Static class features: Stage 2 update

Daniel Ehrenberg
Igalia, in partnership with Bloomberg
TC39 January 2018

Context

- In November, TC39 split off “static” class features and demoted to Stage 2
- Reason: “static private” hazard for subclassing
- Several TC39 members contributed to a new proposal
  - Thanks for taking the extra time to work on this!
    Kevin Gibbons, Allen Wirfs-Brock, Domenic Denicola, Jordan Harband, Michael Saboff, Yehuda Katz, Justin Ridgewell, Adam Klein, Sathya Gunasekaran, Brian Terlson, Ron Buckton, Rob Palmer, Daniel Rosenwasser, and many more
- This presentation: Stage 2 update
- Next meeting: Stage 3?

Summary of proposal

- Keep static public field declarations
  - Syntax: `static x = y;`
  - Semantics: Own data property definition on constructor
- Add lexically scoped functions to class bodies
  - Syntax: `local function f() { }`
  - Semantics: Function declaration hoisted to the top of the class definition
- Keep private instance methods (separate, stage 3 proposal)
  - Syntax: `#method() { }`
  - Semantics: Non-writable own private field on instances
- Do not add static private fields or methods to classes
- Possible extension: `let`, `const`, `class` declarations in class bodies

Outline of presentation

- Go through main proposal points
- Motivate each aspect of the proposal
- Does this seem like a good plan to the committee?
- Request Stage 3 reviewers for March

Static public fields

Proposal: Stick with the original semantics

- Analogous to instance public fields, but on the constructor
- Own, writable, configurable data properties of the constructor
- Scope:
  - Like an instance field declaration or concise method body
  - this is the constructor; super property access
  - arguments is poisoned
  - Class binding is active (no longer TDZ)
- Evaluation order
  - Computed property name evaluated with others
  - Initializer evaluated after class is done (to avoid class binding TDZ)
  - Evaluated once, just for the constructor where they are defined

Semantics case: Set() on the prototype chain

```
static Counter {
  static count = 0;
  static inc() { this.count++; }
}
class SubCounter extends Counter { }

Counter.hasOwnProperty("count");  // true
SubCounter.hasOwnProperty("count");  // false

Counter.count; // 0, own property
SubCounter.count; // 0, inherited

Counter.inc();  // undefined
Counter.count;  // 1, own property
SubCounter.count;  // 1, inherited
```

```
// ++ will read up the prototype chain and
write an own property
SubCounter.inc();

Counter.hasOwnProperty("count");  // true
SubCounter.hasOwnProperty("count");  // true

Counter.count;  // 1, own property
SubCounter.count;  // 2, own property

Counter.inc(); Counter.inc();
Counter.count;  // 3, own property
SubCounter.count;  // 2, own property
```

Semantics case: Set() on the prototype chain

- This is how JS works in general
- Similar situation with object literals--one mental model

```
let x = { a: 1 };
let y = { __proto__: x };
y.a++;
y.a;  // 2
x.a;  // 1
```

- Regularity > Adding special case
- Utility: analogous to class_attributes in Rails

```
// ++ will read up the prototype chain and
write an own property
SubCounter.inc();

Counter.hasOwnProperty("count");  // true
SubCounter.hasOwnProperty("count");  // true

Counter.count;  // 1, own property
SubCounter.count;  // 2, own property

Counter.inc(); Counter.inc();
Counter.count;  // 3, own property
SubCounter.count;  // 2, own property
```

s/static private/lexical declarations in class bodies/g

Motivation: Refactoring example (from Domenic)

```
class JSDOM {
  #createdBy;
  #registerWithRegistry(registry) {
     // ... elided ...
  }

  static async fromURL(url, options = {}) {
     normalizeFromURLOptions(options);

     const body = await getBodyFromURL(url);
     return JSDOM.#finalizeFactoryCreated(
          body, options, "fromURL");
  }

static fromFile(filename, options = {}) {
   const body = await
       getBodyFromFilename(filename);
   return JSDOM.#finalizeFactoryCreated(
       body, options, "fromFile");
}

static #registry = new JSDOMRegistry();
static #finalizeFactoryCreated(
        body, options, factoryName) {
    normalizeOptions(options);
    Jsdom = new JSDOM(body, options);
   jsdom.#createdBy = factoryName;
   jsdom.#registerWithRegistry(
       JSDOM.#registry);
   return jsdom;
 }
}
```

The Hazard of static private (from Justin Ridgewell)

```
class Base {
  static #field = 'hello';

  static get() {
    return this.#field;
  }
}

class Sub extends Base {}

// This one isn't controversial
Base.get() // => 'hello'

// Throws a TypeError!
Sub.get()
```

Resolution: Provide lexically scoped declarations

```
const registry = new JSDOMRegistry();
export class JSDOM {
  #createdBy;

  #registerWithRegistry(registry) {
    // ... elided ...
  }

  static async fromURL(url, options) {
    url = normalizeFromURLOptions(
        url, options);

    const body = await getBodyFromURL(url);
    return finalizeFactoryCreated(body,
options, "fromURL");
  }

  static async fromFile(filename, options) {
    const body = await
        getBodyFromFilename(filename);
    return finalizeFactoryCreated(
        body, options, "fromFile");
  }

  local function finalizeFactoryCreated(
        body, options, factoryName) {
    normalizeOptions(options);
    let jsdom = new JSDOM(body, options):
    jsdom.#createdBy = factoryName;
    jsdom.#registerWithRegistry(registry);
    return jsdom;
  }
}
```

Details

- `local` keyword makes it clear this is not a method (bikeshed)

- f is available in a, c and g
- g can (lexically) access #d
- Async functions, generators, async generators also supported
- Function is created "at the beginning of the scope"; never a ReferenceError

```
class X extends Y {
    [a]() { }
    static b = c;
    #d;
    local function f() { g; }
}
```

let, const and class declarations in class bodies?

