

Proposed ECMAScript 3.1 Static Object Functions: Use Cases and Rationale

*Allen Wirfs-Brock
Microsoft Corporation*

In addition to the above author the following individuals contributed to the design described in this document:

*Mark Miller, Google
Douglas Crockford, Yahoo!
Kris Zyp, Dojo Foundation
Pratap Lakshman, Microsoft
Adam Peller, IBM*

The current draft of the proposed specification for ECMAScript 3.1 includes a number of functions that are accessible as properties of the standard built-in constructor function, `Object`. While these functions might appear to some observers as an arbitrary set of individual feature additions to ECMAScript, they are actually a unified design created to support some specific high value use cases. This note presents the use cases addressed by these features and presents a rationale for the particular design decisions they reflect.

Use Case: Programmatic control of Property Attributes.

One of the most common feature requests from ECMAScript framework authors is for an API that allows them to explicitly control the internal attributes of object properties. These attributes are the ECMAScript 3 internal attributes `ReadOnly`, `DontEnum`, and `DontDelete` (section 8.6.1). Without a mechanism for setting these attributes, framework and library implementers are not on an equal footing with the builders of extended ECMAScript implementations. For example, the Mozilla “array extras” are a set of methods added to `Array.prototype` by Mozilla. These functions generally are perceived to be quite useful and several implementations of them have been created to enable their use on ECMAScript implementations that do indirectly incorporate Mozilla’s extensions. However, there is one aspect of Mozilla’s implementation that these libraries cannot emulate—the attribute settings of the properties that implement the methods. This difference is directly observable by users of such emulation libraries, particularly with regard to the `DontEnum` attribute. Mozilla’s definition of these methods, like all standard built-in methods, all have the `DontEnum` attribute. As a result, when a user writes code that uses the `for...in` statement to iterate over the names of the properties of an array object the names of these methods are not included. However, if an independently loadable library is being used to provide these methods their names are included in a `for...in` iteration over any array and the user must explicitly write code to ignore them.

While the DontEnum attribute is of most concern to framework and library developers, other developers have needs to control the other property attributes. For example, creators of secure execution sandboxes want to be able to control the ReadOnly and DontDelete state of objects passed into the sandbox, and developers trying to emulate host objects such as the DOM need to be able to exactly duplicate all of the attributes of the actual host objects.

Attribute Control API Design and Rationale

Because there was no widely implemented API for controlling the internal ECMAScript attributes, a new set of APIs had to be developed. While the most common use cases considered required the ability to programmatically set the attribute of a property we also assumed that the ability to programmatically inspect the attributes of a property was also needed in order to reliably use the set functionality. A number of alternative API designs were considered before the proposed API was chosen. In the course of considering alternatives we developed a set of informal guidelines that we applied when considering the alternatives. These guidelines are:

- Cleanly separate the meta and application layers.
- Try to minimize the API surface area (i.e., the number of methods and the complexity of their arguments).
- Focus on usability in naming and parameter design.
- Try to repeatedly apply basic elements of a design.
- If possible, enable programmers or implementations to statically optimize uses of the API.

Here are some of the alternatives that were considered that lead to the selected design.

The obvious initial idea, following the example of the already existing standard method `Object.prototype.propertyIsEnumerable`, was to add additional “propertyIs...” query methods on `Object.prototype` for the other attributes and a parallel set of attribute changing methods. For example, the complete set of methods defined on `Object.prototype` might be: `propertyIsEnumerable`, `propertyIsWritable`, `propertyIsDeletable`, `propertyMakeEnumerable`, `propertyMakeWritable`, and `propertyMakeDeletable`. Presumably all the methods would take a string containing a property name as a first argument and in addition the “Make” methods would take a Boolean value indicating the new state of the attribute as a second argument. As an alternative to the Boolean argument three additional methods (`propertyMakeNonEnumerable`, `propertyMakeNonWritable`, `propertyMakeNonDeletable`) could have been used.

