Classical Object-Oriented Programming with ECMAScript

Mike Gerwitz

June 12, 2014
(Working Draft)

Abstract

ECMAScript (more popularly known by the name "JavaScript") is the language of the web. In the decades past, it has been used to augment web pages with trivial features and obnoxious gimmicks. Today, the language is used to write full-featured web applications that rival modern desktop software in nearly every regard and has even expanded to create desktop and server software. With increased usage, there is a desire to apply more familiar development paradigms while continuing to take advantage of the language's incredibly flexible functional and prototypal models. Of all of the modern paradigms, one of the most influential and widely adopted is the Classical Object-Oriented paradigm, as represented in languages such as Java, C++, Python, Perl, PHP and others. ECMAScript, as an object-oriented language, contains many features familiar to Classical OO developers. However, certain features remain elusive. This paper will detail the development of a classical object-oriented framework for ECMAScript — ease.js — which aims to address these issues by augmenting ECMAScript's prototype model to allow the creation of familiar class-like objects. This implementation enforces encapsulation and provides features that most Classical OO developers take for granted until the time that ECMAScript implements these features itself.[1]

Contents

1 Class-Like Objects in ECMAScript **1**
- 1.1 Prototypes . . . 2
- 1.2 Privileged Members . . . 3
- 1.3 Subtypes and Polymorphism . . . 3
  - 1.3.1 Extensible Constructors . . . 5
- 1.4 Shortcomings . . . 6

2 Hacking Around Prototypal Limitations **6**
- 2.1 Extensible Constructors: Revisited . . . 6
- 2.2 Encapsulating Data . . . 7
  - 2.2.1 A Naive Implementation . . . 7
  - 2.2.2 A Proper Implementation . . . 9
  - 2.2.3 Private Methods . . . 10
- 2.3 Protected Members . . . 11
  - 2.3.1 Protected Member Encapsulation Challenges . . . 14

3 Encapsulating the Hacks **14**
- 3.1 Constructor/Prototype Factory . . . 15
  - 3.1.1 Factory Conveniences . . . 17
- 3.2 Private Member Encapsulation . . . 18
  - 3.2.1 Wrapper Implementation Concerns . . . 21

4 Licenses **23**
- 4.1 Document License . . . 23
- 4.2 Code Listing License . . . 23
  - 4.2.1 Code Listing License Exceptions . . . 23
- 4.3 Reference Licenses . . . 23

5 Author's Note **23**

1 Class-Like Objects in ECMAScript

JavaScript is a multi-paradigm scripting language standardized by ECMAScript, incorporating object-oriented, functional and imperative styles. The Object-Oriented paradigm in itself supports two sub-paradigms - prototypal and classical, the latter of which is popular in languages such as Java, C++, Python, Perl, Ruby, Lisp, PHP, Smalltalk, among many others. ECMAScript itself is prototypal.

The creation of objects in ECMAScript can be as simple as using an object literal, as defined by curly braces:

```
var obj = { foo: "bar" };
```

In a classical sense, object literals can be thought of as anonymous singletons; [2] that is, they have no name (they are identified by the variable to which they are assigned) and only one instance of the literal will exist throughout the life of the software.[2] For example, calling a function that returns the same object literal will return a distinct, entirely unrelated object for each invocation:

```
function createObj()
{
    return { name: "foo" };
}

createObj() !== createObj();
```

Using this method, we can create basic objects that act much like class instances, as demonstrated in Listing 1:

```
1  var obj = {
2          name: "Foo",
3
4          setName: function( val )
5          {
6              obj.name = val;
```

[1] There was discussion of including classes in ECMAScript 6 "Harmony", however it is not within the specification at the time of writing. At that time, the framework could be used to transition to ECMAScript's model, should the developer choose to do so.

[2] Technically, one could set the prototype of a constructor to be the object defined by the literal (see Listing 2), however the resulting instances would be prototypes, not instances of a common class shared by the literal and each subsequent instance.

www.dbooks.org

```
7           },
8
9           getName: function()
10          {
11              return obj.name;
12          }
13      };
14
15  obj.getName(); // "Foo"
16  obj.setName( "Bar" );
17  obj.getName(); // "Bar"
```

Listing 1: A "singleton" with properties and methods

1.1 Prototypes

We could re-use `obj` in Listing 1 as a *prototype*, allowing instances to inherit its members. For example:

```
18  function Foo() {}
19  Foo.prototype = obj;
20
21  var inst1 = new Foo(),
22      inst2 = new Foo();
23
24  inst2.setName( "Bar" );
25
26  inst1.getName(); // "Bar"
27  inst2.getName(); // "Bar"
```

Listing 2: Re-using objects as prototyes **(bad)**

In Listing 2 above, we define `Foo` to be a *constructor*[3] with our previous object `obj` as its prototype. Unfortunately, as shown in Listing 1, `name` is being set on `obj` itself, which is a prototype shared between both instances. Setting the name on one object therefore changes the name on the other (and, indeed, all instances of `Foo`). To illustrate this important concept, consider Listing 3 below, which continues from Listing 2:

```
28  obj.foo = "bar";
29  inst1.foo; // "bar"
30  inst2.foo; // "bar"
```

Listing 3: The effect of prototypes on instances

Clearly, this is not how one would expect class-like objects to interact; each object is expected to have its own state. When accessing a property of an object, the members of the object itself are first checked. If the member is not defined on the object itself,[4] then the prototype chain is traversed. Therefore, we can give objects their own individual state by defining the property on the individual instances, rather than the prototype, as shown in Listing 4.[5]

```
31  inst1.foo = "baz";
32  inst1.foo; // "baz"
33  inst2.foo; // "bar"
34
35  delete inst1.foo;
36  inst1.foo; // "bar"
```

Listing 4: Setting properties per-instance

This does not entirely solve our problem. As shown in Listing 1, our `obj` prototype's methods (`getName()` and `setName()`) reference `obj.name` - our prototype. Listing 5 demonstrates the problem this causes when attempting to give each instance its own state in regards to the `name` property:

```
37  // ...
38
39  inst1.name = "My Name";
40  inst1.getName(); // "Foo"
```

Listing 5: Referencing prototype values in `obj` causes problems with per-instance data

ECMAScript solves this issue with the `this` keyword. When a method[6] of an instance's prototype is invoked, `this` is bound, by default,[7] to a reference of that instance. Therefore, we can replace `obj` in Listing 1 with the prototype definition in Listing 6 to solve the issue demonstrated in Listing 5:

```
1   function Foo( name )
2   {
3       this.name = name;
4   };
5
6   Foo.prototype = {
7       setName = function( name )
8       {
9           this.name = name;
10      },
11
12      getName = function()
13      {
14          return this.name;
15      }
16  };
17
```

[3] A "constructor" in ECMAScript is simply any function intended to be invoked, often (but not always) with the `new` operator, that returns a new object whose members are derived from the function's `prototype` property.

[4] Note that "not defined" does not imply *undefined*; `undefined` is a value.

[5] Also demonstrated in Listing 4 is the effect of the `delete` keyword, which removes a member from an object, allowing the values of the prototype to "peek through" as if a hole exists in the object. Setting the value to `undefined` will not have the same effect, as it does not produce the "hole"; the property would return `undefined` rather than the value on the prototype.

[6] A *method* is simply an invokable property of an object (a function).

[7] One can override this default behavior with `Function.call()` or `Function.apply()`.

```
18  var inst = new Foo( "Bar" );
19  inst.name;      // "Bar"
20  inst.getName(); // "Bar"
21
22  inst.setName( "Baz" );
23  inst.getName(); // "Baz"
24
25  inst.name = "Foo";
26  inst.getName(); // "Foo"
```

Listing 6: Re-using objects as prototypes **(good)**

Listing 6 shows that `this` is also bound to the new instance from within the constructor; this allows us to initialize any properties on the new instance before it is returned to the caller.[8] Evaluation of the example yields an additional concern — the observation that all object members in ECMAScript are public.[9] Even though the `name` property was initialized within the constructor, it is still accessible outside of both the constructor and the prototype. Addressing this concern will prove to be an arduous process that will be covered at great length in the following sections. For the time being, we will continue discussion of conventional techniques, bringing us to the concept of *privileged members*.

1.2 Privileged Members

The concept of *encapsulation* is a cornerstone of classical object-oriented programming. Unfortunately, as Listing 6 demonstrates, it becomes difficult to encapsulate data if all members of a given object are accessible publicly. One means of addressing this issue is to take advantage of the fact that functions introduce scope, allowing us to define a local variable (or use an argument) within the constructor that is only accessible to the *privileged member* `getName()`.

```
1  function Foo( name )
2  {
3      this.getName = function()
4      {
5          return name;
6      };
7
8      this.setName = function( newname )
9      {
10         name = newname;
11     };
12 }
```

Listing 7: Using privileged members to encapsulate data

[8] It is worth mentioning that one can explicitly return an object from the constructor, which will be returned in place of a new instance.

[9] That is not to say that encapsulation is not possible; this statement is merely emphasizing that properties of objects do not support access modifiers. We will get into the topic of encapsulation a bit later.

	Heap Usage	Inst. Time	Call Time
Listing 6	49.7M	234ms	17ms
Listing 7	236.0M	1134ms	28ms
% Change	374.8%	384.6%	64.7%

Figure 1: Comparing performance of privileged member and prototype implementations under v8. The heap usage column represents the heap usage after instantiating `Foo` under the respective implementation n times, and the Inst. CPU column reflects the amount of time spent instantiating the n objects. The Call CPU column reflects the amount of time spent invoking *each* member of *one* instance n times. $n = 1,000,000$. Lower numbers are better. Different environments may have different results.

If `name` in Listing 7 is encapsulated within the constructor, our methods that *access* that encapsulated data must *too* be declared within the constructor;[10] otherwise, if placed within the prototype, `name` would be out of scope. This implementation has an unfortunate consequence — our methods are now being *redeclared* each and every time a new instance of `Foo` is created, which has obvious performance penalties (see Figure 1).[11]

Due to these performance concerns, it is often undesirable to use privileged members; many developers will instead prefix, with an underscore, members intended to be private (e.g. `this._name`) while keeping all methods on the prototype.[12] This serves as a clear indicator that the API is not public, is subject to change in the future and should not be touched. It also allows the property to be accessed by subtypes,[13] acting like a protected member. Unfortunately, this does not encapsulate the data, so the developer must trust that the user will not tamper with it.

1.3 Subtypes and Polymorphism

In classical terms, *subtyping* (also known as *subclassing*) is the act of extending a *supertype* (creating a *child* class from a *parent*) with additional functionality. The subtype is said to *inherit* its members from the supertype.[14] Based on our prior examples in section 1.1, one could clearly see how the prototype of any constructor could be replaced with an instance of another constructor, indefinitely, to achieve an inheritance-like effect. This useful consequence of the prototype model is demonstrated in Listing 8.[15]

[10] One may mix prototypes and privileged members.

