Draft Status: Rough

2008-05-18, pratapL, creation

2008-05-29, pratapL, added a 'gotcha' related to interactions with strict mode from MarkM.

Rationale for ES3.1 Proposal

The Array extras added as part of JavaScript 1.6 have proved popular. Most Ajax libraries implement them. Implementations are available Dogo (html links for indexOf, lastIndexOf, every, some, forEach, map, filter) , Prototype (Enumerable now has aliases for equivalent JavaScript 1.6 Array methods), Base2 (html), Qooxdoo (html), etc. Presently, both FireFox and Safari support these Array extras natively.

ES3.1 proposes to codify the following Array "generics": indexOf, lastIndexOf, every, some, forEach, map, filter.

There are several methods on ES3 objects that are referred to as "generic" methods. "Generic" here means that the method's this value need not be an Array object. Therefore, it can be transferred to other kinds of objects. ES3 provides such methods on the respective objects' prototypes. If we want to add more such generic methods, where should they be added? Similar to the rationale used for the extensions to String object, these generics too will be added to **`Array.prototype`**.

Proposed ES4 Proposal

The proposed ES4 proposal (html) adds a static method of the same name for each prototype generic method.

ES3.1 Proposal

The goals of the ES3.1 proposal are to ensure the following:

- Compatible with ES3
- Compatible with proposed ES4

The iterative "Array generics" added in JavaScript 1.6 take a "callback" and an optional second argument that specifies a "thisObject". If a "thisObject" parameter is provided, it is used as the "this" for each invocation of the "callback". If it is not provided, or is null, the global object associated with "callback" is used instead.

In the case of ES3.1, these methods will not take this optional second argument, and the "callback" will always be called as a function. There are three other ES3.1 proposals that have led to this line of thinking:

(1) strict mode behaviour
(2) this binding for functions (and how it interacts with strict mode)
(3) introducing a "bind" property on Function.

In the case of Array generics, if the callback is defined within a non-strict module, the behavior is as in ES3; in a strict module an attempt to access "this" will throw an exception - this is compatible with proposed ES4 behaviour (when 'this' is bound to null or undefined).

Comment [pL1]: Gotcha from Mark Miller: The assumption that the callback function takes three arguments is meant to be friendly to those who define a callback function with too few arguments; the extra arguments are simply ignored. That's why the arguments are ordered strangely -- so that the ones most likely to be needed appear first.

The conflict is with strict mode, where we've been saying that calls to strict functions with too many arguments are rejected by default, and only accepted if the function is explicitly variable arity. This will be a conflict for ES4 strict as well.

15.4 Array Objects

Array objects give special treatment to a certain class of property names. A property name *P* (in the form of a string value) is an *array index* if and only if ToString(ToUint32(*P*)) is equal to *P* and ToUint32(*P*) is not equal to $2^{32}-1$. Every Array object has a **`length`** property whose value is always a nonnegative integer less than 2^{32}. The value of the **`length`** property is numerically greater than the name of every property whose name is an array index; whenever a property of an Array object is created or changed, other properties are adjusted as necessary to maintain this invariant. Specifically, whenever a property is added whose name is an array index, the **`length`** property is changed, if necessary, to be one more than the numeric value of that array index; and whenever the **`length`** property is changed, every property whose name is an array index whose value is not smaller than the new length is automatically deleted. This constraint applies only to properties of the Array object itself and is unaffected by **`length`** or array index properties that may be inherited from its prototype.

15.4.1 The Array Constructor Called as a Function

When **`Array`** is called as a function rather than as a constructor, it creates and initialises a new Array object. Thus the function call **`Array(`**…**`)`** is equivalent to the object creation expression **`new Array(`**…**`)`** with the same arguments.

15.4.1.1 Array ([item1 [, item2 [, …]]])

When the **`Array`** function is called the following steps are taken:

1. Create and return a new Array object exactly as if the array constructor had been called with the same arguments (15.4.2).

15.4.2 The Array Constructor

When **`Array`** is called as part of a **`new`** expression, it is a constructor: it initialises the newly created object.

15.4.2.1 new Array ([item0 [, item1 [, …]]])

This description applies if and only if the Array constructor is given no arguments or at least two arguments.

The [[Prototype]] property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of **`Array.prototype`** (15.4.3.1).

The [[Class]] property of the newly constructed object is set to **`"Array"`**.

The **`length`** property of the newly constructed object is set to the number of arguments.

The **`0`** property of the newly constructed object is set to *item0* (if supplied); the **`1`** property of the newly constructed object is set to *item1* (if supplied); and, in general, for as many arguments as there are, the *k* property of the newly constructed object is set to argument *k*, where the first argument is considered to be argument number **`0`**.