As we considered this approach there were a number of things about it that we didn’t like and that seemed contrary to the above API design guidelines:

- It merges rather than separates the meta and application layers. As methods on `Object.prototype` the methods would be part of the public interface of every application object in a program. As such, they need to be understood by every developer, not just library designers.

- The API has a large surface area, at least 5 and possibly 8 new methods. This seems excessive for a relatively small piece of functionality.
- The method names are long and wordy. This is generally undesirable, even if it is consistent with the precedent of `propertyIsEnumerable`.
- They are difficult to either manually or mechanically optimize. The likely common task of setting a property to `{ReadOnly, DontEnum, DontDelete}` requires at least three individual method calls as this API design had no way to set all the attributes of a property in a single call. Also, as inherited methods of the actual object whose properties are being modified, it would be difficult for an implementation to statically prove for optimization purposes that the methods being called are the actual built-in methods.

One thing we did like from the `propertyIsEnumerable` precedent was the use to simple true/false adjectives like `Enumerable` to name attributes rather than the “DontXXX” phrases used within the ECMAScript 3.0 specification. Because of this precedent and the fact that the specification’s internal attribute names have never been used in any APIs defined by the standard, we felt we could adopt that convention for all the attributes and initially choose the terms “Enumerable” “Writable” and “Deletable” as the attribute names for use in designing the API. For reasons that are explained in a later section, “Deletable” eventually became “Flexible”.

As a solution to the problem of merged meta and application layer APIs we explored moving these methods from `Object.prototype` and making them into what would essentially be global “static” functions. Doing this requires the introduction of an additional parameter to the functions to identify the object that contains the property under consideration. Thus there is a trade-off between the simplicity of the functions’ APIs and the separation of meta and application concerns. In this case we felt that keeping the application API layer simpler was sufficient justification for a slightly more complex meta layer API.

For obvious reason we didn’t want to pollute the global object’s name space with a number of individual function objects so the functions needed to be defined as properties of some global container object. We considered introducing a new global singleton object named `Meta` (think of it as an analog to the `Math` object but for meta layer functions) for this purpose. However, we eventually concluded that the existing global constructor `Object`, which in ES3 defines no methods, could reasonably be used for this purpose.

We also wanted to address the issue of the relatively large number of number of methods need to define or query property attributes. At first, there seemed to be only two general alternatives: a “set” method with a separate positional parameter corresponding to each property attribute; or, a “set” method with a numerically encoded bit vector parameter.

Using multiple positional parameters, a call to set two attribute values might have looked something like this:

```
Object.definePropertyAttributes(Array.prototype, "forEach", undefined, false, true);
```

There are some obvious usability issues with such a call. Both the reader and writer of such a function call have to remember the correspondence between the last three parameters and the specific attributes. For example, does the fourth parameter correspond to Enumerable or Deletable? If the API needs to support setting only individual parameters, a three value logic is needed so that attributes that are not intended to be changed can be indicated. That is the purpose of the use of `undefined` as one of the arguments in the above example. (Note that an alternative to the three valued logic would to make a function like `definePropertyAttributes` available as an alternative rather than a replacement for the `propertyMake...` style functions. However, that would make the total number of functions in the API even larger.) Finally, because ES3 (and the proposed ES3.1) doesn't directly support either functions that return multiple values or by reference parameters there is no obvious analog to use in designing the function that retrieves the current state of the properties.

The use of a bit vector to encode attribute values partially addresses some of these issues but would create others. Consider the following alternative to the above function call:

```
Object.definePropertyAttributes(Array.prototype, "forEach",  
                               Object.Enumerable + Object.Writable, Object.Writable);
```