[11] As a general rule of thumb, one should only use privileged members for methods that access encapsulated data; all other members should be part of the prototype.

[12] One example of a library that uses underscores in place of privileged members is Dojo at http://dojotoolkit.org.

[13] The term "subtype" is not truly the correct term here. Rather, the term in this context was meant to imply that an instance of the constructor was used as the prototype for another constructor, acting much like a subtype (child class).

[14] In the case of languages that support access modifiers, only public and protected members are inherited.

[15] Unfortunately, a responsible implementation is not all so elegant in practice.

www.dbooks.org

```
1  var SubFoo = function( name )
2  {
3      // call parent constructor
4      Foo.call( this, name );
5  };
6  SubFoo.prototype             = new Foo();
7  SubFoo.prototype.constructor = SubFoo;
8
9  // build upon (extend) Foo
10 SubFoo.prototype.hello = function()
11 {
12     return "Hello, " + this.name;
13 };
14
15 var inst = new SubFoo( "John" );
16 inst.getName(); // "John"
17 inst.hello(); // "Hello, John"
```

Listing 8: Extending prototypes (creating subtypes) in ECMAScript

Consider the implications of Listing 8 with a close eye. This extension of `Foo` is rather verbose. The first (and rather unpleasant fact that may be terribly confusing to those fairly inexperienced with ECMAScript) consideration to be made is `SubFoo`'s constructor. Note how the supertype (`Foo`) must be invoked *within the context of `SubFoo`*[16] in order to initialize the variables.[17] However, once properly deciphered, this call is very similar to invocation of parent constructors in other languages.

Following the definition of `SubFoo` is its prototype (line 6). Note from section 1.1 that the prototype must contain the members that are to be accessible to any instances of the constructor. If we were to simply assign `Foo` to the prototype, this would have two terrible consequences, the second of which will be discussed shortly. The first consequence would be that all members of `Foo` *itself* would be made available to instances of `SubFoo`. In particular, you would find that `( new SubFoo() ).prototype === Foo.prototype`, which is hardly your intent. As such, we must use a new instance of `Foo` for our prototype, so that the prototype contains the appropriate members.

We follow the prototype assignment with another alien declaration — the setting of `SubFoo.prototype.constructor` on line 7. To understand why this is necessary, one must first understand that, given any object `o` such that `var o = new O()`, `o.constructor === O`.[18] Recall from section 1.1 that values "peek through holes" in the prototype chain. In this case, without our intervention, `SubFoo.prototype.constructor === Foo` because `SubFoo.prototype = new Foo()`. The `constructor` property is useful for reflection, so it is important that we properly set this value to the appropriate constructor — `SubFoo`. Since `SubFoo.prototype` is an *instance* of `Foo` rather than `Foo` itself, the assignment will not directly affect `Foo`. This brings us to our aforementioned second consequence of assigning `SubFoo.prototype` to a *new* instance of `Foo` — extending the prototype by adding to or altering existing values would otherwise change the supertype's constructor, which would be an unintentional side-effect that could have drastic consequences on the software.

As an example of extending the prototype (we have already demonstrated overwriting the `constructor` and this concept can be applied to overriding any members of the supertype), method `hello()` has been included in Listing 8 on line 10. Note that `this` will be bound to the instance that the method is being invoked upon, since it is referenced within the prototype. Also note that we are assigning the function in a slightly different manner than in Listing 6; this is necessary to ensure that we do not overwrite the prototype we just declared. Any additional members must be declared explicitly in this manner, which has the negative consequence of further increasing the verbosity of the code.

An instance of a subtype can be used in place of any of its supertypes in a concept known as *polymorphism*. Listing 9 demonstrates this concept with `getFooName()`, a function that will return the name of any object of type `Foo`.[19]

```
1  function getFooName( foo )
2  {
3      if ( !( foo instanceof Foo ) )
4      {
5          throw TypeError(
6              "Expected instance of Foo"
7          );
8      }
9
10     return foo.getName();
11 }
12
13 var inst_parent = new Foo( "Parent" ),
14     inst_child  = new SubFoo( "Child" );
15
16 getFooName( inst_parent ); // "Parent"
17 getFooName( inst_child );  // "Child"
18 getFooName( {} );          // throws TypeError
```

Listing 9: Polymorphism in ECMAScript

The concepts demonstrated in this section could be easily used to extend prototypes indefinitely, creating what is called a *prototype chain*. In the case of an instance of `SubFoo`, the prototype chain

[16] If `Function.call()` or `Function.apply()` are not properly used, the function will, depending on the environment, assign `this` to the global scope, which is absolutely not what one wants. In strict mode, this effect is mitigated, but the result is still not what we want.

[17] If the constructor accepts more than a few arguments, one could simply do: `Foo.apply( this, arguments );`

[18] One could apply this same concept to other core ECMAScript objects. For example, `( function() {}).constructor === Function`, `[].constructor === Array`, `{}.constructor === Object`, `true.constructor === Boolean` and sofourth.

[19] Please note that the `typeof` operator is not appropriate in this situation, as both instances of `Foo` and `SubFoo` would be considered typeof "object". The `instanceof` operator is appropriate when determining types of objects in terms of their constructor.

of most environments would likely be: `SubFoo`, `Foo`, `Object` (that is, `Object.getPrototypeOf( new SubFoo() ) === SubFoo`, and so fourth).[20] Keep in mind, however, that the further down the prototype chain the engine must traverse in order to find a given member, the greater the performance impact.

Due to the method used to "extend" prototypes, it should also be apparent that multiple inheritance is unsupported by ECMAScript, as each each constructor may only have one `prototype` property.[21]

1.3.1 Extensible Constructors

Before moving on from the topic of extending prototypes, the assignment of `SubFoo.prototype` deserves some additional discussion. Consider the implications of this assignment; particularity, the invocation of the constructor `Foo`. ECMAScript does not perform assignments to prototypes differently than any other assignment, meaning all the logic contained within the constructor `Foo` will be executed. In our case, this does not have any terrible consequences — `name` will simply be initialized to `undefined`, which will be overridden once `SubType` is invoked. However, consider what may happen if `Foo` performed checks on its arguments.

```
1  function Foo( name )
2  {
3      if ( typeof name !== 'string' )
4      {
5          throw TypeError( "Invalid name" );
6      }
7
8      this.name = name;
9  }
10
11 // ...
12 SubFoo.prototype = new Foo(); // TypeError
```

Listing 10: Potential constructor problems for prototype assignments

As Listing 10 shows, we can no longer use a new instance of `Foo` as our prototype, unless we were to provide dummy data that will pass any type checks and validations that the constructor performs. Dummy data is not an ideal solution — it muddies the code and will cause subtypes to break should any validations be added to the supertype in the future.[22] Furthermore, all constructor logic will still be performed. What if `Foo` were to do something considerably more intensive — perform vigorous data validations or initialize a database connection, perhaps?[23] Not only would we have to provide potentially complicated dummy data or dummy/stubbed objects, our prototype assignment would also incur an unnecessary performance hit. Indeed, the construction logic would be performed $n + 1$ times — once for the prototype and once for each instance, which would overwrite the results of the previous constructor (or duplicate, depending on implementation).

How one goes about solving this problem depends on the needs of the constructor. Let us first consider a very basic solution — ignoring constructor logic if the provided argument list is empty, as is demonstrated in Listing 11.

```
1  function Foo( name )
2  {
3      if ( arguments.length === 0 )
4      {
5          return;
6      }
7
8      // ...
9
10     this.name = name;
11 }
12
13 // ...
14 SubType.prototype = new Foo(); // OK
```

Listing 11: Ignoring construction logic if provided with an empty argument list

This solution has its own problems. The most apparent issue is that one could simply omit all constructor arguments to bypass constructor logic, which is certainly undesirable.[24] Secondly — what if `Foo`'s `name` parameter was optional and additional construction logic needed to be performed regardless of whether or not `name` was provided? Perhaps we would want to provide a default value for `name` in addition to generating a random hash that can be used to uniquely identify each instance of `Foo`. If we are immediately returning from the constructor when all arguments are omitted, then such an implementation is not possible. Another solution is needed in this case.[25]

A solution that satisfies all needs involves a more complicated hack that we will defer to section 2.1.[26]

[20] ECMAScript 5 introduces `Object.getPrototypeOf()`, which allows retrieving the prototype of an object (instance). Some environments also support the non-standard `__proto__` property, which is a JavaScript extension.

[21] Multiple inheritance is well-known for its problems. As an alternative, styles of programming similar to the use of interfaces and traits/mixins in other languages are recommended and are possible in ECMAScript.

[22] Of course, if the constructor of the supertype changes, there are always BC (backwards-compatibility) concerns. However, in the case of validations in the constructor, they may simply enforce already existing docblocks, which should have already been adhered to.

[23] Constructors should take care in limiting what actions they perform, especially if they produce side-effects.

[24] Constructors allow us to initialize our object, placing it in a consistent and predictable state. Allowing a user to bypass this logic could not only introduce unintended consequences during the life of the object, but would mandate additional checks during method calls to ensure the current state is sane, which will add unnecessary overhead.

[25] That is not to say that our first solution — immediately returning if no arguments are provided — is useless. This is a commonly used method that you may find useful for certain circumstances.

[26] One may ask why, given all of the complications of extending prototypes, one doesn't simply set `SubFoo.prototype = Foo.prototype`. The reason for this is simple — we would not be

www.dbooks.org

1.4 Shortcomings

ECMAScript's prototype model is highly flexible, but leaves much to be desired:

Access Modifiers Classical OOP permits, generally, three common access modifiers: public, protected and private. These access modifiers permit encapsulating data that is unique *per instance* of a given type, without the performance penalties of privileged members (see Listing 7).

Not only are access modifiers unsupported, but the concept of protected members is difficult difficult in ECMAScript. In order for a member to be accessible to other objects higher up on the prototype chain ("subtypes"), they must be public. Using privileged members would encapsulate the data within the constructor, forcing the use of public methods to access the data and disallowing method overrides, effectively destroying any chances of a protected API.[27]

Intuitive Subtyping Consider the verbosity of Listing 8. Now imagine how much duplicate code is required to maintain many subtypes in a large piece of software. This only serves to distract developers from the actual business logic of the prototype, forcing them to think in detailed terms of prototypes rather than in terms of the problem domain.[28]

Furthermore, as discussed in section 1.3.1, creating extensible constructors requires considerable thought that must be handled on a case-by-case basis, or requires disproportionately complicated hacks (as will be demonstrated in section 2.1).

Fortunately,[29] those issues can be worked around with clever hacks, allowing us to continue closer toward a classical development model.

2 Hacking Around Prototypal Limitations

Section 1 demonstrated how one would work within the limitations of conventional ECMAScript to produce class-like objects using prototypes. For those coming from other classical object-oriented languages, these features are insufficient. In order to address many of the remaining issues, more elaborate solutions are necessary.

