Ecma/TC39/2011/009

[[strawman:private_names]]

Table of Contents

- Overview
- Private Name values
- The private declaration
- Using Private Identifiers
- Private Identifiers in Object Literals
- Private Declaration Scoping
- Private Declarations Exist in a Parallel Environment
- Accessing Private Names as Values
- Conflict-Free Object Extension Using Private Names
- Enumeration and Reflection
- Private Name Properties Support Only Weak Encapsulation
- Interactions with other Harmony Proposals
  - Enhanced Object Literals
  - Proxies
  - Modules
- References

Overview

2010/12/08 15:16
Revised proposal by Allen Wirfs-Brock
Original proposal by Dave Herman and Sam Tobin-Hochstadt is here.

In existing ECMAScript, it is not possible to create properties that have limited or controlled accessibility. It is possible to create non-enumerable properties, but they can still be discovered by guessing their string-valued property name. The proposed es4 facility for addressing this shortcoming was namespaces, which were complex and suffered from ambiguity and efficiency problems.

This strawman proposes three related changes to support hidden properties.

1. a new, ECMAScript language type (or possibly object `[Class]`) *Private Name*
2. generalizing the ES5 *property name* concept to include either a string (as in ES5) or a *Private Name* value
3. a `private` keyword for automatic use of *Private Name* values instead of strings in syntactic contexts in a lexically scoped fashion.

In addition to creating hidden properties, this also allows properties to be added to existing objects without the possibility of interference with the existing properties, or with other additions by any other code.

Private Name values

`Private Name` is a new ECMAScript language type that will be defined in section 8 of the specification. The `Private Name` type is an open set of distinct `Private Name` values that can be used as the names of object properties. `Private Name` values do not have any corresponding literal representation within ECMAScript code. Distinct `Private Name` values are created by the `CreatePrivateName` abstract operation. Each call to `CreatePrivateName` returns a new distinct `Private Name` value. If `x` and `y` are `Private Name` values then the abstract operation `SameValue(x,y)` returns true if and only `x` and `y` are the same `Private Name` value created by a specific call to `CreatePrivateName`.

A `Private Name` value can be used as the value of the `P` argument to any of the object internal methods defined in section 8.12 of the ECMAScript specification.

If a new ECMAScript type is added then the `typeof` operator will also need to be extended to return a new string value that identifies values of that type. Concerns have been expressed that extending `typeof` in this manner could break existing code that expects to deal with a fixed set of `typeof` values. We need to make a global decision about adding new non-object types and the impact upon `typeof`.

Object values could be used as an alternative to defining a new ECMAScript type to represent `Private Name` values. Each `Private Name` would simply be a distinct object. `Private Name` objects would have a distinct `[Class]` value. To avoid such objects being used for back-channel communication or property garbage dumps they should be created frozen. Conceivably they could all have `null` as their prototype. This may have some benefit if such names are passed between global contexts as it would prevent use of their prototype value as means of identifying their origin context. Throughout the rest of this spec. "private name value" should be read as meaning whichever form is ultimately used.

The private declaration

The `private` declaration creates a new `Private Name` value by calling `CreatePrivateName` and binds that value to a program identifier that may be used in specific syntactic contexts within the lexical scope of the declaration. An identifier that appears in a `private` declaration is call a `private identifier`.

```
private secret;  //create a new ''Private Name'' that is bound to the private identifier ''secret''.
private _x,_y;   //create two ''Private Name'' values bound to two private identifiers
```

Can the names defined in a `private` declaration be any *IdentifierName* or should they be restricted to being an *Identifier* (ie, not a reserved name)? ES5 allows any *IdentifierName* to be used after a dot or as a property name in an object literal so it may be reasonable to allow any *IdetnifierName*. However that will permit strange look formulations such as: `private private;`

Using Private Identifiers

When a private identifier appears as the *IdentifierName* of a *CallExpression* : *CallExpression* . *IdentifierName* production or of a *CallExpression* : *CallExpression* . *IdentifierName* production, the `Private Name` value that is bound to the private identifier is used as the value of the *IdentifierName*. If the identifier in one of these productions is not a private identifier then the identifier name string is used as the value of *IdentifierName*, just as in ECMAScript 5.

This permits object properties to be created whose names are `Private Name` values. It also allows for the values of such properties to be accessed.

```
function makeObj() {
   private secret;
   var obj = {};
   obj.secret = 42;  //obj has a single property whose name is a Private Name value
   print(obj.secret);//42 -- the private identifier can be used in scope to access the property's value
   print(obj["secret"]); //undefined -- the name of the property is not the string "secret"
   return obj;
}
var obj=makeObj();
print(obj["secret"]); //undefined -- the name of the property is still not the string "secret"
print(obj.secret);    //undefined -- this statement is not in the scope of the private declaration so the
```

```
                      //string value "secret" is used to look up the property.  It does
not match the Private Name value
```

This technique can be used to define “instance-private” properties:

```
function Thing() {
    private key;   // each invocation will use a new private key value
    this.key = "instance private value";
    this.hasKey = function(x) {
        return x.key === this.key;  //x.key should be undefined if x!==this
    };
    this.getThingKey = function(x) {
        return x.key;
    };
}

var thing1 = new Thing;
var thing2 = new Thing;

print("key" in thing1);       // false
print(thing2.key);            //undefined
print(thing1.hasKey(thing1)); // true
print(thing1.hasKey(thing2)); // false
```

By changing the scope of the private declaration a similar technique can be used to define “class-private” properties:

```
private key;  //the same private name value is used by every invocation of Thing
function Thing() {
    this.key = "class private value";
    this.hasKey = function(x) {
        return x.key === this.key;
    };
    this.getThingKey = function(x) {
        return x.key;
    };
}

var thing1 = new Thing;
var thing2 = new Thing;

print("key" in thing1);       // false
print(thing1.hasKey(thing1)); // true
print(thing1.hasKey(thing2)); // true
```

Friend visibility similar to that provided by c++ can be obtained by using private declarations that are visible to several related object literals, object constructors or factory functions enclosed in an outer function and returned from it (directly, or stored as effects in objects).

Private Identifiers in Object Literals

A private identifier may also appear as the *IdentifierName* of a *PropertyName* production in an *ObjectLiteral*. If the identifier in such a productions is not a private identifier then the identifier name string is used as the value of *IdentifierName*, just as in ECMAScript 5.

With this feature, object literals can be used as an alternative expression of the previous three examples:

```
function makeObj() {
   private secret;
   var obj = {secret: 42};
   print(obj.secret);//42 -- the private identifier can be used in scope to access the
property's value
   print(obj["secret"]); //undefined -- the name of the property is not the string
"secret"
   return obj;
}
```

```
function Thing() {
    private key;
    return {
       key : "instance private value",
       hasKey : function(x) {
          return x.key === this.key;  //x.key should be undefined if x!==this
       },
       getThingKey : function(x) {
          return x.key;
       }
    };
}
```

```
private key;
function Thing() {
    return {
       key : "class private value",
       hasKey : function(x) {
          return x.key === this.key;  //x.key should be undefined if x!==this
       },
       getThingKey : function(x) {
          return x.key;
       }
    };
}
```

Private Declaration Scoping

`private` declarations are lexically scoped, like all declarations in Harmony. Inner `private` declarations shadow access to like-named `private` declarations in outer scopes. Within a block, the scoping rules for `private` declarations are the same as for `const` declarations.

```
function outer(obj) {
   private name;
   function inner(obj) {
      private name;
      obj.name = "inner name";
      print(obj.name);   //"inner name" because outer name declaration is shadowed
   }
   obj.name = "outer name";
   inner(obj)
   print(obj.name);      //"outer name"
}
var obj = {};
obj.name = "public name";
outer(obj);
print(obj.name);           //"public name"
```

After executing the above code, the object that was created will have three own properties:

Property Name	Property Value
"name"	"public name"
private $\text{name}_{\text{outer}}$	"outer name"
private $\text{name}_{\text{inner}}$	"inner name"

However, the above is not a very realistic example. After execution of the above code, the two private named properties could not be directly accessed because the private identifier bindings that contain their property names are no longer accessible. More typically a private identifier binding will be shared by several functions (methods) that need to have shared access to a private named property.

Private Declarations Exist in a Parallel Environment

Consider the following very common idiom used in a constructor declaration:

```
function Point(x,y) {
   this.x = x;
   this.y = y;
   //... methods that use x and y properties
}
var pt = new Point(1,2);
```

The identifiers `x` and `y` each have two distinct bindings within the scope of function `Point`. On the right-hand side of the two assignment operators, `x` and `y` are identifier references (ES5 11.1.2) that bind to the formal parameter declarations for `Point`. Accessing them produces the values `1` and `2`. However, on the right-hand side of `.` within the left-hand sides of those assignment expressions, `x` and `y` are used as *IdentifierNames* in Property Accessors (ES5 11.2.1) and bind to the constant string values "`x`" and "`y`".

One way to view this is that there are two distinct naming environments in ES5 programs: one used to resolve identifiers as *PrimaryExpressions* and the other used to resolve identifiers as *PropertyAccessors*. However, in ES5 the environment for resolving *PropertyAccessor* identifiers is not particularly interesting because it is a single contour that simply binds all identifiers to the string value that is the *IdentifierName* of the identifier.

When a private declaration is added to the above example, we need to preserve the same basic semantics that we have in ES5.

```
function Point(x,y) {
   private x, y;
   this.x = x;
   this.y = y;
   //... methods that use private x and y properties
}
var pt = new Point(1,2);
```

On the right-hand side of the assignments `x` and `y` still need to refer to the formal parameter bindings, even though there is a local declaration for private names `x` and `y`. Similarly, on the left-hand side *PropertyAccessors*, `x` and `y` should bind to the private names introduced by the `private` declaration and not bind to the formal parameters.

As with ES5 this can be explained by using distinct naming environments for *PrimaryExpressions* vs. *PropertyAccessors*. However, the property name environment is no longer a flat set of identifier bindings. Instead it is a lexically scoped hierarchy of bindings that map from identifiers either to string values or to private name values. The hierarchical structure parallels the *PrimaryExpression* environment hierarchy.

Another way to view this is that each EnvironmentRecord (ES5 10.2.1) has a second set of bindings that are used to map identifiers to property names. `private` declarations create such bindings in the current environment. Syntactic contexts such as *PropertyAccess* and Object Literal *PropertyName* look up identifiers using a new abstract operations GetPrivateName that is exactly like GetIdentiferReference except that it uses the property name bindings. At the top level is an the set of identifier bindings that map all identifiers to the string values of their identifier names.

Accessing Private Names as Values

The `private` declaration normally both creates a new private name value and introduces a name binding that can be used only in "property name" syntactic contexts to access the new private name value by the lexically bound name.

However, in some circumstances it is necessary to access the actual private name value as an expression value, not as a property name on the right of `.` or the left of `:` in an object initialiser. This requires a special form than can be used in an expression to access the private name value binding of a private identifier. The syntactic form is `#.` *IdentifierName*. This may be used as a *PrimaryExpression* and yields the property name value of the *IdentifierName*. This may be either a private name value or a string value, depending upon whether the expression is within the scope of a `private` declaration for that *IdentifierName*;

```
function addPrivateProperty(obj, init) {
   private pname;     //create a new private name
   obj.pname = init;  //add initialize a property with that private name
   return #.pname;    //return the private name value to the requestor
}

var myObj = {};
var answerKey = addPrivateProperty(myObj, 42);
```

```
print(myObj[answerKey]);   //42,  note that answerKey is a regular variable so [ ] must be used to access the property
//myObj can now be made globally available but answerKey can be selectively passed to privileged code
```

Note that simply assigning a private name value to a variable does not make that variable a private identifier. For example, in the above example, the print statement could not validly be replaced with:

```
print(myObj.answerKey);
```

This would produce "`undefined`" because it would access the non-existent property whose string valued property name would be "`answerKey`". Only identifiers that have been explicitly declared using `private` are private identifiers.

Enabling the use of [ ] with private name values requires a minor change to the ES5 specification. In 11.2.1, step 6 must be changed to call ToPropertyName rather than ToString. `ToPropertyName(name)` is defined as follows:

1. If name is a private name value, return name.
2. Return the result of ToString(name).

This will not change the semantics of any existing JavaScript code because such code will not contain any use of private name values.

The only operators that can be successfully be applied to a private name value are == and === both of which return true when both operands is the same private name value.

? If the decision is make to represent private names using a new ECMAScript language type, then `typeof` can be used to test if a value is a private name value. If an object representation is used then a new built-in function, `Object.isPrivateName(v)`, will be provided to perform that test.

Private name values can be converted to strings using the internal `ToString` abstract operation. However, the string value does not have any correspondence to the identifier in the `private` declaration that created the private name value. The string value produced by such a string conversion is simply "`Private Name`".

?It may be useful to allow "`Private Name`" to be followed by additional implementation dependent text. This might be used to provide additional identifying information such as a source text line number or a unique serial number that would be useful for debugging.

If `#.` is not within the scope of a `private` declaration for its *IdentifierName* then the value produced is the string value of the *IdentifierName*. In other words, `#.` *IdentiferName* always produces the same value as would be used as the property name in a *PropertyAccess* using that same *IdentifierName*.

