Ecma/TC39/2010/031

Overview

The `for-in` loop is ECMAScript's convenient mechanism for doing iteration, but only works via the built-in object property enumeration semantics. This proposal generalizes the behavior of `for-in` to obey user-specified iteration protocols via an iteration hook. This generalization allows the use of `for-in` loops for custom iterations, providing concise, readable traversals of user-specified data structures.

Examples

Conveniently iterate over keys, values, or both:

```
for (var key in Iterator.keys(x)) {
    alert(key)
}

for (var val in Iterator.values(x)) {
    alert(val)
}

for (var [key, val] in Iterator.properties(x)) {
    alert("x." + key + " = " + val);
}
```

A custom collection pseudoclass whose instances are iterable via `for-in`:

```
function MyCollection() {
    this.elements = [];

    // make every instance of MyCollection iterable
    var self = this;
    return Iterator.for(self, function() {
        return self.iterator()
    });
}

MyCollection.prototype = {
    iterator: function() {
        var collection = this;
        return {
            index: 0,
            next: function() {
                if (this.index >= collection.elements.length)
                    throw StopIteration;
                return collection.elements[this.index++]
            }
        }
    },
    add: function(x) { ... },
    remove: function(x) { ... },
    ...
}
```

Iteration

The proposed-ES4 mechanism for iterators and generators allowed objects to implement a specially named iteration hook. However, a good goal for an iteration protocol is to support stratification, as in the proxies facility. This proposal avoids the special names by extending the proxies mechanism.

Iterator objects

In the terminology of this proposal, an *iterator* is any object of the following form:

```
{ next: function() -> any }
```

Iteration via proxies

This proposal makes a small change to the proxies API by adding the following derived trap:

```
iterate: function () -> iterator
```

The `for-in` loop uses this trap to provide the values to iterate over, and defaults to the `enumerate` trap in the case where the `iterate` trap is not defined. The complete semantics is given below.

StopIteration

There is a special "StopIteration" [[Class]] recognized by the `for-in` loop semantics. When the `next` method of the `iterate` trap throws an exception, if the exception is an object whose [[Class]] is "StopIteration", the loop terminates.

There is a standard variable `StopIteration` that is bound to an object of [[Class]] "StopIteration".

Iteration library

While proxies provide the most general way of creating `for-in`-enabled iterator objects, there should also be more convenient forms for the common cases.

```
Iterator.create:     function(trap: function() -> iterator) -> iterator
Iterator.for:        function(x: Object, trap: function() -> iterator) -> iterator
Iterator.keys:       function(x: Object, own: boolean = false) -> iterator
Iterator.values:     function(x: Object, own: boolean = false) -> iterator
Iterator.properties: function(x: Object, own: boolean = false) -> iterator
```

- `Iterator.create`: Creates a new iterator proxy using the given trap.
- `Iterator.for`: Creates a new iterator proxy using the given trap for the `iterate` trap and otherwise behaving as a forwarding proxy.
- `Iterator.keys`: Creates a new iterator proxy that iterates over the property names of the object `x`. If `own` is provided and is a truthy value, then the iterator only produces the names of the object's own properties.
- `Iterator.values`: Creates a new iterator proxy that iterates over the property values of the object `x`. If `own` is provided and is a truthy value, then the iterator only produces the values of the object's own properties.
- `Iterator.properties`: Creates a new iterator proxy that iterates over the properties of the object `x`, providing at each iteration a two-element array with the property name at index 0 and the property value at index 1. If `own` is provided and is a truthy value, then the

iterator only produces the object's own properties.

Iteration semantics

Note: assignments containing a question mark are eliding a simple error propagation, i.e.:

```
Let x ?= e
```

and

```
x :?= e
```

are shorthand for:

```
Let x = e
If IsError(x) Return x
```

and

```
x := e
If IsError(x) Return x
```

respectively.

Evaluation of for-in loops

Operation *Eval*(IterationStatement → for (LHSExpression in Expression) Statement)

Return *ForInLoop*(LHSExpression, Expression, Statement, IterationStatement.labels)

Operation *Eval*(IterationStatement → for (VarDeclarationNoIn in Expression) Statement)

```
Let varName = Eval(VarDeclarationNoIn)
Let var = IdentifierReference(varName)
Return ForInLoop(var, Expression, Statement, IterationStatement.labels)
```

For-in loop bodies

The body of a `for-in` loop attempts to use the `iterate` trap but falls back to property enumeration. See the enumeration proposal for the definition of *EnumerateProperties*.

Operation *ForInLoop*(LHS, Expression, Statement, labels)

```
Let expr ?= Eval(Expression)
Let obj = ToObject(expr.value)
Let elements
If IsTrappingProxy(obj)
    Let handler = obj.[[Handler]]
    Let iterate ?= handler.[[Get]]("iterate")
    If iterate.value = undefined
```

```
        Let enum ?= handler.[[Get]](”enumerate”)
        If !IsCallable(enum.value)
            Return (type=error, value=TypeError, target=empty)
        Let names ?= enum.value.[[Call]](handler, [])
        elements := GetArrayElements(names.value)
    Else
        If !IsCallable(iterate)
            Return (type=error, value=TypeError, target=empty)
        elements :?= iterate.[[Call]](handler, [])
        If !IsObject(elements)
            Return (type=error, value=TypeError, target=empty)
Else
    elements := EnumerateProperties(obj.[[Prototype]])
Let V = empty
Repeat
    Let P
    If IsObject(elements)
        Let next ?= elements.[[Get]](”next”)
        Let R = next.value.[[Call]](elements, [])
        If IsError(R)
            If IsObject(R.value) && R.value.[[Class]] = “StopIteration”
                Return (type=normal, value=V, target=empty)
            Return R
        P := R.value
    Else if elements = []
        Return (type=normal, value=V, target=empty)
    Else
        P := elements[0]
        elements := [ elements[1], ... ]
    Let lhsRef ?= Eval(LHS)
    Let put ?= PutValue(lhsRef, P)
    Let stmt ?= Eval(Statement)
    If stmt.value != empty
        V := stmt.value
    If stmt.type = break
        If stmt.targets ∈ labels
            Return (type=normal, value=V, target=empty)
        Return stmt
    If stmt.type = continue && stmt.target ∉ labels
        Return stmt
```

Iteration sequences

Operation *GetArrayElements*(array)

```
Let lenProp ?= array.[[Get]](”length”)
Let length = ToUInt32(lenProp)
Let elements = []
```

```
For each i in 0 ... length - 1
    Let P = ToString(i)
    Let V ?= array.[[Get]](P)
    elements := [ elements, ..., V.value ]
Return elements
```