Extending Built-ins

Kevin Gibbons and Michael Ficarra

89th meeting of Ecma TC39, March 2022

[How] should built-ins support extension?

Simple example: `Set.prototype.addAll`

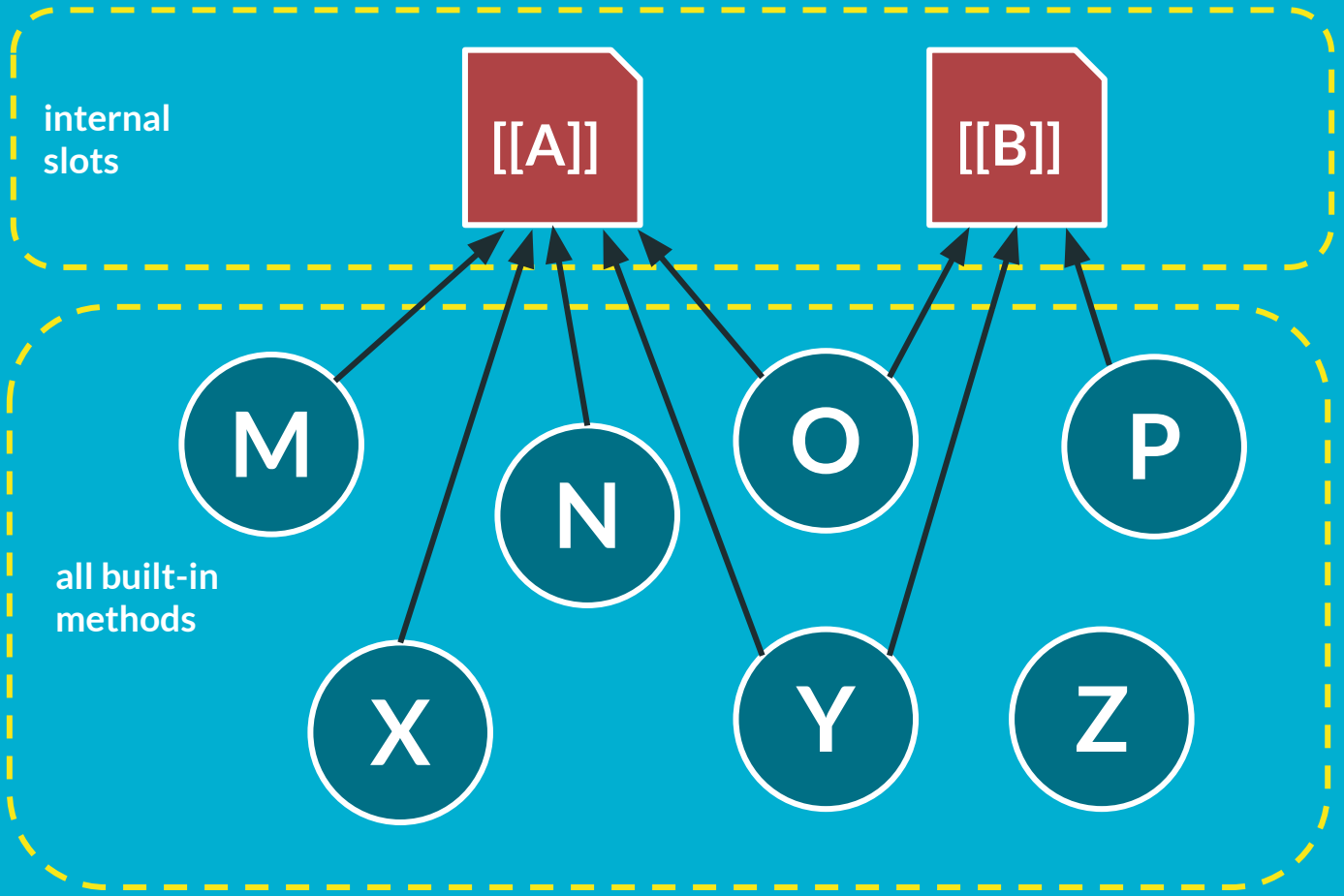
use internal slot

```
Set.prototype.addAll =
  function (iterable) {
    for (let item of iterable) {
      this.[[SetData]].push(item);
    }
  };
```

call public method

```
Set.prototype.addAll =
  function (iterable) {
    for (let item of iterable) {
      this.add(item);
    }
  };
```

