

[[strawman:  
classes\_with\_trait\_composition]]

Trace: » strawman »  
completion\_reform »  
proxy\_drop\_receiver » proxy\_defaulthandler » classes\_with\_trait\_composition

## Classes with Trait Composition

This is a major revision of the earlier classes and traits strawman in order to reconcile [object\\_initialiser\\_extensions](#), especially [obj\\_initialiser\\_class\\_abstraction](#) and [instance\\_variables](#). A prototype implementation of an earlier version of this reconciled strawman is described at [Traceur Classes and Traits](#).

The strawman as presented on this page no longer supports general trait composition, abstract classes, required members, or multiple inheritance, as we felt that was premature to propose at the May 2011 meeting, and therefore premature to propose for inclusion in the EcmaScript to follow ES5. Instead, we have extracted those elements into [trait\\_composition\\_for\\_classes](#), whose existence demonstrates that the single inheritance shown here does straightforwardly generalize to support these extensions.

Table of Contents	▲	□
• Classes with Trait Composition		
◦ Class Declarations and Expressions		
▪ Class Adjectives		
◦ Class and Constructor Elements		
◦ Member Declarations and Definitions		
◦ Inheritance		
◦ One Way to do Encapsulation		
• Open Issues		
• See		

## Class Declarations and Expressions

We extend the Declaration production from [block scoped bindings](#) to accept a *ClassDeclaration*. We extend MemberExpression to accept a *ClassExpression*.

```

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

```

A *ClassDeclaration* or *ClassExpression* defines a constructor function to represent that class. From here on, we refer to such a constructor function as a *class*.

Like a *FunctionDeclaration*, a *ClassDeclaration* brings the class name into scope at the beginning of the immediately containing Block-like unit (Block, *FunctionBody*, Program, *ModuleBody*, *ClassBody*, *ConstructorBody*, ...), and binds this variable on entry to that Block-like unit. This supports mutual recursion among *FunctionDeclarations* and *ClassDeclarations*. Like a *FunctionExpression*, a *ClassExpression* defines an anonymous class if the identifier is omitted, or, if present, binds the class name only in the scope seen by the class being defined. Just as we may refer to *FunctionDeclaration* and *FunctionExpression* together as a *function definition*, we refer to *ClassDeclaration* and *ClassExpression* together as a *class definition*.

A class is a description of instances. A class may serve as a factory for directly making the instances it describes, and it may contribute such descriptions to be inherited by other descriptions of other classes. If the optional *Proto* is absent, then this constructor function's "prototype" property is a fresh object inheriting directly from `Object.prototype`. Otherwise, it is initialized according to the [Inheritance](#) section below.

## Class Adjectives

The ClassAdjectives, when present, clarify the role a given class is intended to serve. The `const` adjective indicates that this class should provide high integrity. Const classes are frozen and they bring their class name into scope as a `const` variable, i.e., a non-assignable variable. For a const class `C`, `C.prototype` is also frozen. The instances of a const class are sealed by default.

## Class and Constructor Elements

```

ClassElement :
  PublicPrototypePropertyDefinition
  PublicClassPropertyDefinition
  Constructor           // at most one
Constructor :
  constructor ( FormalParameterList? ) { ConstructorBody }
ConstructorBody :      // by analogy to FunctionBody
  ConstructorElement*
ConstructorElement :   // by analogy to SourceElement
  Statement
  Declaration

```

On entry to a Block-like unit (Block, FunctionBody, Program, ...), the variables declared by all immediately contained FunctionDeclarations and ClassDeclarations are first bound to their respective functions and classes, and then all ClassBodies are executed in textual order. The body of a class defines and initializes class-wide properties once per evaluation of a class definition – properties both on the class itself and on the value of its “prototype” property. These initializations happen in textual order.

A ClassBody also declares the members it contributes to the layout of its instances – both the instance’s public properties and (see below) its class-private instance variables. And a ClassBody contains at most one Constructor, whose body is the code to run to help initialize instances of this class – both the local members and the constructor chaining to the contributing superclass. This constructor code is also the behavior of the class’ internal `[[Call]]` and `[[Construct]]` methods.

The body of a Constructor consists of ConstructorElements. These are like the interleaved Declarations and Statements that appear in blocks elsewhere in the language, but with the following differences:

- We have yet to decide what the semantics are of a ReturnStatement within a ConstructorBody.
- ProtoChaining (defined below) may occur anywhere *directly* within a ConstructorBody, i.e., excluding nested function or class definitions.

## Member Declarations and Definitions

```

PublicPrototypePropertyDefinition :
  ExportableDefinition
PublicClassPropertyDefinition :
  static ExportableDefinition

ExportableDefinition :
  Declaration
  Identifier = Expression ;           // provided data
  Identifier ( FormalParameterList? ) { FunctionBody } // provided method
  get Identifier ( ) { FunctionBody } // data provided as accessor
  set Identifier ( FormalParameter ) { FunctionBody }
  MemberAdjective ExportableDefinition

```

```

MemberAdjective :
  // attribute control
IdentifierList :
  Identifier
  IdentifierList , Identifier

```

By default, data members define enumerable properties while method members define non-enumerable properties. Members of non-const classes default to writable and configurable. MemberAdjectives, if present, override the default attributes of the property being defined.

## Inheritance

```

Proto :
  extends MemberExpression
  prototype MemberExpression

CallExpression : ...
  ProtoChaining
ProtoChaining :
  super Arguments;

MemberExpression : ... // "... means members defined elsewhere
  super . IdentifierName

```

When inheriting with `extends`, the `MemberExpression` is expected to evaluate to a function with a `prototype` property. We call that function the *superclass*. The value of this class' `prototype` property inherits from the value of the superclass' `prototype` property. When inheriting with `prototype`, the value of `MemberExpression` is used directly as the object for this class' `prototype` to inherit from.

`ProtoChaining` may only appear in classes inheriting using `extends`. It may appear anywhere within a constructor body excluding nested functions and classes, enabling the superclass to contribute its part towards initialization of this instance. The `ProtoChaining` expression `super(x, y);` calls the superclass' `[[Call]]` method with `thisArg` bound to this constructor's `this` and the arguments `x` and `y`. In other words, `super(x, y);` acts like `Superclass.call(this, x, y);`, as if using the original binding of `Function.prototype.call`.

This semantics of constructor chaining precludes defining classes that inherit from various distinguished built-in constructors, such as `Date`, `Array`, `RegExp`, `Function`, `Error`, etc, whose `[[Construct]]` ignores the normal object passed in as the `this`-binding and instead creates a fresh specialized object. Similar problems occur for DOM constructors such as `HTMLElement`. This strawman can be extended to handle such cases, but probably at the cost of making classes something more than syntactic sugar for functions. We leave that to other strawmen to explore.

Within the class `C`, the expression `super.foo` evaluates to a `Reference` with *base* `this` and *referenced name* `"foo"`, but whose `[[GetValue]]` will look up `C.prototype.[[Prototype]].foo`. Operationally, this means only that `super.foo(x, y)` acts like `C.prototype.[[Prototype]].foo.call(this, x, y)`, as if using the original binding of `Function.prototype.call`. The `super.foo` expression may not be used as a `LeftHandSideExpression`.

## One Way to do Encapsulation

A requirement of the classes proposal is that there be some way to express encapsulated per-object state that meets the following requirements:

- Notational convenience and naturalness for declaring and accessing private instance state from within methods of the class.
- Strong encapsulation able to support defensiveness and security.

- Hiding implementation detail for normal software engineering encapsulation concerns.
- Expected efficient implementation, where the per-instance state is allocated with the instance as part of one allocation, and with no undue burden on the garbage collector.
- Ability to have private mutable state on publicly frozen objects.

The following is an extension of the above classes proposal that directly provides such encapsulation. Alternatively, if something like `private_names` or `unique_string_values` gets accepted, such that a pattern composing private names with classes satisfies the above requirements, then there is no need for the following extension to the classes proposal itself.

This encapsulation proposal involves declaring the class-private instance variables within the `ClassBody` but initializing them in the `ConstructorBody`. Although not a requirement, stylistically, once we introduce the notation needed to state the private members of the instances declaratively, it seems right to reuse it to state the public properties of the instances declaratively. Such declarations are the natural place to put the doc-comments that js-doc-like tools should be expected to recognize. We are not proposing any built-in consistency check between these declarations and the initialization in the constructor, but linters or other static analysis tools may well perform such checks. If `guards` are added to JavaScript, then these declarative lists are also the right place to use guards to constrain the possible values of the members.

```

ClassElement : ...
  PublicInstancePropertyDeclaration
  PrivateInstanceVariableDeclaration
PublicInstancePropertyDeclaration :
  public ForwardDeclaration
PrivateInstanceVariableDeclaration :
  private ForwardDeclaration

ForwardDeclaration : ...
  IdentifierList ; // data members
  Identifier ( FormalParameterList? ) ; // method member
  MemberAdjective ForwardDeclaration

// can appear in any Expression context within a class definition
CallExpression : ... // "... means members defined elsewhere
  private ( AssignmentExpression )

```

The `"private ( AssignmentExpression )"` expression can appear anywhere within a class where an expression can appear. The closest enclosing class is the *containing class*. This expression evaluates the `AssignmentExpression`. If it evaluates to an instance of the containing class, including an instance of a class that inherits from the containing class, then the expression evaluates to a record of all the class-private instance variables associated with this instance by the containing class. Otherwise, the expression evaluates to undefined.

Alternatively, adopting the syntactic conventions of `instance_variables`, a less verbose syntax for private variable access might be:

```

MemberExpression : ...
  MemberExpression @ Identifier
UnaryExpression : ...
  @ Identifier

```

where `"expr@foo"` means what `"private(expr).foo"` would have meant, and `"@foo"` means `"this@foo"` which means what `"private(this).foo"` would have meant.

## Open Issues

---

- - this is pronoun mystery-meat. At the front of a *ClassElement* it does not obviously connote “public per-instance state”. Also it will be repeated a lot.
    - - What about per-instance properties that have no good initial value other than the one passed into the constructor?
    - - How does this interact with the constructor writing to the same properties?
    - - Suggest a braced form for per-instance state, if it’s really necessary to declare: `public { ... }, private { ... }`.
    - - Alternative: C++ constructor-head syntax.
- - `new` is misnamed as it preempts a method named “new” (legal since ES5) declared using the same *PublicPrototypePropertyDefinition* syntax.
    - - Meanwhile, `constructor` is missing as a prototype property so it must be assigned the value of the `new` member by magic, to conform to ES1-5.
    - - “Explicit is Better Than Implicit” favors naming `C.prototype.constructor` in `class C { ... }` with exactly one *PublicPrototypePropertyDefinition* whose name is `constructor`, not `new`.
    - - So, why not name the constructor `constructor` instead of `new`?
    - - If too long, use a non-*IdentifierName* sigil or punctuator (or not, but don’t preempt `new`).
- - Whether `extends` means prototypal delegation or traits composition depends on the dynamic type of the *MemberExpression* to its right. Trouble with a capital T.
    - - We should question the union-of-two-specs super-spec assembly that seems to be happening here.
- - Some higher-level thoughts.

— *Brendan Eich* 2011/05/16 04:52

## See

---

[object\\_initialiser\\_extensions](#), especially [obj\\_initialiser\\_class\\_abstraction](#) and [instance\\_variables](#)

Traceur Classes and Traits

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

[classes as sugar](#)

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

[classes as inheritance sugar](#) (not yet ready)

[trait\\_composition\\_for\\_classes](#)



[[strawman:  
traits\_semantics]]Trace: » completion\_reform »  
proxy\_drop\_receiver » proxy\_defaulthandler

» 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 :: [ PMap ] -> PMap
TCompose [] = {}
TCompose (pdm:pdms) =
  compose_pmap pdm (TCompose pdms)

compose_pmap :: PMap -> PMap -> PMap
compose_pmap pdm { } = pdm
compose_pmap pdm { n1:p1, ... , nk:pk } =
  compose_pmap (add_prop pdm n1:p1) { n2:p2, ..., nk:pk }

add_prop :: PMap -> PropertyIdentifier -> PMap
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 :: [ PMap ] -> PMap
TOverride [] = {}
TOverride (pdm:pdms) =
  override_pmap pdm (TOverride pdms)

override_pmap :: PMap -> PMap -> PMap
override_pmap pdm {} = pdm
override_pmap pdm { n1:p1, ... , nk:pk } =
  override_pmap (override_prop pdm n1:p1) { n2:p2, ..., nk:pk }

override_prop :: PMap -> PropertyIdentifier -> PMap
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 -> PMap -> PMap
TResolve r e pdm =
  rename r (exclude e pdm)

exclude :: Exclusions -> PMap -> PMap
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 -> PMap -> PMap
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 -> PMap -> 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 -> PMap -> PMap
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: conflicting property: n>

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

```



[[strawman:  
inherited\_explicit\_soft\_fields]]Trace: »  
proxy\_drop\_receiver

» proxy\_defaulthandler » 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 than 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.

[[strawman:  
names\_vs\_soft\_fields]]Trace: » proxy\_defaulthandler »  
classes\_with\_trait\_composition

» traits\_semantics » inherited\_explicit\_soft\_fields » names\_vs\_soft\_fields

## 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
```

### 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

```

    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
                    //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.hasOwnProperty(thing1)); // true
print(thing1.hasOwnProperty(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 <sub>outer</sub>	"outer name"
private name <sub>inner</sub>	"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://gnu.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 copy;
}

```

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 “west”) 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.

[[strawman:  
quasis]]

Trace: » classes\_with\_trait\_composition »  
traits\_semantics » inherited\_explicit\_soft\_fields

» names\_vs\_soft\_fields » quasis

## EcmaScript Quasi-Literal Strawman

### Table of Contents ▲ □

- EcmaScript Quasi-Literal Strawman
  - Motivation
  - Overview
    - Syntax
    - Semantics
  - Use Cases
    - Secure Content Generation
    - Text L10N
      - Message Extraction
      - Message Meta-data
      - Substitution Meta-data
      - Message replacement and substitution re-ordering
      - Specifying a locale
      - Security
    - Query Languages
    - Message Sends
  - Flexible Literal Syntax
  - Raw Strings
  - Decomposition Patterns
  - Logging
  - Syntax (normative)
    - Literal Portion Syntax
      - QuasiLiteral ::
      - LiteralPortion ::
      - LiteralCharacter ::
      - QuasiLiteralTail ::
      - Substitution ::
    - Literal Portion Array
    - QuasiTag
      - QuasiTag ::
      - QT
      - Default Quasi Tag
    - Substitution Body Syntax
      - SubstitutionBody ::
      - SubstitutionModifier ::
      - SVE
  - Semantics (normative)
    - Desugaring
  - Security Considerations
    - Defensive Code
    - Offensive Code
  - Possible Problems
  - Reasons and Open Issues
    - Quoting Character
    - Nesting
    - Substitutions
    - Raw Escapes in Literal Sections
    - Line Continuation
  - References
    - Quasis in E
    - Secure String Interp
    - PHP String Vars
    - PLT Scheme Scribble
    - SML of New Jersey

### 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 analyzability 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.

#### 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 Code Generation
- Scheme Hygienic Macros
- Paradigm Regained
- Safe Templates

See the [demo REPL](#) for some runnable examples. Especially the drop-down at the top-right.

## Secure Content Generation

```
safehtml`<a href="${url}?q=${query}" onclick=alert(`${message}) style="color:
${color}">${message}</a>`
```

uses [contextual auto-escaping](#) to figure out that `url` and `color` should be filtered, `query` should be percent-encoded, and `message` HTML entity encoded to prevent XSS.

The syntax provides a clear distinction between trusted content such as `<a href="` and substituted values that might be controlled by an attacker such as `url`. This prevents the problem 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.

E.g.

```
url = "http://example.com/",
message = query = "Hello & Goodbye",
color = "red",
safehtml`<a href="${url}?q=${query}" onclick=alert(`${message}) style="color:
${color}">${message}</a>`
```

produces

```
<a href="http://example.com/?q=Hello%20%26%20Goodbye"
  onclick=alert(&#39;Hello&#32;\x26&#32;Goodbye&#39;) style="color: red">Hello
&amp; Goodbye</a>
```

but value filtering can be done so that if instead

```
url = "javascript:alert(1337)"
color = "expression(alert(1337))"
```

then the output is filled with innocuous values instead to produce:

```
<a href="#innocuous?q=Hello%20%26%20Goodbye"
  onclick=alert(&#39;Hello&#32;\x26&#32;Goodbye&#39;) style="color:
innocuous">Hello &amp; Goodbye</a>
```

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

## Text L10N

```
msg`Welcome to ${siteName}, you are visitor number ${visitorNumber}!`
```

where `visitorNumber` should be formatted using locale-specific conventions, e.g. "1,000,000" in some parts of the world, and "1.000.000" in some others.

## Message Extraction

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

For example, a static analyzer could find uses of `msg`...`` in source files to produce a message bundle like

```
<messagebundle>
  <message id="...">Welcome to {0}, you are visitor number {1}!</message>
</messagebundle>
```

Translators can then produce a message bundle with the translations.

## Message Meta-data

Translators often need some context to help them translate human readable message strings.

Meta-data can be attached to comments the way [other systems](#) put type declarations and structured documentation in comments.

```
/**
 * @description Label text for a button that opens a window.
 */
myButton.innerText = msg`Open`;
```

but if the English word "Open" is used in two different forms (adjectival vs imperative), it may need to have two translations, so some L10N approaches would benefit from having disambiguation meta-data available at runtime.

There are two common ways of disambiguating:

1.

Associating the message with an identifier which is used as a message ID : `#MSG_OPEN_BUTTON_TEXT`

```
{ msg`Open` }
```

2.

Adding "meaning" meta-data to the message. `myButton.innerText = msg`Open ; meaning="Button text"`;`

The latter convention makes the meta-data available not just to static analyzers, but also at runtime.

## Substitution Meta-data

Meta-data can also be associated with substitutions in the same way.

```
/**
 * @param siteName The name of the site. @example foo.ru
 * @param visitorNumber an integer. @example 1000000
 */
var message = msg`Welcome to ${siteName}, you are visitor number ${visitorNumber}:d!`;
```

The description of `siteName` and the `@example` meta-data can be extracted along with the message and made available to translators but is not needed at runtime.

The `:d` meta-data is available at runtime to specify that the number should be presented as an integer. Never in scientific notation no matter how many billions of visitors `foo.com` receives.

## Message replacement and substitution re-ordering

Once translators have delivered their translations, there are a number of ways to incorporate those.

If the locale is known statically, then `msg`...`` elements can be **fully rewritten**

```
// Before
alert(msg`Hello, ${world}!`);

// After
alert(msg`Bonjour ${world}!`);
```

If the locale is not known statically, then a source code rewriter can **partially rewrite** the message to a lookup into a side-table by message id.

```
// Before
alert(msg`Hello, ${world}!`);

// After
var messageBundle_fr = {
  MSG_1234: ['Bonjour ', 0 /* An index into substitutions */, '!']
};

alert(getMessage('MSG_1234', [world]));
```

The most natural order in which elements of a thought are expressed may differ between languages. msg`Welcome to \${siteName}, you are visitor number \${visitorNumber}:d!` might be translated into pig-latin as msg`Oo-yay are-yay isitor-vay umber-nay \${visitorNumber}. Elcome-way oo-tay \${siteName}!`.

The index in the message bundle side-table above serve to identify the index of the substitution that fills that hole. For the pig-latin message above, the side-table would look like

```
var messageBundle_piglatin = {
  MSG_5678: ['Oo-yay are-yay isitor-vay umber-nay ', 1, '. Elcome-way oo-tay ', 0, '!']
};
```

Small projects that are not willing to introduce a source-code rewriting step just to get translation can do **purely dynamic** message replacement.

```
// Before
alert(msg`Hello, ${world}!`);

// After
var messageBundle_fr = { // Maps message text and disambiguation meta-data to
  replacement:
  'Hello, {0}!': 'Bonjour {0}!'
};

alert(msg`Hello, ${world}!`);
```

where msg checks the side-table:

```
function msg(parts) {
  var key = ...; // 'Hello, {0}!' given ['Hello, ', world, '!']

  var translation = myMessageBundle[key];

  return (translation || key).replace(/\{(\d+)\}/g, function (_, index) {
    // not shown: proper formatting of substitutions
    return parts[(index << 1) | 1];
  });
}
```

## Specifying a locale

EcmaScript applications running in the browser typically deal with only one user, hence operate in only one locale. But EcmaScript on the server does not, and there are exceptions in browser-based EcmaScript apps.

It is possible to specify a locale for a scope so that a particular message bundle is used for message replacement, and so that locale is used for formatting numbers and dates.

```
let lmsg = msg.withLocale(messageRecipientLocale);
sendRecommendation(msg`Your friend ${friendName} thinks you would like to read
"${articleTitle}".`);
```

## Security

Generating human readable strings often requires combining data from other users to produce human readable strings of HTML. As such, it is a prime vector for XSS attacks.

It is possible to compose the L10N use case described in this with the [secure content generation scheme](#) so there is no need to choose between localizability and security.

```
function msg(parts) {
  var metaData = extractMetaDataFromLiteralParts(parts);
  replaceLiteralPartsWithLocaleSpecificLiteralParts(parts, metaData);
  reorderAndFormatSubstitutions(parts, metaData);
  return parts.join('');
}

function safehtml(parts) {
  var sanitizers = chooseEscapingFunctionsBasedOnLiteralParts(parts);
  applySanitizersToSubstitutions(sanitizers, parts);
  return parts.join('');
}

// The composition
function safehtml_msg(parts) {
  var metaData = extractMetaDataFromLiteralParts(parts);
  replaceLiteralPartsWithLocaleSpecificLiteralParts(parts, metaData);
  reorderAndFormatSubstitutions(parts, metaData);
  var sanitizers = chooseEscapingFunctionsBasedOnLiteralParts(parts);
  applySanitizersToSubstitutions(sanitizers, parts);
  return parts.join('');
}
```

## 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 regular expression 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)

## Literal Portion Syntax

This defines the top quasi literal production and explains how the boundaries between literal portions and substitutions are determined.

