Non-extensible Applies to Private

stage 2.7 status update

Mark S. Miller — Agoric

Shu-yu Guo — Apple

Chip Morningstar — METAMASK

Erik Marks — METAMASK

109th Plenary

July 2025

TC 39

Recap:

1.1 PrivateFieldAdd (*O*, *P*, *value*)

The abstract operation PrivateFieldAdd takes arguments *O* (an Object), *P* (a Private Name), and *value* (an ECMAScript language value) and returns either a normal completion containing UNUSED or a throw completion. It performs the following steps when called:

1. If *O*.[[Extensible]] is **false**, throw a **TypeError** exception.
2. If the host is a web browser, then
    a. Perform ? HostEnsureCanAddPrivateElement(*O*).
3. Let *entry* be PrivateElementFind(*O*, *P*).
4. If *entry* is not EMPTY, throw a **TypeError** exception.
5. Append PrivateElement { [[Key]]: *P*, [[Kind]]: FIELD, [[Value]]: *value* } to *O*.[[PrivateElements]].
6. Return UNUSED.

1.2 PrivateMethodOrAccessorAdd (*O*, *method*)

The abstract operation PrivateMethodOrAccessorAdd takes arguments *O* (an Object) and *method* (a PrivateElement) and returns either a normal completion containing UNUSED or a throw completion. It performs the following steps when called:

1. Assert: *method*.[[Kind]] is either METHOD or ACCESSOR.
2. If *O*.[[Extensible]] is **false**, throw a **TypeError** exception.
3. If the host is a web browser, then
    a. Perform ? HostEnsureCanAddPrivateElement(*O*).
4. Let *entry* be PrivateElementFind(*O*, *method*.[[Key]]).
5. If *entry* is not EMPTY, throw a **TypeError** exception.
6. Append *method* to *O*.[[PrivateElements]].
7. Return UNUSED.

Recap:
Structs change shape

```
class Trojan {
  constructor(key) { return key; }
}

class Tagger extends Trojan {
  #value;

  constructor(key, value) {
    super(key);
    this.#value = value;
  }
}

// struct instances born sealed
new Tagger(struct, 'a'); // adds #value to struct anyway
```

Recap:
Structs fixed shape

```
class Trojan {
  constructor(key) { return key; }
}

class Tagger extends Trojan {
  #value;

  constructor(key, value) {
    super(key);
    this.#value = value;
  }
}

// struct instances born sealed
new Tagger(struct, 'a'); // throws TypeError
```

Recap:
WeakMap reachable by syntax alone

```
class Trojan {
  constructor(key) {
    return key;
  }
}

const makeHiddenWeakMap = () => {
  const tombstone = Symbol('deleted');
  class PrivateTagger extends Trojan {
    #value;

    constructor(key, value) {
      super(key);
      this.#value = value;
    }

    static HiddenWeakMap = class {…}
  };
  return PrivateTagger.HiddenWeakMap;
}
```

```
const makeHiddenWeakMap = () => {
  const tombstone = Symbol('deleted');
  class PrivateTagger extends Trojan {
    …
    static HiddenWeakMap = class {
      set(key, value) { new PrivateTagger(key, value); }

      has(key) {
        try {
          return key.#value !== tombstone;
        } catch { return false; }
      }
      get(key) {
        try {
          const value = key.#value;
          return value === tombstone ? undefined : value;
        } catch { return undefined; }
      }
      delete(key) {
        if (this.has(key)) {
          key.#value = tombstone;
          return true;
        }
        return false;
      }
    }
  };
  return PrivateTagger.HiddenWeakMap;
}
```

Recap:
Fatal for virtual object memory

```
class Trojan {
  constructor(key) {
    return key;
  }
}

const makeHiddenWeakMap = () => {
  const tombstone = Symbol('deleted');
  class PrivateTagger extends Trojan {
    #value;

    constructor(key, value) {
      super(key);
      this.#value = value;
    }

    static HiddenWeakMap = class {…}
  };
  return PrivateTagger.HiddenWeakMap;
}
```