As an expressive convenience, `private` declarations can be used to associate a private identifier with an already existing private name value. This is done by using a `private` declaration of the form:

`private` *Identifier* **=** *Initialiser* **;**

If *Initialiser* does not evaluate to a private name value, a TypeError exception is thrown. (? *for uniformity, should string values be allowed? In that case, local private name bindings could be string valued.*)

```
private name1;   //value is a new private name
private name2 = #.name1  //name2 can be used to access the same property name as name1
```

Other possible syntactic forms for converting a private identifier to an expression value include:
private *IdentifierName*
(private *IdentifierName***)**
.*IdentifierName*
`` ` ``*IdentifierName*
#`` ` ``*IdentifierName*
#′ *IdentifierName*

Conflict-Free Object Extension Using Private Names

Some JavaScript frameworks and libraries extend built-in objects by adding new properties to built-in prototype objects. For example, a framework might choose to add a `clone` method to `Object.prototype` that may be used to make a copy of any object. Problems occur when two or more frameworks both try to add a method named `clone`. Private names can be used to avoid such conflicts. If each framework uses a private name rather than than a string property name than each can install a `clone` method on `Object.prototype` without interfering with the other.

For example, someone might create library that does recursive deep copying of object structures that was organized something like this:

```
function installCloneLibrary() {
   private clone;   // the private name for clone methods

   // Install clone methods in key built-in prototypes:
   Object.prototype.clone = function () { ... };
   Array.prototype.clone = function () {
       ...
       target[i] = this[i].clone();  // recur on clone method
       ...
   }
   String.prototype.clone = function () {...}
   ...
   return #.clone
}

// Example usage of CloneLibrary:
private clone = installCloneLibrary();
installAnotherLibrary();
var twin = [{a:0}, {b:1}].clone();
```

The above client of the `CloneLibrary` will work even if the other library also defines a method named `clone` on `Object.prototype`. The second library would not have visibility of the private name used for `clone` so it would either use a string property name or a different private name for the method. In either case there would be no conflict with the method defined by `CloneLibrary`.

Enumeration and Reflection

Properties whose property names are private name values have all the same attributes as a property whose property name is a string value and the same defaults attribute are generally used. However, in most cases it is likely that properties defined using private names should not show up in for-in enumerations. For this reason, the semantics of the standard internal `[[Put]]` method are modified for cases where a property is created by `[[Put]]` using a Private Name value as the property name. In this case the `[[Enumerable]]` attribute of the newly created property is initially set to **false**. This change in made ES5 8.12.5, step 6.a.

For example:

```
private b;
var obj = {};
obj.a = 1;
obj.b = 2;
obj.c = 3;

var names = [];
for (var p in obj) names.push(obj[p]);
print(names.toString());    // "1,3" -- private name "b" was not enumerated
```

Properties with private names that are created using object literals also are created with their `[[Enumerable]]` attribute **false**. So `obj` could have been created to produce the same result by saying:

```
private b;
var obj = {
   a: 1,
   b: 2,
   c: 3
}
```

Because object literal properties are specified using `[[DefineOwnProperty]]` rather than `[[Put]]` all property descriptors used in ES5 11.1.5 must be updated to set `[[Enumerable]]` to **false** whenever a *PropertyName* is a private name value.

Need to check all other uses of `[[DefineOwnProperty]]` and determine whether any of them should have special treatment of private name values.

Creating a private named property that is enumerable requires use of `Object.defineProperty` and the **`#.`** prefix. For example:

```
private b;
var obj = {};
obj.a = 1;
obj.b = 2;
Object.defineProperty(obj, #.b, {enumerable: true});
obj.c = 3;
```

```
var names = [];
for (var p in obj) names.push(obj[p]);
print(names.toString());    // "1,2,3" -- private name "b" is now enumerated
```

`Object.prototype.hasOwnProperty` (ES5 15.2.4.5), `Object.prototype.PropertyIsEnumerable` (ES5 15.2.4.7) and the `in` operator (ES5 11.8.7) are all extended to accept private name values in addition to string values as property names. Where they currently call ToString on property names they will instead call ToPropertyName. The `JSON.stringify` algorithm (ES5 15.12.3) will be modified such that it does not process enumerable properties that have private name values as their property names.

All the Object reflection functions defined in ES5 section 15.2.3 that accept property names as arguments or return property names are extended to accept or produce private name values in addition to string values as property names. A private name value may appear as a property name in the collection of property descriptors passed to `Object.create` and `Object.defineProperties`. If an object has private named properties then their private name values will appear in the arrays returned by `Object.getOwnPropertyNames` and `Object.keys` (if the corresponding properties are enumerable).

In ES5 `Object.defineProperties` and `Object.create` are specified to look only at the enumerable own properties of the object that is passed containing property descriptors. Usually this object is specified using an object literal. However, we have already specified above that private named properties in object literals are always created as non-enumerable properties. This would generally preclude the use of private name values in the object literals passed to these methods. It may be necessary to modify the specification of `Object.defineProperty` and `Object.create` to process also as property definitions any non-enumerable private named own properties that appear in such objects.

```
// Object.defineProperty probably needs to accept descriptors such as:
private a, b;
Object.defineProperties(obj, {
   a: {configurable: true}, // ES5 ignores non-enumerable properties
   b: {writable: true}      // that appear in such descriptors
});
```

An important use case for reflection using private name values is algorithms that need to perform meta-level processing of all properties of any object. For example, a “universal” object copy function might be coded as:

```
function copyObject(obj) {
   // This doesn't deal with other special [[Class]] objects:
   var copy = Object.isArray(obj) ? [] : Object.create(Object.getPrototypeOf(obj));
   var props = Object.getOwnPropertyNames(obj);
   var pname;
   for (var i = 0; i < props.length; i++) {
      pname = props[i];
      Object.defineProperty(copy, pname, Object.getOwnPropertyDescriptor(obj,pname));
   }
   return obj;
}
```

This function will duplicate all properties, including any that have private name values as their property names. It does not need to have any specific declarations for or knowledge of such private names.

Private Name Properties Support Only Weak Encapsulation

Private names are a simple and pragmatic way to support everyday encapsulation without requiring programmers to change their fundamental conceptualization of JavaScript objects. Private named properties do not provide and are not intended to provide strong impenetrable encapsulation of object state. For example, various reflection operations can be used to access an object's properties that have private names. Private names are instead intended as a simple extensions of the classic JavaScript object model that enables straight-forward encapsulation in non-hostile environments. The design preserves the ability to manipulate all properties of an objects at a meta level using reflection and the ability to perform "monkey patching" when it is necessary. Encapsulation via closure capture should continue to be used for situations where strong encapsulation that can not be penetrated by a hostile attacker is actually needed.

Sand-boxing environments that already need to restrict the use of certain reflection operations can use similar technique to limit access to private named properties. For example, a sandbox implementation might replace `Object.getOwnPropertyNames` with a version that filters out any private name values.

Interactions with other Harmony Proposals

There are potential feature interactions and opportunities for feature integration involving private names and several other Harmony proposals. As features are accepted into Harmony and their details are filled in these interactions need to be resolved.

Enhanced Object Literals

`private` might be supported as either a property modifier keyword that makes the property name a private name whose private identifier is scoped to the object literal:

```
var obj={
   private _x: 0;
   get x() {return this._x},
   set x(val) {this._x=val}
}
```

This might simplify the declarative creation of objects with instance private properties. However, there are internal scoping and hoisting issues that would need to be considered and resolved.

Another alternative is to use meta property syntax to declare object literal local private name declarations:

```
var obj={
   <prototype: myProto; private _x>
   _x: 0;
   get x() {return this._x},
   set x(val) {this._x=val}
}
```

Proxies

All uses of string valued property names in proxy handlers would need to be extended to accept/produce private name values in addition to string values.

As covered above, ECMAScript reflection capabilities provides a means to break the encapsulation of an object's private named properties. Where this is a concern, it can be mitigated by replacing the reflection functions with versions that filter access to private name values. The Proxy proposal provides an additional means to break such encapsulation. If an attacker suspects that some object has private named properties it might sniff out those values by creating a proxy for the object whose handler consisted of traps that monitored all calls looking for private name argument values before delegating the operation to the original object.

One possible mitigation for this attach would be the same as for the other reflection functions. The Proxy object could be replaced with an alternative implementation that added an additional handler layer that would wrapper all private name values passed through its traps. The wrappers would be opaque encapsulations of each private name value and provide a method that could be used to test whether the encapsulated private name was === to an argument value. This would permit handlers to process known private name values but would prevent exposing arbitrary private name values to the handlers.

If there is sufficient concern about proxies exposing private name values in this manner, such wrapping of private names could be built into the primitive trap invocation mechanism.

Modules

It is reasonable to expect that modules will want to define and export private name values. For example, a module might want to add methods to a built-in prototype object using private names and then make those method names available to other modules. Within the present definition of the simple module system that might be done as follows:

```
<script type="harmony">
module ExtendedObject {
   import Builtins.Object;       // however access to Object is obtained.
   private clone;               // the private name for clone methods
   export const clone = #.clone; // export a constant with the private name value;

   Object.prototype.clone = function () { ... };
}
</script>
```

A consumer of this module might look like:

```
<script type="harmony">
import ExtendedObject.clone;
private clone = clone;
var anotherObj = someObj.clone();
</script>
```

The above formulation would work without any additional extensions to the simple module proposal. However, it would be even more convenient if the module system was extended to understand private declarations and the parallel property name environment. In that case this example might be written as:

```
<script type="harmony">
module ExtendedObject {
   import Builtins.Object;     // however access to Object is obtained.
   export private clone;       // export private name for clone methods

   Object.prototype.clone = function () { ... };
}
</script>
```

```
<script type="harmony">
import private ExtendedObject.clone;
var anotherObj = someObj.clone();
</script>
```

In these example the use of `import` and `export` prefixing `private` declarations forces use of the property name environment of the named module. For dynamic access to the exported property name environment of first-class module instances another mechnism would perhaps be needed:

```
<script type="harmony">
module private ExtendedObject_names = ExtendedObject;

private cloneEX = ExtendedObject_names.clone;  //get private name value bound to ''clone'' in reified module ''ExtendedObject''
var yetAnotherObj = someObj.cloneEX();
</script>
```

References

Private name values are akin to what is returned by `gensym` of Lisp and Scheme, and analogous to a capability in object-capability languages.

The inspiration for `private` is Racket's define-local-member-name and "selector namespaces" for Smalltalk and Ruby .

Explicit Inherited Soft Fields

Table of Contents

- Explicit Inherited Soft Fields
  - A transposed representation
  - Should we tolerate primitive keys?
  - Can we subsume Private Names?
- See

The following derived abstraction combines the explicitness of explicit soft own fields with the visibility across inheritance chains of inherited soft fields. Below is an executable specification as a wrapper around weak maps. This strawman page suggests standardizing this derived abstraction because a primitive implementation is likely to be more efficient that the code below.

As with our previous "`EphemeronTable`", the name "`SoftField`" is only a placeholder until someone suggests an acceptable name.

```
  const SoftField() {
    const weakMap = WeakMap();
    const mascot = {}; // fresh and encapsulated, thus differs from any possible
provided value.
    return Object.freeze({
      get: const(base) {
        while (base !== null) {
          const result = weakMap.get(base);
          if (result !== undefined) {
            return result === mascot ? undefined : result;
          }
          base = Object.getPrototypeOf(base);
        }
        return undefined;
      },
      set: const(key, val) {
        weakMap.set(key, val === undefined ? mascot : val);
      },
      has: const(key) {
        return weakMap.get(key) !== undefined;
      },
      delete: const(key) {
        weakMap.set(key, undefined);
      }
    });
  }
```

A transposed representation

An alternative explanation that implements the same semantics but more closely reflects expected implementation follows. This is no longer quite an executable specification in that it builds on a new internal property, here spelled `SoftFields___`. The safety of the following spec depends on the `SoftFields___` property not being used by any other spec beyond the following.

```
const init___(obj) {
  if (obj !== Object(obj)) { throw new TypeError(...) }
  if (!obj.SoftFields___) {
    obj.SoftFields___ = WeakMap();
  }
}

const SoftField() {
  const mascot = {};
  const get(base) {
    init___(base)
    while (base !== null) {
      const result = base.SoftFields___.get(get);
      if (result !== undefined) {
        return result === mascot ? undefined : result;
      }
      base = Object.getPrototypeOf(base);
    }
    return undefined;
  }
  return Object.freeze({
    get: get,
    set: const(key, val) {
      init___(key);
      key.SoftFields___.set(get, val === undefined ? mascot : val);
    },
    has: const(key) {
      init___(key);
      return key.SoftFields___.get(get) !== undefined;
    },
    delete: const(key) {
      init___(key)
      key.SoftFields___.set(get, undefined);
    }
  });
}
```

The overall logic is very similar, except that the underlying weak maps are now stored on the SoftField's key objects, while each SoftField itself only holds on to the fixed state of the key used to look up values in those weak maps. Even though the above algorithm still manually encodes walking the prototype chain, because this walk is now consulting a map stored within each object, two transparent performance benefits may follow:

- The optimizations already in place for normal property lookup may be more readily adapted to soft field lookup.
- The conventional portion of a GC algorithm that does not take account of weak maps will nevertheless collect that soft state that is only reachable from non-reachable objects, even in the presence of cycles between that soft state and those objects. For soft fields, the weak map portion of a GC algorithm is only needed to collect

those soft fields that can no longer be "named" but are still present on objects that are reachable.

This representation parallels the implementation techniques and resulting performance benefits expected for private names but without the semantic problems (leaking via proxy traps and inability to associate soft state with frozen objects).

Regarding the GC point, when soft fields are used in patterns such as class-private instance variables, a soft field adds soft state to a set of objects, each of whom also points at that soft field itself. In that case, the soft field has a lifetime at least as long as any of the objects it indexes. Thus, the conventional portion of GC algorithms is adequate to pick up all the resulting collectable soft state.

Should we tolerate primitive keys?