It should be noted that all the hacks in this section will, in some way or another, introduce additional overhead, although it should be minimal in comparison with the remainder of the software that may implement them. Performance considerations will be mentioned where the author finds it to be appropriate. Do not let this concern deter you from using these solutions in your own code — always benchmark to determine where the bottleneck lies in your software.

2.1 Extensible Constructors: Revisited

Section 1.3.1 discussed improving constructor design to allow for extensibility and to improve performance. However, the solution presented did not provide a consistent means of creating extensible constructors with, for example, optional argument lists.

The only way to ensure that the constructor will bypass validation and initialization logic only when used as a prototype is to somehow indicate that it is being used as such. Since prototype assignment is in no way different than any other assignment, no distinction can be made. As such, we must create our own.

```
1  var Foo = ( function( extending )
2  {
3      var F = function( name )
4      {
5          if ( extending ) return;
6
7          if ( typeof name !== 'string' )
8          {
9              throw TypeError( "Invalid name" );
10         }
11
12         this.name = name || "Default";
13
14         // hypothetical; impl. left to reader
15         this.hash = createHash();
16     };
17
18     F.asPrototype = function()
19     {
20         extending = true;
21
22         var proto = new F();
23
24         extending = false;
25         return proto;
26     };
27
28     F.prototype = {
29         // getName(), etc...
30     };
31
32     return F;
33 } )( false );
34
```

able to extend the prototype without modifying the original, as they would share references to the same object.

[27] As ease.js will demonstrate, protected APIs are possible through a clever hack that would otherwise lead to terrible, unmaintainable code.

[28] The ability to think within the problem domain rather than abstract machine concepts is one of the key benefits of classical object-oriented programming.

[29] Well, fortunately in the sense that ECMAScript is flexible enough that we can work around the issues. It is, however, terribly messy. In ECMAScript's defense — this is a consequence of the prototypal model; our desire to use class-like objects instead of conventional prototypes produces the necessity for these hacks.

```
35  function SubFoo() { /* ... */ }
36  SubFoo.prototype = Foo.asPrototype(); // OK
37  // ...
38
39  var foo1 = new Foo();
40  foo1.getName(); // "Default"
41  foo1.hash;      // "..."
42
43  var foo2 = new Foo( "Bar" );
44  foo2.getName(); // "Bar"
45  foo2.hash;      // "..."
```

Listing 12: Working around prototype extending issues

One solution, as demonstrated in Listing 12, is to use a variable (e.g. `extending`) to indicate to a constructor when it is being used to extend a prototype. The constructor, acting as a closure, can then check the value of this flag to determine whether or not to immediately return, avoiding all construction logic. This implementation would allow us to return only a prototype, which is precisely what we are looking for.

It is unlikely that we would want to expose `extending` directly for modification, as this would involve manually setting the flag before requesting the prototype, then remembering to reset it after we are done. Should the user forget to reset the flag, all future calls to the constructor would continue to ignore all constructor logic, which could lead to confusing bugs in the software. To work around this issue, Listing 12 offers an `asPrototype()` method on `Foo` itself, which will set the flag, create a new instance of `Foo`, reset the flag and return the new instance.[30]

In order to cleanly encapsulate our extension logic, `Foo` is generated within a self-executing function (using much the same concept as privileged members in section 1.2, with a slightly different application).[31] This gives `Foo` complete control over when its constructor logic should be ignored. Of course, one would not want to duplicate this mess of code for each and every constructor they create. Factoring this logic into a common, re-usable implementation will be discussed a bit later as part of a class system (see section 3.1).

2.2 Encapsulating Data

We discussed a basic means of encapsulation with privileged members in section 1.2. Unfortunately, the solution, as demonstrated in Listing 7, involves redeclaring methods that could have otherwise been defined within the prototype and shared between all instances. With that goal in mind, let us consider how we may be able to share data for multiple instances with a single method definition in the prototype.

We already know from Listing 12 that we can truly encapsulate data for a prototype within a self-executing function. Methods can then, acting as closures, access that data that is otherwise inaccessible to the remainder of the software. With that example, we concerned ourselves with only a single piece of data — the `extending` flag. This data has no regard for individual instances (one could think of it as static data, in classical terms). Using Listing 12 as a starting point, we can build a system that will keep track of data *per-instance*. This data will be accessible to all prototype members.

2.2.1 A Naive Implementation

One approach to our problem involves to assigning each instance a unique identifier (an "instance id", or `iid`). For our implementation, this identifier will simply be defined as an integer that is incremented each time the constructor is invoked.[32] This instance id could be used as a key for a data variable that stores data for each instance. Upon instantiation, the instance id could be assigned to the new instance as a property (we'll worry about methods of "encapsulating" this property later).

```
1   var Stack = ( function()
2   {
3       var idata = [],
4           iid   = 0;
5
6       var S = function()
7       {
8           // assign a unique instance identifier
9           // to each instance
10          this.__iid = iid++;
11
12          idata[ this.__iid ] = {
13              stack: []
14          };
15      };
16
17      S.prototype = {
18          push: function( val )
19          {
20              idata[ this.__iid ]
21                  .stack.push( val );
22          },
23
24          pop: function()
25          {
26              return idata[ this.__iid ]
27                  .stack.pop();
28          }
29      };
```

[30] In classical terms, `asPrototype()` can be thought of as a static factory method of `Foo`.

[31] Self-executing functions are most often used to introduce scope, allowing for the encapsulation of certain data. In this case, we encapsulate our extension logic and return our constructor (assigned to `F` within the self-executing function), which is then assigned to `Foo`. Note the parenthesis immediately following the anonymous function, which invokes it with a single argument to give `extending` a default value of `false`. This pattern of encapsulation and exporting specific values is commonly referred to as the *Module Pattern*.

[32] There is, of course, a maximum number of instances with this implementation. Once `iid` reaches `Number.MAX_NUMBER`, its next assignment will cause it to overflow to `Number.POSITIVE_INFINITY`. This number, however, can be rather large. On one 64-bit system under v8, `Number.MAX_NUMBER = 1.7976931348623157e+308`.

www.dbooks.org

```
30
31      return S;
32  } )();
33
34  var first  = new Stack(),
35      second = new Stack();
36
37  first.push( "foo" );
38  second.push( "bar" );
39
40  first.pop();  // "foo"
41  second.pop(); // "bar"
```

Listing 13: Encapsulating data with shared members (a naive implementation)

Listing 13 demonstrates a possible stack implementation using the principals that have just been described. Just like Listing 12, a self-executing function is used to encapsulate our data and returns the `Stack` constructor.[33] In addition to the instance id, the instance data is stored in the array `idata` (an array is appropriate here since `iid` is sequential and numeric). `idata` will store an object for each instance, each acting in place of `this`. Upon instantiation, the private properties for the new instance are initialized using the newly assigned instance id.

Because `idata` is not encapsulated within the constructor, we do not need to use the concept of privileged members (see section 1.2); we need only define the methods in such a way that `idata` is still within scope. Fortunately, this allows us to define the methods on the prototype, saving us method redeclarations with each call to the constructor, improving overall performance.

This implementation comes at the expense of brevity and creates a diversion from common ECMAScript convention when accessing data for a particular instance using prototypes. Rather than having ECMAScript handle this lookup process for us, we must do so manually. The only data stored on the instance itself (bound to `this`) is the instance id, `iid`, which is used to look up the actual members from `idata`. Indeed, this is the first concern — this is a considerable amount of boilerplate code to create separately for each prototype wishing to encapsulate data in this manner.

An astute reader may raise concern over our `__iid` assignment on each instance. Firstly, although this name clearly states "do not touch" with its double-underscore prefix,[34] the member is still public and enumerable.[35] There is no reason why we should be advertising this internal data to the world. Secondly, imagine what may happen if a user decides to alter the value of `__iid` for a given instance. Although such a modification would create some fascinating (or even "cool") features, it could also wreak havoc on a system and break encapsulation.[36]

In environments supporting ECMAScript 5 and later, we can make the property non-enumerable and read-only using `Object.defineProperty()` in place of the `__iid` assignment:

```
Object.defineProperty( this, '__iid', {
    value: iid++,

    writable:     false,
    enumerable:   false,
    configurable: false
} );
```

The `configurable` property simply determines whether or not we can re-configure a property in the future using `Object.defineProperty()`. It should also be noted that each of the properties, with the exception of `value`, default to `false`, so they may be omitted; they were included here for clarity.

Of course, this solution leaves a couple loose ends: it will work only on ECMAScript 5 and later environments (that have support for `Object.defineProperty()`) and it still does not prevent someone from spying on the instance id should they know the name of the property (`__iid`) ahead of time. However, we do need the instance id to be a member of the instance itself for our lookup process to work properly.

At this point, many developers would draw the line and call the solution satisfactory. An internal id, although unencapsulated, provides little room for exploitation.[37] For the sake of discussion and the development of a more concrete implementation, let us consider a potential workaround for this issue.

For pre-ES5[38] environments, there will be no concrete solution, since all properties will always be enumerable. However, we can make it more difficult by randomizing the name of the `__iid` property, which would require that the user filter out all known properties or guess at the name. In ES5+ environments, this would effectively eliminate the problem entirely,[39] since the property name cannot be discovered or be known beforehand. Consider, then, the addition of another variable within the self-executing function — `iid_name` — which we could set to some random value (the implementation of which we will leave to the reader). Then, when initializing or accessing values, one would use the syntax:

[33] The reader should take note that we have omitted our extensible constructor solution discussed in section 2.1 for the sake of brevity.

[34] Certain languages used double-underscore to indicate something internal to the language or system. This also ensures the name will not conflict with any private members that use the single-underscore prefix convention.

[35] The term *enumerable* simply means that it can be returned by `foreach`.

[36] Consider that we know a stack is encapsulated within another object. We could exploit this `__iid` vulnerability to gain access to the data of that encapsulated object as follows, guessing or otherwise calculating the proper instance id: `( new Stack() ).__iid = iid_of_encapsulated_stack_instance`.

[37] You could get a total count of the number of instances of a particular prototype, but not much else.

[38] Hereinafter, ECMAScript 5 and ES5 will be used interchangably.

[39] Of course, debuggers are always an option. There is also the possibility of exploiting weaknesses in a random name implementation; one can never completely eliminate the issue.

```
idata[ this[ iid_name ] ].stack // ...
```

Of course, this introduces additional overhead, although it is likely to be negligible in comparison with the rest of the software.

With that, we have contrived a solution to our encapsulation problem. Unfortunately, as the title of this section indicates, this implementation is naive to a very important consideration — memory consumption. The problem is indeed so severe that this solution cannot possibly be recommended in practice, although the core concepts have been an excellent experiment in ingenuity and have provided a strong foundation on which to expand.[40]

2.2.2 A Proper Implementation

Section 2.2.1 proposed an implementation that would permit the true encapsulation of instance data, addressing the performance issues demonstrated in Listing 7. Unfortunately, the solution offered in Listing 13 is prone to terrible memory leaks. In order to understand why, we must first understand, on a very basic level, how garbage collection (GC) is commonly implemented in environments that support ECMAScript.

