Ecma/TC39/2011/034

# [[strawman: handler_access_to_proxy]]

ES Wiki



Table of Contents


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


## Handler access to proxies

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

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

### Extended API

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

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

#### Proxy as optional argument

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

Pro:

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

Con:

- 

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

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

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

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

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

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

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

## Proxy as additional argument

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

Pro:

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

Con:

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

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

```
// fundamental traps
getOwnPropertyDescriptor: function(proxy, name) -> PropertyDescriptor | undefined //
Object.getOwnPropertyDescriptor(proxy, name)
getOwnPropertyNames:      function([proxy]) -> [ string ]                       //
Object.getOwnPropertyNames(proxy)
defineProperty:           function(proxy, name, propertyDescriptor) -> any      //
Object.defineProperty(proxy,name,pd)
delete:                   function(proxy, name) -> boolean                      //
delete proxy.name
fix:                      function([proxy]) -> { string: PropertyDescriptor }   //
Object.{freeze|seal|preventExtensions}(proxy)
                                                | undefined
// derived traps
getPropertyDescriptor:    function(proxy, name) -> PropertyDescriptor | undefined //
Object.getPropertyDescriptor(proxy, name)   (not in ES5)
```

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

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

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

#### Proxy as argument only for particular traps

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

### Reservations

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

— *Tom Van Cutsem 2011/02/28 06:10*

## References

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

## Feedback

Discussed at the March 2011 TC39 meeting.

Q: Why not consider proxy as an instance variable of the handler? A: Would preclude sharing a single handler among multiple proxies, or else handler would have to keep track of all the proxies it is serving as part of its instance state.

Q: why not bind a handler's `this` to the proxy it is serving? A: confuses meta-levels (this = handler = meta-level, proxy = base-level). Also, makes handler inheritance unworkable: can inherit from a handler, in which case `this` in a parent handler should refer to the inheriting handler.

General agreement that we may want to provide the proxy as an argument to all traps.

Andreas: experimenting with DOM wrappers. All prototype-climbing traps require access to the receiver object (which is not necessarily the proxy object), not just the get/set traps. The get and set trap may want access to both the receiver and the proxy.

Tom: propose to add `proxy` as a first argument to all traps. Andreas, Brendan, Dave: in favor of adding it as an optional last argument:

- in most use cases that came up thus far, there was no need for the `proxy` parameter
- `proxy` parameter is dangerous (runaway recursion hazard), good to be able to ignore it most of the time
- Tom: what about inconsistent ordering w.r.t the trapped code? Dave: Proxy API is for experts, they will cope.

Current consensus:

- add `receiver` as a first argument to all prototype-climbing traps (get, set, has, getPropertyNames, getPropertyDescriptor traps)
- add proxy as an optional last argument to all traps.

— *Tom Van Cutsem 2011/03/30 12:54*

# [[strawman: proxy_drop_receiver]]

ES Wiki

Trace: » handler_access_to_proxy » 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

[[harmony:
specification_drafts]]

ES
Wiki



# Draft Specification for ES.next (Ecma-262 Editon 6)

This page contains a historical record of working draft of the ES.next specification prepared by the project editor.

Error in the current draft should be reported as bugs at bugs.ecmascript.org

## Current Working Draft

July 12, 2011 Draft doc pdf This is the first working draft of the ES.next specification. Notable changes from the ES5.1 spec include:

- 5.1.4 Introduction of the concept of supplemental grammars.
- 5.3 Introduction of concept of Static Semantic Rules
- 8.6.2 and various places. Eliminated [[Class]] internal property. Added various internal trademark properties as a replacement.
- 10.1.2 defined the concept of "extended code" that means code that may use new Es.next syntax. Also redefined "strict code" to mean either ES5 strict mode code or extended code.
- 11.1.4 added syntax and semantics for use of spread operator in array literals
- 11.1.5 added syntax and semantics for property value shorthand and various semantic helper abstract operations.
-

11.2, 11.2.4 added syntax and semantics for spread operator in argument lists

- 11.13 Add syntax and semantics for destructuring assignment operator.

- 12.2 Added BindingPattern syntax and partial semantics to support destructuring in declarations and formal parameter lists.

- 13 Added syntax to support rest parameter, parameter default values, and destructuring patterns in formal parameter lists. Also static semantics for them. However, instantiation of such parameters is not yet done. Defined the argument list "length" for such enhanced parameter lists.

- 15 Clarified that clause 15 function specifications are in effect the definition of [[Call]] internal methods.

- 15.2.4.2 Respecified toString to not use [[Class]]. Note that adding an explicit extension mechanism is still a to-do.

- Annex B Retitled as normative optional features of Web Browser ES implementations

## Older Drafts

Initial Baseline document: doc
This is the starting point for the ES.next spec. It is just the ES5.1 specification with a ES6 draft cover and copyright notice.

harmony/specification_drafts.txt · Last modified: 2011/07/13 15:50 by allen

Table of Contents

- Object Initializer super references
  - Overview
  - super in Accessor Property Definitions
  - Functions that reference super
  - Imperatively binding [[Super]]
  - Rationale

# Object Initializer super references

In object-oriented programming languages, inheritance is used to perform behavioral composition. An object combines together the properties it inherits from its prototypes with new and replacement properties define for the specific object. Occasionally in creating such compositions it necessary for the object to explicitly access prototype behavior that is over-ridden by the object.

Traditionally this has been accomplished in object-oriented languages using the keyword `super`. ECMAScript has reserved this keyword but has never supported it. One of the reasons was that imperative object construction style that was typically used did not provide sufficient context for a function declaration to determine how to access over-ridden properties.

Object Initialisers provide sufficient context and hence this proposal allows use of the keyword in functions defined within an Object Initialiser.

## Overview

Sometimes a method on an object that over-rides a prototype method needs to invoke the over-ridden method. Consider for example:

```
var sup = {
   validate() { /* validate internal invariants */}
}

var sub = sup <| {
   validate() {
       /* validate invariants imposed by prototype */
       /* validate instance specific invariants */
    }
}
```

How should the object `sub` actually code the call to its prototype’s validate method? The idiom that would most likely be used today would be an expression of the form:

```
      sup.validate.call(this);
      /* validate subclass invariants */
```

This formulation does the job, but is idiomatic and its intent may not be obvious to readers. Also, it requires explicitly referencing the prototype object by name. If such references in multiple places is error prone, especially when a prototype hierarchy is being refactored. These issues are addressed by adding a super call expression as an additional form of *PrimaryExpression*:

***PrimaryExpression*** **:** …
**`super`**

The value of `super` is the same as the value of `this` but when `super` is used as the base of a property access the property lookup starts with the object that is the prototype of the object defined by the object literal that contains the reference to `super`.