15.4.2.2 new Array (len)

The [[Prototype]] property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of **`Array.prototype`** (15.4.3.1). The [[Class]] property of the newly constructed object is set to **`"Array"`**.

If the argument *len* is a Number and ToUint32(*len*) is equal to *len*, then the **`length`** property of the newly constructed object is set to ToUint32(*len*). If the argument *len* is a Number and ToUint32(*len*) is not equal to *len*, a **RangeError** exception is thrown.

If the argument *len* is not a Number, then the **`length`** property of the newly constructed object is set to **`1`** and the **`0`** property of the newly constructed object is set to *len*.

15.4.3 Properties of the Array Constructor

The value of the internal [[Prototype]] property of the Array constructor is the Function prototype object (15.3.4).

Besides the internal properties and the **`length`** property (whose value is **1**), the Array constructor has the following properties:

15.4.3.1 Array.prototype

The initial value of **`Array.prototype`** is the Array prototype object (15.4.4).

This property has the attributes { DontEnum, DontDelete, ReadOnly }.

15.4.4 Properties of the Array Prototype Object

The value of the internal [[Prototype]] property of the Array prototype object is the Object prototype object (15.2.3.1).

The Array prototype object is itself an array; its [[Class]] is **`"Array"`**, and it has a **`length`** property (whose initial value is **+0**) and the special internal [[Put]] method described in 15.2.3.1.

In following descriptions of functions that are properties of the Array prototype object, the phrase "this object" refers to the object that is the **this** value for the invocation of the function. It is permitted for the **this** to be an object for which the value of the internal [[Class]] property is not **"Array"**.

NOTE
The Array prototype object does not have a ***valueOf*** *property of its own; however, it inherits the* ***valueOf*** *property from the Object prototype Object.*

15.4.4.1 Array.prototype.constructor

The initial value of **Array.prototype.constructor** is the built-in **Array** constructor.

15.4.4.2 Array.prototype.toString ( )

The result of calling this function is the same as if the built-in **join** method were invoked for this object with no argument.

The **toString** function is not generic; it throws a **TypeError** exception if its **this** value is not an Array object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.4.4.3 Array.prototype.toLocaleString ( )

The elements of the array are converted to strings using their **toLocaleString** methods, and these strings are then concatenated, separated by occurrences of a separator string that has been derived in an implementation-defined locale-specific way. The result of calling this function is intended to be analogous to the result of **toString**, except that the result of this function is intended to be locale-specific.

The result is calculated as follows:

1. Call the [[Get]] method of this object with argument **"length"**.
2. Call ToUint32(Result(1)).
3. Let *separator* be the list-separator string appropriate for the host environment's current locale (this is derived in an implementation-defined way).
4. Call ToString(*separator*).
5. If Result(2) is zero, return the empty string.
6. Call the [[Get]] method of this object with argument **"0"**.
7. If Result(6) is **undefined** or **null**, use the empty string; otherwise, call ToObject(Result(6)).toLocaleString().
8. Let *R* be Result(7).
9. Let *k* be **1**.
10. If *k* equals Result(2), return *R*.
11. Let *S* be a string value produced by concatenating *R* and Result(4).
12. Call the [[Get]] method of this object with argument ToString(*k*).
13. If Result(12) is **undefined** or **null**, use the empty string; otherwise, call ToObject(Result(12)).toLocaleString().
14. Let *R* be a string value produced by concatenating *S* and Result(13).
15. Increase *k* by 1.
16. Go to step 10.

The **toLocaleString** function is not generic; it throws a **TypeError** exception if its **this** value is not an Array object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

NOTE
The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.4.4.4 Array.prototype.concat ([item1 [, item2 [, …]]])

When the **concat** method is called with zero or more arguments *item1*, *item2*, etc., it returns an array containing the array elements of the object followed by the array elements of each argument in order.

The following steps are taken:

1. Let *A* be a new array created as if by the expression **new Array()**.
2. Let *n* be 0.

3. Let *E* be this object.
4. If *E* is not an Array object, go to step 16.
5. Let *k* be 0.
6. Call the [[Get]] method of *E* with argument **"length"**.
7. If *k* equals Result(6) go to step 19.
8. Call ToString(*k*).
9. If *E* has a property named by Result(8), go to step 10, but if *E* has no property named by Result(8), go to step 13.
10. Call ToString(n).
11. Call the [[Get]] method of *E* with argument Result(8).
12. Call the [[Put]] method of *A* with arguments Result(10) and Result(11).
13. Increase *n* by 1.
14. Increase *k* by 1.
15. Go to step 7.
16. Call ToString(*n*).
17. Call the [[Put]] method of *A* with arguments Result(16) and *E*.
18. Increase *n* by 1.
19. Get the next argument in the argument list; if there are no more arguments, go to step 22.
20. Let *E* be Result(19).
21. Go to step 4.
22. Call the [[Put]] method of *A* with arguments **"length"** and *n*.
23. Return *A*.