### QuasiLiteral ::

- *QuasiTag* [no *LineTerminator* here] ` *LiteralPortion QuasiLiteralTail*

### LiteralPortion ::

- *LiteralCharacter LiteralPortion*
- $\epsilon$

### LiteralCharacter ::

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

### QuasiLiteralTail ::

- `
- *Substitution LiteralPortion QuasiLiteralTail*

### Substitution ::

- \$ { *SubstitutionModifier SubstitutionBody* }
- \$ *Identifier*

## Literal Portion Array

The LPA operator defines an array of strings derived from the raw text of the literal portions of the quasi.

E.g. the LPA for the quasi `q`foo${bar}baz`` is `[ 'foo' , `baz` ]`.

Production	Result
<i>QuasiLiteral</i> :: <i>QuasiTag</i> ` <i>LiteralPortion QuasiLiteralTail</i>	array-concat(LPA( <i>LiteralPortion</i> ), LPA( <i>QuasiLiteralTail</i> ))
<i>QuasiLiteralTail</i> :: <i>Substitution LiteralPortion QuasiLiteralTail</i>	array-concat(single-element-array(LPA( <i>LiteralPortion</i> )), LPA( <i>QuasiLiteralTail</i> ))
<i>QuasiLiteralTail</i> :: `	an empty array



<i>LiteralPortion</i> :: <i>LiteralCharacter LiteralPortion</i>	string-concat(LPA( <i>LiteralCharacter</i> ), LPA( <i>LiteralPortion</i> ))
<i>LiteralPortion</i> :: $\epsilon$	the empty string
<i>LiteralCharacter</i> :: <i>SourceCharacter</i>	single character string containing that character.
<i>LiteralCharacter</i> :: <i>LineTerminatorSequence</i>	the single character string containing a LF (" $\backslash n$ ")
<i>LiteralCharacter</i> :: $\$ \backslash$ <i>EscapeSequence</i>	CV( <i>EscapeSequence</i> )

## QuasiTag

Before the open backquote (`) there is a, possibly optional, tag that specifies a function that receives the literal portions and substitutions.

`quasis-quasitag-memberexpr` is another way to define the *QuasiTag* production that allows for arbitrary member expressions. If worthwhile, it should be adopted *instead of* this section.

### QuasiTag ::

- 
- Identifier*
- 
- $\epsilon$

### QT

Production	Result
<i>QuasiTag</i> :: <i>Identifier</i>	an expression of the form <i>PrimaryExpression</i> : <i>Identifier</i> with the given <i>Identifier</i>
<i>QuasiTag</i> :: $\epsilon$	the Default Quasi Tag function below

### Default Quasi Tag

The default quasi tag is a frozen function defined as

```
// mixedLiteralPortionsAndSubstitutions :
//   An odd-length array where even elements (0-indexed) are
function (mixedLiteralPortionsAndSubstitutions) {
  // As per the original Array.prototype.join.
  return mixedLiteralPortionsAndSubstitutions.join('');
}
```

If using a [currying](#) or [thunking](#) version of the desugarring, then this needs to be adapted to interleave and/or apply the substitutions.

## Substitution Body Syntax

Between literal portions there are substitutions of the form  $\$\{ \dots \}$  or  $\$ident$ . The substitution body specifies an expression, e.g. the substitution bodies in `quasitag`literal0  $\$\{1 + 1\}$  literal1 $bar literal2`` are `(1 + 1)`, and `(bar)`.

This defines the substitution body of a quasi using the *PrimaryExpression* syntactic production. Determining where a substitution ends requires, in the general case, the ability to parse an EcmaScript expression.

E.g. the SVE of `quasitag`literalPortion0 $x literalPortion1  $\$\{y + z\}$  literalPortion2`` is `[x, (y + z)]`.

Below are other ways of defining the *SubstitutionBody* production and the SVE spec function. If preferred, they should be used *instead of* this section.

•

- **quasis-substitutions-identonly** - only identifiers are allowed, e.g. `x`.
- **quasis-substitutions-simple-members** - more complex expressions, but easily lexically boundable. e.g. `x.y[z]` but does not allow nested quasis.
- **quasis-substitutions-thunk** - arbitrary expressions are allowed, and the expressions are thunkified so that a quasi handler (QT) may evaluate them zero or multiple time to support branching or looping.
- **quasis-substitutions-slot** - arbitrary expressions are allowed, and expressions that are preceded by the modifier `=` may be used as left hand sides, assigned to by the quasi handler. This enables use cases like the destructuring regular expression match.

### SubstitutionBody ::

- *PrimaryExpression*

### SubstitutionModifier ::

- $\epsilon$

### SVE

Production	Result
<i>QuasiLiteral</i> :: <i>QuasiTag</i> ` <i>LiteralPortion</i> <i>QuasiLiteralTail</i>	<i>SVE(QuasiLiteralTail)</i>
<i>QuasiLiteralTail</i> :: <i>Substitution</i> <i>LiteralPortion</i> <i>QuasiLiteralTail</i>	<i>array-concat(single-element-array(SVE(Substitution)), SVE(QuasiLiteralTail))</i>
<i>QuasiLiteralTail</i> :: `	an empty array
<i>Substitution</i> :: \$ <i>Identifier</i>	<i>PrimaryExpression</i> : <i>Identifier</i>
<i>Substitution</i> :: \$ { <i>SubstitutionModifier</i> <i>SubstitutionBody</i> }	<i>SVE(SubstitutionBody)</i>
<i>SubstitutionBody</i> :: <i>PrimaryExpression</i>	<i>PrimaryExpression</i>

The SVE is an expression that evaluates the specified expression in the scope in which the quasi appears. The SVE of the quasi literal is the array of the SVE for each substitution body.

## Semantics (normative)

---

Given the QT, LPA, and SVE defined above, this specifies the desugaring of the *QuasiLiteral* production.

This version passes all the parts to the function specified by the quasi tag in one argument list instead of passing the literal portions first.

It may be easier for optimizing rewriters to optimize some quasis if the desugaring passes literal portions separately from substitutions. See [quasis-desugaring-curried](#) for an alternate desugaring that can be used *instead of* this section.

## Desugaring

A *QuasiLiteral* in an EcmaScript parse tree is replaced with *(CallExpression, (QT, (ParameterList, (interleave(LPA, SVE)))* where *interleave* is defined as an *ArrayExpression* as defined below.

```
function interleave(lpa, sve) {
```

```

var interleaved = [lpa[0]];
for (var i = 0, k = 0, n = sve.length;) {
  interleaved[k++] = sve[i];
  interleaved[k++] = sve[++i];
}
return interleaved;
}

```

So if for `quasiTag`literalPortion0 $x literalPortion1`` the QT is `quasiTag`, LPA is `['literalPortion0 ', ' literalPortion1']` and SVE is `[x]`, then the desugaring is

```
quasiTag(['literalPortion0 ', x, ' literalPortion1'])
```

## 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 QT 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 < 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 QT 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 QT in the environment in which it is defined,
3. for QTs 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.

## Nesting

There are a number of advantages to allowing quasis to nest. Substitutions are easy to understand if they are just expressions, and quasis are just another kind of expression.

There are concrete use cases as well. We could integrate control flow into the `safehtml` quasi handler available at the [REPL](#), but substitutions with nested quasis can serve just as well.

```
rows = [['Unicorns', 'Sunbeams', 'Puppies'], ['<3', '<3', '<3']],
safehtml`<table>${
  rows.map(function(row) {
    return safehtml`<tr>${
      row.map(function(cell) {
        return safehtml`<td>${cell}</td>`
      })
    }</tr>`
  })
}</table>`
```

produces something like

```
<table>
  <tr><td>Unicorns</td><td>Sunbeams</td><td>Puppies</td></tr>
  <tr><td>&lt;3</td><td>&lt;3</td><td>&lt;3</td></tr>
</table>
```

## 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(
      /\\"(?:([\rnfvtvb])|(\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`(?:\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(''));
  };
}
```



## Line Continuation

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

It would be convenient for some DSL use cases to allow *LineTerminators* 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 *LiteralPortions*'s contents are escaped. Consider the following code where `&#xb6;` indicates where a newline occurs: ``foo&#xb6;bar`` vs ``foo \ &#xb6;bar``. The former is equivalent to ``foob&#xb6;r`` 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 *LineContinuations* as normal content, consistent with the way escapes are treated as raw inside *LiteralPortions*.
- Interpret *LineContinuations* as an empty sequence of characters and allow disallow *LineTerminators* otherwise.
- Disallow *LineTerminators* 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

## Safe Templates

Using Type Qualifiers to Make Web Templates Robust Against XSS

strawman/quasis.txt · Last modified: 2011/04/06 15:52 by mikesamuel





[[strawman:  
concurrency]]Trace: » traits\_semantics »  
inherited\_explicit\_soft\_fields »

names\_vs\_soft\_fields » quasir » concurrency

## Communicating Event-Loop Concurrency and Distribution

Aggregate objects into process-like units called *vats*. Objects in one vat can only send asynchronous messages to objects in other vats. *Promises* represent such references to potentially remote objects. *Eventual message sends* queue *pending deliveries* in the work queue of the vat hosting the target object. A vat's thread processes each pending delivery to completion before proceeding to the next. Each such processing step is a *turn*. A *when expression* registers a callback to happen in a separate turn once a promise is resolved, providing the callback with the promise's *resolution*. The eventual send and when expressions immediately return a promise for the eventual outcome of the operation they register.

This model is free of conventional race condition or deadlock bugs. While a turn is in progress, it has mutually exclusive access to all state to which it has synchronous access, i.e., all state within its vat, avoiding conventional race condition bugs without any explicit locking. The model presented here provides no locks or blocking constructs of any kind, although it does not forbid a host environment from providing blocking constructs (like `alert`). Without blocking, conventional deadlock is impossible. Of course, less conventional forms of race condition and deadlock bugs *remain*.

### Table of Contents

- Communicating Event-Loop Concurrency and Distribution
  - Vats
  - Promises and Promise States
  - Eventual Operations
  - static Q methods
  - Syntactic Sugar
- Examples
  - Spawn
  - Infinite Queue
  - race
  - Timeouts
  - allFulfilled
  - Eventual Equality
- See

## Vats

Partition the JavaScript reference graph into separate units, corresponding to prior concepts variously called vats, workers, processes, tanks, or grains. We adopt the "vat" terminology here for expository purposes. Vats are only asynchronously coupled to each other, and represent the minimal possible unit of concurrency, transparent distribution, orthogonal persistence, migration, partial failure, resource control, preemptive termination/deallocation, and defense from denial of service. Each vat consists of

- a single sequential thread of control,
- a single call-return stack,
- a single fifo queue holding *pending deliveries*,
- an internal object heap,
- and incoming and outgoing *remote references*.

A vat's thread of control dequeues the next pending delivery from the queue and processes it to completion before proceeding to the next. When the queue is empty, the vat is idle.

```
const vat = Vat(); //makes a new vat, as an object local to the creating vat.
// A Vat has an 'evalP' method that evaluates a Program in a turn of that vat.
// The 'evalP' method returns a promise for the evaluation's completion value.
```

```

const funP = vat.evalP('' + function fun(x, y) { return x + y; }); // see below
const sumP = funP ! (3, 5); // sumP will eventually resolve to 8, unless...
const doneP = vat.terminateP(new Error('die')); // that vat is terminated before
''sumP'' is resolved.
// If the vat is terminated first, then ''sumP'' resolves to a broken problem, with
// (Error: die) as its alleged reason for breakage.
// Once the vat is terminated, ''doneP'' will eventually resolve to ''true''.

```

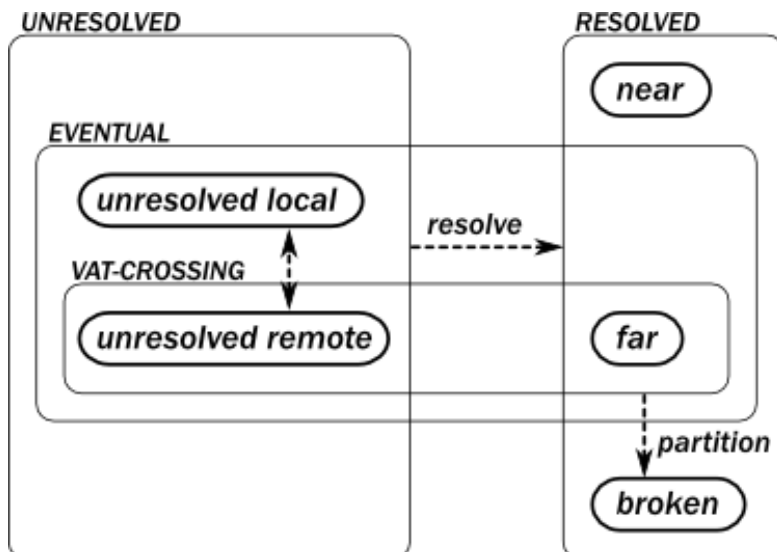
The vat object that represents a new vat is local to the creating vat, so that a vat may be terminated without waiting for that vat's pending delivery queue to drain.

The vat abstraction differs from the WebWorker abstraction, even though both are based on communicating event loops, since inter-vat messages are always directed at objects within a vat, not a vat as a whole. We intend that WebWorkers can be implemented in terms of vats and vice versa. However, when vats are built on WebWorkers, in the absence of some kind of weak reference and gc notification mechanism, it is probably impossible to arrange for the collection of distributed garbage. Even with them, *much more* is needed to enable collection of distributed cyclic garbage. On the other hand, when vats are provided more primitively, multiple vats within an address space can be jointly within the purview of a single concurrent garbage collector, enabling full gc among these co-resident vats. However, truly distributed vats would still be faced with these same distributed garbage collection worries.

The " '' + function... " trick above depends on [function\\_to\\_string](#) to actually pass a string which is the program source for the function, while nevertheless having the function itself appear in the spawning program as code rather than as a literal string. This helps IDEs, refactoring tools, etc. A vat's `evalP` method evaluates that string as a program in a safe scope – a scope containing only the standard global variables such as `Object`, `Array`, etc. Except for these, the source passed in should be *closed* – should not contain free references to any other variables. If the function is closed but for these standard globals, and these standard globals are not shadowed or replaced in the spawning context, then an IDE's scope analysis of the code remains accurate.

## Promises and Promise States

We introduce a new opaque type of object, the *Promise* to represent potentially remote references. A normal JavaScript direct reference may only designate an object within the same vat. Only promises may designate objects in other vats. A promise may be in one of several states:



- *unresolved* – when it is not yet determined what object the promise designates,

- 

- *unresolved local* – when the right to determine what the promise designates resides in the same vat,

-

*unresolved remote* – when that right is either in flight between vats or resides in a remote vat,

- 
- near* – resolved to a direct reference to a local object,
- 
- far* – resolved to designate a remote object,
- 
- broken* – will never designate an object, for an alleged reason represented by an associated error.

A promise may transition from unresolved to any state. Additionally a promise can transition from far to broken. A resolved promise can designate any non-promise value including primitives, null, and undefined. Primitives, null, undefined, and some objects are pass-by-copy. All other objects are pass-by-reference. A promise resolved to designate a pass-by-copy value is always near, i.e., it always designates a local copy of the value.

## Eventual Operations

The existing JavaScript infix `.` (dot or *now*) operator enables synchronous interaction with the local object designated by a direct reference. We introduce a corresponding infix `!` (bang or *eventually*) operator for corresponding asynchronous interaction with objects eventually designated by either direct references or promises.

Abstract Syntax:

```
Expression : ...
  Expression ! [ Expression ] Arguments // eventual send
  Expression ! Arguments // eventual call
  Expression ! [ Expression ] // eventual get
  Expression ! [ Expression ] = Expression // eventual put
  delete Expression ! [ Expression ] // eventual delete
```

The ... means "and all the normal right hand sides of this production. By "abstract" here I mean the distinction that must be preserved by parsing, i.e., in an *ast*, but without explaining the precedence and associativity which explains how this is unambiguously parsed. In all cases, the eventual form of an expression queues a pending delivery recording the need to perform the corresponding immediate form in the vat hosting the (eventually) designated object. The eventual form immediately evaluates to a promise for the result of eventually performing this pending delivery.

```
function add(x, y) { return x + y; }
const sumP = add ! (3, 5); //sumP resolves in a later turn to 8.
```

Attempted Concrete Syntax:

```
MemberExpression : ...
  MemberExpression [nlth] ! [ Expression ]
  MemberExpression [nlth] ! IdentifierName
CallExpression : ...
  CallExpression [nlth] ! [ Expression ] Arguments
  CallExpression [nlth] ! IdentifierName Arguments
  MemberExpression [nlth] ! Arguments
  CallExpression [nlth] ! Arguments
  CallExpression [nlth] ! [ Expression ]
  CallExpression [nlth] ! IdentifierName
UnaryExpression : ...
  delete CallExpression [nlth] ! [ Expression ]
```

```

    delete CallExpression [nlth] ! IdentifierName
LeftHandSideExpression :
    Identifier
    CallExpression [ Expression ]
    CallExpression . IdentifierName
    CallExpression [nlth] ! [ Expression ]
    CallExpression [nlth] ! IdentifierName

```

"[nlth]" above is short for "[No LineTerminator here]", in order to unambiguously distinguish infix from prefix bang in the face of automatic semicolon insertion.

## static Q methods

<code>get(target, name) -&gt; valueP</code>	Returns a promise for the result of eventually getting the value of the name property of target.
<code>post(target, opt_name, args) -&gt; resultP</code>	Eventually invoke the named method of target with these args. Returns a promise for what the result will be.
<code>put(target, name, value) -&gt; voidP</code>	Eventually set the value of the name property of target to value. Return a promise-for-undefined, used for indicating completion.
<code>delete(target, name) -&gt; trueP</code>	Eventually delete the name property of target. Returns a promise for the boolean result.
<code>isPromise(target) -&gt; boolean</code>	Is target a promise? If not, then using target as a target in the various promise operations is equivalent to using <code>Q.ref(target)</code> , i.e., the promise operations will automatically lift all values to promises.
<code>makePromise(promiseHandler) -&gt; promise</code>	By analogy to <code>Proxy.create</code> making and returning a fresh proxy given a proxy handler. It would be good to make these handlers more similar, but that would require proxies to distinguish between simple gets vs method calls.
<code>ref(target) -&gt; targetP</code>	Lifts the target argument into a promise designating the same object. If target is already a promise, then that promise is returned. (A promise for promise for T simplifies into a promise for T. Category theorists will be more pleased than Type theorists ;).)
<code>reject(reason) -&gt; brokenP</code>	Makes and returns a fresh broken promise recording (a sanitized form of) reason as the alleged reason for breakage. reason should generally be an immutable pass-by-copy Error object.
<code>defer() -&gt; {promise, resolve}</code>	Makes a fresh promise, resolve record, where the promise is initially unresolved-local, and the resolve method provides the rights to resolve that promise once.
<code>near(target1) -&gt; target2</code>	Returns the currently most resolved form of target1. If target1 is a fulfilled promise, return its resolution. If target1 is an unresolved or broken promise, or a non-promise, then return target1.

```
when(target, success, opt_failure) -> resultP
```

Registers functions `success` and `opt_failure` to be called back in a later turn once target is resolved. If fulfilled, call `success (resolution)`. Else if broken, call `opt_failure(reason)`. Return promise for callback result.

## Syntactic Sugar

Abstract Syntax	Expansion	Simple Case	Expansion	JSON/RESTful equiv
<code>x ! [i](y, z)</code>	<code>Q.post(x, i, [y, z])</code>	<code>x ! p(y, z)</code>	<code>Q.post(x, 'p', [y, z])</code>	POST <code>https://...q=p {...}</code>
<code>x ! (y, z)</code>	<code>Q.post(x, undefined, [y, z])</code>	-	-	POST <code>https://... {...}</code>
<code>x ! [i]</code>	<code>Q.get(x, i)</code>	<code>x ! p</code>	<code>Q.get(x, 'p')</code>	GET <code>https://...q=p</code>
<code>x ! [i] = v</code>	<code>Q.put(x, i, v)</code>	<code>x ! p = v</code>	<code>Q.put(x, 'p', v)</code>	PUT <code>https://...q=p {...}</code>
<code>delete x ! [i]</code>	<code>Q.delete(x, i)</code>	<code>delete x ! p</code>	<code>Q.delete(x, 'p')</code>	DELETE <code>https://...q=p</code>

## Examples

### Spawn

The following `spawn` function is a simple abstraction built on `Vat` and `when` that captures a simple common case:

```
const spawn(src) {
  const vat = Vat();
  const resultP = vat.evalP(src);
  Q.when(resultP,
    const(_) { vat.terminateP(new Error('done')); });
  return resultP;
}

const sumP = spawn('3+5'); // sumP eventually resolves to 8.
```

The argument string to `spawn` is evaluated in a new `Vat` spawned for that purpose. `Spawn` returns a promise for what that string will evaluate to. Once that promise resolves, the spawned `vat` is shut down.

### Infinite Queue

```
const makeQueue() {
  let {promise: front, resolve: rear} = Q.defer();
  return Object.freeze({
    enqueue: const(elem) {
      const next = Q.defer();
      rear({head: elem, tail: next.promise});
      rear = next.resolve;
    },
    dequeue: const() {
      const result = front ! head;
      front = front ! tail;
      return result;
    }
  });
}
```

Or similarly, using [classes\\_with\\_trait\\_composition](#):

```
class Queue() {
  let {promise: front, resolve: rear} = Q.defer();
  public enqueue(elem) {
    const next = Q.defer();
    rear({head: elem, tail: next.promise});
    rear = next.resolve;
  }
  public dequeue() {
    const result = front ! head;
    front = front ! tail;
    return result;
  }
}
```

In both cases, `queue.dequeue()` will return a promise for the next element that has or will be enqueued.

## race

Given a list of promises, returns a promise for the resolution of whichever promise we notice has completed first.