This example assumes that the intent of the previous call was to leave the Deletable attribute unchanged, set the Enumerable attribute to false, and to set the Writable attribute to true (in which case the third through fifth parameters of the first alternative call corresponded, in order, to Deletable, Enumerable, and Writable). The properties `Object.Writable`, `Object.Enumerable`, and `Object.Deletable` would be integer powers of two such as 1, 2, 4 that define the bit positions corresponding to the state of the correspondingly named attribute. They can be used as a mask for accessing the state of a specific attribute or as the "set" value of the attribute. In order to express that some attributes values are to remain unchanged and that some attribute values are to be cleared (set to false) some sort of masking is required to identify the attributes to be left alone or to be operated upon. In this example assume that the third argument is the mask argument, so it specifies that the Enumerable and Writable attributes are going to be modified and, by implication, that the Deletable attribute will be left unmodified. The fourth argument then specifies which of the mask-selected attribute bits are to be set on (put into the true state). Since only `Object.Writable` is specified by the fourth argument expression, only that attribute is change to the true state. Because `Object.Enumerable` was included in the mask but not in the fourth argument expression, the Enumerable attribute is set to the false state. Other masking schemes are possible, but they would differ from this only in the details of their use.

One advantage of the bit vector approach is that it can also be directly applied for retrieving the current state of the attributes. Assuming that `getPropertyAttributes` returns a bit vector with the same encoding as above, a test of the Deletable attribute might be code like this:

```
if (Object.getPropertyAttributes(Array.prototype, "forEach") & Object.Writable) {  
    ...  
}
```

The bit vector approach is arguably a bit more readable than the positional parameter formulation and could certainly reduce the total number of functions in the attribute API although the elimination of the `propertyIs...` and `propertyMake...` function is partially offset by the addition of the bit position constant properties. However, this sort of bit fiddling just didn't seem appropriate for part of the standard library of a higher level language such as ECMAScript.

Revisiting the positional parameter approach, an observation was made that its usability would be much greater if ECMAScript supported optional keyword parameters. That would eliminate the need for the caller to use a three valued logic and to remember the meanings of parameter positions. Using a hypothetical keyword parameter scheme the first alternative call above might be expressed as:

```
Object.defineProperty(Array.prototype, "forEach",
                      enumerable=false, writable=true);
```

This is clearly both more readable and more writable, but simply isn't an available features of the language. However, ECMAScript object literals provide a very similar syntactic feel. Using the convention that the third argument must simply be an object that optionally may have properties named "enumerable", "deletable", and "writable", a syntactically correct call could be written as:

```
Object.defineProperty(Array.prototype, "forEach",
                      {enumerable: false, writable: true});
```

This convention would also address the issue how to return multiple values from the attribute query function. It could be defined as returning a simple data object with the same three property names. In that case, the above if statement that tests a property's writable attribute could be coded like this:

```
if (Object.getPropertyAttributes(Array.prototype, "forEach").writable) {
    ...
}
```

This style of API proved very compelling and highly in line with the API design guidelines listed above. It separates the attribute meta operations from the application layer. It has a small surface area requiring only two methods, and has high usability. It exploits fundamental features of the language and, as we subsequently discovered, could be applied to other APIs we need to provide. Finally, as is explained in a later section, it is highly amenable to static optimization using simple techniques.

For reasons that are explained below the two function names that were actually chosen for this API are:

- `Object.defineProperty(obj, propertyName, attributeDescriptor)`
- `Object.getProperty(obj, propertyName)`
/* returns an attributeDescriptor */

where `attributeDescriptor` is simply an object that may include Boolean valued properties named "writable", "enumerable", and "flexible". For calls to `defineProperty`, the presence in the `attributeDescriptor` of any of these properties is optional. The object returned by `getProperty` generally has all of these properties present.

Use Cases: Programmatic Creation and Inspection of Getter/Setter Properties.

ES3.1 adds the concept to getter/setter properties to ECMAScript. Such properties, rather than being represented as a variable cell into which values may be stored and retrieved, are represented as a pair of functions that are used to store and retrieve a virtual property value. Getter/setter property functionality, as implemented by several current extended ES3 implementations and as proposed for ES3.1, includes syntactic support for defining getter/setter properties on objects created using object literals but does not provide any syntax for defining them on objects created through other means.

The properties of many applications are defined via programmatic construction rather than using object literal syntax. So, full support of getter/setter properties requires a programmatic means of construction. In addition, in ES3 the cloning, copying, or most other forms of duplication of application objects must currently be accomplished algorithmically. If an object that includes getter/setter is to be duplicated, there must be a programmatic means to inspect the object's properties, determine which are data properties and which are getter/setter properties, and to duplicate the definition of getter/setter properties.