Then above example could then be coded like:

```
var sub = sup <| {
   validate() {
       super.validate();
       /* validate instance specific invariants */
   }
}
```

In this example, the expression `super.validate()` means `sup.validate.call(this)`. But note that `sup` does not have to be referenced by name with the body of validate.

A *PrimaryExpression* containing `super` may only occur within code that is part of a function body. It is an early SyntaxError error for `super` to occur in global.

Use of `super` is not limited to *CallExpression*. It can also be used to get or set the value of accessor properties and to get the value of data properties defined by the prototype, even if the property is over-ridden by the object:

```
var f=0;
var sup = {
   k:= 1,
   get foo() {return f},
   set foo(v){f=v}
}

var sub = sup <| {
   get foo() {
      var supFoo = super.foo;
      print("getting foo: "+supFoo);
      return supFoo;
   },
   set foo(v){super.foo = v+super.k}
}
```

But note that the use of `super` does not change the semantics of assigning to an inherited data property. Such assignments create own properties even if `super` was used to access the property.

```
var f=0;
var sup = {k: 1};

var sub = sup <| {
   setK () {
      print(super.k); // prints 1
      print(this.k);  // prints 1 -- inherited from sup
      super.k = 2;
      print(super.k); // prints 1 -- assignment created own k
      print(this.k);  // prints 2 -- own property created by assignment
   }
}
```

## super in Accessor Property Definitions

In object initialisers the **get** function and **set** function of an accessor property are defined independently of each other and only either one of them need to present to over-ride an inherited accessor property definition. If either the **get** or **set** function is not present then the default definition is used:

```
var f = 1;
var sup = {
   get f() {return f}
};

var sub = sup <| {
   set f(v) {f=v}
};
print(sup.f);  //1
sub.f=2;
print(sup.f);  //2
print(sub.f);  //undefined
```

The reason `sub.f` prints `undefined` is because the definition of a `set f` accessor in sub implicitly cause the default `get f` accessor to also be created. A default `get` accessor always returns the value undefined.

This problem can be avoided by explicitly defining a `get f` accessor that uses `super` to invoke the over-ridden accessor:

```
var f = 1;
var sup = {
   get f() {return f}
};

var sub = sup <| {
   set f(v) {f=v},
   get f() {return super.f}
};
print(sup.f);  //1
sub.f=2;
print(sup.f);  //2
print(sub.f);  //2
```

Needing to over-delegate to a prototype's `get` or `set` accessor function is common enough that a special definition form is provided for such definitions. The syntax is:

***PropertyAssignment*** : …
**set super get** *PropertyName* **( ) {** *FunctionBody* **}**
**get super set** *PropertyName* **(** *PropertySetParameterList* **) {** *FunctionBody* **}**

These forms cause to `set` or `get` accessor function to be automatically set to a function that does a `super set` or `get` access for that property.

Using this form of property definition, the above example object definition could be simplified to:

```
var sub = sup <| {
   get super set f(v) {f=v}
};
print(sup.f);  //1
sub.f=2;
print(sup.f);  //2
print(sub.f);  //2
```

## Functions that reference super

Any function that references super has a [[Super]] internal property whose value is the object that is used as the base

for super property lookups. Functions that *do not* reference **super** do not have [[Super]] internal property. When a function that references **super** is defined as part of an object literal its [[Super]] internal property is set to the same value as the [Prototype]] internal property of the object created by the object literal. Such functions include functions defined using method property definitions, functions defined using **get** or **set** definitions, and functions defined as a *MemberExpression* within the *AssignmentExpression* of a *PropertyAssignment*.

```
var sub = sup <| {
   //all the the following define functions whose [[Super]] internal property is set of the value of sup
   a() {return super.foo()},       // a method
   get b() {return super.bar},     // a get accessor
   set b(v) {super.bar=v},         // a set accessor
   c: function() {super.a()};      // a function in a property initializer
};
```

Functions defined outside of object literals (or class declarations) that reference **super** are created with their [[Super]] internal property set to **null**. Property accesses based off of **super** always perform property lookups start with the object that is the the value of the containing function's [[Super]] internal property. If [[Super]] is **null**, the property lookup immediately fails and produces the value **undefined**.

```
function returnUndefined() {
   return super.foo;
}
print(returnUndefined);  //will print "undefined" because [[Super]] is null
```

If such a function with a [[Super]] internal property is subsequently extracted from its original object and installed (for example, via property value assign or `Object.defineProperty`) as a data property value in some other object, its [[Super]] internal property reference to the original object's [[Prototype]] is not modified. Essentially, when a function references `super` it is statically referencing a specific object that is identified when the function is defined and not the [[Prototype]] of the object from which the function was most recently retrieved.

## Imperatively binding [[Super]]

An object literal is the easiest way to create methods that use super and ensure that the [[Super]] internal property of the function is correctly bounds. However, such bindings can also be made imperatively.

`Object.defineMethod` is a new function that creates a method property of in an object:

```
var obj = Object.create(proto);
Object.defineMethod(obj,'foo', function() {return super.foo});
```

`Object.defineMethod` attempts to create (or modify) a non-enumerable data property whose value is the `defineMethod` call's third argument. That argument must be a function. If the function has a [[Super]] internal property, then a new function object is created and set as the value of the property. The new function is identical in all ways to the argument function except that its [[Super]] internal property is set to the value of the [[Prototype]] internal property of the object upon which the property is being defined.

`Object.defineProperty` and `Object.defineProperties` are extended for situations where they are creating or modify the get or set function of an accessor property. In those situations where the passed get or set function has a [[Super]] internal property it is replaced with a copy of the function with a new [[Super]] internal property in a similar manner to `Object.defineMethod`.

Note that `Object.defineProperty` and `Object.defineProperties` do not create new functions or rebind [[Super]] when modifying the value of data properties. `Object.defineMethod` must be used for that purpose.

