Decorators

Userland Extensions to ES6 Classes

Userland Classes

```
Class.new({
  init: function(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this._super();
  },

  fullName: function() {
    return `${this.firstName} ${this.lastName}`
  }
})
```

ES6 Classes

```
class {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    super();
  },

  fullName() {
    return `${this.firstName} ${this.lastName}`
  }
}
```

ship it!

Userland Classes: Knockout

```
var myViewModel = {
  personName: ko.observable('Bob'),
  personAge: ko.observable(123)
}

function AppViewModel() {
    var self = this;

    self.firstName = ko.observable('Bob');
    self.lastName = ko.observable('Smith');
    self.fullName = ko.computed(function() {
        return self.firstName() + " " + self.lastName();
    });
}
```

Userland Classes: YUI

```
var Person = Y.Base.create('person', Y.Base, [/* mixins */], { /* proto */
}, { /* static */
  ATTRS: {
    firstName: {},
    lastName : {},

    fullName: {
      readOnly: true,

      getter: function () {
        return this.get('firstName') + ' ' + this.get('lastName')
      }
    }
});
```

Userland Classes: Angular

```
module.factory('routeTemplateMonitor', ['$route', 'batchLog', '$rootScope',
  function($route, batchLog, $rootScope) {
    $rootScope.$on('$routeChangeSuccess', function() {
      batchLog($route.current ? $route.current.template : null);
    });
  }]);
```

ES6 Experiments: Angular

```
@NgDirective('[ng-bind]')
class NgBind {
  @Inject([Element])
  constructor(element) {
    this.element = element
  }

  @NgMapExpression('ng-bind')
  setText(value) {
    this.element.textContent = value;
  }
}
```

Userland Classes: Ember

```
App.Person = Ember.Object.extend({
  firstName: null,
  lastName: null,

  fullName: function() {
    return this.get('firstName') + ' ' + this.get('lastName');
  }.property('firstName', 'lastName'),

  fullNameChanged: function() {
    // deal with the change
  }.observes('fullName').on('init')
});
```

Userland Classes: Ember

```
App.Person = Ember.Object.extend({
  firstName: null,
  lastName: null,

  fullName: Em.computed(function() {
    return this.get('firstName') + ' ' + this.get('lastName');
  }, 'firstName', 'lastName'),

  fullNameChanged: Em.observes('fullName', Em.on('init', function() {
    // deal with the change
  }, 'fullName), 'init')
});
```

ES6 Experiments: Ember

```
class Person extends Ember.Object {
  - dependsOn('firstName', 'lastName')
  get fullName() {
    return this.get('firstName') + ' ' + this.get('lastName');
  }

  - on('init')
  - observes('fullName')
  fullNameChanged() {
    // deal with the change
  }
}
```

The General Problem

The General Problem: Expressions

```
{
  key: <expression>,
  ...
}

Expressions
ko.observable(...)
function() { }.on(...) or Ember.on(function() { })
{ readOnly: true, getter: function() { } }
```

Proposal in General

Goals

- Decoration of methods and accessors
- Decoration of future declarative property syntax
- Modification of the property descriptor in addition to its value
- Can work without coordination with a class or other augmentor
- Installation of metadata for use by a class or other augmentor (DI?)
- (Ideally, should be expressible in human-writable ES5)

Property Decorators

```
class PostComponent extends HTMLElement {
  - readonly
  - on('click')
  clicked() { ... }

  - observes('value')
  valueChanged() { ... }
}

Object.defineProperty(PostComponent.prototype, 'clicked',
readonly(PostComponent.prototype, 'clicked',
Object.getOwnPropertyDescriptor(PostComponent.prototype, 'clicked'));
```

- readonly

```
function readonly(prototype, name, descriptor) {
  descriptor.writable = false;
  return descriptor;
}
```

- memoize

```
class {
  - memoize('firstName', 'lastName')
  get fullName() { return `${this.firstName} ${this.lastName}` }
}

let memoized = new WeakMap();

function memoize(...dependencies) {
  return function(prototype, name, descriptor) {
    // wrap getter to memoize result and insert it into the WeakMap
    // wrap setter to invalidate the memoized result
    // in the getter wrapper:
    //   add Object.observe to invalidate the memoized result (with sync flushing)
  }
}
```

```
class {
  - dynamic(function() { ... })
  boringMethod() {}
}
```

