A Declarative Alternative to toMethod

Allen Wirfs-Brock

See
The Semantics of ES6 super clashes with ad hoc extension methods

let aPusher = new Pusher();
Object.assign(aPusher, {
    push(...args) {
        console.log("aPusher mixin");
        super.push(...args);
    }
});

Does not do what the programmer intended
Wrong super binding
toMethod was problematic

```javascript
aPusher.push =
  function pusher(...args) {
    console.log("aPusher mixin");
    super.push(...args);
  }.toMethod(aPusher)
```

toMethod was unintuitive, complex, and error-prone.
toMethod was problematic

aPusher.push =
  {pusher(...args) {
    console.log("aPusher mixin");
    super.push(...args);
  }
  }
pusher.toMethod(aPusher)

toMethod was unintuitive, complex, and error-prone.
Issues with toMethod

- Exposes internal super binding mechanism
- Required copying
- Had the “deep clone” problem
- Low lever imperative manipulation of internal function state
Declarative Object literals and Class Declarations Handle super Just Fine

• Doesn’t expose internal super binding mechanism
• Doesn’t require method cloning
• Method created in the correct object context
• Automatically binds super to the correct object
• Programmers don’t need to think about or even be aware of the internals of super
Let’s make mixing in methods as simple as

```javascript
Object.assign(aPusher, {
    push(...args) {
        console.log("aPusher mixin");
        super.push(...args);
    }
});
```
Let’s make mixing in methods even simpler

```javascript
Object.assign(aPusher, mixin {
    push(...args) {
        console.log("aPusher mixin");
        super.push(...args);
    }
});
```
The `mixin` operator

- `mixin` is a contextual keyword
- first token of a high precedence postfix operator.
- The second part of a `mixin` operator has the syntax of an object literal
  - all of the normal property definition forms are allowed within it except for `__proto__`:
  - the “object literal” is a integral part of the `mixin` operator, not a separate sub-expression
- Properties created using `[[DefineOwnProperty]]` with property attributes just like an `ObjectLiteral`.
Mixin BNF

PostfixExpression :
  MixinExpression
  LeftHandSideExpression [no LineTerminator here] ++
  LeftHandSideExpression [no LineTerminator here] --

MixinExpression :
  LeftHandSideExpression
  MixinExpression [no LineTerminator here] mixin ObjectLiteral
What about abstracting a mixin?

```javascript
let PusherMixins = {
  push(...args) {
    console.log("aPusher mixin");
    super.push(...args);
  }
}

Object.assign(aPusher, PusherMixins);
```
Use functional abstraction

let PusherMixins = obj => obj mixin {
  push(...args) {
    console.log("aPusher mixin");
    super.push(...args);
  }
} 

Object.assign(aPusher, PusherMixins);
PusherMixins(aPusher);
But, some issues with class MyPusher extends Pusher {...}
But, some issues with class

class MyPusher extends Pusher {...}
MyPusher mixin {
  push(...args) {
    console.log("aPusher mixin");
    super.push(...args);
  }
};
But, some issues with class

class MyPusher extends Pusher {...}

MyPusher mixin {
  push(...args) {
    console.log("aPusher mixin");
    super.push(...args);
  }
}

Adds ‘push’ as a static method.
Probably not the programmer’s intent
Need to direct mixin properties to class prototype

class MyPusher extends Pusher {...}

MyPusher.prototype.mixin {
    push(...args) {
        console.log("aPusher mixin");
        super.push(...args);
    }
}
Another Issue

• Class methods defined in this manner using
  mixin {}
  will be enumerable, because *ObjectLiteral*
  property semantics are used
Solution

• Add a second from of the

MixinExpression :
  LeftHandSideExpression
 MixinExpression [no LineTerminator here] mixin ObjectLiteral
MixinExpression [no LineTerminator here] mixin class
  { ClassBody }
Using mixin class

class MyPusher extends Pusher {...}
MyPusher mixin class {
  push(...args) {
    console.log("aPusher mixin");
    super.push(...args);

    static get hasPushMixin() {return true}
  }
}
mixin class { ClassBody }

semantics

• Can only be applied to constructor functions
  – Does an IsConstructor test of LHS, throw if false

• Properties from the ClassBody are installed on the constructor using normal class definition rules.
  – Methods are non-enumerable

• Only restriction on ClassBody is that it must not include a constructor method.
  – Some post ES6 class features might also be restricted
Example, using mixin in a constructor

Class Foo extends Bar {
    constructor(a,b,c) {
        super();
        this :={
            a, b, c,
            _hash: a ^ B ^ c,
            ownMethod() {} 
        };
        SomeMixins(Foo);
    }
}