Since soft fields – unlike weak maps – look up the key's inheritance chain until it find a match, it makes sense to allow primitive data types (numbers, strings, and booleans, but still not null or undefined) to serve as keys. When used as a key, the operations above would first convert it to an object using the internal [[ToObject]] function. (Unlike `Object`, [[ToObject]] on a null or undefined throws a TypeError.) For strings, numbers, and booleans, this results in a fresh wrapper, which therefore has no soft own state. Lookup would therefore always proceed to the respective prototypes, so that, e.g., a primitive string would seem to inherit soft state from `String.prototype`, much as it currently seems to inherit properties from `String.prototype`.

Can we subsume Private Names?

Two use cases shown at private names that simple soft fields cannot provide are a certain form of polymorphism between names and strings, and so-called "weak encapsulation". (MarkM here suspends value judgements about whether we should seek to support so-called "weak encapsulation", and addresses here only how to do so, were we to agree on its desirability.) If value proxies are accepted for Harmony, then Soft fields can grow to support both these use cases without further expansion of kernel semantics, by defining a soft field as equivalent to a value proxy that overloads [], to whit:

```
const softFieldOpHandler = Object.freeze({
  // overload larg[proxy]
  rgeti: const(larg)        { return this.get(larg); },
  // overload larg[proxy] = val;
  rseti: const(larg, val)   {        this.set(larg, val); }
});
// Move the SoftField code into softFieldProto
const softFieldProto = Object.freeze({
  get:    ..., //as above, but with "this.weakMap" instead of "weakMap"
  set:    ..., //as above
  has:    ..., //as above
  delete: ...  //as above
});
const softFieldValueType = Proxy.createValueType(
    softFieldOpHandler, softFieldProto,
    "string", // bad idea, but suspending judgement
    { weakMap: object });
const SoftFieldValue() {
  return Proxy.createValue(softFieldValueType, new WeakMap());
}
const softFieldValue = SoftFieldValue();
```

(Detail: The above code doesn't quite work as is, because there's no where safe to put the mascot. By delegating to an encapsulated explicit soft own fields instead of a `WeakMap`, we can encapsulate the mascot in this extra layer. This is a detail because it effects only the apparent cost of this executable specification, not the actual cost of an implementation.)

A “weakly encapsulating” soft field, or *wesf*, can then be coded as:

```
// Move the SoftField code into softFieldProto
const wesfProto = Object.freeze({
  toString: const()          { return improbableName; },
  get:      const(key)       { return key[improbableName] },
  set:      const(key, val) {          key[improbableName] = val; },
  has:      const(key)       { return improbableName in key; },
  delete:   const(key)       { return delete key[improbableName]; }
});
const wesfValueType = Proxy.createValueType(
    softFieldOpHandler, wesfProto,
    "string", // bad idea, but suspending judgement
    { improbableName: string });
const WesfValue(opt_name) {
  const name = String(opt_name) || Math.random() + '___';
  return Proxy.createValue(wesfValueType, name);
}
const wesfValue = WesfValue();
```

Then polymorphic code such as

```
function foo(n) {
  return base[n];
}
```

can be called with a softFieldValue, a wesfValue, or a string, where each provides the degree of encapsulation and collision avoidance it claims. This supports the ability to modularly refactor code between non-encapsulating, “weakly” encapsulating, and obviously non-encapsulating fields.

See

The thread beginning at WeakMap API questions?

Older GC discussion now obsolete but still potentially interesting.

strawman/inherited_explicit_soft_fields.txt · Last modified: 2011/01/11 23:55 by markm

Show pagesource Login Index
Old revisions Back to top

[[strawman:
names_vs_soft_fields]]

ES
Wiki

Show pagesource
Old revisions
Recent changes
Search

Trace: » private_names » inherited_explicit_soft_fields » names_vs_soft_fields

Table of Contents

- Overview
- The private declaration
- Using Private Identifiers
- Private Identifiers in Object Literals
- Private Declaration Scoping
- Private Declarations Expand to Unique Hidden Variable Names
- Accessing Private Identifiers as Soft Field Values
- Conflict-Free Object Extension Using Soft Fields
  - Crucial difference
- Enumeration and Reflection
- Soft Fields Support Encapsulation
- Interactions with other Harmony Proposals
  - Enhanced Object Literals
  - Proxies
  - Modules
- References

Overview

To better understand the differences between soft fields and private names, this page goes through all the examples from the latter (as of this writing) and explores how they'd look as translated to use soft fields instead. This translation does not imply endorsement of all elements of the names proposal as translated to soft fields, such as the proposed syntactic extensions. However, these translations do establish that these syntactic choices are orthogonal to the semantic controversy and so can be argued about separately.

Identifiers ending with triple underbar below signify unique identifiers generated by expansion that are known not to conflict with any identifiers that appear elsewhere.

The private declaration

Adapted from the private declaration

```
private secret;  //create a new soft field that is bound
to the private identifier ''secret''.
private _x,_y;   //create two soft fields bound to two
private identifiers
... foo.secret ...
foo.secret = val;
const obj = {secret: val, ...};
#.secret
```

expands to

```
const secret___ = SoftField();
const _x___ = SoftField(), _y___ = SoftField();
... secret___.get(foo) ...
secret___.set(foo, val);
const obj = {...}; secret___.set(obj, val);
secret___
```

Using Private Identifiers

Adapted from using private identifiers

```
function makeObj() {
   private secret;
   var obj = {};
   obj.secret = 42;  //obj has a soft field
   print(obj.secret);//42 -- accesses the soft field's value
   print(obj["secret"]); //undefined -- a soft field is not a property
```

```
    return obj;
}
var obj=makeObj();
print(obj["secret"]); //undefined -- a soft field is still not a property
print(obj.secret);    //undefined -- this statement is not in the scope of the private
declaration so the
                      //string value "secret" is used to look up the property.  It is
not a soft field.
```

This technique can be used to define “instance-private” properties:

```
function Thing() {
    private key;   // each invocation will use a new soft field
    this.key = "instance private value";
    this.hasKey = function(x) {
        return x.key === this.key;  //x.key should be undefined if x!==this
    };
    this.getThingKey = function(x) {
        return x.key;
    };
}
```

Instance-private instance state is better done by lexical capture

```
function Thing() {
    const key = "instance private value";
    this.hasKey = function(x) {
        return x === this;
    };
    this.getThingKey = function(x) {
        if (x === this) { return key; }
    };
}
```

Either technique produces the same external effect:

```
var thing1 = new Thing;
var thing2 = new Thing;

print("key" in thing1);       // false
print(thing2.key);            //undefined
print(thing1.hasKey(thing1)); // true
print(thing1.hasKey(thing2)); // false
```

By changing the scope of the private declaration a similar technique can be used to define “class-private” properties:

```
private key;  //the a soft field shared by all instances of Thing.
function Thing() {
    this.key = "class private value";
    this.hasKey = function(x) {
        return x.key === this.key;
    };
    this.getThingKey = function(x) {
        return x.key;
    };
}

var thing1 = new Thing;
var thing2 = new Thing;

print("key" in thing1);       // false
print(thing1.hasKey(thing1)); // true
print(thing1.hasKey(thing2)); // true
```

Private Identifiers in Object Literals

Adapted from private identifiers in object literals

```
function makeObj() {
   private secret;
   var obj = {secret: 42};
   print(obj.secret);//42 -- access the soft field's value
   print(obj["secret"]); //undefined -- a soft field is not a property
   return obj;
}
```

```
function Thing() {
    private key;
    return {
       key : "instance private value",
       hasKey : function(x) {
          return x.key === this.key;  //x.key should be undefined if x!==this
       },
       getThingKey : function(x) {
          return x.key;
       }
    };
}
```

or, preserving the same external behavior:

```
function Thing() {
    const key = "instance private value";
    return {
       hasKey : function(x) {
          return x === this;
       },
       getThingKey : function(x) {
          if (x === this) { return key; }
       }
    };
}
```

```
private key;
function Thing() {
    return {
       key : "class private value",
       hasKey : function(x) {
          return x.key === this.key;  //x.key should be undefined if x!==this
       },
       getThingKey : function(x) {
          return x.key;
       }
    };
}
```

Private Declaration Scoping

Adapted from private declaration scoping

```
function outer(obj) {
   private name;
   function inner(obj) {
      private name;
      obj.name = "inner name";
      print(obj.name);   //"inner name" because outer name declaration is shadowed
   }
   obj.name = "outer name";
   inner(obj)
   print(obj.name);      //"outer name"
}
var obj = {};
obj.name = "public name";
outer(obj);
print(obj.name);           //"public name"
```

After executing the above code, the object that was created will have one property and two associated soft fields:

Property or Fields	Value
"name"	"public name"
private name$_{outer}$	"outer name"
private name$_{inner}$	"inner name"

Private Declarations Expand to Unique Hidden Variable Names

Adapted from private declarations exist in a parallel environment

Consider the following very common idiom used in a constructor declaration:

```
function Point(x,y) {
   this.x = x;
   this.y = y;
   //... methods that use x and y properties
}
var pt = new Point(1,2);
```

```
function Point(x,y) {
   private x, y;
   this.x = x;
   this.y = y;
   //... methods that use private x and y properties
}
var pt = new Point(1,2);
```

```
function Point(x,y) {
   const x___ = SoftField(), y___ = SoftField();
   x___.set(this, x);
   y___.set(this, y);
   //... methods that use private x and y properties
}
var pt = new Point(1,2);
```

Accessing Private Identifiers as Soft Field Values

Adapted from accessing private names as values

The `private` declaration normally both creates a new soft field and introduces a identifier binding that can be used only in "property name" syntactic contexts to access the new soft field by the lexically bound identifier.

However, in some circumstances it is necessary to access the actual soft field as an expression value, not as an apparent property name on the right of **.** or the left of **:** in an object initialiser. This requires a special form than can be used in an expression to access the soft field binding of a private identifier. The syntactic form is `#.` *IdentifierName*. This may be used as a *PrimaryExpression* and yields the soft field of the *IdentifierName*. This may be either a soft field or a string value, depending upon whether the expression is within the scope of a `private` declaration for that *IdentifierName*;

```
function addPrivateProperty(obj, init) {
   private pname;     //create a new soft field
   obj.pname = init;  //set this soft field
   return #.pname;    //return the soft field
}
```

```
function addPrivateProperty(obj, init) {
   const pname___ = SoftField();
   pname___.set(obj, init);
   return pname___;
}
```

```
var myObj = {};
var answerKey = addPrivateProperty(myObj, 42);
print(answerKey.get(myObj));  // AFAICT, this is the *only* claimed advantage of Names
over SoftFields.
//myObj can now be made globally available but answerKey can be selectively passed to
privileged code
```

Note that simply assigning a soft field to a variable does not make that variable a private identifier. For example, in the above example, the print statement could not validly be replaced with:

```
print(myObj.answerKey);
```

This would produce "`undefined`" because it would access the non-existent property whose string valued property name would be "`answerKey`". Only identifiers that have been explicitly declared using `private` are private identifiers.

"can we subsume private names" explains how soft fields as value proxies could support a property-like usage of [], so this code could indeed be written as

```
print(myObj[answerKey]);
```

If `#.` is not within the scope of a `private` declaration for its *IdentifierName* then the value produced is the string value of the *IdentifierName*.

As an expressive convenience, `private` declarations can be used to associate a private identifier with an already existing soft field. This is done by using a `private` declaration of the form:

`private` *Identifier* **=** *Initialiser* **;**

The Names proposal asks: "If *Initialiser* does not evaluate to a soft field, a TypeError exception is thrown. (? *for uniformity, should string values be allowed? In that case, local private name bindings could be string valued.*)"

If the answer is true, the one supposed advantage of Names over soft fields goes away. Our contentious bit of code becomes:

```
private ak = answerKey; // soft field or string
print(obj.ak); // works either way
```

```
private name1;   //value is a new soft field
private name2 = #.name1  //name2 can be used to access the same soft field as name1
```

Other possible syntactic forms for converting a private identifier to an expression value include:
`private` *IdentifierName*
`(private` *IdentifierName***`)`**
`.`*IdentifierName*
`` ` ``*IdentifierName*
`` #` ``*IdentifierName*
`#'` *IdentifierName*

Conflict-Free Object Extension Using Soft Fields

Adapted from conflict-free object extension using private names

```
function installCloneLibrary() {
   private clone;   // the soft field for clone methods

   // Install clone methods in key built-in prototypes:
   Object.prototype.clone = function () { ... };
   Array.prototype.clone = function () {
        ...
        target[i] = this[i].clone();  // recur on clone method
        ...
    }
    String.prototype.clone = function () {...}
    ...
    return #.clone
}

// Example usage of CloneLibrary:
private clone = installCloneLibrary();
installAnotherLibrary();
var twin = [{a:0}, {b:1}].clone();
```

Similarities: The above client of the `CloneLibrary` will work even if the other library also defines a method named `clone` on `Object.prototype`. The second library would not have visibility of the soft field used for `clone` so it would either use a string property name or a different soft field for the method. In either case there would be no conflict with the method defined by `CloneLibrary`.

Crucial difference

For defensive programming, best practice in many environments will be to freeze the primordials early, as the dual of the existing best practice that one should not mutate the primordials. Evaluating the dynamic behaviour of Python applications (See also http://gnuu.org/2010/12/13/too-lazy-to-type/) provides evidence that this will be compatible with much existing content. We should expect these best practices to grow during the time when people feel they can target ES5 but not yet ES6.

Consider if Object.prototype or Array.prototype were already frozen, as they should be, before the code above executes. Using soft fields, this extension works. Using private names, it is rejected. Allen argues at Private names use cases that

```
    Allow third-party property extensions to built-in
    objects or third-party frameworks that are guaranteed
    to not have naming conflicts with unrelated extensions
    to the same objects.
```

is the more important use case. Soft fields provide for this use case. Private names do not.

