Ecma/TC39/2011/015

[[strawman: classes_with_trait_composition]]

ES Wiki

Trace: » classes_with_trait_composition

Table of Contents

- Classes with Trait Composition
  - Class Declarations and Expressions
  - Class Elements
  - Inheritance
- Graduated Examples
  - Adding public declaration shorthands
  - Adding class-private instance variables
    - Helper: SoftField___
  - Adding single inheritance
    - Helpers: TOverride___, MakeClass___, and MakeFixedPoint___
    - Semi-Static Rejection Rules: Simple Inheritance
  - Adding prototype inheritance instead
    - Semi-static Rejection Rules: prototype inheritance
  - Adding abstract classes and required members
    - Helpers: TRequired___, MakeAbstractClass___
    - Semi-Static Rejection Rules: Unresolved required members
  - Adding soft-bound methods
    - Semi-static Rejection Rules: soft-bound method
  - Adding multiple inheritance
    - Helper: TCompose___
    - Semi-static Rejection Rules: Unresolved conflicts
  - Adding trait renaming operations
  - Adding extension methods
    - Semi-static Rejection Rules: extension methods
- See

Classes with Trait Composition

Those who learn best by example may wish to first skip to graduated_examples and then return here afterwards. There are plenty of forward references either way, so be prepared to make two passes before everything fits together.

This strawman merges the best of syntax for efficient traits and classes as sugar. However, the explanation presented here does not depend on those earlier strawmen. It is essentially the classes as sugar strawman enhanced with `extends` and `abstract` classes, for inheritance. When a class inherits from multiple base classes, those multiple bases are combined according to trait composition rules. The derived class is combined with the composed base classes by trait override rules.

This strawman should not be considered all or nothing. As with Allen's object initialiser extensions, we can treat this as an ala carte menu, where some subsets can be accepted without accepting the whole package. Specifically, the following subsettings are sensible.

- By allowing only a single `extends` clause in a class body and no Proto clause, this strawman becomes equivalent to conventional single inheritance.
- By allowing only a Proto clause in a class declaration and no `extends` clauses in the body, this strawman again becomes equivalent to conventional single inheritance. But now the inheritance chain is also mapped onto JavaScript's prototype inheritance chain.
- By not allowing any `extends` or Proto clauses at all, this strawman is similar to classes as sugar and to obj initialiser constructors.
- The support for class-private instance variables could be omitted without any effect on the rest of the strawman.
- The support for `super` could be omitted, or could be provided only in the single inheritance case.
- The support for `self` as a bound variable distinct from `this` could be omitted. We enumerate several more options for self-reference below.
- If `self` is omitted in favor of `this`, we could soft bind methods to their instance or not.
-

If super is omitted, the Renamings production could be enhanced to provide similar functionality.

- The Renamings productions could be omitted or altered without much effect on the rest of this strawman. Since this issue has subtle interactions with possible future type or guards strawmen, we leave this issue open for those strawman to pin down.
- Extension methods could be omitted without effect on the rest of this strawman.

Even though we encourage such ala carte consideration, we present the full package below, to make clear that if we accept a subset now, this subset would not be a dead end – it could grow to regain some of the options initially omitted. We then present graduated examples, starting with the minimal proposal and then showing the impact of each successive ala carte extension.

Class Declarations and Expressions

We extend the Declaration production from block scoped bindings to accept a ClassDeclaration. We also extend MemberExpression to accept a ClassExpression. These expand into a FunctionDeclaration and FunctionExpression, respectively.

```
Declaration :
    LetDeclaration
    ConstDeclaration
    FunctionDeclaration
    ClassDeclaration
ClassDeclaration :       // by analogy to FunctionDeclaration
    ClassAdjective? class Identifier ( FormalParameterList? ) Proto? { ClassBody }
ExpressionStatement :
    [lookahead not-in { "{", "function", "const", "class" }] Expression ";"
MemberExpression : ... // "..." means members defined elsewhere
    ClassExpression
ClassExpression :        // by analogy to FunctionExpression
    ClassAdjective? class Identifier? ( FormalParameterList? ) Proto? { ClassBody }
ClassAdjective :
    abstract
ClassBody :             // by analogy to FunctionBody
    ClassElement*
```

A class serves two purposes: to make the instances it describes, and to contribute descriptions to be composed by other derived classes. Abstract classes cannot make instances, and the descriptions they provide can be partial and conflicted. Concrete (non-abstract) classes must describe complete non-conflicted instances.

Since a class is just sugar for a function, its syntax and scoping are analogous to that for functions. Just as an ExpressionStatement cannot begin with function or const, it would also not be able to begin with class. The identifiers "abstract" is not reserved by ES5/strict, and so we are not proposing it be a keyword. Rather, we propose class adjectives be contextually reserved words, recognized as special only when juxtaposed with "class". Someone please advise whether the lookahead not-in rule above should be adjusted somehow to reflect this.

Class Elements

The remainder of the body of a class consists of statements, declarations, annotated declarations, and extends clauses. Declarations annotated by public are gathered together into a (potentially partial) description of same-named properties contributed towards composing the public API of the object to be created. (This is purposely analogous to the mechanics of export in so-called simple_modules.)

```
ClassElement : // by analogy to SourceElement
    Statement // but not ReturnStatement
    Declaration
    PrivateDeclaration
    PublicDeclaration
    SuperElement
PrivateDeclaration :
    private Declaration
    private Identifier = Expression ;
    private Identifier ( FormalParameterList? ) { FunctionBody }
PublicDeclaration :
    public Declaration
    public Identifier = Expression ;
    public Identifier ( FormalParameterList? ) { FunctionBody }
    require public Identifier ; // required data
    public require Identifier ;
    require public Identifier ( FormalParameterList? ) ; // required method
    public require Identifier ( FormalParameterList? ) ;
MemberExpression : ... // "..." means members defined elsewhere
    private ( AssignmentExpression )
```

The body of a class resembles a function body except

- It also lists those classes, if any, that this class extends. See SuperElement below.
- It must not contain a `return` statement. Rather, it contains `public` declarations to populate the instance it makes and returns (or the description of an instance it provides to derived classes).
- It contains `private` declarations, in order to declare class-private instance variables.
- Within the ClassElements, the keyword `super` can be used on the left of a dot as if it is an expression, to access members from the composition of the classes that this class extends.
- *Optional*: The variable name `self` is in scope, bound to the identity of the object to be instantiated, but guarded by an *initialization barrier* that rejects operations on the object until it is initialized. We mark this feature *optional* since the co-existence of `self` and `this` in one language will be confusing and hard to explain. Alternatives:
  - Simply do not bring any reliable self-reference in scope, leaving the programmer only with the current `this` with all its problems.
  - Bind `this` to the value that would have been bound to `self`.
  - Soft-bind `this` in methods only.

The MemberExpression production above using `private` is used to access the class-private instance variables of other

objects allegedly of the same class. The `private` keyword can also be used to form an expression for obtaining the private facet of another instance of the same class.

Inheritance

If a class extends no other classes, then this strawman is equivalent to classes as sugar. If a class extends just one base class, whether with an `extends` or a Proto clause, then this strawman is equivalent to conventional single inheritance, including the bindings of `self` and `super`. If a class extends multiple base classes, then this strawman uses trait composition among these sibling base classes, and it uses trait override between the composition of these siblings and the Proto base class.

```
Proto :
    : ClassName Arguments
SuperElement :
    extends ClassName Arguments Renamings? ;
ClassName :
    Identifier
Renamings :
    Renaming
    Renamings , Renaming
Renaming
    with Identifier as Identifier
    without Identifier
MemberExpression : ... // "..." means members defined elsewhere
    super . IdentifierName
```

The above `extends` syntax serves two purposes: It both declares the direct inheritance relationship among classes, and it provides the *constructor chaining* needed for all the contributing classes to be able to initialize their respective instance states. *Optional*: The Renamings are used to resolve conflicts or suppress members when inheriting from multiple base classes.

The meaning of the optional Proto clause is similar to that of the `extends` clause, except that it maps inheritance directly onto JavaScript's prototype chain, smoothing co-existence with legacy code. For new programmers working with new code that never say ".prototype" and are blissfully ignorant of the existence of prototype objects – thinking only in classes – the two have an almost identical meaning. The only salient differences are:

- A class can have a most one Proto clause.
- A Proto clause cannot be modified with Renamings
- When A class has both a Proto clause and `extends` clauses, then the ingredients contributed by the Proto clause have lower priority that the ingredients contributed by the `extends` clauses.

For those familiar with systems which mix single inheritance classes with multiple inheritance traits or mixins, the Proto chain is like the class inheritance chain and the `extends` tree is like trait or mixin inheritance – except that there are fewer semantic differences between the two.

Graduated Examples

The minimal coherent subset of this strawman leaves out everything listed as optional in the introduction above. What's left is

```
Declaration :
    LetDeclaration
```

```
        ConstDeclaration
        FunctionDeclaration
        ClassDeclaration
    ClassDeclaration :      // by analogy to FunctionDeclaration
        class Identifier ( FormalParameterList? ) { ClassBody }
    ExpressionStatement :
        [lookahead not-in { "{", "function", "const", "class" }] Expression ";"
    MemberExpression : ... // "..." means members defined elsewhere
        ClassExpression
    ClassExpression :       // by analogy to FunctionExpression
        class Identifier? ( FormalParameterList? ) { ClassBody }
    ClassBody :            // by analogy to FunctionBody
        ClassElement*

    ClassElement : // by analogy to SourceElement
        Statement // but not ReturnStatement
        Declaration
        PublicDeclaration
    PublicDeclaration :
        public Declaration
```

In this minimal subset, code such as

```
class Point(x, y) {
  const yy = y+y;
  public const getX() { return x; }
  public const y = y;
  public let size = x+yy; // don't worry about nonsense math
}
```

is observably equivalent to (but expected to be more efficient than) the following code.

In such explanatory expansions, lower case variable names ending in triple underbar (___) merely represent some variable name guaranteed not to conflict with any user-written variable name. Upper case names ending in triple underbar (___) represent internal helper functions or methods. `OFreeze___` is the original value of `Object.freeze`. `OCreate___` is the original value of `Object.create`.

```
const Point(x, y) {
  const yy = y+y;
  const getX() { return x; }
  const y = y; // See below
  let size = x+yy;
  return OFreeze___(OCreate___(Point.prototype, {
    getX: {value: getX},
    y: {value: y, enumerable: true},
    size: {get: const(){return size;},
           set: const(newSize){size = newSize;},
           enumerable: true}
  }));
}
```

A class declaration is essentially the constructor function for initializing instances of that class. Everytime the constructor function is called, with or without `new`, it will run the code within its body as a regular function would, ignoring the `public` keyword. Then, after executing the last statement within its body, it gathers all the publicly declared variables into like-named properties of the instance it will create and return.

- Public `const` or `let` variable declarations are taken to define data fields, and are therefore enumerable.
- Public `function` or `const` function declarations are taken to define methods, and are therefore not enumerable.
- `const` declarations define an unassignable variable, whose value can therefore not change after initialization. Thus, the corresponding property is simply a frozen data property holding the same value.
- `let` variable declarations and `function` function declarations define assignable variables, whose value can change at runtime. The corresponding property tracks the state of this lexical variable, and is therefore an accessor property. If either is assigned, both now present the new value.

By prohibiting `return` within a class body, we can associate hidden state within a class accurately describing the shape of the instances it makes. For each instance of a given class, we know what public properties it will have. For each public property of the instance, we know whether it is

- an enumerable data property,
- an enumerable acccessor property – in which case we also know what variable it captures from its scope,
- a non-enumerable method property – in which case we know the code of the method and what variables it captures from its scope,
- (on an abstract class as defined below) a required property,
- (on a multiply inheriting abstract class) a conflicted property,

We also know, per class, the shape of the hidden state of its instance – each contains precisely the variables captured by the accessors and methods above. These accessors and methods can therefore be stored in a per-class vtable and obtain their scope, when invoked, from their instance. This per-class shape knowledge is also used in the early rejection rules below.

Note the funny looking `const y = y;`. This depends on a new "let*"-flavored proposal (from a recent TC39 meeting, written down where?), independent of this strawman, for the scoping of `const` and `let` declarations. Like C++ and Java local variable declarations, their scope begin at the point of declaration and covers the remainder of the enclosing block excluding their own initialization expression. If this does not become the agreed meaning of `const` and `let`, the above example becomes invalid, but the remainder of this strawman is not much affected.

Adding public declaration shorthands

```
PublicDeclaration : ...
    public Identifier = Expression ;
    public Identifier ( FormalParameterList? ) { FunctionBody }
```

Using our public declaration shorthand productions, our example could have been written more compactly as:

```
class Point(x, y) {
  const yy = y+y;
  public getX() { return x; }
  public y = y;
  public let size = x+yy;
}
```

For methods, a `const` function is clearly the desired meaning for the shorthand. For variables, it is not so clear. Other possibilities are

- an implicit `let` rather than `const`, or
- an implicit `let` on the variable declaration, but define the corresponding accessor property with a getter but no setter. This makes the lexical variable read-write within the encapsulation boundary of the object but read-only to clients of the object.

Adding class-private instance variables

```
ClassElement : ...
    PrivateDeclaration
PrivateDeclaration :
    private Declaration
    private Identifier = Expression ;
    private Identifier ( FormalParameterList? ) { FunctionBody }
MemberExpression : ... // "..." means members defined elsewhere
    private ( AssignmentExpression )
```

A nonsensical example using class-private instance variables:

```
class Point(x, y) {
  private size = x + y;
  public sum(otherPt) {
    return size + private(otherPt).size;
  }
}
```

Without a `public` or `private` annotation, the `size` variable would be encapsulated within each individual instance, as in Smalltalk, E, or the Crockford objects-as-closures pattern. With the `private` annotation, `size` is instead encapsulated within the Point class, as in Java or C++. Any instance of this Point class can see the `size` of any other instance of the same Point class.

The precise semantics of this can be understood as equivalent to (but expected to be faster than) the following code.

```
const Point = (const(){
  const private___ = SoftField___();
  return const(x, y) {
    const size = x + y;
    const sum(otherPt) {
      return size + private___.get(otherPt).size;
```

```
      }
      const self___ = OFreeze___(OCreate___(Point.prototype, {
        sum: {value: sum}
      }));
      private___.set(self___, OFreeze___(OCreate___(Object.prototype, {
        size: {value: size, enumerable: true}
      }));
      return self___;
    };
  })();
```

If we decide that `self` or `this` is bound to the instantiated object, then it could be rewritten to the `self___` introduced above. If so, then expansions without `private` would still need `self___` but not `private___`, and thus not need the anonymous outer function definition and call.

Helper: SoftField___

`SoftField___` is the original value of `SoftField` from inherited_explicit_soft_fields. Note that an actual implementation should add the shape of the private record to its description of the shape of its instances, and then store this private record in hidden state within its instances.

Adding single inheritance

```
  ClassElement : ... // "..." means members defined elsewhere
      SuperElement
  SuperElement :
      extends ClassName Arguments ;
  ClassName :
      Identifier
  MemberExpression : ...
      super . IdentifierName
```

Brings us to our first slightly sensible example. From here on, we assume that `self` is used for reliable self-reference and that `this` is undisturbed.

```
  class Point(x, y) {
    public getX() { return x; }
    public getY() { return y; }
    public toString() {
      return '<' + self.getX() + ',' + self.getY() + '>';
    }
  }
  class WobblyPoint(x, y, wobble) {
    extends Point(x, y);
    public getX() {
      return super.getX() + wobble * Math.random();
    }
  }
```

This example demonstrates how we reconstruct the traditional single inheritance meanings of `self` and `super` as it exists in other languages from Smalltalk to Java.

```
  const Point = MakeClass___(Object, // at this stage, always Object
```

```
                                    const(self, x, y) {
      const getX() { return x; }
      const getY() { return y; }
      const toString() {
        return '<' + self.getX() + ',' + self.getY() + '>';
      }
      return {
        getX: {value: x},
        getY: {value: y},
        toString: {value: toString}
      };
    });
    const WobblyPoint = MakeClass___(Object, // at this stage, always Object
                                    const(self, x, y, wobble) {
      const superIngredients___ = Point.makeIngredients(self, x, y);
      const super___ = OFreeze___(OCreate___(Point.prototype, superIngredients___));
      const getX() {
        return super___.getX() + wobble * Math.random();
      }
      return TOverride___({
        getX: {value: getX}
      }, superIngredients___);
    });
```

Our class expansion relies most of all on the helper function `MakeClass____`, explained in detail below. `MakeClass____` takes two arguments:

- `protoClass` which is always `Object` until we introduce prototype inheritance below, and
- a `makeIngredients` function, which is a transform of the function-like aspects of the class definition.

`MakeClass____` installs the `makeIngredients` function it is given as the static `makeIngredients` method of the class it creates and returns. Thus, the call in `WobblyPoint`'s expansion to `Point.makeIngredients` calls the function passed in as the second argument of the `MakeClass____` call in `Point`'s expansion.

The two most important differences between the class-as-function as originally written and the `makeIngredients` function it is transformed into are

- An extra `self` first parameter is added in addition to the class' explicit construction parameters. Likewise, the expansion of our `extends` clause inserts an additional `self` argument as part of the constructor chaining. (By analogy with the conventional `Point.call(this, x, y)` which would have appeared in a conventional `WobblyPoint` constructor.))
- Whereas calling a class returns an instance, calling a `makeIngredients` function returns a stateful trait, i.e., an enhanced property descriptor map, enabling stateful trait composition.

Given a `protoClass` and a `makeIngredients` function, `MakeClass____` provides all the remaining logic needed to create a class, which is therefore generic and reusable. `MakeClass____` creates the class constructor function which will

-

create the new self instance,

- call its own makeIngredients method with this self and the constructor arguments to get the ingredients needed to initialize this self,
- use these ingredients to initialize this self, and
- return this self.

MakeClass___ itself then

- sets this constructor's makeIngredients property to be this makeIngredients function,
- sets this constructor's prototype property to inherit from protoClass.prototype,
- freezes extraneously mutable things as needed, and
- return this constructor as the class.

Even though Point does not itself extend anything, once we introduce inheritance, Point must still be defined in terms of MakeClass___ so that other classes can inherit from it by calling its static makeIngredients method.

Helpers: TOverride___, MakeClass___, and MakeFixedPoint___

TOverride___ is defined by toverride and corresponds to Trait.override from traits.js.

The above explanatory expansion depends on our internal helper functions MakeClass___:

```
const MakeClass___(protoClass, makeIngredients) {
  function constructor___(...args) {
    const {self, initialize} = MakeFixedPoint___(constructor___.prototype);
    initialize(makeIngredients(self, ...args));
    return self;
  }
  constructor___.makeIngredients = makeIngredients;
  constructor___.prototype = OCreate___(protoClass.prototype);
  OFreeze___(constructor___.prototype);
  return OFreeze___(constructor___);
}
```

which in turn depends on MakeFixedPoint___:

```
const MakeFixedPoint___(parent) {
  const h0 = OFreeze___({
    get: const(rcvr, name) { throw new ReferenceError(...); },
    has: const() { return true; }
  });
```

```
    const h1 = Proxy.create(h0, null);
    const h2 = OCreate___(h1);
    const self = Proxy.create(h2, parent);
    return OFreeze___({
      self: self,
      initialize: const(pdmap) {
        Object.defineProperty(h2, 'fix', {
          value: const() { return pdmap; }
        });
        Object.preventExtensions(self);
      }
    });
  }
```

`MakeFixedPoint___` solves a long standing tension in the design of class inheritance mechanisms between:

- `self` isn't ready for use until it is fully initialized.
- Internal methods within the class are useful to help initialize an fresh instance during construction.
- `self` (unlike `super`) holds a first class value that can be shared with others during construction.

The three common non-solutions to this problem are:

- In C++, constructors are executed top down (from base to derived). During the execution of each constructor, the object has the vtable of the current class.
- In Java and Smalltalk, constructors are also executed top down (from base to derived), but the object immediately has its final vtable – the vtable of the concrete class being directly instantiated.
- In C# apparently constructors are executed bottom up (from derived to base), and the object immediately has its final vtable. (Is this true?)

In our semantics, although `self` is still a first class value during construction, during that time it acts as a trapping proxy that reacts to all traps by throwing a ReferenceError. During construction, it is therefore only useful for non-trapping operations like

- lexically capturing the value as `self` in methods that will run post-construction, and
- using its identity, as our previous class-private explanatory expansion does, by using `self` as a key in a weak map.

This is an initialization barrier analogous to the dynamic dead zone for `const` variables, but associated with an object rather than a variable.

Despite the enforced uselessness of `self` during initialization, all the variables and methods that this class itself would directly contribute to self are still accessible as lexical variables within the class body. These local methods can be used

to manipulate these local state variables to the extent that they access them as lexical variables. Direct lexical access *always* bypasses the vtable of the instance, going directly to the lexically captured definition irrepective of derived class overrides.

Once we have all the ingredients needed to initialize the `self`, we then turn `self` into a non-extensible regular object whose properties are those described by the accumulated ingredients. In support of high integrity, `self` only goes live after all its invariants should be in place.

Semi-Static Rejection Rules: Simple Inheritance

Class definitions can have semi-static errors, and these are not accounted for by our explanatory expansions. When a class with a semi-static error is evaluated, it throws a TypeError without further effect. Like a FunctionDeclaration, a ClassDeclaration is evaluated on entry to its immediately containing Block or Program. Thus, a semi-static error in a top-level ClassDeclaration happens as early as early errors – before any code within the Program is executed. A ClassExpression is evaluated according to where the expression appears, as with any other expression, and therefore the semi-static error would be reported then.

When no class definitions anywhere have semi-static errors, then our explanatory expansions do accurately describe their semantics. Thus, these explanatory expansions are sound conservative models of the possible behaviors of the program if we add only the proviso that any class definition, when evaluated, might or might not throw a TypeError without further effect.

The ClassName within an `extends` clause must be free within the class definition it appears in, and must evaluate to a class value, i.e., a value created by some other class definitions. Only class definitions can create class values, i.e., values which can be named in extends clauses. The extension relationship among class values may not contain cycles. The error must be reported by the class definition that would otherwise have created a cycle when evaluated.

Note that we do not require the ClassName to be *statically* resolvable to a class. Rather, it simply names a value to be looked up at runtime. Combined with the free variable rule above, this allows errors to be reported as early as they would under a static restriction without prohibiting the dynamic and generative patterns one should expect from a dynamic language. For example:

```
const makeWobblyPointClass(Point) {
  class WobblyPoint(x, y, wobble) {
    extends Point(x, y);
    public getX() {
      return super.getX() + wobble * Math.random();
    }
  }
  return WobblyPoint;
}
```

is valid code. Whether it results in a semi-static error simply cannot be determined statically without static types. However, if `makeWobblyPointClass` is called with a non-class argument, then it throws a TypeError on entry, when it evaluates the `WobblyPoint` definition. Gilad Bracha's *newspeak* language makes much use of this ability to parameterize which class a given class inherits from (citation needed).

From the implementation perspective, this free variable rule makes the hidden shape information of the superclasses available when a derived class' definition is evaluated, which can thereby be accumulated into the hidden shape information stored in the derived class.

