Implementing Private Fields for JavaScript

By Matthew Gaudet

Posted on June 8, 2021 in Featured Article, Firefox, and JavaScript

This post is cross-posted from Matthew Gaudet's blog

When implementing a language feature for JavaScript, an implementer must make decisions about how the language in the specification maps to the implementation. Sometimes this is fairly simple, where the specification and implementation can share much of the same terminology and algorithms. Other times, pressures in the implementation make it more challenging, requiring or pressuring the implementation strategy diverge to diverge from the language specification.

Private fields is an example of where the specification language and implementation reality diverge, at least in SpiderMonkey– the JavaScript engine which powers Firefox. To understand more, I'll explain what private fields are, a couple of models for thinking about them, and explain why our implementation diverges from the specification language.

Private Fields

Private fields are a language feature being added to the JavaScript language through the TC39 proposal process, as part of the class fields proposal, which is at Stage 4 in the TC39 process. We will ship private fields and private methods in Firefox 90.

The private fields proposal adds a strict notion of 'private state' to the language. In the following example, `#x` may only be accessed by instances of class `A`:

```
class A {
  #x = 10;
}
```

This means that outside of the class, it is impossible to access that field. Unlike public fields for example, as the following example shows:

```
class A {
  #x = 10; // Private field
  y = 12; // Public Field
```

```
}

var a = new A();
a.y; // Accessing public field y: OK
a.#x; // Syntax error: reference to undeclared private field
```

Even various other tools that JavaScript gives you for interrogating objects are prevented from accessing private fields (e.g. `Object.getOwnProperty{Symbols,Names}` don't list private fields; there's no way to use `Reflect.get` to access them).

A Feature Three Ways

When talking about a feature in JavaScript, there are often three different aspects in play: the mental model, the specification, and the implementation.

The mental model provides the high-level thinking that we expect programmers to use mostly. The specification in turn provides the detail of the semantics required by the feature. The implementation can look wildly different from the specification text, so long as the specification semantics are maintained.

These three aspects shouldn't produce different results for people reasoning through things (though, sometimes a 'mental model' is shorthand, and doesn't accurately capture semantics in edge case scenarios).

We can look at private fields using these three aspects:

Mental Model

The most basic mental model one can have for private fields is what it says on the tin: fields, but private. Now, JS fields become properties on objects, so the mental model is perhaps 'properties that can't be accessed from outside the class'.

However, when we encounter proxies, this mental model breaks down a bit; trying to specify the semantics for 'hidden properties' and proxies is challenging (what happens when a Proxy is trying to provide access control to properties, if you aren't supposed to be able see private fields with Proxies? Can subclasses access private fields? Do private fields participate in prototype inheritance?) . In order to preserve the desired privacy properties an alternative mental model became the way the committee thinks about private fields.

This alternative model is called the 'WeakMap' model. In this mental model you imagine that each class has a hidden weak map associated with each private field, such that you could hypothetically 'desugar'

```
class A {
  #x = 15;
  g() {
    return this.#x;
  }
}
```

into something like

```
class A_desugared {
  static InaccessibleWeakMap_x = new WeakMap();
```

```
  constructor() {
    A_desugared.InaccessibleWeakMap_x.set(this, 15);
  }

  g() {
    return A_desugared.InaccessibleWeakMap_x.get(this);
  }
}
```

The `WeakMap` model is, surprisingly, not how the feature is written in the specification, but is an important part of the design intention is behind them. I will cover a bit later how this mental model shows up in places later.

Specification

The actual specification changes are provided by the class fields proposal, specifically the changes to the specification text. I won't cover every piece of this specification text, but I'll call out specific aspects to help elucidate the differences between specification text and implementation.

First, the specification adds the notion of `[[PrivateName]]`, which is a globally unique field identifier. This global uniqueness is to ensure that two classes cannot access each other's fields merely by having the same name.

```
function createClass() {
  return class {
    #x = 1;
    static getX(o) {
      return o.#x;
    }
  };
}

let [A, B] = [0, 1].map(createClass);
let a = new A();
let b = new B();

A.getX(a); // Allowed: Same class
A.getX(b); // Type Error, because different class.
```

The specification also adds a new 'internal slot', which is a specification level piece of internal state associated with an object in the spec, called `[[PrivateFieldValues]]` to all objects. `[[PrivateFieldValues]]` is a list of records of the form:

```
{
  [[PrivateName]]: Private Name,
  [[PrivateFieldValue]]: ECMAScript value
}
```

To manipulate this list, the specification adds four new algorithms:

1. `PrivateFieldFind`
2. `PrivateFieldAdd`
3. `PrivateFieldGet`

4. `PrivateFieldSet`

These algorithms largely work as you would expect: `PrivateFieldAdd` appends an entry to the list (though, in the interest of trying to provide errors eagerly, if a matching Private Name already exists in the list, it will throw a `TypeError`. I'll show how that can happen later). `PrivateFieldGet` retrieves a value stored in the list, keyed by a given Private name, etc.

