Ecma/TC39/2010/014

Traits

Tom Van Cutsem
Mark S. Miller

Traits in a nutshell

- An alternative to mixins & multiple inheritance
- Unit of reuse: a trait *provides* and *requires* a set of methods
- Less 'fragile' composition: name clashes lead to conflicts
- Conflicts resolved by *aliasing* or *excluding* method names
- Trait composition is commutative & associative: composition order becomes irrelevant
- "invented" in Smalltalk (~2003), adoption in Perl, Fortress, ...

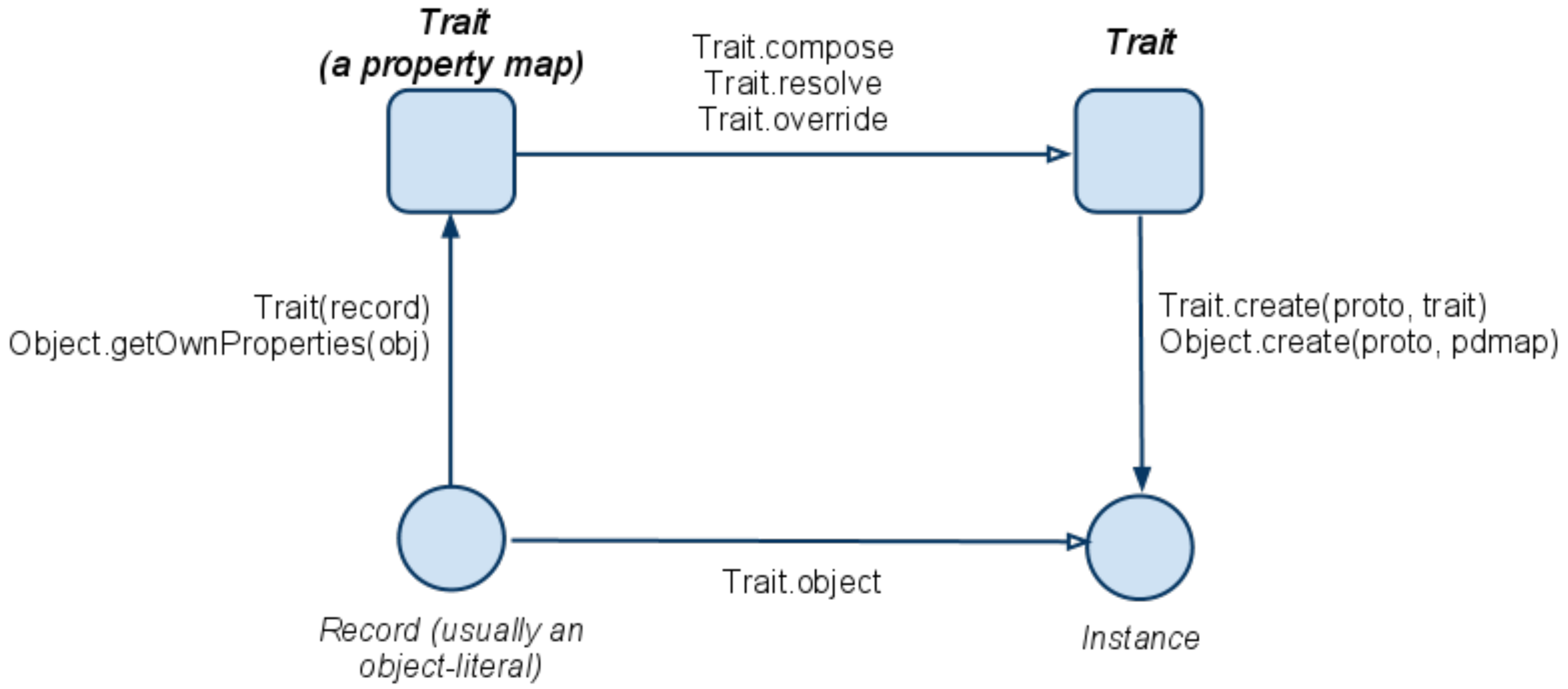
Example

```
var EnumerableTrait = Trait({
  each: Trait.required,
  map: function(fun) { var r = []; this.each(function (e) { r.push(fun(e)); }); return r; },
  inject: function(init, accum) { var r = init; this.each(function (e) { r = accum(r,e); }); return r; },
  ...
});

function Range(from, to) {
  return Trait.create(
    Object.prototype,
    Trait.compose(
      EnumerableTrait,
      Trait({
        each: function(fun) { for (var i = from; i < to; i++) { fun(i); } }
      })));
}

var r = Range(0,5);
r.inject(0,function(a,b){return a+b;}); // 10
```

traits.js API



Traits as property descriptor maps

```
var T = Trait({
    a: Trait.required,
    b: function() { ... this.a ... },
    c: 42
});
```

T =

```
{ 'a' : {
    value: undefined,
    required: true,
    enumerable: false
  },
  'b' : {
    value: function() { ... this.a ... },
    method: true
  },
  'c' : {
    value: 42
  } }
```

```
var o = Trait.create(
  Object.prototype,
  Trait.compose(T, Trait({ a: 0 })));
```

o ~

```
Object.freeze({
  a: 0,
  b: freezeAndBind(function() { ... this.a ... }, o),
  c: 42
});
```

Composition and conflict resolution

```
var T1 = Trait({ a: 0, b: 1});
var T2 = Trait({ a: 1, c: 2});

var Tc = Trait.compose(T1,T2);
```

```
Tc =  { 'a' : {
          get: function() { throw ...; },
          set: function(v) { throw ...; },
          conflict: true
        },
        'b' : { value: 1 },
        'c' : { value: 2 } }
```

```
var Tr = Trait.compose(
           T1,
           Trait.resolve({ a: 'd' }, T2);
```

```
Tr =  { 'a' : { value: 0 },
        'b' : { value: 1 },
        'c' : { value: 2 },
        'd' : { value: 1 } }
```

```
var Te = Trait.compose(
            T1,
            Trait.resolve({ a: undefined }, T2);
```

```
Te =  { 'a' : { value: 0},
        'b' : { value: 1 },
        'c' : { value: 2 } }
```

Optimization

Sharing structure between multiple instances of Trait.create requires support from the runtime:

```
function makeT(x) {
  return Trait.object({
    a: 0,
    m: function() { return this.a + x }
  });
}
var o1 = makeT(1);
var o2 = makeT(2);
```

```
function makeT(x) {
  return Trait.create(Object.prototype, {
     a: { value: 0 },
     m: { value: function() { return this.a + x; },
            method: true }
  });
}
```

```
function makeT(x) {
  return Object.freeze(
    Object.create(Object.prototype, {
      a: { value: 0 },
      m: { value: freezeAndBind(function() { return this.a + x; }, self) }
    }));
}
```

Straightforward method sharing between instances prevented by:

- binding 'this' to instance
- closure over lexical env of instance

Going forward

- Traits as ES5 property descriptor maps
- Can be stateful => no need for classes in addition to traits
- Object.create generates flexible objects
- Trait.create generates defensible objects
- No new syntax required
- But syntax helps distinguish optimizable patterns