Adding prototype inheritance instead

Here, we first explore reusing JavaScript's prototype inheritance as the sole (and therefore single) inheritance mechanism, as an alternative to the single `extends` clause explained above. We deal with their co-existence below when we consider multiple inheritance. So the productions below are in addition to the above productions in the absence of the previous single inheritance additions.

```
ClassDeclaration :
    ClassAdjective? class Identifier ( FormalParameterList? ) Proto? { ClassBody }
ClassExpression :
```

```
    ClassAdjective? class Identifier? ( FormalParameterList? ) Proto? { ClassBody }
Proto :
    : ClassName Arguments
ClassName :
    Identifier
MemberExpression : ...
    super . IdentifierName
```

```
class Point(x, y) {
  public getX() { return x; }
  public getY() { return y; }
  public toString() {
    return '<' + self.getX() + ',' + self.getY() + '>';
  }
}
class WobblyPoint(x, y, wobble) : Point(x, y) {
  public getX() {
    return super.getX() + wobble * Math.random();
  }
}
```

```
const Point = MakeClass___(Object,
                           const(self, x, y) {
  const getX() { return x; }
  const getY() { return y; }
  const toString() {
    return '<' + self.getX() + ',' + self.getY() + '>';
  }
  return {
    getX: {value: x},
    getY: {value: y},
    toString: {value: toString}
  };
});
const WobblyPoint = MakeClass___(Point, // only difference
                                 const(self, x, y, wobble) {
  const superIngredients___ = Point.makeIngredients(self, x, y);
  const super___ = OFreeze___(OCreate___(Point.prototype, superIngredients___));
  const getX() {
    return super___.getX() + wobble * Math.random();
  }
  return TOverride___({
    getX: {value: getX}
  }, superIngredients___);
});
```

The only difference above is that `Point` is also passes as the `protoClass` argument to `MakeClass___`. The only resulting difference of behavior at this stage is that `MakeClass___` initializes `WobblyPoint.prototype` to inherit from `Point.prototype` rather than `Object.prototype`.

Semi-static Rejection Rules: prototype inheritance

Same as for single inheritance.

Adding abstract classes and required members

```
ClassDeclaration :      // by analogy to FunctionDeclaration
    ClassAdjective? class Identifier ( FormalParameterList? ) { ClassBody }
ClassExpression :       // by analogy to FunctionExpression
    ClassAdjective? class Identifier? ( FormalParameterList? ) { ClassBody }
ClassAdjective : ...
    abstract

PublicDeclaration : ...
    require public Identifier ; // required data
    public require Identifier ;
    require public Identifier ( FormalParameterList? ) ; // required method
    public require Identifier ( FormalParameterList? ) ;
```

Say our Point class were instead:

```
abstract class Point(x, y) {
  public require getX();
  public getY() { return y; }
  public toString() {
    return '<' + self.getX() + ',' + self.getY() + '>';
  }
}
```

it would then expand into

```
const Point = MakeAbstractClass___(Object,
                                   const(self, x, y) {
  const getY() { return y; }
  const toString() {
    return '<' + self.getX() + ',' + self.getY() + '>';
  }
  return {
    getX: TRequired___,
    getY: {value: y},
    toString: {value: toString}
  };
});
```

An required member does not declare a local variable, but rather only creates and marks that public property as needing to be provided downstream. At this stage the difference between a public required data member and a public required method is purely documentary. Other strawmen may well leverage this syntactic distinction for other purposes.

Helpers: TRequired___, MakeAbstractClass___

`TRequired___` is defined as `{required: true}`, in order to play nice with our traits_semantics operations.

`MakeAbstractClass___` is just like `MakeClass___` except that the direct constructor behavior is only to throw a TypeError. An abstract class can contribute ingredients towards the initialization of an object, but only a concrete class can use the gathered ingredients to make and initialize an object.

```
const MakeAbstractClass___(protoClass, makeIngredients) {
  function constructor___(...args) {
```

```
      throw new TypeError(...);
    }
    constructor___.makeIngredients = makeIngredients;
    constructor___.prototype = OCreate___(protoClass.prototype);
    OFreeze___(constructor___.prototype);
    return OFreeze___(constructor___);
  }
```

Semi-Static Rejection Rules: Unresolved required members

In a class in which any member is required, whether because of a direct `require` declaration or because of an unresolved inherited required member, then the class itself must be declared `abstract`. If it is not, then a TypeError is thrown when the class definition is evaluated. `Point` above does not exhibit such an error.

If `super.identifier` names an inherited required member, as WobblyPoint's `super.getX` now does, then a TypeError is thrown when WobblyPoint's definition is evaluated.

Adding soft-bound methods

If classes use only `self` and do not traffic in `this`, then soft binding should not be relevant. If classes do make use of `this` to designate the instance of the class, then could adopt the method binding of semantics of tcreate (which corresponds to `Trait.create` from traits.js). Alternatively, we might want to adopt a tcreate modified to soft bind `this` to its methods instead. The semantics of `this` would then resemble the following objects-as-closure pattern, ignoring for the moment the impossibility of using `self___` before it's defined. (Repairing this is easy but verbose.)

```
const Point(x, y) {
  const self___ = OFreeze___(OCreate___(Point.prototype, {
    getX: {value: (const() { return x; }).softBind(self___)},
    getY: {value: (const() { return y; }).softBind(self___)}
  }));
  return self___;
}
```

However, this way of using soft binding only works for inheritance within the class system, not for non-class objects that happen to inherit from an individual point. To accommodate this case as well, we can use a getter to do the soft binding at extraction time. We first define a helper function for define such getters.

```
function makeSoftBindingDescriptor(f) {
  return { get: const() { return f.softBind(this); }}
}
//...
const Point(x, y) {
  return OFreeze___(OCreate___(Point.prototype, {
    getX: makeSoftBindingDescriptor(const() { return x; }),
    getY: makeSoftBindingDescriptor(const() { return y; })
  }));
}
```

This also has the minor benefit that, by postponing the soft binding until extraction time, we no longer need to solve the circular definition puzzle above.

```

```

Semi-static Rejection Rules: soft-bound method

TODO: I suspect there aren't any, in which case delete this subsection

Adding multiple inheritance

No new productions. Syntactically, we just lift the restriction that only one `extends` or Proto clause can appear in each class. The `superIngredients___` are now made from a `TCompose___` of the ingredients defined by each of the superclasses, and a `TOverride___` of this composition with the ingredients defined by Proto. The result of composing these extensions is a set of properties, where each is either

- a description of concrete member (data, accessor, or method),
- a required member,
- a conflicted member.

If two sibling superclasses both define non-required (i.e., concrete or conflicted) members of the same name, then the composition of these superclasses has that member in conflict. If exactly one superclass contributes a non-required member of a given name, then the composition has this non-requires member.

If a derived class itself directly defines a member for a given name, whether concrete or required, then this definition overrides whatever was inherited from the composition of superclasses.

If the composition of superclasses defines a member of a given name, then this definition overrides whatever was inherited from Proto.

```
    class D(x,y) : A(x) {
        extends B(y);
        extends C(x+y);
        public foo() { return x-y; }
    }
```

would expand to

```
  const D = MakeClass___(A,
                         const(self, x, y) {
    const superIngredients___ =
        TOverride___(TCompose___(B.makeIngredients(self, y),
                                 C.makeIngredients(self, x+y)),
                     A.makeIngredients(self, x));
    const super___ = OFreeze___(OCreate___(A.prototype, superIngredients___));
    const foo() {
      return x-y;
    }
    return TOverride___({
      foo: {value: foo}
    }, superIngredients___);
  });
```

More intuitively, if we ignore `super` issues and the distinction between instances and ingredients, the above class defines approximately:

```
  TOverride___({value: foo},                // local members first
               TCompose___(B(y), C(x+y)), // ''extends'' next, symmetrically
               A(x))                       // Proto last
```

Helper: TCompose___

TCompose___ is defined by tcompose, corresponding to Trait.compose of traits.js. The most important feature of TCompose___ is that it is associative and commutative. Commutativity ensures that the order of extends clauses within a class don't affect the result of composition. In a pattern where each class either defines local members or extends other classes but no class does both, then associativity ensures that any extends tree that composes together the same members yields the same composition, easing refactoring.

TODO: Revise MakeClass___ and MakeAbstractClass___ so that classes are defined using proxy instanceof as equivalent to function proxies that trap hasInstance, so that an instance is instanceof all of its ancestor classes, even under multiple inheritance. Although instanceof is still not a valid type or trademark check, we should nevertheless ensure that instanceof remains unsurprising in the face of multiple inheritance. Of course, we are powerless to make the prototype chain reflect multiple inheritance as well, so we live with the symmetry-breaking rule that the prototype chain follows the first extends clause of each class.

Semi-static Rejection Rules: Unresolved conflicts

An abstract class may have unresolved required or conflicted members. A concrete class cannot; instead a TypeError is thrown.

A super.identifier expression causes a semi-static error if that named member of the composition of superclasses is required or conflicted. When a class definition containing such a super expression is evaluated, a TypeError is thrown.

Adding trait renaming operations

```
SuperElement :
    extends ClassName Arguments Renamings? ;
Renamings :
    Renaming
    Renamings , Renaming
Renaming
    with Identifier as Identifier
    without Identifier
```

TODO based on tresolve which corresponds to Trait.resolve from traits.js.

By itself, the above production and the corresponding tresolve semantics is not yet adequate to replace the need for super, since the result of renaming is still accessible as a public property on the instance. By contrast, super accesses the overridden functionality without thereby making it accessible outside the abstraction. Overriding becomes a form of encapsulation. Alternatively, we could omit super and enhance the above production so that the right hand side of a with/as could be a local variable declaration, making the overridden member accessible without exporting it. This would be more expressive than super and more naturally part of a traits system.

Adding extension methods

conflict-free object extension using soft fields Shows how to use soft fields to express the semantics of extension methods using soft fields. Using our running example, we might say:

```
private getSlope; // soft field for slope method
```

```
Point.prototype.getSlope = function() { return this.getY() / this.getX(); };
//...
print(wobblyPoint.getSlope()); // looks up and applies getSlope extension above

export #.getSlope; // if you wish
```

This would expand to

```
const getSlope___ = SoftField();
getSlope.set(Point.prototype, function() { return this.getY() / this.getX(); });
//...
print(getSlope___.get(wobblyPoint).call(wobblyPoint));

export getSlope___;
```

In the context of a class system, the use of `private` in this syntax is unpleasant, as it doesn't have an intuitive (non-implementation oriented) relationship to other meanings of `private`. Worse, the above expresses the extension by explicitly mentioning prototypes, requiring the new class-based programmer to still understand the mapping of classes to prototype inheritance. A more class-oriented sugar based on WebIDL implements statements might be:

```
interface SlopyThing {
  getSlope(); // In an interface, "public" and "required" are implicit
  // other slopy method declarations
};

Point implements SlopyThing {
  public getSlope() {
    return self.getY() / self.getX();
  }
  // other slopy method implementations
}
```

Importing `SlopyThing` could then bring its members into scope. This is hazardous, since these members would then not be enumerated at the importing site.

Whether extension methods are packaged into extension interfaces or not, the important point is that the above `implements` clause is subjective. Code not importing `SlopyThing` would not see any changes, even reflectively, to the semantics of points.

Restating the example from conflict-free object extension using soft fields in these terms, we get

```
interface Cloneable {
  clone();
}

Object implements Cloneable {
  public clone() { ... }
}
Array implements Cloneable {
  public clone() {
    ...
    target[i] = this[i].clone(); // recur on clone method
    ...
  }
```

```
  }
  String implements Cloneable {
    public clone() { ... }
  }
  ...
  export Cloneable;
```

Semi-static Rejection Rules: extension methods

TODO: if none, which seems plausible, remove this section.

See

Encapsulation and Inheritance in Object-Oriented Programming Languages – classic 1986 paper by Alan Snyder

const_functions

guards

trademarks

inherited explicit soft fields

classes as sugar

How To Node article on Traits.js

Jetpack Traits

syntax for efficient traits

object initialiser extensions

Classes as Sugar thread which starts with pointers to earlier threads.

classes as inheritance sugar (not yet ready)

extension methods

strawman/classes_with_trait_composition.txt · Last modified: 2011/03/10 09:41 by markm

[[strawman:traits_semantics]]

ES Wiki

Trace: » classes_with_trait_composition » traits_semantics

Trait Semantics

Table of Contents

- Trait Semantics
  - TraitLiteral
  - TCompose
  - TOverride
  - TResolve
  - TCreate

This page describes the semantics of trait composition for the syntax for efficient traits strawman.

A trait is a "property descriptor map", represented as a set of properties. Only a property descriptor map object's own properties are treated as members of this set. The prototype of the property descriptor map is ignored. Properties are represented as `name:pd` tuples where name is the property name (a string) and pd is a property descriptor object (this corresponds to the "Property Identifier" type in ES-262 5th ed, section 8.10). Property descriptors are either plain ES5 data or accessor property descriptors, or one of the following traits-specific property descriptors: a "required" property (identifying an "abstract" property that should be present in the final trait), a "conflicting" property (identifying a name conflict during composition) or a "method" property, which identifies a data property whose function value should be treated as a "method" (with bound-`this` semantics).

```
PDMap ::= { PropertyIdentifier* }
PropertyIdentifier ::= String:PropDesc
PropDesc ::= { value: v, writable: b }
           | { get: fg, set: fs }
           | { required: true }
           | { conflict: true }
           | { value: f, writable: false, method: true }
```

The functions below are specified using a Haskell-like syntax. Property descriptor maps are represented using the syntax `{ n1:p1, ..., nk:pk }`. These property descriptor maps are treated as sets, so the ordering of the properties `n1:p1` up to `nk:pk` is irrelevant. Property descriptors on this page are assumed to have default attributes `enumerable:true` and `configurable:true`.

Metasyntactic variables used: `v` for any value, `b` for booleans, `f` for functions, `fg` for getter functions, `fs` for setter functions, `n` for property names, `p` for property descriptors, `pdm` for property descriptor maps.

TraitLiteral

The function `TraitLiteral` describes how a `TraitPartList` consisting of a series of property declarations is converted into a property descriptor map.

```
TraitLiteral :: TraitPartList -> PDMap
TraitLiteral [] = {}
TraitLiteral (part:parts) =
  add_prop (TraitLiteral parts) (to_property part)

to_property :: TraitPart -> PropertyIdentifier
to_property 'n : expr'                 = n:{ value: expr, writable: true }
to_property 'get n() { body }'         = n:{ get: const() { body }, set: undefined }
to_property 'set n(arg) { body }'      = n:{ get: undefined, set: const(arg) { body } }
to_property 'method n(args) { body }' = n:{ value: const(args) { body }, writable:
false, method: true }
to_property 'require n'                = n:{ required: true }
```

Notes:

-

we implicitly assume that all created property descriptors have additional attributes { `enumerable: true, configurable: true` }.

- See below for the definition of `add_prop`.

TCompose

`TCompose` takes an arbitrary number of property descriptor maps and returns a property descriptor map that combines all own properties of its arguments. Name clashes lead to the generation of special `conflict` properties in the resulting trait. `TCompose` is commutative: its result is independent of the ordering of its arguments.

```
TCompose :: [ PDMap ] -> PDMap
TCompose [] = {}
TCompose (pdm:pdms) =
  compose_pdmap pdm (TCompose pdms)

compose_pdmap :: PDMap -> PDMap -> PDMap
compose_pdmap pdm { } = pdm
compose_pdmap pdm { n1:p1, … , nk:pk } =
  compose_pdmap (add_prop pdm n1:p1) { n2:p2, …, nk:pk }

add_prop :: PDMap -> PropertyIdentifier -> PDMap
add_prop { n1:p1, …, nk:pk } ni:pi =
  { n1:p1, …, nk:pk, ni:pi } if not member ni { n1, …, nk }
add_prop { n1:p1, … , n:pi1, … nk:pk } n:pi2 =
  { n1:p1, …, n:(compose_pd pi1 pi2), … , nk:pk }

compose_pd :: PropDesc -> PropDesc -> PropDesc
compose_pd { value: v1, writable: bw1, method: b1 } { value: v2, writable bw2, method: b2 } =
  { value: v1, writable: bw1, method: b1 } if (identical v1 v2) and bw1 === bw2 and b1 === b2
compose_pd { value: v1, writable: bw1, method: b1 } { value: v2, writable bw2, method: b2 } =
  { conflict: true } if not (identical v1 v2) or bw1 !== bw2 or b1 !== b2
compose_pd { get: fg1, set: fs1 } { get: fg2, set: fs2 } = { get: fg1, set: fs1 } if (identical fg1 fg2) and (identical fs1 fs2)
compose_pd { get: fg1, set: fs1 } { get: fg2, set: fs2 } = { conflict: true } if not (identical fg1 fg2) or not (identical fs1 fs2)
compose_pd { get: fg, set: undefined } { get: undefined, set: fs } = { get: fg, set: fs }
compose_pd { get: undefined, set: fs } { get: fg, set: undefined } = { get: fg, set: fs }
compose_pd { value: v, writable: bw, method: b } { get: fg, set: fs } = { conflict: true }
compose_pd { get: fg, set: fs } { value: v, writable: bw, method: b } = { conflict: true }
compose_pd { required: true } p = p
compose_pd p { required: true } = p
compose_pd { conflict: true } p = { conflict: true }
compose_pd p { conflict: true } = { conflict: true }
```

Notes:

- { `value: v, writable: b, method: false` } is considered equivalent to the plain data property descriptor { `value: v, writable: b` }.
- We implicitly assume that the `enumerable` and `configurable` attributes of the above property descriptors are equal.

This is the case for property descriptors created using `TraitLiteral`. If these attributes are not equal for a pair of property descriptors, they are treated as non-equal and would generate `{ conflict: true }` if composed.

- `identical(a,b)` has the semantics of egal.

TOverride

`TOverride` takes an arbitrary number of property descriptor maps and combines them into a single property descriptor map. It automatically resolves name clashes by having the left-hand trait's property value take precedence over the right-hand trait's property value. Hence, `TOverride` is not commutative: the ordering of arguments is significant and precedence is from left to right.

```
TOverride :: [ PDMap ] -> PDMap
TOverride [] = {}
TOverride (pdm:pdms) =
  override_pdmap pdm (TOverride pdms)

override_pdmap :: PDMap -> PDMap -> PDMap
override_pdmap pdm {} = pdm
override_pdmap pdm { n1:p1, … , nk:pk } =
  override_pdmap (override_prop pdm n1:p1) { n2:p2, …, nk:pk }

override_prop :: PDMap -> PropertyIdentifier -> PDMap
override_prop { n1:p1, …, nk:pk } ni:pi = { n1:p1, …, nk:pk, ni:pi } if not member ni
{ n1, …, nk }
override_prop { n1:p1, … , n:pi1, … nk:pk } n:pi2 = { n1:p1, …, n:pi1, … , nk:pk }
```

TResolve

`TResolve` renames and excludes property names of a single argument property descriptor map.

Let Renames be a map from String to String and Exclusions be a set of Strings:

```
Renames    ::= [ String -> String ]
Exclusions ::= [ String ]

TResolve :: Renames -> Exclusions -> PDMap -> PDMap
TResolve r e pdm =
  rename r (exclude e pdm)

exclude :: Exclusions -> PDMap -> PDMap
exclude e {} = {}
exclude e { n1:p1, …, nk:pk } =
  add_prop (exclude e { n2:p2, …, nk:pk }) n1:{ required: true } if member n1 e
exclude e { n1:p1, … , nk:pk } =
  add_prop (exclude e { n2:p2, …, nk:pk }) n1:p1 if not member n1 e

rename :: Renames ->  PDMap -> PDMap
rename map {} = {}
rename map { n1:p1, …, nk:pk } =
  add_prop (rename map { n2:p2, …, nk:pk }) m:p1 if member (n1 -> m) map
rename map { n1:p1, …, nk:pk } =
  add_prop (rename map { n2:p2, …, nk:pk }) n1:p1 if not member (n1 -> m) map
```

TCreate

`TCreate` takes a prototype object and a property descriptor map and returns an "instance" of the property descriptor

map. `TCreate` validates the property descriptor map to see if it contains unsatisfied `required` arguments and unresolved `conflict` properties. If so, it fails. `TCreate` also binds and freezes all properties marked as `methods`.

```
TCreate :: Object -> PDMap -> Object
TCreate proto pdm =
  do {
   -- pardon the awkward mixture of Haskell and Javascript syntax
   obj <- Object.create(proto);
   Object.defineProperties(obj, validate obj pdm);
   return Object.freeze(obj);
  }

validate :: Object -> PDMap -> PDMap
validate obj {} = {}
validate obj { n1:p1, …, nk:pk } =
  add_prop (validate obj { n2:p2, …, nk:pk }) n1:(validate_prop obj n1:p1)

validate_prop :: Object -> PropertyIdentifier -> PropDesc
validate_prop self n:{ value: v, writable: b, method: false } = { value: v, writable: b }
validate_prop self n:{ value: v, writable: b, method: true } = { value: freezeAndBind(v,
self), writable: b  }
validate_prop self n:{ get: fg, set: fs } = { get: freezeAndBind(fg,self), set:
freezeAndBind(fs,self) }
validate_prop self n:{ required: true } = <error: required property: n> if not (n in
self)
validate_prop self n:{ required: true } = {} if (n in self)
validate_prop self n:{ conflict: true } = <error: conficting property: n>

freezeAndBind :: Function -> Object -> Function
freezeAndBind fun obj =
  Object.freeze(Function.prototype.bind.call(fun, obj))
```

[[strawman: inherited_explicit_soft_fields]]

ES Wiki

Trace: » classes_with_trait_composition » traits_semantics » inherited_explicit_soft_fields

Explicit Inherited Soft Fields

Table of Contents

- Explicit Inherited Soft Fields
  - A transposed representation
  - Should we tolerate primitive keys?
  - Can we subsume Private Names?
- See

The following derived abstraction combines the explicitness of explicit soft own fields with the visibility across inheritance chains of inherited soft fields. Below is an executable specification as a wrapper around weak maps. This strawman page suggests standardizing this derived abstraction because a primitive implementation is likely to be more efficient that the code below.