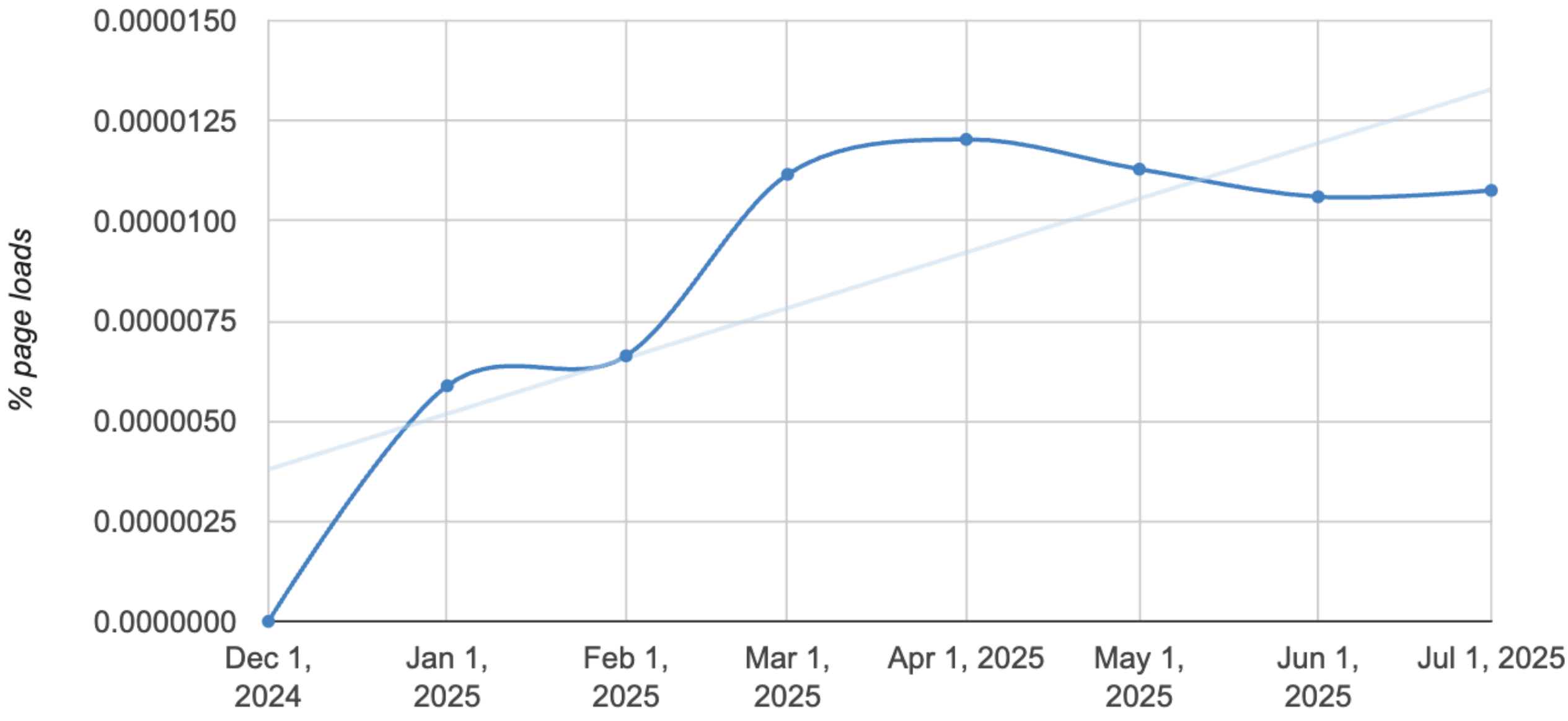
```
const makeHiddenWeakMap = () => {
  const tombstone = Symbol('deleted');
  class PrivateTagger extends Trojan {
    …
    static HiddenWeakMap = class {
      set(key, value) { new PrivateTagger(key, value); }

      has(key) {
        try {
          return key.#value !== tombstone;
        } catch { return false; }
      }
      get(key) {
        try {
          const value = key.#value;
          return value === tombstone ? undefined : value;
        } catch { return undefined; }
      }
      delete(key) {
        if (this.has(key)) {
          key.#value = tombstone;
          return true;
        }
        return false;
      }
    }
  };
  return PrivateTagger.HiddenWeakMap;
}
```

Recap++:

Percentage of page loads over time

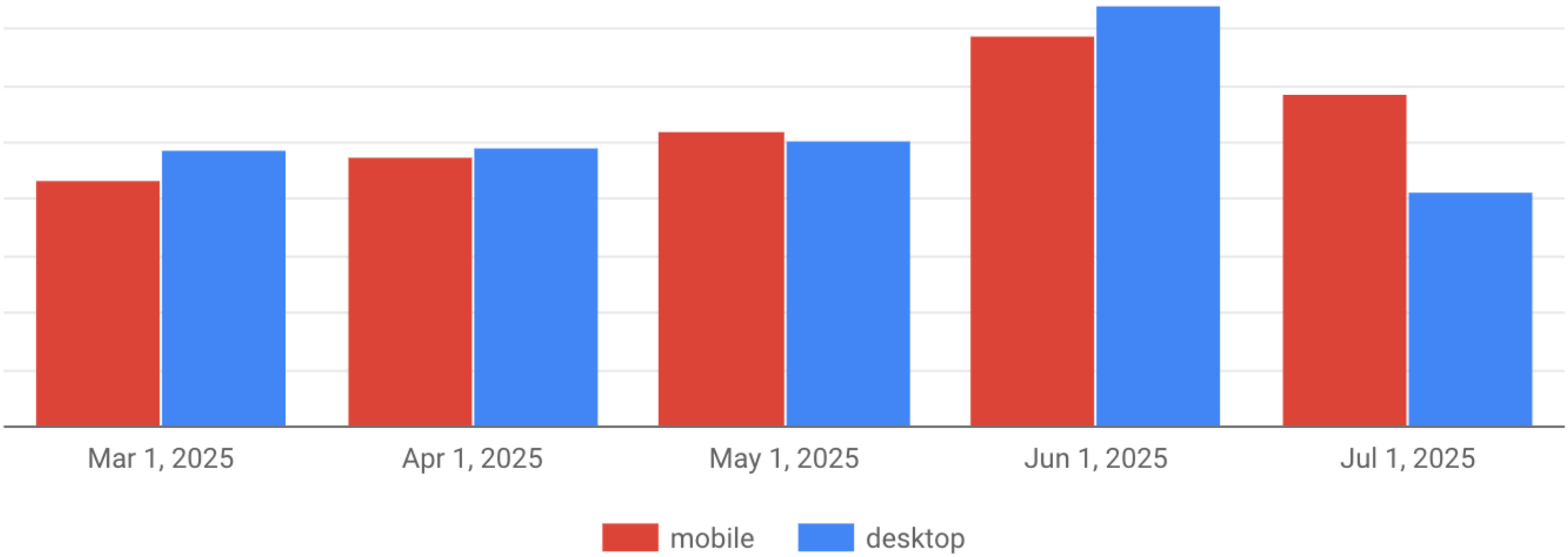
The chart below shows the percentage of page loads (in Chrome) that use this feature at least once. Data is across all channels and platforms. Newly added use counters that are not on Chrome stable yet only have data from the Chrome channels they're on.



Recap++:

Adoption of the feature on top sites

The chart below shows the adoption of the feature by the top URLs on the internet. Data from HTTP Archive.



?

Note: The jump around July and December 2018 are because the corpus of URLs crawled by HTTP Archive increased. These jumps have no correlation with the jump in the top graph. See the announcement for more details.

Recap:

Web compat analysis #1

Open

syg opened 4 days ago

Chrome use counter data show there are indeed in-the-wild usage of extending non-extensible objects with private fields.

The percentage of page load is very low at time of this writing (0.000011%), and are found to be concentrated in two pieces of software.

disy Cadenza

Most breakages come from Cadenza, which seems to be a closed-source German GIS software. The pattern used is freezing a class with nothing but static fields using the RHS of a private field initializer. The minified version looks like the following.

```
class _ {
  static FOO = ...;
  static BAR = ...;
  static #t = void (Object.keys(_).forEach((t) => {
          _[t].type = t;
        }), Object.freeze(_));
}
```

Web compat analysis #1

Open

Jamesernator on Apr 18

> The minified version looks like the following.

Given the field is only used for the initializer, there's a good chance it comes from Babel's downleveling of static blocks as that exhibits basically this exact downleveling.

```
class _ {
  static FOO = "foo";

  static {
    Object.freeze(this);
  }
}
```

```
class _ {
  static FOO = "foo";
  static #_ = (() => Object.freeze(this))();
}
```

Web compat analysis #1

Open

Jamesernator on Apr 18

> The minified version looks like the following.

Given the field is only used for the initializer, there's a good chance it comes from Babel's downleveling of static blocks as that exhibits basically this exact downleveling.

```
class _ {
  static FOO = "foo";

  static {
    Object.freeze(this);
  }
}
```

```
class _ {
  static FOO = "foo";
  static #_ = (() => Object.freeze(this))();
}
```

gibson042 last week · edited by gibson042 — Edits — Member

Yes, downleveling that moves class static blocks to after class definition must also move class static field definitions, because evaluation of static fields and static blocks is interleaved in source order. So I guess it would warrant use of private-field records attached to classes and instances:

▸ **Input**

▾ **Transpiled**

```
// TRANSPILED FROM: export class C {…}
export const C = (() => {
  const __PRIVATE_FIELDS__ = new WeakMap();
  const __DEFINE_PRIVATE_FIELD__ = (obj, name, value) => {
    // Support for proposal-nonextensible-applies-to-private requires each private field
    // definition to verify extensibility of the object.
    if (!isExtensible(obj)) throw TypeError("object is not extensible");
```

```
const __FIELDS__ = new WeakMap();

const __DEFINE_FIELD__ = (obj, name, value) => {
  if (!isExtensible(obj)) throw TypeError('object is not extensible');

  const privateFields = __FIELDS__.get(obj) || { __proto__: null };
  __FIELDS__.set(obj, privateFields);
  privateFields[name] = value;
};

const __HAS_FIELD__ = (obj, name) => {
  if (obj === null || (T !== "object" && T !== "function")) throw TypeError(`...`);
  const privateFields = __FIELDS__.get(obj);
  return privateFields ? hasOwn(privateFields, name) : false;
};

const __GET_FIELD__ = (obj, name) => {
  if (!__HAS_FIELD__(obj, name)) TypeError(`...`);
  return __FIELDS__.get(obj)[name];
};

const __SET_FIELD__ = (obj, name, value) => {
  if (!__HAS_FIELD__(obj, name))  throw TypeError(`...`);
  __FIELDS__.set(obj)[name] = value;
};
```

```
const __FIELDS__ = new WeakMap();

const __DEFINE_FIELD__ = (obj, name, value) => {
  if (!isExtensible(obj)) throw TypeError('object is not extensible');

  const privateFields = __FIELDS__.get(obj) || { __proto__: null };
  __FIELDS__.set(obj, privateFields);
  privateFields[name] = value;
};

const __HAS_FIELD__ = (obj, name) => {
  if (obj === null || (T !== "object" && T !== "function")) throw TypeError(`...`);
  const privateFields = __FIELDS__.get(obj);
  return privateFields ? hasOwn(privateFields, name) : false;
};

const __GET_FIELD__ = (obj, name) => {
  if (!__HAS_FIELD__(obj, name)) TypeError(`...`);
  return __FIELDS__.get(obj)[name];
};

const __SET_FIELD__ = (obj, name, value) => {
  if (!__HAS_FIELD__(obj, name))  throw TypeError(`...`);
  __FIELDS__.set(obj)[name] = value;
};
```

Which JS source version?

```
const __FIELDS__ = new WeakMap();

const __DEFINE_FIELD__ = (obj, name, value) => {
  if (!isExtensible(obj)) throw TypeError('object is not extensible');

  const privateFields = __FIELDS__.get(obj) || { __proto__: null };
  __FIELDS__.set(obj, privateFields);
  privateFields[name] = value;
};

const __HAS_FIELD__ = (obj, name) => {
  if (obj === null || (T !== "object" && T !== "function")) throw TypeError(`...`);
  const privateFields = __FIELDS__.get(obj);
  return privateFields ? hasOwn(privateFields, name) : false;
};

const __GET_FIELD__ = (obj, name) => {
  if (!__HAS_FIELD__(obj, name)) TypeError(`...`);
  return __FIELDS__.get(obj)[name];
};

const __SET_FIELD__ = (obj, name, value) => {
  if (!__HAS_FIELD__(obj, name))  throw TypeError(`...`);
  __FIELDS__.set(obj)[name] = value;
};
```

"Initial" Global WeakMap

Which JS source version?

Summary?

What do Google's new numbers mean?

How bad is the Babel problem, really?

Babel compromise transpilations?

Questions?

1.1 PrivateFieldAdd (*O*, *P*, *value*)

The abstract operation PrivateFieldAdd takes arguments *O* (an Object), *P* (a Private Name), and *value* (an ECMAScript language value) and returns either a normal completion containing UNUSED or a throw completion. It performs the following steps when called:

1. If *O*.[[Extensible]] is **false**, throw a **TypeError** exception.
2. If the host is a web browser, then
    a. Perform ? HostEnsureCanAddPrivateElement(*O*).
3. Let *entry* be PrivateElementFind(*O*, *P*).
4. If *entry* is not EMPTY, throw a **TypeError** exception.
5. Append PrivateElement { [[Key]]: *P*, [[Kind]]: FIELD, [[Value]]: *value* } to *O*.[[PrivateElements]].
6. Return UNUSED.

1.2 PrivateMethodOrAccessorAdd (*O*, *method*)

The abstract operation PrivateMethodOrAccessorAdd takes arguments *O* (an Object) and *method* (a PrivateElement) and returns either a normal completion containing UNUSED or a throw completion. It performs the following steps when called:

1. Assert: *method*.[[Kind]] is either METHOD or ACCESSOR.
2. If *O*.[[Extensible]] is **false**, throw a **TypeError** exception.
3. If the host is a web browser, then
    a. Perform ? HostEnsureCanAddPrivateElement(*O*).
4. Let *entry* be PrivateElementFind(*O*, *method*.[[Key]]).
5. If *entry* is not EMPTY, throw a **TypeError** exception.
6. Append *method* to *O*.[[PrivateElements]].
7. Return UNUSED.