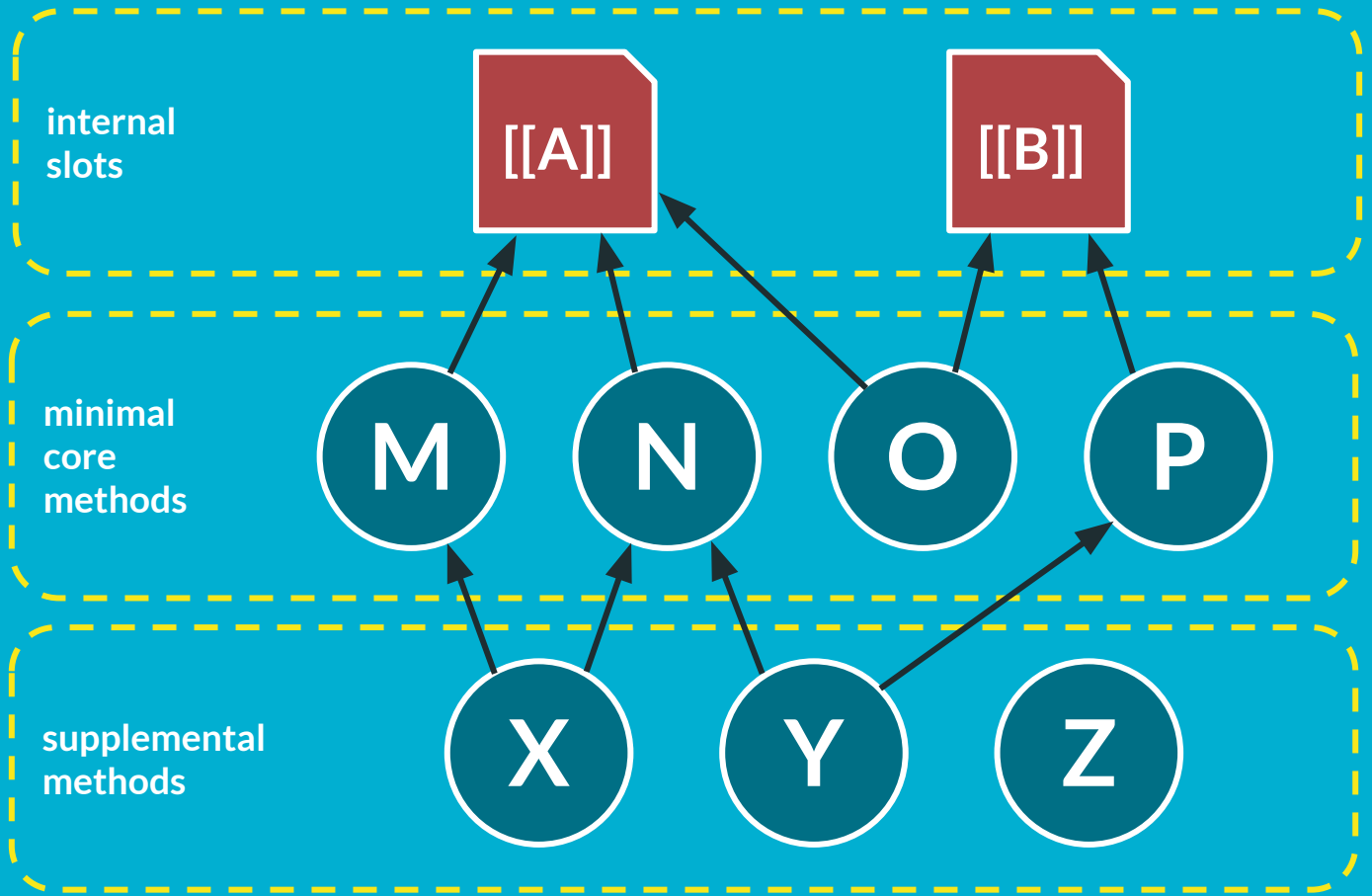
Approach 1: Direct use of internal slots


internal
slots
[[A]]
[[B]]
all built-in
methods
M
N
O
P
X
Y
Z

Approach 1: Direct use of internal slots

- **pro:** simple; likely more efficient implementation in engines
- **con:** subclasses must override all methods, and are more likely to require updates when new methods are added to the language

Approach 2: Methods delegate to a "minimal core"


internal
slots
[[A]]
[[B]]
minimal
core
methods
M
N
O
P
supplemental
methods
X
Y
Z

Approach 2: Methods delegate to a "minimal core"

- **pro:** subclasses which override the "core" get new supplemental methods for free
- **pro:** new supplemental methods do not break existing subclasses
- **con:** overhead in engines
- **con:** requires fully specifying algorithms

Hybrid approaches are also available

use slot when present

```
Set.prototype.addAll =
  function (iterable) {
    if ([[SetData]] in this) {
      for (let item of iterable) {
        this.[[SetData]].push(item);
      }
    } else {
      for (let item of iterable) {
        this.add(item);
      }
    }
  };
```

ctor parameter switch

```
Set.prototype.addAll =
  function (iterable) {
    if (this.[[UseProtocol]]) {
      for (let item of iterable) {
        this.add(item);
      }
    } else {
      for (let item of iterable) {
        this.[[SetData]].push(item);
      }
    }
  };

class SetSubclass extends Set {
  constructor(iterable) {
    super(iterable, { useProtocol: true });
  }
}
```