As with our previous "`EphemeronTable`", the name "`SoftField`" is only a placeholder until someone suggests an acceptable name.

```
  const SoftField() {
    const weakMap = WeakMap();
    const mascot = {}; // fresh and encapsulated, thus differs from any possible
provided value.
    return Object.freeze({
      get: const(base) {
        while (base !== null) {
          const result = weakMap.get(base);
          if (result !== undefined) {
            return result === mascot ? undefined : result;
          }
          base = Object.getPrototypeOf(base);
        }
        return undefined;
      },
      set: const(key, val) {
        weakMap.set(key, val === undefined ? mascot : val);
      },
      has: const(key) {
        return weakMap.get(key) !== undefined;
      },
      delete: const(key) {
        weakMap.set(key, undefined);
      }
    });
  }
```

A transposed representation

The following is an alternative explanation that implements the same semantics but more closely reflects expected implementation. This is no longer quite an executable specification in that it builds on a new internal property, here spelled `SoftFields___`. The safety of the following spec depends on the `SoftFields___` property not being used by any other spec beyond the following.

```
  const init___(obj) {
    if (obj !== Object(obj)) { throw new TypeError(...) }
```

```
    if (!obj.SoftFields___) {
      obj.SoftFields___ = WeakMap();
    }
  }

  const SoftField() {
    const mascot = {};
    const get(base) {
      init___(base)
      while (base !== null) {
        const result = base.SoftFields___.get(get);
        if (result !== undefined) {
          return result === mascot ? undefined : result;
        }
        base = Object.getPrototypeOf(base);
      }
      return undefined;
    }
    return Object.freeze({
      get: get,
      set: const(key, val) {
        init___(key);
        key.SoftFields___.set(get, val === undefined ? mascot : val);
      },
      has: const(key) {
        init___(key);
        return key.SoftFields___.get(get) !== undefined;
      },
      delete: const(key) {
        init___(key)
        key.SoftFields___.set(get, undefined);
      }
    });
  }
```

The overall logic is very similar, except that the underlying weak maps are now stored on the SoftField's key objects, while each SoftField itself only holds on to the fixed state of the key used to look up values in those weak maps. Even though the above algorithm still manually encodes walking the prototype chain, because this walk is now consulting a map stored within each object, two transparent performance benefits may follow:

- The optimizations already in place for normal property lookup may be more readily adapted to soft field lookup.
- The conventional portion of a GC algorithm that does not take account of weak maps will nevertheless collect that soft state that is only reachable from non-reachable objects, even in the presence of cycles between that soft state and those objects. For soft fields, the weak map portion of a GC algorithm is only needed to collect those soft fields that can no longer be "named" but are still present on objects that are reachable.

This representation parallels the implementation techniques and resulting performance benefits expected for private names but without the semantic problems (leaking via proxy traps and inability to associate soft state with frozen objects).

Regarding the GC point, when soft fields are used in patterns such as class-private instance variables, a soft field adds soft state to a set of objects, each of whom also points at that soft field itself. In that case, the soft field has a lifetime at

least as long as any of the objects it indexes. Thus, the conventional portion of GC algorithms is adequate to pick up all the resulting collectable soft state.

Should we tolerate primitive keys?

Since soft fields – unlike weak maps – look up the key's inheritance chain until it find a match, it makes sense to allow primitive data types (numbers, strings, and booleans, but still not null or undefined) to serve as keys. When used as a key, the operations above would first convert it to an object using the internal [[ToObject]] function. (Unlike `Object`, [[ToObject]] on a null or undefined throws a TypeError.) For strings, numbers, and booleans, this results in a fresh wrapper, which therefore has no soft own state. Lookup would therefore always proceed to the respective prototypes, so that, e.g., a primitive string would seem to inherit soft state from `String.prototype`, much as it currently seems to inherit properties from `String.prototype`.

Can we subsume Private Names?

Two use cases shown at private names that simple soft fields cannot provide are a certain form of polymorphism between names and strings, and so-called "weak encapsulation". (MarkM here suspends value judgements about whether we should seek to support so-called "weak encapsulation", and addresses here only how to do so, were we to agree on its desirability.) If value proxies are accepted for Harmony, then Soft fields can grow to support both these use cases without further expansion of kernel semantics, by defining a soft field as equivalent to a value proxy that overloads [], to whit:

```
const softFieldOpHandler = Object.freeze({
  // overload larg[proxy]
  rgeti: const(larg)         { return this.get(larg); },
  // overload larg[proxy] = val;
  rseti: const(larg, val)    {        this.set(larg, val); }
});
// Move the SoftField code into softFieldProto
const softFieldProto = Object.freeze({
  get:    ..., //as above, but with "this.weakMap" instead of "weakMap"
  set:    ..., //as above
  has:    ..., //as above
  delete: ...  //as above
});
const softFieldValueType = Proxy.createValueType(
    softFieldOpHandler, softFieldProto,
    "string", // bad idea, but suspending judgement
    { weakMap: object });
const SoftFieldValue() {
  return Proxy.createValue(softFieldValueType, new WeakMap());
}
const softFieldValue = SoftFieldValue();
```

(Detail: The above code doesn't quite work as is, because there's no where safe to put the mascot. By delegating to an encapsulated explicit soft own fields instead of a `WeakMap`, we can encapsulate the mascot in this extra layer. This is a detail because it effects only the apparent cost of this executable specification, not the actual cost of an implementation.)

A "weakly encapsulating" soft field, or *wesf*, can then be coded as:

```
// Move the SoftField code into softFieldProto
const wesfProto = Object.freeze({
  toString: const()         { return improbableName; },
  get:      const(key)      { return key[improbableName] },
```

```
    set:      const(key, val) {        key[improbableName] = val; },
    has:      const(key)      { return improbableName in key; },
    delete:   const(key)      { return delete key[improbableName]; }
  });
  const wesfValueType = Proxy.createValueType(
      softFieldOpHandler, wesfProto,
      "string", // bad idea, but suspending judgement
      { improbableName: string });
  const WesfValue(opt_name) {
    const name = String(opt_name) || Math.random() + '___';
    return Proxy.createValue(wesfValueType, name);
  }
  const wesfValue = WesfValue();
```

Then polymorphic code such as

```
  function foo(n) {
    return base[n];
  }
```

can be called with a softFieldValue, a wesfValue, or a string, where each provides the degree of encapsulation and collision avoidance it claims. This supports the ability to modularly refactor code between encapsulating, “weakly” encapsulating, and obviously non-encapsulating fields.

See

The thread beginning at WeakMap API questions?

Older GC discussion now obsolete but still potentially interesting.

RSS XML FEED CC LICENSED $1 DONATE
PHP POWERED W3C XHTML 1.0 W3C CSS DOKUWIKI

[[strawman: names_vs_soft_fields]]

ES Wiki

Trace: » classes_with_trait_composition » traits_semantics » inherited_explicit_soft_fields » names_vs_soft_fields

Table of Contents

- Overview
- The private declaration
- Using Private Identifiers
- Private Identifiers in Object Literals
- Private Declaration Scoping
- Private Declarations Expand to Unique Hidden Variable Names
- Accessing Private Identifiers as Soft Field Values
- Conflict-Free Object Extension Using Soft Fields
  - Crucial difference
- Enumeration and Reflection
- Soft Fields Support Encapsulation
- Interactions with other Harmony Proposals
  - Enhanced Object Literals
  - Proxies
  - Modules
- References

Overview

To better understand the differences between soft fields and private names, this page goes through all the examples from the latter (as of this writing) and explores how they'd look as translated to use soft fields instead. This translation does not imply endorsement of all elements of the names proposal as translated to soft fields, such as the proposed syntactic extensions. However, these translations do establish that these syntactic choices are orthogonal to the semantic controversy and so can be argued about separately.

Identifiers ending with triple underbar below signify unique identifiers generated by expansion that are known not to conflict with any identifiers that appear elsewhere.

The private declaration

Adapted from the private declaration

```
private secret;  //create a new soft field that is bound to the private identifier ''secret''.
private _x,_y;   //create two soft fields bound to two private identifiers
... foo.secret ...
foo.secret = val;
const obj = {secret: val, ...};
#.secret
```

expands to

```
const secret___ = SoftField();
const _x___ = SoftField(), _y___ = SoftField();
... secret___.get(foo) ...
secret___.set(foo, val);
const obj = {...}; secret___.set(obj, val);
secret___
```

Using Private Identifiers

Adapted from using private identifiers

```
function makeObj() {
   private secret;
   var obj = {};
   obj.secret = 42;  //obj has a soft field
   print(obj.secret);//42 -- accesses the soft field's value
   print(obj["secret"]); //undefined -- a soft field is not a property
```

```
    return obj;
}
var obj=makeObj();
print(obj["secret"]); //undefined -- a soft field is still not a property
print(obj.secret);    //undefined -- this statement is not in the scope of the private 
declaration so the
                      //string value "secret" is used to look up the property.  It is 
not a soft field.
```

This technique can be used to define "instance-private" properties:

```
function Thing() {
    private key;   // each invocation will use a new soft field
    this.key = "instance private value";
    this.hasKey = function(x) {
        return x.key === this.key;  //x.key should be undefined if x!==this
    };
    this.getThingKey = function(x) {
        return x.key;
    };
}
```

Instance-private instance state is better done by lexical capture

```
function Thing() {
    const key = "instance private value";
    this.hasKey = function(x) {
        return x === this;
    };
    this.getThingKey = function(x) {
        if (x === this) { return key; }
    };
}
```

Either technique produces the same external effect:

```
var thing1 = new Thing;
var thing2 = new Thing;

print("key" in thing1);        // false
print(thing2.key);             //undefined
print(thing1.hasKey(thing1)); // true
print(thing1.hasKey(thing2)); // false
```

By changing the scope of the private declaration a similar technique can be used to define "class-private" properties:

```
private key;  //the a soft field shared by all instances of Thing.
function Thing() {
    this.key = "class private value";
    this.hasKey = function(x) {
        return x.key === this.key;
```

```
    };
    this.getThingKey = function(x) {
        return x.key;
    };
}

var thing1 = new Thing;
var thing2 = new Thing;

print("key" in thing1);        // false
print(thing1.hasKey(thing1)); // true
print(thing1.hasKey(thing2)); // true
```

Private Identifiers in Object Literals

Adapted from private identifiers in object literals

```
function makeObj() {
   private secret;
   var obj = {secret: 42};
   print(obj.secret);//42 -- access the soft field's value
   print(obj["secret"]); //undefined -- a soft field is not a property
   return obj;
}
```

```
function Thing() {
    private key;
    return {
       key : "instance private value",
       hasKey : function(x) {
          return x.key === this.key;  //x.key should be undefined if x!==this
       },
       getThingKey : function(x) {
          return x.key;
       }
    };
}
```

or, preserving the same external behavior:

```
function Thing() {
    const key = "instance private value";
    return {
       hasKey : function(x) {
          return x === this;
       },
       getThingKey : function(x) {
          if (x === this) { return key; }
       }
    };
}
```

```
private key;
function Thing() {
    return {
       key : "class private value",
       hasKey : function(x) {
          return x.key === this.key;  //x.key should be undefined if x!==this
       },
       getThingKey : function(x) {
          return x.key;
       }
    };
}
```

Private Declaration Scoping

Adapted from private declaration scoping

```
function outer(obj) {
   private name;
   function inner(obj) {
      private name;
      obj.name = "inner name";
      print(obj.name);   //"inner name" because outer name declaration is shadowed
   }
   obj.name = "outer name";
   inner(obj)
   print(obj.name);      //"outer name"
}
var obj = {};
obj.name = "public name";
outer(obj);
print(obj.name);           //"public name"
```

After executing the above code, the object that was created will have one property and two associated soft fields:

Property or Fields	Value
"name"	"public name"
private $name_{outer}$	"outer name"
private $name_{inner}$	"inner name"

Private Declarations Expand to Unique Hidden Variable Names

Adapted from private declarations exist in a parallel environment

Consider the following very common idiom used in a constructor declaration:

```
function Point(x,y) {
   this.x = x;
   this.y = y;
   //... methods that use x and y properties
}
var pt = new Point(1,2);
```

```
function Point(x,y) {
   private x, y;
   this.x = x;
   this.y = y;
   //... methods that use private x and y properties
}
var pt = new Point(1,2);
```

```
function Point(x,y) {
   const x___ = SoftField(), y___ = SoftField();
   x___.set(this, x);
   y___.set(this, y);
   //... methods that use private x and y properties
}
var pt = new Point(1,2);
```

Accessing Private Identifiers as Soft Field Values

Adapted from accessing private names as values

The `private` declaration normally both creates a new soft field and introduces a identifier binding that can be used only in "property name" syntactic contexts to access the new soft field by the lexically bound identifier.

However, in some circumstances it is necessary to access the actual soft field as an expression value, not as an apparent property name on the right of **.** or the left of **:** in an object initialiser. This requires a special form than can be used in an expression to access the soft field binding of a private identifier. The syntactic form is **#.** *IdentifierName*. This may be used as a *PrimaryExpression* and yields the soft field of the *IdentifierName*. This may be either a soft field or a string value, depending upon whether the expression is within the scope of a `private` declaration for that *IdentifierName*;

```
function addPrivateProperty(obj, init) {
   private pname;     //create a new soft field
   obj.pname = init;  //set this soft field
   return #.pname;    //return the soft field
}
```

```
function addPrivateProperty(obj, init) {
   const pname___ = SoftField();
   pname___.set(obj, init);
   return pname___;
}
```

```
var myObj = {};
var answerKey = addPrivateProperty(myObj, 42);
print(answerKey.get(myObj));  // AFAICT, this is the *only* claimed advantage of Names
over SoftFields.
//myObj can now be made globally available but answerKey can be selectively passed to
privileged code
```

Note that simply assigning a soft field to a variable does not make that variable a private identifier. For example, in the above

example, the print statement could not validly be replaced with:

```
print(myObj.answerKey);
```

This would produce "`undefined`" because it would access the non-existent property whose string valued property name would be "`answerKey`". Only identifiers that have been explicitly declared using `private` are private identifiers.

"can we subsume private names" explains how soft fields as value proxies could support a property-like usage of [], so this code could indeed be written as

```
print(myObj[answerKey]);
```

If **#.** is not within the scope of a `private` declaration for its *IdentifierName* then the value produced is the string value of the *IdentifierName*.

As an expressive convenience, `private` declarations can be used to associate a private identifier with an already existing soft field. This is done by using a `private` declaration of the form:

`private` *Identifier* **=** *Initialiser* **;**

The Names proposal asks: "If *Initialiser* does not evaluate to a soft field, a TypeError exception is thrown. (? *for uniformity, should string values be allowed? In that case, local private name bindings could be string valued.*)"

If the answer is true, the one supposed advantage of Names over soft fields goes away. Our contentious bit of code becomes:

```
private ak = answerKey; // soft field or string
print(obj.ak); // works either way
```

```
private name1;   //value is a new soft field
private name2 = #.name1  //name2 can be used to access the same soft field as name1
```

Other possible syntactic forms for converting a private identifier to an expression value include:
`private` *IdentifierName*
(`private` *IdentifierName***)**
. *IdentifierName*
` *IdentifierName*
#` *IdentifierName*
#' *IdentifierName*

Conflict-Free Object Extension Using Soft Fields

Adapted from conflict-free object extension using private names

```
function installCloneLibrary() {
   private clone;   // the soft field for clone methods

   // Install clone methods in key built-in prototypes:
   Object.prototype.clone = function () { ... };
   Array.prototype.clone = function () {
```

```
        ...
        target[i] = this[i].clone();  // recur on clone method
        ...
    }
    String.prototype.clone = function () {...}
    ...
    return #.clone
}

// Example usage of CloneLibrary:
private clone = installCloneLibrary();
installAnotherLibrary();
var twin = [{a:0}, {b:1}].clone();
```

Similarities: The above client of the `CloneLibrary` will work even if the other library also defines a method named `clone` on `Object.prototype`. The second library would not have visibility of the soft field used for `clone` so it would either use a string property name or a different soft field for the method. In either case there would be no conflict with the method defined by `CloneLibrary`.

Crucial difference

For defensive programming, best practice in many environments will be to freeze the primordials early, as the dual of the existing best practice that one should not mutate the primordials. Evaluating the dynamic behaviour of Python applications (See also http://gnuu.org/2010/12/13/too-lazy-to-type/) provides evidence that this will be compatible with much existing content. We should expect these best practices to grow during the time when people feel they can target ES5 but not yet ES6.

Consider if Object.prototype or Array.prototype were already frozen, as they should be, before the code above executes. Using soft fields, this extension works. Using private names, it is rejected. Allen argues at Private names use cases that

```
    Allow third-party property extensions to built-in
    objects or third-party frameworks that are guaranteed
    to not have naming conflicts with unrelated extensions
    to the same objects.
```

is the more important use case. Soft fields provide for this use case. Private names do not.

Who knows whether frozen primordials will catch on? Many JS hackers are vehemently opposed. PrototypeJS still extends built-in prototypes and its maintainers say that won't change. Allen clearly was talking about extending non-frozen shared objects in his "Private names use cases" message – he did not assume what you assume here. We need to agree on our assumptions before putting forth conclusions that we hope will be shared. I don't think everyone shares the belief that "We should expect these best practices to grow during [any foreseeable future]."

— *Brendan Eich 2010/12/22 01:37*

Are we still confusing "any" and "all"? The original quote claims only that these best practices will grow in some environments. Regarding your "any foreseeable future", this future is already long past. Google JavaScript Style Guide: Modifying prototypes of builtin objects has long stated:

```
  Modifying prototypes of builtin objects
  [Recommendation:] No
  Modifying builtins like Object.prototype and Array.prototype
  are strictly forbidden. Modifying other builtins like
  Function.prototype is less dangerous but still leads to hard
  to debug issues in production and should be avoided.
```

I'm sure other such quotes about JavaScript best practice can be found.

Also, of course, The last initialization step of initSES is to freeze the primordials of its frame. Only code that does not mutate their primordials will be directly compatible with SES without resort to sandboxing.

Mark, the original quote from you is visible above, and it asserts "many", not "any". That is a bold claim. Not only Prototype, but SproutCore and Moo (and probably others), extend standard objects. SproutCore adds a `w` method to `String.prototype`, along with many other methods inspired by Ruby.

It's nice that Google has recommendations, which it can indeed enforce as mandates on employees, but the Web at large is under no such authority. The Web is the relevant context for quantifying "many", not some number of secure subset languages used in far smaller domains. On the Web, it's hard to rule out maintainers and reusers mixing your code with SproutCore, e.g.

SES is a different language from Harmony, not standardized by Harmony in full. Goal 5 at harmony is about supporting SES, not subsuming it.

I believe we should avoid trying to run social experiments, building up pedagogical regimes, or making predictions about the future, anywhere in the text of future ECMA-262 editions.

— *Brendan Eich 2011/01/12 02:12*

Enumeration and Reflection

enumeration and reflection

Even though soft fields are typically implemented as state within the object they extends, because soft fields are semantically not properties of the object but are rather side tables, they do not show up in reflective operations performed on the object itself.

For example:

```
private b;
var obj = {};
obj.a = 1;
obj.b = 2;
obj.c = 3;

var names = [];
for (var p in obj) names.push(obj[p]);
print(names.toString());    // "1,3" -- soft field "b" was not enumerated
```

Soft fields created using object literals also not part of the object itself. So `obj` could have been created to produce the same result by saying:

```
private b;
var obj = {
    a: 1,
    b: 2,
    c: 3
}
```

Beyond the syntactic expansions explained above, no other change to the definition of object literals is needed.

Creating a soft field that is enumerable makes no sense. Reflective operations that take property names as arguments, such as Object.defineProperty below, if given a non-string argument including a soft field, would coerce it to string and (uselessly) use that as a property name.

```
private b;
var obj = {};
obj.a = 1;
obj.b = 2;
Object.defineProperty(obj, #.b, {enumerable: true});
obj.c = 3;

var names = [];
for (var p in obj) names.push(obj[p]);
print(names.toString());    // "1,2,3" -- property "[object Object]" is now enumerated
```

`Object.prototype.hasOwnProperty` (ES5 15.2.4.5), `Object.prototype.propertyIsEnumerable` (ES5 15.2.4.7) and the `in` operator (ES5 11.8.7) do not see soft fields, again, because they are not part of the object.

The `JSON.stringify` algorithm (ES5 15.12.3) needs no change in order to ignore soft fields, since again they are not part of the object.

All the Object reflection functions defined in ES5 section 15.2.3 remain unchanged, since they need not be aware of soft fields.

An important use case for reflection using soft fields is algorithms that need to perform meta-level processing of all properties of any object. For example, a "universal" object copy function might be coded as:

```
function copyObject(obj) {
   // This doesn't deal with other special [[Class]] objects:
   var copy = Object.isArray(obj) ? [] : Object.create(Object.getPrototypeOf(obj));
   var props = Object.getOwnPropertyNames(obj);
   var pname;
   for (var i = 0; i < props.length; i++) {
      pname = props[i];
      Object.defineProperty(copy, pname, Object.getOwnPropertyDescriptor(obj,pname));
   }
   return obj;
}
```

This function will duplicate all properties but not any soft fields, preserving encapsulation, since neither the definer nor the caller of copyObject knows these soft fields. Of course, a more complex copyObject function could be defined that would also copy and re-index those soft fields it was told of.

Soft Fields Support Encapsulation

Adapted from private name properties support only weak encapsulation

No qualifiers needed.

Should so-called "weak encapsulation" actually be desired, "can we subsume private names" explains how to provide *weakly encapsulating soft fields* (or "wesf") polymorphically with soft fields.

Interactions with other Harmony Proposals

Enhanced Object Literals

Adapted from enhanced object literals

`private` might be supported as either a property modifier keyword that makes the property name a soft field whose private identifier is scoped to the object literal:

```
var obj={
   private _x: 0;
   get x() {return this._x},
   set x(val) {this._x=val}
}
```

This might simplify the declarative creation of objects with instance private soft fields. However, there are internal scoping and hoisting issues that would need to be considered and resolved.

Another alternative is to use meta property syntax to declare object literal local soft field declarations:

```
var obj={
   <prototype: myProto; private _x>
   _x: 0;
   get x() {return this._x},
   set x(val) {this._x=val}
}
```

While the above proposals are perfectly consistent with soft fields, again, for instance-private instance state, using lexical capture seems strictly superior:

```
let x = 0;
var obj={
   get x() {return x},
   set x(val) {x=val}
}
```

Proxies

Adapted from proxies

None of the uses of string valued property names in proxy handlers would need to be extended to accept/produce soft fields in addition to string values.

As covered above, ECMAScript reflection capabilities provides no means to break the encapsulation of an object's soft fields.

Modules

Adapted from modules

It is reasonable to expect that modules will want to define and export soft fields. For example, a module might want to add methods to a built-in prototype object using soft fields and then make those soft fields available to other modules. Within the present definition of the simple module system that might be done as follows:

```
<script type="harmony">
module ExtendedObject {
   import Builtins.Object;        // however access to Object is obtained.
   private clone;                 // the soft field for clone methods
   export const clone = #.clone;  // export a constant with the soft field;

   Object.prototype.clone = function () { ... };
```

```
}
</script>
```

A consumer of this module might look like:

```
<script type="harmony">
import ExtendedObject.clone;
private clone = clone;
var anotherObj = someObj.clone();
</script>
```

The above formulation would work without any additional extensions to the simple module proposal. However, it would be even more convenient if the module system was extended to understand private declarations. In that case this example might be written as:

```
<script type="harmony">
module ExtendedObject {
    import Builtins.Object;      // however access to Object is obtained.
    export private clone;        // export soft field for clone methods

    Object.prototype.clone = function () { ... };
}
</script>
```

```
<script type="harmony">
import private ExtendedObject.clone;
var anotherObj = someObj.clone();
</script>
```

I don't get the point about "dynamic access to the exported property name environment of first-class module instances", so at this time I offer no comparison of this last example.

References

Adapted from references

Any unforgeable reference to a tamper-proof encapsulated object is analogous to a capability in object-capability languages. In this degenerate sense, both Names and Soft Fields are also so analogous. I see no further way in which Names are analogous. In addition, Soft Fields encourage encapsulation friendly patterns, whereas Names encourage unsafe (or "weakly encapsulated") patterns.