## Rationale

A **super** property access is more complex then just skipping the **this** object when doing a property look up. That definition of **super** would actually result in method loops. Consider this example:

```
var top = {identify() {return "top"}};
var middle = top <| {identify()  {return "middle "+ super.identify()}};
var bottom = middle <| {identify()  {return "bottom "+ super.identify()}};
print(bottom.identify());
```

If `super.identify()` is interpret as meaning look of the property `identify` starting with the object that is the [[Prototype]] of **this**, then the following would happen:

1. `bottom.identify()` would call the `identify` method defined in the object literal for `bottom` with **this** set to `bottom`.
2. That method would evaluate `super.identify` based upon the **this** value of `bottom`. Lookup would start with `middle` (the value of `bottom`'s [[Prototype]]). It would find and call the `middle.identify` method, but still pass `bottom` as the **this** value.
3. The `middle.identify` would evaluate `super.identify` based upon the **this** value of `bottom`. Lookup would start with `middle` (the value of `bottom`'s [[Prototype]]) and again find and call `middle.identify`passing `bottom` as the **this** value.
4. execution continue to loop at step 3 with `middle.identify` calling itself.

This shows that super can't be defined in relation to **this**. It has to be defined in relation to the location of the currently executing method in the prototype hierarchy of the **this** value. In particular, a super lookup has to start above the point in the hierarchy where the currently executing method was found in order to avoid looping on that method. There are at least two plausible semantics for such a **super** lookup: dynamic **super** and static **super**.

For dynamic **super**, a **super** lookup would always begin with the [[Prototype]] of the object where the currently executing method had been retrieved as an own property. This presents a reasonable (perhaps the best) usage semantics but presents a problem. How does a super lookup site within a method know where the containing method was retrieved from. The containing method was looked prior to its invocation so the retrieval object is only naturally known outside the invocation of the retrieved function. The retrieval object could be passed as an additional implicit argument in addition to the implicit this argument. This would have to occur on every method invocation. A call site doesn't know whether or not the function it is calling actually needs to use **super** but any function might so it would always be necessary to pass the implicit retrieval object argument. Considering that the most common argument list sizes are zero and one and additional implicit argument on every call would be significant new overhead.

For the static **super** approach, the object to use as the starting point of a **super** lookup is statically associated with each function. This is normally, the [[Prototype]] value of the object that contains the function as an own property (method) value. Because, the **super** lookup object is statically associated with a function it does not need to be passed as an implicit augment to every function. There is no additional per call overhead. However, unlike dynamic **super**, additional mechanism is needed to establish the static **super** lookup association whenever a function that references **super** is installed as a method and this may limit a function to being used as a method of only one object.

Most programming languages including dynamic languages use some form of static **super** in order to avoid the per call overhead of dynamic **super**. This also appears to be the best approach for JavaScript. However, it is necessary to provide^Object.defineMethod" and the related semantics in order to perform the necessary static **super** binding for imperatively constructed objects.



RSS XML FEED CC LICENSED $1 DONATE

PHP POWERED

W3C XHTML 1.0

W3C CSS

DOKUWIKI

# [[strawman:block_vs_object_literal]]

ES Wiki



## Problem Statement

Both arrow function syntax and block lambda revival want, for different reasons, blocks and object literals to be usable in the same context: as an expression-like body of an arrow function, as the zero-argument form of a block-lambda.

Independent interest in statements as expressions comes up on es-discuss from time to time, e.g. here.

let expressions proposed new syntax allowing explicit wrapping of statements to make them expressions, but this strawman was deferred.

Blocks-as-expressions and object literals must have different evaluation semantics (blocks are quoted, their evaluation is deferred until "invocation"; object literals evaluate eagerly). The purpose of this strawman is to refactor the grammar to eliminate grammatical conflicts that prevent treating blocks as expressions in JS today.

## Grammar Changes

```
Block:
    { UnlabeledStatementFirstList? }
    { WellLabeledStatement StatementList? }

UnlabeledStatementFirstList:
    UnlabeledStatement
    UnlabeledStatementFirstList Statement

Statement:
    UnlabeledStatement
    LabeledStatement

UnlabeledStatement:
    VariableStatement
    EmptyStatement
    ExpressionStatement
    ContinueStatement
    ReturnStatement
    LabelUsingStatement
```

```
    DebuggerStatement

LabelUsingStatement:
    Block
    IfStatement
    IterationStatement
    BreakStatement
    WithStatement
    SwitchStatement
    ThrowStatement
    TryStatement
```

A *LabelUsingStatement* is a statement that might possibly use a label in ES3-5 but not in any ES extended to support blocks-as-expressions or block-lambdas. For maximum backward compatibility, *LabeledStatement* (spelled *LabelledStatement* in ECMA-262) remains the same, and *WellLabeledStatement* restricts the statement after one or more labels to be a *LabelUsingStatement*.

```
LabeledStatement:
    Identifier : Statement

WellLabeledStatement:
    Identifier : LabelUsingStatement
    Identifier : WellLabeledStatement
```

We retain the `[lookahead ∉ {{, function}]` restriction in *ExpressionStatement*. At the start of a statement, **{** can be the start of a block only, never an object literal.

```
PrimaryExpression:
    ...
    BlockExpression

BlockExpression:
    { UnlabeledStatementFirstList }
    { WellLabeledStatement StatementList? }

PropertyAssignment:
   IdentifierName : AssignmentExpression
   StringLiteral : AssignmentExpression
   NumericLiteral : AssignmentExpression
   get PropertyName ( ) { FunctionBody }
   set PropertyName ( PropertySetParameterList ) { FunctionBody }

PropertyName:
```

```
    IdentifierName
    StringLiteral
    NumericLiteral
```

Thus non-empty blocks may be used as expressions without ambiguity. An empty pair of braces **{}** other than at start of *Statement* is an *ObjectLiteral*. *PropertyAssignment* must inline-expand *PropertyName* to avoid an LR(1) shift-reduce conflict with *WellLabeledStatement*.

## Semantics

The refactored *Block* semantics are straightforward and backward-compatible. The semantics for *PrimaryExpression* : *BlockExpression* depend on block lambda revival: a non-empty block used as an expression is equivalent to a zero-parameter block-lambda.

## Compatibility

This proposal is mostly backward-compatible. In particular, the "stray label" problem whereby

```
javascript:foo()
```

migrates from URL contexts (links, `src` attribute values, the browser's address toolbar) into script content, but not at the start of a block, continues to work. Note that such a "label" is not used by the statement or expression to which it is affixed.

Useless labels are thus allowed other than at the start of a block (immediately after the **{** that starts the block).

A block in JS today, or a block-lambda if that extension is supported, may be prefixed by a label and actually use that label, e.g. via a `break` targeting that label. The grammar changes above support such a label-using block(-lambda).