The Constructor Override Trick

When I first started to read the specification, I was surprised to see that `PrivateFieldAdd` could throw. Given that it was only called from a constructor on the object being constructed, I had fully expected that the object would be freshly created, and therefore you'd not need to worry about a field already being there.

This turns out to be possible, a side effect of some of the specification's handling of constructor return values. To be more concrete, the following is an example provided to me by André Bargull, which shows this in action.

```
class Base {
  constructor(o) {
    return o; // Note: We are returning the argument!
  }
}

class Stamper extends Base {
  #x = "stamped";
  static getX(o) {
    return o.#x;
  }
}
```

`Stamper` is a class which can 'stamp' its private field onto any object:

```
let obj = {};
new Stamper(obj); // obj now has private field #x
Stamper.getX(obj); // => "stamped"
```

This means that when we add private fields to an object we cannot assume it doesn't have them already. This is where the pre-existence check in `PrivateFieldAdd` comes into play:

```
let obj2 = {};
new Stamper(obj2);
new Stamper(obj2); // Throws 'TypeError' due to pre-existence of private field
```

This ability to stamp private fields into arbitrary objects interacts with the WeakMap model a bit here as well. For example, given that you can stamp private fields onto any object, that means you could also stamp a private field onto a sealed object:

```
var obj3 = {};
Object.seal(obj3);
```

```
new Stamper(obj3);
Stamper.getX(obj3); // => "stamped"
```

If you imagine private fields as properties, this is uncomfortable, because it means you're modifying an object that was sealed by a programmer to future modification. However, using the weak map model, it is totally acceptable, as you're only using the sealed object as a key in the weak map.

PS: Just because you *can* stamp private fields into arbitrary objects, doesn't mean you *should*: Please don't do this.

Implementing the Specification

When faced with implementing the specification, there is a tension between following the letter of the specification, and doing something different to improve the implementation on some dimension.

Where it is possible to implement the steps of the specification directly, we prefer to do that, as it makes maintenance of features easier as specification changes are made. SpiderMonkey does this in many places. You will see sections of code that are transcriptions of specification algorithms, with step numbers for comments. Following the exact letter of the specification can also be helpful where the specification is highly complex and small divergences can lead to compatibility risks.

Sometimes however, there are good reasons to diverge from the specification language. JavaScript implementations have been honed for high performance for years, and there are many implementation tricks that have been applied to make that happen. Sometimes recasting a part of the specification in terms of code already written is the right thing to do, because that means the new code is also able to have the performance characteristics of the already written code.

Implementing Private Names

The specification language for Private Names already almost matches the semantics around `Symbols`, which already exist in SpiderMonkey. So adding `PrivateNames` as a special kind of `Symbol` is a fairly easy choice.

Implementing Private Fields

Looking at the specification for private fields, the specification implementation would be to add an extra hidden slot to every object in SpiderMonkey, which contains a reference to a list of `{PrivateName, Value}` pairs. However, implementing this directly has a number of clear downsides:

- It adds memory usage to objects without private fields
- It requires invasive addition of either new bytecodes or complexity to performance sensitive property access paths.

An alternative option is to diverge from the specification language, and implement only the semantics, not the actual specification algorithms. In the majority of cases, you really *can* think of private fields as special properties on objects that are hidden from reflection or introspection outside a class.

If we model private fields as properties, rather than a special side-list that is maintained with an object, we are able to take advantage of the fact that property manipulation is already extremely optimized in a JavaScript engine.

However, properties are subject to reflection. So if we model private fields as object properties, we need to ensure that reflection APIs don't reveal them, and that you can't get access to them via Proxies.

In SpiderMonkey, we elected to implement private fields as hidden properties in order to take advantage of all the optimized machinery that already exists for properties in the engine. When I started implementing this feature André Bargull – a SpiderMonkey contributor for many years – actually handed me a series of patches that had a good chunk of the private fields implementation already done, for which I was hugely grateful.

