewind"); //Enables tracing

rincewind.hatName = "Wizzard";
```

```
rincewind.weapon = "sock & half-brick";
rincewind.attack = function(enemyName) {
  print("Hit " + enemyName + " with " + rincewind.weapon);
}

rincewind.attack("Hell-Demon");
rincewind.weapon = "other sock & half-brick";
rincewind.attack("Nastier Hell-Demon");
```

Running this example would give very detailed logging:

```
***Setting Rincewind's hatName to 'Wizzard'
***Setting Rincewind's weapon to 'sock & half-brick'
***Setting Rincewind's attack to '
function (enemyName) {
    print("Hit " + enemyName + " with " + rincewind.weapon);
}
'
***Getting attack from Rincewind
***Calling attack with args: Hell-Demon
***Getting weapon from Rincewind
Hit Hell-Demon with sock & half-brick
***Setting Rincewind's weapon to 'other sock & half-brick'
***Getting attack from Rincewind
***Calling attack with args: Nastier Hell-Demon
***Getting weapon from Rincewind
Hit Nastier Hell-Demon with other sock & half-brick
```

However, after this, you might not care about the rest of the results. At this point, you can disable tracing:

```
rincewind.__metaobject__.stopTrace();
rincewind.weapon = "turnip";
rincewind.attack("Evil Warlord");
```

The behavior is normal for this section, and much less verbose:

```
Hit Evil Warlord with turnip
```

This code is included in RhinoFaces, the web development framework discussed in chapter 7. It provides a useful tool for monitoring the behavior of an object, and it proved invaluable for troubleshooting.

5.3.3 Security Applications

Another frequent use of MOPs is for security [23]. By intercepting the setting and getting of all properties, it becomes a very simple matter to prevent all access to an object.

By locking down an object in the constructor, the API designer can prevent developers from accidentally giving access to restricted information. We will start with a simple `Employee` example:

```
function Employee(firstName, lastName, salary) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.salary = salary;

  //This variable temporarily allows us to modify variables.
  var authorized = true;

  var mop = {};
  //This will make all properties read only
  mop.set = function(thisObj, propertyName, newVal) {
    if (authorized || (typeof newVal) == 'function') {
      thisObj[propertyName] = newVal;
    }
    else print("Sorry, " + propertyName + " is read-only.");
  }
  //This will make all properties private
  mop.get = function(thisObj, propertyName) {
    if (authorized || (typeof thisObj[propertyName])=='function') {
      return thisObj[propertyName];
    }
    print("Sorry, " + propertyName + " is private.");
    return null;
  }

  this.__metaobject__ = mop;

  //The object is now locked down
  authorized = false;
}
```

This takes advantage of the fact that JavaScript functions are closures. The `authorized` variable is private. After an employee has been created, the variable cannot be modified. No user can then inadvertently modify an employee's contents, or inadvertently display information that should be secure.

The logic of Employee could be made more complex. One easy change would be to have `lock` and `unlock` methods that would change the `authorized` variable.

5.3.4 Advanced Metaprogramming

JOMP can also be used to emulate more advanced metaprogramming techniques, like Ruby's method_missing idiom. Although it does not execute anything itself, it can create a new function and return that. Here is an example mimicking the Ruby Lisp list example:

```
Array.prototype.car = function() {
```

```
   return this[0];
}
Array.prototype.cdr = function() {
   return this.slice(1);
}

var mop = {};
mop.get = function(thisObj,propName) {
  if (propName.match(/^c(a|d)(a|d)+r$/)) {
     var list = thisObj;
     return function() {
        var chars = propName.match(/a|d/g).reverse();
        for (var i=0; i<chars.length; i++) {
           var op = chars[i];
           if (op === 'a') {
              list = list.car();
           }
           else if (op === 'd') {
              list = list.cdr();
           }
        }
        return list;
     }
  }
  else return thisObj[propName];
}

var list = [[0, [1, 2], 3], 4];
list.__metaobject__ = mop;
```

The downside of this approach compared to Ruby's method_missing or Rhino's existing __noSuchMethod__ is that it creates a new function object, which is slower. However, with a little adjustment, we could make this newly created function a method of the object, which would greatly speed future calls.

5.3.5 Multiple Inheritance

With JOMP, we can change some of the core features of JavaScript. For a good example of this, we will add multiple inheritance. To truly be multiple inheritance, we need to make the following changes:

- An object should be able to inherit properties from multiple prototype chains.
- The `instanceof` operator should return true for any of the object's parents.
- Enumerating over an object's properties should return those from all of its parents.

These changes will require modifications to the behavior of both the object and its prototype. To illustrate this, we will create some prototypes for a role-playing game.

The game will have heroes, which are under the user's control, and non-player characters (NPCs), which will be controlled by the computer. NPCs are further divided into allies and villains.

The `Hero` and `NPC` definitions do not illustrate a great deal; `Ally` and `Villain` are more central to the problem. These will both define a `move` method, but will have different implementations.