The **length** property of the **concat** method is **1**.

NOTE
*The **concat** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **concat** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.5 Array.prototype.join (separator)

The elements of the array are converted to strings, and these strings are then concatenated, separated by occurrences of the *separator*. If no separator is provided, a single comma is used as the separator.

The **join** method takes one argument, *separator*, and performs the following steps:

1. Call the [[Get]] method of this object with argument **"length"**.
2. Call ToUint32(Result(1)).
3. If *separator* is **undefined**, let *separator* be the single-character string **","**.
4. Call ToString(*separator*).
5. If Result(2) is zero, return the empty string.
6. Call the [[Get]] method of this object with argument **"0"**.
7. If Result(6) is **undefined** or **null**, use the empty string; otherwise, call ToString(Result(6)).
8. Let *R* be Result(7).
9. Let *k* be **1**.
10. If *k* equals Result(2), return *R*.
11. Let *S* be a string value produced by concatenating *R* and Result(4).
12. Call the [[Get]] method of this object with argument ToString(*k*).
13. If Result(12) is **undefined** or **null**, use the empty string; otherwise, call ToString(Result(12)).
14. Let *R* be a string value produced by concatenating *S* and Result(13).
15. Increase *k* by 1.
16. Go to step 10.

The **length** property of the **join** method is **1**.

NOTE
*The **join** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the **join** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.6 Array.prototype.pop ( )

The last element of the array is removed from the array and returned.

1. Call the [[Get]] method of this object with argument **"length"**.
2. Call ToUint32(Result(1)).
3. If Result(2) is not zero, go to step 6.
4. Call the [[Put]] method of this object with arguments **"length"** and Result(2).
5. Return **undefined**.
6. Call ToString(Result(2)–1).
7. Call the [[Get]] method of this object with argument Result(6).
8. Call the [[Delete]] method of this object with argument Result(6).
9. Call the [[Put]] method of this object with arguments **"length"** and (Result(2)–1).
10. Return Result(7).

NOTE
*The **pop** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **pop** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.7 Array.prototype.push ([item1 [, item2 [, …]]])

The arguments are appended to the end of the array, in the order in which they appear. The new length of the array is returned as the result of the call.

When the **push** method is called with zero or more arguments *item1*, *item2*, etc., the following steps are taken:

1. Call the [[Get]] method of this object with argument **"length"**.
2. Let *n* be the result of calling ToUint32(Result(1)).
3. Get the next argument in the argument list; if there are no more arguments, go to step 7.
4. Call the [[Put]] method of this object with arguments ToString(*n*) and Result(3).
5. Increase *n* by 1.
6. Go to step 3.
7. Call the [[Put]] method of this object with arguments **"length"** and *n*.
8. Return *n*.

The **length** property of the **push** method is **1**.

NOTE
*The **push** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **push** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.8 Array.prototype.reverse ( )

The elements of the array are rearranged so as to reverse their order. The object is returned as the result of the call.

1. Call the [[Get]] method of this object with argument **"length"**.
2. Call ToUint32(Result(1)).
3. Compute floor(Result(2)/2).
4. Let *k* be **0**.
5. If *k* equals Result(3), return this object.
6. Compute Result(2)–*k*–1.
7. Call ToString(*k*).
8. Call ToString(Result(6)).
9. Call the [[Get]] method of this object with argument Result(7).
10. Call the [[Get]] method of this object with argument Result(8).
11. If this object does not have a property named by Result(8), go to step 19.
12. If this object does not have a property named by Result(7), go to step 16.
13. Call the [[Put]] method of this object with arguments Result(7) and Result(10).
14. Call the [[Put]] method of this object with arguments Result(8) and Result(9).

15. Go to step 25.
16. Call the [[Put]] method of this object with arguments Result(7) and Result(10).
17. Call the [[Delete]] method on this object, providing Result(8) as the name of the property to delete.
18. Go to step 25.
19. If this object does not have a property named by Result(7), go to step 23.
20. Call the [[Delete]] method on this object, providing Result(7) as the name of the property to delete..
21. Call the [[Put]] method of this object with arguments Result(8) and Result(9).
22. Go to step 25.
23. Call the [[Delete]] method on this object, providing Result(7) as the name of the property to delete.
24. Call the [[Delete]] method on this object, providing Result(8) as the name of the property to delete.
25. Increase *k* by 1.
26. Go to step 5.