RSS XML FEED | LICENSED | $1 DONATE
PHP POWERED | W3C XHTML 1.0 | W3C CSS | DOKUWIKI

[[strawman:
quasis]]

ES
Wiki

Trace: » classes_with_trait_composition » traits_semantics » inherited_explicit_soft_fields » names_vs_soft_fields » quasis

Table of Contents

- EcmaScript Quasi-Literal Strawman
  - Motivation
  - Overview
    - Syntax
    - Semantics
  - Use Cases
    - Secure Content Generation
    - Text L10N
    - Query Languages
    - Message Sends
    - Flexible Literal Syntax
    - Raw Strings
    - Decomposition Patterns
    - Logging
  - Syntax (normative)
    - QuasiLiteral ::
    - QuasiTypeTag ::
    - LiteralPortion ::
    - LiteralCharacter ::
    - QuasiLiteralTail ::
    - Substitution ::
    - SubstitutionBody ::
    - SubstitutionBodyPart ::
  - Semantics (normative)
    - Desugaring
    - QFN
    - LPA
    - SVE
  - Security Considerations
    - Defensive Code
    - Offensive Code
    - Possible Problems
  - Reasons and Open Issues
    - Quoting Character
    - Substitutions
    - Raw Escapes in Literal Sections
    - Determining Where a Backquoted Section Ends
    - Line Continuation
  - References
    - Quasis in E
    - Secure String Interp
    - PHP String Vars
    - PLT Scheme Scribble
    - SML of New Jersey
    - Secure Code Generation
    - Scheme Hygienic Macros
    - Paradigm Regained

EcmaScript Quasi-Literal Strawman

Motivation

EcmaScript is frequently used as a glue language for dealing with content specified in other languages : HTML, CSS, JSON, XML, etc. Libraries have implemented query languages and content generation schemes for most of these : CSS selectors, XPath, various templating schemes. These tend to suffer from interpretation overhead, or from injection vulnerabilities, or both.

This scheme extends EcmaScript syntax with syntactic sugar to allow libraries to provide DSLs that easily produce, query, and manipulate content from other languages that are immune or resistant to injection attacks such as XSS, SQL Injection, etc.

This scheme aims to preserve ES5 strict mode's static analyability while allowing details of the DSL implementation to be dynamic.

Overview

Syntax

```
x`foo${bar}baz`
```

Syntactically, a quasi-literal is a **function name** (`x`) followed by zero or more characters enclosed in back quotes. The contents of the back quotes are grouped into **literal sections** (`foo` and `baz`) and **substitutions** (`bar`).

A substitution is an unescaped substitution start character (`$`) followed by either a valid *Identifier* or a curly bracket block. E.g., `$foo` or `${foo + bar}`.

The literal sections are the runs of characters not contained in substitutions. They may be blank so the number of literal sections is always one greater than the number of substitutions.

The body of a substitution should be a valid *Expression* but the syntax specified below is independent of that so that the boundaries between literal sections and substitutions are independent of any vendor language extensions or future changes to the language.

Semantics

The semantics of quasi-literals are specified in terms of a desugaring which has the property that the free variables of the desugaring are the same as the union of the free variables of the substitutions and the function name.

Use Cases

This syntactic sugar will let library developers experiment with a wide range of language features.

Quasi-literals desugar a back quoted string to a function call that operates on the literal portions. That handler can return a function (possibly from a cache) that receives thunks of the substituted expressions.

E.g. `quasiHandlerName`quasiLiteralPart1 ${quasiSubstitution} quasiLiteralPart2`` desugars to

```
quasiHandlerName(
    ['quasiLiteralPart1 ', ' quasiLiteralPart2'])(
    [function () { return quasiSubstitution; }])
```

Secure Content Generation

```
safehtml`<a href="//example.org/main?q=${query}">${linkText}</a>`
```

where `safehtml` analyzes the literal chunks to figure out that `query` should be percent-encoded and `linkText` HTML entity encoded to prevent XSS.

The syntax provides a clear distinction between trusted content such as

```
<a href="//example.org/main?q=
```

and values that might be controlled by an attacker such as `query`. This prevents all the problems that arise in other languages when format strings can be controlled by an attacker. Although EcmaScript's memory abstractions are not vulnerable, it is very vulnerable to quoting confusion attacks and developers have trouble distinguishing content from an untrusted format string from that produced from a trusted one.

Similar schemes can work for securely composing URLs, JSON and XML data bundles, and for allowing composable SQL prepared statements.

Text L10N

```
msg`Your account has a balance of ${balance}:.2${currency}`
```

where `+.2` can be treated as meta-data by the `msg` function and used to format the balance number with 2 digits of precision.

Since there is a convenient simple format for human-readable messages, a static analyzer can more easily find them (to substitute locale-specific versions) than if messages were simply the first argument to a function call.

Query Languages

```
$`a.${className}[href=~'//${domain}/']`
```

might specify a DOM query for all `<a>` elements with the given class name and that link to URLs with the given domain.

The `className` and `domain` do not need to be encoded then decoded by a query-engine so mis-encodings can be eliminated as a class of bugs and source of inefficiency.

Message Sends

Message sends can be specified using a syntax that looks like an HTTP request.

```
GET`http://example.org/service?a=${a}&b=${b}
    Content-Type: application/json
    X-Credentials: ${credentials}

    { "foo": ${foo}, "bar": ${bar} }`(myOnReadyStateChangeHandler);
```

might configure an

```
XMLHttpRequest
```

object to the specified (securely composed) URL with the given (securely composed) headers, and after the end of the headers could switch to context-sensitive composition based on the content-type header : JSON in this case, or an XML message in another case.

Flexible Literal Syntax

Often, developers use the `new RegExp(...)` constructor because they want a tiny part of their regular expression to be dynamic, and fail to properly escape character classes such as "`\s`".

A quasi syntax for regex construction

```
re`\d+(${localeSpecificDecimalPoint}\d+)?`
```

gets the benefit of the literal syntax with dynamism where needed.

Raw Strings

Python raw strings are trivial:

```
raw`In JavaScript '\n' is a line-feed.`
```

Decomposition Patterns

A pattern decomposition handler `re_match` invoked thus

```
if (re_match`foo (${=x}\d+) bar`(myString)) {
  ...
}
```

could use assignable substitutions to achieve the same effect as

```
{
  let match = myString.match(/foo (\d+) bar/);
  if (match) {
    x = match[1];
    ...
  }
}
```

Logging

```
warn`Bad result $result from $source`
```

can provide `console.log("o=%s", o)` style logging of structured data without the need for positional parameters.

Syntax (normative)

QuasiLiteral is a kind of *PrimaryExpression*.

QuasiLiteral ::

-

QuasiTypeTag [no *LineTerminator* here] ` *LiteralPortion QuasiLiteralTail* `

QuasiTypeTag ::

- *Identifier*

LiteralPortion ::

- *LiteralCharacter LiteralPortion*
- ε

LiteralCharacter ::

- *SourceCharacter* **but not** back quote ` **or** *LineTerminator* **or** dollar $
- *LineTerminatorSequence*
- $ \ *EscapeSequence*

QuasiLiteralTail ::

- *Substitution LiteralPortion QuasiLiteralTail*
- ε

Substitution ::

- $ *Identifier*
- $ { *SubstitutionBody* }
- $ { = *SubstitutionBody* }

SubstitutionBody ::

- *SubstitutionBodyPart SubstitutionBody*
- ε

SubstitutionBodyPart ::

- *LiteralCharacter* **but not** { **or** } **or** ` **or** " **or** ' **or** \
- { *SubstitutionBody* }
- ` *LiteralPortion QuasiLiteralTail* `
- *StringLiteral*
- \ [**not** *LineTerminator*]
- *LineContinuation*

Semantics (normative)

A quasi-literal desugars to a call to a javascript function specified by the *QuasiTypeTag*. The function called is specified in terms of the quasi function name (QFN) which is called with the literal portion arguments (LPA) to produce a curried function which is then called with the substituted value expressions (SVE).

Desugaring

The desugaring of *QuasiLiteral* :: *QuasiTypeTag* ` *LiteralPortion QuasiLiteralTail* ` is a *CallExpression* whose *Arguments* is the SVE of the *QuasiLiteralTail* and whose *MemberExpression* is a nested *CallExpression* whose *Arguments* is the LPA of the *QuasiLiteral* and whose *MemberExpression* is the QFN of the *QuasiTypeTag*.

QFN

The QFN of *QuasiTypeTag* :: *Identifier* is the text of the *Identifier* after decoding *EscapeSequence*s.

LPA

Production	Result
QuasiLiteral :: `*LiteralPortionQuasiLiteralTail*` *QuasiTypeTag*	array-concat(LPA(*LiteralPortion*), LPA(*QuasiLiteralTail*))
QuasiLiteralTail :: *SubstitutionLiteralPortionQuasiLiteralTail*	array-concat(single-element-array(LPA(*LiteralPortion*)), LPA (*QuasiLiteralTail*))
QuasiLiteralTail :: ε	an empty array
LiteralPortion :: *LiteralCharacterLiteralPortion*	string-concat(LPA(*LiteralCharacter*), LPA(*LiteralPortion*))
LiteralPortion :: ε	the empty string
LiteralCharacter :: *SourceCharacter*	single character string containing that character.
LiteralCharacter :: *LineTerminatorSequence*	the single character string containing a LF ("\n")
LiteralCharacter :: $*EscapeSequence*	CV(*EscapeSequence*)

SVE

Production	Result
QuasiLiteral :: *QuasiTypeTag*`*LiteralPortion QuasiLiteralTail*`	SVE(*QuasiLiteralTail*)
QuasiLiteralTail :: *Substitution LiteralPortion QuasiLiteralTail*	array-concat(single-element-array(SVE(*Substitution*)), SVE (*QuasiLiteralTail*))
QuasiLiteralTail :: ε	an empty array

Substitution :: $*Identifier*	getter(SV(*Identifier*))
Substitution :: ${*SubstitutionBody*}	getter(str-concat("(", SVE(*SubstitutionBody*), ")"))
Substitution :: ${=*SubstitutionBody*}	setter(str-concat("(", SVE(*SubstitutionBody*), ")"))
SubstitutionBody :: *SubstitutionBodyPart SubstitutionBody*	str-concat(SVE(*SubstitutionBodyPart*), SVE(*SubstitutionBody*))
SubstitutionBodyPart :: *LiteralCharacter*	CV(*LiteralCharacter*)
SubstitutionBodyPart :: {*SubstitutionBody*}	str-concat("{", SVE(*SubstitutionBody*), "}")
SubstitutionBodyPart :: `` `*SubstitutionBody*` ``	str-concat("`", literalText(*SubstitutionBodyPart*), "`")
SubstitutionBodyPart :: *StringLiteral*	literalText(*StringLiteral*)
SubstitutionBodyPart :: \	"\\"
SubstitutionBodyPart :: *LineContinuation*	"\\\n"

`getter(jsExpressionSource)` produces a function by parsing `jsExpressionSource` as a the target of a `return` statement and creating a parameterless function whose body is that `return` statement and binding `this` in that function. It is equivalent to invoking the below if brackets in the input are balanced or throwing a `SyntaxError` otherwise in a context where `eval` has not been masked and `Function.prototype.bind` is unmodified:

```
eval('function () { return ' + jsExpressionSource + ' }').bind(this)
```

`setter(jsExpressionSource)` is similar to `getter` but produces a return statement like the following:

```
eval('function () { return arguments.length ? ' + jsExpressionSource + ' = arguments[0] : '
     + jsExpressionSource + ' }').bind(this)
```

It is a `ReferenceError` for `jsExpressionSource` to contain a free use of the identifier `arguments`.

Alternatively, `getter` and `setter` can be specified using lambdas which would let us expand `String` to be any syntactically valid `Program` which would let macros be used to implement flow control constructs and properly handle `this` and `arguments` inside `jsExpressionSource`.

Security Considerations

This strawman should also fall in the language subset defined by SES (Secure EcmaScript). As such, neither its presence in the language nor its use in a program should make it substantially more difficult to reason about the security properties of that program.

Developers expect that object only escape a scope by being explicitly passed or assigned. This strawman needs to preserve both the scope invariants of EcmaScript 5 functions and catch blocks, and those introduced by the modules and `let` proposals.

The below discusses the interaction between a quasi function defined in one scope/module and the code it produces to be executed in another scope/module. The actors include

- library author – the author of the module / scope in which the quasi function is defined
- quasi author – the author of the quasi-literal and any symbols defined in the module / scope containing it.

Defensive Code

A module needs to be able to defend its invariants against bugs or deliberate malice by another module. SES does not attempt to guarantee availability since trivial programs can loop infinitely, but a module must be able to guarantee that its invariants hold when control leaves it.

This proposal does not complicate defensive code reasoning because:

- only symbols mentioned in a substitution are observable by the library author
- only symbols marked as writable can be written by the library author

The quasi author has to be aware that the order of evaluation is unclear. For quasis to specify new control constructs, substitutions need to be evaluable out of order, repeatedly, or not at all.

By writing a substitution, the quasi author is conveying the authority to evaluate an expression in the quasi scope any number of times from that point on. (Assuming the quasi module has the authority to cause delayed evaluation as by `setTimeout`). A substitution conveys the same authority as a zero argument or `function`.

Offensive Code

The library author's quasi function may be used by multiple mutually suspicious or intentionally isolated modules. It can ensure that bugs or malice in one module do not affect its ability to serve another module by freezing the symbols it exports and by coding defensively.

This proposal does not complicate its ability to do that, since it imposes no mutable data requirements on quasi functions.

Possible Problems

This syntax is, by design, similar to that of string interpolation in other languages. Users may assume the result of the quasi-literal is a string as occurs in languages like Perl and PHP (3), and that subsequent mutations to values substituted in do not affect the result of the interpolation. It is the responsibility of QFN implementers to match these expectations or to educate users. Specifically, developer surprise might result from the below if `q` kept a reference to the mutable fib array which is modified by subsequent iterations of the loop.

```
var quasis = [];
var fib = [1, 1];  // State shared across loop bodies
for (var i = 1; i &lt; 10; ++i) {
  fib[1] += fib[0];
  fib[0] = fib[1] - fib[0];
  quasis.push(q`Fib${i-1} and fib${i} are $fib`);
}
```

String interpolation in other languages is often a vector for quoting confusion attacks : XSL, SQL Injection, Script Injection, etc.. It is the responsibility of QFN implementers to properly escape substituted values, and a lazy escaping scheme (2) can provide an intelligent default. It is a goal of the proposed scheme to reduce the overall vulnerability of EcmaScript applications to quoting confusion by making it easy for developers to generate properly escaped strings in other languages.

Quasi-literals contain embedded expressions, but the set of lexical bindings accessible to the quasi handler is restricted to the union of the below so they do not complicate static analysis

1. the set of identifiers mentioned by the author in the lexical environment in which the quasi-literal appears,
2. the lexical environment of the QFN in the environment in which it is defined,
3. for QFNs defined in non-strict mode, the global object as bound to `this`.

Reasons and Open Issues

Quoting Character

The meaning of existing programs should not change, so this proposal must extend the grammar without introducing ambiguity. It is meant to enable secure string interpolation and DSLs, so using a syntax reminiscent of

strings seems reasonable, and many widely used languages have string interpolation schemes which will reduce the learning curve associated with the proposed feature.

Backquote (`) was chosen as the quoting character for string interpolations because it is unused outside strings and comments; and is obviously a quoting character.

It is already used in other languages that many EcmaScript authors use – perl, PHP, and ruby where it allows interpolation though with a more specific meaning than macro expansion. It is used as a macro construct in Scheme where it is called a "quasiquote." In Python 2.x and earlier, it is a shorthand for the `repr` function, so contained an expression and applied a specific transformation to it.

As such, many syntax highlighters deal with it reasonably well, and programmers are used to seeing it as a quote character instead of as a grave accent.

Alternatives include:

- ```
  q"""Interpolate $this!"""
  ```

  which could conflict with long strings.

- ```
  q"Interpolate $this!"
  ```

  which simply uses an existing quoting character.

- ```
  q{{"Interpolate $this!"}}
  ```

  which simplifies nesting.

- ```
  q(:"Interpolate $this!":)
  ```

  which is friendly even if not user friendly.

Substitutions

Since we're choosing syntax to reduce the learning curve, we chose `${...}` since it is used to allow arbitrary embedded expressions in PHP and JQuery templates. We also include the abbreviated form (`$ident`) to be compatible with Bash, Perl, PHP, Ruby, etc.

We decided against `sprintf` style formatting, since, although widely understood, it does not allow many DSL applications, and imposes an O(n) cognitive load (2).

Alternatives include:

-

- Bash: `$(...)`
- Ruby: `#{...}`
- PHP: `${...}`

Raw Escapes in Literal Sections

A backslash (\) in a quasi-literal could be interpreted immediately as an *EscapeSequence* or passed as a raw value to the quasi function. A quasi function can always be wrapped to decode escapes:

```
function quasiFunctionWithJsDecodedLiteralPortions(quasiFunctionWithRawLiteralPortions) {
  "use strict";
  var DECODE = { n: '\n', r: '\r', v: '\x0b', f: '\f', t: '\t', b: '\b' };
  function decode(s) {
    return s.replace(
         /\\(?:([rnftvb])|(\r\n?|[\n\u2028\u2029])|(x[0-9A-Fa-f]{2}|u[0-9A-Fa-f]{4})|([0-
3][0-7]{0,2}|[4-7][0-7]?)|(.))/g,
        function (_, e, lt, hex, oct, lit) {
          return e  ? DECODE[e]
              : lt  ? '\n'
              : hex ? parseInt(hex.substring(1), 16)
              : oct ? parseInt(oct, 8)
              : lit;
        });
  }
  return function (literalPortions) {
    return quasiFunctionWithRawLiteralPortions(
        map(decode, literalPortions));
  };
}
```

The *Substitution* :: $ \ *EscapeSequence* production allows for an arbitrary JS escape, so any representable string literal is representable as a *LiteralPortion* in a quasi-literal.

We lose no generality by treating escapes as raw, and there are use cases where raw escapes are useful, as in a regular expression composing scheme

```
var my regexp = re`(?i:\w+$foo\w+)`;

function re(literalPortions) {
  for (var i = arguments.length; --i >= 0;) {
    literalPortions[i * 2] = arguments[i];
  }
  return function (substitutions) {
    var regexBody = literalPortions.slice(0);
    for (var i = 0, n = substitutions.length; i < n; ++i) {
      var sub = substitutions[i]
      regexBody[i * 2 + 1] = sub().replace(
          /[\\(){}\[\]^$.+*?|\-]/g, '\\$&');
    }
    return new RegExp(regexBody.join(''));
  };
}
```

Determining Where a Backquoted Section Ends

The *SubstitutionBody* production is meant to allow embedding of an arbitrary JS expression, so absent other constraints it could be specified as:

- *SubstitutionBody* :: *Expression*

This proposal instead makes *SubstitutionBody* self-contained so that future changes to the *Expression* production cannot affect where a string ends.

To work with existing syntax highlighters and code analyzers, we need to not overly complicate the grammar. With the existing string productions, finding the end of the string is as simple as /"([^"\\]|\\.)*"/. DSLs do need to nest though, so we chose a slightly more complex grammar that requires balanced backquotes and curly brackets and complete quoted strings, but otherwise nests well.

Future additions to the language grammar should not change how programs that don't use the new productions parse.

Line Continuation

Both strings and regular expressions in EcmaScript 5 allow *LineContinuation*s, escaped line breaks that are treated as lexially insignificant.

It would be convenient for some DSL use cases to allow *LineTerminator*s inside code, but it is unclear how this will interact with revision control systems that rewrite newlines on checkout.

Allowing embedded line terminators and line continuations interacts badly with the way that *LiteralPortion*s' contents are escaped. Consider the following code where ¶ indicates where a newline occurs: `foo\¶bar` vs `foo \ ¶bar`. The former is equivalent to `foobar` while the latter is equivalent to `foo\ bar` though the difference is not visible. Existing string productions do not suffer this problem because a line terminator cannot appear inside a string unescaped.

Options include

- Allow newlines inside quasi literals and treat *LineContinuation*s as normal content, consistent with the way escapes are treated as raw inside *LiteralPortion*s.
- Interpret *LineContinuation*s as an empty sequence of characters and allow disallow *LineTerminator*s otherwise.
- Disallow *LineTerminator*s in quasi literals.

References

Quasis in E

Quasiliterals in E

Secure String Interp

Secure String Interpolation

PHP String Vars

PHP String variable parsing

PLT Scheme Scribble

PLT Scheme Scribble Syntax

SML of New Jersey

SML/NJ has a similar Quote/Antiquote feature (whose documentation, ironically enough, has an HTML bug in a quoted code snippet, resulting in the bottom third or so of the page being in monospaced font).

Secure Code Generation

"Secure Code Generation for Web Applications" by Martin Johns.

Scheme Hygienic Macros

Scheme Macros FAQ

Paradigm Regained

Paradigm Regained : Abstraction Mechanisms for Access Control

Table of Contents

- Function to String conversion
  - Problematic cases:
    - Built-in functions
    - Callable non functions, including callable host objects
    - Bound functions
    - Function proxies
- Discussion
- Acks

Function to String conversion

`Function.prototype.toString.call(fn)` must return source code for a FunctionDeclaration or FunctionExpression that, if `eval()`uated in an equivalent-enough lexical environment, would result in a function with the same [[Call]] behavior as the present one. Note that the new function would have a fresh identity and none of the original's properties, not even `.prototype`. (The properties could of course be transferred by other means but the identity will remain distinct.)

This returned source code must not mention freely any variables that were not mentioned freely by the original function's source code, even if these "extra" names were originally in scope. With this restriction, an equivalent-enough lexical environment need only provide bindings for names used freely in the original source code. For purposes of this scope analysis, a use of the direct `eval` operator is statically considered a free usage of all variables in scope at that point.

Allowing FunctionExpression in the spec above acknowledges reality. All major JS engines will convert an anonymous function to an anonymous FunctionExpression, even though the ES3 and ES5 specs disallow it. This behavior is useful, so we should make it official.

Problematic cases:

Built-in functions

As of this writing, most JS engines convert these to, for example, "`function join() { [native code] }`". As a widespread convention this will be hard to displace. However, it is unpleasant on several grounds:

- It does not parse as a FunctionDeclaration (violating the de-jure spec) nor as a FunctionExpression (violating the rest of the de-facto spec).
- It does not parse as any valid JavaScript production, making it useless as input to `eval()`.

- It conflicts with the spec's use of the term "native", which includes all function written in JavaScript. Rather, it is probably derived from the Java meaning of "native" which ES5 and ES3 call "built ins". (Another way to resolve this conflict is to change our terminology to conform to the rest of the world's meaning of "native".)

If this behavior could be displaced, for primordial built ins, an alternative with some virtues is to have it print as a FunctionExpression that calls whatever is at the conventional location at which this built-in is normally found. For example: "`function(...args) { return Array.prototype.join.apply(this, args); }`".

