Harmonious Classes

There are two previous strawmen proposals to add some form of class syntax to Harmony: Allen Wirfs-Brock's Class Declarations Based on Object Initialisers and an an earlier version of Mark S. Miller's Classes with Trait Composition. On 5/10/2011, Mark, Allen, Peter Hallam and Bob Nystrom (the author) spent a day working towards unifying them. This document is my best attempt to describe results of that work. It is intentionally informal.

Eventually, the material here will merge with the real proposal at Classes with Trait Composition which is being updated with the results of this meeting and further discussion with Brendan Eich, Dave Herman, and Allen.

Motivation

Today's Javascript already has excellent features for defining abstractions for *kinds of things.* The trinity of constructor functions, prototypes, and instances are more than adequate for solving the problems that classes solve in other languages. The intent of a Harmony class syntax is *not* to change those semantics. Instead, it's to provide a *terse* and *declarative* surface for those semantics so that programmer *intent* is shown instead of the *underlying imperative machinery*.

For example, here's a sample from three.js (simplified and modified slightly). For each line, I've annotated what the intent behind that code is.

```
// define a new type SkinnedMesh and a constructor for it
function SkinnedMesh(geometry, materials) {
  // call the superclass constructor
  THREE.Mesh.call(this, geometry, materials);

  // initialize instance fields
  this.identityMatrix = new THREE.Matrix4();
  this.bones = [];
  this.boneMatrices = [];
  ...
};

// inherit behavior from Mesh
SkinnedMesh.prototype = Object.create(THREE.Mesh.prototype);
SkinnedMesh.prototype.constructor = SkinnedMesh;

// define an overridden update() method
```

```
SkinnedMesh.prototype.update = function(camera) {
  ...
  // call base version of same method
  THREE.Mesh.prototype.update.call(this);
};
```

While we have plenty we hope to accomplish with classes including better abstraction, modularity, encapsulation and security; an important first step is to improve the lives of everyday Javascripters by making the above code read like the annotations above it.

With Classes

Modulo relatively minor syntactic variation, the above code would look like the following using our class syntax:

```
class SkinnedMesh extends THREE.Mesh {
  constructor(geometry, materials) {
    super(geometry, materials);

    this.identityMatrix = new THREE.Matrix4();
    this.bones = [];
    this.boneMatrices = [];
    ...
  }

  update(camera) {
    ...
    super.update();
  }
}
```

This simple example doesn't show very much, so we'll go into a slightly more involved one, but first, the underlying reasoning that led us to it.

Existing Abstractions

Javascript has two main syntactic facilities for defining abstractions: *object literals* and *functions*. (Informally, "two kinds of things that go in curlies".) Objects are *declarative* and *informational*. Functions are *imperative* and *behavioral.* The question with classes is, "do we build on one of those abstractions and if so, which one?" Adding an entirely new kind of abstraction adds weight to the language that would ideally be avoided. We serve our users well by getting as much mileage as possible out of existing concepts. Both of the two extant proposals do so by taking

one of those abstractions as a starting point. Mark's earlier proposal builds classes starting from a function body, and Allen starts from an object literal.

Our consensus proposal addresses this religious disagreement by using both: an object literal-like form for the class body itself, and a function for the constructor. This lines up nicely with the needs of each. A class body is mostly a *static declarative description* of a collection of members. A class *is* an object. A constructor, meanwhile, is an *imperative initialization process*.

The challenge we faced was that in both initial proposals, the constructor body itself also had a declarative component: it was used to define the members of a new instance of the class. Our solution is to move that declarative component out of constructors completely. Instance members are *declared* in the class body, but *initialized* in the constructor.

The Proposal

```
class Monster {...}
```

A class is a *PrimaryExpression* which defines four objects and their properties: a constructor function, a prototype, a new instance, and a private record bound to the new instance. The curly body of the class contains a collection of member definitions, each of which is one of:

Constructor

```
constructor(name, health) {
  this.name = name;
  private(this).health = health;
}
```

The constructor name followed by an argument list and a body defines the body of the class's constructor function. It is used to initialize new instances of the class, and will be invoked in response to a call to new on the class. A class may only define one constructor body.

Method

```
attack(target) {
  log('The monster attacks ' + target);
}
```

An identifier followed by an argument list and body defines a method. A "method" here is simply a function property on some object.

Getter

```
get isAlive() {
  return this.health > 0;
```

```
}
```

The contextual keyword get followed by an identifier and a curly body defines a getter in the same way that get defines one in an object literal.

Setter

```
set health(value) {
  if (value < 0) throw new Error('Health must be non-negative.')
  private(this).health = value
}
```

Likewise, set can be used to define setters.