— *Brendan Eich 2011/07/01 16:19*

strawman/block_vs_object_literal.txt · Last modified: 2011/07/06 04:47 by brendan

# [[strawman: block_lambda_revival]]

ES Wiki



Table of Contents


- Prologue
- Proposal
- Syntax
- Semantics
- See Also


## Prologue

As a mutually exclusive alternative to arrow function syntax, inspired by the long es-discuss thread Allen's lambda syntax proposal and echoes and followups since, here is a proposal for Tennent Correspondence Principle, AKA "Principle of Abstraction" full-strength blocks as better functions, both syntactically and semantically.

An essential part of this proposal is a paren-free call syntax, but only for calls bearing block arguments. A more general paren_free call syntax could perhaps be separated and considered on its own, but I'm specializing and combining proposals here to work through details holistically, and to emphasize that usability demands this syntax, based on Smalltalk, Ruby, and E expert witness testimony.

Also, general paren-free call syntax poses parsing and compatibility challenges that add (I believe) much more complexity than the paren-free block-argument-bearing call syntax proposed here.

— *Brendan Eich 2011/05/20 21:52*

## Proposal

```
let empty = {||};                          // empty block-lambda (note || not | |)

assert(empty() === undefined);
assert(typeof empty === "function");       // native and does implement [[Call]]
assert(empty.length === 0);

let identity = {|x| x};                    // reformed completion is return value

assert(identity(42) === 42);
assert(identity.length === 1);

let a = [1, 2, 3, 4];
let b = a.map {|e| e * e}                  // paren-free call with block is
                                           // idiomatic control structure so
                                           // no semicolon at end

print(b);                                  // [1, 4, 9, 16]

b = a.map {|e|                             // newline in block ok
           e * e * e}                      // newline after ends call

function find_first_odd(a) {
  a.forEach { |e, i|
              if (e & 1) return i; }       // return from find_first_odd
  return -1;
}
```

```
function escape_return() {
  return {|e| return e};
}
b = escape_return();
try { b(42); } catch (e) {}                    // error, return from inactive function

function find_odds_in_arrays(list,           // array of arrays
                             skip)           // if found, skip rest of current array
{
  let a = [];
  for (let i = 0; i < list.length; i++) {
    list[i].forEach {
      |e|
      if (e === skip) continue;              // continue the for loop
      if (e & 1) a.push(e);
    }
  }
  return a;
}

function find_more_odds(list, stop) {
  let a = [];
  for (let i = 0; i < list.length; i++) {
    list[i].forEach {
      |e|
      if (e === stop) break;                 // break from the for loop
      if (e & 1) a.push(e);
    }
  }
  return a;
}

function arguments_in_block() {
  return {|| arguments};
}
b = arguments_in_block("hi", "there");
a = b();

print(Array.prototype.join.call(a));        // "hi","there"

function this_in_block() {
  return {|| this};
}
let o = {m: this_in_block};
let b = o.m();
let t = b();

assert(t === o);

let p = {};
let u = b.call(p);

assert(u === o);

/* Another block-lambdas example, courtesy Claus Reinke. */
```

```
function A() {
  let ret = {|x| return x; };                  // Tennent "sequel"
  let inc = {|x| x+1 };
  let j = 0;
  while (true) {
    if (j > 3)
      ret(j);                                  // leave function A
    j = inc(j);
  }
}

/* A Smalltalk ifTrue:ifFalse: homage requested by Peter Michaux. */

Object.defineProperty(
  Boolean.prototype,
  'ifElse',
  {
    value: function (ifTrue, ifFalse) {
      return this ? ifTrue() : ifFalse();
    }
  }
);

print(false.ifElse {|| "true"} {|| "false"}); // "false"
print(true.ifElse {|| "true"} {|| "false"});  // "true"
```

## Syntax

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

```
ElementList :                                  // See 11.1.4
    Elision_opt InitialValue
    ElementList , Elision_opt 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

InitialValue :
    AssignmentExpression
    CallWithBlockArguments

Statement :
    ...
    CallWithBlockArguments
```

```
    LeftHandSideExpression = CallWithBlockArguments
    LeftHandSideExpression AssignmentOperator CallWithBlockArguments

MemberExpression :
    ...
    BlockLambda

PrimaryExpression :
    ...
    ( CallWithBlockArguments )

CallWithBlockArguments :
    CallExpression [no LineTerminator here] BlockArguments

BlockArguments :
    BlockLambda
    BlockArguments [no LineTerminator here] BlockLambda
    BlockArguments [no LineTerminator here] ( InitialValue )

BlockLambda :
    { || StatementList_opt }
    { | BlockParameterList_opt | StatementList_opt }

BlockParameterList :
    BlockParameter
    BlockParameterList , BlockParameter

BlockParameter :
    Identifier BlockParameterInitialiser_opt
    Pattern BlockParameterInitialiser_opt

BlockParameterInitialiser :
    = BitwiseXorExpression
```

Notes:

- The grammar above confines paren-free calls with block-lambda initial arguments and comma-free horizontal spaces only between arguments to certain contexts:
  - Initial values, in object and array initialisers, argument lists, default parameter values, and variable declaration initialiser.
  - Statements, either as the whole of an expression statement variant, or on the right of a left-hand side expression followed by = or an *AssignmentOperator*.
- The grammar changes show *Pattern* for destructuring, but I left out rest_parameters syntax for simplicity's sake.
- With GLR parsing for the spec grammar, we could consider, e.g. `(x) {x}` instead of `{|x| x}` with *LineTerminator*

excluded between parameters and body.

- But this syntax does not look like a block, and it may lead to missing newline errors between calls and blocks being parsed without error.
- So this is a cautionary tale: we do not want something that looks like a function expression more than a block, given TCP purity.

## Semantics

**8.6.2 Object Internal Properties and Methods**

**Table 9 – Internal Properties Only Defined for Some Objects**

| Internal Property | Value Type Domain | Description |
|---|---|---|
| `[[Call]]` | SpecOp(any, a List of any) → any or Reference or Completion | Executes code associated with the object. Invoked via a function call expression. The arguments to the SpecOp are this object and a list containing the arguments passed to the function call expression. Objects that implement this internal method are callable. Only callable objects that are host objects may return Reference values. Only block-lambda objects (11.1.7) may return Completion values (8.9) |

**11.1.7 Block Lambda**

The production *BlockLambda* **:** { | *BlockParameterList*$_{opt}$ | *StatementList*$_{opt}$ } is evaluated as follows:

1. Create a new native ECMAScript object and let *B* be that object.
2. Set all the internal methods of *B* as described in 8.12.
3. Set the `[[Class]]` internal property of *B* to "Function".
4. Set the `[[Prototype]]` internal property of *B* to the standard built-in Function prototype object as specified in 15.3.3.1.
5. Set the `[[Call]]` internal property of *B* as described in 11.1.7.1.
6. Set the `[[Scope]]` internal property of *B* to the LexicalEnvironment of the running execution context.
7. Set the `[[Context]]` internal property of *B* to the running execution context.
8. 