```
const race(answerPs) {
  const deferredResult = Q.defer();
  answerPs.forEach(const(answerP) {
    Q.when(answerP, const(answer) {
      deferredResult.resolve(answer);
    }, const(err) {
      deferredResult.resolve(Q.reject(err));
    });
  });
  return deferredResult.promise;
}
```

## Timeouts

```
const delay(millis, answer = undefined) {
  const deferredResult = Q.defer();
  setTimeout(const() { deferredResult.resolve(answer); }, millis);
  return deferredResult.promise;
}

// timeout an eventual request
var answer = race(bob ! foo(carol),
  delay(5000, Q.reject(new Error("timeout"))));
```

## allFulfilled

Often it's useful to collect several promised answers, in order react either when *all* the answers are ready or when *any* of the promises becomes broken.

```

const allFulfilled(answerPs) {
  let countDown = answerPs.length;
  const answers = [];
  if (countDown === 0) { return answers; }
  const deferredResult = Q.defer();
  answerPs.forEach(const(answerP, index) {
    Q.when(answerP, const(answer) {
      answers[index] = answer;
      if (--countDown === 0) { deferredResult.resolve(answers); }
    }, const(err) {
      deferredResult.resolve(Q.reject(err));
    });
  });
  return deferredResult.promise;
}

```

## Eventual Equality

```

const join(xP, yP) {
  return Q.when(allFulfilled([xP, yP]), const([x, y]) {
    if (Object.identical(x, y) {
      return x;
    } else {
      throw new Error("not the same");
    }
  });
}

```

## See

threads suck

Concurrency Among Strangers and Part III of Robust Composition

Ambient References: Object Designation in Mobile Ad hoc Networks

NodeJS

ref\_send and web\_send

CommonJS Promises/B with implementation as node npm package.

Similar to ref\_send, but connection-oriented and symmetric

CapTP and caja-captp

Causeway, a message oriented distributed debugger

JCoBox (has a formal semantics of a similar model)

Towards Fearless Distributed Computing

Orleans: A Framework for Cloud Computing with video. Starts with the same general model – async messages returning promises, vats (called grains) processing the reception of such messages in sequential turns that run to completion, pipelined polymorphism between promises and far references. Adds scalability by on-demand grain replication and optimistic reconciliation.





[[strawman:  
simple\_module\_functions]]Trace: »  
inherited\_explicit\_soft\_fields »

names\_vs\_soft\_fields » quasis » concurrency » simple\_module\_functions

## Simple Module Functions

**Table of Contents** ▲

- Simple Module Functions
  - The problem
  - The proposal
- Module parameters
- See
  - Reasons to avoid mutable static state

### The problem

Here we propose an enhancement to **simple modules** to cope with the problems created by its combination of mutable static state and the second-classness of its modules. To understand the problem, consider its example of **shared state**:

```
module Counter {
  var counter = 0;
  export function increment() { return counter++ }
  export function current() { return counter }
}
```

This could be imported by

```
import Counter. {increment, current};
```

or loaded by

```
module Counter = /*MRL or Counter.js*/;
```

(Note that the simple modules strawman may change the concrete syntactic details shown above, but this does not affect the points made here.)

This is similar to the following pre-module code, intended to be "imported" by evaluating it and using its completion value as the module instance:

```
// Counter.js
(function(){
  var counter = 0;
  return Object.freeze({
    increment: function() { return counter++ },
    current: function() { return counter }
  });
})();
```

Say this text has already been fetched by XHR and stored in the variable `CounterSrc`. The corresponding load operation would then be

```
var Counter = eval(CounterSrc);
```

In almost all ways, the new module-based way of doing this is better than the old eval-based way. (Both are better than present practice of linkage via global variables.) However, the new module-based way does have one comparative problem.

In both cases, the original Counter module has made a choice that **many consider bad practice**: the use of top level static mutable state. The customer of the Counter module may wish to avoid this choice in the system into which they import the Counter module, by nesting the load within a multiply instantiable context, such as a function body:

```
function Something() {
  //...
  module Counter = /*MRL or Counter.js*/;
  //...
}
```

or

```
function Something() {
  //...
  var Counter = eval(CounterSrc);
  //...
}
```

However, the first module-based nested load is illegal, in order to preserve the second-class nature of modules and be able to report early errors, as explained in the modules strawman.

The design of **simple modules** makes it possible to multiply instantiate a module by using **dynamic evaluation**:

```
function Something() {
  CurrentModuleLoader.evalSrc(
    "/*...*/" +
    "module Counter = /*MRL or Counter.js*/;" +
    "/*...*/");
}
```

The normal early errors here are postponed until the evalSrc happens. The problem is that the above synchronous evalSrc is not actually possible (without further mechanism) under the normal event-loop concurrency constraints, because the module can no longer static tell that the importing "Something" module synchronously depends on the contents of /\*MRL or Counter.js\*/, and therefore cannot know that it needs to prefetch it before execution begins.

## The proposal

We enhance the grammar at **syntax** with the following additional productions. In all cases, "... " means the present right hand side of an existing production.

```
ModuleDeclaration ::= ... | 'module' ModuleFunctionDefinition

ModuleFunctionDefinition ::= Identifier '(' (ModuleParameter (',' ModuleParameter)
*)? ')' '{' ModuleElement* '}'

ModuleParameter ::= FormalParameter | 'module' FormalParameter

QualifiedPath ::= ... | ModuleFunctionCall

ModuleFunctionCall ::= QualifiedPath Arguments
```

A module function is a parameterized module definition. A module function call to a module function results in a module instance. By analogy with parameterized types, we can see that this need not violate the second class nature of modules. Regarding the time of error reporting, the body of a module function is still like the previous evalSrc call – a "static" module error within the module function body is thrown on entry to the module function.

We can now express our previous example as

```

module Something() {
  //...
  module Counter = /*MRL or Counter.js*/;
  //...
}

```

Using an explicit abstraction form is notationally more pleasant than combining an eval form and an inline literal string. We also see a more fundamental advantage: It is now statically apparent that the importing “Something” module synchronously depends on */\*MRL or Counter.js\*/*, and so the module system can know to fetch the code for the latter before starting execution of the former.

Note that the fundamental advantage above may well get fixed by simple modules by other means, in which case simple module functions provides only notational advantages over the `evalSrc` pattern.

## Module parameters

---

(Rough – to be rewritten) Module loaders also explains [module registration](#), how to dynamically demote a first class value to a second class module instance, by use of `attachModule`. In our grammar above, a module parameter annotated with `module` serves the same purpose. To a module function’s caller, this is just a normal parameter, for which the caller should provide a first class frozen object as argument. The module function does the equivalent of `attachModule` of these arguments around the equivalent of `evalSrc`ing the module function body. This has the effect of binding the parameter name as a module name to this object as module instance.

## See

---

[simple modules](#)

[simple modules examples](#)

[module loaders](#)

## Reasons to avoid mutable static state

---

[Singletons Considered Harmful](#) by Kenton Varda.

[Cutting Out Static](#) by Gilad Bracha. [Discussion on Lambda the Ultimate](#).

[Paradigm Regained: Abstraction Mechanisms for Access Control](#) by Mark Miller.

[Joe-E: A Security-Oriented Subset of Java](#) by Adrian Mettler, David Wagner, and Tyler Close.

[Root Cause of Singletons](#) by Miško Hevery.

[Why Singletons Are Controversial at the Google Singleton Detector project](#).

[Global Variables Are Bad and Singletons Are Evil](#) on the c2 wiki.

[Why Singletons are Evil perhaps](#) by Scott Densmore.

[Re-engineering global variables in Ada](#) by Sward and Chamillard.

[Static is Evil in ColdFusion](#).



# [[strawman:random-er]]

Trace: » [names\\_vs\\_soft\\_fields](#) » [quasis](#) » [concurrency](#) » [simple\\_module\\_functions](#) » [random-er](#)

Upgrade the specification of `Math.random()` to require a cryptographically strong random number generator. If that doesn't take, at least decouple the random generators per global context (e.g., per browser frame), so the numbers emitted within one context provide no information about random state in a different context.

## References

---

[Mozilla bug 322529](#)

[David Wagner's links on randomness](#)

[Wikipedia's Cryptographically secure pseudorandom number generator](#)

strawman/random-er.txt · Last modified: 2010/06/05 07:02 by markm



# [[strawman: function\_to\_string]]

Trace: » [quasis](#) » [concurrency](#) » [simple\\_module\\_functions](#) » [random-er](#) » [function\\_to\\_string](#)

## 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()`.

#### 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](#)

•

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 `pN` 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/function\_to\_string.txt · Last modified: 2010/09/19 19:40 by markm





# [[strawman:simple\_maps\_and\_sets]]

Trace: » concurrency » simple\_module\_functions » random-er » function\_to\_string » simple\_maps\_and\_sets

## Simple Maps and Sets

Similar in style to [weak maps](#) but without the funny garbage collection semantics or non-enumerability. Depends on the [iterators](#) and [egal](#) proposals. Depends on the [classes\\_with\\_trait\\_composition](#) only for expository purposes.

### Map

Given

```
/** A non-stupid alternative to Array.prototype.indexOf */  
function indexOfIdentical(keys, key) {  
  for (var i = 0; i < keys.length; i++) {  
    if (Object.is(keys[i], key)) { return i; }  
  }  
  return -1;  
}
```

Executable spec

```
class Map {  
  private keys, vals;  
  constructor() {  
    private(this).keys = [];  
    private(this).vals = [];  
  }  
  get(key) {  
    const keys = private(this).keys;  
    const i = indexOfIdentical(keys, key);  
    return i < 0 ? undefined : private(this).vals[i];  
  }  
  has(key) {  
    const keys = private(this).keys;  
    return indexOfIdentical(keys, key) >= 0;  
  }  
}
```

```

set(key, val) {
  const keys = private(this).keys;
  const vals = private(this).vals;
  let i = indexOfIdentical(keys, key);
  if (i < 0) { i = keys.length; }
  keys[i] = key;
  vals[i] = val;
}
delete(key) {
  const keys = private(this).keys;
  const vals = private(this).vals;
  const i = indexOfIdentical(keys, key);
  if (i < 0) { return false; }
  keys.splice(i, 1);
  vals.splice(i, 1);
  return true;
}
// todo: iteration
}

```

## Set

### Executable Spec

```

class Set {
  private map;
  constructor() {
    private(this).map = Map();
  }
  has(key) { return private(this).map.has(key); }
  add(key) { private(this).map.set(key, true); }
  delete(key) { return private(this).delete(key); }
  // todo: iteration
}

```

strawman/simple\_maps\_and\_sets.txt · Last modified: 2011/05/17 06:57 by markm

[RSS](#) [XML FEED](#)



[[strawman:  
scoped\_object\_extensions]]Trace: »  
simple\_module\_functions

» random-er » function\_to\_string » simple\_maps\_and\_sets » scoped\_object\_extensions

## Scoped Object Extensions

Scoped object extensions allows different libraries to define and reuse monkey patches without conflicting with each other.

### Goals

- Allow property extensions to any object
- Extensions are only available in lexical scopes where they have been explicitly defined or imported
- When in scope property extensions are indistinguishable from normal properties
- Extensions may be used to extend objects to support interfaces that non-extended objects support. Extensions can be used to support duck-type polymorphism between extended and non-extended objects.

#### Table of Contents

- Scoped Object Extensions
  - Goals
  - Examples
  - Related Work
  - Syntax
  - Extension Definition
  - Exporting Extensions
  - Importing Extensions
  - Multiple Extensions
  - Lexical Scope
  - Property Lookup
  - Property Lookup Spec Changes
  - Property Iteration
  - Reflective API for Extensions
  - Implementation Notes

### Examples

Extensions add properties to an object. Extensions are specified declaratively as part of a module declaration. The properties added via an extension are only visible within the module within which the extension is declared.

```

module {
  extension Array.prototype {
    where: Array.prototype.filter,
    select: Array.prototype.map
  }

  // extensions are in scope in their defining module
  var evens = [1, 2, 3, 4].where(function (value) { return {value %2} == 0; });
  alert(typeof([].where)); // function
  alert(typeof(Array.prototype.select)); // function
}
// extensions are not in scope outside the lexical scope of the module
alert(typeof([].where)); // undefined
alert(typeof(Array.prototype.select)); // undefined

```

Extensions can be exported and imported across modules:

```

module Collections {
  export extension Extensions = Array.prototype {
    where: function(condition) { ... }
    select: function(transformer) { ... }
  }
}

module LolCatzDotCom {
  // imports Array.prototype extensions where and select into this module
  import Collections.Extensions;

  var allCatz = someArrayValue;
  // Array extensions are in scope
  var cuteCatz = allCatz.where(function(cat) { return cat.isCute; });

  alert(typeof([].where)); // function
}

```

Extension properties appear to the programmer the same as any regular property defined on an object. In the above example, allCatz might be an array, or it might be any other object which contains a suitable 'where' property. Extensions may be used in this way to introduce duck type polymorphism.

Any object may be extended:

```

module DOMExtensions {
  extension window {
    width: function () { return this.innerWidth; }
  }

  alert(window.width());
}

```

## Related Work

Very similar to Ruby refinements which are based on ClassBlocks. Open Classes from MultiJava. C# extension methods.

## Syntax

Scoped object extension syntax builds on the syntax from the modules proposal.

```

ExtensionDeclaration ::= "extension" [Identifier "="] Expression ObjectLiteral
ExportDeclaration ::= ...
                    | export ExtensionDeclaration
ModuleElement ::= ...
                | ExtensionDeclaration

```

## Extension Definition

Object extensions are defined by ExtensionDeclarations:

```
ExtensionDeclaration ::= "extension" [Identifier "="] Expression ObjectLiteral
```

The `Expression` identifies the object being extended. The `Expression` is evaluated once at module startup (TODO: compile-time, link-time, run-time). The `ObjectLiteral` specifies the extension object containing the extension properties for the extended object. Extension objects are frozen and are prototype-less.

If the `Identifier` is present in an `ExtensionDeclaration` then the extension is a named extension. The `Identifier` is bound to a `const` variable whose value is the extension object. TODO: Naming extensions is needed for importing/exporting them. Should the `Identifier` be in scope outside of import declarations? If not, is there a better syntax for naming extensions?

As a matter of style, it is recommended that extension names be "Extensions" where possible, or end on "Extensions". This will increase clarity when importing extensions.

## Exporting Extensions

Object extensions may be exported from a module:

```
ExportDeclaration ::= ...
                  | export ExtensionDeclaration
```

An object extension declared in an `ExportDeclaration` is exported from the containing module. An exported object extension which is named adds the named extension identifier to the set of identifiers exported by the module. Exported `ExtensionDeclarations` may be named or unnamed.

## Importing Extensions

Extensions may be imported from another module.

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

An import specifier of `*` imports all exported object extensions from the identified module - both named and unnamed extensions. If an import specifier of `IdentifierName` identifies an exported named extension then that extension is imported.

TODO: Individual extension properties may also be imported by name. Need to wrangle the syntax...

## Multiple Extensions

A module may contain multiple extension definitions for the same object. When multiple extension definitions exist for the same object, the individual extension objects are removed from the extended object and a new extension object is created. The new extension object is created by adding all the extension properties from the defining extensions in the lexical order the extensions are defined in. The new extension object is frozen and has no prototype. In the case of conflicting extension definitions to the same object with the same property name, the last definition wins. TODO: could make this an error.

A module may both import and locally define extensions for the same object. In the more general case, the conflicting extensions are removed and a new extension is added. The new extension is created by first adding all

imported extensions in lexical order of importing, then local extension definitions are added. The new extension object is frozen and has no prototype. Conflicts between imported extensions and conflicts between imported extensions and locally defined extensions are not an error. Locally defined extensions always win over imported extensions, regardless of lexical order of import and definition. TODO: Is this the right rule?

TODO: nested modules have extensions from outer modules in scope.

Inside a given lexical scope an object will have at most one extension object. Extension objects are frozen and have no prototype. The complete set of extension properties in a lexical scope can be determined statically.

TODO: This should be reworked in terms of property descriptors.

## Lexical Scope

---

Each module declaration has a unique lexical scope. The lexical scope of an expression is the lexical scope of the immediately containing module. The notion of lexical scope propagates similarly to 'strict mode'. Calls to `eval` inherit the lexical scope of their immediate caller.

## Property Lookup

---

Property lookup on an object proceeds by first doing a lookup on the extension object if one is in scope. If the object does not have an extension object, or the extension object does not have a matching named property, then lookup proceeds normally on the object. When an object extension is in scope, the extension properties win. Note that all objects may be extended.

Given:

```
var P = {};
var O = Object.create(P);
```

The expression `O.M` searches for a property `M` in the following order:

1. extension object for `O`
2. `O`
3. extension object for `P`
4. `P`
5. extension object for `Object.prototype`
6. `Object.prototype`

## Property Lookup Spec Changes

---

Section 8.12.1 `[[GetOwnProperty]](P)` is modified as follows:

When the `[[GetOwnProperty]]` internal method of `O` is called with property name `P` and lexical scope `L`, the following

steps are taken:

1.  
Let D be the result of calling `[[GetExtensionProperty]]` on object O with property name P and lexical scope L.
2.  
If D is not undefined, return D.
3.  
Else return `[[GetUnextendedOwnProperty]]` on object O with property name P.

When the `[[GetExtensionProperty]]` internal method of O is called with property name P and lexical scope L, the following steps are taken:

1.  
If O doesn't have an object extension in lexical scope L return undefined.
2.  
Else let E be the object extension for O in lexical scope L.
3.  
Return `[[GetUnextendedOwnProperty]]` with object E and property name P.

When the `[[GetUnextendedOwnProperty]]` internal method of O is called with property name P, the following steps are taken: NOTE: this is just the old version of `[[GetOwnProperty]]`

1.  
If O doesn't have an own property with name P, return undefined.
2.  
Let D be a newly created Property Descriptor with no fields.
3.  
Let X be O's own property named P.
4.  
If X is a data property, then
  1.  
Set D.`[[Value]]` to the value of X's `[[Value]]` attribute.
  2.  
Set D.`[[Writable]]` to the value of X's `[[Writable]]` attribute
5.  
Else X is an accessor property, so
  1.  
Set D.`[[Get]]` to the value of X's `[[Get]]` attribute.
  2.  
Set D.`[[Set]]` to the value of X's `[[Set]]` attribute.

**6.**

Set D.`[[Enumerable]]` to the value of X's `[[Enumerable]]` attribute.

**7.**

Set D.`[[Configurable]]` to the value of X's `[[Configurable]]` attribute.

**8.**

Return D.

A consequence of the changes to `[[GetOwnPropertyDescriptor]]` is that extensions apply to MemberExpressions (both ``.`` and ``['` operators), internal methods from 8.12 (Get, CanPut, Put, HasProperty, ...), as well as `Object.getPrototypeOf`. The lexical scope will need to be propagated from all reflection APIs (Get, CanPut, Put, HasProperty, ...) down to `[[GetOwnPropertyDescriptor]]`.

Extension properties are non-configurable so assignment and `Object.defineProperty` will not allow modification of extension properties.

Alternative:

An alternative design is to only apply extension lookup to the ``.`` and ``['` operators.

## Property Iteration

---

Extension properties participate in object property iteration.

In sections:

- 
- 12.6.4 The for-in Statement
- 
- 15.2.3.4 `Object.getOwnPropertyNames`
- 
- 15.2.3.14 `Object.keys`

Extension properties are included in property iterations and shadow non-extension properties of the same name. When iterating over the properties of an object, if an object has both an extension property and a regular (non-extension) property then the iteration will proceed as if the non-extension property is not present on the object.

TODO: formalize this section.

## Reflective API for Extensions

---

TODO: Need API on `Object` to access `[[GetUnextendedOwnProperty]]` and `[[GetExtensionProperty]]` directly.

## Implementation Notes

---

- 
- An existing compiling implementation will not need to change its code gen for legacy code because it can determine statically that no object extensions are in scope.
-



The set of extension objects in scope is a compile time constant. Extensions objects are frozen and the set of extensions is specified declaratively. Taking advantage of this fact should allow implementations to minimize the performance impact.



[[strawman:  
deferred\_functions]]Trace: » random-er » function\_to\_string  
» simple\_maps\_and\_sets »

scoped\_object\_extensions » deferred\_functions

## Deferred Functions

Deferred functions ease the burden of asynchronous programming.

### Asynchronous Programming

EcmaScript programming environments typically are single threaded and pausing execution for long periods is undesirable. ES host environments use callbacks for operations which may take a long time like network IO or system timers.

For Example:

```
function animate(element, callback) {
  var i = -1;
  function continue() {
    i++;
    if (i < 100) {
      element.style.left = i;
      window.setTimeout(continue, 20);
    } else {
      callback();
    }
  }
  continue();
};
animate(document.getElementById('box'), function() { alert('Done!'); });
```

Calling functions which have callback arguments does not compose easily. Many libraries include a Deferred constructor function to address this issue.