Getter/Setter API Design and Rationale

The starting point for considering getter/setter API designs has to be the API that is already used in existing extended ECMAScript implementations. The Mozilla API, which has been copied by other implementations, consists of four methods:

- `Object.prototype.__defineGetter__(propertyName, functionObject)`
- `Object.prototype.__defineSetter__(propertyName, functionObject)`
- `Object.prototype.__lookupGetter__(propertyName)`
- `Object.prototype.__lookupSetter__(propertyName)`

The most obvious concern regarding this API is the naming convention of beginning and ending each function name with two underscore characters. This is a convention that Mozilla uses to identify non-standard functions. This is a useful naming convention (and probably one that other implementations should be encouraged to follow). However, it is not a naming convention that makes sense for items that are actually incorporated into the ECMAScript standard. Such names would be inconsistent with all other naming conventions already used with the standard and have an “ugliness” factor that probably should not be inflicted upon all future generations of ECMAScript programmers. In addition, incorporation of such names into the standard would also eliminate the utility of the naming convention as a means of identifying non-standard features. In addition, while there are multiple extended ECMAScript implementations that use these names it isn't clear that they all actually implement the same semantics. There is a solution to all of these issues – simply eliminate the underscores from the names. The getter/setter API would consist of four methods on `Object.prototype`: `defineGetter`, `defineSetter`, `lookupGetter`, and `lookupSetter`.

The next item to consider was whether this set of four functions was an API that made sense to standardize upon. When we looked closer at the API we encountered several of the same issues that we also encountered when considering the attribute control API. The first issue is again a mingling of meta and application layer API. By defining these functions on `Object.prototype` they become part of public interface of all application layer objects. Clearly, this could be addressed by moving them from methods on `Object.prototype` to being methods on the `Object` constructor just as we did for `definePropertyAttributes`.

The API also has a fairly large surface area consisting of four methods and even that set seems incomplete or requires use of idiomatic sequences to obtain some information. For example, how would a programmer who needed to duplicate an object programmatically determine whether or not an object implements some property as a getter/setter? Perhaps it could be accomplished by first calling `Object.prototype.hasOwnProperty` to determine whether a property actually exists and then calling both `lookupGetter` and `lookupSetter` and interpreting a result of `undefined` from both calls as an indication that the property is not a getter/setter property. However, is this really correct? Is it actually possible using this API to distinguish between a data property whose value is `undefined` and a getter/setter property whose functions have both been defined to be `undefined`? Does setting both the getter and setter functions of a property to `undefined` revert it to a normal data property or does it remain a getter/setter with default function definitions? The actual answer would ultimately depend upon a comprehensive specification of the semantics of the functions but it is possible that additional functions might be needed to reliably test whether or not a property is a getter/setter or to change property definitions between being getter/setters or conventional data properties.

One way to reduce the number of functions required for this API would be to combine `defineGetter` and `defineSetter` into a single function using positional parameters, similar to what we considered for manipulating property attributes. This creates similar issues with similar solutions as was discussed above. Applying the same principles as before you can arrive at two methods:

- `Object.defineProperty(obj, propertyName, propertyDescriptor)`
- `Object.getProperty(obj, propertyName)`
 /* returns a propertyDescriptor */

where a `propertyDescriptor` is an object that optionally may have properties named “getter” and “setter” which might be provided using an object literal. For example:

```
Object.defineProperty(obj, "length", {
  getter: function () {
    return this.computeLength()
  },
  setter: function (value) {
    this.changeLength(value)
  }
});
```

If only a getter or a setter property was specified in the property descriptor object then the other function would be undefined or left unmodified if it had already been defined by a previous call. If neither was specified it would not be a valid getter/setter definition. For the `getProperty` function the object returned would have both getter and setter properties when inspecting a getter/setter property but either or both of them might be `undefined` indicating that no actual function has been specified. It also seemed appropriate for the `getProperty` result to explicitly distinguish between getter/setter properties and conventional ES3 style data properties. We accomplished this by returning an object with a property named “value” instead of a pair of properties named “getter” and “setter”. The value of the “value” property is the actual current value of the requested property. For symmetry and to simplify the copying of properties from one object to another we also recognize “value” in a property descriptor passed to `defineProperty` as a request to create a new data property rather than a getter/setter property.