Let *names* be a List containing, in left to right textual order, the Strings corresponding to the identifiers of *BlockParameterList*. If no parameters are specified, let *names* be the empty list.

9. Set the `[[FormalParameters]]` internal property of *B* to *names*.

10. Set the `[[Code]]` internal property of *B* to $StatementList_{opt}$. If there is no *StatementList*, set `[[Code]]` to an *EmptyStatement*.

11. Set the `[[Extensible]]` internal property of *B* to true.

12. Let *len* be the number of formal parameters specified in *BlockParameterList*. If no parameters are specified, let *len* be 0.

13. Call the `[[DefineOwnProperty]]` internal method of *B* with arguments **"length"**, Property Descriptor {`[[Value]]`: *len*, `[[Writable]]`: **false**, `[[Enumerable]]`: **false**, `[[Configurable]]`: **false**}, and **false**.

14. Return the block-lambda object *B*.

Block-lambda parameter lists should support everything in Harmony that one can do in function formal parameter lists: parameter default values, rest parameters, and destructuring.

The *BlockParameterList*, *BlockParameter*, and *BlockParameterInitialiser* productions have straightforward semantics, elided here mainly because I don't want to duplicate all the existing *FormalParameterList*, etc. semantics.

#### 11.1.7.1 [[Call]]

When the `[[Call]]` internal method for a block-lambda object *B* is called with 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 *B*'s `[[FormalParameters]]` internal property, the passed arguments List *args*, and the **this** value given by the ThisBinding component of the execution context in *B*'s `[[Context]]` internal property.

2. Let *result* be the result of evaluating the *StatementList* or *EmptyStatement* that is the value of *B*'s `[[Code]]` internal property.

3. Exit the execution context *funcCtx*, restoring the previous execution context.

4. If *result*.type is **normal** and *result*.value is **empty** then return (**normal**, **undefined**, **empty**).

5.

Else if *result*.type is **break**, **continue**, or **return** and the execution context in *B*'s `[[Context]]` internal property has already exited, throw a **TypeError** exception.

6\.

Else if *result*.type is **break** or **continue** and evaluation of the label-set *result*.target has already completed, throw a **TypeError** exception.

7\.

Else return the Completion value *result*.

TODO: forbid `var` in block-lambdas via early error (yeah!)

**11.2.3 Function Calls**

*…*

TODO: Process Completion return type

The production *CallWithBlockArguments* **:** *CallExpression* `[no` *LineTerminator* `here]` *BlockArguments* is evaluated in exactly the same manner, except that *BlockArguments* is evaluated instead of *Arguments* in step 3.

(This paragraph goes just before the NOTE at the bottom of 11.2.3.)

**11.2.4 Argument Lists**

*…*

The production *BlockArguments* **:** *BlockLambda* is evaluated as follows:

1\.

Let *ref* be the result of evaluating *BlockLambda*.

2\.

Let *arg* be GetValue(*ref*).

3\.

Return a List whose sole item is *arg*.

The production *BlockArguments* **:** *BlockArguments* `[no` *LineTerminator* `here]` *BlockLambda* is evaluated as follows:

1\.

Let *precedingArgs* be the result of evaluating *BlockArguments*.

2\.

Let *ref* be the result of evaluating *BlockLambda*.

3\.

Let *arg* be GetValue(*ref*).

4\.

Return a List whose length is one greater than the length of *precedingArgs* and whose items are the items of *precedingArgs*, in order, followed at the end by *arg* which is the last item of the new list.

The production *BlockArguments* **:** *BlockArguments* `[no` *LineTerminator* `here]` `(` *InitialValue* `)` is evaluated in exactly the same manner, except that *InitialValue* is evaluated instead of *BlockLambda* in step 2.

#### 12.7 The `continue` Statement

The verbiage about "(but not crossing function boundaries)" should be clarified to exclude block-lambda boundaries, which are treated like *Block* boundaries.

#### 12.8 The `break` Statement

The verbiage about "(but not crossing function boundaries)" should be clarified to exclude block-lambda boundaries, which are treated like *Block* boundaries.

#### 12.9 The `return` Statement

The verbiage about "A return statement causes a function to cease execution and return a value to the caller" should be clarified to include `return` in block-lambda attempting to exit the nearest enclosing *FunctionBody*.

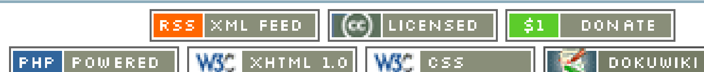
Notes:

- completion reform should be considered since the completion value is the return value for block lambdas.
- Existing semantic early error checks on label use, `break` outside of `switch` and loop, etc. work as before, but cross the block-lambda boundary.
- The runtime semantics for block-lambda `[[Call]]` must handle `return` from an outer function activation that has already returned.

## See Also

- arrow function syntax, an alternative considering only syntax, without new semantics
- Allen's "Comments on Smalltalk block closure designs, part 1"





# [[strawman: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, must use 'return' explicitly if not void
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(! "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 intialiser 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

Extend *AssignmentExpression* and define *ArrowFunctionExpression*:

```
AssignmentExpression :
    ArrowFunctionExpression
    ...

ArrowFunctionExpression :
    ArrowFormalParameters_opt Arrow AssignmentExpression
    ArrowFormalParameters_opt Arrow ArrowBodyBlock_opt

ArrowFormalParameters :
    ( FormalParameterList_opt )
    ( this Initialiser_opt )
    ( this Initialiser_opt , FormalParameterList )

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

The *ArrowFormalParameters* production requires GLR parsing or equivalent to disambiguate against the other right-hand sides of *AssignmentExpression*. For an LR(1) grammar, we can use:

```
ArrowFormalParameters :
    ( Expression_opt )
    ( this Initialiser_opt )
    ( this Initialiser_opt , Expression )
```

and require that *Expression* reductions on the right-hand sides match *FormalParameterList*.

This works because *Expression* is a cover grammar for *FormalParameterList*, with *Identifier* primary expressions covering formal parameter names, array and object literals for destructuring, assignment for parameter default values, and spread for rest parameters.

This cover grammar approach may be future-hostile without, e.g., extending guards to be legal in expressions.

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