```
function Ally(name, hitpoints, experience, xpValue) {
  NPC.call(this, name, hitpoints, experience, xpValue);
}
Ally.prototype = new NPC();
Ally.prototype.move = function() {
  print(" (" + this.name + "'s action:  Help hero)");
}

function Villain(name, hitpoints, experience, xpValue) {
  NPC.call(this, name, hitpoints, experience, xpValue);
}
Villain.prototype = new NPC();
Villain.prototype.move = function() {
  print(" (" + this.name + "'s action:  Attack hero)");
}
```

However, the game could use more classes than this. For instance, some characters might be able to use magic. A Wizard definition might look like the following:

```
function Wizard() {};
Wizard.prototype.castSpell = function(spellName) {
  if (this.spells[spellName]) {
    var spell = this.spells[spellName];
    return spell();
  }
}
```

Unfortunately, we could have wizards that are heroes, villains, or allies. In Java, the solution would be to create a Wizard interface, and then to have HeroWizard, VillainWizard, and AllyWizard implementations. However, this could get increasingly complex as more roles are added, and at some point a new approach would need to be designed.

This tends to be less of an issue in most scripting languages. In JavaScript and Ruby, for instance, we could add mix-ins to include all of the extra methods we needed for an object. But there are two problems with this.

The first is that `instanceof` will not work as a means to identify an object's type. We could work around this by adding a method to the prototype or to the objects themselves, though this is not ideal.

A second problem is that the extra functions lose their association once they are mixed-in to the object. As a result, it becomes difficult to cleanly remove them. This could be a problem in some cases.

Instead, we will change the behavior of these prototypes and their instances to allow for an array of prototypes to be specified. All prototypes in the array will be treated as if they were the object's prototype.

The object's behavior must be changed to use the array for both getting the ids and looking up properties:

```
var objMop = {};
objMop.getIds = function(thisObj) {
  var ids = []
  for (var ind in thisObj) {
    ids.push(ind);
  }
  if (thisObj.__proto__ instanceof Array) {
    for (var ind in thisObj.__proto__) {
      var proto = thisObj[ind];
      if (proto) {
        for (var name in proto.prototype) {
          if (!ids[name]) ids.push(name);
        }
      }
    }
  }
  return ids;
}
objMop.get = function(thisObj,prop) {
  if (thisObj[prop]) return thisObj[prop];
  else if (thisObj.__proto__ instanceof Array) {
    for (var ind in thisObj.__proto__) {
      var proto = thisObj.__proto__[ind];
      if (proto.prototype[prop]) {
        return proto.prototype[prop];
      }
    }
  }
  return thisObj[prop];
}
```

We also need to change the behavior of the prototype definitions in order for `instanceof` to work as we would like:

```
var multiMop = {};
multiMop.hasInstanceOf = function(thisObj,instance) {
  if (instance.__proto__ instanceof Array) {
    for (var key in instance.__proto__) {
      var prot = instance.__proto__[key];
      if (prot == thisObj) return true;
```

```
    }
    return false;
  }
  //Note that instanceof can be used normally inside the method.
  else return (instance instanceof thisObj);
}
Wizard.__metaobject__ = multiMop;
Hero.__metaobject__ = multiMop;
Ally.__metaobject__ = multiMop;
Villain.__metaobject__ = multiMop;
```

These prototype definitions and the new object behavior have added multiple inheritance to JavaScript. For an example, we will show a game excerpt about Jason and the Argonauts. In his quest, Jason meets and later marries Medea. This is a case where we want a new instance that is both an Ally and Wizard. (This uses Mozilla's __proto__ property to reassign the prototype chain.)

```
var medea = new Ally("Medea", 4);
medea.__metaobject__ = objMop;
medea.__proto__ = [Ally, Wizard];
medea.spells = {
  old2new: function(ram) { print("'Look, the ram is young now!"); }
};
```

Both `medea instanceof Ally` and `medea instanceof Wizard` will be true. When `move` is called she will help Jason. However, we might want to give Jason the option of leaving Medea. We can account for this action by adding a new method to the `jason` instance.