NOTE
The ***reverse*** *function is intentionally generic; it does not require that its* ***this*** *value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the* ***reverse*** *function can be applied successfully to a host object is implementation-dependent.*

15.4.4.9 Array.prototype.shift ( )

The first element of the array is removed from the array and returned.

1. Call the [[Get]] method of this object with argument **"length"**.
2. Call ToUint32(Result(1)).
3. If Result(2) is not zero, go to step 6.
4. Call the [[Put]] method of this object with arguments **"length"** and Result(2).
5. Return **undefined**.
6. Call the [[Get]] method of this object with argument **0**.
7. Let *k* be 1.
8. If *k* equals Result(2), go to step 18.
9. Call ToString(*k*).
10. Call ToString(*k*–1).
11. If this object has a property named by Result(9), go to step 12; but if this object has no property named by Result(9), then go to step 15.
12. Call the [[Get]] method of this object with argument Result(9).
13. Call the [[Put]] method of this object with arguments Result(10) and Result(12).
14. Go to step 16.
15. Call the [[Delete]] method of this object with argument Result(10).
16. Increase *k* by 1.
17. Go to step 8.
18. Call the [[Delete]] method of this object with argument ToString(Result(2)–1).
19. Call the [[Put]] method of this object with arguments **"length"** and (Result(2)–1).
20. Return Result(6).

NOTE
The ***shift*** *function is intentionally generic; it does not require that its* ***this*** *value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the* ***shift*** *function can be applied successfully to a host object is implementation-dependent.*

15.4.4.10 Array.prototype.slice (start, end)

The **slice** method takes two arguments, *start* and *end*, and returns an array containing the elements of the array from element *start* up to, but not including, element *end* (or through the end of the array if *end* is **undefined**). If *start* is negative, it is treated as (*length*+*start*) where *length* is the length of the array. If *end* is negative, it is treated as (*length*+*end*) where *length* is the length of the array. The following steps are taken:

1. Let *A* be a new array created as if by the expression **`new Array()`**.
2. Call the [[Get]] method of this object with argument **`"length"`**.
3. Call ToUint32(Result(2)).
4. Call ToInteger(*start*).
5. If Result(4) is negative, use max((Result(3)+Result(4)),0); else use min(Result(4),Result(3)).
6. Let *k* be Result(5).
7. If *end* is **undefined**, use Result(3); else use ToInteger(*end*).
8. If Result(7) is negative, use max((Result(3)+Result(7)),0); else use min(Result(7),Result(3)).
9. Let *n* be 0.
10. If *k* is greater than or equal to Result(8), go to step 19.
11. Call ToString(*k*).
12. If this object has a property named by Result(11), go to step 13; but if this object has no property named by Result(11), then go to step 16.
13. Call ToString(*n*).
14. Call the [[Get]] method of this object with argument Result(11).
15. Call the [[Put]] method of *A* with arguments Result(13) and Result(14).
16. Increase *k* by 1.
17. Increase *n* by 1.
18. Go to step 10.
19. Call the [[Put]] method of *A* with arguments **`"length"`** and *n*.
20. Return *A*.

The **`length`** property of the **`slice`** method is **2**.

NOTE
*The **`slice`** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **`slice`** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.11 Array.prototype.sort (comparefn)

The elements of this array are sorted. The sort is not necessarily stable (that is, elements that compare equal do not necessarily remain in their original order). If *comparefn* is not **undefined**, it should be a function that accepts two arguments *x* and *y* and returns a negative value if $x < y$, zero if $x = y$, or a positive value if $x > y$.

If *comparefn* is not **undefined** and is not a consistent comparison function for the elements of this array (see below), the behaviour of **`sort`** is implementation-defined. Let *len* be ToUint32(**`this.length`**). If there exist integers *i* and *j* and an object *P* such that all of the conditions below are satisfied then the behaviour of **`sort`** is implementation-defined:

$0 \le i < len$

$0 \le j < len$

`this` does not have a property with name ToString(*i*)

P is obtained by following one or more [[Prototype]] properties starting at **`this`**

P has a property with name ToString(*j*)

Otherwise the following steps are taken.