Garbage collection refers to an automatic cleaning of data (and subsequent freeing of memory, details of which vary between implementations) that is no longer "used". Rather than languages like C that require manual allocation and freeing of memory, the various engines that implement ECMAScript handle this process for us, allowing the developer to focus on the task at hand rather than developing a memory management system for each piece of software. Garbage collection can be a wonderful thing in most circumstances, but one must understand how it recognizes data that is no longer being "used" in order to ensure that the memory is properly freed. If data lingers in memory in such a way that the software will not access it again and that the garbage collector is not aware that the data can be freed, this is referred to as a *memory leak*.[41]

One method employed by garbage collectors is reference counting; when an object is initially created, the reference count is set to one. When a reference to that object is stored in another variable, that count is incremented by one. When a variable containing a reference to a particular object falls out of scope, is deleted, or has the value reassigned, the reference count is decremented by one. Once the reference count reaches zero, it is scheduled for garbage collection.[42] The concept is simple, but is complicated by the use of closures. When an object is referenced within a closure, or even has the *potential* to be referenced through another object, it cannot be garbage collected.

In the case of Listing 13, consider `idata`. With each new instance, `iid` is incremented and an associated entry added to `idata`. The problem is — ECMAScript does not have destructor support. Since we cannot tell when our object is GC'd, we cannot free the `idata` entry. Because each and every object within `idata` has the *potential* to be referenced at some point in the future, even though our implementation does not allow for it, it cannot be garbage collected. The reference count for each index of `idata` will forever be ≥ 1.

To resolve this issue without altering this implementation, there is only one solution — to offer a method to call to manually mark the object as destroyed. This defeats the purpose of garbage collection and is an unacceptable solution. Therefore, our naive implementation contains a fatal design flaw. This extends naturally into another question — how do we work with garbage collection to automatically free the data for us?

The answer to this question is already known from nearly all of our prior prototype examples. Unfortunately, it is an answer that we have been attempting to work around in order to enforce encapsulation — storing the data on the instance itself. By doing so, the data is automatically freed (if the reference count is zero, of course) when the instance itself is freed. Indeed, we have hit a wall due to our inability to explicitly tell the garbage collector when the data should be freed.[43] The solution is to find a common ground between Listing 7 and Listing 13.

Recall our original goal — to shy away from the negative performance impact of privileged members without exposing each of our private members as public. Our discussion has already revealed that we are forced to store our data on the instance itself to ensure that it will be properly freed by the garbage collector once the reference count reaches zero. Recall that section 2.2.1 provided us with a number of means of making our only public member, `__iid`, considerably more difficult to access, even though it was not fully encapsulated. This same concept can be applied to our instance data.

```
1  var Stack = ( function()
2  {
3      // implementation left to reader
4      var _privname = genRandomName();
5
6      var S = function()
7      {
8          Object.defineProperty( this, _privname, {
9              enumerable:   false,
10             writable:     false,
11             configurable: false,
12
13             value: {
14                 stack: []
```

[40] It is my hope that the title of this section will help to encourage those readers that simply skim for code to continue reading and consider the flaws of the design rather than adopting them.

[41] The term "memory leak" implies different details depending on context — in this case, it varies between languages. A memory leak in C is handled much differently than a memory leak in ECMAScript environments. Indeed, memory leaks in systems with garbage collectors could also be caused by bugs in the GC process itself, although this is not the case here.

[42] What happens after this point is implementation-defined.

[43] There may be an implementation out there somewhere that does allow this, or a library that can interface with the garbage collector. However, it would not be portable.

www.dbooks.org

```
15              }
16          } );
17      };
18
19      S.prototype = {
20          push: function( val )
21          {
22              this[ _privname ].stack.push( val );
23          },
24
25          pop: function()
26          {
27              return this[ _privname ].stack.pop();
28          }
29      };
30
31      return S;
32  } )();
```

Listing 14: Encapsulating data on the instance itself (see also Listing 13)

Listing 14 uses a random, non-enumerable property to make the discovery of the private data considerably more difficult.[44] The random name, `_privname`, is used in each of the prototypes to look up the data on the appropriate instance (e.g. `this[ _privname ].stack` in place of `this.stack`).[45] This has the same effect as Listing 13, with the exception that it is a bit easier to follow without the instance management code and that it does not suffer from memory leaks due to GC issues.

Of course, this implementation depends on features introduced in ECMAScript 5 — namely, `Object.defineProperty()`, as introduced in section 2.2.1. In order to support pre-ES5 environments, we could define our own fallback `defineProperty()` method by directly altering `Object`,[46] as demonstrated in Listing 15.

```
1  Object.defineProperty = Object.defineProperty
2      || function( obj, name, config )
3      {
4          obj[ name ] = config.value;
5      };
```

Listing 15: A fallback `Object.defineProperty()` implementation

Unfortunately, a fallback implementation is not quite so simple. Certain dialects may only partially implement `Object.createProperty()`. In particular, I am referring to Internet Explorer 8's incomplete implementation.[47] Surprisingly, IE8 only supports this action on DOM elements, not all objects. This puts us in a terribly awkward situation — the method is defined, but the implementation is "broken". As such, our simple and fairly concise solution in Listing 15 is insufficient. Instead, we need to perform a more complicated check to ensure that not only is the method defined, but also functional for our particular uses. This check is demonstrated in Listing 16, resulting in a boolean value which can be used to determine whether or not the fallback in Listing 15 is necessary.

```
1  var can_define_prop = ( function()
2  {
3      try
4      {
5          Object.defineProperty( {}, 'x', {} );
6      }
7      catch ( e ) { return false; }
8
9      return true;
10 } )();
```

Listing 16: Working around IE8's incomplete `Object.defineProperty()` implementation (taken from ease.js)

This function performs two checks simultaneously — it first checks to see if `Object.defineProperty()` exists and then ensures that we are not using IE8's broken implementation. If the invocation fails, that will mean that the method does not exist (or is not properly defined), throwing an exception which will immediately return false. If attempting to define a property using this method on a non-DOM object in IE8, an exception will also be thrown, returning false. Therefore, we can simply attempt to define a property on an empty object. If this action succeeds, then `Object.defineProperty()` is assumed to be sufficiently supported. The entire process is enclosed in a self-executing function to ensure that the check is performed only once, rather than a function that performs the check each time it is called. The merriment of this result to Listing 15 is trivial and is left to the reader.

It is clear from this fallback, however, that our property is enumerable in pre-ES5 environments. At this point, a random property name would not be all that helpful and the reader may decide to avoid the random implementation in its entirety.

2.2.3 Private Methods

Thus far, we have been dealing almost exclusively with the issue of encapsulating properties. Let us now shift our focus to the encapsulation of other private members, namely methods (although this could just as easily be applied to getters/setters in ES5+ environments). Private methods are actually considerably easier to conceptualize, because the data does not vary between instances — a method is a method and is shared between all instances. As such, we do not have to worry about the memory management

[44] The property is also read-only, but that does not necessarily aid encapsulation. It prevents the object itself from being reassigned, but not any of its members.

[45] One may decide that the random name is unnecessary overhead. However, note that static names would permit looking up the data if the name is known ahead of time.

[46] The only circumstance I ever recommend modifying built-in bojects/prototypes is to aid in backward compatibility; it is otherwise a very poor practice that creates tightly coupled, unportable code.

[47] IE8's dialect is JScript.

issues addressed in section 2.2.2.

Encapsulating private members would simply imply moving the members outside of the public prototype (that is, `Stack.prototype`). One would conventionally implement private methods using privileged members (as in section 1.2), but it is certainly pointless redefining the methods for each instance, since Listing 14 provided us with a means of accessing private data from within the public prototype. Since the self-executing function introduces scope for our private data (instead of the constructor), we do not need to redefine the methods for each new instance. Instead, we can create what can be considered a second, private prototype.

```
1  var Stack = ( function()
2  {
3      var _privname = getRandomName();
4
5      var S = function()
6      {
7          // ... (see previous examples)
8      };
9
10     var priv_methods = {
11         getStack: function()
12         {
13             return this[ _privname ].stack;
14         }
15     };
16
17     S.prototype = {
18         push: function( val )
19         {
20             var stack = priv_methods.getStack
21                 .call( this );
22             stack.push( val );
23         },
24
25         pop: function()
26         {
27             var stack = priv_methods.getStack
28                 .call( this )
29             return stack.pop( val );
30         }
31     };
32
33     return S;
34 } )();
```

Listing 17: Implementing shared private methods without privileged members

Listing 17 illustrates this concept of a private prototype.[48] The object `priv_methods` acts as a second prototype containing all members that are private and shared between all instances, much like the conventional prototype. `Stack.prototype` then includes only the members that are intended to be public. In this case, we have defined a single private method — `getStack()`.

Recall how `this` is bound automatically for prototype methods (see section 1.1). ECMAScript is able to do this for us because of the standardized `prototype` property. For our private methods, we have no such luxury. Therefore, we are required to bind `this` to the proper object ourselves through the use of `Function.call()` (or, alternatively, `Function.apply()`). The first argument passed to `call()` is the object to which `this` will be bound, which we will refer to as the *context*. This, unfortunately, increases the verbosity of private method calls, but successfully provides us with a private prototype implementation.

Since private members needn't be inherited by subtypes, no additional work needs to be done.

2.3 Protected Members

We have thus far covered two of the three access modifiers (see section 2.2) — public and private. Those implementations allowed us to remain blissfully ignorant of inheritance, as public members are handled automatically by ECMAScript and private members are never inherited by subtypes. The concept of protected members is a bit more of an interesting study since it requires that we put thought into providing subtypes with access to encapsulated data, *without* exposing this data to the rest of the world.

From an implementation perspective, we can think of protected members much like private; they cannot be part of the public prototype, so they must exist in their own protected prototype and protected instance object. The only difference here is that we need a way to expose this data to subtypes. This is an issue complicated by our random name implementation (see section 2.2.2); without it, subtypes would be able to access protected members of its parent simply by accessing a standardized property name. The problem with that is — if subtypes can do it, so can other, completely unrelated objects. As such, we will focus on a solution that works in conjunction with our randomized name (an implementation with a standardized name is trivial).