Who knows whether frozen primordials will catch on? Many JS hackers are vehemently opposed. PrototypeJS still extends built-in prototypes and its maintainers say that won't change. Allen clearly was talking about extending non-frozen shared objects in his "Private names use cases" message – he did not assume what you assume here. We need to agree on our assumptions before putting forth conclusions that we hope will be shared. I don't think everyone shares the belief that "We should expect these best practices to grow during [any foreseeable future]."

— *Brendan Eich 2010/12/22 01:37*

Are we still confusing "any" and "all"? The original quote claims only that these best practices will grow in some environments. Regarding your "any foreseeable future", this future is already long past. Google JavaScript Style Guide: Modifying prototypes of builtin objects has long stated:

```
  Modifying prototypes of builtin objects
  [Recommendation:] No
  Modifying builtins like Object.prototype and Array.prototype
  are strictly forbidden. Modifying other builtins like
  Function.prototype is less dangerous but still leads to hard
  to debug issues in production and should be avoided.
```

I'm sure other such quotes about JavaScript best practice can be found.

Also, of course, The last initialization step of initSES is to freeze the primordials of its frame. Only code that does not mutate their primordials will be directly compatible with SES without result to sandboxing.

Mark, the original quote from you is visible above, and it asserts "many", not "any". That is a bold claim. Not only Prototype, but SproutCore and Moo (and probably others), extend standard objects. SproutCore adds a `w` method to `String.prototype`, along with many other methods inspired by Ruby.

It's nice that Google has recommendations, which it can indeed enforce as mandates on employees, but the Web at large is under no such authority. The Web is the relevant context for quantifying "many", not some number of secure subset languages used in far smaller domains. On the Web, it's hard to rule out maintainers and reusers mixing your code with SproutCore, e.g.

SES is a different language from Harmony, not standardized by Harmony in full. Goal 5 at harmony is about supporting SES, not subsuming it.

I believe we should avoid trying to run social experiments, building up pedagogical regimes, or making predictions about the future, anywhere in the text of future ECMA-262 editions.

— *Brendan Eich 2011/01/12 02:12*

Enumeration and Reflection

enumeration and reflection

Even though soft fields are typically implemented as state within the object they extends, because soft fields are semantically not properties of the object but are rather side tables, they do not show up in reflective operations performed on the object itself.

For example:

```
private b;
var obj = {};
obj.a = 1;
obj.b = 2;
obj.c = 3;

var names = [];
for (var p in obj) names.push(obj[p]);
print(names.toString());    // "1,3" -- soft field "b" was not enumerated
```

Soft fields created using object literals also not part of the object itself. So `obj` could have been created to produce the same result by saying:

```
private b;
var obj = {
   a: 1,
   b: 2,
   c: 3
}
```

Beyond the syntactic expansions explained above, no other change to the definition of object literals is needed.

Creating a soft field that is enumerable makes no sense. Reflective operations that take property names as arguments, such as Object.defineProperty below, if given a non-string argument including a soft field, would coerce it to string and (uselessly) use that as a property name.

```
private b;
var obj = {};
obj.a = 1;
obj.b = 2;
Object.defineProperty(obj, #.b, {enumerable: true});
obj.c = 3;

var names = [];
for (var p in obj) names.push(obj[p]);
print(names.toString());    // "1,2,3" -- property "[object Object]" is now enumerated
```

`Object.prototype.hasOwnProperty` (ES5 15.2.4.5), `Object.prototype.propertyIsEnumerable` (ES5 15.2.4.7) and the `in` operator (ES5 11.8.7) do not see soft fields, again, because they are not part of the object.

The `JSON.stringify` algorithm (ES5 15.12.3) needs no change in order to ignore soft fields, since again they are not part of the object.

All the Object reflection functions defined in ES5 section 15.2.3 remain unchanged, since they need not be aware of soft fields.

An important use case for reflection using soft fields is algorithms that need to perform meta-level processing of all properties of any object. For example, a "universal" object copy function might be coded as:

```
function copyObject(obj) {
   // This doesn't deal with other special [[Class]] objects:
   var copy = Object.isArray(obj) ? [] : Object.create(Object.getPrototypeOf(obj));
   var props = Object.getOwnPropertyNames(obj);
   var pname;
   for (var i = 0; i < props.length; i++) {
      pname = props[i];
      Object.defineProperty(copy, pname, Object.getOwnPropertyDescriptor(obj,pname));
   }
   return obj;
}
```

This function will duplicate all properties but not any soft fields, preserving encapsulation, since neither the definer nor the caller of copyObject knows these soft fields. Of course, a more complex copyObject function could be defined that would also copy and re-index those soft fields it was told of.

Soft Fields Support Encapsulation

Adapted from private name properties support only weak encapsulation

No qualifiers needed.

Should so-called "weak encapsulation" actually be desired, "can we subsume private names" explains how to provide *weakly encapsulating soft fields* (or "wesf") polymorphically with soft fields.

Interactions with other Harmony Proposals

Enhanced Object Literals

Adapted from enhanced object literals

`private` might be supported as either a property modifier keyword that makes the property name a soft field whose private identifier is scoped to the object literal:

```
var obj={
   private _x: 0;
   get x() {return this._x},
   set x(val) {this._x=val}
}
```

This might simplify the declarative creation of objects with instance private soft fields. However, there are internal scoping and hoisting issues that would need to be considered and resolved.

Another alternative is to use meta property syntax to declare object literal local soft field declarations:

```
var obj={
   <prototype: myProto; private _x>
   _x: 0;
   get x() {return this._x},
   set x(val) {this._x=val}
}
```

While the above proposals are perfectly consistent with soft fields, again, for instance-private instance state, using lexical capture seems strictly superior:

```
let x = 0;
var obj={
   get x() {return x},
   set x(val) {x=val}
}
```

Proxies

Adapted from proxies

None of the uses of string valued property names in proxy handlers would need to be extended to accept/produce soft fields in addition to string values.

As covered above, ECMAScript reflection capabilities provides no means to break the encapsulation of an object's soft fields.

Modules

Adapted from modules

It is reasonable to expect that modules will want to define and export soft fields. For example, a module might want to add methods to a built-in prototype object using soft fields and then make those soft fields available to other modules. Within the present definition of the simple module system that might be done as follows:

```
<script type="harmony">
module ExtendedObject {
   import Builtins.Object;       // however access to Object is obtained.
   private clone;                // the soft field for clone methods
   export const clone = #.clone; // export a constant with the soft field;

   Object.prototype.clone = function () { ... };
}
</script>
```

A consumer of this module might look like:

```
<script type="harmony">
import ExtendedObject.clone;
private clone = clone;
var anotherObj = someObj.clone();
</script>
```

The above formulation would work without any additional extensions to the simple module proposal. However, it would be even more convenient if the module system was extended to understand private declarations. In that case this example might be written as:

```
<script type="harmony">
module ExtendedObject {
   import Builtins.Object;     // however access to Object is obtained.
   export private clone;       // export soft field for clone methods

   Object.prototype.clone = function () { ... };
}
</script>
```

```
<script type="harmony">
import private ExtendedObject.clone;
var anotherObj = someObj.clone();
</script>
```

I don't get the point about "dynamic access to the exported property name environment of first-class module instances", so at this time I offer no comparison of this last example.

References

Adapted from references

Any unforgeable reference to a tamper-proof encapsulated object is analogous to a capability in object-capability languages. In this degenerate sense, both Names and Soft Fields are also so analogous. I see no further way in which Names are analogous. In addition, Soft Fields encourage encapsulation friendly patterns, whereas Names encourage unsafe (or "weakly encapsulated") patterns.

[[strawman:
guards]]

ES
Wiki

Guards

Table of Contents

- Guards
  - Guarding Variables
  - Guarding Parameters and Results
  - Guarding Properties
- Open Issues
- See

For checking purposes, guards are used as the runtime analog of types. Use guards to annotate variable and parameter declarations, function return results, and property definitions. Each guard is asked to approve an incoming value, the *specimen*, to determine whether to

- let it pass,
- reject it, or
- coerce it to a value it would approve.

```
Guard :
    :: GuardExpression
GuardExpression :
    CallExpression
```

A Guard annotation evaluates its GuardExpression to a value approximately as any expression would be evaluated to a value in that scope. (See clarifications below for which scope that is, and when the evaluation occurs.) A specimen is always coerced by the guard value before being bound or returned. The meaning of a program can correctly rely on the presence of these guards. The coercion is performed by the internal function `Coerce___`. The general contract of `Coerce___` is

```
const Coerce___(guard, specimen, opt_exit) {
  // if guard is not an acceptable guard value, throw a TypeError.
  // if specimen is acceptable as an output of this guard, return specimen.
  // if specimen is acceptable as an input to this guard,
  //   then return a corresponding value that would be acceptable as an output of this
guard.
  // if opt_exit is truthy, throw opt_exit
  // throw a TypeError
}
```

Those proposals (like trademarks) that wish to build on the guard syntax would parameterize this proposal by defining the concrete behavior of the internal `Coerce___` function, where that behavior is within the abstract contract above. For example, you could define a `Coerce___` function that will only admit or reject but never coerce.

Guarding Variables

Here, we define the ConstDeclaration and LetDeclaration from block_scoped_bindings to generalize the defining position from an Identifier to a Pattern. This is a somewhat different factoring than the Pattern at destructuring, but the "…" below should include all the Pattern productions there.

We do it this way, rather than extend the ES5 VariableDeclaration, since we are not trying to enhance the deprecated VariableStatement.

```
ConstDeclataion :
    const Pattern = AssignmentExpression
LetDeclaration :
    let Pattern = AssignmentExpression
Pattern : ... //
    Identifier Guard?
```

The guard expression, if present, is the (only?) portion of the pattern which is actually evaluated in left to right order, and therefore before the right hand side AssignmentExpression.

A guarded const variable simply inserts an initialize-time check:

```
  const x ::G = y;

// expands to

  const x = Coerce___(G, y);
```

A guarded let variable inserts the corresponding check on initialization and on all assignments to this variable. Therefore, reading the variable, which is not translated, can only either

- throw a ReferenceError if read before initialization, or
- yield a value which `Coerce___` considers acceptable as an output of the guard.

```
  let x ::G = y;
  ... x ... // x as an rvalue
  ... (x = z) ..., x as an lvalue

// expands to

  const G___ = G; // evaluate G only once here
  let x = Coerce___(G___, y);
  ... x ... // reading x is unaffected
  ... (x = Coerce___(G___, z)) ... // assignments are rewritten
```

Guarding Parameters and Results

```
FunctionDeclaration :
    function Identifier FunctionHead { FunctionBody }
    const Identifier FunctionHead { FunctionBody }
FunctionExpression :
    function Identifier? FunctionHead { FunctionBody }
    const Identifier? FunctionHead { FunctionBody }
FunctionHead :
    ( FormalParameterList? ) Guard?

FormalParameterList :
        FormalParameter
        FormalParameterList , FormalParameter
        // extend for optional and rest parameters
    FormalParameter :
        const? Pattern
    Pattern : ...
        Identifier Guard?
```

When a FormalParameter variable is annotated with a Guard, the corresponding argument value is first coerced through the guard and, if successful, the resulting coerced value is bound to the parameter. When a Guard appears after the close paren of a FunctionHead, then whatever value would be returned is coerced using that guard and, if successful, the resulting coerced value is returned instead.

Guard expressions on function parameters and results are *not* evaluated when the function is called, but rather when the function definition is evaluated. Thus, they are *not* evaluated in the scope defined by this function, but rather in the scope in which this function definition itself is evaluated. The resulting guard values are therefore captured by the created function.

```
  function foo(x ::G1) ::G2 { return x; }

// expands to

  const G1___ = G1;
  const G2___ = G2;
  function foo(x___) {
    let x ::G1___ = x___;
    return Coerce___(G2___, x);
  }
```

Guarding Properties

The reason our guard syntax uses "`::`" rather than the "`:`" of ES4 is so that we can annotate property definitions in object literals.

```
  PropertyAssignment : ...
      PropertyName Guard? : AssignmentExpression
```

As with function parameter and result guards, literal property guards are evaluated in the scope in which the object literal appears, and are evaluated prior to evaluating the object literal. The annotated property itself turns into an accessor property whose setter enforces the guard. Thus, the values of these guards are captured by these generated setters. Deviating from the expansion shown below, the actual generated getter and setters functions would be const functions.

```
  ({ foo ::G : 33 })

// expands to

  let foo___ ::G = 33;
  ({ get foo() { return foo___; }, // actually would be const getter
     set foo(newFoo___) { foo___ = newFoo___; } // actually would be const setter
  })

// expands to

  let G___ = G;
  let foo___ = Coerce___(G___, 33);
  ({ get foo() { return foo___; }, // actually would be const getter
     set foo(newFoo___) { foo___ = Coerce___(G___, newFoo___); } // actually would be
const setter
  })
```

Open Issues

We should find a better syntax than "::" which does not create ambiguities, and which can still annotate properties in object literals. "@" might work. Since guard expressions are annotations, this might become memorable by virtue of the Java precedent for annotation syntax. I admit it's a reach, but there aren't many choices.

Other prominent proposals from the es-discuss thread are ":", where guarded property names in object literals would be parenthesized, and ":", where only an object literal as a whole, rather than individual properties, would be guarded.