Using our special PrivateName symbols, we effectively desuagar

```
class A {
  #x = 10;
  x() {
    return this.#x;
  }
}
```

to something that looks closer to

```
class A_desugared {
  constructor() {
    this[PrivateSymbol(#x)] = 10;
  }
  x() {
    return this[PrivateSymbol(#x)];
  }
}
```

Private fields have slightly different semantics than properties however. They are designed to issue errors on patterns expected to be programming mistakes, rather than silently accepting it. For example:

1. Accessing an a property on an object that doesn't have it returns `undefined`. Private fields are specified to throw a `TypeError`, as a result of the `PrivateFieldGet` algorithm.
2. Setting a property on an object that doesn't have it simply adds the property. Private fields will throw a `TypeError` in `PrivateFieldSet`.
3. Adding a private field to an object that already has that field also throws a `TypeError` in `PrivateFieldAdd`. See "The Constructor Override Trick" above for how this can happen.

To handle the different semantics, we modified the bytecode emission for private field accesses. We added a new bytecode op, `CheckPrivateField` which verifies an object has the correct state for a given private field. This means throwing an exception if the property is missing or present, as appropriate for Get/Set or Add. `CheckPrivateField` is emitted just before using the regular 'computed property name' path (the one used for `A[someKey]`).

`CheckPrivateField` is designed such that we can easily implement an inline cache using CacheIR. Since we are storing private fields as properties, we can use the Shape of an object as a guard, and simply return the appropriate boolean value. The Shape of an object in SpiderMonkey determines what properties it has, and where they are located in the storage for that object. Objects that have the same shape are guaranteed to have the same properties, and it's a perfect check for an IC for `CheckPrivateField`.

Other modifications we made to make to the engine include omitting private fields from the property enumeration protocol, and allowing the extension of sealed objects if we are adding private field.

Proxies

Proxies presented us a bit of a new challenge. Concretely, using the `Stamper` class above, you can add a private field directly to a Proxy:

```
let obj3 = {};
let proxy = new Proxy(obj3, handler);
new Stamper(proxy)

Stamper.getX(proxy) // => "stamped"
Stamper.getX(obj3)  // TypeError, private field is stamped
                    // onto the Proxy Not the target!
```

I definitely found this surprising initially. The reason I found this surprising was I had expected that, like other operations, the addition of a private field would tunnel through the proxy to the target. However, once I was able to internalize the WeakMap mental model, I was able to understand this example much better. The trick is that in the WeakMap model, it is the `Proxy`, not the target object, used as the key in the `#x` WeakMap.

These semantics presented a challenge to our implementation choice to model private fields as hidden properties however, as SpiderMonkey's Proxies are highly specialized objects that do not have room for arbitrary properties. In order to support this case, we added a new reserved slot for an 'expando' object. The expando is an object allocated lazily that acts as the holder for dynamically added properties on the proxy. This pattern is used already for DOM objects, which are typically implemented as C++ objects with no room for extra properties. So if you write `document.foo = "hi"`, this allocates an expando object for `document`, and puts the `foo` property and value in there instead. Returning to private fields, when `#x` is accessed on a Proxy, the proxy code knows to go and look in the expando object for that property.

In Conclusion

Private Fields is an instance of implementing a JavaScript language feature where directly implementing the specification as written would be less performant than re-casting the specification in terms of already optimized engine primitives. Yet, that recasting itself can require some problem solving not present in the specification.

At the end, I am fairly happy with the choices made for our implementation of Private Fields, and am excited to see it finally enter the world!

Acknowledgements

I have to thank, again, André Bargull, who provided the first set of patches and laid down an excellent trail for me to follow. His work made finishing private fields much easier, as he'd already put a lot of thought into decision making.

Jason Orendorff has been an excellent and patient mentor as I have worked through this implementation, including two separate implementations of the private field bytecode, as well as two separate

implementations of proxy support.

Thanks to Caroline Cullen, and Iain Ireland for helping to read drafts of this post, and to Steve Fink for fixing many typos.

About Matthew Gaudet

More articles by Matthew Gaudet...

2 comments

A I

Thanks for the writeup. Very interesting to learn about the underlying machinery.

Had no idea the constructor trick was even possible. I wonder how many actual people do use it and why this isn't a case of simply changing the spec to better fit the most likely use.

June 11th, 2021 at 17:18

> **Matthew Gaudet** AUTHOR
>
> I did a dive into why that exists when I discovered it, and wrote that up here. Essentially this all ties back to trying to maintain compatibility with some pre-class patterns was my take-away.
>
> June 15th, 2021 at 09:25

Comments are closed for this article.