By this point, it is obvious that the differences (at least in the API) between `defineProperty/definePropertyAttributes` and `getProperty/getPropertyAttributes` are very minor and in addition that a definer of a property might reasonably also need to define the property’s attributes. Similarly, an inspector of a property definition is likely to also want to inspect the property’s attributes. So we merged the definitions of property descriptors and attribute descriptor and also merged the functionality of `definePropertyAttributes /getPropertyAttributes` into `defineProperty/getProperty`. A call to define a getter/setter property and specify its attributes looks like this:

```
Object.defineProperty(obj, "length", {
  writable: false, enumerable: false, flexible: false,
  getter: function () {
    return this.computeLength()
  },
  setter: function (value) {
    this.changeLength(value)
  }
});
```

We were able to reduce the total number of functions needed to support getters/setters and attribute access and control from a total of at least nine to only two. We then added in an additional `getOwnProperty` function because many use cases, including object duplication, require a program to explicitly distinguish between own and inherited properties.

API Object Literals Performance and Optimization Considerations

It is reasonable to ask whether there are performance consequences associated with the use of object literals essentially as a means for specifying keyword parameters. For the situations in which we are using them, we believe that performance will not be an issue. Our expectation is that the setting of attribute values on pre-existing objects (such as marking all the properties of a built-in object as being non-writable) would be one-time operations and that any object creation overhead in such a case would be inconsequential. Similarly, we expect that most programmatic property definition (particular of

getter/setter) properties will occur on objects that are intended for use as prototypes and hence also will be one time activities.

We expect that the most common use cases for the `defineProperty` function will take an object literal as a descriptor, and that all of the literal's defined properties will also have literal values. There are various ways that an implementation could optimize the creation of such pure literal objects.

Finally, we expect that in many situations it will be possible for any implementation to completely optimize away any object creation for such parameters. Consider a call like:

```
Object.defineProperty(Array.prototype, "forEach",  
                        {enumerable: false, writable: true});
```

An implementation can easily recognize this as a likely candidate for optimization simply because it directly calls a property on a object referenced using the name "Object". Assuming that `defineProperty` is itself specified as a "DontDelete,ReadOnly" property, the only runtime optimization guard that would be required to ensure the validity on any optimization would be an identify check that the current value of `Object` is still the built-in constructor object. In that case, fast-path code could directly set the specified attribute values using whatever implementation specific encoding (for example bit encoded attribute flags) was convenient and there would be no need to actually instantiate the object literal as an object or to even call the actual `defineProperty` function. Similar optimizations are equally applicable to calls that define new properties, including getter/setter properties.

The Flexible Property Attribute Name Rationale

In ES3 the possible basic state transitions of a property are to create the property, to change the value of the property, and to delete the property. The ES3 `ReadOnly` and `DontDelete` property attributes regulate the ability make the latter two transitions. With the addition of getter/setter properties and reified property attributes, there are additional possible property state transitions such as replacing a getter or setter function, changing an attribute of a property, changing a data property into a getter/setter property, and changing a getter/setter property into a data property. An issue that needed to be explored was if and how these property transformations should be regulated. Simply forbidding all such transformation seemed to go against the general permissive spirit of ECMAScript. However, introducing additional properties to control each of the possible transitions seemed excessive particularly in light of the new necessity to reify them in the property descriptors used by `defineProperty` and `getProperty`.

The solution we finally agreed to is based upon a reinterpretation of the original intent of the `DontDelete` attribute. While `ReadOnly` is concern with the value or data state of a property, it can be argued that `DontDelete` is instead concerned with the existential state of the property. When a programmer sets `DontDelete` (if they could, in ES3 only an implementer can actually set it) they are saying that they do not want to allow any changes to the fundamental existential nature of the property. Other than creation, the only existential state transition possible in ES3 was deletion of the property but

with the addition of getter/setter properties additional existential transitions are now possible. For ES3.1 we choose to reinterpret DontDelete as forbidding all such existential changes, not just deletion.