See

trademarks (not ready yet)

classes with trait composition

email thread

[[strawman: private_names]]

ES Wiki

Show pagesource
Old revisions
Trace: » inherited_explicit_soft_fields » names_vs_soft_fields » guards » private_names
Recent changes
Search

Overview

Table of Contents

- Overview
- Private Name values
- The private declaration
- Using Private Identifiers
- Private Identifiers in Object Literals
- Private Declaration Scoping
- Private Declarations Exist in a Parallel Environment
- Accessing Private Names as Values
- Conflict-Free Object Extension Using Private Names
- Enumeration and Reflection
- Private Name Properties Support Only Weak Encapsulation
- Interactions with other Harmony Proposals
  - Enhanced Object Literals
  - Proxies
  - Modules
- References

2010/12/08 15:16
Revised proposal by Allen Wirfs-Brock
Original proposal by Dave Herman and Sam Tobin-Hochstadt is here.

In existing ECMAScript, it is not possible to create properties that have limited or controlled accessibility. It is possible to create non-enumerable properties, but they can still be discovered by guessing their string-valued property name. The proposed es4 facility for addressing this shortcoming was namespaces, which were complex and suffered from ambiguity and efficiency problems.

This strawman proposes three related changes to support hidden properties.

1. a new, ECMAScript language type (or possibly object `[Class]`) *Private Name*
2. generalizing the ES5 *property name* concept to include either a string (as in ES5) or a *Private Name* value
3. a `private` keyword for automatic use of *Private Name* values instead of strings in syntactic contexts in a lexically scoped fashion.

In addition to creating hidden properties, this also allows properties to be added to existing objects without the possibility of interference with the existing properties, or with other additions by any other code.

Private Name values

`Private Name` is a new ECMAScript language type that will be defined in section 8 of the specification. The `Private Name` type is an open set of distinct `Private Name` values that can be used as the names of object properties. `Private Name` values do not have any corresponding literal representation within ECMAScript code. Distinct `Private Name` values are created by the `CreatePrivateName` abstract operation. Each call to `CreatePrivateName` returns a new distinct `Private Name` value. If `x` and `y` are `Private Name` values then the abstract operation `SameValue(x,y)` returns true if and only `x` and `y` are the same `Private Name` value created by a specific call to `CreatePrivateName`.

A `Private Name` value can be used as the value of the `P` argument to any of the object internal methods defined in section 8.12 of the ECMAScript specification.

If a new ECMAScript type is added then the `typeof` operator will also need to be extended to return a new string value that identifies values of that type. Concerns have been expressed that extending `typeof` in this manner could break existing code that expects to deal with a fixed set of `typeof` values. We need to make a global decision about adding new non-object types and the impact upon `typeof`.

Object values could be used as an alternative to defining a new ECMAScript type to represent `Private Name` values. Each `Private Name` would simply be a distinct object. `Private Name` objects would have a distinct `[Class]` value. To avoid such objects being used for back-channel communication or property garbage dumps they should be created frozen. Conceivably they could all have `null` as their prototype. This may have some benefit if such names are passed between global contexts as it would prevent use of their prototype value as means of identifying their origin context. Throughout the rest of this spec. "private name value" should be read as meaning whichever form is ultimately used.

The private declaration

The `private` declaration creates a new `Private Name` value by calling `CreatePrivateName` and binds that value to a program identifier that may be used in specific syntactic contexts within the lexical scope of the declaration. An identifier that appears in a `private` declaration is call a `private identifier`.

```
private secret;  //create a new ''Private Name'' that is bound to the private identifier
''secret''.
private _x,_y;   //create two ''Private Name'' values bound to two private identifiers
```

Can the names defined in a `private` declaration be any *IdentifierName* or should they be restricted to being an *Identifier* (ie, not a reserved name)? ES5 allows any *IdentifierName* to be used after a dot or as a property name in an object literal so it may be reasonable to allow any *IdetnifierName*. However that will permit strange look formulations such as: `private private;`

Using Private Identifiers

When a private identifier appears as the *IdentifierName* of a *CallExpression* : *CallExpression* . *IdentifierName* production or of a *CallExpression* : *CallExpression* . *IdentifierName* production, the `Private Name` value that is bound to the private identifier is used as the value of the *IdentifierName*. If the identifier in one of these productions is not a private identifier then the identifier name string is used as the value of *IdentifierName*, just as in ECMAScript 5.

This permits object properties to be created whose names are `Private Name` values. It also allows for the values of such properties to be accessed.

```
function makeObj() {
   private secret;
   var obj = {};
   obj.secret = 42;  //obj has a single property whose name is a Private Name value
   print(obj.secret);//42 -- the private identifier can be used in scope to access the
property's value
   print(obj["secret"]); //undefined -- the name of the property is not the string
"secret"
   return obj;
}
```

```
var obj=makeObj();
print(obj["secret"]); //undefined -- the name of the property is still not the string
"secret"
print(obj.secret);    //undefined -- this statement is not in the scope of the private
declaration so the
                      //string value "secret" is used to look up the property.  It does
not match the Private Name value
```

This technique can be used to define "instance-private" properties:

```
function Thing() {
    private key;   // each invocation will use a new private key value
    this.key = "instance private value";
    this.hasKey = function(x) {
        return x.key === this.key;  //x.key should be undefined if x!==this
    };
    this.getThingKey = function(x) {
        return x.key;
    };
}

var thing1 = new Thing;
var thing2 = new Thing;

print("key" in thing1);       // false
print(thing2.key);            //undefined
print(thing1.hasKey(thing1)); // true
print(thing1.hasKey(thing2)); // false
```

By changing the scope of the private declaration a similar technique can be used to define "class-private" properties:

```
private key;  //the same private name value is used by every invocation of Thing
function Thing() {
    this.key = "class private value";
    this.hasKey = function(x) {
        return x.key === this.key;
    };
    this.getThingKey = function(x) {
        return x.key;
    };
}

var thing1 = new Thing;
var thing2 = new Thing;

print("key" in thing1);       // false
print(thing1.hasKey(thing1)); // true
print(thing1.hasKey(thing2)); // true
```

Friend visibility similar to that provided by c++ can be obtained by using private declarations that are visible to several related object literals, object constructors or factory functions enclosed in an outer function and returned from it (directly, or stored as effects in objects).

Private Identifiers in Object Literals

A private identifier may also appear as the *IdentifierName* of a *PropertyName* production in an *ObjectLiteral*. If the identifier in such a productions is not a private identifier then the identifier name string is used as the value of *IdentifierName*, just as in ECMAScript 5.

With this feature, object literals can be used as an alternative expression of the previous three examples:

```
function makeObj() {
   private secret;
   var obj = {secret: 42};
   print(obj.secret);//42 -- the private identifier can be used in scope to access the
property's value
   print(obj["secret"]); //undefined -- the name of the property is not the string
"secret"
   return obj;
}
```

```
function Thing() {
    private key;
    return {
       key : "instance private value",
       hasKey : function(x) {
          return x.key === this.key;  //x.key should be undefined if x!==this
       },
       getThingKey : function(x) {
          return x.key;
       }
    };
}
```

```
private key;
function Thing() {
    return {
       key : "class private value",
       hasKey : function(x) {
          return x.key === this.key;  //x.key should be undefined if x!==this
       },
       getThingKey : function(x) {
          return x.key;
       }
    };
}
```

Private Declaration Scoping

`private` declarations are lexically scoped, like all declarations in Harmony. Inner `private` declarations shadow access to like-named `private` declarations in outer scopes. Within a block, the scoping rules for `private` declarations are the same as for `const` declarations.

```
function outer(obj) {
   private name;
   function inner(obj) {
      private name;
      obj.name = "inner name";
      print(obj.name);   //"inner name" because outer name declaration is shadowed
   }
   obj.name = "outer name";
   inner(obj)
   print(obj.name);      //"outer name"
}
var obj = {};
obj.name = "public name";
outer(obj);
print(obj.name);            //"public name"
```

After executing the above code, the object that was created will have three own properties:

Property Name	Property Value
"name"	"public name"
private name$_{outer}$	"outer name"
private name$_{inner}$	"inner name"

However, the above is not a very realistic example. After execution of the above code, the two private named properties could not be directly accessed because the private identifier bindings that contain their property names are no longer accessible. More typically a private identifier binding will be shared by several functions (methods) that need to have shared access to a private named property.

Private Declarations Exist in a Parallel Environment

Consider the following very common idiom used in a constructor declaration:

```
function Point(x,y) {
   this.x = x;
   this.y = y;
   //... methods that use x and y properties
}
var pt = new Point(1,2);
```

The identifiers `x` and `y` each have two distinct bindings within the scope of function `Point`. On the right-hand side of the two assignment operators, `x` and `y` are identifier references (ES5 11.1.2) that bind to the formal parameter declarations for `Point`. Accessing them produces the values `1` and `2`. However, on the right-hand side of `.` within the left-hand sides of those assignment expressions, `x` and `y` are used as *IdentifierNames* in Property Accessors (ES5 11.2.1) and bind to the constant string values "`x`" and "`y`".

One way to view this is that there are two distinct naming environments in ES5 programs: one used to resolve identifiers as *PrimaryExpressions* and the other used to resolve identifiers as *PropertyAccessors*. However, in ES5 the environment for resolving *PropertyAccessor* identifiers is not particularly interesting because it is a single contour that simply binds all identifiers to the string value that is the *IdentifierName* of the identifier.

When a private declaration is added to the above example, we need to preserve the same basic semantics that we have in ES5.

```
function Point(x,y) {
   private x, y;
   this.x = x;
   this.y = y;
   //... methods that use private x and y properties
}
var pt = new Point(1,2);
```

On the right-hand side of the assignments `x` and `y` still need to refer to the formal parameter bindings, even though there is a local declaration for private names `x` and `y`. Similarly, on the left-hand side *PropertyAccessors*, `x` and `y` should bind to the private names introduced by the `private` declaration and not bind to the formal parameters.

As with ES5 this can be explained by using distinct naming environments for *PrimaryExpressions* vs. *PropertyAccessors*. However, the property name environment is no longer a flat set of identifier bindings. Instead it is a lexically scoped hierarchy of bindings that map from identifiers either to string values or to private name values. The hierarchical structure parallels the *PrimaryExpression* environment hierarchy.

Another way to view this is that each EnvironmentRecord (ES5 10.2.1) has a second set of bindings that are used to map identifiers to property names. `private` declarations create such bindings in the current environment. Syntactic contexts such as *PropertyAccess* and Object Literal *PropertyName* look up identifiers using a new abstract operations GetPrivateName that is exactly like GetIdentiferReference except that it uses the property name bindings. At the top level is an the set of identifier bindings that map all identifiers to the string values of their identifier names.

Accessing Private Names as Values

The `private` declaration normally both creates a new private name value and introduces a name binding that can be used only in "property name" syntactic contexts to access the new private name value by the lexically bound name.

However, in some circumstances it is necessary to access the actual private name value as an expression value, not as a property name on the right of `.` or the left of `:` in an object initialiser. This requires a special form than can be used in an expression to access the private name value binding of a private identifier. The syntactic form is `#.` *IdentifierName*. This may be used as a *PrimaryExpression* and yields the property name value of the *IdentifierName*. This may be either a private name value or a string value, depending upon whether the expression is within the scope of a `private` declaration for that *IdentifierName*;

```
function addPrivateProperty(obj, init) {
   private pname;     //create a new private name
   obj.pname = init;  //add initialize a property with that private name
   return #.pname;    //return the private name value to the requestor
}

var myObj = {};
var answerKey = addPrivateProperty(myObj, 42);
print(myObj[answerKey]);   //42,  note that answerKey is a regular variable so [ ] must be used to access the property
//myObj can now be made globally available but answerKey can be selectively passed to privileged code
```

Note that simply assigning a private name value to a variable does not make that variable a private identifier. For example, in the above example, the print statement could not validly be replaced with:

```
print(myObj.answerKey);
```

This would produce "`undefined`" because it would access the non-existent property whose string valued property name would be "`answerKey`". Only identifiers that have been explicitly declared using `private` are private identifiers.

Enabling the use of [ ] with private name values requires a minor change to the ES5 specification. In 11.2.1, step 6 must be changed to call ToPropertyName rather than ToString. `ToPropertyName(name)` is defined as follows:

1. If name is a private name value, return name.
2. Return the result of ToString(name).

This will not change the semantics of any existing JavaScript code because such code will not contain any use of private name values.

The only operators that can be successfully be applied to a private name value are == and === both of which return true when both operands is the same private name value.

? If the decision is make to represent private names using a new ECMAScript language type, then `typeof` can be used to test if a value is a private name value. If an object representation is used then a new built-in function, `Object.isPrivateName(v)`, will be provided to perform that test.

Private name values can be converted to strings using the internal `ToString` abstract operation. However, the string value does not have any correspondence to the identifier in the `private` declaration that created the private name value. The string value produced by such a string conversion is simply "`Private Name`".

?It may be useful to allow "`Private Name`" to be followed by additional implementation dependent text. This might be used to provide additional identifying information such as a source text line number or a unique serial number that would be useful for debugging.

If `#.` is not within the scope of a `private` declaration for its *IdentifierName* then the value produced is the string value of the *IdentifierName*. In other words, `#.` *IdentiferName* always produces the same value as would be used as the property name in a *PropertyAccess* using that same *IdentifierName*.

As an expressive convenience, `private` declarations can be used to associate a private identifier with an already existing private name value. This is done by using a `private` declaration of the form:

private *Identifier* **=** *Initialiser* **;**

If *Initialiser* does not evaluate to a private name value, a TypeError exception is thrown. (? *for uniformity, should string values be allowed? In that case, local private name bindings could be string valued.*)

```
private name1;   //value is a new private name
private name2 = #.name1  //name2 can be used to access the same property name as name1
```

Other possible syntactic forms for converting a private identifier to an expression value include:
private *IdentifierName*
(private *IdentifierName***)**
.*IdentifierName*
`*IdentifierName*
#`*IdentifierName*
#'*IdentifierName*

Conflict-Free Object Extension Using Private Names

Some JavaScript frameworks and libraries extend built-in objects by adding new properties to built-in prototype objects. For example, a framework might choose to add a `clone` method to `Object.prototype` that may be used to make a copy of any object. Problems occur when two or more frameworks both try to add a method named `clone`. Private names can be used to avoid such conflicts. If each framework uses a private name rather than than a string property name than each can install a `clone` method on `Object.prototype` without interfering with the other.

