Access to out-of-scope private class members in the DevTools console

Attention - this doc is public and shared with the world!

Contact: Joyee Cheung <joyee@igalia.com>
Contributors: Joyee Cheung <joyee@igalia.com>, Caitlin Potter <caitp@igalia.com>
Status: Inception | Draft | **Accepted** | Done

LGTMs needed

Name	**Write (not) LGTM in this row**
verwaest@chromium.org	
szuend@chromium.org	LGTM DevTools side
leszeks@chromium.org	LGTM

Abstract

This document describes ideas about implementing access to out-of-scope private class members in the DevTools console/REPL.

At the moment it is possible to access the private class members from the console when the user is debugging within a scope that has valid access to the members, but the access is no longer available once they are out of any valid scopes. The goal of this work is to support inspection of this kind of extraordinary inspection that would not be otherwise valid in normal execution, in order to provide a better debugging experience.

Related issue: chromium:1381806
Patch: https://chromium-review.googlesource.com/c/v8/v8/+/4020267

Background

Currently, when the user is debugging an object that has private class members while being out of the class body, like this:

```
1  'use strict';
2
3  function run() {
4    class Klass {  Klass = class Klass
5      #name = 1;
6    }
7    const obj = new Klass;  obj = Klass {#name: 1}, Klass = class Klass
8    return obj;
9  }
10
11 const obj = run();
12
```

They can inspect all the private fields with by logging the object with console calls, but it's not possible to evaluate a specific private class member once they are out of the class body:

```
> obj
◂ ▸ Klass {#name: 1}
> obj.#name
✕ ▸ Uncaught SyntaxError: Private field '#name' must be declared in an enclosing class
      at run (index.js:8:14)
      at index.js:11:13
```

When debugging an object with many private class members, it would be difficult for the user to locate and inspect one specific private class member that they are interested in out of a long list of members. Neither can the user evaluate an expression with the private class member to facilitate debugging. The user experience would be nicer if the user can directly evaluate `obj.#name` from the console when they are outside the class body, and/or modifying it, like what's possible with native debuggers when debugging private class members in C++.

There are two Chrome DevTools protocol methods related to this kind of access:

- `Debugger.evaluateOnCallFrame()`, used by the console when the user evaluates an expression while step-debugging
- `Runtime.evalaute()`, used by the console when the user evaluates an expression when the debugger is not running (for example, if they make the object available to the global scope, and attempt to evaluate private names in that object when they are not debugging).

Complications

Unlike ordinary properties, there can be more than one private field with the same name in an object, each belonging to different classes. For example:

```
1  'use strict';
2
3  function run() {
4    class Klass {  Klass = class Klass
5      #name = 1;
6    }
7    class ChildKlass extends Klass {  ChildKlass = class ChildKlass, Klass = class Klass
8      #name = 1;
9    }
10
11   const obj = new ChildKlass;  obj = ChildKlass {#name: 1, #name: 1}, ChildKlass = class ChildKlass
12   console.log(obj);
13   return obj;
14 }
15
16 run();
17
```

Leads to an object with two private fields named `#name`:

```
▼ ChildKlass {#name: 1, #name: 1} i
    #name: 1
    #name: 1
  ▶ [[Prototype]]: Klass
```

Or the conflict can come from classes from the same source position:

```
3  function factory(Base) {
4    class Klass extends Base {
5      #name = 1;
6    }
7    return Klass;
8  }
9
10 const Klass1 = factory(class {});
11 const Klass2 = factory(Klass1);
12 const obj = new Klass2;
13 console.log(obj);
14
```

```
▼ Klass {#name: 1, #name: 1} i
    #name: 1
    #name: 1
  ▶ [[Prototype]]: Klass
```

It would be even nicer if DevTools supports distinguishing these conflicting private fields visually, or selecting the desired private field when the user attempts to modify it, though for the initial support we can just throw a “no-support” error for ambiguous cases like this.

Design

Parser

Normally, access to private names are eagerly checked in the parser as well as in the scope resolution. For example, with a snippet like this:

```
class Klass {
 #name = 1;
}
const obj = new Klass;
obj.#name;  // Throws
```

When parsing the expression `obj.#name` outside the class scope in normal execution, V8 currently immediately checks the scope chain and see if there's any outer class scope surrounding the expression that can contain private class members. If there isn't any, which is true in this case, V8 directly throws a `SyntaxError` and stop parsing, since it already knows that it's impossible for to resolve `#name` to any valid private names in a scope chain without valid class scopes.

In order to support the extraordinary access in the console, when we are parsing an expression containing `obj.#name` for the console, we should bypass the eager checks and delegate the lookup of private class members to the runtime. We can reuse two fields in the `UnoptimizedCompileFlags` to know when to relax the checks in the parser:

- `parsing_while_debugging` for `Debugger.evaluateOnCallframe()`
- `is_repl_mode` for `Runtime.evaluate()`

And if we are parsing for the console, we should proceed even when we are aware that we are not in any scope valid for the access.