For the non-primordial built ins, or perhaps for all built ins, we could convert them to a FunctionExpression that uses freely a conventional name that represents the "actual" built-in function, so that `eval()`ing the FunctionExpression in an environment in which `original` was bound to that built in would preserve call behavior. For example: "`function(...args) { return original.apply(this, args); }`"

Callable non functions, including callable host objects

Solutions for built ins should apply to these as well, since all we're trying to preserve is [[Call]] behavior.

Bound functions

Applying the same trick, "`f.bind(self, a, b)`" might print as "`function(...args) { return original.call(p1, p2, ...args); }`". The `eval()`uates to a function with the same [[Call]] behavior if evaluated in an environment in which `original` is bound to the original function and `p`*N* is bound to each of the arguments originally provided to that call to `bind()`.

Function proxies

If `fp` is a function proxy with `ct` as its *call trap*, then `Function.prototype.toString.call(fp)` is already specified to return whatever `Function.prototype.toString.call(ct)` would return. Since function proxies have precisely the [[Call]] behavior of their call trap, both before and after fixing, this works.

Discussion

The goal assumed here – that `eval()`uating the string in an equivalent enough environment would preserve [[Call]] behavior – to be useful, we would need to be able to construct an equivalent enough environment. For many reasons, this seems impossible in the general case, so it is questionable whether it's worth much trouble to provide this feature. Alternatively, we could make current reality official and mandate that built ins must convert to a string that does *not* parse

as any valid JavaScript production. The current de-facto behavior already satisfies that spec.

Going in the other direction, various useful recognition tricks need a stronger spec. Preserving equivalence under `eval()` doesn't help these. Preserving exactly the original source code, or preserving ASTs, or some abstraction over equivalent ASTs such as alpha renaming of non-free variables, would all enable these recognition tricks. Are we willing to go that far?

Acks

Someone, please edit this to credit whoever originally made this suggestion on the es-discuss list.

[[strawman:
simple_modules]]

ES
Wiki

Trace: » inherited_explicit_soft_fields » names_vs_soft_fields » quasis » function_to_string » simple_modules

Table of Contents

- Simple Modules
  - Goals
  - Terminology
  - Syntax
  - Module declarations
  - Inline module declarations
  - External module load
  - Import declarations
  - Export declarations
  - Compile-time resolution and linking
  - Run-time execution
  - First-class module references
  - Module instance objects
  - Reflective evaluation
  - This

Simple Modules

This proposal describes a module system and a new top-level scoping semantics for ECMAScript. It builds off of ideas from ES5 strict mode and lexical scope mode. It is backwards-incompatible and relies on versioning to indicate that code is evaluated with the new semantics.

See the simple modules examples page for some highlights.

Goals

- Static scoping
- Orthogonality from existing features
- Smooth refactoring from global code to modular code
- Fast compilation
- Simplicity and usability
- Standardized protocol for sharing libraries
- Compatibility with browser and non-browser environments
- Easy external loading

Terminology

- *Module*: A unit of source contained within a module declaration or within an externally-loaded file.
- *Module instance*: An evaluated module, linked to other modules and containing lexically encapsulated data/state as well as exported bindings.
- *Module instance object*: A first-class object that reflects the exported bindings of a module instance.

- *Module binding*: A binding in a scope chain record that maps to a statically loaded module.

Syntax

```
Program(load) ::= Directive* ProgramElement(load)*
ProgramElement(load) ::= Statement
                       | VariableDeclaration
                       | FunctionDeclaration
                       | ImportDeclaration(load)
                       | ExportDeclaration(load)

ModuleDeclaration ::= "module" ModuleSpecifier(load) ("," ModuleSpecifier(load))* ";"
                    | ModuleDefinition(load)
ModuleDefinition(load) ::= "module" Identifier "{" ModuleBody(load) "}"
ModuleSpecifier(load) ::= Identifier "=" ModuleExpression(load)

ImportDeclaration(load) ::= "import" ImportPath(load) ("," ImportPath(load))* ";"
ImportPath(load) ::= ModuleExpression(load) "." ImportSpecifierSet
ImportSpecifierSet ::= "*"
                     | IdentifierName
                     | "{" (ImportSpecifier ("," ImportSpecifier)*)? ","? "}"
ImportSpecifier ::= IdentifierName (":" Identifier)?

ExportDeclaration(load) ::= "export" VariableDeclaration
                          | "export" FunctionDeclaration
                          | "export" ModuleDeclaration(load)
                          | "export" ExportPath ("," ExportPath)* ";"
ExportPath ::= ModuleExpression(false) "." ExportSpecifierSet
             | ExportPathSpecifierSet
             | Identifier
ExportSpecifierSet ::= IdentifierName
                     | "{" ExportSpecifier ("," ExportSpecifier)* ","? "}"
ExportSpecifier ::= IdentifierName (":" IdentifierName)?
ExportPathSpecifierSet ::= "{" ExportPathSpecifier ("," ExportPathSpecifier)* ","? "}"
ExportPathSpecifier ::= Identifier
                      | IdentifierName ":" Identifier
                      | IdentifierName ":" QualifiedReference

ModuleExpression(load) ::= ModuleReference(load)
                         | ModuleExpression(load) "." IdentifierName
ModuleReference(load) ::= Identifier
                        | [load = true] Require "(" StringLiteral ")"
Require ::= "require" | "from"
QualifiedReference ::= ModuleExpression(false) "." IdentifierName

ModuleBody(load) ::= Directive* ModuleElement(load)*
ModuleElement(load) ::= Statement
                      | VariableDeclaration
                      | FunctionDeclaration
                      | ModuleDeclaration(load)
                      | ImportDeclaration(load)
                      | ExportDeclaration(load)
```

Module declarations

Module declarations can only appear at the top level of a program or module body. They are compiled and linked during the compilation of their containing program or module.

Inline module declarations

Modules can be declared inline:

```
module Foo {
    export let x = 42;
}
```

External module load

Modules can be loaded from external resoures:

```
    module Bar = require("bar.js");
    import Bar.y;
```

It is not necessary to bind a module to a local name, if the programmer simply wishes to import directly from the module:

```
    import require("bar.js").y;
```

The external module is fetched and compiled during the compilation of the loading module. (Depending on the current module loader, this may trigger user-defined compilation hooks. See module loaders for more information.)

External modules do not name themselves; rather, their files simply contain the contents of the module. This prevents wasteful indentation and allows clients to determine the most appropriate local name for the third-party libraries they load.

An external module is compiled and executed in a completely empty scope chain.

Depending on the module loader, multiple requires may resolve to a shared, single module instance. In this case, the first require that is evaluated executes the module body, and subsequent requires simply produce the same instance without re-executing the body.

Import declarations

Import declarations bind another module's exports as local variables. Imported variables may be locally renamed to avoid conflicts. The * form imports all non-module-exports from a module.

The static variable resolution and linking pass enforces that no conflicts occur.

Export declarations

Export declarations declare that a local binding at the top-level of a module is visible externally to the module. The set of exports of a module is fixed at the module's compile-time. Other modules can read (get) the module exports but cannot modify (set) them. Exports can be renamed so that their external name is different from their local name.

Modules can exports their child modules (including child modules loaded from external files), but they cannot export modules defined elsewhere.

Compile-time resolution and linking

Compilation resolves and validates all variable definitions and references. Linking also happens at compile-time; linking resolves and validates all module imports and exports.

Run-time execution

At run-time, the program is evaluated top-down. Before the program body begins executing, all child modules are instantiated, which is a recursive operation that transitively instantiates all descendent modules. Module instantiation initializes all module top-level `function` bindings, and initializes all variable bindings to the undefined value. Each externally-required module is executed the first time a module binding requires it.

First-class module references

Modules are bound in the same scope chain as other bindings. At run-time, a reference to a module returns a module instance object, which is a run-time reflection of the module instance.

Module instance objects

A module instance object is a prototype-less object that provides read-only access to the exports of the module. All of the exports are provided as getters without setters.

Reflective evaluation

Reflective evaluation, via `eval` or the module loading API starts a new compilation and linking phase for the dynamically evaluated code. As in ES5, the direct `eval` operator inherits its caller's scope chain.

The BNF grammar is parameterized by a `load` parameter, which indicates whether external modules can be loaded. The `eval` operator is a blocking API, so a host environment is permitted to accept the *Program* non-terminal with a `load` value of **false** for the `load` parameter. This prevents the evaluated code from blocking on reads from external resources. (Host environments may instead choose to allow the full grammar, but browsers would not.)

This

The initial binding of `this` at program top-level is a prototype-less, non-extensible, non-configurable reflection of the global environment record. Programs can get and set the global bindings through this object, but cannot add or remove bindings.

The initial binding of `this` at module top-level is the module instance object for that module.

[[strawman:
binary_data]]

ES
Wiki

Trace: » names_vs_soft_fields »
quasis » function_to_string »
simple_modules » binary_data

Table of Contents

- Binary data
  - Goals
  - Examples
- Blocks: compact binary data
  - Block types
  - Block objects
- Numeric data
- Arrays
- Structs
- Block references

Binary data

See also:

- binary data semantics
- binary data discussion

Goals

Provide portable, memory-safe, efficient, and structured access to compact (i.e., contiguously allocated) binary data, as well as an interface for external binary I/O facilities such as XMLHttpRequest, HTML5 File API, and WebGL.

Desiderata:

- expressive and convenient way to create structured binary data
- no new primitive (i.e., non-object) ECMAScript values
- admit architecture-native internal representation while preserving portability:
  - hide struct layout/padding
  - hide endianness
  - prevent multiple interpretations of the same binary data structure at different types
- convenient conversion to native ECMAScript values
- reference semantics without changing ECMAScript evaluation model
- familiar behavior by analogy to C

The design of this library allows implementations to represent allocated binary data in architecture-specific formats – in particular, using the architecture’s native padding/alignment and endianness – without exposing these details to ECMAScript. This allows for efficient implementation while avoiding cross-platform portability hazards.

Examples

```
const Point2D = new StructType({ x: uint32, y: uint32 });
const Color = new StructType({ r: uint8, g: uint8, b: uint8 });
const Pixel = new StructType({ point: Point2D, color: Color });

const Triangle = new ArrayType(Pixel, 3);

let t = new Triangle([{ point: { x:  0, y: 0 }, color: { r: 255, g: 255, b: 255 } },
                      { point: { x:  5, y: 5 }, color: { r: 128, g: 0,   b: 0   } },
                      { point: { x: 10, y: 0 }, color: { r: 0,   g: 0,   b: 128 } }]);
...
```

TODO: more examples

Blocks: compact binary data

This spec introduces an internal datatype called ***blocks***, which intuitively represent contiguously-allocated binary data. Blocks are not themselves ECMAScript values; they live in the program store (i.e., the heap). Blocks can be:

- numbers of various common fixed-size machine types
- arrays of fixed length
- structs of fixed size, with ordered fields

Block types

Every block is associated with a fixed ***block type***, which describes the permanent shape, size, and interpretation of the block, somewhat like a runtime type tag. All references to a given block in the program store are associated with the same block type. Consequently, implementations can allocate blocks as untagged memory buffers (e.g., raw C data structures) without violating memory safety.

Block type objects have a `bytes` property, which reports the logical size of blocks of that type, in bytes. Note that the `bytes` property does not expose information about the *actual* size of a block type, just the *logical* size of its components. This avoids exposing architecture- and implementation-specific details like struct padding.

Block types also mediate conversion from ECMAScript values to raw block data. This is specified via two internal methods:

- [[Convert]] converts an ECMAScript value to a block
- [[Reify]] converts a block to an ECMAScript value

In the semantics, types are compared via an internal [[IsSame]] method. Types are compared similarly to their corresponding C types: numeric and array types are compared structurally, whereas struct types are generative and compared "nominally." (More on this below.)

Block objects

The spec introduces a new object type called ***block objects***, which encapsulate references to block data as ECMAScript values. Reads and writes to the block data underlying the object are marshalled through the conversions specified by the

block types.

Numeric data

Numeric data can be stored in blocks with any of the pre-defined block types:

```
var uint8, uint16, uint32 : BlockType
var int8, int16, int32 : BlockType
var float32, float64 : BlockType
```

Each of these types defines [[Reify]] and [[Convert]] internal methods that convert to and from (respectively) ECMAScript values in a straightforward manner. For example, the ECMAScript value 17 converts to/from the `uint32` value 17, and the ECMAScript value 300 fails to convert to a `uint8` with a TypeError. See binary data semantics for details.

The numeric types can also be called as functions on ECMAScript values. This acts like a C cast, and uses a more permissive casting algorithm, based on the C casting rules.

The numeric types cannot be used as constructors to instantiate block object; using a numeric type with `new` throws an exception. (Objects have reference semantics, and numeric types should have value semantics.)

See binary data discussion for discussion of 64-bit integer types `uint64` and `int64`.

Arrays

Array block types describe fixed-length sequences of block data of homogeneous block-type. Given a block type object `elementType` and a non-negative integer `length`, it is possible to define a new array block-type object *t* using the `ArrayType` constructor:

```
t = new ArrayType(elementType, length)
```

The [[Convert]] operation converts an array-like ECMAScript value to block data by recursively converting its elements in order.

The [[Reify]] operation creates an array block object.

Given an array block-type object such as *t*, it is possible to construct new array blocks:

```
a = new t()
a = new t(val)
```

Elements of the array are accessible by getting or setting their index.

Structs

Struct block types describe fixed-length sequences of block data of heterogeneous block-types. Given an ECMAScript object `fields`, it is possible to define a new struct type object *t* using the `StructType` constructor:

```
t = new StructType(fields)
```

The implementation enumerates the own-properties of `fields` (in the standard enumeration order) to create the internal struct type descriptor.

The [[Convert]] operation converts an ECMAScript object to block data by reading each of the properties described by the struct type and converting their values.

The [[Reify]] operation creates a struct block object.

Given a struct block-type object such as *t*, it is possible to construct new struct blocks:

```
s = new t()
```

Each of the fields of the struct can be accessed or updated by name.

Block references

Struct and array blocks are encapsulated by objects. For some high-performance applications, it may be important to avoid the extra allocation of objects to access components of potentially very large block data structures.

For this reason, the spec also exposes a somewhat lower-level operation on struct and array objects, which allows a program to reuse a block object by updating its reference to point to a different block of the same block type. For example, in an array `a` of structs of type `T`, a struct object `s` of type `T` can be updated to point to subsequent elements of `a`:

```
for (i = 0; i < a.length; i++) {
    s.updateRef(a, i);
    // ...
}
```

As a convenience, `updateRef` can take more than one index or field name to refer to deeply-nested sub-blocks:

```
s.updateRef(a, i, "foo", "bar");
```

This convenience avoids the allocation of intermediate block objects without the need for the program to pre-allocate reference objects as "temporary pointers."

Given a struct or array type `t`, it is possible to create a new reference object via `t.ref()`. The object is initially not pointing to any block data, and its accessors and mutators throw exceptions until it is updated to refer to valid block data.

TODO: disallow `updateRef` on block objects that own their data?

[[strawman: records]]

ES Wiki

Records

Concise syntax for prototype-less, immutable data structures with named properties.

Syntax

```
RecordLiteral ::= "#" "{" ("..." AssignmentExpression)? "}"
                |   "#" "{" PropertyDataAssignment ("," PropertyDataAssignment)* (","
"..." AssignmentExpression)? "}"

PropertyDataAssignment ::= PropertyName ":" AssignmentExpression
```

Semantics

- `typeof` produces "`record`"
- enumeration always proceeds in lexicographic order of property names
- ellipsis form provides functional update: splices in own-properties of the result if `typeof` result is "`object`", or properties if result of `typeof` result is "`record`"
- record properties are immutable

[[strawman:
tuples]]

ES
Wiki

Trace: » function_to_string » simple_modules »
binary_data » records » tuples

Tuples

Concise syntax for prototype-less, immutable, dense sequences with indexed properties. Unlike arrays, tuples cannot have holes.

Syntax

```
TupleLiteral ::= "#" "[" (TupleElement ("," TupleElement)*)? "]"

TupleElement ::= AssignmentExpression
              |   "..." AssignmentExpression
```

Semantics

- `typeof` produces “`tuple`”
- enumeration always proceeds in index order
- length cannot be greater than 2^32 - 1
- `value.length` produces the tuple length
- ellipsis form splices in result of [[Get]] on each element from 0 to `length` - 1 of ellipsis argument
- elements are immutable

strawman/tuples.txt · Last modified: 2011/02/28 20:48 by dherman

[[strawman: array_comprehensions]]

ES Wiki

Trace: » simple_modules » binary_data » records » tuples » array_comprehensions

Overview

Array comprehensions were introduced in JavaScript 1.7. Comprehensions are a well-understood and popular language feature of list comprehensions, found in languages such as Python and Haskell, inspired by the mathematical notation of set comprehensions.

Array comprehensions are a convenient, declarative form for creating computed arrays with a literal syntax that reads naturally.

Examples

Filtering an array:

```
[ x for (x in a) if (x.color === 'blue') ]
```

Mapping an array:

```
[ square(x) for (x in values([1,2,3,4,5])) ]
```

Cartesian product:

```
[ [i,j] for (i in values(rows)) for (j in values(columns)) ]
```

Syntax

```
ArrayLiteral ::= ...
              |   "[" Expression ("for" "(" LHSExpression "in" Expression")")+ ("if" "("
Expression ")")? "]"
```

Translation

An array comprehension:

[*Expression$_0$* **for (** *LHSExpression$_1$* **in** *Expression$_1$* **)** … **for (** *LHSExpression$_n$* **) if (** *Expression* **)**$_{opt}$ **]**

can be defined by expansion to the expression:

let (*result* = **[]) {**
  for (let *LHSExpression$_1$* **in** *Expression$_1$* **) {**
    …
    for (let *LHSExpression$_n$* **in** *Expression$_n$* **) {**
      if (*Expression* **)**$_{opt}$
        *ArrayPush***(***result*, *Expression$_0$***)**;

```
            }
        }
    }
=> result
}
```

strawman/array_comprehensions.txt · Last modified: 2010/06/25 18:22 by dherman

[[strawman: generators]]

ES Wiki

Trace: » binary_data » records » tuples » array_comprehensions » generators

Table of Contents

- Overview
- Examples
- API
  - Generator objects
- Static semantics
  - Attribute grammar
  - Expressions
  - Statements
  - Functions
- Generator functions
  - Calling
  - Yielding
  - Delegating yield
  - Returning
- Generator methods
  - Method: next
  - Method: send
  - Method: throw
  - Method: close
- Generator objects
  - States
  - New completion type
  - Internal method: send
  - Internal method: throw
  - Internal method: close
  - Resuming generators
- References

Overview

First-class coroutines, represented as objects encapsulating suspended (single) function activations. Prior art: Python, Icon.

Examples

The “infinite” sequence of Fibonacci numbers (notwithstanding behavior around 2^{53}):

```
function fibonacci() {
    var [prev, curr] = [0, 1];
    for (;;) {
        [prev, curr] = [curr, prev + curr];
        yield curr;
    }
}
```

Generators are iterable:

```
for (let n in fibonacci()) {
    // truncate the sequence at 1000
    if (n > 1000)
        break;
    print(n);
}
```

Generators are iterators:

```
var seq = fibonacci();
print(seq.next()); // 1
print(seq.next()); // 2
print(seq.next()); // 3
print(seq.next()); // 5
print(seq.next()); // 8
```

API

`Function.isGenerator(f)`: returns `true` if `f` is a generator function, `false` if `f` is a function but not a generator function, and throws a `TypeError` otherwise.

Generator objects

Every generator object has the following internal properties:

- [[Prototype]] : the original value of `Object.prototype`

- [[Code]] : the code for the generator function body
- [[ExecutionContext]] : either **null** or an execution context
- [[Scope]] : the scope chain for the suspended execution context
- [[Handler]] : a standard generator handler for performing iteration
- [[State]] : “newborn”, “executing”, “suspended”, or “closed”
- [[Send]] : see semantics below
- [[Throw]] : see semantics below
- [[Close]] : see semantics below

There are four function objects, **send**, **next**, **throw**, and **close**. Every generator object has four properties, `send`, `next`, `throw`, and `close`, all respectively pointing to their corresponding function value. The functions' behavior is specified below.

Static semantics

Attribute grammar

The static semantics can be specified with an attribute grammar, similar to proper tail calls. The attribute grammar expresses two important properties of syntax nodes. First, it identifies which function nodes are *generator functions* (see below). Second, it specifies a static restriction that must be rejected at compile time.

Attribute	Directionality	Node type	Meaning
`isGeneratorFunction`	synthesized	functions	if **true**, the function is a generator function
`invalidGeneratorFunction`	synthesized	functions	if **true**, the function is an invalid generator function and must be rejected at compile-time
`mayYield`	synthesized	all	if **true**, contains a non-nested `yield` expression within the function body
`mayReturnResult`	synthesized	statements, clauses	if **true**, contains a non-nested `return` statement with an argument

Expressions

```
    E -> E1 , E2                                 E.mayYield = E1.mayYield ∨ E2.mayYield
    E -> E1 ? E2 : E3                            E.mayYield = E1.mayYield ∨ E2.mayYield ∨ E3.
mayYield
    E -> yield E1                                E.mayYield = true
    E -> yield* E1                               E.mayYield = true
    // etc
```

Statements

```
    S -> { S1 ... Sn }                          S.mayYield = S1.mayYield  ∨ ...  ∨ Sn.mayYield
                                                S.mayReturnResult = S1.mayReturnResult
∨ ...  ∨ Sn.mayReturnResult
    S -> E;                                     S.mayYield = E.mayYield
                                                S.mayReturnResult = false
    S -> return;                                S.mayYield = false
                                                S.mayReturnResult = false
    S -> return E;                              S.mayYield = E.mayYield
                                                S.mayReturnResult = true
    // etc
```

Functions

```
    F -> function f(x1,...,xn) S                F.isGeneratorFunction = S.mayYield
                                                F.mayYield = false
                                                F.invalidGeneratorFunction = F.
isGeneratorFunction  ∧ S.mayReturnResult
```

Generator functions

This section describes the semantics of *generator functions*, i.e., function nodes for which the `isGeneratorFunction` attribute is **true**.

Calling

Let `f` be a generator function. The semantics of a function call `f(x1, ..., xn)` is:

Let E = a new VariableEnvironment record with mappings for `x1` … `xn`
Let S = the current scope chain extended with E
Let V = a new generator object with
  [[Scope]] = S
  [[Code]] = f.[[Code]]
  [[ExecutionContext]] = **null**
  [[State]] = "newborn"
  [[Handler]] = the standard generator handler
Return V

Yielding

The semantics of evaluating an expression of the form `yield e` is:

Let V ?= Evaluate(e)
Let K = the current execution context
Let O = K.currentGenerator
O.[[ExecutionContext]] := K
O.[[State]] := "suspended"
Pop the current execution context
Return (**normal**, V, **null**)

Delegating yield

The `yield*` operator delegates to another generator. This provides a convenient mechanism for composing generators.