- Execution order: Y, c, b, d, f, h
- Scope of Y, c, b, f: Lexical scope
  - this, super.x, yield, await, arguments inherit from outside of class
- Scope of d, h: Method scope
  - this, super.x work against constructor
  - Disallowed yield, await, arguments
- Leave out var (not block scoped)
- Other kinds of statements disallowed
- Complicated and less clear use cases

```
class X extends Y {
    local let a = b
    static [c] = d;
    local class e extends f { }
    static g = h;
}
```

- Proposal: Not yet
- Consider as a follow-on

Private methods

Private methods and accessors

Introducing for Stage 2

(Blast from the past--these are previously presented slides, with new notes in red)

July 2017

(Currently, Stage 3)

Daniel Ehrenberg
Igalia

Code sample

```
class Counter extends HTMLElement {
  #xValue = 0;

  get #x() { return this.#xValue; }
  set #x(value) {
    this.#xValue = value;
    window.requestAnimationFrame(
      this.#render.bind(this));
  }

  #clicked() {
    this.#x++;
  }

  constructor() {
     super();
     this.onclick = this.#clicked.bind(this);
   }

   connectedCallback() { this.#render(); }

   #render() {
     this.textContent = this.#x.toString();
   }
}
window.customElements.define('num-counter',
Counter);
```

Why?

```
class Counter extends HTMLElement {
  #x = 0;

  connectedCallback() { this.#render(); }

  #render() {
    this.textContent = this.#x.toString();
  }
}
```

- Private methods encapsulate behavior
- You can access private fields inside private methods

Choice of syntax

Private method

```
class Counter extends HTMLElement {
  #x = 0;

  connectedCallback() { this.#render(); }

  #render() {
    this.textContent = this.#x.toString();
  }
}
```

- Similar to other methods
- Easy to change public <-> private
- Conclusion: Select this option

Alternative: Lexically scoped function

```
class Counter extends HTMLElement {
  #x = 0;

  connectedCallback() { render.call(this) }

  function render() {
    this.textContent = this.#x.toString();
  }
}
```

- Incongruous
- Pass receiver with call

Type checking or just a function?

- What does this do?

```
class C {
  #foo() { alert("hi"); }

  bar() {
    this.#foo();
  }
}

C.prototype.bar.call();
```

- TypeError or alert?

- Option: A funny lexically scoped function declaration
  - Simpler to implement
- Option: Similar to a private field
  - Occasionally catch errors sooner
  - Difference between static and instance methods
  - Conclusion: These semantics

Private accessors?

- Pro:
  - Analogous to private methods; why not?
  - Could be useful for large classes
- Con:
  - Often, users could just call the method instead
  - Could be strange to have getter/setters but no reflection
- Open question
- Conclusion: include private accessors

```
class Counter extends HTMLElement {
  #xValue = 0;

  get #x() { return this.#xValue; }
  set #x(value) {
    this.#xValue = value;
  }
}
```

Both private methods and lexically scoped fns?

- Advantages of private instance methods:
  - Easy refactoring between public and private--just add #
  - this, super
  - Terse, convenient, analogous to public methods
- No known hazards of instance private methods (unlike static private)
- JS has always had function-based and method-based phrasing available
- Programming w/ methods often about code organization, not dispatch

Conclusion

Summary

- Keep static public field declarations
  - Syntax: `static x = y;`
  - Semantics: Own data property definition on constructor
- Add lexically scoped functions to class bodies
  - Syntax: `local function f() { }`
  - Semantics: Function declaration hoisted to the top of the class definition
- Keep private instance methods (separate, Stage 3 proposal)
  - Syntax: `#method() { }`
  - Semantics: Non-writable own private field on instances
- Do not add static private fields or methods to classes
- Possible extension: `let`, `const`, `class` declarations in class bodies

Proposal status

- Detailed explainer (including alternatives)
- Specification text
- Static public fields
  - Test262 tests (currently backed out)
  - V8 implementation (behind a flag)
- Lexically scoped declarations in classes
  - No implementations or tests
- Private instance methods
  - Separate Stage 3 proposal
  - No implementations or tests

Next steps

- Follow up on issues
    - Bikeshedding about the token choice [Bug]
    - OK to leave class, let, const as a follow-on? [Bug]
    - Any other sources of hesitation? [File an issue]
    - Happy to have another VC meeting if anyone is interested
- Draft tests, prototype implementations
- Stage 3 reviewers?

Bonus: Analysis of alternatives