Metadata

```
class NgBind {
  // ...
  - NgMapExpression('ng-bind')
  setText(value) {
    this.element.textContent = value;
  }
}
```

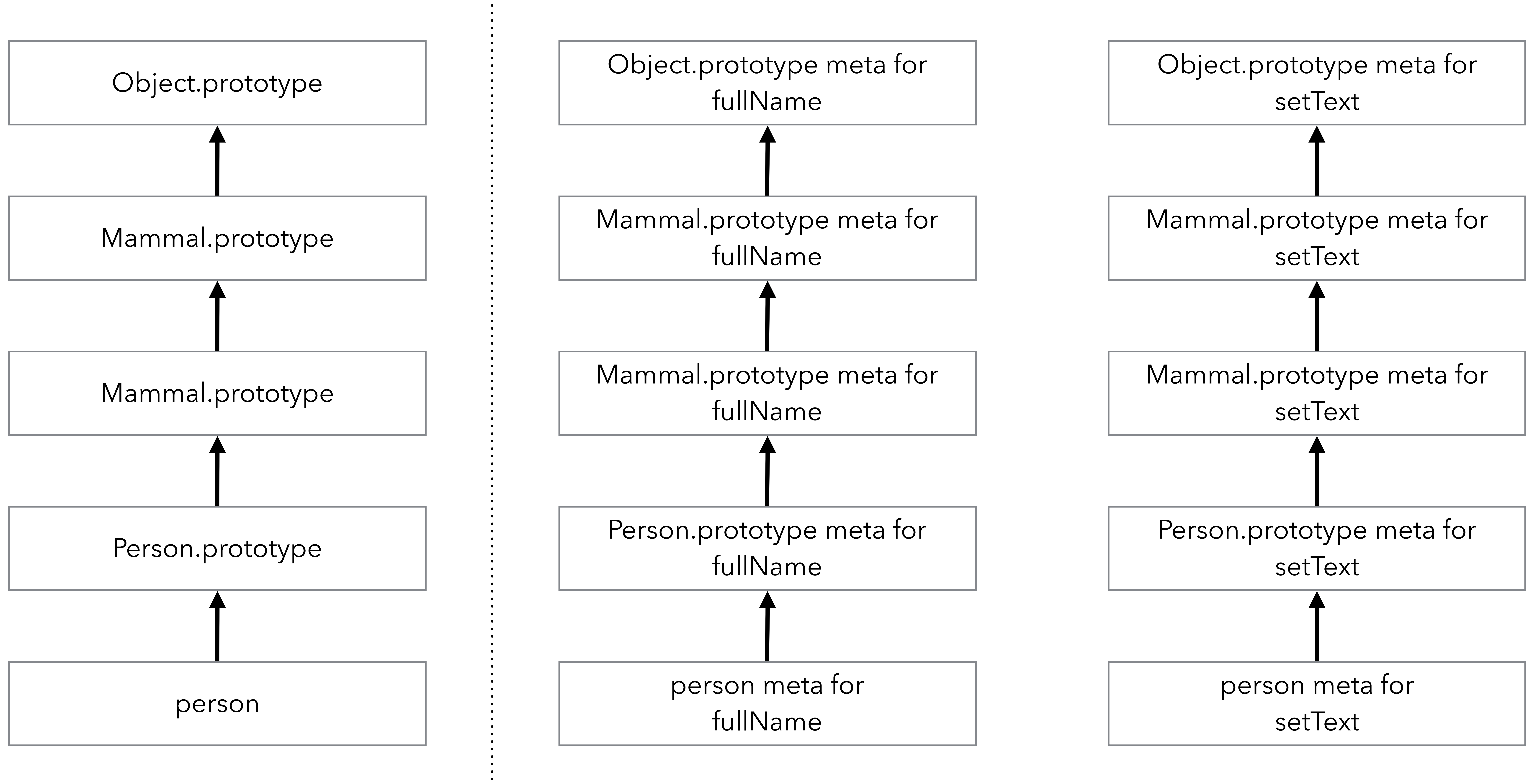
Implementation

```
var metadata = new WeakMap();

export function getMetadata(obj, propName, key) {
  return metadataFor(obj, propName)[name];
}

export function setMetadata(obj, propName, key, value) {
  metadataFor(obj, propName)[key] = value;
}

// returns a Object with slots for metadata (with parallel proto hierarchy)
function metadataFor(obj, propName) {
  // if there is a metadata dict for this object and propName, return it
  // otherwise, create a parallel hierarchy of metadata objects (to the nearest object in the
  //   prototype chain with metadata for this property) and return the metadata for this obj's
  //   propName.
}
```