The expression `yield* <<expr>>` is equivalent to:

```
let (g = <<expr>>) {
    let received = void 0, send = true;
    try {
        while (true) {
            let next = send ? g.send(received) : g.throw(received);
            try {
                received = yield next;
                send = true;
            } catch (e) {
                received = e;
                send = false;
            }
        }
    } catch (e if e === StopIteration) {
    } finally {
        try { g.close(); } catch (ignored) { }
    }
    void 0;
}
```

This is similar to a `for-in` loop over the generator, except that it propagates

Returning

The semantics of returning from a generator function is:

> Let K = the current execution context
> Let O = K.currentGenerator
> O.[[State]] := "closed"
> Throw StopIteration

See iterators for a discussion of `StopIteration`.

Generator methods

Method: next

The **next** function's behavior is:

> If **this** is not a generator object, Throw Error
> Call **this**.[[Send]] with single argument **undefined**
> Return the result

Method: send

The **send** function's behavior is:

> If **this** is not a generator object, Throw Error
> Call **this**.[[Send]] with the first argument
> Return the result

Method: throw

The **throw** function's behavior is:

> If **this** is not a generator object, Throw Error
> Call **this**.[[Throw]] with the first argument
> Return the result

Method: close

The **close** function's behavior is:

If **this** is not a generator object, Throw Error
Call **this**.[[Close]] with no arguments
Return the result

Generator objects

States

A generator object can be in one of four states:

- "newborn": G.[[Code]] != **null** $\wedge$G.[[ExecutionContext]] = **null**
- "executing": G.[[Code]] = **null** $\wedge$G.[[ExecutionContext]] != **null** $\wedge$G.[[ExecutionContext]] is the current execution context
- "suspended": G.[[Code]] = **null** $\wedge$G.[[ExecutionContext]] != **null** $\wedge$G.[[ExecutionContext]] is not the current execution context
- "closed": G.[[Code]] = **null** $\wedge$G.[[ExecutionContext]] = **null**

It is never the case that G.[[Code]] != **null** $\wedge$G.[[ExecutionContext]] != **null**.

New completion type

This spec introduces a new completion type: **close**. This is used for terminating a suspended generator early. The **close** completion type informs the generator to exit early from its activation (roughly as if via `return`), running any active `finally` blocks first before completing.

Internal method: send

G.[[Send]]

Let State = G.[[State]]
If State = "executing" Throw Error
If State = "closed" Throw Error
Let X be the first argument
If State = "newborn"
    If X != **undefined** Throw TypeError
    Let K = a new execution context as for a function call
    K.currentGenerator := G
    K.scopeChain := G.[[Scope]]
    Push K onto the stack
    Return *Execute*(G.[[Code]])
G.[[State]] := "executing"
Let Result = *Resume*(G.[[ExecutionContext]], **normal**, X)
Return Result

Internal method: throw

G.[[Throw]]

Let State = G.[[State]]
If State = "executing" Throw Error

If State = "closed" Throw Error
Let X be the first argument
If State = "newborn"
G.[[State]] := "closed"
G.[[Code]] := **null**
Return (**error**, X, **null**)
G.[[State]] := "executing"
Let Result = *Resume*(G.[[ExecutionContext]], **error**, X)
Return Result

Internal method: close

G.[[Close]]

Let State = G.[[State]]
If State = "executing" Throw Error
If State = "closed" Return **undefined**
If State = "newborn"
G.[[State]] := "closed"
G.[[Code]] := **null**
Return (**normal**, **undefined**, **null**)
G.[[State]] := "executing"
Let Result = *Resume*(G.[[ExecutionContext]], **close**, **undefined**)
G.[[State]] := "closed"
Return Result

Resuming generators

Operation *Resume*(K, completionType, V)

Push K onto the execution context stack
Let G = K.currentGenerator
Set the current scope chain to G.[[Scope]]
Continue executing K as if its last expression produced (completionType, V, **null**)

References

- Generators in SpiderMonkey
- PEP 255, "Simple generators"
- PEP 380, "Syntax for delegating to a sub-generator"—their desugaring for `yield from` (their syntax for `yield*`) seems broken to me (we should look into this)

RSS XML FEED | CC LICENSED | $1 DONATE | PHP POWERED | W3C XHTML 1.0 | W3C CSS | DOKUWIKI

[[strawman: generator_expressions]]

ES Wiki

Trace: » records » tuples » array_comprehensions » generators » generator_expressions

Table of Contents

- Overview
- Examples
- Syntax
- Translation
- Notes

Overview

Generator expressions were introduced in JavaScript 1.8. Generator expressions are a convenient, declarative form for creating generators with a syntax based on array comprehensions. Generator expressions also provide a convenient refactoring pattern, making it easy to switch between eager and on-demand generation of values in a sequence simply by changing the bracketing.

Examples

Extracting pages on demand from an array of URL's:

```
(xhrGet(url) for (url in getURLs()))
```

Filtering a sequence:

```
(x for (x in generateValues()) if (x.color === 'blue'))
```

Lazy cartesian product

```
(xhrGet(row, column) for (row in rows()) for (column in columns()))
```

Syntax

```
PrimaryExpression ::= ...
                   |   "(" Expression ("for" "(" LHSExpression "in" Expression")")+ ("if"
"(" Expression ")")? ")"
```

Translation

A generator expression:

(*Expression*$_0$ **for (** *LHSExpression*$_1$ **in** *Expression*$_1$ **)** ... **for (** *LHSExpression*$_n$ **) if (** *Expression* **)**$_{opt}$ **)**

can be defined by expansion to the expression:

(function () {
    for (let *LHSExpression*$_1$ **in** *Expression*$_1$ **) {**
        ...
        for (let *LHSExpression*$_n$ **in** *Expression*$_n$ **) {**
            if (*Expression* **)**$_{opt}$
                yield (*Expression*$_0$**);**
            }

```
        }
    }
})()
```

Notes

Background motivation for the syntactic sugar afforded by generator expressions:

- Peter Norvig's Sudoku solver based on constraint propagation, written in Python
- My port of Peter's solver to JS1.8

The critical uses of generator expressions, e.g., the actual parameter to `all` in:

```
    if (all(eliminate(values, s, d2) for (d2 in values[s]) if (d2 != d)))
        return values;
```

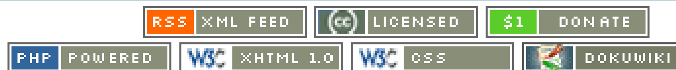
can only be desugared to generation function applications or an equivalent lazy iterator construct. They cannot be replaced with array comprehensions or any such eager construct without the solver taking exponential time and space creating eagerly populated arrays where it would have stopped early using lazy generator expressions, thanks to constraint propagation. Note how `all` is defined:

```
function all(seq) {
    for (let e in seq)
        if (!e)
            return false;
    return true;
}
```

so as to stop as soon as a value in the iterated sequence is falsy.

The JS1.8 version has some XXX comments and helper functions that show where methods such as the Array extras (`Array.prototype.every` instead of the custom `all` shown above, e.g.) are not iterator-friendly. This suggests the need for more generic methods that abstract over arrays and iterators.

— *Brendan Eich 2010/06/27 19:47*



[[strawman:
pattern_matching]]

ES
Wiki

Table of Contents

- Pattern matching
- Patterns
  - Syntax
  - Static validation
  - Refutable semantics
  - Irrefutable semantics
- Pattern matching switch
  - Syntax
  - Semantics
- Destructuring assignment
- Irrefutable matching contexts

Pattern matching

JS 1.7 destructuring is almost pattern matching, but it lacks the refutable semantics needed to support a conditional pattern matching form. This strawman adds a notion of *refutable matching* and adds certain contexts to the language where matching is interpreted refutably. Other contexts perform a looser, irrefutable matching.

Patterns

Syntax

This is written in parameterized BNF for conciseness. (Think of it as a grammar macro which could be pre-expanded to produce the full grammar.)

The grammar is factored into *PrimaryPattern* and *Pattern*. The former does not admit outermost parentheses; this disambiguates between e.g. `let` expressions and `let` declarations.

```
PrimaryPattern(refutable) ::= "*"
                           |  [if refutable] Literal
                           |  Identifier(refutable)
                           |  ArrayPattern(refutable)
                           |  TuplePattern(refutable)
                           |  ObjectPattern(refutable)
                           |  RecordPattern(refutable)
                           |  [if refutable] PrimaryPattern(refutable) "if"
AssignmentExpression

Pattern(refutable) ::= PrimaryPattern(refutable)
                    |  "(" Pattern(refutable) ")"
                    |  [if refutable] "(" Pattern(refutable) ")" "if"
AssignmentExpression

ArrayPattern(refutable) ::= "[" (EllipsisPattern(refutable) ",")? (ElementPattern
(refutable) ("," ElementPattern(refutable))*)? "]"
                         |  "[" ElementPattern(refutable) ("," ElementPattern(refutable))
* ("," EllipsisPattern(refutable)) "]"

EllipsisPattern(refutable) ::= "..." Pattern(refutable)

TuplePattern(refutable) ::= "#" "[" EllipsisPattern(refutable) "]"
                         |  "#" "[" (EllipsisPattern(refutable) ",")? (ElementPattern
(refutable) ("," ElementPattern(refutable))*)? "]"
                         |  "#" "[" ElementPattern(refutable) ("," ElementPattern
(refutable))* ("," EllipsisPattern(refutable))? "]"

ElementPattern(refutable) ::= Pattern(refutable)?

ObjectPattern(refutable) ::= "{" (PropertyPattern(refutable) ("," PropertyPattern
(refutable))*)? "}"

RecordPattern(refutable) ::= "#" "{" EllipsisPattern(refutable) "}"
```

```
                              |   "#" "{" PropertyPattern(refutable) ("," PropertyPattern
(refutable))* ("," EllipsisPattern(refutable))? "}"
                              |   "#" "{" (PropertyPattern(refutable) ("," PropertyPattern
(refutable))*)? "}"

PropertyPattern(refutable) ::= PropertyName ":" Pattern(refutable)

GuardedPattern(refutable) ::= Pattern(refutable) "if" AssignmentExpression
```

Static validation

If a pattern binds the same variable name in multiple positions, a compile-time error must be raised.

Refutable semantics

Roughly:

- “*”: match anything, bind nothing
- Literal: match if the value is equal to the value of the literal (using ===), bind nothing
- Identifier: match anything, bind the value to the identifier
- ArrayPattern:
  - if `typeof` the value is not “object”, fail
  - if value has no `length` property, fail
  - let len = ToUint32(`length`)
  - if pattern starts with ellipsis:
    - let n be the number of elements (empty or non-empty) in the pattern after the ellipsis
    - for each i’th non-empty element in the pattern, match pattern[i] against value[length - n + i]
  - else if pattern ends with ellipsis:
    - let n be the index of the last (empty or non-empty) element in the pattern
    - if length ⇐n, fail
    -

for each i'th non-empty element in the pattern, match pattern[i] against value[i]

- let a = slice.call(value, n + 1) where slice is the original value of Array.prototype.slice
- match a against the sub-pattern of the ellipsis

- otherwise:
  - let n be the index of the last non-empty element in the pattern
  - if length ⇐n, fail
  - for each i'th non-empty element in the pattern, match pattern[i] against value[i]
- match if all the sub-patterns matched, and bind all their collected bindings

- TuplePattern: similar to array pattern but require it to be a tuple
- ObjectPattern:
  - if `typeof` the value is not "object", fail
  - test that each property pattern is in the object and match its [[Get]] result agains its sub-pattern
- RecordPattern: similar to object pattern, but require it to be a record, and also match a new record of remaining elements against ellipsis sub-pattern
- GuardedPattern: match subpattern and create bindings, then run the guard expression with just those bindings (not any from surrounding pattern) in local scope, and fail if the result is falsey

Note that any time any of these operations triggers code that raises an exception, the exception is *not* converted to a failure; the exception propagates as the result of the pattern match.

Irrefutable semantics

Roughly: do the same as the refutable semantics, but if failure occurs at any point, bind all variables to the undefined value and succeed.

Pattern matching switch

This section adds a switch expression form for pattern matching that is grammatically unambiguous from switch

statements, even in the context of a statement expression.

Syntax

```
SwitchExpression ::= "switch" "(" Expression ")" "{" MatchClause+ DefaultBlockClause? "}"

MatchClause ::= "match" Pattern(true) Block

DefaultBlockClause ::= "default" Block
```

Semantics

Perform each match refutably in sequence until one of them succeeds, then evaluate the RHS expression with the environment extended with the bindings of the match. If at any point a pattern match raises an exception, that exception propagates as the result of the switch expression. The RHS expressions are block statements but are evaluated for their completion value.

Destructuring assignment

Destructuring assignment is parsed as an `LHSExpression` but then post-processed as a `Pattern(`**`false`**`)`. Note that this non-terminal represents a subset of `LHSExpression`.

Irrefutable matching contexts

Binding contexts which are not conditional use the irrefutable pattern syntax and matching semantics. These include:

- formal parameters (`Pattern(`**`false`**`)`)
- `let`/`var`/`const` bindings (`PrimaryPattern(`**`false`**`)`)
- assignment (`LHSExpression`, reinterpreted as `Pattern(`**`false`**`)`)

strawman/pattern_matching.txt · Last modified: 2011/02/28 23:58 by dherman

[[strawman:catch_guards]]

Rationale

Programs often need to catch exceptions conditionally. In portable ES, they have to write it like so:

```
try {
    ...
} catch (e) {
    if (p(e)) {
        ...
    } else {
        throw e;
    }
}
```

where `p(e)` is any predicate.

SpiderMonkey supports a simple syntax for *catch guards*, which abstract over this pattern. With catch guards the above could be rewritten:

```
try {
    ...
} catch (e if p(e)) {
    ...
}
```

This is simpler, more readable, and less error-prone (no forgetting to rethrow). It also means that if implementations choose to attach meta-data at `throw` points, there's no danger in overwriting this metadata by explicit rethrows.

This strawman exploits the pattern matching extensions to generalize catch blocks even further to use *refutable pattern matching*. Each catch clause takes a pattern and attempts to match it; if the match fails, the catch clause is not chosen and exception propagates, either to the next clause or, if there aren't any more, to continue throwing.

Syntax

The syntax of `try-catch[-finally]` statements changes to:

```
TryStatement ::= "try" Block Finally
              |   "try" Block Catch+ Finally?

Catch        ::= "catch" "(" Pattern(true) ")" Block
Finally      ::= "finally" Block
```

Restrictions

A pattern is treated as a *catchall* if it is of the form "`*`" or `Identifier`. It is a compile-time error for a catch clause with a catchall pattern to be followed by more clauses.

Translation

The translations in this section assume an invisible identifier `$tmp` that is not visible even to direct `eval`.

A statement of the form:

```
try B0
catch (P1) B1
...
catch (Pn) Bn
[finally BB]
```

translates to:

```
try B0
catch ($tmp) {
    case P1 B1
    ...
    case Pn Bn
    default { throw $tmp; }
}
[finally BB]
```

where, of course, the resulting `catch` is ES5 `catch` (i.e., the translation doesn't recur!).

LICENSED

$1 DONATE

XHTML 1.0

CSS

DOKUWIKI

[[strawman:completion_reform]]

ES Wiki

Trace: » generators » generator_expressions » pattern_matching » catch_guards » completion_reform

Completion value

The existing specs describe a completion value that every statement can produce, but some statements do not *necessarily* produce a completion value. This means that it's not always statically predictable which sub-statement of a compound statement will produce the completion. For example:

```
{
    41;           // completion is 41
    if (...) 42;  // either no completion or 42
}                 // block's completion is either 41 or 42
```

```
{
    41;              // completion is 41
    while (...) 42;  // either no completion (if 0 iterations) or 42
}                    // block's completion is either 41 or 42
```

For proper tail calls, the completion position will be important for identifying tail position in expression forms with statement bodies (e.g., shorter function syntax, pattern matching, switch expressions, and completion let).

Completion for conditionally executed statements

This strawman proposes breaking compatibility of the definition of completion values, such that completion position becomes statically predictable. The basic idea is that these conditional cases would produce the undefined value as their completion, rather than no completion.

```
{
    41;           // completion is 41
    if (...) 42;  // either undefined or 42
}                 // block's completion is either undefined or 42
```

```
{
```

```
    41;               // completion is 41
    while (...) 42;   // either undefined (if 0 iterations) or 42
}                     // block's completion is either undefined or 42
```

Backwards compatibility

While this is backwards-incompatible, the completion value only showed up in ES5 and earlier as the result of `eval`. The hope is that this is an obscure enough corner case of completion values, that it wouldn't be likely to break many programs.

I like it. The strange "nothing means previous statement's completion value" semantics were a just-so story from JS1.0 that we codified in ES1.

Can we get away with this kind of migration-tax (remember, only five fingers of fate to use up)? We probably can IMHO, and anyway we should test and scan the web harder to check before making a hard decision.

At the least, I'd rather we have this completion-value semantics for sharp-functions and other new syntactic forms than the bad old completion semantics.

— *Brendan Eich 2011/03/01 00:24*

RSS XML FEED

[[strawman:completion_let]]

ES Wiki

Completion-let

A simple expression form for doing local bindings.

Syntax

```
Expression ::= ... | "let" "(" (LetBinding ("," LetBinding)*)? ")" Block
LetBinding ::= Pattern(false) ("=" AssignmentExpression)?
```

See pattern matching for the definition of `Pattern`.

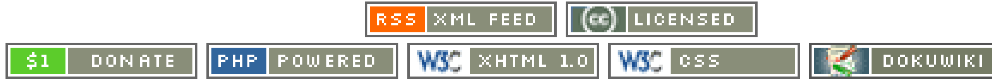
Scope

- all variable bindings in LetHead are in scope in the body
- each variable binding in scope in subsequent head expressions? (let*) or not? (let)

Evaluation

- evaluate the RHS expressions in left-to-right order
- extend scope chain
- variables with no RHS are bound to the `undefined` value
- evaluate body
- result value is completion of the block

strawman/completion_let.txt · Last modified: 2011/03/01 00:02 by dherman



[[strawman: proxy_derived_traps]]

More derived traps

David Bruant noted that the `getPropertyDescriptor` and `getPropertyNames` traps can become derived. They are currently specced as fundamental traps.

Since fundamental traps are required and derived traps are optional, and since derived traps add no new complexity beyond the fundamental traps, there is an incentive to keep the number of fundamental traps as small as possible.

The `getPropertyDescriptor` and `getPropertyNames` traps can be defined in terms of `getOwnPropertyDescriptor` and `getOwnPropertyNames` respectively, and by walking the proxy's prototype chain:

```
getPropertyDescriptor: function(proxy, name) {
  var pd = Object.getOwnPropertyDescriptor(proxy, name); // calls
getOwnPropertyDescriptor trap
  var proto = Object.getPrototypeOf(proxy);
  while (pd === undefined && proto !== null) {
    pd = Object.getOwnPropertyDescriptor(proto, name);
    proto = Object.getPrototypeOf(proto);
  }
  return pd;
}

getPropertyNames: function(proxy, name) {
  var props = Object.getOwnPropertyNames(proxy); // calls getOwnPropertyNames trap
  var proto = Object.getPrototypeOf(proxy);
  while (proto !== null) {
    props = props.concat(Object.getOwnPropertyNames(proto));
    proto = Object.getPrototypeOf(proto);
  }
  // remove duplicate property names from props (not shown)
  return props;
}
```

Note that the above definitions assume that `getPropertyDescriptor` and `getPropertyNames` get passed the `proxy` for which they are intercepting, which is proposed in a separate strawman.

Currently, handlers don't have a way of accessing the proxy they're currently intercepting, so they cannot get at a proxy's prototype. The Proxy implementation does have access to these parts and can perform the correct default behavior, it's just that this default behavior cannot be expressed fully in Javascript code, which is a big difference compared to all existing derived traps.

MarkM suggests refining our notion of derived traps by distinguishing ("optional" vs "mandatory") versus ("fundamental" vs "derived") traps. Optional vs. mandatory indicates whether or not the trap must be present for the proxy to work, while fundamental vs. derived indicates whether or not the default behavior for a missing trap can be defined in Javascript itself in terms of other traps.

	Optional	**Mandatory**

Fundamental	`getPropertyDescriptor` and `getPropertyNames`	all existing “fundamental” traps
Derived	all other existing “derived” traps	none

For developers, really the only distinction that matters is optional vs. mandatory. The fundamental vs. derived distinction is there primarily to help us, spec. writers, distinguish the traps. Again, if `proxy` would become a parameter to the above two derived traps, this discussion is moot and the table again collapses into our current simple distinction of fundamental vs derived traps, without the need to special-case `getPropertyDescriptor` and `getPropertyNames`.

— *Tom Van Cutsem 2011/02/28 06:09*

References

- Discussion thread on es-discuss.

strawman/proxy_derived_traps.txt · Last modified: 2011/02/28 20:32 by tomvc

[[strawman: handler_access_to_proxy]]

ES Wiki

Trace: » catch_guards » completion_reform » completion_let » proxy_derived_traps » handler_access_to_proxy

Table of Contents

- Handler access to proxies
  - Extended API
    - Proxy as optional argument
    - Proxy as additional argument
    - Proxy as argument only for particular traps
  - Reservations
- References

Handler access to proxies

We should consider the possibility of extending the Proxy Handler API such that handlers get access to the proxy for which they are currently intercepting. Motivating use cases:

- A handler shared by many proxy instances may want to identify the proxy for which it is currently "servicing" an operation. For instance, the shared handler could use a WeakMap keyed by the proxy's identity to store per-proxy state.
- Without access to the proxy, the handler has no way of accessing the prototype passed to `Proxy.create`. It also cannot reliably distinguish whether it is servicing an object or a function proxy. When given access to a proxy, the handler could perform `Object.getPrototypeOf(proxy)`, `typeof proxy` and `proxy instanceof Fun` to get at this data (credit goes to David Bruant)
- If the (currently fundamental) `getPropertyDescriptor` and `getPropertyNames` traps would have access to the `proxy`, they can easily be turned into derived traps, as indicated in this strawman.

Extended API

The easiest way to allow a handler access to the proxy it is currently servicing is to pass the proxy as an additional argument to the handler traps. From here, there are multiple routes to take:

1. Add `proxy` as an optional last argument to all traps.
2. Add `proxy` as an argument at the most appropriate position for each trap.
3. Add `proxy` only as an argument to the `getPropertyDescriptor` and `getPropertyNames` traps (for the purpose of defining their derived behavior).

Proxy as optional argument

We could add a `proxy` parameter as an optional last argument to all existing traps.

Pro:

- regular API
- traps that aren't interested in accessing the proxy can simply ignore it

Con:

- adding an optional last argument restricts our options if one of the Object API methods would change in the future. If e.g. `Object.getOwnPropertyDescriptor` takes an extra argument in a later edition, how can we reconcile this with existing code that assumes that the second parameter is the proxy? (requires refactoring)

- for some traps, getting the proxy as the last argument is counter-intuitive.

For example, the trap `getOwnPropertyDescriptor` is triggered by code like:

```
Object.getOwnPropertyDescriptor(proxy, name)
```

Yet the order in which the params are passed to the trap is reversed:

```
getOwnPropertyDescriptor: function(name, proxy) {...}
```

Passing `proxy` as the last argument is odd in this way for `get{Own}PropertyDescriptor`, `defineProperty`, `delete`, `hasOwn`. It is OK for `get{Own}PropertyNames`, `keys`, `fix` (freeze/seal/preventExtensions), `has`, `enumerate`. The `get` and `set` traps already have implicit access to `proxy` via `receiver` (which is either the `proxy` or an object inheriting from the `proxy`). The `proxy` argument could be passed either as an extra first argument or as an extra last argument.