Scope Resolution

When parsing a private name that needs to be resolved in normal execution, V8 currently finds the nearest class scope in the scope chain and prepend it to a list containing all private names pending to be resolved. Once a class literal is parsed, V8 attempts to resolve all unresolved private names in it using the private names declared inside the class literal (with `ClassScope::ResolvePrivateNamesPartially()`). If there are still some private name access that cannot be resolved using known private names, V8 checks whether there's any other outer class scope, if there is one, we push all the remaining unresolved private names to

the outer class scope, otherwise we would know that these names cannot be resolved further so we would simply throw there.

To support the extraordinary access, we should delay the resolution of these unresolvable private names to the runtime, and look them up dynamically then. In order to do this we can pass the flags from the parser into `ClassScope::ResolvePrivateNamesPartially()` as well as `ClassScope::ResolvePrivateNames()`, and make sure that when we are compiling for the DevTools console, whenever we find an unresolvable private name, instead of returning to throw an error, we simply declare a non-local variable with `VariableMode::kDynamic` (it does not matter in which scope the variable is declared as we are not going to use the scope information) and bind the unresolvable `VariableProxy` to the dynamic non-local variable. Then we proceed resolving other private names, as if the previous resolution succeeds.

The resolvable private names will still be resolved to the right private name in its outer scope. The unresolvable ones will have a mode of `VariableMode::kDynamic`, which we will use to determine if it's necessary to do a runtime lookup of the private name when emitting the bytecode.

Bytecode Generation

We can add a new assign type `PRIVATE_DEBUG_DYNAMIC` for the extraordinary access in the console, and associate every `Property` node with a private name reference bound to a `VariableMode::kDynamic` variable with it. We already use the assign types to generate special handling of private names (e.g. adding brand checks for private methods, loading getter/setters from accessors for private accessors), so we can just emit runtime calls for looking up the private names dynamically when dealing with the new `PRIVATE_DEBUG_DYNAMIC` assign type.

To implement extraordinary access, we can add two new runtime calls that are responsible for both the lookup and the actual load/store operation:

- `%GetPrivateMember()` which accepts a receiver and a #-prefixed string description of the private name (e.g. "`#name`")
- `%SetPrivateMember()` which accepts a receiver, a string description of the private name, and a value to be set.

For example, when evaluating `obj.#name` expression out of the class body in which the private name `#name` is defined, the following code would be generated:

```
LdaLookupSlot [0]  // Lookup obj dynamically
Star1              // Save obj to r1
LdaConstant [1]    // Load string '#name'
Star3              // Save '#name' to r3
Mov r1, r2         // Save obj to r2
CallRuntime [GetPrivateMember], r2-r3  // Call %GetPrivateMember(obj, '#name')
```

When evaluating `obj.#name + 1`, the following code would be generated

```
LdaLookupSlot [0]  // Lookup obj dynamically
Star1              // Save obj to r1
LdaSmi [1]         // Load Smi 1
Star2              // Save Smi 1 to r2
LdaConstant [1]    // Load string '#name'
Star4              // Save '#name' to r4
Mov r1, r3         // Save obj to r3
Mov r2, r5         // Save Smi 1 to r5
CallRuntime [SetPrivateMember], r3-r5  // Call %SetPrivateMember(obj, '#name', 1)
```

Runtime calls

In the implementation of the runtime calls `%GetPrivateMember()` and `%SetPrivateMember()`:

1. First, we collect all the private names available to the receiver. This can be done by iterating over the receiver to find all private name symbols on it. Private fields are directly stored in the instance, while private methods and accessors can be retrieved by looking at the contexts stored in the instances keyed by the private brand symbols (which are also private names), see Inspector support for private methods.
2. For now, we check the available private names to see if there are more than one private name matching the description (for example, to see if there are more than one `#name` available on the `obj`). If there are any conflicts, throw an error, because it's ambiguous and we are not yet supporting selection among conflicting names.
3. If there is no private name in the receiver matching the description, throw an error too.
4. If there is precisely one private name matching the description, dispatch behaviors based on the type of the private member.

To handle access to the found private name:

1. If it's a private field, simply load from/store in the receiver with the found private name symbol using `Object::GetProperty()` or `Object::SetProperty()`
2. If it's a private method, load the JSFunction found from the context stored in the receiver and return it, or in the case of a store, throw an error, as private methods are not supposed to be writable.
3. If it's a private accessor, invoke the accessor component found from the context stored in the receiver. If the appropriate accessor component is missing (for example, trying to read from a write-only private accessor), throw an error.

Test plan

Tests will be added to V8 for both `Debugger.evaluateOnCallFrame()` and `Runtime.evalaute()` with `replMode: true`. We might want some additional tests from DevTools/Chromium to make sure that they work together as well.

Rollout considerations

As the extraordinary access should be guarded by flags that are only set by the DevTools console, the usage should be limited, so it should be fine to just roll out the feature.