For example, someone might create library that does recursive deep copying of object structures that was organized something like this:

```
function installCloneLibrary() {
   private clone;   // the private name for clone methods

   // Install clone methods in key built-in prototypes:
   Object.prototype.clone = function () { ... };
   Array.prototype.clone = function () {
       ...
       target[i] = this[i].clone();  // recur on clone method
       ...
   }
   String.prototype.clone = function () {...}
   ...
   return #.clone
}

// Example usage of CloneLibrary:
private clone = installCloneLibrary();
installAnotherLibrary();
var twin = [{a:0}, {b:1}].clone();
```

The above client of the `CloneLibrary` will work even if the other library also defines a method named `clone` on `Object.prototype`. The second library would not have visibility of the private name used for `clone` so it would either use a string property name or a different private name for the method. In either case there would be no conflict with the method defined by `CloneLibrary`.

Enumeration and Reflection

Properties whose property names are private name values have all the same attributes as a property whose property name is a string value and the same defaults attribute are generally used. However, in most cases it is likely that properties defined using private names should not show up in for-in enumerations. For this reason, the semantics of the standard internal `[[Put]]` method are modified for cases where a property is created by `[[Put]]` using a Private Name value as the property name. In this case the `[[Enumerable]]` attribute of the newly created property is initially set to **false**. This change in made ES5 8.12.5, step 6.a.

For example:

```
private b;
var obj = {};
obj.a = 1;
obj.b = 2;
obj.c = 3;

var names = [];
for (var p in obj) names.push(obj[p]);
print(names.toString());    // "1,3" -- private name "b" was not enumerated
```

Properties with private names that are created using object literals also are created with their `[[Enumerable]]` attribute **false**. So `obj` could have been created to produce the same result by saying:

```
private b;
var obj = {
   a: 1,
   b: 2,
   c: 3
}
```

Because object literal properties are specified using `[[DefineOwnProperty]]` rather than `[[Put]]` all property descriptors used in ES5 11.1.5 must be updated to set `[[Enumerable]]` to **false** whenever a *PropertyName* is a private name value.

! Need to check all other uses of `[[DefineOwnProperty]]` and determine whether any of them should have special treatment of private name values.

Creating a private named property that is enumerable requires use of `Object.defineProperty` and the `#.` prefix. For example:

```
private b;
var obj = {};
```

```
obj.a = 1;
obj.b = 2;
Object.defineProperty(obj, #.b, {enumerable: true});
obj.c = 3;

var names = [];
for (var p in obj) names.push(obj[p]);
print(names.toString());    // "1,2,3" -- private name "b" is now enumerated
```

`Object.prototype.hasOwnProperty` (ES5 15.2.4.5), `Object.prototype.PropertyIsEnumerable` (ES5 15.2.4.7) and the `in` operator (ES5 11.8.7) are all extended to accept private name values in addition to string values as property names. Where they currently call ToString on property names they will instead call ToPropertyName. The `JSON.stringify` algorithm (ES5 15.12.3) will be modified such that it does not process enumerable properties that have private name values as their property names.

All the Object reflection functions defined in ES5 section 15.2.3 that accept property names as arguments or return property names are extended to accept or produce private name values in addition to string values as property names. A private name value may appear as a property name in the collection of property descriptors passed to `Object.create` and `Object.defineProperties`. If an object has private named properties then their private name values will appear in the arrays returned by `Object.getOwnPropertyNames` and `Object.keys` (if the corresponding properties are enumerable).

In ES5 `Object.defineProperties` and `Object.create` are specified to look only at the enumerable own properties of the object that is passed containing property descriptors. Usually this object is specified using an object literal. However, we have already specified above that private named properties in object literals are always created as non-enumerable properties. This would generally preclude the use of private name values in the object literals passed to these methods. It may be necessary to modify the specification of `Object.defineProperty` and `Object.create` to process also as property definitions any non-enumerable private named own properties that appear in such objects.

```
// Object.defineProperty probably needs to accept descriptors such as:
private a, b;
Object.defineProperties(obj, {
   a: {configurable: true}, // ES5 ignores non-enumerable properties
   b: {writable: true}      // that appear in such descriptors
});
```

An important use case for reflection using private name values is algorithms that need to perform meta-level processing of all properties of any object. For example, a “universal” object copy function might be coded as:

```
function copyObject(obj) {
   // This doesn't deal with other special [[Class]] objects:
   var copy = Object.isArray(obj) ? [] : Object.create(Object.getPrototypeOf(obj));
   var props = Object.getOwnPropertyNames(obj);
   var pname;
   for (var i = 0; i < props.length; i++) {
      pname = props[i];
      Object.defineProperty(copy, pname, Object.getOwnPropertyDescriptor(obj,pname));
   }
```

```
    return obj;
}
```

This function will duplicate all properties, including any that have private name values as their property names. It does not need to have any specific declarations for or knowledge of such private names.

Private Name Properties Support Only Weak Encapsulation

Private names are a simple and pragmatic way to support everyday encapsulation without requiring programmers to change their fundamental conceptualization of JavaScript objects. Private named properties do not provide and are not intended to provide strong impenetrable encapsulation of object state. For example, various reflection operations can be used to access an object's properties that have private names. Private names are instead intended as a simple extensions of the classic JavaScript object model that enables straight-forward encapsulation in non-hostile environments. The design preserves the ability to manipulate all properties of an objects at a meta level using reflection and the ability to perform "monkey patching" when it is necessary. Encapsulation via closure capture should continue to be used for situations where strong encapsulation that can not be penetrated by a hostile attacker is actually needed.

Sand-boxing environments that already need to restrict the use of certain reflection operations can use similar technique to limit access to private named properties. For example, a sandbox implementation might replace `Object.getOwnPropertyNames` with a version that filters out any private name values.

Interactions with other Harmony Proposals

There are potential feature interactions and opportunities for feature integration involving private names and several other Harmony proposals. As features are accepted into Harmony and their details are filled in these interactions need to be resolved.

Enhanced Object Literals

`private` might be supported as either a property modifier keyword that makes the property name a private name whose private identifier is scoped to the object literal:

```
var obj={
   private _x: 0;
   get x() {return this._x},
   set x(val) {this._x=val}
}
```

This might simplify the declarative creation of objects with instance private properties. However, there are internal scoping and hoisting issues that would need to be considered and resolved.

Another alternative is to use meta property syntax to declare object literal local private name declarations:

```
var obj={
   <prototype: myProto; private _x>
   _x: 0;
   get x() {return this._x},
   set x(val) {this._x=val}
}
```

Proxies

All uses of string valued property names in proxy handlers would need to be extended to accept/produce private name values in addition to string values.

As covered above, ECMAScript reflection capabilities provides a means to break the encapsulation of an object’s private named properties. Where this is a concern, it can be mitigated by replacing the reflection functions with versions that filter access to private name values. The Proxy proposal provides an additional means to break such encapsulation. If an attacker suspects that some object has private named properties it might sniff out those values by creating a proxy for the object whose handler consisted of traps that monitored all calls looking for private name argument values before delegating the operation to the original object.

One possible mitigation for this attach would be the same as for the other reflection functions. The Proxy object could be replaced with an alternative implementation that added an additional handler layer that would wrapper all private name values passed through its traps. The wrappers would be opaque encapsulations of each private name value and provide a method that could be used to test whether the encapsulated private name was === to an argument value. This would permit handlers to process known private name values but would prevent exposing arbitrary private name values to the handlers.

If there is sufficient concern about proxies exposing private name values in this manner, such wrapping of private names could be built into the primitive trap invocation mechanism.

Modules

It is reasonable to expect that modules will want to define and export private name values. For example, a module might want to add methods to a built-in prototype object using private names and then make those method names available to other modules. Within the present definition of the simple module system that might be done as follows:

```
<script type="harmony">
module ExtendedObject {
   import Builtins.Object;       // however access to Object is obtained.
   private clone;                // the private name for clone methods
   export const clone = #.clone; // export a constant with the private name value;

   Object.prototype.clone = function () { ... };
}
</script>
```

A consumer of this module might look like:

```
<script type="harmony">
import ExtendedObject.clone;
private clone = clone;
var anotherObj = someObj.clone();
</script>
```

The above formulation would work without any additional extensions to the simple module proposal. However, it would be even more convenient if the module system was extended to understand private declarations and the parallel property name environment. In that case this example might be written as:

```
<script type="harmony">
module ExtendedObject {
    import Builtins.Object;     // however access to Object is obtained.
    export private clone;       // export private name for clone methods

    Object.prototype.clone = function () { ... };
}
</script>
```

```
<script type="harmony">
import private ExtendedObject.clone;
var anotherObj = someObj.clone();
</script>
```

In these example the use of `import` and `export` prefixing `private` declarations forces use of the property name environment of the named module. For dynamic access to the exported property name environment of first-class module instances another mechnism would perhaps be needed:

```
<script type="harmony">
module private ExtendedObject_names = ExtendedObject;

private cloneEX = ExtendedObject_names.clone;  //get private name value bound to
''clone'' in reified module ''ExtendedObject''
var yetAnotherObj = someObj.cloneEX();
</script>
```

References

Private name values are akin to what is returned by `gensym` of Lisp and Scheme, and analogous to a capability in object-capability languages.

The inspiration for `private` is Racket's define-local-member-name and "selector namespaces" for Smalltalk and Ruby .

strawman/private_names.txt · Last modified: 2010/12/09 17:29 by allen

Show pagesource | Login | Index

Old revisions | Back to top

RSS XML FEED | LICENSED | $1 DONATE

PHP POWERED | W3C XHTML 1.0 | W3C CSS

DOKUWIKI

Table of Contents

- Default Proxy forwarding handler
  - Forwarding Handler constructor
    - Alternative names
    - Alternative implementation for default set trap
  - Feedback and History

Default Proxy forwarding handler

Goal: to standardize a default forwarding handler that delegates all meta-level operations applied to a proxy to a given target object, as exemplified here.

Rationale: this is a common handler, required as a starting point by most abstractions that wrap existing JS objects. The default forwarding handler is also required in the double lifting pattern.

Advantages of standardizing a default forwarding handler:

- Wrapper proxies don't need to define this handler over and over again,
- The code for the default handler doesn't need to be downloaded over and over again,
- The default handler evolves in sync with potential changes to the Proxy API,
- A built-in implementation is likely to be faster than a no-op forwarding handler defined in JS itself

Forwarding Handler constructor

The following is a revised API based on the standard Javascript constructor pattern.

```
Proxy.Handler = function(target) {
  this.target = target;
};

Proxy.Handler.prototype = {

  // == fundamental traps ==

  // Object.getOwnPropertyDescriptor(proxy, name) -> pd | undefined
  getOwnPropertyDescriptor: function(name) {
    var desc = Object.getOwnPropertyDescriptor(this.target);
    desc.configurable = true;
    return desc;
  },

  // Object.getPropertyDescriptor(proxy, name) -> pd | undefined
  getPropertyDescriptor: function(name) {
    var desc = Object.getPropertyDescriptor(this.target);
    desc.configurable = true;
    return desc;
  },
  // Object.getOwnPropertyNames(proxy) -> [ string ]
  getOwnPropertyNames: function() {
    return Object.getOwnPropertyNames(this.target);
  },

  // Object.getPropertyNames(proxy) -> [ string ]
  getPropertyNames: function() {
    return Object.getPropertyNames(this.target);
  },

  // Object.defineProperty(proxy, name, pd) -> undefined
  defineProperty: function(name, desc) {
    return Object.defineProperty(this.target, name, desc);
  },

  // delete proxy[name] -> boolean
  delete: function(name) { return delete this.target[name]; },

  // Object.{freeze|seal|preventExtensions}(proxy) -> proxy
  fix: function() {
    // As long as target is not frozen, the proxy won't allow itself to be fixed
    if (!Object.isFrozen(this.target))
      return undefined;
    var props = {};
    for (var name in this.target) {
      props[x] = Object.getOwnPropertyDescriptor(this.target, name);
    }
    return props;
  },

  // == derived traps ==

  // name in proxy -> boolean
  has: function(name) { return name in this.target; },

  // ({}).hasOwnProperty.call(proxy, name) -> boolean
  hasOwn: function(name) { return ({}).hasOwnProperty.call(this.target, name); },

  // proxy[name] -> any
  get: function(receiver, name) { return this.target[name]; },

  // proxy[name] = value
  set: function(receiver, name, value) {
   if (canPut(this.target, name)) { // canPut as defined in ES5 8.12.4 [[CanPut]]
     this.target[name] = value;
     return true;
   }
   return false; // causes proxy to throw in strict mode, ignore otherwise
  },

  // for (var name in proxy) { ... }
  enumerate: function() {
    var result = [];
    for (name in this.target) { result.push(name); };
    return result;
  },

  /*
  // if iterators would be supported:
  // for (var name in proxy) { ... }
  iterate: function() {
    var props = this.enumerate();
    var i = 0;
    return {
      next: function() {
        if (i === props.length) throw StopIteration;
        return props[i++];
      }
    };
  },*/

  // Object.keys(proxy) -> [ string ]
  keys: function() { return Object.keys(this.target); }
};
```

To create a default forwarding proxy to an object `obj`, one would write:

```
var h = new Proxy.Handler(obj);
var p = Proxy.create(h);
```

To modify one of the default traps, one can either override traps on a default handler or use prototype inheritance. For example:

```
// using assignment
var h = new Proxy.Handler(obj);
h.get = function(rcvr, name) { ... };
var p = Proxy.create(h);

// using inheritance
function MyHandler(target) {
  Proxy.Handler.call(this, target); // constructor chaining
}
MyHandler.prototype = Object.create(Proxy.Handler.prototype);
MyHandler.prototype.get = function(rcvr, name) { ... };

var h2 = new MyHandler(obj);
var p2 = Proxy.create(h2);
```

Pros of this API:

- Familiarity to Javascript developers.
- Handler inheritance is straightforward.
- All default traps are shared among all default handler instances.

