[[strawman:names]]

ES Wiki

Table of Contents

Overview

In existing ECMAScript, it is not possible to create hidden properties. It is possible to create non-enumerable properties, but they can still be discovered by guessing their 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, propertyless, object called a `Name`
2. generalizing the `propertyName` concept to include either a string (as in ES5) or a `Name` as above
3. a `private` keyword for automatic use of `Name` objects instead of strings in certain places 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 methods, or with other additions by any other code.

Name objects

The `Name` constructor allows for the creation of fresh, opaque names:

```
var name = new Name;
print(name); // [object Name]
```

`Name` objects never have properties, aside from those inherited from `Object`. Attempting to add a property or mutate an existing property of a `Name` object results in a type error.

Name objects as property names

Name objects can be used as property names:

```
var name = new Name;
print(name); // [object Name]
```

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

obj[name] = "secret";
print(obj[name]); // secret

for (var key in obj) {
    print("obj." + key + " = " + obj[key]); // obj.a = 1, obj.b = 2, obj.c = 3
}
```

Properties with `Name` objects as keys can never be enumerable. Attempting to create an enumerable property with a `Name` object as its key produces a TypeError. This preserves the invariant that enumerated keys are always strings.

`Object.getOwnPropertyNames`, `Object.keys`, and `Object.getOwnPropertyDescriptor` skip all properties with `Name` objects as keys.

Binding private names

The keyword `private` allows for block-scoping of an identifier as a special name. The declaration:

```
private x;
```

hoists the creation of a new `Name` to the top of the block (evaluated after functions but before `var`s and `let`s). All uses of `x` as a `propertyName` in the block are converted to computed uses of the name object, instead of static uses of "`x`". In particular, uses of `x` after a `.` and as a property name in an object literal are so converted. `x` is also bound as a plain variable to the `Name` value.

This can be used both for "instance-private" properties:

```
function Thing() {
    private key;
    this.key = "secret";
    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)); // false
```

as well as "class-private" properties:

```
private key;
function Thing() {
    this.key = "secret";
    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
```

Creating private object properties

The `private` keyword can also be used as a modifier on properties in object literals. For example:

```
let obj = { private foo: 42,
            getFoo: function() this.foo };
```

is equivalent to:

```
let obj = let () {
              private foo;
          => { foo: 42,
               getFoo: function() { return this.foo } }
         };
```

Note that this relies on the conversion for object literals described above.

Semantics

Property Selection:

A `private x` declaration is equivalent to a declaration of the form `let x = new Name`, with the following differences:

- The declaration is hoisted before all other `var` and `let` bindings.

- The declarative environment record associated with the declaration records that `IsBindingPrivate("x")` (discussed below) is `true`.

The key semantic question is the resolution of expressions of the form `e.x`. The proposed semantics are as follows:

- Each declarative environment record maintains a marker with each entry declared with `private`. This is accessed with a new internal function `IsBindingPrivate(String)`.

- Iterate up the chain of environment records until one is found where `HasBinding("x")` produces `true`. If that same environment record produces `true` for `IsBindingPrivate("x")`, then the expression is evaluated as if it was `e[x]`. Otherwise it is evaluated as if it was `e["x"]`.

- If no binding for `x` is found, `e.x` is treated normally.

Of course, the semantics need not be implemented via runtime lookup. However, this semantics ensures that `e.x` references a `Name` object as the `propertyName` only if a declaration of `private x` is lexically in scope. This ensures both that programmers can reason about code lexically, and that implementations can statically compile all property references to either `String` or `Name` lookup.

Operation *ToName*(x):

This converts `x` to a string, unless `x` is a `Name` object, in which case `x` is produced. Changing the text of ECMA 262 will require more editing, since the spec relies on the string nature of property names implicitly in a number of places.

GC Semantics

Since names are not enumerable and unforgeable and are always leaves in the object graph, if a name `N` cannot be reached, the property `N` of any object may be safely deleted by the garbage collector. This is not required, and is not observable by the program (except by measuring memory consumption externally). If gc semantics is adopted, the spec could state that a value is only reachable through a property whose name is a `Name` if the name is independently reachable.

Potential Extensions

Currently, `private x` always brings a fresh `Name` into scope. It might be valuable to support

```
private x = E;
```

where `E` is any expression producing a `Name` object. If the value is not a `Name` object, a `TypeError` is raised. This allows both renaming via lexical scope as well as converting `Name` values provided as function arguments, for example, to the `f.x` syntax.

Also, for debugging purposes, it might be useful to give the `Name` constructor an optional string argument to be used in printing. This would mean that `Name`s were no longer leaves in the object graph, however.

References

The `Name` object is akin to `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.

Similar ideas have been proposed for Smalltalk and Ruby under the name "selector namespaces".