```
function deferredTimeout(delay) {
  var deferred = new Deferred();
  window.setTimeout(function() {
    deferred.callback({});
  },
  delay);
  return deferred;
}

function deferredAnimate(element) {
  var i = -1;
  var deferred = new Deferred();
  function continue() {
    i++;
    if (i < 100) {
      element.style.left = i;
      deferredTimeout(20).then(continue);
    } else {
      deferred.callback();
    }
  }
}
```

#### Table of Contents



- Deferred Functions
  - Asynchronous Programming
  - Deferred Functions
  - Returning Values From Deferred Functions
  - Throwing From Deferred Functions
  - Deferred Pattern
  - Syntax
  - 13.2 Creating Deferred Function Objects
    - 13.2.1 Deferred Function [[Call]]
  - Deferred Object [[Continue]]
  - Await Expressions
  - Deferred Object [[Then]]
  - Deferred Object [[Cancel]]
  - Deferred Object [[Callback]]
  - Deferred Object [[Errback]]
  - Deferred Object [[CreateCallback]]
  - Deferred Object [[CreateErrback]]

```

    continue();
    return deferred;
};
deferredAnimate(document.getElementById('box')).then(function () { alert('Done!'); });

```

The Deferred API pattern improves composability of callback patterns but there are still drawbacks in programming in this style. The completion of the computation must be enclosed in a callback function passed to the 'then' function.

Authoring the callback function has proved difficult for ES programmers. Control flow constructs (if, while, for, try) do not compose across function boundaries. The programmer must manually twist the control flow into continuation passing style. The callback function does not by default have the same 'this' binding as the enclosing function which is a frequent source of programmer error.

## Deferred Functions

Deferred functions allow asynchronous code to be written using existing control flow constructs.

```

function deferredTimeout(delay) {
    var deferred = new Deferred();
    window.setTimeout(function() {
        deferred.callback({
            called: true
        });
    },
    delay);
    return deferred;
}

function deferredAnimate(element) {
    for (var i = 0; i < 100; ++i) {
        element.style.left = i;
        await deferredTimeout(20);
    }
};
deferredAnimate(document.getElementById('box')).then(function () { alert('Done!'); });

```

This proposal adds 'await expression' syntax, a new kind of expression. A function containing an 'await expression' is a deferred function.

The 'await expression' evaluates the expression. The result of evaluating the expression in an 'await expression' is the 'awaited object'. The expectation is that the 'awaited object' supports the 'Deferred pattern' common to many ES libraries. After computing the 'awaited object' the 'await expression' suspends execution of the current function, attaches the continuation of the current function to the 'awaited object' by calling its then function, and then returns.

The return value of a 'deferred function' is itself a 'deferred object'. Deferred objects support the 'deferred pattern'. Deferred objects have a then function which allows registration of callbacks. When the deferred object completes its computation all callbacks registered to the then function are invoked.

## Returning Values From Deferred Functions

Deferred functions may return a value. When a deferred function completes by returning a value, the returned value is passed as the argument when invoking callbacks registered to the deferred object's then function. The value of an await expression is the value of the argument passed to the callback registered on the awaited object's then function.

```

function deferredXHR(url) {
    var deferred = new Deferred();
    var request = new XMLHttpRequest();
    request.open('GET', url, true);
    request.send(null);
    request.onreadystatechange = function() {

```

```

        if (request.readyState == 4) {
            // call all callback's registered by deferred.then with 'request' as argument
            deferred.callback(request);
        }
    };
    return request;
}

function deferredLoadRedirectUrl(redirectUrl) {
    // redirectUrl contains another url
    var urlXHR = await deferredXHR(redirectUrl);
    var url = urlXHR.responseText;

    var valueXHR = await deferredXHR(url);
    // call all callback's registered by return value's 'then' with 'valueXHR.
    // responseText' as argument
    return valueXHR.responseText;
}

// alert the value of the redirected url
deferredLoadRedirectUrl('http://lolcatz.com/redirect').then(function (value) { alert
(value); });

```

## Throwing From Deferred Functions

The then function on a deferred object takes two arguments. The first is the callback to be invoked if the deferred object completes normally. The second is the callback to be invoked if the deferred object completes erroneously. When a deferred function completes by throwing an exception the registered error callbacks are invoked with the thrown exception as the argument.

```

// alert the value of the redirected url
deferredLoadRedirectUrl('http://lolcatz.com/redirect').then(
    function (value) { alert('Success: ' + value); },
    function (err) { alert('Failure: ' + err); });

```

Similarly, when an awaited on object completes with an error - ie. its error callback is invoked - the result of the await expression is to throw the error value.

TODO: cancelling a deferred function.

Open Issue: chaining the return value of callbacks/errbacks.

## Deferred Pattern

An object implementing the deferred pattern represents a computation which will complete at a later time. For example, the completion of an XHR. The deferred pattern contains two methods 'then' and 'cancel':

**then:** function(callback, errback)

The 'then' function adds a listener to the completion of the deferred object's computation. When the deferred object completes its computation it will notify all registered listeners. If the completed computation succeeds, then the 'callback' entry of each listener will be invoked with the result of the completed computation as its argument. If the completed computation fails (throws an exception), then the 'errback' entry of each listener will be invoked with the error of the completed computation as its argument. If the deferred object's computation has already completed then the 'then' function will immediately call the 'callback' or 'errback' with the result of the computation.

**cancel:** function()

The 'cancel' function attempts to cancel the computation in progress. If the computation has not completed, then all listeners will be notified as if the computation completed with a 'CancelledError'. If the computation has already completed, then the 'cancel' method throws an 'AlreadyCompletedError'.

TODO: Error names.

TODO: Alternative Pattern: **addCallback**, **addErrback** and **addCallbacks** in lieu of **then**.

## Syntax

---

Deferred functions adds 'await expression' a new primary expression:

TODO:

Need to work the syntax. 'await' as a keyword will likely not fly. An alternative is to add a modifier on the deferred function and make 'await' a contextual keyword only within deferred functions.

```
var f = function () { await(1); } // regular function. 'await' is an identifier.
var df = deferred function () { await(1); } // deferred function. 'await' is a keyword.
```

/TODO

```
PrimaryExpression ::= ...
                    AwaitExpression

AwaitExpression ::= "await" Expression
```

It is an error for an `AwaitExpression` to occur in the finally block of a try statement. It is an error for a function to be both a deferred function and a [generator function](#).

## 13.2 Creating Deferred Function Objects

---

A function containing an `AwaitExpression` is a deferred function. When creating a function object for a 'deferred function' via 13.2, bullet 6 is replaced by the below.

### 13.2.1 Deferred Function `[[Call]]`

---

When the `[[Call]]` internal method for a Deferred Function object `F` is called with a `this` value and a list of arguments, the following steps are taken:

1.
  - Let `funcCtx` be the result of establishing a new execution context for function code using the value of `F`'s `[[FormalParameters]]` internal property, the passed arguments `List args`, and the `this` value as described in 10.4.3.
2.
  - Create a new object and let `D` be that object. Call `D` a 'Deferred Object'.
    1.
      - Set the internal methods of `D` as described in 8.12.
    2.
      - Set the `[[Class]]` internal property of `D` to "Object"
    3.
      - Set the `[[Prototype]]` internal property of `D` to `Object.prototype`.
    4.
      - Set the `[[ExecutionContext]]` of `D` to `funcCtx`. TODO: This includes variable environment. Does it also include, the function code and current IP? I'm assuming yes, otherwise those need to be included in the state of a Deferred Object.

Set the `[[State]]` internal property of D to "newborn".

6.

Set the `[[Listeners]]` internal property of D to a new empty array.

7.

Set the `[[Then]]` internal property of D to as specified below.

8.

Set the `[[Cancel]]` internal property of D to as specified below.

9.

Set the `[[Continue]]` internal property of D to as specified below.

10.

Set the `[[Callback]]` internal property of D to as specified below.

11.

Set the `[[Errback]]` internal property of D to as specified below.

12.

Set the `[[CreateCallback]]` internal property of D to as specified below.

13.

Set the `[[CreateErrback]]` internal property of D to as specified below.

14.

Set `D.then` to the `[[Then]]` internal property of D.

15.

Set `D.cancel` to the `[[Cancel]]` internal property of D.

3.

Evaluate the `[[Continue]]` property of D.

4.

Return D. D is an object which satisfies the Deferred Pattern.

## Deferred Object `[[Continue]]`

---

When the `[[Continue]]` internal method of deferred object D is called, the following steps are taken:

1.

If the `[[State]]` property of D is "running", or "finished" then throw.

2.

Set the `[[State]]` internal property of D to "running".

3.

Set the current execution context to the `[[Execution Context]]` internal property of D.

4.

If the `[[State]]` property is "newborn", then begin executing at the start of the Deferred Function.

5.

Otherwise the `[[State]]` property is "suspended", continue execution after the point of suspension.

6.

Execution will continue until an `await` expression is encountered or the function terminates.

7.

If an `await` expression is encountered, evaluate the expression as below.

8.

Otherwise, if the function terminates with a `(return, value, empty)`, then invoke the `[[Callback]]` internal method of `D`.

9.

Otherwise, the function terminates with a `(throw, value, empty)`. Invoke the `[[Errback]]` internal method of `D`.

10.

Return `(return, undefined, empty)`.

## Await Expressions

---

Await expressions may only appear in deferred functions, and can only be evaluated during the execution of a deferred object's `[[Continue]]` internal method.

The production `AwaitExpression`: **await** `Expression` is evaluated as follows:

1.

Let the deferred object of the `[[Continue]]` method be `D`.

2.

Let `e` be the result of evaluating `Expression`.

3.

Set the `[[ExecutionContext]]` internal property of `D` to the current execution context.

4.

Set the `[[State]]` internal property of `D` to "suspended".

5.

Let `callback` be the result of invoking the `[[CreateCallback]]` internal method of `D`.

6.

Let `errback` be the result of invoking the `[[CreateErrback]]` internal method of `D`.

7.

Call the `then` method on `e` with arguments `(callback, errback)`.

8.

Return `(return, undefined, empty)` from the currently executing `[[Continue]]` method.

## Deferred Object `[[Then]]`

---

## Deferred Object `[[Cancel]]`

---

## Deferred Object `[[Callback]]`

---

## Deferred Object `[[Errback]]`

---

## Deferred Object [[CreateCallback]]

---

## Deferred Object [[CreateErrback]]

---

strawman/deferred\_functions.txt · Last modified: 2011/04/27 22:07 by peterhal





[[strawman:  
guards]]

Trace: » [function\\_to\\_string](#) » [simple\\_maps\\_and\\_sets](#) » [scoped\\_object\\_extensions](#) » [deferred\\_functions](#) » [guards](#)

## Guards

For checking purposes, guards are used as the runtime analog of types. Use guards to annotate variable and parameter declarations, function return results, and property definitions. Each guard is asked to approve an incoming value, the *specimen*, to determine whether to let it pass or reject it. Examples of uses of guards include:

```
let x :: Number = 37;
function f(p :: String, q :: MyType) :: Boolean { ... }
let o = {a :: Number : 42, b: "b"};
```

Guards do not do any coercion. It is the intent that removing guards will not turn a correct program into an incorrect program. There are ways to write programs that violate this guideline by either catching type error exceptions or writing GuardExpressions that produce visible side effects, but both are considered poor style.

```
Guard :
  :: GuardExpression
GuardExpression :
  ConditionalExpression
```

A Guard annotation evaluates its GuardExpression to a value in that scope in the usual manner. (See clarifications below for which scope that is, and when the evaluation occurs.) The GuardExpression should evaluate to an object with a [[Brand]] internal property which is then consulted to determine whether to pass the specimen or not. In detail:

1.

Let *t* be the result of evaluating GuardExpression.

2.

If *t* is not an object or doesn't have a [[Brand]] internal property, throw a TypeError. The [[Brand]] lookup is done on *t* only; it does not follow *t*'s prototype chain.

3.

### Table of Contents



- Guards
  - Guarding Variables
  - Guarding Parameters and Results
  - Guarding Properties
- Open Issues
- See

Let  $brand = t.[[Brand]]$  and save it for the guard check.

Later, when checking a specimen  $s$ :

1.

Call  $brand.validate(s)$ . This will either let  $s$  pass, in which case it has no side effects, or throw a `TypeError`.

See the [trademarks](#) proposal for the means of creating objects with a `[[Brand]]` property and its semantics.

## Guarding Variables

---

Here, we define the `ConstDeclaration` and `LetDeclaration` from [block\\_scoped\\_bindings](#) to generalize the defining position from an `Identifier` to a `Pattern`. This is a somewhat different factoring than the `Pattern` at [destructuring](#), but the “...” below should include all the `Pattern` productions there.

We do it this way, rather than extend the ES5 `VariableDeclaration`, since we are not trying to enhance the deprecated `VariableStatement`.

```
ConstDeclaration :
    const Pattern = AssignmentExpression
LetDeclaration :
    let Pattern = AssignmentExpression
Pattern : ... //
    Identifier Guard?
```

The guard expression, if present, is evaluated in left to right order, and therefore before the `AssignmentExpression` initializer.

A guarded `const` variable simply inserts an initialize-time check — the value of the *brand* guard is used immediately to check the initializer.

A guarded `let` variable inserts the corresponding check on initialization and on all assignments to the variable. Therefore, reading the variable, which is not translated, can only either

- throw a `ReferenceError` if read before initialization, or
- yield a value which the guard considers acceptable.

There intentionally is no way to guard `var` variables. Doing that would be problematic because

var variables can be written before their definitions are evaluated and there can be multiple definitions for the same var variable.

## Guarding Parameters and Results

```

FunctionDeclaration :
  function Identifier FunctionHead { FunctionBody }
  const Identifier FunctionHead { FunctionBody }
FunctionExpression :
  function Identifier? FunctionHead { FunctionBody }
  const Identifier? FunctionHead { FunctionBody }
FunctionHead :
  ( FormalParameterList? ) Guard?

FormalParameterList :
  FormalParameter
  FormalParameterList , FormalParameter
  // extend for optional and rest parameters
FormalParameter :
  const? Pattern
Pattern : ...
  Identifier Guard?

```

When a FormalParameter variable is annotated with a Guard, the corresponding argument value is first checked via the guard before binding to the parameter. When a Guard appears after the close paren of a FunctionHead, then whatever value would be returned is checked via the guard before being returned.

Guard expressions on function parameters and results are *not* evaluated when the function is called, but rather when the function definition is evaluated. Thus, they are *not* evaluated in the scope defined by this function, but rather in the scope in which this function definition itself is evaluated. The resulting guard values are therefore captured by the created function.

## Guarding Properties

The reason our guard syntax uses ":" rather than the ":" of ES4 is so that we can annotate property definitions in object literals.

```

PropertyAssignment : ...
  PropertyName Guard? : AssignmentExpression

```

As with function parameter and result guards, literal property guards are evaluated in the scope in which the object literal appears, and are evaluated prior to evaluating the object literal.

Here we have a choice of either making the guard an attribute of the annotated property or

turning the annotated property into an accessor property whose setter enforces the guard.

## Open Issues

---

- What, if any, syntax do we want to integrate guards with classes? A lot depends on the choice of class proposal.
- Should guarded properties be enforced by attributes or be turned into accessors?

## See

---

[trademarks](#)

[classes with trait composition](#)

[email thread](#)

strawman/guards.txt · Last modified: 2011/05/11 22:16 by brendan

[RSS](#) [XML FEED](#)



Trace: » [simple\\_maps\\_and\\_sets](#) » [scoped\\_object\\_extensions](#)  
 » [deferred\\_functions](#) » [guards](#) » [trademarks](#)

## Trademarks

A trademark represents a generative nominal type. A trademark has two facets:

- A trademark's **brander** allows one to brand objects or sets of objects with that trademark. Objects can be branded by a trademark when they're created or at any later time, but a brand is irrevocable: Once an object is branded by a trademark *tm*, it is forever branded by trademark *tm*.
- A trademark's **guard** allows one to test a specimen object to see if it is branded by that trademark. The companion [guards](#) proposal provides convenient syntax for doing such tests.

To make brands unforgeable, the two facets of a trademark are represented by different ECMAScript objects, linked to each other under the scenes. Given the brander one can readily create a guard. On the other hand, one cannot obtain the brander given just the guard of a trademark. Thus the brander of a trademark is a capability.

Some of the API relating to trademarks also consists of static global methods. These are gathered into a `Type` object, which is a collection of static methods analogous to `Math` or `JSON`. `Type` is imported into scope using the module mechanism.

### Brander

The branding facet of a trademark is an object with the following frozen methods. Here *specimen* is an arbitrary ECMAScript value and *guard* is a guard.

- - `test(specimen)`
    - Returns either `true` or `false` indicating whether *specimen* has been branded with this trademark.

#### Table of Contents



- Trademarks
  - Brander
  - Guard
  - Type
  - Built-In Trademarks
- Alternatives and Open Issues
- Bibliography

- `validate(specimen)`
  - If `test(specimen)` returns `true`, return `specimen`. Otherwise throw a `TypeError`.
- `brand(specimen)`
  - Brand `specimen` with this trademark. Once branded, `specimen` is branded forever. Branding the same `specimen` again has no effect.
- `brandPrototype(specimen)`
  - Brand `specimen` and all, current or future, objects with `specimen` on their prototype chain with this trademark. Once branded, `specimen` is prototype-branded forever. Branding the same `specimen` again has no effect.
- `append(guard)`
  - Derive this trademark from `guard`'s trademark. Every object branded by `guard`, either now or in the future, automatically acquires this trademark's brand as well.
- `isGuard(guard)`
  - If `guard` is a guard whose `[[Brand]]` internal property is this brander, return `true`. Otherwise return `false`.
- `setGuard(guard)` (optional, not needed if the facilities for specifying that an object is a guard at creation time are sufficient)
  -

If *guard* is not an extensible object or is already a guard (i.e. has a `[[Brand]]` internal property), throw an error. Otherwise create a `[[Brand]]` internal property on *guard* and set it to this brander.

Branderers are created by `Type.make` below.

## Guard

---

The guarding facet of a trademark is an object with a `[[Brand]]` internal property. There are two ways to set this property to a brander *b* on a guard *g*:

- Set *g*'s `[[Brand]]` internal property to *b* at the time *g* is created using whatever syntax or API is approved for setting metaproperties during object creation.
- Call `b.makeGuard(g)` (not needed if the above mechanism is adequate)

Once *g*'s `[[Brand]]` is set, it cannot be deleted or changed. Furthermore, user code cannot directly read the `[[Brand]]` internal property.

Guards are typically used for constraining the values of variables, fields, or parameters as part of [guards](#):

```
let x :: guard = value;
function f(a :: guard1, b :: guard2, c) :: resultGuard {...}
```

One can also invoke a guard manually using `Type` methods. These methods are on `Type` instead of directly on the guard to make it easy to make any object into an guard without polluting its property name space. In particular, `Object`, `Number`, `String`, etc. can serve as built-in guards.

The `~` and `|` operators have special behavior when their operands are guards. When given guards as both operands, the `|` operator invokes `Type.union` on the operands. When given a guard as the operand, the `~` operator invokes `Type.union` on the guard with `Null` and `Void`. These let one conveniently make guard annotations such as:

```
let x :: Number | Boolean = value; // Either a Number of a Boolean
let y :: ~Number = value; // Either a Number or null or undefined
```

## Type

---

The `Type` global object is a collection of static methods for working with branders and guards. It provides the following methods:

- `make()`
  - Create and return a new brander, initially with no branded specimens.
- `make(guard1, guard2, ...)`
  - Same as `make` to create a new brander *b* followed by a series of *b.append* calls to append all of the given guards to *b*. This will make the new brander the union of the guards' types.
- `union(guard1, guard2, ...)`
  - Same as `make` above, but returns the guard, not the brander. Other than containing the `[[Brand]]` internal property, the returned guard object is like an object created by `{ }`. There is no way to access the brander.
  - Open issues: Should the returned object be frozen? If so, can calling this multiple times return the same object?
- `test(guard, specimen)`
  - Return either `true` or `false` indicating whether *specimen* has been branded with *guard*'s trademark. Throw an error if *guard* is not a guard. `test` invokes `guard.[[Brand]].test(specimen)`.
- `validate(guard, specimen)`
  - If `test(guard, specimen)` returns `true`, return *specimen*. Otherwise throw a `TypeError`. `validate` invokes `guard.[[Brand]].validate(specimen)`.



There intentionally is no way to make complement or difference types because doing so would violate monotonicity: if specimen  $s$  is in  $t1$  but not  $t2$ , then  $s$  will be in  $t1 - t2$ . However, it will disappear from  $t1 - t2$  if someone later brands  $s$  with  $t2$ .

## Built-In Trademarks

---

We seed the trademark graph by providing guards for built-in types:

- Null accepts null
- Void accepts undefined
- Number accepts all primitive numbers, including infinities and NaN
- Integer accepts all primitive numbers whose values are finite integers, regardless of magnitude. Integer includes both +0 and -0.
- Boolean accepts the primitives true and false
- String accepts all primitive strings
- Type accepts all guards (i.e. objects directly containing a [[Brand]] internal property)
- Object accepts all non-primitive objects. Note that null and undefined are not objects.
- Any accepts everything

Because guards introduce no extra methods, these uses do not conflict with these objects' other APIs.

## Alternatives and Open Issues

---

- Coordinate syntax of creating a guard and/or a brander with the private, class, and object initializer proposals.

- Can we take out `setGuard`?
- Do we want an `appendBrander` method on a `brander` that takes a `brander` instead of a `guard` argument?
- Do we want a way to freeze branders to prevent future calls to `brand`, `brandPrototype`, or `append`? If so, do it by piggybacking on `Object.freeze` (i.e. `brand`, `brandPrototype`, and `append` would throw an error if the `brander` is frozen) or by introducing a `freeze` method on the `brander`? In either case, knowing that a `brander` is frozen might make some code patterns easier to understand and analyze. Note that, just like for all other objects, freezing is shallow, so even a frozen `brander` can acquire new branded instances by having earlier appending a different `brander` or by deriving from a prototype-branded prototype.
- Currently `test` and `validate` will never run user code. This guarantees monotonicity — there is no way to revoke a brand. Is this desirable, or do we want to relax this restriction and add a way to write potentially problematic user-specified guards?

## Bibliography

---

[Efficient Type Inclusion Tests](#)

[Protection in Programming Languages](#)

[A Security Kernel Based on the Lambda-Calculus](#)

[Auditors: An Extensible, Dynamic Code Verification Mechanism](#)

[Guard-based Auditing](#)

[Virtual Values for Language Extension](#)

strawman/trademarks.txt · Last modified: 2011/05/06 20:37 by waldemar



# [[strawman:modulo\_operator]]

Trace: » [scoped\\_object\\_extensions](#) » [deferred\\_functions](#) » [guards](#) » [trademarks](#) » [modulo\\_operator](#)

## Modulo Operator

---

ECMAScript does not currently have a [modulo operator](#). Instead, it has a remainder operator `%`. The result of remainder takes the sign of the dividend, not the divisor, so it does not have the nice limit-and-force-to-the-positive characteristics of modulo. Programs that use remainder in place of modulo are probably in error.

So it proposed that a modulo operator `mod` be added. The result of modulo takes the sign of the divisor, not the dividend.

The `mod` keyword is contextual, and backward-compatible only if productions in which it occurs have `[no LineTerminator here]` immediately to the left of `mod`.

— *Douglas Crockford* 2011/02/13 00:51 — *Brendan Eich* 2011/03/23 20:12

We may want `div` too, for integer division. We could even consider `divmod`, returning a tuple:

```
alert(12 divmod 5) // #[2, 2]
```

— *Brendan Eich* 2011/02/16 05:15

I like that a lot. Assuming the grammar works out, I think it wins. There are other infix operators that could be added on a similar basis, like `min` and `max`.

— *Douglas Crockford* 2011/02/16 23:44

I will write this up later today. The `min` and `max` cases seem better done via the existing and variadic `Math.min` and `Math.max` functions than infix operators (e.g., `Math.min(1, 2, 3, 0)` returns 0), but we can hash that out elsewhere.

— *Brendan Eich* 2011/02/28 21:29



[[strawman:  
match\_web\_reality]]

Trace: » [deferred\\_functions](#) » [guards](#) » [trademarks](#) » [modulo\\_operator](#) » [match\\_web\\_reality](#)

## Make the RegExp Specification Match What Browsers Actually Implement

The regular expressions that web browsers interoperability implement is not what is defined in the current ES5 specification. For Harmony, the spec. should be updated to match reality.

Lasse Reichstein posted to [esdiscuss](#) the following summary of how browser reality differs the the current spec:

On Wed, 08 Dec 2010 21:43:06 +0100, Gavin Barraclough <[barraclough at apple.com](mailto:barraclough at apple.com)> wrote:

According to the ES5 spec a regular expression such as `/[\w-_]/` should generate a syntax error. Unfortunately there appears to be a significant quantity of existing code that will break if this behavior is implemented (I have been experimenting with bringing WebKit's RegExp implementation into closer conformance to the spec), and looking at other implementations it appears common for this error to be ignored.

It's far from the only extension to RegExp syntax that is common to most implementations. In fact, the extensions are both extensive and consistent across browsers. A quick check through the possible syntax errors show the following:

*Invalid ControlEscape/IdentityEscape character treated as literal. `/\z/`; Invalid escape, same as `/z/`  
Incomplete/Invalid ControlEscape treated as either `"\c"` or `"c"` `/\c/`; same as `/c/` or `/\c/`*

```
/\c2/; // same as /c2/ or /\c2/
```

*Incomplete HexEscapeSequence escape treated as either `"\x"` or `"x"`. `/\x/`; incomplete x-escape*

```
/\x1/; // incomplete x-escape
/\x1z/; // incomplete x-escape
```

*Incomplete UnicodeEscapeSequence escape treated as either `"\u"` or `"u"`. `/\u/`; incomplete u-escape*

```
/\uz/; // incomplete u-escape
/\u1/; // incomplete u-escape
```

```

/\u1z/; // incomplete u-escape
/\u12/; // incomplete u-escape
/\u12z/; // incomplete u-escape
/\u123/; // incomplete u-escape
/\u123z/; // incomplete u-escape

```

*Bad quantifier range: /x{z/; same as /x\{z/*

```

/x{1z/; // same as /x\{1z/
/x{1,z/; // same as /x\{1,z/
/x{1,2z/; // same as /x\{1,2z/
/x{10000,20000z/; // same as /x\{10000,20000z/

```

*Notice: It needs arbitrary lookahead to determine the invalidity, except Mozilla that limits the numbers.*

*Zero-initialized Octal escapes. /\012/; same as /\x0a/*

*Nonexisting back-references treated as octal escapes: /\5/; same as /\x05/*

*Invalid PatternCharacter accepted unescaped /]/; /{/; /}/; Bad escapes also inside CharacterClass.*

```

/[\z]/;
/[\c]/;
/[\c2]/;
/[\x]/;
/[\x1]/;
/[\x1z]/;
/[\u]/;
/[\uz]/;
/[\u1]/;
/[\u1z]/;
/[\u12]/;
/[\u12z]/;
/[\u123]/;
/[\u123z]/;
/[\012]/;
/[\5]/;

```

*And in addition: /[\B]/; /()[\2]/; Valid backreference should be invalid.*

None of these RegExps cause a syntax error in any of the current "top-5" browsers, even though they are (AFAICS) invalid syntax.

Most of the RegExps treat a malformed (start of a multi-character) escape sequence as a simple identity escape or octal escape, and extends identity escapes to all characters that doesn't already

have another meaning (ControlEscape, CharacterClassEscape or one of c, x, u, or b, and B outside a CharacterClass).

To match the current behavior, IdentityEscape shouldn't exclude all of IdentifierPart, but only the characters that already mean something else.

Allowing `\/c2/` to match `"c2"`, but requiring `\/CB/` to match `"\x02"` seems like it would be better explained in prose than in the BNF.

...

I'd like to propose a minimal change to hopefully allow implementations to come into line with the spec, without breaking the web. I'd suggest changing the first step of CharacterRange to instead read: 1. If A does not contain exactly one character or B does not contain exactly one character then create a CharSet AB containing the union of the CharSets A and B, and return the union of CharSet AB and the CharSet containing the one character -.

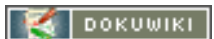
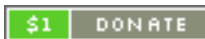
I think this matches the current actual behavior of all the browsers, and is short and understandable.

/Lasse R.H. Nielsen

---

Also note that `RegExp.prototype.compile` is implemented by web browsers and is probably essential for browser interoperability. For this reason, it probably should be added to the spec.

— *Allen Wirfs-Brock* 2011/02/01 19:54



[[strawman:  
array\_create]]

Trace: » guards » trademarks »  
modulo\_operator »

match\_web\_reality » array\_create

## Array.create

It is proposed that a `create` property be added to the `Array` constructor, being a function that is similar to `Object.create`, creating a new array that inherits from the first argument (which will usually be an object. If there is an optional second argument (which will usually be an array or array-like object) then copy the elements numbered between 0 and `length - 1` to the new array. The result is seen as an array by `Array.isArray`, and the result has a magic `length` property, but it does not necessarily inherit from `Array.prototype`.

```
myArray = Array.create(augmented_array_prototype, ['a', 'b', 'c']);
```

`Array.create` is simpler, more elemental, and shares more continuity with ES5 than [array\\_subtypes](#).

— Douglas Crockford 2011/01/03 17:48

The problem with the above solution is that it does not allow for inclusion of the properties descriptor argument that is the second argument of `Object.create`. For consistency I would like to see all constructor.create functions accept such an argument. As an alternative I played around with a flexible `Array.create` signature:

```
Array.create(proto) //creates a zero length array object with [[Prototype]] set
to proto
Array.create(proto, obj-not-array) //like Object.create
Array.create(proto, number [, properties]) //sets initial length to number +
defines own properties from property descriptor
Array.create(proto, array [, properties]) //Doug's proposal + optional properties
descriptor
```

However, note that this still requires at least one extra array creation for the new `Array(0,1,2,3,4)` use case.

Having looked at these, I think that the best alternative is to keep it simple as:

```
Array.create(proto [, properties]) //like Object.create
```

If somebody wants to support other use cases it would be easy enough for them to do things like:

```
Array.with = function(proto, args /*any number of additional args*/) {
  var arr = Array.create(proto);
  arr.push([].slice.call(arguments,1));
  return arr;
}
```

But [obj initialiser meta](#) is a better solution for this use case:

```
{<proto: myProto>, 1,2,3,4,5,6}
```



---

— Allen Wirfs-Brock 2011/03/18 18:01

strawman/array\_create.txt · Last modified: 2011/03/18 18:05 by allen



[[strawman:  
array\_subtypes]]

Trace: » trademarks »  
modulo\_operator »

match\_web\_reality » array\_create » array\_subtypes

## Array Subtypes

### Motivation

A common request from developers is for a mechanism that allows the creation of a “Array subtype”. An Array subtype is in essence an Array instance with a custom prototype chain.

### Proposal

We propose the addition of a ‘createConstructor’ function to the Array constructor object that returns a new function object with the same call semantics as the built in Array constructor, but returning an array instance with a custom prototype.

A pseudo implementation of this function would be:

```
Array.createConstructor = function() {
  var constructor = function() {
    var result = <<Initial Array Constructor>>.apply(null, arguments);
    result.__proto__ = constructor.prototype; // recognising that this isn't
actually possible in raw ES
    return result;
  }
  constructor.prototype.constructor = constructor;
  return constructor;
}
```

### Discussion

Beware the case where `arguments.length == 1 && arguments[0]` is a valid length value. Or did you intend to preserve that wart in Array? I hope not! — *Brendan Eich 2010/11/17 22:14*

I was preserving existing behaviour as it would allow code to use the Array constructor or the subtype interchangeably, but I could go either way without any real problem — *Oliver Hunt 2010/12/15 21:08*

The `result.constructor` property is instance-invariant and should be on `constructor.prototype`, as with all the built-in and user-defined constructor functions.

— *Brendan Eich 2011/01/14 01:40*

I believe I have now fixed the example code — *Oliver Hunt 2011/01/20 01:09*



[[strawman:  
proto\_operator]]Trace: » modulo\_operator »  
match\_web\_reality »

array\_create » array\_subtypes » proto\_operator

## Set Literal **[[Prototype]]** operator

**Table of Contents**

- Set Literal **[[Prototype]]** operator
  - Overview
  - Usage Examples
  - ES5 Compatability
  - Relationship to Other Proposals
  - Commentary and rationales

The `<|` operator (pronounced “prototype for”) is used in conjunction with a literal to create a new object whose **[[Prototype]]** is set to an explicitly specified value. It can be used to address several distinct use cases for which separate solutions have been proposed. Use cases include:

- Specifying an explicit **[[Prototype]]** for object literals
- Specifying an explicit **[[Prototype]]** for array literals
- “Subclassing” arrays
- Setting the prototype of a function to something other than `Function.prototype`
- Setting the prototype of `RegExp` and other built-in objects.
- Replace the most common uses of the mutable `__proto__` extension

### Overview

The `<|` operator may appears within a *MemberExpression*. Its basic syntax is:

```
MemberExpression : ...
MemberExpression <| ProtoLiteral
```

```
ProtoLiteral :
LiteralObject
LiteralValue
```

```
LiteralObject :
RegularExpressionLiteral
ArrayLiteral
ObjectLiteral
FunctionExpression
```

```
LiteralValue :
NumberLiteral
StringLiteral
BooleanLiteral
```

This basic semantics is to create a new object exactly as would normally be created by the *ProtoLiteral* except for the value of the object’s **[[Prototype]]** internal property. If the *ProtoLiteral* is a *LiteralValue* the new Object is created as if by `ToObject` except for the value of for the value of the object’s **[[Prototype]]** internal property.

The `[[Prototype]]` internal property of the new object is set to the value of the *MemberExpression*. The value of the *MemberExpression* must be **typeof** "object". The value may be **null**. A `TypeError` exception is thrown if **typeof** the *MemberExpression* is not "object".

## Usage Examples

---

- 

Specifying an explicit `[[Prototype]]` for object literals

```
MyObject.prototype <| {a:1,b:2}
```

- 

Specifying an explicit `[[Prototype]]` for array literals

```
appBehavior <| [0,1,2,3,4,5]
```

- 

"Subclassing" arrays

```
Array.create=function(proto,props) {return Object.defineProperties(proto <| [ ],
props)};
```

- 

Setting the prototype of a function to something other than `Function.prototype`

```
let f = EnhancedFunctionPrototype <| function () {}
```

- 

Setting the prototype of `RegExp` and other built-in objects.

```
var p = newRegExpMethods <| /[a-m][3-7]/
```

- 

Replace the most common uses of the mutable `__proto__` extension

Replace:

```
var o = {
  __proto__: myProto,
  a:0,
  b: function () {}
}
```

With:

```
var o = myProto <| {
  a: 0,
  b: function () {}
}
```

## ES5 Compatability

We may still want to define `Array.create`, `RegExp.create`, and `Function.create` so they can be used in non-Harmony implementations and contexts. In that case they should be defined in terms of the semantics of the `<|` operator.

## Relationship to Other Proposals

This proposal is an alternative to the [array\\_subtypes](#) and [array\\_create](#) proposals.

It is also an alternative to the `[[Prototype]]` functionality of the [Object Initialiser Meta Properties](#) proposal.

## Commentary and rationales

- The right hand side could potentially be extended to be any object value if we were define a generalized object clone operator. In that case, we would be setting the `[[Prototype]]` on a clone of the RHS. This would maintain the immutability of `[[Prototype]]`.
- The contextual keyword `proto` was originally proposed as this operator symbol. Experimentation suggested that a keyword operator was harder to read and might commonly result in awkward code phrasing such as:

```
function (proto) {
  return proto proto {a:1, b:2}
}
```

- The prototype value is placed on the left of the operator because object literals, array literals, and function expressions often span multiple source lines. Placing the prototype value to the right operator would mean that it would occur at the end of such multi-line sequences where it is likely to go unnoticed. The explicitly setting the prototype of a literal object is significant enough that it should be expressed in a manner that is likely to be noticed by code readers. Compare:

```
var obj = someProto <| {
  prop1: expr,
  get prop2 () {
    return computeSomeValue();
  }
}
```

```

    },
    method: function (a,b,c) {
        return this.prop2+a+b+c;
    }
};

```

and

```

var obj = {
    prop1: expr,
    get prop2 () {
        return computeSomeValue();
    },
    method: function (a,b,c) {
        return this.prop2+a+b+c;
    }
} |> someProto;

```

- There are many other possible special character alternates to `<|`. For example, `|>`, `^^`, `*>`, `&>`, `^|`, `<|-`, etc. It isn't clear that any of these is more meaningful or mnemonic than `<|`.
- The `<|` symbol is somewhat suggestive of the [UML Generalization arrow](#) which is the way inheritance is represented in UML class diagrams.
- *BooleanLiterals* are included for completeness. Specifying the prototype of a Boolean wrapper object is unlikely to be of little practical value.
- Earlier proposals proposed used [Object Initialiser Meta Properties](#) within object and array literals to specify a `[[Prototype]]` value. That approach did not extend to other literal forms such as *FunctionExpressions* and *RegularExpressionLiterals*.

strawman/proto\_operator.txt · Last modified: 2011/05/18 01:37 by allen



# [[strawman:object\_initializer\_shorthand]]

Trace: » [match\\_web\\_reality](#) » [array\\_create](#) » [array\\_subtypes](#) » [proto\\_operator](#) » [object\\_initializer\\_shorthand](#)

## Goals

### Table of Contents



- Goals
- Precedent
- Proposal
- Relation to other extensions
- Discussion

- Provide a shorthand for object initialisers whose property keys are initialized by variables of the same name.

◦

Example longhand: `function f(x, y) { return {x: x, y: y}; }`

◦

Example shorthand: `function f(x, y) { return {x, y}; }`

- Reversibility or symmetry with the **destructuring** shorthand object patterns.

## Precedent

- JS1.8.1+ (SpiderMonkey and Rhino) support `{x, y}` in an expression context as shorthand for `{x: x, y: y}`.

## Proposal

- Extend the *PropertyAssignment* production in ES5 as follows (the first right-hand side is new):

```
PropertyAssignment:
  Identifier
```



```

PropertyName : AssignmentExpression
get PropertyName ( ) { FunctionBody }
set PropertyName ( PropertySetParameterList ) { FunctionBody }

```

The production *PropertyAssignment* : *Identifier* is evaluated as follows:

1.

Let *propName* be the String value containing the same sequence of characters as the *Identifier*.

2.

Let *exprValue* be the result of evaluating *Identifier* as a *PrimaryExpression* (per 10.3.1, Identifier Resolution).

3.

Let *propValue* be *GetValue(exprValue)*.

4.

Let *desc* be the Property Descriptor { *[[Value]]*: *propValue*, *[[Writable]]*: **true**, *[[Enumerable]]*: **true**, *[[Configurable]]*: **true** }.

5.

Return Property Identifier (*propName*, *desc*).

## Relation to other extensions

---

The *Identifier* form of *PropertyAssignment* could be integrated with other object literal extensions including **var**, **const**, **method** and **private**. However, it isn't obvious that the use cases for this form significantly overlap with the use cases for the other extensions. It may be better to simply keep the form specified here as is without the complexity of integrating it with the other extensions. — *Allen Wirfs-Brock 2010/09/08 05:26*

## Discussion

---

How about allowing *IdentifierName* ("*."* *IdentifierName*)\* as a shorthand just like C# allows:

```
var object = {b.c.d};
```

as a shorthand for:

```
var object = {d: b.c.d};
```

— Erik Arvidsson 2011/04/28 17:42

First I've heard of this RFE. It's unambiguous but I wonder how much it is worth its (small but non-trivial) weight. If you are creating a near-clone of another object (b.c in the example), then some kind of functional record update might pay off better. A way of saying b.c with {e: 42} and possibly without {a}.

— Brendan Eich 2011/05/01 21:25

I think we need to change from *IdentifierName* to *Identifier* since even though we could allow keywords here it makes little sense. What would the following do?

```
var object = {var}
```

Also, would it be worth allowing strings and numbers?