Proxy as additional argument

We could add a `proxy` parameter as an extra argument to all existing traps. Depending on the trap, the argument is added either as a mandatory argument or as an optional trailing argument.

Pro:

- The position of `proxy` is consistent with its position in the intercepted code.

Con:

- Less consistent.

- Some traps can't ignore the `proxy` parameter.

Below is a proposed updated API (when the `proxy` parameter is optional, it is enclosed in square brackets):

```
// fundamental traps
getOwnPropertyDescriptor: function(proxy, name) -> PropertyDescriptor | undefined //
Object.getOwnPropertyDescriptor(proxy, name)
getOwnPropertyNames:      function([proxy]) -> [ string ]                         //
Object.getOwnPropertyNames(proxy)
defineProperty:           function(proxy, name, propertyDescriptor) -> any        //
Object.defineProperty(proxy,name,pd)
delete:                   function(proxy, name) -> boolean                        //
delete proxy.name
fix:                      function([proxy]) -> { string: PropertyDescriptor }     //
Object.{freeze|seal|preventExtensions}(proxy)
                                               | undefined
// derived traps
getPropertyDescriptor:    function(proxy, name) -> PropertyDescriptor | undefined //
```

```
Object.getPropertyDescriptor(proxy, name)   (not in ES5)
getPropertyNames:         function([proxy]) -> [ string ]                       //
Object.getPropertyNames(proxy)              (not in ES5)
has:        function(name, [proxy]) -> boolean                  // name in proxy
hasOwn:     function(proxy, name) -> boolean                   // ({}).hasOwnProperty.call
(proxy, name)
get:        function(receiver, name, [proxy]) -> any            // receiver.name
set:        function(receiver, name, val, [proxy]) -> boolean   // receiver.name = val
enumerate:  function([proxy]) -> [string]                      // for (name in proxy)
(return array of enumerable own and inherited properties)
keys:       function([proxy]) -> [string]                      // Object.keys(proxy)
(return array of enumerable own properties only)
```

Other ways to decide on the optionality of `proxy`:

- make `proxy` mandatory for the traps that trap methods on `Object`, and optional (trailing) for all others.
- make it optional for derived traps, mandatory for fundamental traps.

Proxy as argument only for particular traps

Only add the `proxy` parameter to the `getPropertyDescriptor` and `getPropertyNames` traps. Pro: keeps the overall API simple while still allowing derived behavior for these traps. Con: inconsistent, doesn't cater to all motivating use cases.

Reservations

- Should a handler really be able to distinguish whether it is handling an object or a function proxy?
- Having access to the proxy by default increases the risk for infinite recursion hazards.

— *Tom Van Cutsem 2011/02/28 06:10*

References

- Discussion thread on es-discuss (the idea originated while discussing the default behavior of the `getPropertyDescriptor` and `getPropertyNames` traps).

strawman/handler_access_to_proxy.txt · Last modified: 2011/02/28 14:53 by tomvc

Table of Contents

- Derived Traps of Default Forwarding Handler
  - Context
  - Analysis
  - Proposal
- References

Derived Traps of Default Forwarding Handler

The default Proxy forwarding handler provides an implementation for all fundamental and all derived traps. For the derived traps, two possible default implementations are possible, and it is not entirely obvious which one is better.

Context

David Bruant, while experimenting with proxied Arrays, came across the following situation: his proxied Array uses a default forwarding handler to forward all operations to a target Array instance, but overrides the `defineProperty` trap. However, assignments to the array of the form `proxiedArray[i] = val` did not trigger his overridden `defineProperty` trap, since the default forwarding handler provides a derived `set` trap that just forwards the operation to the wrapped array.

To make this use case work, one has to either override `set` in conjunction with `defineProperty`, or delete the default `set` trap of `Proxy.Handler`. When the derived trap is deleted, the implementation correctly falls back on the (overridden) fundamental `defineProperty` trap.

```
var target = {...};
var h = new Proxy.Handler(target);
h.defineProperty = function(name, pd) {...}; // override a fundamental trap
var p = Proxy.create(h);

p["foo"] = "bar"; // triggers default forwarding 'set' trap. Sets "foo" on target.
// programmer may have expected this to trigger overridden fundamental trap instead

delete Object.getPrototypeOf(h).set;
p["foo"] = "bar"; // triggers overridden defineProperty trap
```

Analysis

There are two possible default implementations for the derived traps of the default forwarding handler:

- “forwarding” semantics: forward the derived operation to the `target` (that is how they are currently specified)
- “fallback” semantics: implement the “default” semantics in terms of the fundamental forwarding traps (or equivalently, state that the default forwarding handler does not define any derived traps)

The issue with “forwarding semantics” is that overriding a fundamental trap requires developers to override all dependent derived traps in sync. The relationship between fundamental and dependent derived traps may not be immediately obvious to developers, so this could lead to surprising behavior in practice. OTOH, in many situations developers will want to override the derived traps anyway to allow for a more efficient implementation.

The issue with “fallback semantics” is that if the `target` object to which the proxy forwards is itself a proxy `p2`, then `p2`'s derived traps will never be called. Instead, only `p2`'s fundamental traps will be called. Things won't break, but it is

suboptimal if p2 has ad hoc (presumably more efficient) implementations for its derived traps. Presumably, even if “target” is a native object, “forwarding semantics” is more efficient than “fallback semantics”.

```
var p2 = Proxy.create(h2);
var h = new Proxy.Handler(p2); // p forwards to p2
var p = Proxy.create(h);

p["foo"] = "bar"; // assuming "h.set" is undefined, this triggers h.defineProperty
// which in turn calls Object.defineProperty(p2, "foo", {value: "bar"});
// which in turn triggers h2's "defineProperty" trap, not its "set" trap
```

Proposal

It is difficult to come up with a solution that provides support for both forwarding and fallback semantics.

One proposal is to:

1. Add a `Proxy.BaseHandler` that defines no derived traps, only fundamental traps (i.e. `Proxy.BaseHandler` supports fallback semantics).
2. Modify `Proxy.Handler` so that it inherits its fundamental traps from `BaseHandler` and adds forwarding derived traps (i.e. `Proxy.Handler` then supports forwarding semantics).

Depending on the required semantics, developers can make their custom handler inherit from either one. The question is whether the advantage of choice outweighs the complexity cost of this proposed API.

Another option is to simply live with the dependency between fundamental and derived traps and to properly document the relationships between traps, and the hazards of not overriding the traps in sync.

— *Tom Van Cutsem 2011/02/28 07:37*

References

- Discussion thread on es-discuss.

[[strawman:function_proxy_prototype]]

ES Wiki

Custom prototypes for function proxies

Programmers want to be able to create subtypes of the core built-in types like `Array` and `Function`. The binary data spec in particular wants callable `Function` subtypes.

This strawman proposes adding an optional fourth argument to `Proxy.createFunction` specifying a custom prototype object for the function proxy. If the argument is unspecified or given the undefined value, the prototype defaults to `Function.prototype`.

In order to preserve the spec invariant that callable objects are subtypes of `Function` (and testing `instanceof Function` is sufficient to determine whether an object is callable), `Proxy.createFunction` would test that the given prototype is an `instanceof Function`, and throw an exception if the test returns `false`.

Same comment as for name_property_of_functions – passing `otherWindow.Function.prototype` would be disallowed by this proposal since `instanceof` starts one hop up the prototype chain before testing.

— *Brendan Eich 2011/02/28 21:56*

References

- Kangax on extending core pseudo-classes
- binary data
- discussion on es-discuss

RSS
XML FEED

LICENSED

$1
DONATE

PHP
POWERED

W3C
XHTML 1.0

W3C
CSS

DOKUWIKI

Completion-let

A simple expression form for doing local bindings.

Syntax

```
Expression ::= ... | "let" "(" (LetBinding ("," LetBinding)*)? ")" Block
LetBinding ::= Pattern(false) ("=" AssignmentExpression)?
```

See pattern matching for the definition of `Pattern`.

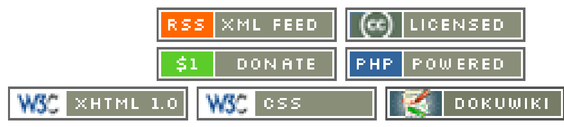
Scope

- all variable bindings in LetHead are in scope in the body
- each variable binding in scope in subsequent head expressions? (let*) or not? (let)

Evaluation

- evaluate the RHS expressions in left-to-right order
- extend scope chain
- variables with no RHS are bound to the `undefined` value
- evaluate body
- result value is completion of the block

strawman/completion_let.txt · Last modified: 2011/03/01 00:02 by dherman


RSS XML FEED
LICENSED
$1 DONATE
PHP POWERED
W3C XHTML 1.0
W3C CSS
DOKUWIKI

[[strawman: name_property_of_functions]]

ES Wiki

Precedent

- `(new Function).name === "anonymous"` wanted by the Web, according to this webkit bug
- `(function(){}).name === ""` may be wanted too, we suspect – we aren't sure, though, so this behavior of some browser-based implementations is not strong precedent
- `function f(){} assert(f.name === "f")` is implemented by several browsers, with name not writable and not configurable
- Most browsers that implement `name` for functions use it in the result of `toString` as the function identifier (detailed results of testing by Allen)
- `toString` according to ES3 is not well-defined for anonymous function expressions
- Writable displayName property used for console logging in webkit

Goals

These conflict if achieved for all functions.

- Support *de facto* standards per above precedent
- Avoid adding unnecessary properties
- Keep `name` and `toString` results consistent
- Automatically derive names for synthesized functions such as get, set, and bind functions
  - e.g., for `obj = {get prop() { return 42; }}` extracting the getter for `prop` would recover a

function `g` such that `g.name === "get prop"` in one proposal

- Allow some functions to be given arbitrary names, e.g. by code generators (Objective-J)

Proposals

These are not mutually exclusive.

- For function declarations and named function expressions, create a non-writable, non-configurable `name` property whose value is the function's identifier as a string
- For anonymous function expressions, create no `name` property at all
- `Function.prototype` would have no `name` property (in some implementations it is a function created as if by evaluating an anonymous function expression taking no arguments and having an empty body)
- Add `Function.create(`*name*`, `*params...*`, `*body*`)` per Maciej's suggestion
- As an alternative to a `name` property for functions, standardize the WebKit `displayName` property.

A variation

I like the spirit of Maciej's proposal, but I don't like repeating the string-pasting, `eval`-like interface of the `Function` constructor. Here's a variation:

```
Function.create(name, call[, construct[, proto]])
```

Creates a function with the given display name, call behavior, optional construct behavior (which defaults to the usual call-with-fresh-object behavior), and optional prototype (which defaults to the original value of `Function.prototype`).

```
Function.getDisplayNameOf(f)
```

Returns the display name of a function.

Some more detail:

- Every function has an internal [[DisplayName]] property
- The semantics automatically infers this property for function literals in at least the following contexts:

- function declarations: the declared name is the inferred display name
- named function expressions: the function name is the inferred display name
- `var/let/const` declarations that assign function literals: the variable name is the inferred display name
- object literals that assign function literals to property names: the property name is the inferred display name

Sample implementation:

```
(function() {
    var names = new WeakMap();

    Function.create = function(name, call, construct, fproto) {
        if (!fproto)
            fproto = Function.prototype;
        if (fproto !== Function.prototype && !(fproto instanceof Function))
            throw new TypeError("expected instance of Function, got " + fproto);
        var f;
        if (!construct) {
            construct = function() {
                var oproto = f.prototype;
                if (typeof oproto !== "object")
                    oproto = Object.prototype;
                var newborn = Object.create(oproto, {});
                var result = Function.prototype.apply.call(call, arguments);
                return typeof result === "object" ? result : newborn;
            }
        }
        var handler = Proxy.Handler(Object.create(fproto, {}));
        f = Proxy.createFunction(handler, call, construct, fproto);
        return f;
    };

    Function.getDisplayNameOf = function(f) {
        return names.get(f);
    };
})();
```

— *Dave Herman 2011/02/24 06:00*

The major objection to losing the "compile this string as the function body" `Function` design on which Maciej built comes from the use-case: Objective J compilation and similar want to create a function per "method", not two (one returned by this variation and its `call` function). Maciej's `Function.create` proposal was simply a `Function` variant that allowed the intrinsic name to be specified. This variation is more like a proxy-maker.

A minor objection:

```
Function.prototype instanceof Function  // => false
```

This means you cannot pass `otherWindow.Function.prototype` as the `proto` parameter.

— *Brendan Eich 2011/02/28 21:34*

[[strawman:
paren_free]]

ES
Wiki

Trace: » handler_access_to_proxy »
derived_traps_forwarding_handler » function_proxy_prototype » completion_let » name_property_of_functions

This topic does not exist yet

You've followed a link to a topic that doesn't exist yet. You can create it by using the `Create this page` button.

RSS XML FEED

[[strawman:multiple_globals]]

ES Wiki

Trace: » derived_traps_forwarding_handler » function_proxy_prototype » completion_let » name_property_of_functions » multiple_globals

Table of Contents

- Multiple globals
  - Existence of multiple globals
  - Global objects and non-method calls
  - Global objects and eval
  - Global objects and navigation
  - Terminology question

Multiple globals

The ECMAScript spec has never said anything about the presence of multiple global objects interacting with one another, but this has long been the reality on the web. Browsers have not always been interoperable in this area, so it deserves standardization. Some amount of backwards incompatibility may be acceptable, since some of the cases may be fairly rare in practice.

Existence of multiple globals

Where historically the spec refers to “the global object,” this needs to be made more precise by specifying *which* global object.

In past versions of the standard, every closure contains a scope chain that ends with the (a) global object. In SpiderMonkey terminology, this is the closure’s “parent.” We will use the term “global context.”

Since Harmony may not include the global object in the scope chain, this concept needs to be generalized to encompass either the global object (for legacy mode) or the module loader context associated with the scope chain.

Global objects and non-method calls

When a function is called as a non-method, the spec is unclear as to which global object ends up bound to `this`. While ES5 strict passes `undefined`, legacy mode should still specify which global object is bound to `this`.

Firefox created precedent for a reasonably consistent and legalistic interpretation of ES3. At top-level, a non-method call ends up with the global object associated with the **caller**, because the callee evaluates to an object reference with the call site’s global object as the base of the reference. When nested within a function body, though, a non-method call ends up with the global object associated with the **callee**, because the callee evaluates to an object reference with an activation object as the base of the reference, which is then censored to **null** (ES3) or **undefined** (ES5), and it’s in the callee’s body that this is replaced with the global object—so SpiderMonkey interprets this as the callee’s global object.

This is all consistent with the way the language has been specified, but that doesn’t mean we couldn’t change it. The fact that function calls behave differently depending on whether they are at top-level or nested is *extremely* subtle.

Global objects and eval

Jeff Walden raised this question about direct and indirect eval: what happens when one context reassigns `eval` to the `eval` function of another context?

```
var indirect = otherGlobal.eval;
eval = indirect;
eval("this")     // which global?
indirect("this") // which global?
```

Global objects and navigation

On the web, a global object maintains its object identity even following a programmatic navigation to a new location. This swaps out the contents of the global object with a fresh state (recently, Firefox has implemented this with the same part of the engine that implements proxies), and navigating back can recover the previous contents of the global object. Closures that are created on one page are hard-wired to the contents of that page's global object internals, even if navigation moves away from that page. And yet throughout this navigation, that global object maintains one single object identity.

Most of this is web-specific detail that shouldn't be specified in the language standard. But the fact that closures are not actually looking up the contents of the live object, but rather an internal frame that the object delegated to at one point, seems to violate the existing spec.

Update: This may actually be spec-compliant. A global object can implement whatever behavior it wants for the internal methods; so in principle, it could always respond differently to [[Get]] based on the source of the variable lookup. In the spec, there's nothing that identifies the source of the lookup, but that doesn't mean a particular engine can't make that information available. This maybe seems a little fishy, but I'm happy if we can avoid specifying any of the mechanics of navigation in Ecma-262.

— *Dave Herman 2011/03/04 19:40*

Terminology question

We need good terminology for this concept of "global context" in a way that doesn't confuse with "execution context" as it's traditionally been used in the ECMAScript specs. Our terminology needs to account for:

- multiple global objects
- multiple module loaders

multiple modes (legacy, legacy strict, Harmony)

[[strawman: object_initialiser_extensions]]

ES Wiki

Trace: » function_proxy_prototype » completion_let » name_property_of_functions » multiple_globals » object_initialiser_extensions

Strawman: Declarative Object and Class Abstractions Based Upon Extended Object Initialisers

Allen Wirfs-Brock Original Proposal August 10, 2009
Revised Proposal March 2011

Abstraction creation is a central theme of object-based programming and ECMAScript provides many mechanisms that support patterns for creating object based abstractions. However, most of these patterns are constructive in nature using procedural code to create the abstractions. This approach is powerful in that it allows a wide variety of different technical mechanism to be used to construct various abstractions. It also allows for programmer defined abstractions with application specific construction semantics. However, this variety can also be problematic. It creates complexity for both readers and writers of ECMAScript program and making it difficult to ECMAScript implementations to recognize common abstraction patterns so they can be optimized. Most other programming language solve these issues by providing a straightforward declarative mechanism for defining object abstractions based upon a standardize semantics.

ECMAScript does provided a basic declarative mechanism for defining object-based abstractions. Object initialisers provide a declarative mechanism for defining objects that in most situations is more concise, readable, and maintainable than programmatic object creation using constructor functions and dynamic property insertion. The declarative nature of object initialisers also makes it easier for implementations to perform various object representation optimization. However, existing ECMAScript object initialisers do not provide declarative solutions for a number of abstraction capabilities that are common used with ECMAScript objects.

This strawman proposed ways in which ECMAScript object initialisers can be extended to make them more useful for building complete object abstractions. A number of individual candidate extensions are identified. These extensions could be selectively and individually added to the language. However, the individual extensions in combination turn ECMAScript object initialisers into a declarative abstraction mechanism that is powerful enough to serve as the primary abstraction mechanism of the language.

The goal of these extensions is not to introduce a new semantics of objects that differs from what is already in ECMAScript. Instead, it attempts to incrementally improve the existing ECMAScript abstraction mechanisms without introducing anything that a typical user might perceive as new fundamental concepts. The proposal does introduce a more concrete manifestation of "classes", but the semantics they exhibit are exactly those that are already used by the built-in ECMAScript library objects.

Individual Extensions

-

- Object Initialiser Meta Properties
- Method Properties
- Other Object Initialiser Property Attribute Modifiers
- Implicit property initialization expressions
- Class Initialisers
  - super in Object Initialisers

The following describes how the Private Names extension integrates with extended Object Initialisers.

- Private Names in Object Initialisers

Combined Syntax

The following provides an integrated syntax definition for all of the individual extensions combined with the ES5 Object initialiser syntax:

ObjectLiteral : **{ }**
{ MetaProperties **}**
{ MetaProperties **, }**
{ MetaProperties **,** *PrivateNamesList*$_{opt}$ PropertyNameAndValueList **}**
{ MetaProperties **,** *PrivateNamesList*$_{opt}$ PropertyNameAndValueList **, }**
{ *PrivateNamesList*$_{opt}$ PropertyNameAndValueList **}**
{ *PrivateNamesList*$_{opt}$ PropertyNameAndValueList **, }**

ArrayLiteral :
[*Ellision*$_{opt}$ **]**
[*MetaProperties Ellision*$_{opt}$ **]**
[*MetaProperties* **,** *ElementList* **]**
[*MetaProperties* **,** *ElementList* **,** *Ellision*$_{opt}$ **]**
[*ElementList* **]**
[*ElementList* **,** *Ellision*$_{opt}$ **]**

MetaProperties :

< MetaPropertyList >

MetaPropertyList :
MetaProperty
MetaPropertyList **,** *MetaProperty*

MetaProperty :
proto : *MemberExpression*
sealed
frozen
closed

PrivateNamesList :
PrivateName
PrivateNamesList PrivateName

PrivateName :
private *identifier* **,**
private *identifier* **:** *AssignmentExpression* **,**

PropertyNameAndValueList :
PropertyAssignment
PropertyNameAndValueList **,** *PropertyAssignment*

PropertyAssignment :
IdentifierName
sealed$_{opt}$ *PropertyName* **: const**$_{opt}$ *AssignmentExpression*
sealed$_{opt}$ **var** *PropertyName* **: const**$_{opt}$ *AssignmentExpression*
sealed$_{opt}$ **get** *PropertyName* **( ) {** *FunctionBody* **}**
sealed$_{opt}$ **set** *PropertyName* **(** *PropertySetParameterList* **) {** *FunctionBody* **}**
sealed$_{opt}$ **method** *PropertyName* **(** *FormalParameterListopt* **) {** *FunctionBody* **}**

PropertyName :
IdentifierName
StringLiteral
NumericLiteral

PropertySetParameterList :
Identifier

ClassDeclaration :
class *Identifier ClassBody*

ClassExpression :
class *Identifier*$_{opt}$ *ClassBody*

ClassBody
{ }
{ *ClassMetaProperties* **}**
{ *ClassMetaProperties* **, }**
{ *ClassMetaProperties* **,** *PrivateNamesList*$_{opt}$ *ClassPropertyNameAndValueList* **}**
{ *ClassMetaProperties* **,** *PrivateNamesList*$_{opt}$ *ClassPropertyNameAndValueList* **, }**

{ *PrivateNamesList*$_{opt}$ *ClassPropertyNameAndValueList* **}**
{ *PrivateNamesList*$_{opt}$ *ClassPropertyNameAndValueList* **, }**

ClassMetaProperties :
< ClassMetaPropertyList **>**

ClassMetaPropertyList :
ClassMetaProperty
ClassMetaPropertyList **,** *MetaProperty*

ClassMetaProperty :
proto : *AssignmentExpression*
superclass : *AssignmentExpression*
sealed *ComponentQualifier*$_{opt}$
frozen *ComponentQualifier*$_{opt}$
closed *ComponentQualifier*$_{opt}$

ComponentQualifier :
: **class**
: **prototype**
: **instance**

ClassPropertyNameAndValueList :
ClassPropertyAssignment
ClassPropertyNameAndValueList **,** *ClassPropertyAssignment*

ClassPropertyNameAndValueList :
ClassPropertyAssignment
ClassPropertyNameAndValueList **,** *ClassPropertyAssignment*

ClassPropertyAssignment :
class *PropertyAssignment*
ConstrutorBody
PropertyAssignment

ConstrutorBody :
new (*FormalParameterListopt* **) {** *FunctionBody* **}**

[[strawman: private_names]]

ES Wiki

Trace: » completion_let » name_property_of_functions » multiple_globals » object_initialiser_extensions » private_names

Table of Contents

- Private Names: Unique Unforgable Property Names
- Private Name values
- The private declaration
- Using Private Identifiers
- Private Identifiers in Object Literals
- Private Declaration Scoping
- Private Declarations Exist in a Separate "Name Space" Parallel to the Variable Binding Environment
- Accessing Private Names as Values
- Conflict-Free Object Extension Using Private Names
- Enumeration and Reflection
- Private Name Properties Provide a Weak But Usful Form of Information Hiding
- Interactions with other Harmony Proposals
  - Enhanced Object Literals
  - Proxies
  - Modules
- References
- Discussion

Private Names: Unique Unforgable Property Names

Updated — *Allen Wirfs-Brock 2011/03/10 00:39 2010/12/08 15:16*
Revised proposal by Allen Wirfs-Brock
Original proposal by Dave Herman and Sam Tobin-Hochstadt is here.