Cons of this API:

- The constructor pattern is subject to the bug of forgetting `new`, in which case `Proxy.Handler(obj)` will set a `target` property on `Proxy`.
- It's awkward that handlers are created using constructor functions (requiring `new`) whereas proxies are created using a factory method (not requiring `new`). This makes the API feel a little inconsistent.

— *Tom Van Cutsem 2010/12/14 3:10*

Alternative names

We can debate about alternative names for `Handler` and `target`.

If the Proxy API would be contained in a Harmony module, it may make sense to introduce `Handler` as an exported variable, next to `Proxy`, instead of making it a property on `Proxy`.

Alternative implementation for default set trap

As currently defined, the default `set` trap's behavior is counter-intuitive in the case of a "chain" of proxies (a proxy forwarding to another proxy):

```
  set: function(receiver, name, value) {
   if (canPut(this.target, name)) { // canPut as defined in ES5 8.12.4 [[CanPut]]
     this.target[name] = value;
     return true;
   }
   return false; // causes proxy to throw in strict mode, ignore otherwise
  },
```

If `this.target` is a proxy, the `canPut` auxiliary function will trigger that proxy's `getOwnPropertyDescriptor` and `getPropertyDescriptor` traps to determine whether the property can be set. Only then is the assignment performed on `this.target` and is that proxy's `set` trap invoked.

Part of the awkwardness lies in the fact that the "inner" `set` returns its own boolean to indicate success, but that boolean isn't accessible to the "outer" `set`. Instead each proxy in the chain tests the boolean and either throws or ignores it. MarkM suggests the following refactoring of the internal spec methods which would make this chaining of `set` calls more intuitive:

Since the system itself will provide the default traps, the default `set` trap could call a new internal [[Set]](P,V) method which returns a boolean, such that [[Put]] would be redefined as:

```
8.12.5 [[Put]](P,V,Throw)

  If the result of calling [[Set]](P,V) is true, return.
  Else if Throw is true, throw a TypeError exception.
  else return.
```

The [[Set]] method on regular objects would be defined as is the current [[Put]] but returning a boolean rather than conditionally throwing. The [[Set]] method on proxies would call the `set` trap. The default `set` trap would call [[Set]] on `this.target`. So this default `set` trap is the primitive by which the ability to call [[Set]] is exposed.

The one problem with this plan is [[Set]] on an object that inherits from a proxy. This plan would still go through the

[[CanPut]] logic on the derived object which would still trigger the [[GetProperty]] and [[GetOwnProperty]] traps on the proxy. So there’s not much difference in the inherited case. But the direct chaining case is more direct and intuitive.

— *Tom Van Cutsem 2011/01/12 3:10*

Feedback and History

TC39 November 2010 meeting: agreement that a default forwarding handler should become part of the spec.

A first iteration of this API required handlers to be created using a factory method:

```
var handler = Proxy.handlerFor(target);
```

Drawback of this API: one cannot inherit from a shared handler prototype. Waldemar: why not define an API based on prototypes and constructor functions? The revised API was formulated in response to this.

— *Tom Van Cutsem 2010/12/15 2:53*

[[strawman: parameters_property_of_functions]]

Proposal

This is a companion to name property of functions.

Function instances will have a non-writeable, non-configurable `parameters` property whose value is a frozen array of strings, the strings being the names of the function's parameters.

This further improves reflection, which can benefit the reliability of some callback and registration patterns.

— *Douglas Crockford 2010/12/15 19:28*

Could you expand on the use case of this. Why does any code need to dynamically know the formal parameter names of functions. Do we need to consider the impact of obfuscators or minimizers. Will people write code that depends upon the parameter names in the original code?

In general, I would prefer that we move to mirror-based approach to function reflection rather than adding additional properties to Function.prototype:

```
var fm= new FunctionInspector(f);
if (fm) {
   ...fm.parameterNames...
   ...fm.statements[n]...
   ...fm.breakAt(n)...
   ...fm.whatever...
}
```

Of course, the mirror object could be provided in an implementation provided library or modules.

— *Allen Wirfs-Brock 2011/01/14 19:15*

There are callback patterns that could be made more robust by allowing for the checking of function signatures. I am not proposing adding anything to `Function.prototype`. These can be added to the function instances.

I don't want to give access to the guts of the functions. I just want to check their interfaces. So I don't need any of the other reflectivity.

— *Douglas Crockford 2011/01/19 17:23*

Proper Tail Calls

Table of Contents

- Proper Tail Calls
  - Lost Opportunity
- Tail position
  - Attribute grammar
  - Expressions
  - Statements
  - Clauses
  - Functions
  - Declarations
- References

We need to define when a call is in *possible tail position* for safety purposes, and *proper tail position* for liveness purposes. An implementation MAY implement a call in possible tail position as a proper tail call (PTC) and SHOULD implement a call in proper tail position as a PTC. (Loosely speaking, implementing a call as a PTC is also known as *tail call optimization*.) The old ES4 proposal for proper tail calls should be adapted to provide these definitions for ES-Harmony.

Since ES5/strict and ES-Harmony

- poisons `<function>.caller` and `<function>.arguments`,
- prohibits `<non-strict-function>.caller` from revealing a strict function,
- poisons `arguments.callee` and `arguments.caller`, and
- does not join `arguments` with parameters

all calls in possible tail position MAY safely be implemented as PTC by our gc semantics safety rule.

When calculating the asymptotic space complexity of an algorithm, we are concerned with GC liveness, in which case we may assume that all calls in proper tail position are implemented as PTC. To support such assumptions, implementers SHOULD implement such calls as PTC. We model the call stack as a *gc root*. All active stack frames are strongly reachable from the call stack. All active local variables within a stack frame are strongly reachable from a stack frame. Once stack frame X initiates a PTC, X is no longer active, and so X's stack frame is no longer reachable from the call stack. X, and any objects which were transitively reachable only from X's local variables are thus no longer transitively reachable from roots, and so SHOULD eventually be collected, if needed to avoid failure from memory exhaustion.

Just a note after some offline conversation with Mark: he wants a distinction between the classic notion of tail position and what he calls "possible tail position," but I am skeptical that the latter concept makes sense. I also don't think the distinction matters.

— *Dave Herman 2010/05/18 23:56*

Lost Opportunity

ES3 had reserved the identifier `goto`. ES4 had proposed that a call in tail-call position be considered a proper tail call only if preceded by `goto`, in which case the programmer could expect the PTC implementation implied by our liveness constraint above. If the programmer used `goto` on a call that cannot be implemented as PTC, this would be a static error. As a result, the programmer's PTC assumptions when reasoning about the asymptotic space complexity of their programs would be maintainable. A subtle change to a function call that preserved its normal semantics but lost its assumed PTC nature would also cause a static error.

Unfortunately, ES5 unreserved `goto`. Once unreserved, it would be too costly to reclaim it. None of the remaining reserved words are a plausible substitute.

Tail position

Part of specifying proper tail calls requires a specification of what are the *proper tail positions* of the language. In the ES4 proposal, Lars and I made an initial attempt, but it was pretty buggy. Here's an updated spec. It will still need more scrutiny, but this is a start.

— *Dave Herman 2010/05/18 00:03*

Attribute grammar

The spec is written as an *attribute grammar*. Attributes are properties of nodes in the abstract syntax tree. In general, attributes are either *synthesized* (computed bottom-up) or *inherited* (computed top-down). In this spec, there are only inherited attributes. (This is particularly pleasant for implementations, since it can easily be computed in any pass of a compiler or interpreter, with no need for an additional pass.)

Attribute	Directionality	Node type	Meaning
`tail`	inherited	expression	if **true**, the node is in tail position
`wrapped`	inherited	all	if **true**, no child expressions in the same function are in tail position

Expressions

```
    E -> E1 , E2                              E1.tail = false
                                              E2.tail = E.tail
                                              E1.wrapped = E2.wrapped = E.wrapped
    E -> E1 ? E2 : E3                         E2.tail = E3.tail = E.tail
                                              E1.wrapped = E2.wrapped = E3.wrapped

    /* from the "let expressions" proposal */
    E -> let (V1 = E1, ..., Vn = En) {        E1.tail = ... = En.tail = false
             S1 ... Sm                        E{n+1}.tail = E.tail
        }                                     S1.wrapped = ... = Sn.wrapped = S1.wrapped
 = ... = Sm.wrapped = E.wrapped
    E -> let (V1 = E1, ..., Vn = En) {        E1.tail = ... = En.tail = false
             S1 ... Sm => E{n+1}              E{n+1}.tail = E.tail
        }                                     S1.wrapped = ... = Sn.wrapped = S1.wrapped
 = ... = Sm.wrapped = E{n+1}.wrapped = E.wrapped
```

Statements

```
    S -> { S1 ... Sn }                        S1.wrapped = ... = Sn.wrapped = S.wrapped
    S -> E;                                   E.tail = false
    S -> do S1 while (E)                      S1.wrapped = S.wrapped
                                              E.tail = false
    S -> while (E) S1                         S1.wrapped = S.wrapped
                                              E.tail = false
    S -> for (E1; E2; E3) S1                  E1.tail = E2.tail = E3.tail = false
                                              S1.wrapped = S.wrapped
    S -> for (E1 in E2) S1                    E1.tail = E2.tail = false
                                              S1.wrapped = S.wrapped
    S -> if (E) S1 else S2                    S1.wrapped = S2.wrapped = S.wrapped
                                              E.tail = false
    S -> L: S1                                S1.wrapped = S.wrapped
    S -> with (E) S1                          S1.wrapped = S.wrapped
                                              E.tail = false


    /* tail unless wrapped by try */
    S -> return E;                            E.tail = !s.wrapped
    /* no tail positions inside */
    S -> try S1 K                           S1.wrapped = true
                                            K1.wrapped = ... = Kn.wrapped = S.wrapped
    S -> try S1 K finally S2                S1.wrapped = true
                                            K1.wrapped = ... = Kn.wrapped = true
                                            S2.wrapped = false

    S -> throw E;                           E.tail = false
    S -> switch (E) C1 ... Cn               E.tail = false
                                            C1.wrapped = ... = Cn.wrapped = S.wrapped
```

Clauses

```
    K -> catch (V) S                        S.wrapped = K.wrapped
    C -> case E: S1 ... Sn                  E.tail = false
                                            S1.wrapped = ... = Sn.wrapped = C.wrapped
    C -> default: S1 ... Sn                 E.tail = false
                                            S1.wrapped = ... = Sn.wrapped = C.wrapped
```

Functions

```
    F -> function f(x1,...,xn) S            S.wrapped = false
```

Declarations

```
    D -> var x1 = E1, ..., xn = En;         E1.tail = ... = En.tail = false
    D -> let x1 = E1, ..., xn = En;         E1.tail = ... = En.tail = false
    D -> const x1 = E1, ..., xn = En;       E1.tail = ... = En.tail = false
```

References

ES4 proposal for proper tail calls and discussion of that page.

Trac ticket 323, on implicit vs. explicit `goto` syntax, tail position assertion, etc.

Proper tail calls in Scheme.

Dave Herman's blog on Clinger's proper tail calls.

Brendan discovers that ES5/strict enables TCO.

Guy Steele says OO requires tail call optimization

Gilad Bracha on "Steele's brilliant post"

[[strawman: regexp_x_flag]]

ES Wiki

Show pagesource
Recent changes
Old revisions
Search

Trace: » proxy_defaulthandler » parameters_property_of_functions » proper_tail_calls » proper_tail_calls » regexp_x_flag

Proposal

Add `x` to the set of flags `gim`. The `x` flag causes line endings and unescaped spaces in the regular expression to be ignored. By allowing whitespace within regular expression literals, the literals may be significantly easier to read and modify.

This proposal does not allow for comments within regular expression literals because of the confusion of `/` closing regexp literals and also being a component of `//` and `/* */`.

— *Douglas Crockford 2010/12/15 19:43*

This was proposed for ES4. The wiki saves us from being condemned to repeat history: proposal, comments in x-flagged regexps discussion, issues with the x flag.

In particular, the comment syntax Lars proposed was `# till end of line`. This should be considered, but let's not ignore Waldemar's objections. Best I can find right now from Waldemar is this es-discuss post. In that message, Waldemar was poking one hole (there are others) in a forward-compatible syntax idea, but I recall him objecting to the x flag on its own.

I should add that CoffeeScript uses triple-slash delimiters for multiline regexp literals: CoffeeScript docs under "Extended Regular Expressions". Note that CoffeeScript also uses `# till end of line` internal-to-extended-regexp comment syntax.

Of course JS isn't CoffeeScript: `///` is already lexed as comment to end of line with an extra `/` in JS, but something like this approach has the advantage that the lexer knows to handle a multiline regexp up front, instead of only after getting a newline (after which it must assume `/x` at the end, and require `x` to be there).

— *Brendan Eich 2010/12/16 08:41*

The important thing is to get the whitespace. I can live without the comments.

Triple slash looks to be hazardous.