```
var x = 42
var object = {0, 'a', x}
```

as shorthand for

```
var x = 42
var object = {0: 0, 'a': 'a', x: x}
```

I can see strings being useful in code that uses objects as enums but it is mostly for symmetry.

— Erik Arvidsson 2011/05/06 02:53

Good point about *IdentifierName* – I changed to *Identifier*. SpiderMonkey (and probably Rhino, I haven't tested) will need fixing.

I do not understand how string and number property names would work. Even in global code, var does not bind properties in Harmony (no global object as scope). I say: YAGNI and too complicated, when in doubt leave it out.

— Brendan Eich 2011/05/11 21:22

[[strawman:  
concise\_object\_literal\_extensions]]Trace: » array\_create » array\_subtypes »  
proto\_operator »

object\_initialiser\_shorthand » concise\_object\_literal\_extensions

## Concise Object Literal Extensions

This is a set of syntactically concise extensions that cover the most common object property creation use cases that are not covered by the current *ObjectLiteral* syntax:

### Motivation

All properties of objects created using object literal syntax currently have the attributes enumerable: true, configurable: true, writable: true. This limits the utility of object literals as ECMAScript's primary declarative form of object creation. It is particularly problematic in the case of method properties as it is seldom desirable for such properties to be enumerable.

ES5 permits constructions of objects with arbitrary property attribute setting using the `Object.create` function. However, this form is much more verbose and its usage is complicated by the fact that the default attribute values are different than what is used for object literals. This can be see as follow:

```
var obj = {a:x, k:0.5, m: function(z) {return z+this.a+this.k}};
```

If the programmer desires for the property `k` to be non-writable, non-configurable, and non-enumerable and for the method property `m` to be non-writable and non-enumerable, they would express like this using `Object.create`:

```
var obj = Object.create(Object.prototype, {
  a: {value: x, writable: true, enumerable: true, configurable: true},
  k: {value: 0.5}, //use default false values for all attributes
  m: {value: function(z) {return z+this.a+this.k}, writable: false, enumerable:
false, configurable: true}
});
```

or perhaps in a slightly less verbose (but arguably more obscure) form using `Object.defineProperties`:

```
var obj = Object.defineProperties(
  {a:x, k:0.5, m: function(z) {return z+this.a+this.k}},
  {k: {writable: false, enumerable: false, configurable: false},
  m: {writable: false, enumerable: false}
});
```

Using the extensions in this proposal the above could be directly expressed using an object literal as:

```
var obj = {a:x, ~!k:0.5, m(z) {return z+this.a+this.k}};
```

### Summary

This proposal extends object literals in these four ways:

#### 1.

If a property definition (a *PropertyAssignment* in the ES grammar) is prefixed with `~` the property is non-enumerable

#### Table of Contents

- Concise Object Literal Extensions
  - Motivation
  - Summary
  - Non-Enumerable and Non-Configurable Properties
    - Examples
  - Non-Writable Data Properties
    - Examples
  - methods
    - Examples

2.

If a property definition is prefixed with **!** the property is non-configurable

3.

If a data property definition uses **=** in place of **:** the property is non-writable

4.

If a property definition has the form of a *FunctionDeclaration* without the keyword `function` it is a non-enumerable, non-writable data property definition whose name is the function name

## Non-Enumerable and Non-Configurable Properties

---

Define a property to be non-enumerable or non-configurable:

### **PropertyNameAndValueList :**

*PrefixedPropertyAssignment*

*PropertyNameAndValueList* , *PrefixedPropertyAssignment*

### **PrefixedPropertyAssignment:**

*PropertyPrefix*<sub>opt</sub> *PropertyAssignment*

### **PropertyPrefix :**

!  
~  
~!  
!~

Prefixing a property with **!** makes it non-configurable. Prefixing a property with **~** makes it non-enumerable. Prefixing a property with either **~!** or **!~** makes it both non-configurable and non-enumerable.

## Examples

```
var = {
  //non-configurable properties
  !x : 1,
  !"non identifier name" : 2,
  !get 3() {return 3},
  !set 3(v) {}, //both get and set must have the same confiburablility
  //non-enumerable properties
  ~y: 2,
  ~0: 0,
  ~get "a b"() {return 3},
  ~set "a b"(v) {}, //both get and set must have the same enumerability
  //non-configurable, non-enumerable properties
  !~z : 3,
  ~!a: 4,
  ~!get c() {return 4}
}
```

## Non-Writable Data Properties

---

Define a property to be non-writable:

### **PropertyAssignment :**

*PropertyName* = *AssignmentStatement*

## Examples

```

var obj = {
  //constant properties
  a = 1,
  "non identifier name" = 2,
  3 = 3,
  //constant non-enumerable property
  ~b = 4
};

//or more compactly:
var obj={a=1,"non identifier name "=2,3=3,~b=4};

```

## methods

As a convenience for the most common use case, the most concise forms of methods definitions define a non enumerable, non-writable property:

*PropertyAssignment* :  
*PropertyName* ( *FormalParameterList*<sub>opt</sub> ) { *FunctionBody* }  
*namedHashFunction* **pending specification**

## Examples

```

var = {
  toString () {return "literal"},//enumerable: false, writable: false configurable: true
  !"non identifier name" () {}, /enumerable: false, writable: false configurable:
false
  ~length () {return 0}, // ~ doesn't change enumerability for methods
  #valueOf{0}, //enumerable: false, writable: false configurable: true

  // the following are just regular data property definitions using the ~ and =
  // extensions defined above
  f: function() {}, //enumerable: true, writable: true configurable: true
  ~g: function() {}, //enumerable: false, writable: true configurable: true
  h = function() {} //enumerable: true, writable: false configurable: true
}

```

strawman/concise\_object\_literal\_extensions.txt · Last modified: 2011/04/28 16:54 by allen



[[strawman:  
support\_full\_unicode\_in\_strings]]

Trace: » [array\\_subtypes](#) » [proto\\_operator](#) » [object\\_initialiser\\_shorthand](#) » [concise\\_object\\_literal\\_extensions](#) » [support\\_full\\_unicode\\_in\\_strings](#)

## Support Full Unicode

Allen Wirfs-Brock

ECMAScript currently only directly supports the 16-bit basic multilingual plane (BMP) subset of Unicode which is all that existed when ECMAScript was first designed. Since then Unicode has been extended to require up to 21-bits per code. As currently defined, characters in this expanded character set cannot be used in the source code of ECMAScript programs and cannot be directly included in runtime ECMAScript string values.

There are very few places where the ECMAScript specification has actual dependencies upon the size of individual characters so the compatibility impact of supporting full Unicode is quite small. There is a larger impact on actual implementations but even there the impact is probably smaller than someone might initially expect.

See [JavaScript Internationalization](#) for the W3C Internationalization WG's take on some issues regarding Unicode support in ECMAScript.

### Table of Contents

- Support Full Unicode
  - Support Full Unicode in ECMAScript Source Code
    - [Input Encoding](#)
    - [Unicode Escape Sequences](#)
    - [Likely Implementation Impacts](#)
  - Support Full Unicode in the ECMAScript Runtime Strings
    - [The String Type](#)
    - [String Operators and Functions](#)
      - [15.5.3.2 String.fromCharCode](#)
      - [15.5.4.5 String.prototype.charCodeAt](#)
      - [15.1.3 URI Handling Function Properties](#)
    - [Comparability Impacts](#)
    - [Possible Implementation Impacts](#)
  - [Discussion](#)

## Support Full Unicode in ECMAScript Source Code

### Input Encoding

Clause 6 of the ECMAScript 5.1 specification states "ECMAScript source text is presented as a sequence of characters in the Unicode character encoding" and it goes on to say that the text "is expected to have been normalised to Unicode Normalization Form C". However, it also states that "source text is assumed to be a sequence of 16-bit code units for the purpose of this specification".

As part of this proposal, the statement about assuming 16-bit code units will be deleted. In addition throughout the specification all occurrence of "code unit" implying 16-bit characters will be replaced with "code point" which means the canonical encoding of any possible Unicode character. In particular, the definition of *SourceCharacter* is changed to:

*SourceCharacter* ::  
any Unicode codepoint

More generally, it is beyond the scope of a language specification to require any specific external encoding of source programs of that language. Dealing with various possible external encodings is more a matter for communication protocols, host platforms, and language implementations. What is appropriate (and necessary) is for the language specification to define a specific input alphabet for its lexical grammar. Clauses 6 and 7 will be updated to clarify that the alphabet of the lexical grammar is full Unicode Normalization Form C. Any implications concerning external source code encoding or implications that implementations must use some specific internal encoding of source program text will be removed.

## Unicode Escape Sequences

ECMAScript 5.1 currently includes the lexical production *UnicodeEscapeSequence* defined as follows:

```
UnicodeEscapeSequence ::  
u HexDigit HexDigit HexDigit HexDigit
```

This production is limited to only expressing 16-bit codepoint values. In order to fully support Unicode it is necessary to have an escape sequence that can express 21-bit codepoint values. This cannot be accomplished simply by allowing *UnicodeEscapeSequence* to be extended by two optional additional *HexDigits* because ES5.1 has contexts where a *UnicodeEscapeSequence* is allowed to immediately about a *HexDigit* that is not part of the *UnicodeEscapeSequence*. For example, the following is a valid three character identifier in ES5.1: `\u004101`. This is actually equivalent to the identifier `A01`. to address this issue, a new form of *UnicodeEscapeSequence* is added that is explicitly tagged as containing six hex digits. The new definition is:

```
UnicodeEscapeSequence ::  
u HexDigit HexDigit HexDigit HexDigit  
u+ HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit
```

The 6 digit extended *UnicodeEscapeSequence* is a syntactic extension that is only recognized after explicit versioning opt-in to the extended "Harmony" syntax.

In clause 7.8.4 an additional CV definition is provided for *UnicodeEscapeSequence* :: **u+** *HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit*.

## Likely Implementation Impacts

Whether or not requiring full Unicode Source code will have a significant impact upon existing implementations is likely to be highly dependent upon the approach that an implementation currently uses internally for representing source code. If an implementation currently uses an UTF-8 or UTF-16 internal encoding then this change will likely have minimal implementation impact as these encodings can already deal with the full Unicode character set. There is likely to be a larger impact on implementations that currently use a direct 16-bit UCS-2 encoding of source programs. However if the intent to switch to full Unicode of ECMAScript is communicated in the near future there should be plenty of time of such implementations to adapted prior to completion of the next edition of the ES standard.

## Support Full Unicode in the ECMAScript Runtime Strings

---

## The String Type

The definition of the String Type in 8.4 will be modified as follows (new text is underlined):

---

The String type is the set of all finite ordered sequences of zero or more

~~16-bit~~ 21-bit unsigned integer values ("elements"). The String type is generally used to represent textual data in a running ECMAScript program, in which case each element in the String is treated as a ~~code-unit~~ codepoint value (see Clause 6). Each element is regarded as occupying a position within the sequence. These positions are indexed with nonnegative integers. The first element (if any) is at position 0, the next element (if any) at position 1, and so on. The length of a String is the number of elements (i.e., ~~16-bit~~ 21-bit values) within it. The empty String has length zero and therefore contains no elements.

When a String contains actual textual data, each element is considered to be a single

~~UTF-16 Unicode code-unit~~ codepoint. Whether or not this is the actual storage format of a String, the characters within a String are numbered by their initial ~~code-unit~~ codepoint element position as though they were represented using ~~UTF-16~~ UTF-32. All operations on Strings (except as otherwise stated) treat them as sequences of undifferentiated ~~16-bit~~ 21-bit unsigned integers; they do not ensure the resulting String is in normalised form, nor do they ensure language-sensitive results.

---

Note that the current usage of UTF-16 in the above ES5.1 clause is an editorial error and dates back to at least ES3. It probably was intended to mean the same as UCS-2. ES3-5.1 did not intend to imply that the ECMAScript strings perform any sort of automatic UTF-16 encoding or interpretation of codepoints that are outside of the BMP.

## String Operators and Functions

String operators (concatenation and comparison) and methods associated with the String object are generally specified as operating upon logical sequences of characters in a manner that is independent of the the number of bits in the character encoding. Generally no changes are needed to the specification of these operations and methods other than the general replacement of the term "code unit" with "codepoint". There are only a few places where explicit changes have to be made to accommodate full Unicode characters.

### 15.5.3.2 String.fromCharCode

This function constructs a string value from a sequence of "character codes" that are passed as arguments to the function. A ToUint16 conversion is applied to each argument before constructing the string value.

To support full Unicode we need to support constructing such strings where some of the character codes may have up to 21-bits. Changing `String.fromCharCode` to perform a ToUint21 conversion instead of a ToUnit16 conversion could cause incompatibility with existing code that is using strings to store 16-bit binary integer values. Such code could be calling `String.fromCharCode` passing arbitrary numeric values with the expectation that the values will be truncated to 16-bit.

Instead of changing the specification of `String.fromCharCode`, a new function is



added. `String.fromCharCode` will be specified identically to `String.fromCharCode` with the exception that a `ToUint21` conversion will be performed on each argument value in place of `ToUint16`.

#### 15.5.4.5 `String.prototype.charCodeAtAt`

This method is currently specified to return the “code unit” at a specific character position. Currently, this is a `Uint16` value. When “code unit” is pervasively replaced with “codepoint” the returned result will be a `Uint21` value. It is unclear whether this might cause comparability issues as existing code is unlikely to encounter codepoint values greater than 65535. However, the fact that a character is being accessed as an encoded numeric value implies that some sort of numeric computation is likely to be performed of the character code. Such computations might be sensitive to the magnitude of the value. For that reason it is proposed that `charCodeAtAt` will be respecified to explicitly return a `Uint16` value. In addition, a new method `String.prototype.codepointAt` will be added. `codepointAt` will function identically to `charCodeAtAt` with the exception that `codepointAt` returns a `Uint21` result.

### 15.1.3 URI Handling Function Properties

The functions defined in this section are the only ECMAScript functions that explicitly deal with UTF-8 and UTF-16 encodings including UTF-16 surrogate pairs. For compatibility reasons these functions should generally not be changed. The `encode` algorithm will continue to recognize UTF-16 surrogate pairs and translate them to the corresponding UTF-8 encoding. Similarly, the `decode` algorithm will translate UTF-8 encodings of large codepoint values into UTF-16 surrogate pairs.

However, the `encode` algorithm does not currently deal with codepoint values greater than `0xffff`. The algorithm will be modified so that if such characters are encountered they will be properly translated into a UTF-8 encoding.

Alternative versions of `decodeURI` and `decodeURIComponent` will be provided that translated UTF-8 encoded codepoints greater than `0xffff` into single characters rather than a surrogate pair. These alternative functions will be named `decodeURI21` and `decodeURIComponent21`.

## Comparability Impacts

Other than for issues already discussed above concerning explicit manipulation of character codes and UTF-8/UTF-16 these changes should have no impact upon existing code that does string processing. Such code should simply continue to work, even when encountering Unicode characters with codepoints greater than `0xffff`. Programs that perform their own internal UTF-16 or UTF-8 processing should continue to operate without modification as all 16-bit character code values continue to be allowable character values within the expanded character size.

## Possible Implementation Impacts

Supporting full Unicode characters does not necessarily require either the doubling of character sizes or the use of algorithmically expensive encoding such as UTF-8 for all strings.

It is important to consider that all JavaScript strings are immutable after their initial creation. That means that the maximum character size is known at the time the string is created. Also, consider that many ECMAScript implementation already use multiple internal representations of strings for purposes such as concatenation optimization. Strings containing large characters could simply be implemented as one or more additional alternative internal string representation. As very few strings actually contain large character values their should be minimal storage or computational overhead for typical programs.

There will, however, be implementation work required to ensure that all string functions correctly deal with any string representations contain large characters.

## Discussion

---

strawman/support\_full\_unicode\_in\_strings.txt · Last modified: 2011/05/17 20:34 by allen



# [[strawman: array\_comprehensions]]

Trace: » proto\_operator »  
object\_initialiser\_shorthand »

concise\_object\_literal\_extensions » support\_full\_unicode\_in\_strings » 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:

```
[ Expression0 for ( LHSExpression1 in Expression1 ) ... for ( LHSExpressionn ) if ( Expression )opt ]
```

can be defined by expansion to the expression:

```
let (result = []) {  
  for (let LHSExpression1 in Expression1) {  
    ...  
    for (let LHSExpressionn in Expressionn) {  
      if ( Expression )opt  
        ArrayPush(result, Expression0);  
    }  
  }  
}
```

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



# [[strawman:simple\_maps\_and\_sets]]

Trace: » [object\\_initialiser\\_shorthand](#) » [concise\\_object\\_literal\\_extensions](#) » [support\\_full\\_unicode\\_in\\_strings](#) » [array\\_comprehensions](#) » [simple\\_maps\\_and\\_sets](#)

## Simple Maps and Sets

Similar in style to [weak maps](#) but without the funny garbage collection semantics or non-enumerability. Depends on the [iterators](#) and [egal](#) proposals. Depends on the [classes\\_with\\_trait\\_composition](#) only for expository purposes.

### Map

Given

```
/** A non-stupid alternative to Array.prototype.indexOf */
function indexOfIdentical(keys, key) {
  for (var i = 0; i < keys.length; i++) {
    if (Object.is(keys[i], key)) { return i; }
  }
  return -1;
}
```

Executable spec

```
class Map {
  private keys, vals;
  constructor() {
    private(this).keys = [];
    private(this).vals = [];
  }
  get(key) {
    const keys = private(this).keys;
    const i = indexOfIdentical(keys, key);
    return i < 0 ? undefined : private(this).vals[i];
  }
  has(key) {
    const keys = private(this).keys;
    return indexOfIdentical(keys, key) >= 0;
  }
}
```

```

set(key, val) {
  const keys = private(this).keys;
  const vals = private(this).vals;
  let i = indexOfIdentical(keys, key);
  if (i < 0) { i = keys.length; }
  keys[i] = key;
  vals[i] = val;
}
delete(key) {
  const keys = private(this).keys;
  const vals = private(this).vals;
  const i = indexOfIdentical(keys, key);
  if (i < 0) { return false; }
  keys.splice(i, 1);
  vals.splice(i, 1);
  return true;
}
// todo: iteration
}

```

## Set

### Executable Spec

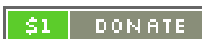
```

class Set {
  private map;
  constructor() {
    private(this).map = Map();
  }
  has(key) { return private(this).map.has(key); }
  add(key) { private(this).map.set(key, true); }
  delete(key) { return private(this).delete(key); }
  // todo: iteration
}

```

strawman/simple\_maps\_and\_sets.txt · Last modified: 2011/05/17 06:57 by markm

[RSS](#) [XML FEED](#)



# [[strawman:string\_extras]]

Trace: » [concise\\_object\\_literal\\_extensions](#) » [support\\_full\\_unicode\\_in\\_strings](#) » [array\\_comprehensions](#) » [simple\\_maps\\_and\\_sets](#) » [string\\_extras](#)

## String extras

---

A few basic conveniences that many languages (e.g., Java, C#, Python, Ruby) have by default are missing in the ECMAScript string library.

- String.prototype.startsWith

Behaves the same as:

```
String.prototype.startsWith = function(s) {  
    return this.indexOf(s) === 0;  
};
```

- String.prototype.endsWith

Behaves the same as:

```
String.prototype.endsWith = function(s) {  
    var t = String(s);  
    return this.lastIndexOf(t) === (this.length - t.length);  
};
```

- String.prototype.contains

Behaves the same as:

```
String.prototype.contains = function(s) {  
    return this.indexOf(s) !== -1;  
};
```

•

`String.prototype.toArray`

Behaves the same as:

```
String.prototype.toArray = function() {  
    return this.split('');  
};
```

— *Dave Herman* 2011/03/13 18:54

strawman/string\_extras.txt · Last modified: 2011/03/13 20:59 by arv

[RSS](#) [XML FEED](#)

 LICENSED

 DONATE

 POWERED

 XHTML 1.0

 CSS

 DOKUWIKI



# [[strawman:pragmas]]

Trace: » support\_full\_unicode\_in\_strings » array\_comprehensions » simple\_maps\_and\_sets » string\_extras » pragmas

## Pragmas

ECMAScript Edition 5 introduced a backwards-compatible syntax of the *directive prologue* with string-literal-expression-statement directives for pragmas. With Harmony, we are allowing the introduction of new syntax via [versioning](#), so we should move towards a future-proof pragma syntax.

## Use

This spec proposes the introduction of a keyword `use` in place of the previous string-literal directives. In Harmony, the string literal form would no longer be recognized as special.

A hypothetical example is the [lexical scope](#) pragma:

```
use lexical scope;
```

While this spec does not propose any *specific* pragmas, its primary purpose is to create a syntactic space for future pragmas, i.e., for future-proofing.

## Syntax

The syntax of pragmas is designed to be very permissive, so that implementations may introduce custom pragmas.

```
UsePragma ::= "use" PragmaItem ( "," PragmaItem)* ";"
PragmaItem ::= PragmaWord+
PragmaWord ::= Keyword | Identifier
```

*TODO:*

- semicolon insertion issues (no newline between PragmaItem keywords?)
-

more future-proofing for PragmaWord? allow literals? probably don't want to allow operators, right?

- best practices for non-standard prefixing? (e.g., "use moz widgets")

## Semantics

---

- In Harmony, string literal statements are no longer recognized as prologue directives.
- A pragma that is unrecognized must be ignored. This way implementations remain future-proof.

strawman/pragmas.txt · Last modified: 2010/09/21 21:57 by dherman

[RSS](#) [XML FEED](#)



Trace: » [array\\_comprehensions](#) » [simple\\_maps\\_and\\_sets](#) » [string\\_extras](#) » [pragmas](#) » [multiple\\_globals](#)

## 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](#)

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/multiple\_globals.txt · Last modified: 2011/03/04 19:43 by dherman



[[strawman:  
enumeration]]

Trace: » [simple\\_maps\\_and\\_sets](#) » [string\\_extras](#) » [pragmas](#) » [multiple\\_globals](#) » [enumeration](#)

## Overview

The semantics of enumeration in existing editions of ECMA-262 is very loosely specified. This strawman proposes more fully specifying the semantics of property enumeration.

## Versioning

The semantics below is incompatible with existing web behavior, and would be enabled only through opting into Harmony. For compatibility, code run in legacy JavaScript versions may need to preserve the previous enumeration behavior.

## For-in loops

The semantics of [iteration](#) uses the `iterate` proxy trap to drive iteration if it exists, but falls back to using the enumeration behavior of this proposal.

## Semantics

The following operation produces an eagerly-computed sequence of the own-properties of an object.

**Operation** *EnumerateProperties*(obj)

Execution	Error propagation
Let suppress = $\emptyset$	
Let props = []	
While (obj <b>!= empty</b> )	
If <i>IsTrappingProxy</i> (obj)	
Let handler = obj. <b>[[Handler]]</b>	
Let enum = handler. <b>[[Get]]</b> ("enumerate")	If <i>IsError</i> (enum) Return enum
If <b>!IsCallable</b> (enum)	
Return (type= <b>error</b> , value=TypeError, target= <b>empty</b> )	
Let rest = enum.value. <b>[[Call]]</b> (handler, [])	If <i>IsError</i> (rest) Return rest
Return [ props, ..., rest, ... ]	

Let $own = OwnProperties(obj)$
For each $i$ in $0 \dots own.length - 1$
Let $P = own[i]$
If $P.attributes.enumerable \ \&\& \ P \notin props \ \&\& \ P \notin suppress$
$props := [ props, \dots, P.name ]$
If $\neg P.attributes.enumerable$
$suppress := suppress \cup \{ P \}$
$obj := obj.[[Prototype]]$
Return $props$

**Operation** *OwnProperties*(obj)

We could specify this in a number of different ways. Conceptually, this operation should produce a sequence of property descriptors, in the following order:

1.  
index properties (see definition below) in ascending numeric order
2.  
all other properties, in the order in which they were created

An *index*, as defined in 15.4, is a property name  $P$  such that  $ToString(ToUint32(P))$  is equal to  $P$  and  $ToUint32(P)$  is not equal to  $2^{*}32-1$ .

Several specification approaches:

- all objects have two sequential property tables:
  - properties with uint32 names, kept in integer order
  - all other properties, kept in creation order
- objects have one sequential property table, kept in creation order; *OwnProperties* then filters out uint32 properties
- properties in property tables include an internal creation-order attribute

See the thread starting at [this message from Charles Kendrick](#) for criticisms (some aesthetic or theoretical) and compatibility concerns (these look significant to me) about enumerating own indexed properties first for non-Array objects.

— *Brendan Eich* 2011/03/13 20:04





[[strawman:  
arrow\_function\_syntax]]

Trace: » string\_extras » pragmas »  
multiple\_globals » enumeration »

arrow\_function\_syntax

## Proposal