Object.prototype
Mammal.prototype
Mammal.prototype
Person.prototype
person
Object.prototype meta for fullName
Mammal.prototype meta for fullName
Mammal.prototype meta for fullName
Person.prototype meta for fullName
person meta for fullName
Object.prototype meta for setText
Mammal.prototype meta for setText
Mammal.prototype meta for setText
Person.prototype meta for setText
person meta for setText

Other Experiments Possible

- NgMapExpression

```
export function NgMapExpression(value) {
  return function(prototype, name, descriptor) {
    setMetadata(prototype, name, 'NgMapExpression', true);
  }
}
```

Other Metadata

```
class Person extends Ember.Object {
  - property('firstName', 'lastName')
  get fullName() {
    return this.get('firstName') + ' ' + this.get('lastName');
  }
}
```

- property

```
export function property(...args) {
  return function(prototype, name, descriptor) {
    setMetadata(prototype, name, 'computed', args);
  }
}
```

Proposal in Detail

Static Semantics

- `MethodDefinition` and `static MethodDefinition` have a list of `DecoratorExpressions (AssignmentExpression)`

Runtime Semantics

- Extend `DefinePropertyOrThrow` to take the `DecoratorExpressions`.
- (before the current algorithm) For each *expr* in *decorator expressions*:
  - let *func* be the result of evaluating *expr*
  - let *desc* be the result of calling *func* with *obj*, *name*, and *desc*
- Continue with the algorithm, using *desc* as the descriptor for the remainder.
- NOTE: With the exception of an apparent spec bug, this spec strategy means that getters and setters will get decorated **together**. This is intentional, and falls out of the fact that we're decorating property descriptors, not functions.
- NOTE: The `ClassDefinitionEvaluation` has already set the running execution context to an appropriate lexical environment.

Future (?) Considerations

Class Decorators

```
class Articles {
  + hasMany('comments')                // "metaprogramming" style
  + belongsTo('user')
}

class NgBind {
  + NgDirective('[ng-bind]')           // "attribute" style

  - NgMapExpression('ng-bind')
  setText(value) {
    this.element.textContent = value;
  }
}
```

Why Not Above the Class?

- It's a totally different kind of thing (**property descriptor** decorator vs. **class** decorator)
- Class Expressions make stacking decorators awkward
- When using class decorators to generate properties or methods, grouping them together with other class elements is clearer.

Awkward

```
register('articles',
  @hasMany('comments')
  @belongsTo('user')
  @attr('title')
  @attr('author')
  @attr('body')
  class Articles {
    constructor() {
      // ...
    }

    @property('title', 'author')
    get byline() {
      return `${this.title} by ${this.author}`
    }
  }
)
```

More Like Custom Syntax

```
register('articles',
 class Articles {
   + hasMany('comments')
   + belongsTo('user')
   + attr('title')
   + attr('author')
   + attr('body')

   constructor() {
     // ...
   }

    - property('title', 'author')
    get byline() {
      return `${this.title} by ${this.author}`
    }
  }
)
```

Syntax Options

```
class Articles {
  - on('click')    - readonly   // my choice because of class decorators

  @on('click')     @readonly        !on('click')      !readonly
  [on('click')]    [readonly]       #on('click')      #readonly
  <on('click')>    <readonly>       %on('click')      %readonly
 #[on('click')]   #[readonly]       |on('click')      |readonly
  |on('click')|    |readonly|       &on('click')      &readonly

  // even if we don't put class decorators inside the class body, we may
  // still not want to use the same sigil for property decorators and
  // class (or function?) decorators
}
```

Attributes?

- Work well with a superclass that can understand them or a global augmentor that uses the attributes to implement decorators
- Work poorly for attributes that need to imperatively manipulate descriptor metadata
- Userland metadata can be implemented easily on top of decorators, so decorators are strictly more powerful
- We could consider supporting the attribute pattern through additional metadata in the property descriptor (or other options)

Decorating the `constructor`?

- Default position: No, because the constructor is actually not part of the same algorithm, but open to arguments.
- Question: Are you decorating the property descriptor on the prototype?
- Question: What happens if you try to replace the constructor function? Does it become constructor we're building via `class`?
- No matter what, it seems like we would have to special-case decorating the constructor if we wanted to support it.