In order for the data to remain encapsulated, the name must too remain encapsulated. This means that the subtype cannot request the name from the parent; instead, we must either have access to the random name or we must *tell* the parent what the name should be. The latter will not work per-instance with the implementation described in section 2.2.2, as the methods are not redefined per-instance and therefore must share a common name. Let us therefore first consider the simpler of options — sharing a common protected name between the two classes.

```
1  var _protname = getRandomName();
2
3  var Stack = ( function()
4  {
5      var _privname = getRandomName();
```

[48] Alternatively, to reduce code at the expense of clarity, one could simply define functions within the closure to act as private methods without assigning them to `priv_methods`. Note that `call()` is still necessary in that situation.

www.dbooks.org

```
    var S = function()
    {
        // ... (see previous examples)

        Object.defineProperty( this, _privname, {
            value: { stack: [] }
        } );

        Object.defineProperty( this, _protname, {
            value: { empty: false }
        } );
    };

    // a means of sharing protected methods
    Object.defineProperty( S, _protname, {
        getStack: function()
        {
            return this[ _privname ].stack;
        }
    } );

    S.prototype = {
        push: function( val )
        {
            var stack = S[ _protname ].getStack
                .call( this );
            stack.push( val );

            this[ _protname ].empty = false;
        },

        pop: function()
        {
            var stack = this[ _protname ]
                .getStack.call( this )

            this[ _protname ].empty =
                ( stack.length === 0 );

            return stack.pop( val );
        }
    };

    S.asPrototype = function()
    {
        // ... (see previous examples)
    };

    return S;
} )();


var MaxStack = ( function()
{
    var M = function( max )
    {
        // call parent constructor
        Stack.call( this );

        // we could add to our protected members
        // (in practice, this would be private, not
        // protected)
        this[ _protname ].max = +max;
    };

    // override push
    M.prototype.push = function( val )
    {
        var stack = Stack[ _protname ].getStack
            .call( this );

        if ( stack.length ===
            this[ _protname ].max
        )
        {
            throw Error( "Maximum reached." );
        };

        // call parent method
        Stack.prototype.push.call( this, val );
    };

    // add a new method demonstrating parent
    // protected property access
    M.prototype.isEmpty = function()
    {
        return this[ _protname ].empty;
    };

    M.prototype             = Stack.asPrototype();
    M.prototype.constructor = M;

    return M;
} )();


var max = new MaxStack( 2 );
max.push( "foo" );
max.push( "bar" );
max.push( "baz" ); // Error
max.pop();         // "bar"
max.pop();         // "foo"
```

Listing 18: Sharing protected members with subtypes

Listing 18 makes an attempt to demonstrate a protected property and method implementation while still maintaining the distinction between it and the private member implementation (see section 2.2.2). The example contains two separate constructors — `Stack` and `MaxStack`, the latter of which extends `Stack` to limit the number of items that may be pushed to it. `Stack` has been modified to include a protected property `empty`, which will be set to `true` when the stack contains no items, and a protected method `getStack()`, which both `Stack` and its subtype `MaxStack` may use to access the private property `stack` of `Stack`.

The key part of this implementation is the declaration of `_protname` within the scope of both types (`Stack` and `MaxStack`).[49] This declaration allows both prototypes to

[49] One would be wise to enclose all of Listing 18 within a function to prevent `_protname` from being used elsewhere, exporting `Stack` and `MaxStack` however the reader decides.

access the protected properties just as we would the private data. Note that `_privname` is still defined individually within each type, as this data is unique to each.

Protected methods, however, need additional consideration. Private methods, when defined within the self-executing function that returns the constructor, work fine when called from within the associated prototype (see section 2.2.3). However, since they're completely encapsulated, we cannot use the same concept for protected methods — the subtype would not have access to the methods. Our two options are to either declare the protected members outside of the self-executing function (as we do `_privname`), which makes little organizational sense, or to define the protected members on the constructor itself using `_protname` and `Object.defineProperty()`[50] to encapsulate it the best we can. We can then use the shared `_protname` to access the methods on `Stack`, unknown to the rest of the world.

An astute reader may realize that Listing 18 does not permit the addition of protected methods without also modifying the protected methods of the supertype and all other subtypes; this is the same reason we assign new instances of constructors to the `prototype` property. Additionally, accessing a protected method further requires referencing the same constructor on which it was defined. Fixing this implementation is left as an exercise to the reader.

Of course, there is another glaring problem with this implementation — what happens if we wish to extend one of our prototypes, but are not within the scope of `_protname` (which would be the case if you are using Listing 18 as a library, for example)? With this implementation, that is not possible. As such, Listing 18 is not recommended unless you intended to have your prototypes act like final classes.[51] As this will not always be the case, we must put additional thought into the development of a solution that allows extending class-like objects with protected members outside of the scope of the protected name `_protname`.

As we already discussed, we cannot request the protected member name from the parent, as that will provide a means to exploit the implementation and gain access to the protected members, thereby breaking encapsulation. Another aforementioned option was *telling* the parent what protected member name to use, perhaps through the use of `asPrototype()` (see section 2.1). This is an option for protected *properties*, as they are initialized with each new instance, however it is not a clean implementation for *members*, as they have already been defined on the constructor with the existing `_protname`. Passing an alternative name would result in something akin to:

```
Object.defineProperty( S, _newname, {
    value: S[ _protname ]
} );
```

This would quickly accumulate many separate protected member references on the constructor — one for each subtype. As such, this implementation is also left as an exercise for an interested reader; we will not explore it further.[52]

The second option is to avoid exposing protected property names entirely. This can be done by defining a function that can expose the protected method object. This method would use a system-wide protected member name to determine what objects to return, but would never expose the name — only the object references. However, this does little to help us with our protected properties, as a reference to that object cannot be returned until instantiation. As such, one could use a partial implementation of the previously suggested implementation in which one provides the protected member name to the parent(s). Since the protected members would be returned, the duplicate reference issue will be averted.

The simplest means of demonstrating this concept is to define a function that accepts a callback to be invoked with the protected method object. A more elegant implementation will be described in future sections, so a full implementation is also left as an exercise to the reader. Listing 19 illustrates a skeleton implementation.[53] The `def` function accepts the aforementioned callback with an optional first argument — `base` — from which to retrieve the protected methods.

```
1  var def = ( function()
2  {
3      var _protname = getRandomName();
4
5      return function( base, callback )
6      {
7          var args = Array.prototype.slice.call(
8                  arguments
9              ),
10
11             callback = args.pop(),
12             base     = args.pop() || {};
13
14         return callback( base[ _protname ] );
15     };
16 } )();
17
18 var Stack = def( function( protm )
19 {
20     // ...
21
22     return S;
23 } );
24
25 var MaxStack = def( Stack, function( protm )
26 {
```

[50] See section 2.2.2 for `Object.defineProperty()` workarounds/-considerations.

[51] A *final* class cannot be extended.

[52] The reader is encouraged to attempt this implementation to gain a better understanding of the concept. However, the author cannot recommend its use in a production environment.

[53] Should the reader decide to take up this exercise, keep in mind that the implementation should also work with multiple supertypes (that is, type3 extends type2 extends type1).

www.dbooks.org

```
27      // for properties only
28      var _protname = getRandomName();
29
30      // ...
31
32      // asPrototype() would accept the protected
33      // member name
34      M.protoype = S.asPrototype( _protname );
35      M.prototype.constructor = M;
36
37      return M;
38  } );
```

Listing 19: Exposing protected methods with a callback (brief illustration; full implementation left as an exercise for the reader)

2.3.1 Protected Member Encapsulation Challenges

Unfortunately, the aforementioned implementations do not change a simple fact — protected members are open to exploitation, unless the prototype containing them cannot be extended outside of the library/implementation. Specifically, there is nothing to prevent a user from extending the prototype and defining a property or method to return the encapsulated members.

Consider the implementation described in Listing 19. We could define another subtype, `ExploitedStack`, as shown in Listing 20. This malicious type exploits our implementation by defining two methods — `getProtectedProps()` and `getProtectedMethods()` — that return the otherwise encapsulated data.

```
1   var ExploitedStack = def( Stack, function( protm )
2   {
3       var _protname = getRandomName();
4
5       var E = function() { /* ... */ };
6
7       E.prototype.getProtectedProps = function()
8       {
9           return this[ _protname ];
10      }:
11
12      E.prototype.getProtectedMethods = function()
13      {
14          return protm;
15      };
16
17      E.prototype = Stack.asPrototype( _protname );
18      E.prototype.constructor = E;
19
20      return E;
21  } )();
```

Listing 20: Exploiting Listing 19 by returning protected members.

Fortunately, our random `_protname` implementation will only permit returning data for the protected members of that particular instance. Had we not used random names, there is a chance that an object could be passed to `getProtectedProps()` and have its protected properties returned.[54] As such, this property exploit is minimal and would only hurt that particular instance. There could be an issue if supertypes contain sensitive protected data, but this is an implementation issue (sensitive data should instead be private).

Methods, however, are a more considerable issue. Since the object exposed via `def()` is *shared* between each of the instances, much like its parent prototype is, it can be used to exploit each and every instance (even if the reader has amended Listing 18 to resolve the aforementioned protected member addition bug, since `Object.getPrototypeOf()` can be used to work around this amendment). Someone could, for example, reassign `Stack[ _protname ].getStack()` to do something else; `Object.defineProperty()` in Listing 18 only made `Stack[ _protname ]` *itself* read-only. The object itself, however, can be modified. This can be amended by using `Object.defineProperty()` for each and every protected method, which is highly verbose and cumbersome.

Once we rule out the ability to modify protected method definitions,[55] we still must deal with the issue of having our protected methods exposed and callable. For example, one could do the following to gain access to the private `stack` object:

```
( new ExploitedStack() ).getProtectedMethods()
    .getStack.call( some_stack_instance );
```

Unfortunately, there is little we can do about this type of exploit besides either binding[56] each method call (which would introduce additional overhead per instance) or entirely preventing the extension of our prototypes outside of our own library/software. By creating a protected API, you are exposing certain aspects of your prototype to the rest of the world; this likely breaks encapsulation and, in itself, is often considered a poor practice.[57] An alternative is to avoid inheritance altogether and instead favor composition, thereby evading this issue entirely. That is a pretty attractive concept, considering how verbose and messy this protected hack has been.

3 Encapsulating the Hacks

Imagine jumping into a project in order to make a simple modification and then seeing the code in Listing 18. This is a far cry from the simple protected member declarations in traditional classical object-oriented languages. In fact,

[54] Details depend on implementation. If a global protected property name is used, this is trivial. Otherwise, it could be circumstantial — a matching name would have to be guessed, known, or happen by chance.

[55] Of course, this doesn't secure the members in pre-ES5 environments.

[56] See `Function.bind()`.

[57] `Stack.getStack()` breaks encapsulation because it exposes the private member `stack` to subtypes.

there becomes a point where the hacks discussed in the previous sections become unmaintainable messes that add a great deal of boilerplate code with little use other than to distract from the actual software itself.

However, we do not have to settle for those messy implementations. Indeed, we can come up with some fairly elegant and concise solutions by encapsulating the hacks we have discussed into a classical object-oriented framework, library or simple helper functions. Let's not get ahead of ourselves too quickly; we will start exploring basic helper functions before we deal with diving into a full, reusable framework.