```
ArrowBodyBlock :
    { [lookahead(2) ∉ { Identifier ":", "get" Identifier, "set" Identifier }]
StatementList }
```

where `lookahead(2)` peeks ahead two tokens, correctly lexing / after *Identifier* as a division operator. *ArrowBodyBlock* is a non-empty block not starting with a label, to parse {} as an empty object initialiser produced from the *AssignmentExpression* body form. The concise object literal extensions proposal (exact still being hammered out) wants further lookahead restrictions.

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)
- -> 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
  - 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*

## See Also

block lambda revival, an alternative adding new semantics, not only new syntax

strawman/arrow_function_syntax.txt · Last modified: 2011/06/05 18:18 by brendan

# [[strawman: paren_free]]

ES Wiki

Trace: » object_initialiser_super » block_vs_object_literal » block_lambda_revival » arrow_function_syntax » paren_free

## Motivation

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
    if ( Expression ) OtherStatement else Statement
    if ( Expression ) OtherStatement
```

Where *SubStatement* is

```
SubStatement :
    Block
    KeywordStatement
```

and *KeywordStatement* is

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

This leaves

```
OtherStatement :
    EmptyStatement
    ExpressionStatement
    VariableStatement
    LabelledStatement
```

and

```
Statement :
    Block
    KeywordStatement
    OtherStatement
```

The same pattern applied to *IfStatement* above applies to *IterationStatement*, *SwitchStatement*, and `catch` clauses in *TryStatement* – except `catch` blocks must still be braced (as with `try` and `finally` since their introduction in ES3), so no *OtherStatement* `catch` body production.

TODO: expand these all into a complete sub-grammar.

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 (idiomatic since K&R C and unproblematic as far as dangling-else goes) 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`.

Note that the paren-free `if-else` production requires the `else` clause to be a *SubStatement*. You cannot write `if x > y { alert("win"); } else alert("lose")`.

For backward compatibility, we support parenthesized heads with any sub-statement. To preserve the LR(1) grammar this requires factoring out *OtherStatement*.

Thus, this proposal is intended to make no backward-incompatible syntactic or semantic changes to ES5. "Paren-free" is now purely a relaxation of syntax rules.

— Brendan Eich *2011/06/05 21:06*



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

[[harmony:classes]] ES Wiki

Trace: » block_vs_object_literal » block_lambda_revival » arrow_function_syntax » paren_free » classes

# Classes

Table of Contents


- Classes
  - Motivation
  - The Proposal in a Nutshell
    - Class Body
    - Member Modifiers
  - The Proposal In Full
    - Class Declarations and Expressions
      - Grammar
    - Class Adjective
      - const
      - Grammar
    - Class Members
      - Grammar
  - Refinements
    - Inheritance
      - Grammar
    - Constructor Chaining
      - Grammar
    - Member Delegation
      - Grammar
    - Private Instance Members
      - Grammar
      - Semantics
  - Proposal History
  - Open Issues
  - See


## Motivation

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

For example, here is code from three.js (simplified and modified slightly), with comments for the intent behind each line.

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

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

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

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

With class syntax, this becomes:

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

    public identityMatrix = new THREE.Matrix4();
    public bones = [];
    public boneMatrices = [];
```

```
    ...
  }

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

## The Proposal in a Nutshell

Before we lay out the detailed grammar and semantics, we'll show the core concepts by example. If there are places where this section disagrees with later parts of the proposal, those parts take precedence.

### Class Body

A class defines four objects and their properties: a constructor function, a prototype, a new instance, and a private record bound to the new instance. The body of a class is a collection of member definitions. This example shows each of the kinds of members that can be defined:

```
class Monster {
  // The contextual keyword "constructor" followed by an argument
  // list and a body defines the body of the class's constructor
  // function. public and private declarations in the constructor
  // declare and initialize per-instance properties. Assignments
  // such as this.foo = bar; set public properties.
  constructor(name, health) {
    public name = name;
    private health = health;
  }

  // An identifier followed by an argument list and body defines a
  // method. A "method" here is simply a function property on some
  // object.
  attack(target) {
    log('The monster attacks ' + target);
  }

  // The contextual keyword "get" followed by an identifier and
  // a curly body defines a getter in the same way that "get"
  // defines one in an object literal.
  get isAlive() {
    return private(this).health > 0;
  }

  // Likewise, "set" can be used to define setters.
  set health(value) {
    if (value < 0) {
      throw new Error('Health must be non-negative.')
    }
    private(this).health = value
  }

  // An identifier optionally followed by "=" and an expression
```

```
  // declares a prototype property and initializes it to the value
  // of that expression. "public" as a member modifier is allowed.
  numAttacks = 0;

  // The keyword "const" followed by an identifier and an
  // initializer declares a constant prototype property.
  const attackMessage = 'The monster hits you!';
}
```

### Member Modifiers

Since a class body defines properties on *two* objects, syntax is needed to indicate on which object, constructor or prototype, the member becomes a property. Keyword prefixes are used:

```
class Monster {
  // "static" places the property on the constructor.
  static allMonsters = [];

  // "public" declares on the prototype.
  public numAttacks = 0;
}
```

## The Proposal In Full

### Class Declarations and Expressions

A class is both a blueprint for describing instances and a factory to create them. Like functions, a class can either be a *declaration* or an *expression*. Both define a constructor function to represent that class. We'll refer to that function as a "class". We'll use "class definition" to refer to either a class declaration or expression when the distinction doesn't matter.

Like a function declaration, a class *declaration* defines a variable with the class's name whose declaration is hoisted to the beginning of the surrounding scope. This supports mutual recursion among function and class declarations. Like a function expression, a class *expression* 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.

When a scope (*Block*, *FunctionBody*, *Program*, etc.) is entered, the variables declared by all immediately contained function and class declarations are bound to their respective functions and classes. Then all class bodies are executed in textual order. A class body defines and initializes class-wide properties once when the class definition is evaluated. This includes properties on the constructor function (the "class" itself) and on its prototype property. These initializations happen in textual order.