```
jason.divorce = function(wife) {
  for (var i in wife.__proto__) {
    if (wife.__proto__[i] == Ally) wife.__proto__[i] = Villain;
  }
}
```

After `jason.divorce(medea)` is called, `medea` still refers to the same object. Her wizard abilities are unchanged, but she is now a Villain instead of an Ally. From that point on, `medea.move()` will use the Villain version of the method instead.

This is a key point, and one advantage of a prototype-based object design in general. Class-based designs are great for defining static behavior, but modifying that behavior on the fly becomes more challenging. The typical solution for this example would be to create a new instance of Medea. However, any other modifications to Medea's state could be lost without careful programming. If Medea happened to be holding the `goldenFleece` object in her inventory, it might suddenly disappear.

Prototype-based systems do not need to worry about this. The only change to Medea is her switch from Ally to Villain. Nothing else is affected.

This type of change occurs frequently in role-playing games, and this solution makes that easy to model. Being able to compartmentalize and alter behavior at will is not needed for all problems. However, when it is, prototype chains are an ideal solution. By using JOMP to create multiple inheritance, we can make this even more powerful.

6 RhinoFaces

The previous examples offer some insights into how these extensions could be useful. However, to offer a truly practical example of JOMP in action, I have built RhinoFaces. RhinoFaces is a framework built upon JavaServer Faces, but using Rhino JavaScript as the server-side language.

RhinoFaces will still work without the JavaScript extensions; however, in this case, it will lose some functionality. This will help to illustrate what improvements are directly attributable to the new metaprogramming features.

6.1 JavaServer Faces

JavaServer Faces, more often referred to as simply JSF, is a web development framework from Sun. It is focused on the view portion of the Model View Controller pattern.

JSF was built by many of the core developers of Struts, at one time the de-facto standard for Java web development. For this reason, JSF was seen as the heir-apparent to Struts.

However, several criticisms arose of the early implementations of JSF, and other frameworks have gained much ground. RhinoFaces will address a number of these issues. The principal difference will be a reliance on convention over configuration. This is the design philosophy behind Ruby on Rails, and this strategy will help to greatly simplify JSF development.

For any piece that developers prefer to leave in a more traditional Java/JSF design, they may do so. None of the additional tools or shortcuts needs to be used. They are optional extensions, and any or all may be ignored.

6.2 Reduced Configuration

Though this feature does not use JOMP, it nonetheless simplifies development greatly. Missing properties are searched for in the session's JavaScript environment. A few basic rules help determine what should be done.

When the session first starts, `application.js` is loaded. This typically specifies database properties and models, but any variable or function loaded here will be available to RhinoFaces.

JSF value expressions are assumed to be JavaScript property references. For example, `<h:outputText value='#{order.description}'/>` would look for a `description` property in `order` and display that value. Method expressions are expected to be method calls instead, so `<h:commandLink action='#{cart.remove}'>` will result in method call of `cart.remove()`. The return value of this method will be set as the value of the action. For any action, the name is assumed to correspond to a page. So, if `browse/album` is the action, it will default to the page `browse/album.faces`.

If a variable is unavailable, and the variable name matches the controller part of the URL, it will look for a JavaScript backing bean of the same name. Furthermore, if that script contains a constructor with a matching name, it will create a new instance.

For example, `cart/viewCart.faces` could be the url for customers to see the contents of their shopping cart. An excerpt of the JSP page might look like this:

```
<h2>Items in your order</h2>
  <h:dataTable value='#{cart.items}' var='album' border="0"
         cellspacing="5" cellpadding="5">
    <h:column>
      <f:facet name='header'>
      ...
```

The first time this loads, `cart` in `#{cart.items}` is not recognized. RhinoFaces then loads `cart.js` and finds this constructor:

```
function Cart() {
  this.items = new ArrayList();
  this.totalPrice = 0;
  if (flash.album) {
    var album = flash.album;
    this.items.add(album);
    this.totalPrice += Number(album.price);
  }
}
```

It then creates a new cart controller instance by executing the following code:

```
var cart = new Cart();
```

On subsequent visits to this page, the cart will already exist in the session's JavaScript environment, so no new cart will be created.

All of these defaults may be overridden in the faces-config.xml file. However, the use of defaults greatly eases the burden on the developer. This is particularly noticeable with the navigation rules. Since an action navigates by default to a page matching its name, we can remove any case where `from-outcome` is the same as `to-view-id`. Here is the configuration for the JavaQuiz example in chapter 3 of the Core JSF book [7]:

```
<faces-config>
   <navigation-rule>
      <navigation-case>
         <from-outcome>success</from-outcome>
         <to-view-id>/success.jsp</to-view-id>
         <redirect/>
      </navigation-case>
      <navigation-case>
         <from-outcome>again</from-outcome>
         <to-view-id>/again.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
         <from-outcome>failure</from-outcome>
         <to-view-id>/failure.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
         <from-outcome>done</from-outcome>
         <to-view-id>/done.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
         <from-outcome>startOver</from-outcome>
         <to-view-id>/index.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>
   <managed-bean>
      <managed-bean-name>quiz</managed-bean-name>
      <managed-bean-class>com.corejsf.QuizBean</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
   </managed-bean>
</faces-config>
```

In contrast, here is the same configuration file for the RhinoFaces version of the application:

```
<faces-config>
   <factory>
      <application-factory>
         edu.sjsu.rhinofaces.RhinoApplicationFactory
      </application-factory>
   </factory>
   <navigation-rule>
      <navigation-case>
         <from-outcome>startOver</from-outcome>
         <to-view-id>/index.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>
</faces-config>
```

Furthermore, the navigation rules can be eliminated entirely by just returning `index` as the final action of the quiz. Unless the developer wishes to override the default settings, this configuration file will never need to specify `navigation-rule`.

6.3 Flash Scope

RhinoFaces includes a "flash" object, which is another concept taken from Ruby on Rails. This is reset to an empty object after each time that a page is rendered. As a result, this is a useful way to pass information from page view to page view without worrying that it will not get cleaned out.

As an example of how this is used, here is the logout method for MobileMusic:

```
JukeBox.prototype.logout = function() {
  this.loginText = "login";
  flash.message = "Good-bye, " + this.currentUser.username + ".";
  delete this.currentUser;
  if (__GLOBAL__['cart']) cart.empty();
  return "browse/index";
}
```

Among other things, this sets a good-bye message that will be displayed by this section of the JSP page:

```
<strong><em><h:outputText value="#{flash.message}"/></em></strong>
```

However, if the page is reloaded, this message will disappear.

This feature was implemented by simply resetting the object, but another approach would have been to use JOMP. Instead of replacing the object, the flash could be set to automatically delete a property after it had been used. This would have the advantage that its properties would survive a redirect. However, it also makes the flash more complicated to use. Therefore, this approach was abandoned.

6.4 Simplified Database Access

One of the major advantages of Rails is the ease of database access. This is done through ActiveRecord, Rails' object-relational tool. RhinoFaces includes RhinoRecord, which offers many of the same benefits that ActiveRecord offers.

RhinoRecord handles all database access, which improves the security of the application. Since the web developer does not have to access the database directly, there is no risk of a SQL injection attack.

This is a fringe benefit though; the main focus of RhinoRecord is simplifying development. The base RhinoRecord achieves this by following the same conventions as ActiveRecord. Database table names are assumed to be the plural of the class (for Rails) or constructor (for RhinoFaces). The object's properties are taken directly from the database field names. The only difference in this is that RhinoRecord converts names with underscores to camel case. For example, `first_name` becomes `firstName`.

However, more advanced benefits are only available with JOMP.

6.4.1 Associations

ActiveRecord relies on the user to specify the associations in the class itself. It has a variety of methods to do this. They are `has_one`, `has_many`, `belongs_to`, and `has_and_belongs_to_many`.

RhinoRecord takes a different approach. It only offers the equivalent of `has_many` and `belongs_to`, but instead of forcing the user to specify these, they are created when they are first needed.

Like ActiveRecord, RhinoRecord relies on certain conventions. First of all, it assumes that each record has an `id` column that uniquely identifies it. Secondly, it assumes that each foreign key refers to the table name. For example, if a table named `albums` has an `artist_id` field, it assumes that this refers to the `id` column in the `artists` table.

This is done by intercepting the getting of properties. The first time that a script refers to `album.artist`, this method will look for `artistId` in the object's properties. If this does exist, it will load the relevant artist and store it as a property for the album. This means that future calls to the artist will not need to go through this process. The relevant part of `__metaobject__.get` is here:

```
if (this.hasOwnProperty(propName + 'Id')) {
    var constr = eval(RhinoRecord.capitalize(propName));
    this[propName] = constr.findFirst({id: this[propName+'Id']});
    return this[propName];
 }
```

This satisfies the `belongs_to` relationship. As mentioned earlier, only the `has_many` relationship is supported of the others. Since we can therefore automatically assume that the relationship is one to many, we can take some shortcuts.

When `album.songs` is first referred to, this method will look for a `Song` constructor. If found, it will search the `songs` table for all records with a `album_id` matching the current album object:

```
if (propName.match(/s$/)) {
    var constr = eval(RhinoRecord
         .calcConstrNameFromPlural(propName));
    if (constr) {
      var options = {};
      options.params = {};
      options.params[this.tableName.slice(0,
         this.tableName.length-1)+'Id'] = this.id;
      this[propName] = constr.findAll(options);
      return this[propName];
    }
```

```
}
```

This setup is a little less flexible, but it means that there is less of a burden on the programmer. One benefit of ActiveRecord's approach is that it is able to pay the performance cost up front, whereas RhinoRecord pays it when the reference is first needed.

However, RhinoRecord could easily add methods to explicitly set up these relationships. The benefit of the RhinoRecord approach is that a developer is not required to do so.

Without these features, here is the code needed to initialize the objects for a music application.

```
this.albums = new Array();
this.artists = new Array();
this.songs = Song.findAll({orderByDesc: 'numDownloads'});

var iter = this.songs.iterator();
while (iter.hasNext()) {
  var tempSong = iter.next();
  if (!this.albums[tempSong.albumId]) {
    var album = Album.findFirst({id: tempSong.albumId});
    album.songs = new ArrayList();
    this.albums[tempSong.albumId] = album;

    if (!this.artists[album.artistId]) {
      var artist = Artist.findFirst({id: album.artistId});
      artist.albums = new ArrayList();
      this.artists[album.artistId] = artist;
    }
    album.artist = this.artists[album.artistId];
    album.artist.albums.add(album);
  }
  tempSong.album = this.albums[tempSong.albumId];
  tempSong.album.songs.add(tempSong);
}
```

With the association logic, this instead becomes:

```
this.songs = Song.findAll({orderByDesc: 'numDownloads'});
```

6.4.2 Advanced find methods

One nice feature of ActiveRecord is that it supports more advanced find features. A programmer could type `Album.find_by_title("Surf Cinema")`, and ActiveRecord would convert it to the less intuitive `Album.find(:first, :title=>"Surf Cinema")`.

With JOMP, JavaScript can do this as well. This again uses `__metaobject__.get`. Here is the excerpt:

```
if (propName.match(/^findBy/)) {
  var field = propName.match(/^findBy(.*)$/)[1];
  this[propName] = function (val) {
    var params = {};
    params[field] = val;
    return this.findFirst(params);
  }
  return this[propName];
}
```

While this does not add any additional functionality, it does allow for more aesthetic method calls, which arguably make the code more readable.

6.5 MobileMusic

In order to illustrate the advantages of RhinoFaces, I have created a music store web application called "MobileMusic". I had originally intended to include an interface for cell phones, but this was later abandoned. Nonetheless, the name stuck.

Figure 1 shows the homepage of the application.

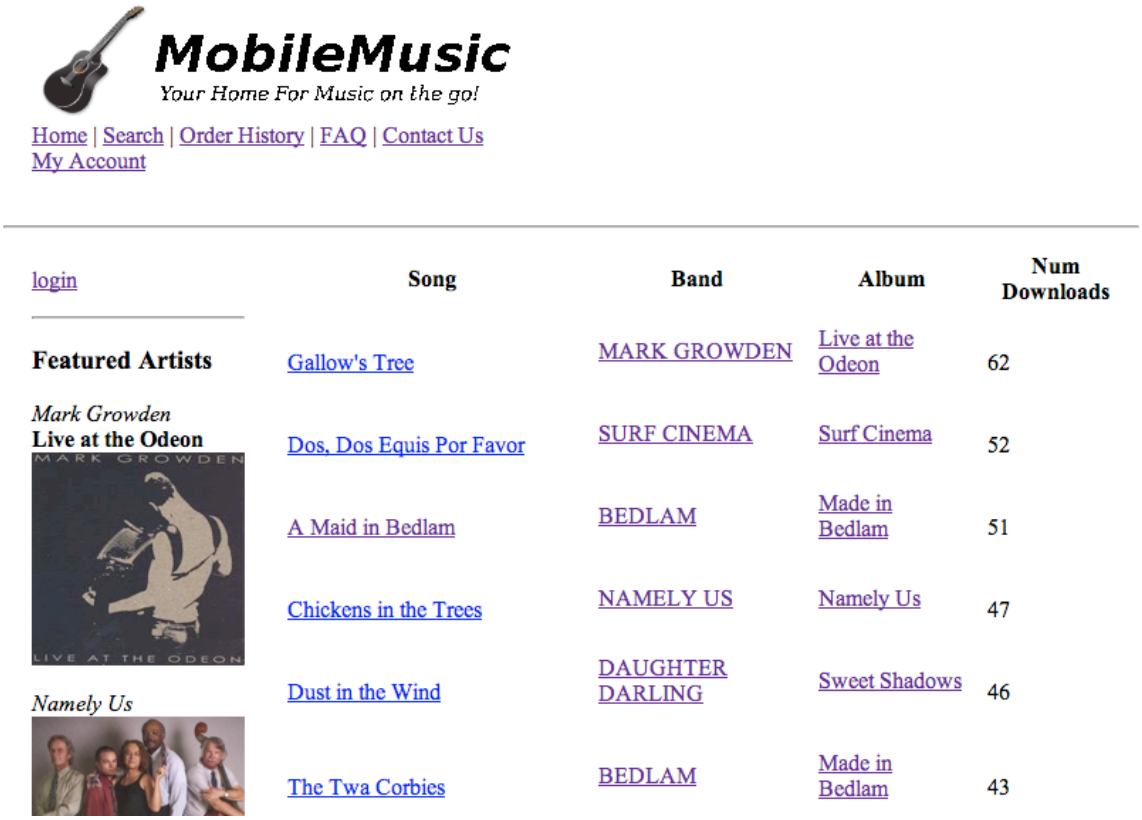

Figure 1: MobileMusic Homepage

6.5.1 Features

MobileMusic was built using RhinoFaces and a MySQL database. The sample music, artwork, and band information was taken from CDBaby.com, an existing online music store. This helped to give a realistic feel of how the application would work if it were a production system.

MobileMusic has public pages for browsing songs, viewing albums, and viewing artists. It allows customers to listen to excerpts of songs in mp3 format.

Customers can also buy albums and view their order history, though both actions require the customer to login first. The user has a shopping cart so that a separate transaction is not needed for every single item.

There is also a page for viewing orders. This is intended for MobileMusic employees.

6.5.2 Security

One of the principal security risks to any web application is a sloppy web developer. By giving API designers an easy way to restrict access at a granular level, this risk can be minimized. We've seen this already, but we will illustrate a more concrete example with MobileMusic.

For MobileMusic, we have a page for employees to view pending orders. This will need both billing and shipping information. Figure 2 shows this page when viewed by a MobileMusic administrator.

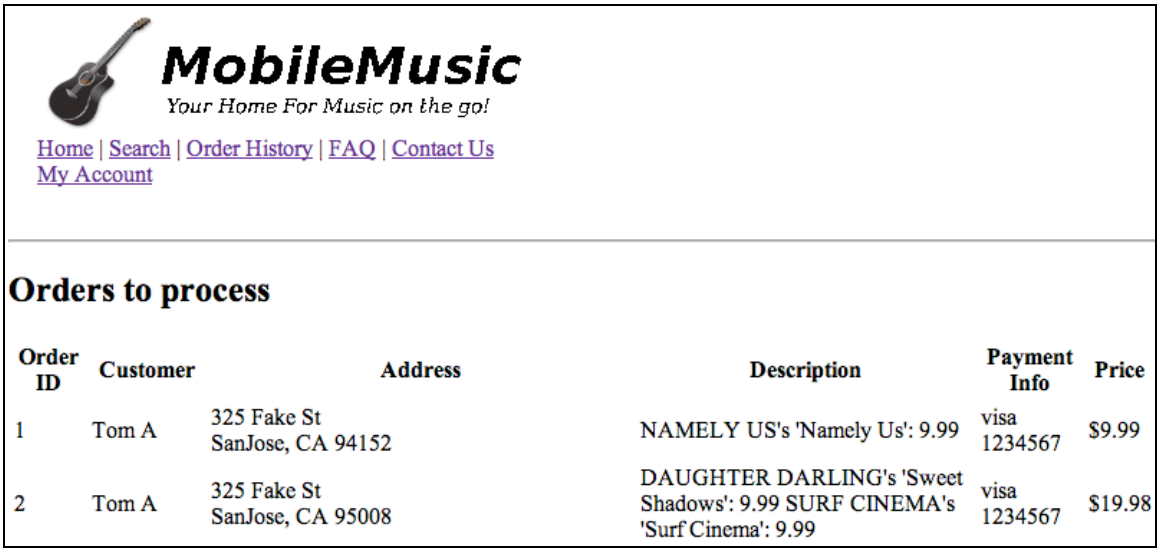

Figure 2: Admin View of Pending Orders

However, this page is not secure. I have made this page publicly accessible to simulate a careless developer. Any customer who discovers it would be able to see all orders.

However, the credit card information was protected through JOMP at the object level. Here is the relevant code:

```
var orderMO = Order.prototype.__metaobject__;
var oldOrderGet = orderMO.get;
orderMO.get = function(thisObj, prop) {
  if (prop=='creditCardNum' &&
      !jukebox.isAuthorized(thisObj.userId)){
    return "***RESTRICTED***";
  }
  else return oldOrderGet(thisObj, prop);
}
Order.prototype.__metaobject__ = orderMO;
```

As a result, even though the customer can see a page intended for employees, the most sensitive information remains secure. This is demonstrated in figure 3.

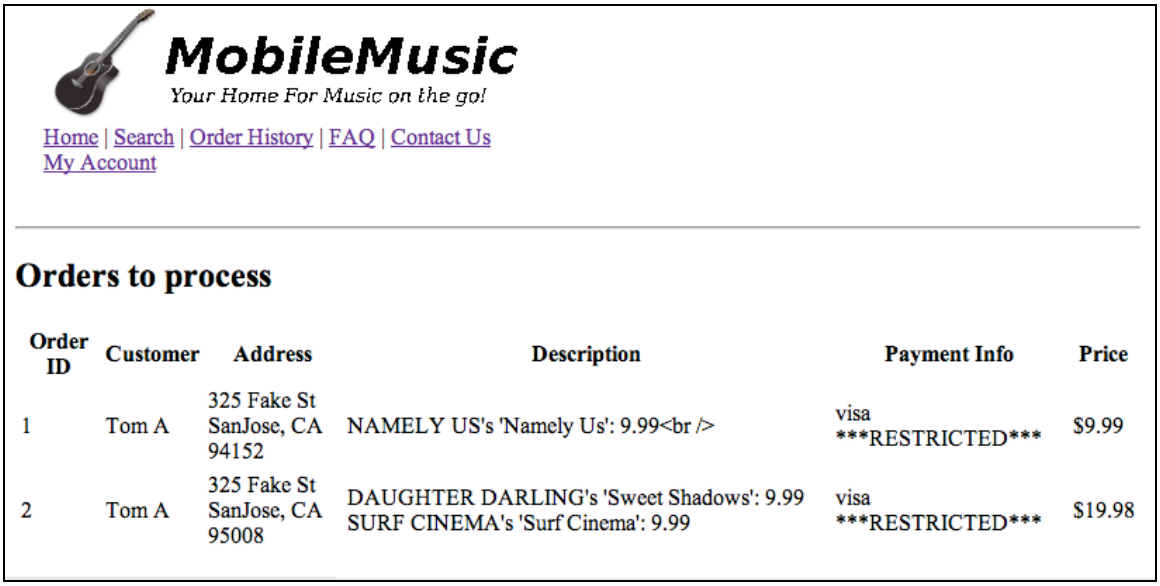

Figure 3: Non-Admin View of Pending Orders

This is not a very sophisticated protection, but it illustrates the basic concept. We can use a metaobject protocol to protect sensitive data at the object level. While this should not be the only source of security, it can help to give an extra layer of defense in case other security measures fail.

7 Related Work

Other work has been done to allow the intercepting of properties in JavaScript. In particular, Mozilla's implementations have added new features, and Java 6 has an interesting tool hidden in its version of Rhino. Also, PHP now includes methods to intercept properties, and it shares many characteristics with JavaScript

7.1 Mozilla JavaScript Getters and Setters

Mozilla has done some work on intercepting properties. Their description of this feature is in [2].

Unfortunately, their implementation does not offer the full functionality of `__metaobject__.get` or `__metaobject__.set`. It does not even allow you to intercept the setting and getting of existing properties. This design loses many of the advantages of getters and setters.

The main focus of the change appears to be to allow Firefox JavaScript to interact with Microsoft-specific JavaScript code. While this is an advantage for web developers, it does seem that the designers were too narrowly focused on this one specific issue. In their defense, a more powerful design might have cost more in terms of performance. Perhaps that was their primary concern.

However, there is an interesting parallel to CLOS. One of the primary concerns of the CLOS designers was to smoothly interact with the various Lisp object systems that preceded it [10]:

> The prospective CLOS user community was already using a variety of object-oriented extensions to Lisp. They were committed to large bodies of existing code, which they needed continue using and maintaining. … although they differed in surface details, they were all based, at a deeper level, on the same fundamental approach.

They dealt with this variety of systems through a powerful MOP. In some ways, this is a similar problem to interacting with the different JavaScript implementations of different browsers. Even with this limited addition to the language, the Mozilla team has given a powerful tool to developers to resolve this issue.

Another new feature of interest is the `__noSuchMethod__` method. This works just like Ruby's `method_missing`. However, due to the different designs of the language, this is less powerful. In Ruby, property references are indistinguishable from getting and setting properties. As a result, `method_missing` also intercepts missing property references. This is not the case for Mozilla's `__noSuchMethod__`.

7.2 Java 6 JavaScript

Java 6 has added support for scripting frameworks. As part of this, it includes a version of Mozilla's Rhino. For the most part, this is a more limited implementation. It does not include support for continuations or E4X, for example. However, there is one interesting, almost entirely undocumented feature in Sun's implementation.

Sun's Java 6 version of Rhino includes a `JSAdapter` class [17]. This offers much of the same functionality as my proposed extensions.

Instead of modifying the behavior of all objects in the language, this approach instead creates a special object with additional functionality. This object can be used to wrap other objects. When you attempt to get or set a property for this special object, it will call its `__get__` or `__put__` method, if one exists. Here is an example that will restrict access to the salary field (unless you refer to the emp object directly):