No strong precedent today

- few examples of intentional affordances for extension
- but, few examples where an affordance would make sense

Adding new Set/Map methods requires choosing a design

Using `[[SetData]]` on both `this` and `other`

```
Set.prototype.intersection = function(other) {
  let result = new %Set%;
  result.[[SetData]] = %IntersectLists%(this.[[SetData]], other.[[SetData]]);
  return result;
};
```

Using `[[SetData]]` on `this`; methods on `other`

```
Set.prototype.intersection = function(other) {
  let thisSize = this.[[SetData]].length;
  let otherSize = %ToNumber%(other.size);
  let resultList = new %List%;
  if (thisSize > otherSize) {
    for (let item of other)
      if (%ListContainsItem%(this.[[SetData]], item))
        %AddItemToList%(resultList, item);
  } else {
    let otherHas = %BoundFunctionCreate%(other.has, other, []);
    for (let item of this.[[SetData]])
      if (otherHas(item))
        %AppendToList%(resultList, item);
  }
  let result = new %Set%;
  result.[[SetData]] = resultList;
  return result;
};
```

Using methods on `this` and `other`

```
Set.prototype.intersection = function(other) {
  let thisSize = %ToNumber%(this.size);
  let otherSize = %ToNumber%(other.size);
  let resultList = new %List%;
  if (thisSize > otherSize) {
    let thisHas = %BoundFunctionCreate%(this.has, this, []);
    for (let item of other)
      if (thisHas(item))
        %AddItemToList%(resultList, item);
  } else {
    let otherHas = %BoundFunctionCreate%(other.has, other, []);
    for (let item of this)
      if (otherHas(item))
        %AppendToList%(resultList, item);
  }
  let result = new %Set%;
  result.[[SetData]] = resultList;
  return result;
};
```

Q: Which built-ins should support extension?

higher level: yes?

- Map
- Set
- Other future data structures

lower level: no?

- ArrayBuffer
- RegExp
- Most things in the standard library today

Further Questions

- Should `Symbol.species` continue to be an extension affordance generally?
    - If not, should we always construct the base class, or use `this.constructor`?
- Do static constructors (`Constructor.from`, etc) construct the base class, or `this`?
- When an argument to a method is expected to be an instance of a class, how should it be consumed?

Fundamentally: what is our design philosophy for extending built-ins?

Subclasses cannot enforce additional invariants

You don’t own the internal slots. Consumers can bypass invariant enforcement:

- Calling built-in methods from the superclass directly
- New methods may be added (in the minimal core design, to the minimal core)

So if you want to enforce new invariants, you need a wrapper, not a subclass.

Subclassing is only useful for adding new functionality, not new invariants.

(∴ Set.prototype.union should accept Set-likes as an argument)

Minimal core is too complicated

- Adding new methods requires new considerations
  - Like implementation freedom vs user-visible API
  - Existing methods become incoherent if minimal core expands
- Per last slide there's little benefit in overriding built-in methods anyway

Appendix: taxonomy of existing affordances

- using `Symbol.species` to create new instances, in
  - `RegExp.prototype`
    - `Symbol.matchAll`, `Symbol.split`
  - `Array.prototype`
    - `concat`, `filter`, `flat`, `flatMap`, `map`, `slice`, `splice`
  - `TypedArray.prototype`
    - `filter`, `map`, `slice`, `subarray`
  - `ArrayBuffer.prototype.slice`
  - `Promise.prototype`
    - `then`, `finally`

Appendix: taxonomy of existing affordances

- delegation to other methods
  - Set/Map constructors call `this.add/set`
  - `RegExp` methods call `this.exec`
  - `Promise.prototype.{catch, finally}` call `this.then`
  - `Array.prototype.toString` calls `this.join`
  - non-examples:
    - `Set/Map/Array forEach` methods do not use the iterable protocol on `this`

Appendix: taxonomy of existing affordances

- static methods consult `this`
  - `Promise.{all,race,any,allSettled}` invoke `this.resolve`
  - `Array.{of,from}` and `TypedArray.{of,from}` construct `this`

Appendix 2: Existing users of subclassing

Set and Map already have an extension point

24.2.1.1 Set ([*iterable*])

When the **Set** function is called with optional argument *iterable*, the following steps are taken:

1. If NewTarget is **undefined**, throw a **TypeError** exception.
2. Let *set* be ? OrdinaryCreateFromConstructor(NewTarget, "%**Set.prototype**%", « [[SetData]] »).
3. Set *set*.[[SetData]] to a new empty List.
4. If *iterable* is either **undefined** or **null**, return *set*.
5. Let *adder* be ? Get(*set*, **"add"**).
6. If IsCallable(*adder*) is **false**, throw a **TypeError** exception.
7. Let *iteratorRecord* be ? GetIterator(*iterable*).
8. Repeat,
    a. Let *next* be ? IteratorStep(*iteratorRecord*).
    b. If *next* is **false**, return *set*.
    c. Let *nextValue* be ? IteratorValue(*next*).
    d. Let *status* be Call(*adder*, *set*, « *nextValue* »).
    e. IfAbruptCloseIterator(*status*, *iteratorRecord*).