This section is intended for educational and experimental purposes. Before using these examples to develop your own class system for ECMAScript, ensure that none of the existing systems satisfy your needs; your effort is best suited toward the advancement of existing projects than the segregation caused by the introduction of additional, specialty frameworks.[58] These are discussed a bit later.

3.1 Constructor/Prototype Factory

Section 2.1 offered one solution to the problem of creating an extensible constructor, allowing it to be used both to instantiate new objects and as a prototype. Unfortunately, as Listing 12 demonstrated, the solution adds a bit of noise to the definition that will also be duplicated for each constructor. The section ended with the promise of a cleaner, reusable implementation. Perhaps we can provide that.

Consider once again the issue at hand. The constructor, when called conventionally with the `new` operator to create a new instance, must perform all of its construction logic. However, if we wish to use it as a prototype, it is unlikely that we want to run *any* of that logic — we are simply looking to have an object containing each of its members to use as a prototype without the risk of modifying the prototype of the constructor in question. Now consider how this issue is handled in other classical languages: the `extend` keyword.

ECMAScript has no such keyword, so we will have to work on an implementation ourselves. We cannot use the name `extend()`, as it is a reserved name;[59] as such, we will start with a simple `Class` factory function with which we can create new "classes" without supertypes. We can than provide a `Class.extend()` method to define a "class" *with* a supertype.

```
var Class = ( function( extending )
{
    var C = function( dfn )
    {
        // extend from an empty base
        return C.extend( null, dfn );
    };

    C.extend = function( base, dfn )
    {
        base = base || function() {};

        // prevent ctor invocation
        extending = true;

        var ctor = function()
        {
            // do nothing if extending
            if ( extending )
            {
                return;
            }

            // call "actual" constructor
            this.__construct &&
                this.__construct.apply(
                    this, arguments
                );
        };

        ctor.prototype = new base();
        ctor.prototype.constructor = ctor;

        copyTo( ctor.prototype, dfn );

        // done extending; clear flag to
        // ensure ctor can be invoked
        extending = false;

        return ctor;
    };

    function copyTo( dest, members )
    {
        var hasOwn = Object.prototype
            .hasOwnProperty;

        for ( var member in members )
        {
            if ( !hasOwn.call( members, member ) )
            {
                continue;
            }

            dest[ member ] = members[ member ];
        }
    }

    return C;
} )( false );
```

Listing 21: Constructor factory

Listing 21 demonstrates one such possible implementation of a constructor factory. Rather than thinking of "creating a class" and "creating a class with a supertype" as two separate processes, it is helpful to consider them one and the same; instead, we can consider the former to

[58] That is not to discourage experimentation. Indeed, one of the best, most exciting and fun ways to learn about these concepts are to implement them yourself.

[59] Perhaps for future versions of ECMAScript.

www.dbooks.org

be "creating a class *with an empty supertype*". As such, invoking `Class()` simply calls `Class.extend()` with `null` for the base (on line 6), allowing `Class.extend()` to handle the creation of a new constructor without a supertype.

Both `Class()` and `Class.extend()` accept a `dfn` argument, which we will refer to as the *definition object*; this object is to contain each member that will appear on the prototype of the new constructor. The `base` parameter, defined on `Class.extend()`, denotes the constructor from which to extend (the constructor that will be instantiated and used as the prototype). Line 11 will default `base` to an empty function if one has not been provided (mainly, to satisfy the `Class()` call on line 6).

With that, we can now continue onto creating our constructor, beginning on line 16. Section 2.1 introduced the concept of using an `extending` flag to let the constructor know when to avoid all of its construction logic if being used only as a prototype (see Listing 12). The problem with this implementation, as discussed, was that it required that *each* constructor that wishes to use this pattern implement it themselves, violating the DRY[60] principle. There were two main areas of code duplication in Listing 12 — the checking of the `extending` flag in the constructor and the setting (and resetting) of the flag in `F.asPrototype()`. In fact, we can eliminate the `asPrototype()` method altogether once we recognize that its entire purpose is to set the flags before and after instantiation.

To address the first code duplication issue — the checking of the flag in the constructor — we must remove the need to perform the check manually for each and every constructor. The solution, as demonstrated in Listing 21, is to separate our generic constructor logic (shared between all constructors that use the factory) from the logic that can vary between each constructor. `ctor` on line 16 accomplishes this by first performing the `extending` check (lines 19–22) and then forwarding all arguments to a separate function (`__construct()`), if defined, using `Function.apply()` (lines 25–28). One could adopt any name for the constructor method; it is not significant.[61] Note that the first argument to `Function.apply()` is important, as it will ensure that `this` is properly bound within the `__construct()` method.

To address the second code duplication issue and remove the need for `asPrototype()` in Listing 12 entirely, we can take advantage of the implications of `Class.extend()` in Listing 21. The only time we wish to use a constructor as a prototype and skip `__construct()` is during the process of creating a new constructor. As such, we can simply set the `extending` flag to `true` when we begin creating the new constructor (see line 14, though this flag could be placed anywhere before line 31) and then reset it to `false` once we are done (line 38). With that, we have eliminated the code duplication issues associated with Listing 12.

The remainder of Listing 21 is simply an abstraction around the manual process we have been performing since section 1.1 — setting the prototype, properly setting the constructor and extending the prototype with our own methods. Recall section 2.3 in which we had to manually assign each member of the prototype for subtypes in order to ensure that we did not overwrite the existing prototype members (e.g. `M.prototype.push()` in Listing 18). The very same issue applies here: Line 31 first sets the prototype to an instance of `base`. If we were to then set `ctor.prototype = dfn`, we would entirely overwrite the benefit gained from specifying `base`. In order to automate this manual assignment of each additional prototype member of `dfn`, `copyTo()` is provided, which accepts two arguments — a destination object `dest` to which each given member of `members` should be copied (defined on line 43 and called on line 34).

Like the examples provided in section 2, we use a self-executing function to hide the implementation details of our `Class` function from the rest of the world.

To demonstrate use of the constructor factory, Listing 22 defines two classes[62] — `Foo` and `SubFoo`. Note that how, by placing the curly braces on their own line, we can create the illusion that `Class()` is a language construct:

```
61 var Foo = Class(
62 {
63     __construct: function( name )
64     {
65         if ( !name )
66         {
67             throw TypeError( "Name required" );
68         }
69
70         this._name = ''+( name );
71     },
72
73     getName: function()
74     {
75         return this._name;
76     }
77 } );
78
79 var SubFoo = Class.extend( Foo,
80 {
81     setName: function( name )
82     {
83         this._name = ''+( name );
84     }
85 } );
86
87
88 var myfoo = new Foo( "Foo" ),
89     mysub = new SubFoo( "SubFoo" );
90
91 myfoo.getName(); // "Foo"
92 mysub.getName(); // "SubFoo"
```

[60] "Don't repreat yourself", *The Pragmatic Programmer*.

[61] The `__construct` name was taken from PHP.

[62] The reader should take care in noting that the term "class", as used henceforth, will refer to a class-like object created using the systems defined within this article. ECMAScript does not support classes, so the use of the term "class" in any other context is misleading.

```
93
94  mysub.setName( "New Name" );
95  mysub.getName(); // "New Name"
96
97  // parent Foo does not define setName()
98  myfoo.setName( "Error" ); // TypeError
99
100 // our name will be required, since we
101 // are not extending
102 new Foo(); // TypeError
```

Listing 22: Demonstrating the constructor factory

The reader should note that an important assertion has been omitted for brevity in Listing 21. Consider, for example, what may happen in the case of the following:

```
Class.extend( "foo", {} );
```

It is apparent that `"foo"` is not a function and therefore cannot be used with the `new` operator. Given that, consider line 31, which blindly invokes `base()` without consideration for the very probable scenario that the user mistakenly (due to their own unfamiliarity or a simple bug) provided us with a non-constructor for `base`. The user would then be presented with a valid, but not necessarily useful error — did the error occur because of user error, or due to a bug in the factory implementation?

To avoid confusion, it would be best to perform a simple assertion before invoking `base` (or wrap the invocation in a try/catch block, although doing so is not recommended in case `base()` throws an error of its own):

```
if ( typeof base !== 'function' )
{
    throw TypeError( "Invalid base provided" );
}
```

Note also that, although this implementation will work with any constructor as `base`, only those created with `Class()` will have the benefit of being able to check the `extending` flag. As such, when using `Class.extend()` with third-party constructors, the issue of extensible constructors may still remain and is left instead in the hands of the developer of that base constructor.

3.1.1 Factory Conveniences

Although our constructor factory described in section 3.1 is thus far very simple, one should take the time to realize what a powerful abstraction has been created: it allows us to inject our own code in any part of the constructor creation process, giving us full control over our class-like objects. Indeed, this abstraction will be used as a strong foundation going forward throughout all of section 2.2. In the meantime, we can take advantage of it in its infancy to provide a couple additional conveniences.

First, consider the syntax of `Class.extend()` in Listing 21. It requires the extending of a constructor to be done in the following manner:

```
var SubFoo = Class.extend( Foo, {} );
```

Would it not be more intuitive to instead be able to extend a constructor in the following manner?

```
var SubFoo = Foo.extend( {} );
```

The above two statements are semantically equivalent — they define a subtype `SubFoo` that extends from the constructor `Foo` — but the latter example is more concise and natural. Adding support for this method is trivial, involving only a slight addition to Listing 3.1's `C.extend()` method, perhaps around line 30:

```
31          ctor.extend = function( dfn )
32          {
33              C.extend.call( this, dfn );
34          };
```

Listing 23: Adding a static `extend()` method to constructors

Of course, one should be aware that this implementation is exploitable in that, for example, `Foo.extend()` could be reassigned at any point. As such, using `Class.extend()` is the safe implementation, unless you can be certain that such a reassignment is not possible. Alternatively, in ECMAScript 5 and later environments, one can use `Object.defineProperty()`, as discussed in sections 2.2.1 and 2.2.2, to make the method read-only.

Now consider the instantiation of our class-like objects, as was demonstrated in Listing 22:

```
var inst = new Foo( "Name" );
```

We can make our code even more concise by eliminating the `new` operator entirely, allowing us to create a new instance as such:

```
var inst = Foo( "Name" );
```

Of course, our constructors do not yet support this, but why may we want such a thing? Firstly, for consistency — the core ECMAScript constructors do not require the use of the keyword, as has been demonstrated throughout this article with the various `Error` types. Secondly, the omission of the keyword would allow us to jump immediately into calling a method on an object without dealing with awkward precedence rules: `Foo( "Name" ).getName()` vs. `( new Foo( "Name" ) ).getName()`. However, those reasons exist more to offer syntactic sugar; they do little to persuade those who do want or not mind the `new` operator.