1. Call the [[Get]] method of this object with argument **`"length"`**.
2. Call ToUint32(Result(1)).
3. Perform an implementation-dependent sequence of calls to the [[Get]] , [[Put]], and [[Delete]] methods of this object and to SortCompare (described below), where the first argument for each call to [[Get]], [[Put]], or [[Delete]] is a nonnegative integer less than Result(2) and where the arguments for calls to SortCompare are results of previous calls to the [[Get]] method.
4. Return this object.

The returned object must have the following two properties.

There must be some mathematical permutation π of the nonnegative integers less than Result(2), such that for every nonnegative integer *j* less than Result(2), if property **`old[`***j***`]`** existed, then

new[π(*j*)**]** is exactly the same value as **old[***j***],**. but if property **old[***j***]** did not exist, then **new[**π(*j*)**]** does not exist.

Then for all nonnegative integers *j* and *k*, each less than Result(2), if SortCompare(*j*,*k*) < 0 (see SortCompare below), then $\pi(j) < \pi(k)$.

Here the notation **old[***j***]** is used to refer to the hypothetical result of calling the [[Get]] method of this object with argument *j* before this function is executed, and the notation **new[***j***]** to refer to the hypothetical result of calling the [[Get]] method of this object with argument *j* after this function has been executed.

A function *comparefn* is a consistent comparison function for a set of values *S* if all of the requirements below are met for all values *a*, *b*, and *c* (possibly the same value) in the set *S*: The notation $a <_{CF} b$ means *comparefn*(*a*,*b*) < 0; $a =_{CF} b$ means *comparefn*(*a*,*b*) = 0 (of either sign); and $a >_{CF} b$ means *comparefn*(*a*,*b*) > 0.

Calling *comparefn*(*a*,*b*) always returns the same value *v* when given a specific pair of values *a* and *b* as its two arguments. Furthermore, *v* has type Number, and *v* is not NaN. Note that this implies that exactly one of $a <_{CF} b$, $a =_{CF} b$, and $a >_{CF} b$ will be true for a given pair of *a* and *b*.

$a =_{CF} a$ (reflexivity)

If $a =_{CF} b$, then $b =_{CF} a$ (symmetry)

If $a =_{CF} b$ and $b =_{CF} c$, then $a =_{CF} c$ (transitivity of $=_{CF}$)

If $a <_{CF} b$ and $b <_{CF} c$, then $a <_{CF} c$ (transitivity of $<_{CF}$)

If $a >_{CF} b$ and $b >_{CF} c$, then $a >_{CF} c$ (transitivity of $>_{CF}$)

NOTE
The above conditions are necessary and sufficient to ensure that comparefn divides the set S into equivalence classes and that these equivalence classes are totally ordered.

When the SortCompare operator is called with two arguments *j* and *k*, the following steps are taken:

1. Call ToString(*j*).
2. Call ToString(*k*).
3. If this object does not have a property named by Result(1), and this object does not have a property named by Result(2), return **+0**.
4. If this object does not have a property named by Result(1), return 1.
5. If this object does not have a property named by Result(2), return –1.
6. Call the [[Get]] method of this object with argument Result(1).
7. Call the [[Get]] method of this object with argument Result(2).
8. Let *x* be Result(6).
9. Let *y* be Result(7).
10. If *x* and *y* are both **undefined**, return **+0**.
11. If *x* is **undefined**, return 1.
12. If *y* is **undefined**, return –1.
13. If the argument *comparefn* is **undefined**, go to step 16.
14. Call *comparefn* with arguments *x* and *y*.
15. Return Result(14).
16. Call ToString(*x*).
17. Call ToString(*y*).
18. If Result(16) < Result(17), return –1.
19. If Result(16) > Result(17), return 1.
20. Return **+0**.

NOTE 1
Because non-existent property values always compare greater than ***undefined*** *property values, and* ***undefined*** *always compares greater than any other value, undefined property values always sort to the end of the result, followed by non-existent property values.*

NOTE 2

The **`sort`** *function is intentionally generic; it does not require that its* ***this*** *value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the* **`sort`** *function can be applied successfully to a host object is implementation-dependent.*

15.4.4.12 Array.prototype.splice (start, deleteCount [, item1 [, item2 [, ...]]])

When the **`splice`** method is called with two or more arguments *start*, *deleteCount* and (optionally) *item1*, *item2*, etc., the *deleteCount* elements of the array starting at array index *start* are replaced by the arguments *item1*, *item2*, etc. The following steps are taken:

1. Let *A* be a new array created as if by the expression **`new Array()`**.
2. Call the [[Get]] method of this object with argument **`"length"`**.
3. Call ToUint32(Result(2)).
4. Call ToInteger(*start*).
5. If Result(4) is negative, use max((Result(3)+Result(4)),0); else use min(Result(4),Result(3)).
6. Compute min(max(ToInteger(*deleteCount*),0),Result(3)–Result(5)).
7. Let *k* be 0.
8. If *k* equals Result(6), go to step 16.
9. Call ToString(Result(5)+*k*).
10. If this object has a property named by Result(9), go to step 11; but if this object has no property named by Result(9), then go to step 14.
11. Call ToString(*k*).
12. Call the [[Get]] method of this object with argument Result(9).
13. Call the [[Put]] method of *A* with arguments Result(11) and Result(12).
14. Increment *k* by 1.
15. Go to step 8.
16. Call the [[Put]] method of *A* with arguments **`"length"`** and Result(6).
17. Compute the number of additional arguments *item1*, *item2*, etc.
18. If Result(17) is equal to Result(6), go to step 48.
19. If Result(17) is greater than Result(6), go to step 37.
20. Let *k* be Result(5).
21. If *k* is equal to (Result(3)–Result(6)), go to step 31.
22. Call ToString(*k*+Result(6)).
23. Call ToString(*k*+Result(17)).
24. If this object has a property named by Result(22), go to step 25; but if this object has no property named by Result(22), then go to step 28.
25. Call the [[Get]] method of this object with argument Result(22).
26. Call the [[Put]] method of this object with arguments Result(23) and Result(25).
27. Go to step 29.
28. Call the [[Delete]] method of this object with argument Result(23).
29. Increase *k* by 1.
30. Go to step 21.
31. Let *k* be Result(3).
32. If *k* is equal to (Result(3)–Result(6)+Result(17)), go to step 48.
33. Call ToString(*k*–1).
34. Call the [[Delete]] method of this object with argument Result(33).
35. Decrease *k* by 1.
36. Go to step 32.
37. Let *k* be (Result(3)–Result(6)).
38. If *k* is equal to Result(5), go to step 48.
39. Call ToString(*k*+Result(6)–1).
40. Call ToString(*k*+Result(17)–1)
41. If this object has a property named by Result(39), go to step 42; but if this object has no property named by Result(39), then go to step 45.
42. Call the [[Get]] method of this object with argument Result(39).
43. Call the [[Put]] method of this object with arguments Result(40) and Result(42).
44. Go to step 46.
45. Call the [[Delete]] method of this object with argument Result(40).
46. Decrease *k* by 1.
47. Go to step 38.

48. Let *k* be Result(5).
49. Get the next argument in the part of the argument list that starts with *item1*; if there are no more arguments, go to step 53.
50. Call the [[Put]] method of this object with arguments ToString(*k*) and Result(49).
51. Increase *k* by 1.
52. Go to step 49.
53. Call the [[Put]] method of this object with arguments **"length"** and (Result(3)–Result(6)+Result(17)).
54. Return *A*.

The **length** property of the **splice** method is **2**.

NOTE
*The **splice** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **splice** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.13 Array.prototype.unshift ([item1 [, item2 [, ...]]])

The arguments are prepended to the start of the array, such that their order within the array is the same as the order in which they appear in the argument list.

When the **unshift** method is called with zero or more arguments *item1*, *item2*, etc., the following steps are taken:

1. Call the [[Get]] method of this object with argument **"length"**.
2. Call ToUint32(Result(1)).
3. Compute the number of arguments.
4. Let *k* be Result(2).
5. If *k* is zero, go to step 15.
6. Call ToString(*k*–1).
7. Call ToString(*k*+Result(3)–1).
8. If this object has a property named by Result(6), go to step 9; but if this object has no property named by Result(6), then go to step 12.
9. Call the [[Get]] method of this object with argument Result(6).
10. Call the [[Put]] method of this object with arguments Result(7) and Result(9).
11. Go to step 13.
12. Call the [[Delete]] method of this object with argument Result(7).
13. Decrease *k* by 1.
14. Go to step 5.
15. Let *k* be 0.
16. Get the next argument in the part of the argument list that starts with *item1*; if there are no more arguments, go to step 21.
17. Call ToString(*k*).
18. Call the [[Put]] method of this object with arguments Result(17) and Result(16).
19. Increase *k* by 1.
20. Go to step 16.
21. Call the [[Put]] method of this object with arguments **"length"** and (Result(2)+Result(3)).
22. Return (Result(2)+Result(3)).

The **length** property of the **unshift** method is **1**.