```
// Empty arrow function is minimal-length
let empty = ->;

// Expression bodies needs no parentheses or braces
let identity = (x) -> x;

// Fix: object initialiser need not be parenthesized, see Grammar Changes
let key_maker = (val) -> {key: val};

// Nullary arrow function starts with arrow (cannot begin statement)
let nullary = -> preamble + ': ' + body;

// No need for parens even for lower-precedence expression body
let square = (x) -> x * x;

// Statement body needs braces (completion return TODO)
let oddArray = [];
array.forEach((v, i) -> { if (i & 1) oddArray[i >>> 1] = v; });

// Use # to freeze and join to nearest relevant closure
function return_pure() {
  return #(a) -> a * a;
}

let p = return_pure(),
    q = return_pure();
assert(p === q);

function check_frozen(o) {
  try {
    o.x = "expando";
    assert(0 == "not reached");
  } catch (e) {
    // e is something like "TypeError: o is not extensible"
    assert(e.name == "TypeError");
  }
}

check_frozen(p);

function partial_mul(a) {
  return #(b) -> a * b;
}

let x = partial_mul(3),
    y = partial_mul(4),
    z = partial_mul(3);

assert(x !== y);
```

```

assert(x !== z);
assert(y !== z);

check_frozen(x);
check_frozen(y);
check_frozen(z);

// Use '=>' (fat arrow) for lexical 'this', as in CoffeeScript
// ("fat" is apt because this form costs more than '-->')
const obj = {
  method: function () {
    return => this;
  }
};
assert(obj.method()() === obj);

// And *only* lexical 'this' for => functions
let fake = {steal: obj.method()};
assert(fake.steal() === obj);

// But 'function' still has dynamic 'this'
let real = {borrow: obj.method};
assert(real.borrow()() === real);

// Recap:
// use '-->' instead of 'function' for lighter syntax
// use '=>' instead of calling bind or writing a closure
const obj2 = {
  method: () -> (=> this)
};
assert(obj2.method()() === obj2);

let fake2 = {steal: obj2.method()};
assert(fake2.steal() === obj2);

let real2 = {borrow: obj2.method};
assert(real2.method()() === real2);

// An explicit 'this' parameter can have an initializer
// Semantics are as in the "parameter default values" Harmony proposal
const self = {c: 0};
const self_bound = (this = self, a, b) -> {
  this.c = a * b;
};
self_bound(2, 3);
assert(self.c === 6);

const other = {c: "not set"};
self_bound.call(other, 4, 5);
assert(other.c === "not set");
assert(self.c === 20);

// A special form based on the default operator proposal
const self_default_bound = (this ??= self, a, b) -> {
  this.c = a * b;
}
self_default_bound(6, 7);
assert(self.c === 42);

self_default_bound.call(other, 8, 9);
assert(other.c === 72);

```

```

assert(self.c === 42);

// '=>' is short for '->' with an explicit 'this' parameter
function outer() {
  const bound    = () => this;
  const bound2   = (this = this) -> this; // initializer has outer 'this' in scope
  const unbound  = () -> this;
  const unbound2 = (this) -> this;

  return [bound, bound2, unbound, unbound2];
}

const t = {},
      u = {};

const v = outer.call(t);

assert(v[0]() === t);
assert(v[1]() === t);
assert(v[2]() === t);
assert(v[3]() === t);

assert(v[0].call(u) === t);
assert(v[1].call(u) === t);
assert(v[2].call(u) === u);
assert(v[3].call(u) === u);

// Object initialiser shorthand: "method" = function-valued property with dynamic 'this'
const obj = {
  method() -> {
    return => this;
  }
};

// Name binding forms hoist to body (var) or block (let, const) top
var warmer(a) -> {...};
let warm(b) -> {...};
const colder(c) -> {...};
const #coldest(d) -> {...};

```

## Grammar Changes

Change all uses of *AssignmentExpression* outside of the *Expression* sub-grammar to *InitialValue*:

```

ElementList :          // See 11.1.4
  Elisionopt InitialValue
  ElementList , Elisionopt InitialValue
  ...
PropertyAssignment : // See 11.1.5
 PropertyName : InitialValue
  ...
ArgumentList :       // See 11.2
  InitialValue
  ArgumentList , InitialValue
  ...
Initialiser :        // See 12.2
  = InitialValue

```

```
InitialiserNoIn : // See 12.2
  = InitialValueNoIn
```

Define *InitialValue* and *ArrowFunctionExpression*:

```
InitialValue :
  AssignmentExpression
  ArrowFunctionExpression

ArrowFunctionExpression :
  ArrowFormalParametersOpt Arrow InitialValue
  ArrowFormalParametersOpt Arrow BlockOpt

ArrowFormalParameters :
  ( FormalParameterListOpt )
  ( this InitialiserOpt )
  ( this InitialiserOpt , FormalParameterList )

Arrow : one of -> or =>
```

To enable unparenthesized *ObjectLiteral* expressions as bodies of arrow functions, without ambiguity with *Block* bodies, restrict *LabelledStatement* as follows:

```
LabelledStatement :
  Identifier : LabelledStatement
  Identifier : LabelUsingStatement

LabelUsingStatement :
  NonEmptyBlock
  IfStatement
  IterationStatement
  BreakStatement
  SwitchStatement
  TryStatement

NonEmptyBlock : // See 12.1
  { StatementList }
```

The resulting grammar should be unambiguously LR(1) (ignoring ASI), because { L: *expr...* } can parse only as an *ObjectLiteral*, and { L: { if (*cond*)... } } or similar *LabelUsingStatement* variations can parse only as a *LabelledStatement*.

The grammar is not LR(1) because of the conflict between *ArrowFormalParameters* and *AssignmentExpression*. Trying to resolve this conflict is not easy and would produce much ugliness. — *Waldemar Horwat 2011/05/18 00:35*

TODO: evaluate top-down parsing difficulty

TODO: This conflicts with the [object initialiser shorthand](#) strawman.

To allow arrow functions to appear unparenthesized as the right-hand side of assignment statements, split assignment out of *ExpressionStatement* into *AssignmentStatement*:

```
Statement :
  Block
  VariableStatement
  EmptyStatement
  AssignmentStatement
```

```

    ExpressionStatement
    ...

AssignmentStatement :
    [lookahead ∉ {{, function}}] AssignmentList ;

AssignmentList :
    Assignment
    AssignmentList , Assignment

Assignment :
    LeftHandSideExpression = InitialValue
    LeftHandSideExpression AssignmentOperator InitialValue

ExpressionStatement :
    [lookahead ∉ {{, function}}] ConditionalExpression ;

```

Finally, *PrimaryExpression* produces a parenthesized *ArrowFunctionExpression*:

```

PrimaryExpression :
    ...
    ( ArrowFunctionExpression )

```

These changes are intended to be backward-compatible: existing JS parses as before, with the same semantics. New opt-in Harmony JS may use arrow functions where allowed.

## Rationale

---

TODO

Notes

- 

Hard to beat C# and CoffeeScript here (but no unparenthesized single-parameter form as in C#)

- 

TC39 should embrace, clean-up, and extend rather than re-invent or compete with de-facto and nearby de-jure standards

- 

It's hard to say what is a precedent, but CoffeeScript is "just syntax", no elaborate compilation – JS runtime semantics

- 

Main worry about `->` was top-down parsing burden but olliej and I agree it's tolerable (comparable to destructuring and top-down `LeftHandExpression` parsing)

- 

`->` cannot start an *ExpressionStatement* in the grammar

- 

Analogous to `function` being excluded along with `{` in the lookahead set (see ES5.1 12.4)

- 

`->` parses as if it were a low-precedence operator joining a restricted comma expression (implicitly quoted) to a body

- 

`()` for nullary case is optional, to reduce boilerplate punctuation, after CoffeeScript and similar to [shorter function syntax](#)

- - # opt-in addresses Alex's good point that mutability should not go out window along with verbosity of function
  - - Hash still consistently implies frozen value-type-ness, as in [records](#) and [tuples](#)
- - Probably should allow `(this ?? self)` as a shorthand for `(this ??= self)...`
- - Dependencies (some are optional pieces)
    - - [parameter default values](#)
    - - [completion\\_reform](#)
    - - [const functions](#) for the joining algorithm
    - - [default\\_operator](#)
    - - [soft\\_bind](#) - the `(this ?? self)` syntax addresses this case, IINM

— *Brendan Eich* 2011/05/02 23:49



# [[strawman: multiline\_regexp]]

Trace: » [string\\_extras](#) » [pragmas](#) » [multiple\\_globals](#) » [enumeration](#) » [arrow\\_function\\_syntax](#)

## 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](#)



LICENSED



DONATE



POWERED



XHTML 1.0



CSS



DOKUWIKI

[[strawman:  
name\_property\_of\_functions]]

Trace: » [pragmas](#) »  
[multiple\\_globals](#) »

[enumeration](#) » [arrow\\_function\\_syntax](#) » [name\\_property\\_of\\_functions](#)

## 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](#)
- Rather than adding a WebKit-inspired `displayName` writable property, specify `Function.displayName` (`f`) as follows:
  - If `f.name` exists, return `f.name`.
  - Else if `f` was invoked via an expression evaluated from a Reference whose `propertyName` was `N`, return `N`.
  - Else if `f` was assigned to a Reference whose `propertyName` was `M`, return `M`. There could be more than one such `M`. Implementations should pick the most recently used name.
  - Else if `f` was the initial value in an object initializer or a property descriptor for a property named `P`, return `P`.
  - Else returned `undefined`.

## Discussion

---

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

Despite being one of the people responsible for `displayName` existing i kind of wish that it didn’t. I feel that a lot of what it provides needn’t being exposed to content in general. In hindsight i feel the better solution would have been to have the platform development tools provide APIs to associate names with some functions and so not have it be part of ES core.

If we were wanting to standardise some kind of developer tools API (essentially the frequently reverse engineered ‘console’ APIs) I think `displayName` (or similar) would make sense there.

— *Oliver Hunt* 2011/05/01 23:33



# [[strawman:paren\_free]]

Trace: » [multiple\\_globals](#) » [enumeration](#) » [arrow\\_function\\_syntax](#) » [name\\_property\\_of\\_functions](#) » [paren\\_free](#)

## Motivation

### Table of Contents



- Motivation
- History
- Proposal
- Compatibility
- TODO

Syntax matters, keystrokes count. Both readability and writability can be impaired by too much punctuation and unnecessary bracketing. Some languages even prefer indentation-based block structure to bracing, and their fans report read and write (including keystroke and RSI avoidance) wins.

JS has a number of statement forms with mandatory parentheses around the head. Can we relax syntax without introducing ambiguity or bad human read/write factors? This proposal makes an attempt, first described [here](#) and prototyped in [Narcissus](#) via the `-paren-free` option.

A specific motivation for this proposal is the irredeemable `for-in` loop, whose semantics have been underspecified forever, with ongoing divergence among implementations, and where [array comprehensions](#) and [generator expressions](#) do not want parenthesized `for-in` heads, yet where users *do* want better semantics for all `for-in` variants.

## History

JS derives from Java from C++ from C (via early C and B), from BCPL. BCPL had paren-free `if`, etc., heads disambiguated via `do` reserved words to separate an expression consequent, avoiding ambiguity.

JS style guides often favor mandatory bracing of `if` consequents and other sub-statement bodies, which also suffice to avoid ambiguity about where the condition or head expression ends and the dependent sub-statement starts.

## Proposal

Consider ES5 12.5, "The `if` Statement", modified as follows:

```
IfStatement :
    if Expression SubStatement else SubStatement
    if Expression SubStatement
```

Where *SubStatement* is

```
SubStatement :
  Block
  KeywordStatement
```

and *KeywordStatement* is

```
KeywordStatement :
  VariableStatement
  IfStatement
  IterationStatement
  ContinueStatement
  BreakStatement
  ReturnStatement
  SwitchStatement
  ThrowStatement
  TryStatement
  DebuggerStatement
```

And so on for *IterationStatement*, *SwitchStatement*, and catch clauses in *TryStatement* – except catch blocks must still be braced (as with `try` and `finally` since their introduction in ES3).

We allow single sub-statements starting with unconditionally reserved keywords to be unbraced after a paren-free head, since the keyword acts as BCPL's `do` separator to disambiguate head from body expression.

Notice how this relaxation from requiring braces around the body allows `if-else-if` chains without a special case:

```
if x < y {
} else if x < z {
} else if x < w {
} else {
}
```

instead of the perfidious rightward drift of:

```
if x < y {
} else {
  if x < z {
  } else {
    if x < w {
    } else {
```

```

|
| }
| }
| }
|

```

This keyword-or-brace refinement also matches some popular style guides that recommend braced bodies except where the body is a short keyword-prefixed statement starting with `break`, `continue`, `throw`, or `return`.

## Compatibility

---

Apart from `for` and `catch` heads, the relaxation proposed here that removes mandatory parentheses still allows overparenthesization of the head *Expression*. For such non-`for`, non-`catch` heads, the new syntax has no new semantics and old source parses the same (extra parentheses do not alter the abstract syntax tree).

Thus we preserve backward compatibility of `if`, `while`, `do-while`, and `switch` statements, where the head syntax is *Expression*. Old content migrates painlessly, and developers may remove parentheses at their leisure, once pre-Harmony user agents dwindle to unsupportably small hit rates on servers.

Parenthesizing a `catch` head would be an error. This proposal values consistency over backward compatibility, so `catch` heads would have to lose their parentheses to migrate into Harmony. This is a cost, but `catch` heads may change semantics in Harmony per [catch\\_guards](#). If semantics change, then the syntax change creates a migration early-error “speed bump” – a benefit compared to allowing existing code to run with different results.

`for` loops change incompatibly in order to reform the syntax and semantics of `for-in`, which is otherwise not compatibly reformable under parenthesized syntax. This is an intentional step to clean the slate, avoid migration mistakes detectable only at runtime, and still preserve the wanted `for` and `in` keywords in the obvious form. More on this below.

## TODO

---

`for` and `for-in` paren-free (migration speed-bump for early error in order to reform `for-in` semantics to be based on [iterators](#)).

Also, `for-in` on arrays iterates values not keys.

*KeywordStatement* : *VariableStatement* is dubious but plausible in ES1-5, but `let` and `const` are not allowed as direct (unbraced) kids of `if`, etc. So *VariableStatement* must not produce `let` or `const` declarations, only `var` declarations.

— *Brendan Eich* 2011/03/24 20:45

[RSS](#) [XML FEED](#)

 [LICENSED](#)

 [DONATE](#)

[PHP](#) [POWERED](#)

 [XHTML 1.0](#)

 [CSS](#)

 [DOKUWIKI](#)

Trace: » enumeration » arrow\_function\_syntax »  
name\_property\_of\_functions » paren\_free » versioning

## Prior Work

See [versioning](#) for the back-story.

## Goal Thoughts

The goal is to allow new versions of ES with syntactic and non-syntactic extensions to degrade gracefully in downrev browsers, while upgrading the user experience for uprev browsers without taxing developers too much.

Audiences competing here include implementors, developers, and ultimately users who benefit from progressive enhancement. On the Web, with Ajax apps, progressive enhancement may mean the regressive experience is slightly less shiny (no CSS transition), or perhaps a "Web 1.0" window-popup or form field instead of CSS popup or editable text widget.

For JS hackers wanting to use new versions of ES, the problems consist of selecting and detecting support for the new version and compensating with fallback for lack of new features. For browser implementors, complexity and version combination explosion may be overriding.

## Issue Dump

This is a brain-dump or laundry-list. It might better be factored into audience-specific use-cases from which we can derive requirements.

- 

- Object model extensions generally fly under "the default version" you get with

- `<script>...</script>`.

- 

- Developers can object-detect in order to feature-detect, they do not need to use `<script>` attributes or extra `version=...` parameters.

- 

- We should try to continue this to avoid explosive combination of feature-hiding code in implementations.



- - But for new syntax, developers need to hide `<script>` content using that syntax from downrev browsers.
- Script tag `version` parameter values should be well-defined for ES standards.
- Script tag `version` parameter values defined also for multi-vendor experimental versions if it makes sense.
- In-language version selection via a use `version n;` pragma. This wins for many developers who are not good script-tag configurators.
- Something like `versioning`'s `__MAX_ECMA_SCRIPT_VERSION__` property, for runtime script version selection.
- A different notion of "max", for frame-wise restriction to a maximum version of the language to preserve security proofs.

  - Crock points out this could be abused and tend to freeze progress.
- A frame-wise version selection that sets the default for all script tags and event handlers was `specified` in HTML4 but not implemented by all browsers.

  - We might want an HTML5 solution that is easier to express than an HTTP header equated via a `<meta>` tag!
  - "Min" as well as "Max" version stipulation might be necessary for certain web apps or pages.
  -

Better syntax with alternative content than the old `<script>` tag, which has no fallback markup within its container, might be appropriate.

— *Brendan Eich* 2009/09/23 22:52

[RSS](#) [XML FEED](#)










Trace: » enumeration » arrow\_function\_syntax »  
name\_property\_of\_functions » paren\_free » versioning

## Recent Changes

---

The following pages were changed recently.

- 2011/05/18 06:32   [harmony:binary\\_data\\_discussion](#) lukeh
- 2011/05/18 05:58   [strawman:typed\\_arrays](#) lukeh
- 2011/05/18 02:08   [strawman:classes\\_with\\_trait\\_composition](#) markm
- 2011/05/18 01:49   [strawman:private\\_names](#) lukeh
- 2011/05/18 01:37   [strawman:proto\\_operator](#) allen
- 2011/05/18 01:07   [conventions:avoid\\_strictness\\_contagion](#) markm
- 2011/05/18 00:55   [strawman:unique\\_string\\_values](#) allen
- 2011/05/18 00:38   [strawman:arrow\\_function\\_syntax](#) waldemar
- 2011/05/17 20:34   [strawman:support\\_full\\_unicode\\_in\\_strings](#) added ref to W3C requests document allen
- 2011/05/17 07:39   [strawman:trait\\_composition\\_for\\_classes](#) markm
- 2011/05/17 06:57   [strawman:simple\\_maps\\_and\\_sets](#) markm
- 2011/05/15 22:57   [strawman:branding\\_classes](#) markm
- 2011/05/15 22:03   [harmony:weak\\_maps](#) markm
- 2011/05/15 17:26   [strawman:deferred\\_functions\\_draft](#) markm
- 2011/05/15 16:06   [strawman:concurrency](#) markm
- 2011/05/15 07:12   [strawman:strawman](#) markm
- 2011/05/14 00:49   [strawman:name\\_property\\_of\\_functions](#) brendan
- 2011/05/11 22:16   [strawman:guards](#) typo fix brendan
- 2011/05/11 21:27   [strawman:object\\_initialiser\\_shorthand](#) brendan
- 2011/05/11 20:31   [harmony:egal](#) more concrete syntax and naming brendan
- 2011/05/06 22:48   [strawman:i18n\\_api](#) cira



Trace: » [arrow\\_function\\_syntax](#) »  
[name\\_property\\_of\\_functions](#) » [paren\\_free](#) » [versioning](#) » [proposals](#)

## Harmony

---

See [harmony](#) for requirements, goals, and means informing and guiding the proposals under development for ES-Harmony.

Lacking a formal language specification, the following list is not definitive, but it should reflect consensus achieved so far in TC39. Anything can be revisited for a good reason, of course.

The following list will grow, but not without bound before the next ECMA-262 Edition is constructed. Without prematurely triaging or using "ES6" (which might have to be used for a short-term edition, worst-case), for now let's focus on near-term proposals while keeping important longer-term [strawman](#) proposals warm.

— *Brendan Eich 2009/07/29 23:51*

## Proposals

---

- - [Block scoped bindings](#) (markm):
    - [let](#), the new `var` but block-scoped and with better use-before-set semantics (dherman, markm)
    - [const](#), for constant `let`-like bindings (dherman,markm)
    - [block functions](#), `let`-scoped functions declared directly in block statements (markm)
- [destructuring](#) assignment and binding declaration forms, based on object and array initialiser syntax (allen)
-

[parameter default values](#), supplying default arguments to trailing optional parameters

(allen)

- [rest parameters](#), for a trailing formal parameter capturing variable actual arguments as an array (allen)

- [spread](#), the . . . prefix operator for expanding an array actual parameter into its elements (allen)

- [proxies](#), a “catchall” mechanism for intercepting property accesses generically (tomvc)

- [proxy defaultHandler](#), a default Proxy forwarding handler. (tomvc, markm)

- [proxies\\_semantics](#), explains the semantics in terms of the ES5 spec’s internal operations (tomvc)

- [extended object api](#) standardizing some “missing” ES5 methods on Object. (tomvc)

- Collections

- [weak maps](#) non-enumerable object-identity-based tables. Fixes a crucial memory leak in conventional weak-key tables. (markm)

- [egal](#) (markm) an identity-testing operator inspired by Henry Baker’s [egal](#). (Not itself a collection, but some collections will rely on it.)

- [proper tail calls](#) (dherman)

- [modules](#) and [module loaders](#) (dherman)

- [iterators](#), for better collections, more usable `for-in` constructs, and proxy enumeration that scales (brendan, dherman)
- [generators](#), Pythonic/Iconic generator functions that can yield multiple values while suspending their activation in between yields
- [generator expressions](#), convenient, expressive syntax for creating lazily-computed generators (brendan, dherman)
- [binary data](#), convenient, high-level, structured binary data (dherman)
  - [binary data semantics](#)
  - [binary data discussion](#)
- Number improvements
  - [Number.isFinite](#)
  - [Number.isNaN](#)
  - [Number.isInteger](#)
  - [Number.toInteger](#)
- Regular Expression improvements
  -

[regexp y flag](#), the "sticky" flag causing a regexp to anchor at lastIndex

- 

String improvements

- 

[String.prototype.repeat](#)

- 

[typeof null](#), a long-awaited bug fix to typeof (brendan crock)

harmony/proposals.txt · Last modified: 2011/04/28 00:46 by brendan

[RSS](#) [XML FEED](#)





Trace: » [name\\_property\\_of\\_functions](#) » [paren\\_free](#) »  
[versioning](#) » [proposals](#) » [strawman](#)

## About this Directory

---

This 'strawman' namespace is intended to contain proposals for the "ES-Harmony" language that are not yet approved [harmony proposals](#), and to clearly separate them from the legacy ES4 pages.

## Proposals

---

- - Concurrency, Asynchrony, and Distributed Programming
    - [concurrency](#) (markm)
    - [deferred\\_functions](#) allow writing asynchronous code in a linear style where you would otherwise use callbacks and manual CPS (peterhal)
- [proto operator](#) (allen)
- Array
  - [array create](#) (crock)
    - also see [proto operator](#)
  - [array subtypes](#), for allowing construction of array instances with prototype other than Array.prototype (olliej)

[array statics](#) (dherman)

•

## Number and Math Enhancements

◦

[more Math fun](#), see [Alistair Braidwood's message](#)

◦

[Spreadsheet comparing ECMAScript Math functions to various C/C++ math libraries](#)

◦

[number compare](#) (crock)

◦

[number EPSILON](#) (crock)

◦

[number MAX\\_INTEGER](#) (crock)

◦

[random-er](#), or a better (cryptographically strong) random number generator (see [Mozilla bug 322529](#))

▪

See also the [crypto.getRandomValues spec](#) from Adam Barth

•

## String

◦

[string format](#) (crock)

◦

[string format](#) (shanjian)

◦

[string extras](#) (dherman)

•

## Functions

- [name property of functions](#) (brendan)

- 

- [parameters property of functions](#) (crock)

- 

- [function to string](#) – greater specification for `problematic Function.prototype.toString` (allen)

- 

- [shorter function syntax](#), for defining and (or possibly only) expressing functions concisely (arv)

- 

- [const functions](#) (markm) now with a joining optimization

- 

- [arrow\\_function\\_syntax](#), a synthesis of the above two with “Harmony of My Dreams” and CoffeeScript’s syntax (brendan)

- 

- [fix function name binding](#) (Allen) Make the binding of function names consistent between *FunctionExpressions* and *FunctionDeclarations*

- 

- [soft\\_bind](#) (alex, arv, markm) A binding operator intermediate between JavaScript’s current loose binding and the tight binding of `Function.prototype.bind`.

- 

## Regular Expressions

- 

- [regexp x flag](#) (crock)

- 

- [multiline regexps](#) (brendan, crock)

- 

- [match web reality](#) (allen)

-

## Steve Levithan RegExp API improvements

- 

- [regexp look-behind support](#)

- 

## JSON

- 

- [JSON path](#) (crock)

- 

- [lexical scope](#) (dherman)

- 

## Modules

- 

- [simple modules](#) (with [examples](#)) and [module\\_loaders](#) (dave, samth)

- 

- [simple module functions](#) (markm)

- 

- ([modules\\_harmonic](#), (ihab\_awad) temporary placeholder)

- 

- [modules\\_primordials](#)

- 

- [modules\\_emaker\\_style](#)

- 

- [system](#), a place to put powerful objects provided by the embedding without having to introduce names into the global scope every time.

- 

- [modules\\_packages](#)

- 

## Operators

- [default operator](#) (crock)

- 

- [modulo operator](#) (crock)

- 

- [has operator](#) (crock)

- 

## Value types

- 

- [value types](#), requirements for first-class number-like objects with operators and (we hope) literal syntax (brendan)

- 

- [value proxies](#), extending [proxies](#) to implement value types (cormac)

- 

## Garbage collection

- 

- [gc semantics](#) Thoughts on specifying the semantics of garbage collection. (markm)

- 

- [weak references](#) and post-mortem finalization. (markm)

- 

- [proper tail calls](#) (markm)

- 

- [inherited explicit soft fields](#) (markm) – an encapsulation-respecting alternative to [private names](#) above. See comparison at [names vs soft fields](#).

- 

## Data structures

- 

- [binary data](#), convenient, high-level, structured binary data (dherman)

-

## binary data semantics

- 

## binary data discussion

- 

**typed arrays**, similar to the above **byte arrays**, as originally defined by **webgl** (arun)

- 

**encapsulated hashcodes** (allen – see also **weak maps**)

- 

**simple maps and sets** (markm)

- 

**records** (brendan, dherman)

- 

**tuples** (brendan, dherman)

- 

**dicts** (dherman)

- 

## Loops, iteration, enumeration

- 

**iterator conveniences**, an API for convenient construction of iterators (dherman)

- 

**enumeration**, more fully-specified semantics for property enumeration in `for-in` loops (brendan, dherman)

- 

**array comprehensions**, convenient, expressive syntax for creating eagerly-computed arrays (brendan, dherman)

- 

## **Private Names** providing unique, unforgable property names (allen)

-

[Instance Variables](#) work in conjunction with [Private Names](#) to provide strong encapsulation/information hiding.

o

Also see discussion links at end of [Private Names](#) page.

•

Declarative Abstractions Based on Object Literals (allen)

o

[Declarative Object and Class Abstractions Based Upon Extended Object Initialisers](#)

including:

▪

[Object Initialiser Meta Properties](#)

▪

[Method Properties](#)

▪

[Other Object Initialiser Property Attribute Modifiers](#)

▪

see [proto operator](#) and [concise object literal extensions](#) for an alternative to the above 3 proposals

▪

[Object Initialiser Initialization Blocks](#)

▪

[Implicit property initialization expressions](#) (brendan)

▪

[Class Initialisers](#)

▪

[super in Object Initialisers](#)

o

The [Private Names](#) extension integrates with extended Object Initialisers:

## Private Names in Object Initialisers

- 

Also see [destructuring](#) for “record extension and row capture” via . . . – this may need its own proposal page

- 

Similar declarative object extension mechanisms

- 

[scoped object extensions](#) (peterhal)

- 

High Integrity Factories

- 

[classes with trait composition](#) (markm)

- 

[trait\\_composition\\_for\\_classes](#) (markm, tom)

- 

[traits semantics](#) (tom)

- 

[guards](#) A syntax for dynamic type-like checks (markm, waldemar)

- 

[trademarks](#) Simple semantics for dynamic type-like checks (markm, waldemar)

- 

Versions and Configuration

- 

[versioning](#), the full versioning of script tag content and whole-frame/window object model monty (brendan)

- 

[pragmas](#), a future-proof “use” directive for language modes and implementation options (dherman)



- [quasis](#), quasi-literals provide DSL support for content generation and introspection (mikesamuel)
- [ast](#), a parser built into conforming Harmony implementations which returns a standard abstract syntax tree (dherman)
- [extended Object API](#), standardizing some “missing” ES5 methods on Object. (tomvc)
- [proxy extensions](#), not-yet-harmonized extensions to [proxies](#). (tomvc)
  - Deferred: [proxy instanceof](#), enabling proxies to trap `instanceof` directly. (tomvc, markm)
  - [function proxy prototype](#), enabling function proxies to specify a custom prototype object (dherman)
  - [proxy derived traps](#), derived `getOwnPropertyDescriptor` and `getOwnPropertyNames` traps (tomvc)
  - [handler access to proxy](#), giving Proxy handlers access to the intercepted proxy (tomvc)
  - [proxy set trap](#), remove a discrepancy between the default set trap behavior of Proxies vs regular ES5 objects (tomvc)
  - [derived traps forwarding handler](#), semantics of the derived traps of the Proxy default forwarding handler (tomvc)
  - [proxy drop receiver](#), dropping the `receiver` parameter from `get` and `set` traps

(tomvc)

- [paren free](#), relaxing the rules about mandatory parentheses (brendan, dherman)
- [completion reform](#), to make "completion position" a statically predictable attribute (dherman)
- [completion let](#), a variation on `let` expressions that uses completions (dherman)
- [debugger expressions](#), to extend the syntax of `debugger` to be an expression (dherman)
- [multiple globals](#), bringing the spec in line with the reality of multiple global objects (dherman)

#### Conditionals

- [catch guards](#), for conditionally catching exceptions (dherman)
- [pattern matching](#), a conditional form based on destructuring (dherman)
- [cond expressions](#), for concise linear nesting of conditional expressions (dherman)

#### Specification Techniques

- [specification language](#), using Harmony to describe Harmony.
- Another alternative, translating the ES5 spec. into a JavaScript-based definitional interpreter [es5definterp.js.txt](#)

## Internationalization support

- [Support full Unicode in strings](#): remove all 16-bit Unicode assumptions concerning strings and source code
- [Unicode support](#), tools to aid in using Unicode.
- [i18n API](#), internationalization API (locales, sorting, formatting, parsing).

## ECMAScript object model and internal metaobject protocol (Allen)

- [ES5 internal methods](#) and aligning them with Proxy handlers
- [ES5 internal nominal typing](#) and generalizing usage of `[[Class]]`

## Deferred proposals

---

See the [deferred](#) proposals page.

strawman/strawman.txt · Last modified: 2011/05/15 07:12 by markm



[[strawman:  
completion\_reform]]

Trace: » paren\_free » versioning » proposals » strawman  
» 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
}
```

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

```
{
```

```
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.

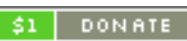
— *Brendan Eich* 2011/03/01 00:24

strawman/completion\_reform.txt · Last modified: 2011/03/01 00:27 by brendan

[RSS](#) [XML FEED](#)



LICENSED



DONATE



POWERED



XHTML 1.0



CSS



DOKUWIKI

[[strawman:  
proxy\_drop\_receiver]]

Trace:

»

versioning » proposals » strawman » completion\_reform »

proxy\_drop\_receiver

## Dropping receiver from get and set traps

As noted by Sean Eagan on es-discuss, the `receiver` parameter to the `get` and `set` traps of Proxy handlers is not strictly necessary.

The purpose of this parameter was to allow proxy handlers to refer to the original receiver of a property access/assignment operation, in the case where the proxy acts as a prototype:

```
// according to the current Proxy spec:
var p = Proxy.create({
  get: function(receiver, name) { ... }
});
var o = Object.create(p);
o.foo; // triggers p's get trap with receiver === o and name === "foo"
```

However, according to the ES5 `[[Get]]` algorithm (section 8.12.3), this is not how inherited property lookup will occur. The algorithm calls `[[GetProperty]]` (section 8.12.2), which in turn walks the prototype chain to look up a *property descriptor*. This will trigger `p`'s `getPropertyDescriptor` trap, not its `get` trap:

```
// according to the current ES5 semantics:
var p = Proxy.create({
  getPropertyDescriptor: function(name) {
    ...
  }
});
var o = Object.create(p);
o.foo; // triggers p's getPropertyDescriptor trap with name === "foo"
```

Under this semantics, the `get` trap will only ever be invoked for direct invocations on `p`, in which case `receiver` will always be equal to `p`. Pending the acceptance of the [strawman](#) that adds a `proxy` argument to each trap, `receiver` becomes unnecessary.

Note: the same reasoning applies to the `set` trap: ES5 `[[Put]]` (section 8.12.5) also calls `[[GetProperty]]` to find the appropriate property descriptor when it is not found on the receiver object itself.

Note: it is still possible for a proxy handler to get hold of the receiver object when its proxy is used

as a prototype, via the `this`-binding of a function data property or getter/setter:

```
var p = Proxy.create({
  getPropertyDescriptor: function(name) {
    return { value: function() { /* this === receiver */ } };
  }
});
var o = Object.create(p);
o.foo(); // calls the above nested function with this === o
```

The above also works for accessor properties.

— Tom Van Cutsem 2011/05/04 11:55

## Consequences

---

Dropping `receiver` as the first argument to `get` and `set` traps is not backwards-compatible with the current API. Under this strawman, the first argument to `get` and `set` would be `name` (a string), not `receiver` (an object/proxy).

## References

---

- 

[discussion on es-discuss](#)

## Feedback

---

strawman/proxy\_drop\_receiver.txt · Last modified: 2011/05/04 07:01 by tomvc



[[harmony:  
proxy\_defaulthandler]]Trace: » proposals » strawman »  
completion\_reform »

proxy\_drop\_receiver » proxy\_defaulthandler

## Default Proxy forwarding handler

Goal: to standardize a default forwarding handler that delegates all meta-level operations applied to a proxy to a given target object, as exemplified [here](#).

Rationale: this is a common handler, required as a starting point by most abstractions that wrap existing JS objects. The default forwarding handler is also required in the [double lifting](#) pattern.

Advantages of standardizing a default forwarding handler:

- 
- Wrapper proxies don't need to define this handler over and over again,
- 
- The code for the default handler doesn't need to be downloaded over and over again,
- 
- The default handler evolves in sync with potential changes to the Proxy API,
- 
- A built-in implementation is likely to be faster than a no-op forwarding handler defined in JS itself

### Forwarding Handler constructor

The following is a revised API based on the standard Javascript constructor pattern.

```

Proxy.Handler = function(target) {
  this.target = target;
};

Proxy.Handler.prototype = {

  // == fundamental traps ==

  // Object.getOwnPropertyDescriptor(proxy, name) -> pd | undefined
  getOwnPropertyDescriptor: function(name) {
    var desc = Object.getOwnPropertyDescriptor(this.target, name);
    if (desc !== undefined) { desc.configurable = true; }
    return desc;
  },

  // Object.getPropertyDescriptor(proxy, name) -> pd | undefined
  getPropertyDescriptor: function(name) {
    var desc = Object.getPropertyDescriptor(this.target, name);
    if (desc !== undefined) { desc.configurable = true; }
    return desc;
  },
};

```

#### Table of Contents



- Default Proxy forwarding handler
  - Forwarding Handler constructor
- Open Issues
  - Alternative names
  - Alternative implementation for default set trap
  - Default implementation of fix ()
- Feedback and History



```

// Object.getOwnPropertyNames(proxy) -> [ string ]
getOwnPropertyNames: function() {
    return Object.getOwnPropertyNames(this.target);
},

// Object.getPropertyNames(proxy) -> [ string ]
getPropertyNames: function() {
    return Object.getPropertyNames(this.target);
},

// Object.defineProperty(proxy, name, pd) -> undefined
defineProperty: function(name, desc) {
    return Object.defineProperty(this.target, name, desc);
},

// delete proxy[name] -> boolean
delete: function(name) { return delete this.target[name]; },

// Object.{freeze/seal/preventExtensions}(proxy) -> proxy
fix: function() {
    // As long as target is not frozen, the proxy won't allow itself to be fixed
    if (!Object.isFrozen(this.target)) {
        return undefined;
    }
    var props = {};
    for (var name in this.target) {
        props[x] = Object.getOwnPropertyDescriptor(this.target, name);
    }
    return props;
},

// == derived traps ==

// name in proxy -> boolean
has: function(name) { return name in this.target; },

// ({}).hasOwnProperty.call(proxy, name) -> boolean
hasOwn: function(name) { return ({}).hasOwnProperty.call(this.target, name); },

// proxy[name] -> any
get: function(receiver, name) { return this.target[name]; },

// proxy[name] = value
set: function(receiver, name, value) {
    if (canPut(this.target, name)) { // canPut as defined in ES5 8.12.4 [[CanPut]]
        this.target[name] = value;
        return true;
    }
    return false; // causes proxy to throw in strict mode, ignore otherwise
},

// for (var name in proxy) { ... }
enumerate: function() {
    var result = [];
    for (name in this.target) { result.push(name); };
};

```

```

    return result;
  },

  /*
  // if iterators would be supported:
  // for (var name in proxy) { ... }
  iterate: function() {
    var props = this.enumerate();
    var i = 0;
    return {
      next: function() {
        if (i === props.length) throw StopIteration;
        return props[i++];
      }
    };
  }, */

  // Object.keys(proxy) -> [ string ]
  keys: function() { return Object.keys(this.target); }
};

```

To create a default forwarding proxy to an object obj, one would write:

```

var h = new Proxy.Handler(obj);
var p = Proxy.create(h);

```

To modify one of the default traps, one can either override traps on a default handler or use prototype inheritance. For example:

```

// using assignment
var h = new Proxy.Handler(obj);
h.get = function(rcvr, name) { ... };
var p = Proxy.create(h);

// using inheritance
function MyHandler(target) {
  Proxy.Handler.call(this, target); // constructor chaining
}
MyHandler.prototype = Object.create(Proxy.Handler.prototype);
MyHandler.prototype.get = function(rcvr, name) { ... };

var h2 = new MyHandler(obj);
var p2 = Proxy.create(h2);

```

Pros of this API:

- Familiarity to Javascript developers.
- Handler inheritance is straightforward.
- All default traps are shared among all default handler instances.

Cons of this API:

- The constructor pattern is subject to the bug of forgetting `new`, in which case `Proxy.Handler(obj)` will set a `target` property on `Proxy`.
- It's awkward that handlers are created using constructor functions (requiring `new`) whereas proxies are created using a factory method (not requiring `new`). This makes the API feel a little inconsistent.

— Tom Van Cutsem 2010/12/14 3:10

## Open Issues

---

### Alternative names

We can debate about alternative names for `Handler` and `target`.

If the `Proxy` API would be contained in a `Harmony` module, it may make sense to introduce `Handler` as an exported variable, next to `Proxy`, instead of making it a property on `Proxy`.

It was noted that calling the default forwarding handler `Handler` is potentially confusing (not all handlers are forwarding handlers). Possible alternative: `Forwarder`.

— Tom Van Cutsem 2011/05/04 12:15

### Alternative implementation for default set trap

As currently defined, the default `set` trap's behavior is counter-intuitive in the case of a "chain" of proxies (a proxy forwarding to another proxy):

```
set: function(receiver, name, value) {
  if (canPut(this.target, name)) { // canPut as defined in ES5 8.12.4 [[CanPut]]
    this.target[name] = value;
    return true;
  }
  return false; // causes proxy to throw in strict mode, ignore otherwise
},
```

If `this.target` is a proxy, the `canPut` auxiliary function will trigger that proxy's `getOwnPropertyDescriptor` and `getPropertyDescriptor` traps to determine whether the property can be set. Only then is the assignment performed on `this.target` and is that proxy's `set` trap invoked.

Part of the awkwardness lies in the fact that the "inner" `set` returns its own boolean to indicate success, but that boolean isn't accessible to the "outer" `set`. Instead each proxy in the chain tests the boolean and either throws or ignores it. MarkM suggests the following refactoring of the internal spec methods which would make this chaining of `set` calls more intuitive:

Since the system itself will provide the default traps, the default `set` trap could call a new internal `[[Set]](P,V)` method which returns a boolean, such that `[[Put]]` would be redefined as:

```
8.12.5 [[Put]](P,V,Throw)
```

```
If the result of calling [[Set]](P,V) is true, return.
```

```

    Else if Throw is true, throw a TypeError exception.
    else return.

```

The `[[Set]]` method on regular objects would be defined as is the current `[[Put]]` but returning a boolean rather than conditionally throwing. The `[[Set]]` method on proxies would call the `set` trap. The default `set` trap would call `[[Set]]` on `this.target`. So this default `set` trap is the primitive by which the ability to call `[[Set]]` is exposed.

The one problem with this plan is `[[Set]]` on an object that inherits from a proxy. This plan would still go through the `[[CanPut]]` logic on the derived object which would still trigger the `[[GetProperty]]` and `[[GetOwnProperty]]` traps on the proxy. So there's not much difference in the inherited case. But the direct chaining case is more direct and intuitive.

— Tom Van Cutsem 2011/01/12 3:10

## Default implementation of `fix()`

The above implementation of `fix()` was written without giving much thought to the consequences. On second thought, the above default implementation is potentially unsafe: if a proxy handler inherits from the default forwarding handler, but does not override `fix()`, and it forwards to a frozen object, then freezing the proxy will fix it by fully bypassing the handler. This would mean that any intercepting behavior (e.g. for access control, logging, ...) that the handler specified would no longer be called, which can be surprising. A more conservative and safer default implementation of the `fix` trap would just be to always return `undefined`. This implies that default forwarding proxies can't be fixed unless `fix` is explicitly overridden.

— Tom Van Cutsem 2011/05/04 12:20

## Feedback and History

TC39 January 2011 meeting:

From Waldemar's notes:

Proxy default handler: Some trivial bugs in the code: Calling `getOwnPropertyDescriptor` etc. with only one argument. `desc` in `"desc.configurable = true"` can be `undefined`.

`set/put/canPut` problem discussion. Allen: Clean up the list of primitive methods and handlers. MarkM: All existing uses of `put` can be written in terms of `[[Set]]`. Waldemar: Would want a more generic way of invoking `[[Set]]` rather than having to instantiate a new default proxy. Brendan: Issue remains with prototype chain.

Agreed to move this to proposal stage, with some open issues.

— Tom Van Cutsem 2011/01/24 10:39

TC39 November 2010 meeting: agreement that a default forwarding handler should become part of the spec.

A first iteration of this API required handlers to be created using a factory method:

```

var handler = Proxy.handlerFor(target);

```

Drawback of this API: one cannot inherit from a shared handler prototype. Waldemar: why not define an API based on prototypes and constructor functions? The revised API was formulated in response to this.

— Tom Van Cutsem 2010/12/15 2:53