The stronger argument against the `new` operator is what happens should someone *omit* it, which would not be at all uncommon since the keyword is not required for the core ECMAScript constructors. Recall that `this`, from within the constructor, is bound to the new instance when invoked with the `new` operator. As such, we expect to be able to make assignments to properties of `this` from within the constructor without any problems. What, then, happens if the constructor is invoked *without* the keyword?

www.dbooks.org

`this` would instead be bound (according to the ECMA-Script standard [1]) to "the global object",[63] unless in strict mode. This is dangerous:

```
1  function Foo()
2  {
3      this.Boolean = true;
4  }
5
6  // ok
7  var inst = new Foo();
8  inst.Boolean; // true
9
10 // bad
11 Foo();
12 new Boolean(); // TypeError
```

Listing 24: Introducing unintended global side-effects with constructors

Consider Listing 24 above. Function `Foo()`, if invoked with the `new` operator, results in an object with a `Boolean` property equal to `true`. However, if we were to invoke `Foo()` *without* the `new` operator, this would end up *overwriting the built-in global `Boolean` object reference*. To solve this problem, while at the same time providing the consistency and convenience of being able to either include or omit the `new` operator, we can add a small block of code to our generated constructor `ctor` (somewhere around line 23 of Listing 21, after the extend check but before `__construct()` is invoked):

```
24              if ( !( this instanceof ctor ) )
25              {
26                  return new ctor.apply(
27                      null, arguments
28                  );
29              }
```

Listing 25: Allowing for omission of the `new` operator

The check, as demonstrated in Listing 25, is as simple as ensuring that `this` is properly bound to a *new instance of our constructor `ctor`*. If not, the constructor can simply return a new instance of itself through a recursive call.

Alternatively, the reader may decide to throw an error instead of automatically returning a new instance. This would require the use of the `new` operator for instantiation, while still ensuring that the global scope will not be polluted with unnecessary values. If the constructor is in strict mode, then the pollution of the global scope would not be an issue and the error would instead help to point out inconsistencies in the code. However, for the reason that the keyword is optional for many core ECMAScript constructors, the author recommends the implementation in Listing 25.

[63] In most browser environments, the global object is `window`.

3.2 Private Member Encapsulation

Section 2.2 discussed the encapsulation of private member data by means of private property and method objects, thereby avoiding the performance impact of privileged members (see section 1.2). In order to avoid memory leaks, the private data was stored on the instance itself rather than a truly encapsulated object. The amount of code required for this implementation was relatively small, but it is still repeated unnecessarily between all constructors.

The private member implementation had two distinct pieces — private properties, as demonstrated in Listing 14, and private methods, as demonstrated in Listing 17. This distinction is important, as private methods should not be redefined for each new instance (see Figure 1). Properties, however, *must* have their values copied for each new instance to prevent references from being shared between them (see Listing 2; note that this is not an issue for scalars). For the time being, we will focus on the method implementation and leave the manual declaration of private properties to the `__construct()` method.

The listings in section 2.2 were derived from a simple concept — the private member objects were within the scope of the prototype members. However, if we are to encapsulate this hack within our constructor factory, then the members (the definition object) would be declared *outside* the scope of any private member objects that are hidden within our factory. To expose the private "prototype" object, we could accept a function instead of a definition object, which would expose a reference to the object (as in Listing 19). However, this would be very unnatural and unintuitive; to keep our "class" declarations simple, another method is needed.

Consider the private member concept in a classical sense — the private data should be available only to the methods of the class and should not be accessible outside of them. That is, given any class `C` with private property `C._priv` and public method `C.getPrivValue()`, and an instance `i` of class `C`, `i._priv` should not be defined unless within the context of `i.getPrivValue()`. Consider then the only means of exposing that data to the members of the prototype in ECMAScript without use of closures: through the instance itself (`this`). This naturally derives an implementation that had not been previously considered due to the impracticality of its use without factory — exposing private members before a method invocation and revoking them after the method has returned.

To accomplish this, the factory must be able to intelligently determine when a method is being invoked. This leads us into a somewhat sensitive topic — function wrapping. In order to perform additional logic on invocation of a particular method, it must be wrapped within another function. This *wrapper* would expose the private data on `this`, invoke the original function associated with the method call, remove the reference and then return whatever value was returned by the original function. This

creates the illusion of invoking the method directly.[64]

```
1  function wrap( func )
2  {
3      return function()
4      {
5          return 'one ' +
6              func.apply( this, arguments ) +
7              ' three';
8      };
9  }
10
11 function str( value )
12 {
13     return value;
14 }
15
16 wrap( str )( 'two' );  // "one two three"
```

Listing 26: Wrapping a function by returning a *new* function which calls the original

Listing 26 demonstrates the basic concept of a function wrapper. `wrap()` accepts a single argument, `func`, and returns a new anonymous function which invokes `func`, returning its value with a prefix and a suffix. Note how all arguments are forwarded to `func`, allowing us to invoke our wrapped function as if it were the original. Also note the context in which `func` is being called (the first argument of `apply()`). By binding `this` of `func` to `this` of our wrapper, we are effectively forwarding it. This detail is especially important if we are using a wrapper within a prototype, as we *must* bind `this` to the instance that the method is being invoked upon. Use of `wrap()` with a prototype is demonstrated in Listing 27 below.

```
17 function Foo( value )
18 {
19     this._value = value;
20 };
21
22 Foo.prototype = {
23     bar: wrap( function()
24     {
25         return this._value;
26     } )
27 };
28
29 var inst = new Foo( '2' );
30 inst.bar();  // "one 2 three"
```

Listing 27: Using `wrap()` from Listing 26 with prototypes

It is this concept that will be used to implement method wrapping in our constructor factory. For each function f of definition object D, f' will be created using a method similar to Listing 27. f' will invoke f after setting the private member object on `this`, then reset it after f returns. Finally, the return value of f will be returned by f'. It should be noted that f' must exist even if f is public, since public methods may still need access to private members.[65]

Many readers are likely to be concerned about a decision that wraps every function of our definition object, as this will require two function calls each time a method is invoked. Figure 2a (page 20) shows why this detail is likely to be a concern — invoking our wrapped function is so slow in comparison to invoking the original function directly that the solution seems prohibitive. However, one must consider how functions are *actually* used — to perform some sort of business logic. It is rare that we would invoke bodiless functions continuously in a loop. Rather, we should take into consideration the *percent change between function invocations that contain some sort of business logic*. This is precisely what Figure 2b (page 20) takes into consideration, showing that our invocation worry is would actually be a micro-optimization. For example, in software that performs DOM manipulation, the performance impact of wrapper invocation is likely to be negligible due to repaints being highly intensive operations.

One legitimate concern of our wrapper implementation, however, is limited call stack space. The wrapper will effectively cut the remaining stack space in half if dealing with recursive operations on itself, which may be a problem for environments that do not support tail call optimizations, or for algorithms that are not written in such a way that tail call optimizations can be performed.[66] In such a situation, we can avoid the problem entirely by recommending that heavy recursive algorithms do not invoke wrapped methods; instead, the recursive operation can be performed using "normal" (unwrapped) functions and its result returned by a wrapped method call.

That said, call stack sizes for ECMAScript environments are growing ever larger. Call stack limits for common browsers (including historical versions for comparison) are listed in Figure 3 (page 21). Should this limit be reached, another alternative is to use `setTimeout()` to reset the stack and continue the recursive operation. This can also have the benefit of making the operation asynchronous.

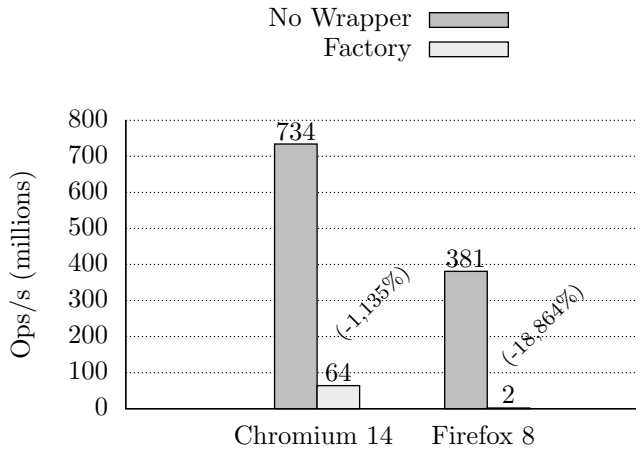
Factoring this logic into the constructor factory is further complicated by our inability to distinguish between members intended to be public and those intended to be private. In section 2.2, this issue was not a concern because the members could be explicitly specified separately per implementation. With the factory, we are provided only a single definition object; asking for multiple would be confusing, messy and unnatural to those coming from other classical object-oriented languages. Therefore, our second task shall be to augment `copyTo()` in Listing 21 to distinguish between public and private members.

[64] This is the same concept used to emulate `Function.bind()` in pre-ECMAScript 5 environments. This concept can also be easily extended to create *partially applied functions*.

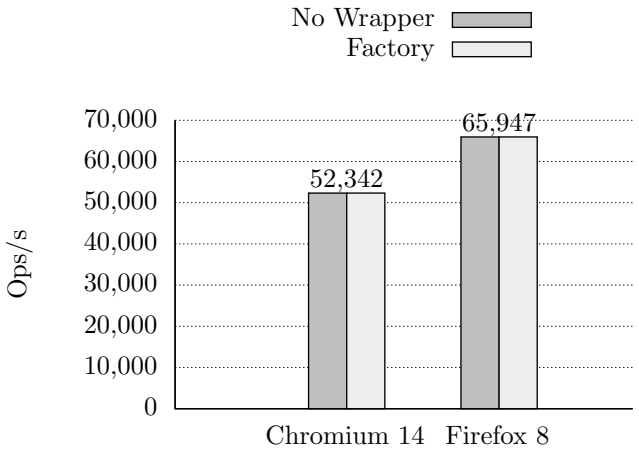
[65] As we will see in the examination of Figure 2, the performance impact of this decision is minimal.

[66] Another concern is that the engine may not be able to perform tail call optimization because the function may recurse on the wrapper instead of itself.

www.dbooks.org



(a) Wrapper performance *(invocation only)*. Operations per second rounded to millions. [3] Numbers in parenthesis indicate percent change between the two values, demonstrating a significant performance loss.



(b) Wrapper performance *with business logic* (`(new Array(100)).join('|').split('|')`; performance impact is negligible. Operations per second. [4]

Figure 2: Function wrapping performance considerations. When measuring invocation performance, the wrapper appears to be prohibitive. However, when considering the business logic that the remainder of the software is likely to contain, the effects of the wrapper are negligible. As such, worrying about the wrapper is likely to be a micro-optimization, unless dealing with call stack limitations. The wrapper in these tests simply invokes the wrapped method with `Function.apply()`, forwarding all arguments.