This change of interpretation meant that “DontDelete” or “deletable” was too narrow a name for the that attribute. Initially we proposed to rename the attribute “dynamic” because it controls the ability to dynamically modify the nature of a property. The ES4 designers felt that use of “Dynamic” was inconsistent with the ES4 meaning of that same term and suggested the use of the term “isDynamic” for that attribute. However, such a compound name is inconsistent with the naming convention that we were following for other attributes. “flexible” was proposed as an alternative term that does not have any ES4 terminology conflicts.

The general interpretation of the “flexible” attribute should be that it allows the definition of the property to be arbitrarily changed. When a property’s flexible attribute is true its definition can be modified in many ways including modifying individual attribute values, changing the kind of the property, modifying getter or setter functions, or even deleting the property. If the flexible attribute is false, the definition of the property has been fixed and it cannot be changed or deleted.

Multiple Property/Attribute Definition Use Cases and Rationale.

It is relatively rare to define only a single property or to want to set the attributes of only a single property. More typical would be the definition of a set of related properties or perhaps all the properties of an object that is going to be used as a prototype object. We felt that this use case was common enough to warrant direct API support. This is accomplished by providing a `defineProperties` function that is similar to `defineProperty` except that it accepts as an argument a set of property descriptors instead of a single one. A property descriptor set is simply an object whose property names are the names of the properties to be defined (or updated) and whose property values are the individual property descriptor objects. Such an object structure will most commonly be written as an object literal. For example:

```
/* update Array.prototype with latest "extras" */
Object.defineProperties(Array.prototype, {
  reduce: {
    enumerable: false,
    value: function (fn /*, initial*/) {
      /**function body omitted**/
    },
  },
  reduceRight: {
    enumerable: false,
    value: function (fn /*, initial*/) {
      /**function body omitted**/
    }
  }
});
```

This is mostly a convenient combination of what would otherwise have taken two separate calls to `Object.defineProperty` but there is one important difference from the individual calls. `defineProperties` is specified as performing an atomic update to the properties of the object

passed as its first argument -- either all the property changes specified by the descriptor are applied or none of the changes are applied.

As with `defineProperty`, the actual creation of the descriptor objects may be optimized away in many common situations. In addition, the specification of a set of properties as an atomically installed unit may present implementers with additional optimization opportunities.

A proposal that was considered and rejected was to provide `getProperties` and `getOwnProperties` functions that would return a property set descriptor of the form accepted by `defineProperties`. While the symmetry was attractive, such functions begins to infringe upon the territory of a full featured mirrors-style reflection model. While such a reflection model may make an attractive addition to some future version of ECMAScript we don't believe that ES3.1 is the appropriate place to introduce it. Instead we concluded that it was better to focus on providing the primitive functions that might be used build such a higher level reflection model.

This led to the identification of two additional features that are required to fill out the primitive reflection functions. These features are navigation of the prototype chain and the identification of property names defined by an object. These features in conjunction with `getOwnProperty` enable functions like `getProperties` to be easily implemented in user code.

Prototype chain navigation is provided via the `Object.getPrototypeOf` function. This function takes an object as an argument and returns its prototype object (or null). The naming of this function is modeled after the existing `Object.prototype.isPrototypeOf` function. However, consistent with our goal of maintaining separation between the meta and application layer, it is a static function rather than an `Object.prototype` method.

The function `Object.getOwnPropertyNames` takes an object as an argument and returns an `Array` of strings containing the names of all own properties of the argument object. Such a list of defined property names is essential for any meta operations that needs to iterate over all of an object's properties. Such operations could include both application level operations such as copying an object or higher level meta operations such as `getProperties` function that we choose not to include in the ES3.1 specification.