— *Douglas Crockford 2011/01/03 17:50*

strawman/regexp_x_flag.txt · Last modified: 2011/01/03 17:55 by crock

Show pagesource
Login
Index

Old revisions

Back to top

[[strawman:regexp_y_flag]]

ES Wiki

Table of Contents

- Proposal
  - Rationale
  - Compatibility
  - Reflection
- Precedent

Proposal

The regular expression literal flag "y" shall be permitted.

The flag states that the regular expression peforms sticky matching in the target string by attempting to match starting at `lastIndex`. If matching at that location fails then null is returned, i.e., no forward "anchoring" search is performed. If matching succeeds, then `lastIndex` is updated as for the flag "g".

Rationale

This flag makes it easier to write simple and efficient lexical analyzers for embedded languages using ECMAScript regular expressions. The current language has quadratic complexity because each match may potentially search to the end of the input for a match. (That can be worked around in a couple of ways, e.g., by slicing successive tail substrings at quadratic space complexity, but they are cumbersome.)

Compatibility

The flag is illegal in 3rd Edition, and the matching behavior is a subset of existing behavior, so this flag should not cause trouble for any implementation.

Reflection

The state of the flag would show up as a property `sticky` on RegExp objects, which value `true` if "y" was supplied, `false` otherwise.

Precedent

The "y" flag was implemented in JavaScript 1.8 in Firefox 3

JSON.path

JSONpath is an XPath-like language for extracting information from data structures. See http://goessner.net/articles/JsonPath/ and http://code.google.com/p/jsonpath/.

It is proposed that a `path` property be added to the `JSON` object, whose value is a function that takes an object or array, and a JSONpath string, and returns a value.

— *Douglas Crockford 2010/12/15 19:53*

[[strawman:object_isobject]]

Object.isObject

It is proposed that an `isObject` property be added to the `Object` object, whose value is a function that could be implemented as `Code Text`

```
Object.isObject = function isObject(value) {
     return typeof value === 'object' && value !== null;
}
```

— *Douglas Crockford 2010/12/15 20:04*

This is presumably a convenience method to get around the fact that `typeof null` is "`object`". I want to explore the actual use cases to see if adding this method is actually worth the effort.

It seems to me that there are two use cases for this sort of test. The first is a "switch" over all the built-in types:

```
var t = typeof arg;
if (t==="number") { /*do something for numbers*/ }
else if (t==="string") { /* do something for strings */ }
...
else if (t==="object" && arg===null) { /* do something for null */ }
else if (t==="object") { /* do something for objects */ }
else { /* do something for any types that weren't explicitly tested for */ }
```

To code this the programmer had to remember that they needed to explicitly test for the null case. Adding `isObject` does not seem to significantly change what has to be done for this use case. (Of course the guard of the null case could be reduced to `arg===null` in which case `isObject` probably adds no improvement of readability.) The programmer might replace the explicit `null` test with a call to `isObject`:

```
var t = typeof arg;
if (t==="number") { /*do something for numbers*/ }
else if (t==="string") { /* do something for strings */ }
...
else if (t==="object" && !Object.isObject(arg)) { /* do something for null */ }
else if (t==="object") { /* do something for objects */ }
else { /* do something for any types that weren't explicitly tested for */ }
```

or perhaps restructure the code:

```
if (Object.isObject(arg)) { /* do something for objects */ }
```

```
else switch (typeof arg) {
   case "number":  /*do something for numbers*/  break;
   case "string":  /*do something for strings*/ break;
   case "object":  /*do something for null*/    break;
   default:  /* do something for any types that weren't explicitly tested for
*/
}
```

However, in both formulations the programmer still has to remember the fact that `typeof null` is "object" and explicitly deal with it in the code. The only thing that has changed is the specific test that is used to discriminate the `null` case.

The other use case is a simple determination of whether or not a value is an object:

```
if (obj && typeof obj ==="object") { /* do something for objects */ }
else { /* do something for other types of values */ }
```

Using `isObject` this simplifies to:

```
if (Object.isObject(obj)) { /* do something for objects */ }
else { /* do something for other types of values */ }
```

I would agree that both the above reformulation and the switch statement based formulation of the first use case, are clearer to read. However, it isn't clear that `isObject` actually reduces the complexity burden on the programmer. Because the `typeof` operator exists the programmer still needs to understand it and know about its pitfalls and when it is and isn't appropriate to use it.

What does stand out, is that situations that need to dispatch on primitive types would be simplified by a function that yields a unique value for all of the built-in types. From that perspective a function such as `Object.getTypeOf` might be even better:

```
switch (Object.getTypeOf(arg)) {
   case "number":  /*do something for numbers*/ break;
   case "string":  /*do something for strings*/ break;
   case "object":  /*do something for objects*/ break;
   case "null":    /*do something for null*/    break;
   default:  /* do something for any types that weren't explicitly tested for
*/
}
```

however it doesn't improve the readability of the second use case as much as `isObject`:

```
if (Object.getTypeOf(obj)==="object") { /* do something for objects */ }
else { /* do something for other types of values */ }
```

I'm not sure that either of these functions add enough value to justify adding them to the core language. They could be provided in a library. However, if we were only going to add one, I would probably prefer the one that provides a switchable value.

— *Allen Wirfs-Brock 2011/01/14 18:43*

Why don't we just fix `typeof`?

Sure, doing so adds yet another runtime meaning shift that can't be caught by early errors. But it could be worth it, compared to bloating `Object` with new methods.

The best case of adding `Object` methods means we caused everyone migrating code into Harmony, or writing fresh code, to change how they would have worked if only `typeof` were sane in this regard. In such a best case, we would rather have just fixed `typeof`.

The worst case is that hardly anyone bothers to use the new `Object` method or methods, but we have bloated the spec and the language slightly.

Of course, with a `typeof null === "null"` change, there is a bad case where programmers migrate or write fresh code but forget about the change, and only find out via testing.

Realistically, auditing code moving into Harmony is required due to the lexical-global-scope change. Auditing needs tools. Tools can find and check (static analysis) or at least warn about `typeof` usage and any dependent potential null dereferences. We would be happy to get DoctorJS doing this, and I hope Doug can make JSLint do something too.

In this light, why not fix `typeof`?

— *Brendan Eich 2011/01/14 20:18*

I would very much prefer to fix `typeof`. We talked about this before, and decided that there is code out there that depends on `typeof` 's bad behavior. Are we now saying that the migration tax is acceptable?

— *Douglas Crockford 2011/01/19 17:17*

Migration tax was an issue for ES5 (strict or not). In the context of Harmony and focused on `typeof`, it is not clearly a showstopper.

There is a non-migration tax too, with developers sticking to the old broken `typeof` and making mistakes or (at some grinding, minor cost that adds up over time) writing null tests carefully. This taxes the committee, too, with proposals to extend built-on objects with compensating methods such as the one proposed here: `Object.isObject`.

For me the killer argument is Allen's use-case analysis: in the best case, we've just imposed a different tax of about the same size, with cleaner partial semantics but the underlying bug in `typeof` unfixed. If people are to invest effort in migrating `typeof` code to use a new form, then fixing `typeof` seems strictly better.

The downside of breaking code migrated without enough auditing and testing remains, but I'm proposing in http://brendaneich.com/2011/01/harmony-of-my-dreams/ that we build a Harmonizer tool that can do enough static analysis to find missing null tests (not just around `typeof`, but at least around `typeof`).

Such a Harmonizer tool may be great, but I wouldn't bet too big on it. We shouldn't change every little corner of the language that seems messy (fall-through in `switch`, e.g. – from C originally, and in Java). For `typeof`, though, I'm willing to make a focused bet.

— *Brendan Eich 2011/01/19 20:56*

strawman/object_isobject.txt · Last modified: 2011/01/19 21:03 by brendan

Array.create

It is proposed that a `create` property be added to the `Array` constructor, being a function that is similar to `Object.create`, creating a new array that inherits from the first argument (which will usually be an object. If there is an optional second argument (which will usually be an array or array-like object) then copy the elements numbered between 0 and `length - 1` to the new array. The result is seen as an array by `Array.isArray`, and the result has a magic `length` property, but it does not necessarily inherit from `Array.prototype`.

```
  myArray = Array.create(augmented_array_prototype, ['a', 'b', 'c']);
```

`Array.create` is simpler, more elemental, and shares more continuity with ES5 than array_subtypes.

— *Douglas Crockford 2011/01/03 17:48*

Array Subtypes

Motivation

A common request from developers is for a mechanism that allows the creation of a "Array subtype". An Array subtype is in essence an Array instance with a custom prototype chain.

Proposal

We propose the addition of a 'createConstructor' function to the Array constructor object that returns a new function object with the same call semantics as the built in Array constructor, but returning an array instance with a custom prototype.

A pseudo implementation of this function would be:

```
Array.createConstructor = function() {
    var constructor = function() {
        var result = <<Initial Array Constructor>>.apply(null, arguments);
        result.__proto__ = constructor.prototype; // recognising that this isn't
actually possible in raw ES
        return result;
    }
    constructor.prototype.constructor = constructor;
    return constructor;
}
```

Discussion

Beware the case where `arguments.length == 1 && arguments[0]` is a valid `length` value. Or did you intend to preserve that wart in `Array`? I hope not! — *Brendan Eich 2010/11/17 22:14*

I was preserving existing behaviour as it would allow code to use the Array constructor or the subtype interchangeably, but I could go either way without any real problem — *Oliver Hunt 2010/12/15 21:08*

The `result.constructor` property is instance-invariant and should be on `constructor.prototype`, as with all the built-in and user-defined constructor functions.

— *Brendan Eich 2011/01/14 01:40*

I believe I have now fixed the example code — *Oliver Hunt 2011/01/20 01:09*

Number.isFinite

An `isFinite` function is added to the `Number` constructor. It could be implemented as this:

```
Number.isFinite = function isFinite(value) {
    return typeof value === 'number' && isFinite(value);
};
```

This is better than the global `isFinite` because it is not confused by type coercion.

— *Douglas Crockford 2010/12/15 20:22*

Number.isNaN

It is proposed that an `isNaN` property be added to the `Number` constructor. It could be implemented like this:

```
Number.isNaN = function isNaN(value) {
    return typeof value === 'number' && isNaN(value);
};
```

This is better than the global `isNaN` function because it is not confused by type coercion.

— Douglas Crockford *2010/12/15 20:27*

String.prototype.dup

It is proposed that a `dup` method be added to `String` takes an integer argument and returns a string that is repeated according to the argument. So `"*".dup(3)` produces `"***"`.

Currently, the way this is commonly done is with `new Array(4).join('*')`. There should be a better alternative.

An exception is thrown if the argument is not a small positive integer.

Full bikeshed paint color debate required: `dup` is short and Unixy (it's the system call that duplicates an open file table reference into a per-process file descriptor table entry), while our other `String.prototype` methods have Java-esque names such as `toString`. The Pythonic latecomers such as `slice` are full words too. The Perl-based `substr` has been banished to Annex B.

In this light, I suggest `repeat`.

The exception thrown needs to be pinned down. `RangeError`?

The small positive integer needs to be specified for interop's sake, at the limit.

— *Brendan Eich 2010/12/22 22:58*

I think `repeat` is a swell name. `RangeError` fits.

255 might be a reasonable arbitrary limit. It makes no sense, but it is familiar.

— *Douglas Crockford 2011/01/03 17:38*

I don't see why we would impose a limit here when we don't explicitly impose such limits on other functions. I certainly can imagine situations where a repeat >255 could be desirable.

— *Allen Wirfs-Brock 2011/01/10 01:32*

That's fine with me. Do we do a toInt32 on the argument then? If it is a weird value, should it throw or return empty string?

— *Douglas Crockford 2011/01/10 23:08*

Generally, the string functions don't impose the toInt32 restriction. They just apply ToInteger to numeric arguments.

— *Allen Wirfs-Brock 2011/01/14 17:07*

```
String.prototype.repeat = function (count) {
    var result = '', string = String(this);
    while (count > 0) {
        result += string;
        count -= 1;
    }
    return result;
}
```

— *Douglas Crockford 2011/01/19 17:27*

```

```

String.prototype.format

It is proposed that a `format` method be added to `String.prototype` that replaces variables in strings, with an option for contextual encoding.

```
string.format(values, encoder, open, close, separator)
```

`string` is a string the contains variable names which are wrapped in `{ }`. Optionally, the name can include a `:` followed by an encoding.

```
"Page not found: {url:html_text}."
```

`values` is either an object or a function. If it is an object, then the {name} is used to retrieve a value from the object. If the result of the retrieval is not `undefined`, then the variable is replaced with the value. If `values` is a function, then the {name} is passed to the function as an argument. If the return value is not `undefined`, then the variable is replaced with the value.

`encoder` is optional. It is either an object or a function. If it is an object, then the text following the `:` is used to find a method in the `encoder` object. The method is passed the translated value that obtain from `values`. The result is used as the replacement. If `encoder` is a function, then it is called with the translated value and thetext following the `:`. The result is used as the replacement.

`open` is optional. It is a string that identifies the start of a variable. The default is "`{`".

`close` is optional. It is a string that identifies the end of a variable. The default is '"`}`"'.

`separator` is optional. It is a string that separates a name from an encoding. The default is "`:`".

— *Douglas Crockford 2010/12/15 20:51*

Could you give a usage example? I'm not 100% convinced that I like the behaviour of listing multiple parameters inside a single set of braces. I also think that the format instruction should allow the open/close tokens to be escaped. — *Oliver Hunt 2010/12/15 21:11*

Is this based on prior art from any popular JS libraries? We should survey what's out there.

Also, Mike Samuel’s quasis proposal. ES4-era discussion. Python’s solution. — *Brendan Eich 2010/12/15 23:31*