People are extending built-ins today

```
660  // Contains a set of values and will yell at you if you try to add a value twice.
661  class UniqueSet extends Set {
662    constructor(items) {
663      super();
664      if (items !== undefined) {
665        for (const item of items) {
666          this.add(item);
667        }
668      }
669    }
670
671    add(value, message) {
672      if (message === undefined) {
673        message = `Value '${value}' needs to be unique but it is already in the set`;
674      }
675      assert_true(!this.has(value), message);
676      super.add(value);
677    }
678  }
```

UniqueSet in [Web Platform Tests]

Sidebar: many current examples shouldn't

```
class FakeDisk extends Map {
  constructor() {
    super(...arguments);
    this.getter = super.get.bind(this);
    this.setter = super.set.bind(this);
    this.deleter = super.delete.bind(this);
  }

  async get(key, cb) {
    return asyncify(() => cb(this.getter(key)));
  }

  async set(key, value, cb) {
    return asyncify(() => cb(this.setter(key, value)));
  }

  async delete(key, cb) {
    return asyncify(() => cb(this.deleter(key)));
  }
  async remove(key, cb) {
    return this.delete(key, cb);
  }
}
```

FakeDisk in Privacy Possum

Sidebar: many current examples shouldn't

```
module.exports = class SeekOffsets extends Map {
  set(topic, partition, offset) {
    super.set([topic, partition], offset)
  }

  has(topic, partition) {
    return Array.from(this.keys()).some(([t, p]) => t === topic && p === partition)
  }

  pop() {
    if (this.size === 0) {
      return
    }

    const [key, offset] = this.entries().next().value
    this.delete(key)
    const [topic, partition] = key
    return { topic, partition, offset }
  }
}
```

SeekOffsets in KafkaJS

People aren't aware of the current extension point

```
/** @extends {Set<string>} */
export class InsensitiveStringSet extends Set {
  /**
   * @param {Array<String>} [keys] Optional, initial keys
   */
  constructor(keys = []) {
    super();
    for (const key of keys) {
      this.add(key);
    }
  }
  /**
   * @param {string} key
   */
  add(key) {
    if (!this.has(key) && !this.getCanonicalKey(key)) {
      return super.add(key);
    }
    return this;
  }
```

InsensitiveStringSet in W3C ReSpec

Sometimes these affordances break users

```
673  class StateMap extends Map {
674
675      /**
676       * Create a reactive Map.
677       *
678       * @param {string} name the property name
679       * @param {StateManager} stateManager the state manager
680       * @param {iterable} iterable an iterable object to create the Map
681       */
682      constructor(name, stateManager, iterable) {
683          // We don't have any "this" until be call super.
684          super(iterable);
685          this.name = name;
686          this.stateManager = stateManager;
687      }
688
689      /**
690       * Set an element into the map.
691       *
692       * Each value needs it's own id attribute. Objects without id will be rejected.
693       * The function will throw an error if the value id and the key are not the same.
694       *
695       * @param {*} key the key to store
696       * @param {*} value the value to store
697       * @returns {Map} the resulting Map object
698       */
699      set(key, value) {
700
701          // Only mutations should be able to set state values.
702          if (this.stateManager.readonly) {
703              throw new Error(`State locked. Use mutations to change ${key} value in ${this.name}.`);
704          }
```

StateMap in Moodle

Sometimes these affordances break users

```
class LimitedSet extends Set {
  constructor(limit, slop = Math.round(limit * 0.25), iterable = undefined) {
    super(iterable);
    this.limit = limit;
    this.slop = slop;
  }

  truncate(limit) {
    for (let item of this) {
      // Live set iterators can ge relatively expensive, since they need
      // to be updated after every modification to the set. Since
      // breaking out of the loop early will keep the iterator alive
      // until the next full GC, we're currently better off finishing
      // the entire loop even after we're done truncating.
      if (this.size > limit) {
        this.delete(item);
      }
    }
  }

  add(item) {
    if (this.size >= this.limit + this.slop && !this.has(item)) {
      this.truncate(this.limit - 1);
    }
    super.add(item);
  }
}
```

LimitedSet in Firefox