#### Grammar

We extend the Declaration production from block scoped bindings to accept a *ClassDeclaration*. We extend MemberExpression to accept a *ClassExpression*.

```
Declaration :
    ClassDeclaration
    ...
ClassDeclaration :
    class Identifier { ClassBody }

MemberExpression :
    ClassExpression
```

```
    ...
ClassExpression :
    class Identifier? { ClassBody }

ExpressionStatement :
    [lookahead ∉ { "{", "function", "class" }] Expression ;

ClassBody :
    ClassElement*

// "..." means existing members defined elsewhere
```

## Class Adjective

A class definition may be prefixed with an adjective to clarify its role. Currently, the only adjective proposed is `const`, but this set may expand.

### const

A `const` class provides high integrity. Both the constructor function and prototype object are frozen and the variable the class is bound to is const (non-assignable). Instances of the class are sealed.

In other words, given this definition:

```
const class Empty { }
```

The variable `Empty` is const, the constructor function it references is frozen, and `Empty.prototype` is frozen. Calling `new Empty()` returns a sealed object.

### Grammar

We revise the previous grammar to allow adjectives before `class`.

```
ClassDeclaration :
    ClassAdjective* class Identifier { ClassBody }

ClassExpression :
    ClassAdjective* class Identifier? { ClassBody }

ClassAdjective :
    const

ExpressionStatement :
    [lookahead ∉ { "{", "function", "class", ...ClassAdjective }]  Expression ;
```

## Class Members

The body of a class definition is a collection of members each of which becomes a property on one of the objects associated with the class. By default, data properties define enumerable prototype properties while method members define non-enumerable prototype properties. Members of non-const classes default to writable and configurable. Member adjectives, if present, override the default attributes of the property being defined.

A class body may contain one constructor, whose body is the code run to initialize instances of the class. This constructor code provides the behavior of the class’s internal `[[Call]]` and `[[Construct]]` methods.

### Grammar

```
ClassElement :
    Constructor
    PrototypePropertyDefinition
    ClassPropertyDefinition

Constructor :
    constructor ( FormalParameterList? ) { ConstructorBody }

ConstructorBody :
    ConstructorElement*

ConstructorElement :
    Statement
    Declaration
    InstancePropertyDefinition

PrototypePropertyDefinition :
    ExportableDefinition
    public ExportableDefinition

ClassPropertyDefinition :
    static ExportableDefinition

InstancePropertyDefinition :
    public ExportableDefinition

ExportableDefinition :
    Declaration
    Identifier = Expression ;                            // data property
    Identifier ( FormalParameterList? ) { FunctionBody } // method
    get Identifier ( ) { FuntionBody }                   // getter
    set Identifier ( FormalParameter ) { FunctionBody }  // setter
    MemberAdjective ExportableDefinition

MemberAdjective :
    // attribute control
```

## Refinements

We refine the above syntax with additional features until we reach a complete, usable class proposal.

### Inheritance

We extend this class syntax to allow users to *declaratively* specify the prototypal inheritance they can already express *imperatively*. There are two forms: `extends` and `prototype`, each followed by an expression. When this class's `prototype` object is created, either of those clauses will be used to determine which object it inherits from. The expression following `extends` or `prototype` is evaluated. Then, if `extends` is used, the `prototype` property of that object will be used. If `prototype` is used, the object itself will be. If neither clause is given, the class's `prototype` will inherit from `Object.prototype`.

By example:

```
class Base {}
class Derived extends Base {}
```

Here, `Derived.prototype` will inherit from `Base.prototype`.

```
let parent = {};
class Derived prototype parent {}
```

Here, `Derived.prototype` will inherit directly from `parent`.

### Grammar

We revise the previous grammar to allow these inheritance clauses.

```
ClassDeclaration :
    ClassAdjective* class Identifier Heritage? { ClassBody }

ClassExpression :
    ClassAdjective* class Identifier? Heritage? { ClassBody }

Heritage :
    extends MemberExpression
    prototype MemberExpression
```

## Constructor Chaining

Building on inheritance, we provide a cleaner syntax for invoking the parent constructor in classes defined with an `extends` clause. Within the body of the constructor, an expression `super(x, y)` calls the superclass's `[[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`. A call like this may appear anywhere within the constructor body, excluding nested functions and classes.

These semantics for constructor chaining preclude 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.

### Grammar

To enable this, we add a production to CallExpression.

```
CallExpression :
    ...
    super Arguments
```

with a post-parsing early error if this production occurs outside a ConstructorBody.

## Member Delegation

Similar to constructor chaining, we extend `super` to allow delegating to any inherited member when called from within the body of a class. Within the class `Derived`, the expression `super.member` evaluates to a Reference with *base* `this` and *referenced name* `member`, but whose `[[GetValue]]` will look up `Derived.prototype.`

`[[Prototype]].member`. In other words, `super.member(x, y)` acts like `Derived.prototype.[[Prototype]].member.call(this, x, y)`, as if using the original binding of `Function.prototype.call`. The `super.member` expression may not be used as a LeftHandSideExpression.

### Grammar

To enable this, we add a new production to MemberExpression.

```
MemberExpression :
    super . IdentifierName
    ...
```

with a post-parsing early error if this occurs outside a class.

## Private Instance Members

A requirement of this proposal is the ability to define private per-object state that meets the following requirements:

- A usable syntax for defining and accessing private instance state from within methods of the class.
- Information hiding to encourage decoupling for software engineering concerns.
- Strong encapsulation in order to support defensiveness and security.
- An efficient implementation. The private state should be allocated with the instance as part of a single allocation, and with no undue burden on the garbage collector.
- The ability to have private mutable state on publicly frozen objects.

Now that the private name objects has been accepted, a pattern composing private names with classes can satisfy the above requirements.

### Grammar

To enable this, we add a new production to *CallExpression* as a special form to retrieve the private variable record:

```
CallExpression :
    ...
    private ( AssignmentExpression )

ConstructorElement :
    ...
    PrivateVariableDefinition

PrivateVariableDefinition :
    private ExportableDefinition
```

### Semantics

The production *PrivateVariableDeclaration* `:` `private` *ExportableDefinition* is evaluated roughly as follows:

1. If **this** object does not have a private variable record, create one.
2. Let *privRec* = the private variable record of the **this** object.
3. Let *env* = NewObjectEnvironment(*privRec*, **`null`**).
4. Evaluate *ExportableDefinition* using *env* to bind a property in the private variable record.

More work is needed on this semantics, but a couple of intentional design points should be apparent:

- the private variable record cannot be accessed by the programmer, therefore it can be fused with the instance allocation.
- `Object.freeze` on the instance of the class does not freeze the private variable record.
- You cannot use the same name in the same class for a public instance property and private instance variable.