Section 1.2 mentioned the convention of using a single underscore as a prefix for member names to denote a private member (e.g. `this._foo`). We will adopt this convention for our definition object, as it is both simple and performant (only a single-character check). Combining this concept with the wrapper implementation, we arrive at Listing 28.

```
var Class = ( function( extending )
{
    // implementation left to reader
    var _privname = getRandomName();

    var C = function( dfn )
    {
        // extend from an empty base
        return C.extend( null, dfn );
    };

    C.extend = function( base, dfn )
    {
        base = base || function() {};

        // prevent ctor invocation
        extending = true;

        var ctor = function()
        {
            // do nothing if extending
            if ( extending )
            {
                return;
            }

            // call "actual" constructor
            this.__construct &&
                this.__construct.apply(
                    this, arguments
                );
        };

        // public prototype
        ctor.prototype = new base();
        ctor.prototype.constructor = ctor;

        // private prototype (read-only,
        // non-configurable, non-enumerable)
        Object.defineProperty(
            ctor, _privname, { value: {} }
        );

        copyTo(
            ctor.prototype,
            ctor[ _privname ],
            dfn
        );

        // done extending; clear flag to
        // ensure ctor can be invoked
        extending = false;

        return ctor;
    };

    function copyTo( pub, priv, members )
    {
        var hasOwn = Object.prototype
```

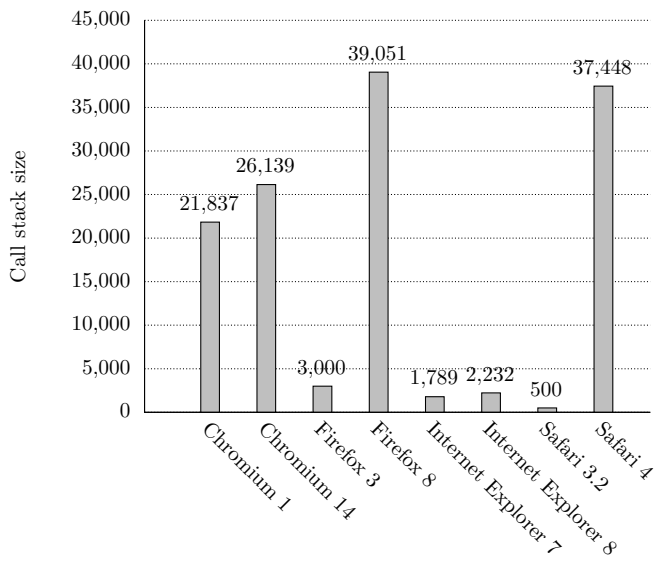


Figure 3: Call stack limits of various common browsers. [5] Determining the call stack limit for your own environment is as simple as incrementing a counter for each recursive call until an error is thrown.

```
60              .hasOwnProperty;
61
62          for ( var member in members )
63          {
64              if ( !hasOwn.call( members, member ) )
65              {
66                  continue;
67              }
68
69              // if prefixed with an underscore,
70              // assign to private "prototype"
71              var dest = ( member[ 0 ] === '_' )
72                  ? priv : pub;
73
74              dest[ member ] = wrap(
75                  members[ member ]
76              );
77          }
78      }
79
80      function wrap( method )
81      {
82          return function()
83          {
84              this.__priv =
85                  this.constructor[ _privname ];
86
87              var retval = method.apply(
88                  this, arguments
89              );
90
91              this.__priv = undefined;
92              return retval;
93          };
94      }
95
96          return C;
97      } )( false );
```

Listing 28: Altering the constructor factory in Listing 21 to support private methods in a manner similar to Listing 17

In order to expose the private methods *only* from within wrapped methods, Listing 28 relies on the fact that only the constructor factory knows the name of the private "prototype" object (as denoted by `_privname`). Wrappers, before invoking the wrapped function (method), will assign the private object to `this._priv` (line 84) and unassign it after the wrapped function returns (line 91).[67] Methods may then access private members by referencing `this._priv`.

`copyTo()`, now receiving both public and private destination objects as arguments, will place all members prefixed with an underscore on the private member object (lines 71–72). As has already been mentioned, the member will be wrapped regardless of whether or not it is private (line 74), ensuring that public methods also have access to private members.

Listing 29 demonstrates how this implementation may be used to define a private method `_getPrivValue()` that is accessable only to other methods; attempting to invoke the method publically would result in a `TypeError` (resulting from an attempt to call `undefined` as if it were a function). Also note that `Foo.__priv`, although defined from within the method `getValue()`, is `undefined` after the method returns.

```
1   var Foo = Class(
2   {
3       getValue: function()
4       {
5           return this.__priv._getPrivValue();
6       },
7
8       _getPrivValue: function()
9       {
10          return 'private';
11      }
12  } );
13
14  var inst = new Foo();
15  inst.getValue();      // "private"
16  inst._getPrivValue(); // TypeError
17  inst.__priv;          // undefined
```

Listing 29: Demonstrating use of private methods with constructor factory

3.2.1 Wrapper Implementation Concerns

The wrapper implementation is not without its dilemmas. Firstly, consider how `wrap()` clears `__priv` in Listing 28

[67] We set the value to `undefined` rather than using the `delete` operator because the latter causes a slight performance hit under v8.

www.dbooks.org

(lines 84–91). The wrapper requires that the call stack be cleared up to the point of the invocation of the wrapped function. Consequently, this means that any code executed before the call stack is cleared to that point will have access to the instance during which time `this._priv` is assigned, giving that code access to private members and breaking encapsulation.

```
1  var Database = Class(
2  {
3      // ...
4
5      forEachRow: function( callback )
6      {
7          for ( row in this._rows )
8          {
9              callback( row );
10         }
11     },
12
13     _getPassword: function()
14     {
15         return 'secret';
16     }
17 );
18
19 var db     = Database( '...' ),
20     passwd = '';
21
22 // ...
23 db.forEachRow( function( row )
24 {
25     passwd = db.__priv._getPassword();
26 } );
27
28 // oops
29 unauthorizedDbOperation( 'user', passwd );
```

Listing 30: Exploiting wrapper implementation via callbacks to gain access to private members outside of the class

This fatal flaw is demonstrated in Listing 30, which executes a callback before the wrapped function returns. That callback, which has access to the instance that called it, is able to access the private members because it is executed before its caller returns. There are three ways to work around this:

1. Remove the assignment before invoking the callback,
2. Use `setTimeout()` or `setInterval()` to invoke the callback, allowing the stack to clear before the callback is invoked, or
3. Do not invoke any functions that are not defined on the class itself.

None of the above options are acceptable solutions. The first option adds unnecessary logic to the method that makes assumptions about the underlying system, which is especially dangerous if the implementation of `wrap()` were to ever change. The second solution does not suffer from the same design issues as the first, but forces the method to be asynchronous, which is not always desirable. The third option is terribly prohibitive, as it not only disallows any type of serial callback, but also disallows invoking any methods of injected dependencies.

A proper solution to this issue is obvious, but its discussion will be deferred to future implementations due to additional complexities raised when dealing with properties. Until that time, the reader should be aware of the issue and consider potential solutions.

The second concern is a bit more subtle. Once again, we focus around lines 82–91 in Listing 28. Consider what problems that this wrapper may cause when dealing with nested method calls — that is, one method calling another on the same instance.

```
1  var Database = Class(
2  {
3      __construct: function( host, user, pass )
4      {
5          // __priv contains private members
6          var ip = this.__priv._resolveHost();
7
8          // __priv is undefined
9          this.__priv._connect( ip, user, pass );
10     },
11
12     // ...
13 } );
```

Listing 31: Problems with nested method calls given the `wrap()` implementation in Listing 28

This issues is demonstrated by Listing 31. The `Database` class's `__construct()` method performs two private method calls — `_resolveHost()`, to get the IP address of the host, and `_connect()`, which attempts to connect to the database. Unfortunately, after the call to `_resolveHost()`, its wrapper sets `__priv` to `undefined` (line 91 in Listing 28), which will cause the second method call to fail!

To resolve this issue, `wrap()` could store the previous value of `this.__priv` and then, instead of setting the value to `undefined` after the wrapped function has returned, restore `this.__priv` to its original value. This modification is shown in Listing 32.

```
80     function wrap( method )
81     {
82         return function()
83         {
84             var prev = this.__priv;
85
86             // expose private member object
87             this.__priv =
88                 this.constructor[ _privname ];
89
90             var retval = method.apply(
91                 this, arguments
92             );
```

```
93
94              // restore previous value
95              this.__priv = prev;
96              return retval;
97          };
98      }
```

Listing 32: Fixing private object assignment in nested wrapped function calls by restoring the previous value

When the first wrapper is invoked, the previous value of `this.__priv` will be `undefined`, allowing the wrapper to continue to operate as it used to. When nested wrappers are invoked, the previous value will contain the private member object and will allow `this.__priv` to be properly restored on return. This functions much like a stack, using the call stack instead of an array.[68]

4 Licenses

This document and all source code contained within is free,[69] released under the GNU FDL. The source code contained within the listings are also licensed under the GNU GPL, unless otherwise noted within this section, to permit their use in free software. The code listings are intended primarily for instruction and example and may not be directly applicable to your software. If licensing is a concern, one may use the listings to implement the code from scratch rather than using the code verbatim.

Each license may be found at http://www.gnu.org/licenses/.

4.1 Document License

Copyright © 2012 Mike Gerwitz.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

4.2 Code Listing License

Copyright © 2012 Mike Gerwitz.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU Affero General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Affero General Public License for more details.

4.2.1 Code Listing License Exceptions

The following listings are provided under alternative licenses.

Listing 16 GNU LGPL (taken from ease.js)

4.3 Reference Licenses

The reader should be warned that certain references mentioned within this article are non-free (that is, their licenses do not permit redistribution, modification, or derivative works). These references are denoted with "[NF]". While these resources are not the official documentation for free software (and consequently do not pose a direct threat to the users of free software), be aware that they do prohibit you from helping your friends and neighbors by giving them a copy.

This article itself holds no such restrictions.

5 Author's Note

Please note that this article is an incomplete draft; it is under heavy development to include relevant information from the development of ease.js. In the meantime, the reader is welcome to browse through the technical manual for the project at `http://easejs.org/manual/Implementation-Details.html`. The manual contains implementation details and rationale for much of what will be elaborated upon in this paper.

References

[1] ECMA International. *ECMA-262*, 5.1 edition, Jun 2011. Section 10.4.3.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[3] Mike Gerwitz. Function wrapper (invocation). `http://jsperf.com/coope-function-wrapper`. Accessed: 25 Feb 2012.

[4] Mike Gerwitz. Function wrapper (w/ business logic). `http://jsperf.com/coope-function-wrapper-w-blogic`. Accessed: 25 Feb 2012.

[5] Nicholas C. Zakas. *High Performance JavaScript*. O'Reilly Media, Inc, 2010. Reference for some of the call stack limits mentioned in the article [NF].

[68] This solution could have easily been worked into Listing 28, but hopefully the additional discussion provided insight into critical design decisions.

[69] Free as in "free speech", not "free beer".

www.dbooks.org