ES6 Max-min class semantics

Annotated with July 26 Decisions

TC39 – July 2012
Allen Wirfs-Brock
Mozilla
Class BNF

ClassDeclaration : class BindingIdentifier ClassTail

ClassExpression : class BindingIdentifier opt ClassTail

ClassTail : ClassHeritage opt { ClassBody opt }

ClassHeritage : extends AssignmentExpression

ClassBody : ClassElementList

ClassElementList :
  ClassElement
  ClassElementList ClassElement

ClassElement :
  MethodDefinition
  ;

MethodDefinition :
  PropertyName ( FormalParameterList ) { FunctionBody }
  * PropertyName ( FormalParameterList ) { FunctionBody }
  get PropertyName ( ) { FunctionBody }
  set PropertyName ( PropertySetParameterList ) { FunctionBody }
Class Definitions: Early Errors

• Class name is either `eval` or `arguments`
• Duplicate class element names (except for get/set pairs)
• `extends` expression contains a `yield` subexpression – no, yield is ok here (if in generator)
• The method name `constructor` is used on a get, set, or generator method definition.
Semantic Decisions?

• Class declarations create `const` bindings
  ```javascript
  class Foo {}
  Foo = 42; //error, assigning to const
  ```
  No, make them let bindings

• Class initialization is not hoisted
  ```javascript
  new Bar; //runtime error, not init’ed
  ```
  ```javascript
  class Bar{}
  ```

• Class bodies are `strict mode` contexts.
  ```javascript
  class Baz{
      m(){with (x){}} //syntax error
  }
  ```
  No, strictness determined contextually like in ES5

• Default constructor has an empty body
  ```javascript
  class E1 {} //these are equivalent
  ```
  ```javascript
  class E2 {constructor(){}}
  ```
  No, adopted Luke’s suggestion. See last slide
Semantics Decisions?

Local class name scoping

• Similar to *FunctionExpression*, the name in a class definition is (const) bound in a new scope:

```javascript
let F = class Foo {
  meth () {return new Foo}
}
console.log(Foo);  //undefined
```

Only create local binding for *ClassExpressions*. *ClassDeclarations* don’t get one. This is maintaining parallel with function definitions.
extends rules

- **Missing extends**: `class Foo {}`
  
  Foo.[[Prototype]]: *intrinsic* Function.prototype  
  Foo.prototype.[[Prototype]]: *intrinsic* Object.prototype

- **extends null**: `class Foo extends null {}`
  
  Foo.[[Prototype]]: *intrinsic* Function.prototype  
  Foo.prototype.[[Prototype]]: null

- **extends a constructor**:
  
  `class Foo extends Object {}`
  
  Foo.[[Prototype]]: (Object)  
  Foo.prototype.[[Prototype]]: (Object).prototype  //may be null

- **extends a non-constructor**:
  
  `class Foo extends Object.prototype {}`
  
  Foo.[[Prototype]]: *intrinsic* Function.prototype  
  Foo.prototype.[[Prototype]]: (Object.prototype)
extends Errors

- extends value is neither an object or null
  ```javascript
  class foo extends 42 {}  //type error
  class bar extends undefined {}  //type error
  ```
- extends value is a constructor but its prototype value neither an object or null
  ```javascript
  function F1() {};
  f1.prototype = undefined;
  class foo extends F1{}  //type error
  f1.prototype = “some string”;
  class bar extends F1{}  //type error
  ```
Yes to all
## Semantic Decisions? constructor/proto invariants

- **Prototype’s** `constructor` **property is “frozen”**

```javascript
class Foo {constructor () {}}
Foo.prototype.constructor = 0;
delete Foo.prototype.constructor;
```

- **Constructor’s** `prototype` **property is “frozen”**

```javascript
Foo.prototype = 0;
delete Foo.prototype;
```

### Table: Invariants of `constructor` and `prototype` properties

<table>
<thead>
<tr>
<th></th>
<th><code>function () {}</code></th>
<th><code>Built-in constructors</code></th>
<th><code>Class (){}</code></th>
</tr>
</thead>
</table>
Semantic Decisions?
method attributes

• Methods are “sealed”
  writable: true, configurable: false, enumerable: false
  – Why configurable: false
    A class definition establishes the shape of the prototype
  – Why writable: true
    Permits dynamic patching of method definitions

• Implications for prototype properties
  – *Can* overwrite class defined prototype methods
  – Can’t delete them
  – Can’t change method attributes, except -> writable: false
  – Can’t change method to/from an accessor
  – Can’t overwrite class defined prototype accessors

No, they will be configurable

• Methods are not constructors
  ```javascript
  class C { m() {} }
  new C.prototype.m; //type error
  ```

• Get/Set accessor functions are constructors
  ```javascript
  class C { get g() {} }
  new Object.getOwnPropertyDescriptor(C.prototype.g).get;
  //creates an obj
  ```

• Accessor properties of prototype are enumerable

Are these precedents important for internal consistency?
Class/obj lit parallels FYC

• Concise methods should be the same for both
  – Strict No, strictness determine by context
  – Non-enumerable Yes
  – Not constructable Yes
  – Same attributes? (writable: true, configurable: false?) Yes, but configurable: false

• Literal get/set functions not constructable
  – Breaking obj literal change from ES5 (low risk?) Yes

• Class accessor properties No: enumerable and configurable, just like obj lits
  – Enumerable: false, configurable: false (same as other class properties)

• Object Literal accessor properties
  – Enumerable: true, configurable: true (same as ES5)
Semantic Issues via Luke

• Should default constructor be:  
  `constructor(...args) {super(...args)}`
  – Sounds reasonable
  – Likely to be default expectation
  – Perhaps really needs to be:
    `constructor (...args) {
        if (super.constructor !== Object) super(...args);
    }

• Should constructors without an explicit super call implicitly do one.
  no
  – No
    • Calling super constructor not always desired
    • Arguments may not make sense
    • Something a lint program can do