## Proposal History

This is a fork of classes_with_trait_composition that revives the idea of declarative public property and private variable syntax within the constructor body.

That strawman 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.

## Open Issues

- What are the semantics of a `return` within a constructor body?
- Should we avoid `private` and possibly `public` proliferation by allowing either:
  - `private:` sections a la C++;
  -

`private foo = 42, bar = "hi";` – i.e., multiple definitions after the prefix keyword. [RESOLVED to this, update and remove]

- One or two namespaces for public properties and private instance variables [RESOLVED two, Mark's argument]
- `numAttacks = 0;` without prefix `public` or `proto` or something looks too much like an assignment (misplaced) [RESOLVED require `public` prefix]
- Should `private` prototype properties based on private name objects be supported? [RESOLVED via `private` prefix]
- `private(this)`, e.g., is
  - unbearably verbose;
  - leaks an implementation detail.
- Need concise attribute controls. [RESOLVED same as revised Allen basic object literal extensions proposal]
- Want accessor "half-override", e.g. `get super set x`.... [RESOLVED same as previous, see Allen's proposal]
- Are static methods inherited with `this` bound to the class receiver? See @wycats' CoffeeScript/Ruby example

## See

- Traceur Classes and Traits
- Encapsulation and Inheritance in Object-Oriented Programming Languages: classic 1986 paper by Alan Snyder.
- Classes as Sugar thread which starts with pointers to earlier threads.
- Harmonious Classes: write-up of discussion to unify this and obj_initialiser_class_abstraction proposals.

Related and historical strawmen

- object_initialiser_extensions, especially obj_initialiser_class_abstraction and instance_variables
- classes as sugar

- classes as inheritance sugar (not yet ready)
- trait_composition_for_classes
- classes_with_trait_composition

harmony/classes.txt · Last modified: 2011/06/09 01:17 by markm

# [[harmony: generators]]

ES Wiki



Table of Contents


- Overview
- Examples
- API
  - Generator objects
- Syntax
- Generator functions
  - Calling
  - Yielding
  - Delegating yield
  - Returning
- Generator methods
  - Method: next
  - Method: send
  - Method: throw
  - Method: close
- Generator objects
  - States
  - Internal method: send
  - Internal method: throw
  - Internal method: close
  - Resuming generators
- References


## Overview

First-class coroutines, represented as objects encapsulating suspended execution contexts (i.e., function activations). Prior art: Python, Icon, Lua, Scheme, Smalltalk.

## Examples

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

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

Generators can be iterated over in loops:

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

Generators are iterators:

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

## API

See the "@iter" module in iterators.

## Generator objects

Every generator object has the following internal properties:

- [[Prototype]] : the original value of `Object.prototype`
- [[Code]] : the code for the generator function body
- [[ExecutionContext]] : either **null** or an execution context
- [[Scope]] : the scope chain for the suspended execution context
- [[Handler]] : a standard generator handler for performing iteration
- [[State]] : "newborn", "executing", "suspended", or "closed"
- [[Send]] : see semantics below
- [[Throw]] : see semantics below
- [[Close]] : see semantics below

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

# Syntax

The function syntax is extended to add an optional * token:

```
FunctionDeclaration:
    "function" "*"? Identifier "(" FormalParameterList? ")" "{" FunctionBody "}"

FunctionExpression:
    "function" "*"? Identifier? "(" FormalParameterList? ")" "{" FunctionBody "}"
```

A function with a * token is known as a *generator function*. The following two unary operators are only allowed in the immediate body of a generator function (i.e., in the body but not nested inside another function):

```
AssignmentExpression:
```

```
    ...
    YieldExpression

YieldExpression:
    "yield" "*"? AssignmentExpression
```

An early error is raised if a `yield` or `yield*` expression occurs in a non-generator function.

# Generator functions

This section describes the semantics of generator functions.

## Calling

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

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

## Yielding

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

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

## Delegating yield

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

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

```
let (g = <<expr>>) {
    let received = void 0, send = true, result = void 0;
    try {
        while (true) {
            let next = send ? g.send(received) : g.throw(received);
            try {
                received = yield next;
                send = true;
```

```
            } catch (e) {
                received = e;
                send = false;
            }
        }
    } catch (e) {
        if (!isStopIteration(e))
            throw e;
        result = e.value;
    } finally {
        try { g.close(); } catch (ignored) { }
    }
    result
}
```

This is similar to a `for-in` loop over the generator, except that it propagates exceptions thrown via the outer generator's `throw` method into the delegated generator.

## Returning

The semantics of `return e` inside a generator function is:

Let V ?= Evaluate(e)
Let K = the current execution context
Let O = K.currentGenerator
O.[[State]] := "closed"
Let R = a new object with
  [[Class]] = "StopIteration"
R.value := V
Throw R

See iterators for a discussion of `StopIteration`.

As in ordinary functions, `return;` is equivalent to `return (void 0);`, and if control falls off the end of a generator function body, the generator function performs an implicit `return;`.

# Generator methods

## Method: next

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

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

## Method: send

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

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

## Method: throw

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

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

## Method: close

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

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

# Generator objects

## States

A generator object can be in one of four states:

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

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

## Internal method: send

G.[[Send]]

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

G.[[State]] := "executing"
Let Result = *Resume*(G.[[ExecutionContext]], **normal**, X)
Return Result

## Internal method: throw

G.[[Throw]]

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

## Internal method: close

The **close** method terminates a suspended generator. This informs the generator to resume roughly as if via `return`, running any active `finally` blocks first before completing.

G.[[Close]]

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

## Resuming generators

(This operation assumes that we re-specify expressions to have completion types just like statements.)

**Operation** *Resume*(K, completionType, V)

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

# References


- Generators in SpiderMonkey
- PEP 255, "Simple generators"

- PEP 380, “Syntax for delegating to a sub-generator”
- PEP 3152, “Cofunctions”

harmony/generators.txt · Last modified: 2011/07/20 23:15 by dherman

Table of Contents


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


# Binary data

See also:

- binary data semantics
- binary data discussion

## Goals

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

Desiderata:

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

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

### Examples

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

const Triangle = new ArrayType(Pixel, 3);

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

*TODO*: more examples

## Blocks: compact binary data

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

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

### Block types

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

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

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

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

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

### Block objects

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

block types.

## Numeric data

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

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

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

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

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

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

## Arrays

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

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

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

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

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

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

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

## Structs

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

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

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

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

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

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

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

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

## Block references

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

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

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

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

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

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

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

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



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