Object Creation Functions Use Cases and Rationale

ECMAScript does not currently provide a straight forward way to create an object with an arbitrarily specified prototype. Creating an object whose prototype is `Object.prototype` is trivially accomplished by the expression `new Object()` or just `{}`. However, creation of an object with some other prototype requires creation of a unique constructor function and the setting of the constructor's prototype property.

The expectation is that a new object will frequently be used as the first argument to `Object.defineProperties` and a common idiom is likely to be something like:

```

var myProto = Object.defineProperties(new Object(), {
  method1: {
    enumerable: false, writable: false,
    value: function (arg) {
      /**function body omitted**/
    }},
  method2: {
    enumerable: false, writable: false,
    value: function (fn /*, initial*/) {
      /**function body omitted**/
    }
  }
});

```

Note that the first argument is just a newly constructed object. However, if a programmer then wanted to define an additional object whose prototype was `myProto`, there would be no equally directly way to accomplish this using ES3 functionality. The function `Object.create` is provided as a solution to this use case. In its basic form it takes an arbitrary object as an argument and returns a new “empty” object that has the argument object as its prototype. With `Object.create` the programmer could then code:

```

var myOtherProto = Object.defineProperties(Object.create(myProto), {
  method3: {
    enumerable: false, writable: false,
    value: function (arg) {
      /**function body omitted**/
    }},
  method4: {
    enumerable: false, writable: false,
    value: function (fn /*, initial*/) {
      /**function body omitted**/
    }
  }
});

```

Because we expect this particular idiom to be very common we specified `Object.create` to accept an optional second argument that is the same sort of property set descriptor that is used by `defineProperties`. This enables the above example to be alternatively written like:

```

var myOtherProto = Object.create(myProto, {
  method3: {
    enumerable: false, writable: false,
    value: function (arg) {
      /**function body omitted**/
    }},
  method4: {
    enumerable: false, writable: false,
    value: function (fn /*, initial*/) {
      /**function body omitted**/
    }
  }
});

```

This is primarily a useful shorthand but it also provides additional or simplified implementation specific optimization opportunities.

Note that `Object.create` without its optional second argument is essentially the same operation as the `beget` function that was been widely promoted. We (perhaps not surprisingly) agree with the utility of this function but felt that the word “beget” is probably confusing to many non-native English speakers.

The other object creation functionality that is missing from ES3 is the ability to create an exact copy of an object. The new static meta functions are sufficient to construct such a copy in most situations. However, even with those functions available there is currently no way to replicate the internal properties and internal methods of some objects such as the special `[[Put]]` method used for `Array` instances. In addition, in many cases an implementation should be able to perform the complete replication of an object must faster than could be accomplished using multiple calls to meta functions. The function `Object.clone` provides this functionality. It takes an arbitrary object as an argument and is specified to return a new object that is an exact copy of the argument, except for its object identity.

Object Lock-Down Functions Rationale

In order to construction secure execution sand boxes it is necessary to “lock-down” the state of some objects that will be passed into the sandbox. One form of lock-down ensures that the definitional structure of the object is fixed. Essentially, the definition of the object’s properties are fixed and cannot be modified. However, the state of data properties could still be modifiable. A more aggressive form of lock-down fixes the definitional structure and in addition makes all data properties immutable.

ES3.1 proposes two functions for placing objects into such locked-down states. `Object.seal` fixes the definitional state of its argument object. It does this by setting the flexible attribute of each own property to the false state. In addition, it marks the object such that additional properties cannot be added. `Object.const` performs the same actions as `Object.seal`. In addition, it sets the writable attribute to false for every own data property of the object. ES3.1 also provides `Object.isSealed` and `Object.isConst` functions for query whether or not an object is in one of these locked-down states.

Various alternative names such as “fix” and “freeze” were proposed and considered as names for the two lock-down states. The final names were chosen because it was felt that the selected terms had the greatest cognitive distance between them and hence would be less subject to confusion.

Note that the only thing that can be accomplished by `Object.seal` and `Object.const` that can’t be accomplished using the other available meta function is imposing the restriction of adding new properties. In addition, they are specified as atomically placing their argument objects into the specified locked-down state. Whether or not the ability to control property additions should be separately controllable is probably worth further consideration.