NOTE
*The **unshift** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **unshift** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.14 Array.prototype.indexOf (searchElement[, fromIndex])

indexOf compares *searchElement* to the elements of the array in ascending order using strict equality, and if found at one or more positions, returns the index of the first such position; otherwise, -1 is returned.

The optional second argument *fromIndex* defaults to 0 (i.e. the whole array is searched). If it is greater than or equal to the length of the array, -1 is returned, i.e. the array will not be searched. If it is negative, it is used as the offset from the end of the array to compute *fromIndex*. If the computed index is less than 0, the whole array will be searched.

When the **`indexOf`** method is called with one or more arguments, the following steps are taken:

1. Let *E* be this object
2. Call the [[Get]] method of *E* with the argument "**`length`**".
3. Call ToUint32(Result(2)).
4. If Result(3) is 0 go to step 18.
5. Call ToInt32(*fromIndex*) (if *fromIndex* is undefined this step produces 0)
6. Let *n* be Result(5).
7. If *n* is greater than or equal to Result(3) go to step 18.
8. If *n* is greater than or equal to 0, let *k* be *n*, and go to step 11.
9. Let *k* be Result(3) - abs(n).
10. If *k* is less than 0, let k be 0.
11. Call ToString(*k*).
12. Call the [[Get]] method of *E* with the argument Result(11).
13. Perform the comparison *searchElement* === Result(12).
14. If Result(13) is **false** go to step 16.
15. Return *k*.
16. Increase *k* by 1.
17. If *k* is less than Result(3) go to step 11.
18. Return -1.

The **`length`** property of the **`indexOf`** method is **1**.

NOTE

*The **`indexOf`** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **`indexOf`** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.15 Array.prototype.lastIndexOf (searchElement[, fromIndex])

`lastIndexOf` compares *searchElement* to the elements of the array in descending order using strict equality, and if found at one or more positions, returns the index of the last such position; otherwise, -1 is returned.

The optional second argument *fromIndex* defaults to the array's length (i.e. the whole array is searched). If it is greater than or equal to the length of the array, the whole array will be searched. If it is negative, it is used as the offset from the end of the array to compute *fromIndex*. If the computer index is less than 0, -1 is returned.

When the **`lastIndexOf`** method is called with one or more arguments, the following steps are taken:

1. Let *E* be this object
2. Call the [[Get]] method of *E* with the argument "**`length`**".
3. Call ToUint32(Result(2)).
4. If Result(3) is 0 go to step 18.
5. Call ToInt32(*fromIndex*) (if *fromIndex* is undefined this step produces the same values as Result(3))
6. Let *n* be Result(5).
7. If *n* is greater than or equal to Result(3), let *k* be Result(3) - 1, and go to step 11.
8. If *n* is greater than or equal to 0, let *k* be *n*, and go to step 11.
9. Let *k* be Result(3) - abs(*n*).
10. If *k* is less than 0 go to step 18.
11. Call ToString(*k*).
12. Call the [[Get]] method of *E* with the argument Result(11).
13. Perform the comparison searchElement === Result(12).
14. If Result(13) is false go to step 16.
15. Return *k*.

16. Decrease *k* by 1.
17. If *k* is greater than or equal to 0 go to step 11.
18. Return -1.

The **`length`** property of the **`lastIndexOf`** method is **1**.

NOTE

*The **`lastIndexOf`** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **`lastIndexOf`** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.16 Array.prototype.every (callbackfn)

callbackfn should be a function that accepts three arguments and returns the boolean value **true** or **false**. **`every`** calls the provided callback, as a function, once for each element present in the array, in ascending order, until it finds one where *callbackfn* returns a **false** value. If such an element is found, **`every`** immediately returns **false**. Otherwise, if *callbackfn* returned a **true** value for all elements, **`every`** will return **true**. *callbackfn* is called only for indexes of the array which have assigned values; it is not called for indexes which have been deleted or which have never been assigned values.

callbackfn is called with three arguments: the value of the element, the index of the element, and the Array object being traversed.

`every` does not mutate the array on which it is called.

The range of elements processed by **`every`** is set before the first call to *callbackfn*. Elements which are appended to the array after the call to **`every`** begins will not be visited by *callbackfn*. If existing elements of the array are changed, their value as passed to *callbackfn* will be the value at the time **`every`** visits them; elements that are deleted are not visited. **`every`** acts like the "for all" quantifier in mathematics. In particular, for an empty array, it returns **true**.