```
var emp = {name:'Joe Bob Briggs', salary:5000}
emp.__get__ = function(fieldName) {
  if (fieldName == 'salary') {
    throw new Error("Salary is restricted");
  }
  return this[fieldName];
}

var wrapper = new JSAdapter(emp);
print("Reading details for employee '" + wrapper.name + "'.\n");
try {
  print('Salary is ' + wrapper.salary);
}
catch(e) {
  print(e.name + ": " + e.message);
}
```

The JSAdapter objects also have `__has__`, `__delete__`, and `__getIds__`. They effectively cover every way that a JavaScript object can be accessed, and almost match JOMP's functionality. The only missing piece is JOMP's `hasInstanceOf` method.

One disadvantage of this approach is the need for a special wrapper object. While this minimizes the change to the language, it also makes it more difficult to use this functionality within an object's constructor.

With this approach, we cannot modify the behavior of an object itself at runtime. It is not a true MOP. This manner of adding these extensions is very clever. However, it would be better to adapt the JavaScript Object itself rather than relying on a new, special wrapper object.

Still, `JSAdapter` deserves credit for introducing a useful feature to the language with a negligible impact on the language's design.

7.3 PHP 5 Comparison

JavaScript and PHP have some striking similarities in their basic design. In particular, it is common in both languages to access properties directly. This is getting to be less true for PHP, but it is still far from unusual to see code like the following:

```
echo user->full_name;
```

In contrast, you never access variables directly in Java or Ruby. It can be done, but is against the conventions of the language.

A more important point is that both of these languages will accept new properties for existing objects. In Java, you cannot add a property to an object if it is not available for its class. In Ruby, you can do so through the use of singleton classes, but it is a much more complicated process.

Also, PHP has the ability to intercept references to properties with its `__get` and `__set` methods. It is not as powerful as what I have proposed; it only catches properties that do not exist. However, this should still be enough to replicate `method_missing`.

Unfortunately for PHP developers, functions are not first class citizens in the language. Function references are never intercepted by `__get` or `__set`. And while you can make anonymous functions in PHP with `create_function`, these functions cannot be set as methods. This will fail:

```
$emp->work = create_function('$beg,$end', 'echo "Work from "
   . $beg . " to " . $end;');
$emp->work("9", "5");
```

The function is set as a property of `$emp`, but it is only a property. It cannot be treated as a method. So while the above example fails, this will work:

```
$emp->work = create_function('$beg,$end', 'echo "Work from "
   . $beg . " to " . $end;');
$foo = $emp->work;
$foo("9", "5");
```

PHP has a `__call` method that is invoked for unrecognized methods. However, because of its more complicated structure, it needs `__get`, `__set`, and `__call` to mimic the functionality of Ruby's `method_missing`. And unlike JavaScript and Ruby, it has no ability to add methods to an existing object.

By introducing a MOP that can intercept property references for JavaScript objects, we gain the ability to replicate `method_missing`, in addition to allowing a wide variety of other behavior. PHP's similar design nearly gives it the same possibilities, but it lacks the key element of JavaScript's first class functions.

8 Conclusion

JavaScript has only a few constructs in its language. However, these are very powerful and well designed. This gives it an elegance more associated with languages like Scheme than with other languages in the C family.

The central construct in JavaScript is the object. Except for operators and the global object, everything in JavaScript is a property of some other object.

Because of this, we can create a powerful and sophisticated MOP by allowing programmers to modify the behavior of objects. The prototype-based object system lets us modify large groups of objects or individual objects with equal ease. The fact that functions are properties of objects allows us to modify those as well without having to alter the implementation of functions.

In this project, I have created JOMP, a new metaobject protocol for JavaScript. I have used it to demonstrate a number of traditional MOP uses, including security, tracing, and introducing multiple inheritance. I have also shown that intercepting the getting and setting of properties lets us replicate almost all of the advanced metaprogramming features in Ruby.

Furthermore, as a practical example I have created the RhinoFaces web development framework, built with JSF, Rhino JavaScript, and JOMP. With the sample MobileMusic application, I have illustrated how JOMP can improve a developer's productivity. In particular, I have demonstrated how JOMP can simplify database access and improve security.

JavaScript already dominates the client-side of web development. In addition, it is becoming an increasingly viable contender for the server-side. With these additional features, it could become an even stronger choice.

References

[1] Black, David. Ruby for Rails. Manning Publications Co. 2006.

[2] Core JavaScript 1.5 Guide: Creating New Objects: Defining Getters and Setters. (Accessed October 2007). http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Guide:Creating_New_Objects:Defining_Getters_and_Setters.

[3] Crockford, Douglas. The JavaScript Programming Language. Yahoo presentation. (Accessed May 2007). http://yuiblog.com/blog/2007/01/24/video-crockford-tjpl/.

[4] Crockford, Douglas. JSLint: The JavaScript Verifier. (Accessed April 2007). http://www.jslint.com/lint.html.

[5] Denker, Marcus, Stéphane Ducasse, Andrian Lienhard, Philippe Marschall. Sub-Method Reflection. Journal of Object Technology, special issue TOOLS Europe 2007, October 2007, vol. 6, no. 9. http://www.jot.fm/issues/issue_2007_10/paper14/.

[6] ECMA (European Computer Manufacturers Association). ECMAScript Language Specification. (Accessed August 2007). http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[7] Geary, David and Cay Horstmann. Core JavaServer Faces, 2nd ed. Prentice Hall. 2007.

[8] Flanagan, David. JavaScript: the Definitive Guide, 5th ed. O'Reilly Media Inc. 2006.

[9] HttpUnit project homepage. (Accessed December 2007). http://httpunit.sourceforge.net/.

[10] Kiczales, Gregor, Jim des Rivières, and Daniel G. Bobrow. The Art of the Metaobject Protocol. MIT Press, 1991.

[11] Kidd, Eric. Why Ruby is an Acceptable Lisp. (Accessed April 2007). http://www.randomhacks.net/articles/2005/12/03/why-ruby-is-an-acceptable-lisp.

[12] Lee, Arthur H. and Joseph L Zachary. Reflections on Metaprogramming. IEEE Transactions on Software Engineering, November 1995, vol. 21, no. 11.

[13] Matsumoto, Yukihiro. Ruby's Lisp features. Ruby-talk mailing list archives. (Accessed May 2007). http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/179642.

[14] Paepcke, Andreas. User-level Crafting Introducing the CLOS Metaobject Protocol. (Accessed December 2006). http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps.

[15] Phobos project homepage. (Accessed December 2007). https://phobos.dev.java.net/.

[16] Rivard, Fred. Smalltalk: a Reflective Language. (Accessed November 2006). http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/rivard/rivard.html.

[17] Sundararajan, A. Self, JavaScript, and JSAdapter. (Accessed October 2007). http://blogs.sun.com/sundararajan/entry/self_javascript_and_jsadapter.

[18] Tanter, Éric, Noury M. N. Bouraqadi-Saadani, and Jacques Noyé. Reflex – Towards an Open Reflective Extension of Java. (Accessed December 2006).

http://www.dcc.uchile.cl/~etanter/research/publi/2001/tanterBouraqadiNoye-reflection2001.pdf.
[19] Tate, Bruce. Beyond Java. O'Reilly Media Inc. 2005.
[20] Thomas, Dave. Programming the World in a Browser - Real Men Don't Do JavaScript Do They?!. Journal of Object Technology, vol. 6 no. 10 November-December 2007, pp. 25-29 http://www.jot.fm/issues/issue_2007_10/column3.
[21] Yegge, Steve. Lisp is Not an Acceptable Lisp. (Accessed April 2007). http://steve-yegge.blogspot.com/2006/04/lisp-is-not-acceptable-lisp.html.
[22] Ungar, Dan and Randall B. Smith. Self: the Power of Simplicity. (Accessed October 2007). http://www.cs.ucsb.edu/~urs/oocsb/self/papers/self-power.html.
[23] Welch, Ian and Fan Lu. Policy-driven Reflective Enforcement of Security Policies. Proceedings of the 2006 ACM symposium on Applied computing.