Constant

```
const attackMessage = 'The {0} hits {1}!';
```

The keyword const followed by an identifier and an initializer declares a constant field.

Field

```
numAttacks = 0;
```

An identifier followed by = and an expression declares a field and initializes it to the value of that expression. Fields defined on the constructor and prototype objects have their initializers evaluated at class definition time.

Member Modifiers

Since a class body defines members on four objects, syntax is needed to distinguish them. A member with no qualifier defines a property on the class's *prototype.* The above examples all define members on the prototype.

```
static allMonsters = [];
```

A member prefixed by static defines the property on the constructor object itself. The above property could then be accessed by Monster.allMonsters.

```
public name;
```

A member name prefixed by public declares a property on new instances of the class. Note that simply *declaring* an instance field without an initializer has no effect: it's *initializing* it in the body of the constructor that gives it a value.

```
private health;
```

A member name prefixed by private declares a property on a private record associated with each instance of the class. Private properties can be initialized in the constructor like so:

```
constructor(name, health) {
  this.name = name;
  private(this).health = health;
}
```

Refinements

What we have so far is the core of the proposal. A class body defines the members for all four key objects, using static, public, private, and as prefixes to distinguish them. The constructor body is used for *initialization* but not *definition* of new instances. This section adds refinements to that, any of which can be omitted or changed without breaking the main idea.

Inheritance

```
class Monster extends Game.Dungeon.Entity {}
```

To provide a declarative syntax for wiring up the prototype chain, an extends clause can be added before the class body. Following the keyword is a *MemberExpression* that evaluates to the object whose .prototype property will be used as the parent object for the class's prototype object.

Given an inheritance relationship, chaining to call the base class's constructor is necessary and access to base versions of overridden methods is useful. Currently, that's unpleasant in JS:

```
constructor(x, y, name, health) {
  Game.Dungeon.Entity.call(this, x, y);
}
```

To remedy this, super can be used:

```
constructor(x, y, name, health) {
  super(x, y);
}
```

Within the body of a constructor, invoking super chains to the base class constructor. Likewise, base class methods can be called:

```
update(time) {
```

```
  super.update(time);
}
```

In addition to being shorter and simpler, it avoids mentioning the base class by name at callsites, which simplifies refactoring class hierarchies.

Auto-initialized Properties

Much of the work a constructor does is simply copying an argument into an identically-named property:

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
```

Here, we've had to stutter x and y each three times. This gets worse in our class proposal if the user wants to explicitly declare those properties:

```
class Point {
  public x, y;

  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
}
```

Now each is stuttered *four* times. To mitigate that, an argument to a constructor can be prefixed with this.. Doing so automatically copies it to a like-named property on the new instance.

```
class Point {
  public x, y;
  constructor(this.x, this.y) {}
}
```

If the user doesn't want to explicitly declare the instance fields, it can be simplified to:

```
class Point {
  constructor(this.x, this.y) {}
}
```

Rationale

While not exhaustive, this covers some of the rationale for why we chose the design decisions

made here.

Why use extends instead of : for inheritance?

Several reasons:

1. It's been reserved for this use.
2. It's semantics are close enough to Java's use of the term to be pedagogically helpful.
3. It avoids overloading : with yet another meaning (it's already property separator and label definer).
4. It provides room for expansion. For example, we could define:

    class Monster prototype Entity {}

    for specifying the direct prototype object to have Monster's prototype inherit from.

Why a *MemberExpression* for the base class?

Mainly so that you can inherit from namespaced objects like Game.Dungeon.Entity.

Why might we want to use var to define fields, rather than simply the absence of an annotation?

Using some keyword makes it clear that a class or this modifier is modifying something. If properties on the prototype were declared using:

    health = 123

That implies a constructor property would be:

    static maxHealth = 234

And then it reads as if static were a noun here and not an adjective. Using var makes it clear that static is modifying something:

    static var maxHealth = 234

Why no lexical scoping for the class body?

One feature this proposal loses over one of the earlier proposals is the ability to use the class body itself as a lexical scope. Local variables can be declared within it that are visible to class members, but hidden from the outside world.

The problem is that allowing arbitrary statements within a class body strongly curtails the syntax available to define class members. For example, the proposal here uses const field = value; to define constant fields on the prototype, but that already has meaning as a declaration. To avoid that, we would have to add more syntax to the language while not stepping on things that are already statements and declarations.

By allowing *only* member definitions within the class body, we have a clean slate to design a terse syntax for the things users need to express. This also gives us room for future expansion if we want to add other kinds of contents to a class (such as mixins).

Meanwhile, if you *do* want a lexical closure that the class members can access, you're always free to just nest the entire class definition inside its own scope.

- Bob Nystrom (rnystrom@google.com)
(with edits by Mark S. Miller)