In existing ECMAScript, property names are just strings. It is not possible to create unique, unforgable property names that are only known to a limited or controlled set of property accessors. It is possible to create non-enumerable properties, but they can still be discovered by guessing their string-valued property name. The proposed es4 facility for addressing this shortcoming was namespaces, which were complex and suffered from ambiguity and efficiency problems.

This strawman proposes three related changes to support unique, unforgable property names.

1. a new, ECMAScript language type (or possibly object `[Class]`) *Private Name*
2. generalizing the ES5 *property name* concept to include either a string (as in ES5) or a *Private Name* value
3. a `private` keyword for automatic use of *Private Name* values instead of strings in syntactic contexts in a lexically scoped fashion.

In addition to supporting various information hiding scenarios, this also allows properties to be added to existing objects without the possibility of interference with the existing properties, or with other uncoordinated additions by any other code.

Private Name values

`Private Name` is a new ECMAScript language type that will be defined in section 8 of the specification. The `Private Name` type is an open set of distinct `Private Name` values that can be used as the names of object properties. `Private Name` values do not have any corresponding literal representation within ECMAScript code. Distinct `Private Name` values are created by the `CreatePrivateName` abstract operation. Each call to `CreatePrivateName` returns a new distinct `Private Name` value. If `x` and `y` are `Private Name` values then the abstract operation `SameValue(x,y)` returns true if and only `x` and `y` are the same `Private Name` value created by a single specific call to `CreatePrivateName`.

A `Private Name` value can be used as the value of the `P` argument to any of the object internal methods defined in section 8.12 of the ECMAScript specification.

If a new ECMAScript type is added then the `typeof` operator will also need to be extended to return a new string value that identifies values of that type. Concerns have been expressed that extending `typeof` in this manner could break existing code that expects to deal with a fixed set of `typeof` values. We need to make a global decision about adding new non-object types and the impact upon `typeof`.

Object values could be used as an alternative to defining a new ECMAScript type to represent `Private Name` values. Each `Private Name` would simply be a distinct object. `Private Name` objects would have a distinct `[Class]` value. To avoid such objects being used for back-channel communication or property garbage dumps they should be created frozen. Conceivably they could all have `null` as their prototype. This may have some benefit if such names are passed between global contexts as it would prevent use of their prototype value as means of identifying their origin context. Throughout the rest of this spec. "private name value" should be read as meaning whichever form is ultimately used.

The private declaration

The `private` declaration creates a new `Private Name` value by calling `CreatePrivateName` and binds that value to a program identifier that may be used in specific syntactic contexts within the lexical scope of the declaration. An identifier that appears in a `private` declaration is call a `private identifier`.

```
private unique;  //create a new ''Private Name'' that is bound to the private identifier
''unique''.
private _x,_y;   //create two ''Private Name'' values bound to two private identifiers
```

Can the names defined in a `private` declaration be any *IdentifierName* or should they be restricted to being an *Identifier* (ie, not a reserved name)? ES5 allows any *IdentifierName* to be used after a dot or as a property name in an object literal so it may be reasonable to allow any *IdetnifierName*. However that will permit strange look formulations such as: `private private;`

Using Private Identifiers

When a private identifier appears as the *IdentifierName* of a *CallExpression* : *CallExpression* . *IdentifierName* production or of a *CallExpression* : *CallExpression* . *IdentifierName* production, the `Private Name` value that is bound to the private identifier is used as the value of the *IdentifierName*. If the identifier in one of these productions is not a private identifier then the identifier name string is used as the value of *IdentifierName*, just as in ECMAScript 5.

This permits object properties to be created whose names are `Private Name` values. It also allows for the values of such properties to be accessed.

```
function makeObj() {
   private unique;
   var obj = {};
   obj.unique = 42;  //obj has a single property whose name is a Private Name value
   print(obj.unique);//42 -- the private identifier can be used in scope to access the
property's value
   print(obj["unique"]); //undefined -- the name of the property is not the string
"unique"
   return obj;
}
```

```
var obj=makeObj();
print(obj["unique"]); //undefined -- the name of the property is still not the string
"unique"
print(obj.unique);    //undefined -- this statement is not in the scope of the private
declaration so the
                      //string value "unique" is used to look up the property.  It does
not match the Private Name value
```

This technique can be used to define "instances-only" visibility for properties. Each instance uses a unique property name and only code that is associated with the instance knows the unique name:

```
function Thing() {
    private key;   // each invocation will use a new unique private key value
    this.key = "instance private value";
    this.hasKey = function(x) {
        return x.key === this.key;  //x.key should be undefined if x!==this
    };
    this.getThingKey = function(x) {
        return x.key;
    };
}

var thing1 = new Thing;
var thing2 = new Thing;

print("key" in thing1);       // false
print(thing2.key);            //undefined
print(thing1.hasKey(thing1)); // true
print(thing1.hasKey(thing2)); // false
```

By changing the scope of the private declaration a similar technique can be used to define "class-only" visibility properties. Each instance uses the same unique property key and knowledge of the key is shared by the all the instances so they can mutually access each others private named properties:

```
private key;  //the same private name value is used by every invocation of Thing
function Thing() {
    this.key = "class private value";
    this.hasKey = function(x) {
        return x.key === this.key;
    };
    this.getThingKey = function(x) {
        return x.key;
    };
}

var thing1 = new Thing;
var thing2 = new Thing;
```

```
print("key" in thing1);        // false
print(thing1.hasKey(thing1)); // true
print(thing1.hasKey(thing2)); // true
```

Friend visibility similar to that provided by c++ can be obtained by using private declarations that are visible to several related object literals, object constructors or factory functions enclosed in an outer function and returned from it (directly, or stored as effects in objects).

Private Identifiers in Object Literals

A private identifier may also appear as the *IdentifierName* of a *PropertyName* production in an *ObjectLiteral*. If the identifier in such a productions is not a private identifier then the identifier name string is used as the value of *IdentifierName*, just as in ECMAScript 5.

With this feature, object literals can be used as an alternative expression of the previous three examples:

```
function makeObj() {
   private unique;
   var obj = {unique: 42};
   print(obj.unique);//42 -- the private identifier can be used in scope to access the property's value
   print(obj["unique"]); //undefined -- the name of the property is not the string "unique"
   return obj;
}
```

```
function Thing() {
    private key;
    return {
       key : "instance private value",
       hasKey : function(x) {
          return x.key === this.key;  //x.key should be undefined if x!==this
       },
       getThingKey : function(x) {
          return x.key;
       }
    };
}
```

```
private key;
function Thing() {
    return {
       key : "class private value",
       hasKey : function(x) {
          return x.key === this.key;  //x.key should be undefined if x!==this
       },
       getThingKey : function(x) {
          return x.key;
```

```
        }
    };
}
```

Private Declaration Scoping

`private` declarations are lexically scoped, like all declarations in Harmony. Inner `private` declarations shadow access to like-named `private` declarations in outer scopes. Within a block, the scoping rules for `private` declarations are the same as for `const` declarations.

```
function outer(obj) {
   private name;
   function inner(obj) {
      private name;
      obj.name = "inner name";
      print(obj.name);   //"inner name" because outer name declaration is shadowed
   }
   obj.name = "outer name";
   inner(obj)
   print(obj.name);      //"outer name"
}
var obj = {};
obj.name = "public name";
outer(obj);
print(obj.name);            //"public name"
```

After executing the above code, the object that was created will have three own properties:

Property Name	Property Value
"name"	"public name"
private $name_{outer}$	"outer name"
private $name_{inner}$	"inner name"

However, the above is not a very realistic example. After execution of the above code, the two private named properties could not be directly accessed because the private identifier bindings that contain their property names are no longer accessible. More typically a private identifier binding will be shared by several functions (methods) that need to have shared access to a private named property.

Private Declarations Exist in a Separate "Name Space" Parallel to the Variable Binding Environment

Consider the following very common idiom used in a constructor declaration:

```
function Point(x,y) {
   this.x = x;
   this.y = y;
   //... methods that use x and y properties
```

```
}
var pt = new Point(1,2);
```

The identifiers x and y each have two distinct bindings within the scope of function Point. On the right-hand side of the two assignment operators, x and y are identifier references (ES5 11.1.2) that bind to the formal parameter declarations for Point. Accessing them produces the values 1 and 2. However, on the right-hand side of . within the left-hand sides of those assignment expressions, x and y are used as *IdentifierNames* in Property Accessors (ES5 11.2.1) and bind to the constant string values "x" and "y".

One way to view this is that there are two distinct naming environments in ES5 programs: one used to resolve identifiers as *PrimaryExpressions* and the other used to resolve identifiers as *PropertyAccessors*. The environemnt of expressions has nested scoping contours corresponding to the local declaration of nested functions. However, in ES5 the environment for resolving *PropertyAccessor* identifiers is not particularly interesting because it is just a single global contour that implicitly binds all identifiers to the string value that is the *IdentifierName* of the identifier.

When a private declaration is added to the above example, we need to preserve the same basic semantics that we have in ES5.

```
function Point(x,y) {
   private x, y;
   this.x = x;
   this.y = y;
   //... methods that use private x and y properties
}
var pt = new Point(1,2);
```

On the right-hand side of the assignments x and y still need to refer to the formal parameter bindings, even though there is a local declaration for private names x and y. Similarly, on the left-hand side *PropertyAccessors*, x and y should bind to the private names introduced by the private declaration and not bind to the formal parameters.

As with ES5 this can be explained by using distinct naming environments for *PrimaryExpressions* vs. *PropertyAccessors*. However, the property name environment is no longer a flat set of identifier bindings. Instead it is a lexically scoped hierarchy of bindings that map from identifiers either to string values or to private name values. The hierarchical structure exactly parallels the *PrimaryExpression* environment hierarchy.

Another way to view this is that each EnvironmentRecord (ES5 10.2.1) has a second set of bindings that are used to map identifiers to property names. private declarations create such bindings in the current environment. Syntactic contexts such as *PropertyAccess* and Object Literal *PropertyName* look up identifiers using a new abstract operations GetPrivateName that is exactly like GetIdentiferReference except that it uses the property name bindings. At the top level is an the set of identifier bindings that map all identifiers to the string values of their identifier names.

Accessing Private Names as Values

The private declaration normally both creates a new private name value and introduces a name binding that can be used only in "property name" syntactic contexts to access the new private name value by the lexically bound name.

However, in some circumstances it is necessary to access the actual private name value as an expression value, not as a property name on the right of . or the left of : in an object initialiser. This requires a special form than can be used in an expression to access the private name value binding of a private identifier. The syntactic form is **#.** *IdentifierName*. This may be used as a *PrimaryExpression* and yields the property name value of the *IdentifierName*. This may be either a private name value or a string value, depending upon whether the expression is within the scope of a private declaration for that *IdentifierName*;

```
function addPrivateProperty(obj, init) {
   private pname;      //create a new private name
   obj.pname = init;  //add initialize a property with that private name
   return #.pname;    //return the private name value to the requestor
}

var myObj = {};
var answerKey = addPrivateProperty(myObj, 42);
print(myObj[answerKey]);   //42,  note that answerKey is a regular variable so [ ] must be used to access the property
//myObj can now be made globally available but answerKey can be selectively passed to privileged code
```

Note that simply assigning a private name value to a variable does not make that variable a private identifier. For example, in the above example, the print statement could not validly be replaced with:

```
print(myObj.answerKey);
```

This would produce "`undefined`" because it would access the non-existent property whose string valued property name would be "`answerKey`". Only identifiers that have been explicitly declared using `private` are private identifiers.

Enabling the use of [ ] with private name values requires a minor change to the ES5 specification. In 11.2.1, step 6 must be changed to call ToPropertyName rather than ToString. `ToPropertyName(name)` is defined as follows:

1. If name is a private name value, return name.
2. Return the result of ToString(name).

This will not change the semantics of any existing JavaScript code because such code will not contain any use of private name values.

The only operators that can be successfully be applied to a private name value are == and === both of which return true when both operands is the same private name value.

Private name values can be converted to strings using the internal `ToString` abstract operation. However, the string value does not have any correspondence to the identifier in the `private` declaration that created the private name value. The string value produced by such a string conversion is simply "`Private Name`".

?It may be useful to allow "`Private Name`" to be followed by additional implementation dependent text. This might be used to provide additional identifying information such as a source text line number or a unique serial number that would be useful for debugging.

If **#.** is not within the scope of a `private` declaration for its *IdentifierName* then the value produced is the string value of the *IdentifierName*. In other words, **#.** *IdentiferName* always produces the same value as would be used as the property name in a *PropertyAccess* using that same *IdentifierName*.

As an expressive convenience, `private` declarations can be used to associate a private identifier with an already existing private name value. This is done by using a `private` declaration of the form:

private *Identifier* **=** *Initialiser* **;**

If *Initialiser* does not evaluate to a private name value, a TypeError exception is thrown. (? *for uniformity, should string values be allowed? In that case, local private name bindings could be string valued.*)

```
private name1;   //value is a new private name
private name2 = #.name1  //name2 can be used to access the same property name as name1
```

Conflict-Free Object Extension Using Private Names

Some JavaScript frameworks and libraries extend built-in objects by adding new properties to built-in prototype objects. For example, a framework might choose to add a `clone` method to `Object.prototype` that may be used to make a copy of any object. Problems occur when two or more frameworks both try to add a method named `clone`. Private names can be used to avoid such conflicts. If each framework uses a private name rather than than a string property name than each can install a `clone` method on `Object.prototype` without interfering with the other.

For example, someone might create library that does recursive deep copying of object structures that was organized something like this:

```
function installCloneLibrary() {
   private clone;   // the private name for clone methods

   // Install clone methods in key built-in prototypes:
   Object.prototype.clone = function () { ... };
   Array.prototype.clone = function () {
       ...
       target[i] = this[i].clone();  // recur on clone method
       ...
   }
   String.prototype.clone = function () {...}
   ...
   return #.clone
}

// Example usage of CloneLibrary:
private clone = installCloneLibrary();
installAnotherLibrary();
var twin = [{a:0}, {b:1}].clone();
```

The above client of the `CloneLibrary` will work even if the other library also defines a method named `clone` on `Object.prototype`. The second library would not have visibility of the private name used for `clone` so it would either use a string property name or a different private name for the method. In either case there would be no conflict with the method defined by `CloneLibrary`.

Enumeration and Reflection

Properties whose property names are private name values have all the same attributes as a property whose property name is a string value and the same defaults attribute are generally used. However, in most cases it is likely that properties defined using private names should not show up in for-in enumerations. For this reason, the semantics of the

standard internal `[[Put]]` method are modified for cases where a property is created by `[[Put]]` using a Private Name value as the property name. In this case the `[[Enumerable]]` attribute of the newly created property is initially set to **false**. This change in made ES5 8.12.5, step 6.a.

For example:

```
private b;
var obj = {};
obj.a = 1;
obj.b = 2;
obj.c = 3;

var names = [];
for (var p in obj) names.push(obj[p]);
print(names.toString());    // "1,3" -- private name "b" was not enumerated
```

Properties with private names that are created using object literals also are created with their `[[Enumerable]]` attribute **false**. So `obj` could have been created to produce the same result by saying:

```
private b;
var obj = {
   a: 1,
   b: 2,
   c: 3
}
```

Because object literal properties are specified using `[[DefineOwnProperty]]` rather than `[[Put]]` all property descriptors used in ES5 11.1.5 must be updated to set `[[Enumerable]]` to **false** whenever a *PropertyName* is a private name value.

> Need to check all other uses of `[[DefineOwnProperty]]` and determine whether any of them should have special treatment of private name values.

Creating a private named property that is enumerable requires use of `Object.defineProperty` and the **#.** prefix. For example:

```
private b;
var obj = {};
obj.a = 1;
obj.b = 2;
Object.defineProperty(obj, #.b, {enumerable: true});
obj.c = 3;

var names = [];
for (var p in obj) names.push(obj[p]);
print(names.toString());    // "1,2,3" -- private name "b" is now enumerated
```

`Object.prototype.hasOwnProperty` (ES5 15.2.4.5), `Object.prototype.PropertyIsEnumerable`

(ES5 15.2.4.7) and the `in` operator (ES5 11.8.7) are all extended to accept private name values in addition to string values as property names. Where they currently call ToString on property names they will instead call ToPropertyName. The `JSON.stringify` algorithm (ES5 15.12.3) will be modified such that it does not process enumerable properties that have private name values as their property names.

All the Object reflection functions defined in ES5 section 15.2.3 that accept property names as arguments or return property names are extended to accept or produce private name values in addition to string values as property names. A private name value may appear as a property name in the collection of property descriptors passed to `Object.create` and `Object.defineProperties`. If an object has private named properties then their private name values will appear in the arrays returned by `Object.getOwnPropertyNames` and `Object.keys` (if the corresponding properties are enumerable).

In ES5 `Object.defineProperties` and `Object.create` are specified to look only at the enumerable own properties of the object that is passed containing property descriptors. Usually this object is specified using an object literal. However, we have already specified above that private named properties in object literals are always created as non-enumerable properties. This would generally preclude the use of private name values in the object literals passed to these methods. It may be necessary to modify the specification of `Object.defineProperty` and `Object.create` to process also as property definitions any non-enumerable private named own properties that appear in such objects.

```
// Object.defineProperty probably needs to accept descriptors such as:
private a, b;
Object.defineProperties(obj, {
   a: {configurable: true}, // ES5 ignores non-enumerable properties
   b: {writable: true}      // that appear in such descriptors
});
```

An important use case for reflection using private name values is algorithms that need to perform meta-level processing of all properties of any object. For example, a "universal" object copy function might be coded as:

```
function copyObject(obj) {
   // This doesn't deal with other special [[Class]] objects:
   var copy = Object.isArray(obj) ? [] : Object.create(Object.getPrototypeOf(obj));
   var props = Object.getOwnPropertyNames(obj);
   var pname;
   for (var i = 0; i < props.length; i++) {
      pname = props[i];
      Object.defineProperty(copy, pname, Object.getOwnPropertyDescriptor(obj,pname));
   }
   return obj;
}
```

This function will duplicate all properties, including any that have private name values as their property names. It does not need to have any specific declarations for or knowledge of such private names.

Some reflection function could potentially be used to discover private name values used for an object's properties that would not have otherwise been known to the caller to the reflection function. Some environment, particularly sandboxes may wish to precluded such discovery. This can be done by replace the built-in reflection functions with wrapper that filter access to private name keyed properties. This is similar in concept to the wrapping that some environments do to secure functions such as eval or the Function constructor.

Private Name Properties Provide a Weak But Usful Form of Information Hiding

Private names are a simple and pragmatic way to support information hiding within ECMAScript objects without requiring programmers to change their fundamental conceptualization of JavaScript objects. They make available unforgable unique values that can be used to name properties. While private names are unforagable, there are no mechanisms that guarantee that a private name can never leak to an unintended agent within a program.

Private named properties do not provide and are not intended to provide strong impenetrable encapsulation of object state. For example, various reflection operations can be used to access an object's properties that have private names. Private names are instead intended as a simple extensions of the classic JavaScript object model that enables straight-forward encapsulation in non-hostile environments. The design preserves the ability to manipulate all properties of an objects at a meta level using reflection and the ability to perform "monkey patching" when it is necessary. Encapsulation via closure capture should continue to be used for situations where strong encapsulation that can not be penetrated by a hostile attacker is actually needed.

Sand-boxing environments that already need to restrict the use of certain reflection operations can use similar technique to limit access to private named properties. For example, a sandbox implementation might replace `Object.getOwnPropertyNames` with a version that filters out any private name values.

Interactions with other Harmony Proposals

There are potential feature interactions and opportunities for feature integration involving private names and several other Harmony proposals. As features are accepted into Harmony and their details are filled in these interactions need to be resolved.

Enhanced Object Literals

The Object Initialiser Extensions include a Private Names in Object Initialisers proposal that integrates private name declarations with extended object literals.

Proxies

All uses of string valued property names in proxy handlers would need to be extended to accept/produce private name values in addition to string values.

As covered above, ECMAScript reflection capabilities provides a means to break the encapsulation of an object's private named properties. Where this is a concern, it can be mitigated by replacing the reflection functions with versions that filter access to private name values. The Proxy proposal provides an additional means to break such encapsulation. If an attacker suspects that some object has private named properties it might sniff out those values by creating a proxy for the object whose handler consisted of traps that monitored all calls looking for private name argument values before delegating the operation to the original object.

One possible mitigation for this attach would be the same as for the other reflection functions. The Proxy object could be replaced with an alternative implementation that added an additional handler layer that would wrapper all private name values passed through its traps. The wrappers would be opaque encapsulations of each private name value and provide a method that could be used to test whether the encapsulated private name was === to an argument value. This would permit handlers to process known private name values but would prevent exposing arbitrary private name values to the handlers.

If there is sufficient concern about proxies exposing private name values in this manner, such wrapping of private names could be built into the primitive trap invocation mechanism.

Modules

It is reasonable to expect that modules will want to define and export private name values. For example, a module might

want to add methods to a built-in prototype object using private names and then make those method names available to other modules. Within the present definition of the simple module system that might be done as follows:

```
<script type="harmony">
module ExtendedObject {
   import Builtins.Object;        // however access to Object is obtained.
   private clone;                 // the private name for clone methods
   export const clone = #.clone; // export a constant with the private name value;

   Object.prototype.clone = function () { ... };
}
</script>
```

A consumer of this module might look like:

```
<script type="harmony">
import ExtendedObject.clone;
private clone = clone;
var anotherObj = someObj.clone();
</script>
```

The above formulation would work without any additional extensions to the simple module proposal. However, it would be even more convenient if the module system was extended to understand private declarations and the parallel property name environment. In that case this example might be written as:

```
<script type="harmony">
module ExtendedObject {
   import Builtins.Object;      // however access to Object is obtained.
   export private clone;        // export private name for clone methods

   Object.prototype.clone = function () { ... };
}
</script>
```

```
<script type="harmony">
import private ExtendedObject.clone;
var anotherObj = someObj.clone();
</script>
```

In these example the use of `import` and `export` prefixing `private` declarations forces use of the property name environment of the named module. For dynamic access to the exported property name environment of first-class module instances another mechnism would perhaps be needed:

```
<script type="harmony">
module private ExtendedObject_names = ExtendedObject;

private cloneEX = ExtendedObject_names.clone;  //get private name value bound to
```

```
''clone'' in reified module ''ExtendedObject''
var yetAnotherObj = someObj.cloneEX();
</script>
```

References

Private name values are akin to what is returned by `gensym` of Lisp and Scheme, and analogous to a capability in object-capability languages.

The inspiration for `private` is Racket's define-local-member-name and "selector namespaces" for Smalltalk and Ruby .

Discussion

- names vs soft fields (markm) – compares names to inherited explicit soft fields.