When the **`every`** method is called with one argument, the following steps are taken:

1. Let *E* be this object
2. Call the [[Get]] method of *E* with the argument "**`length`**".
3. Call ToUint32(Result(2)).
4. If Result(3) is 0 go to step 16.
5. If Type(*callbackfn*) is not a function, throw a **TypeError** exception.
6. Let *k* be 0.
7. Call ToString(*k*).
8. If *E* does not have a property named by Result(7), go to step 14.
9. Call the [[Get]] method of *E* with argument Result(7).
10. Call *callbackfn* with arguments Result(9), *k*, and *E*.
11. Call ToBoolean(Result(10)).
12. If Result(11) is true go to step 14.
13. Return **false**.
14. Increase *k* by 1.
15. If *k* is less than Result(3) go to step 7.
16. Return **true**.

The **`length`** property of the **`every`** method is **1**.

NOTE

*The **`every`** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **`every`** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.17 Array.prototype.some (callbackfn)

callbackfn should be a function that accepts three arguments and returns the boolean value **true** or **false**. **`some`** calls the callback, as a function, once for each element present in the array, in ascending order,

until it finds one where *callbackfn* returns a **true** value. If such an element is found, **some** immediately returns **true**. Otherwise, **some** returns **false**. *callbackfn* is called only for indexes of the array which have assigned values; it is not called for indexes which have been deleted or which have never been assigned values.

callbackfn is called with three arguments: the value of the element, the index of the element, and the Array object being traversed.

some does not mutate the array on which it is called.

The range of elements processed by **some** is set before the first call to *callbackfn*. Elements that are appended to the array after the call to **some** begins will not be visited by *callbackfn*. If an existing, unvisited element of the array is changed by *callbackfn*, their value as passed to *callbackfn* will be the value at the time that **some** visits them; elements that are deleted are not visited.

When the **some** method is called with one argument, the following steps are taken:

1. Let *E* be this object
2. Call the [[Get]] method of *E* with the argument "**length**".
3. Call ToUint32(Result(2)).
4. If Result(3) is 0 go to step 15.
5. If Type(*callbackfn*) is not a function, throw a **TypeError** exception.
6. Let *k* be 0.
7. Call ToString(*k*).
8. If *E* does not have a property named by Result(7), go to step 14.
9. Call the [[Get]] method of *E* with argument Result(7).
10. Call callbackfn with arguments Result(9), *k*, and *E*.
11. Call ToBoolean(Result(10)).
12. If Result(11) is false go to step 13.
13. Return **true**.
14. Increase *k* by 1.
15. If *k* is less than Result(3) go to step 7.
16. Return **false**.

The **length** property of the **some** method is **1**.

NOTE

*The **some** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **some** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.18 Array.prototype.forEach (callbackfn)

callbackfn should be a function that accepts three arguments. **forEach** calls the provided callback, as a function, once for each element present in the array, in ascending order. *callbackfn* is called only for indexes of the array which have assigned values; it is not called for indexes which have been deleted or which have never been assigned values.

callbackfn is called with three arguments: the value of the element, the index of the element, and the Array object being traversed.

forEach does not mutate the array on which it is called.

The range of elements processed by **forEach** is set before the first call to *callbackfn*. Elements which are appended to the array after the call to **forEach** begins will not be visited by *callbackfn*. If existing elements of the array are changed, or deleted, their value as passed to callback will be the value at the time **forEach** visits them; elements that are deleted are not visited.

When the **forEach** method is called with one argument, the following steps are taken:

1. Let *E* be this object
2. Call the [[Get]] method of E with the argument "**length**".
3. Call ToUint32(Result(2)).

4. If Result(3) is 0 go to step 13.
5. If Type(*callbackfn*) is not a function, throw a **TypeError** exception.
6. Let *k* be 0.
7. Call ToString(*k*).
8. If *E* does not have a property named by Result(7), go to step 11.
9. Call the [[Get]] method of *E* with argument Result(7).
10. Call *callbackfn* with arguments Result(9), k, and E
11. Increase *k* by 1.
12. If *k* is less than Result(3) go to step 7.
13. Return.

The **`length`** property of the **`forEach`** method is **1**.

NOTE

*The **`forEach`** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **`forEach`** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.19 Array.prototype.map (callbackfn)

callbackfn should be a function that accepts three arguments. **`map`** calls the provided callback, as a function, once for each element in the array, in ascending order, and constructs a new array from the results. *callbackfn* is called only for indexes of the array which have assigned values; it is not called for indexes which have been deleted or which have never been assigned values.

callbackfn is called with three arguments: the value of the element, the index of the element, and the Array object being traversed.

`map` does not mutate the array on which it is called.

The range of elements processed by **`map`** is set before the first call to *callbackfn*. Elements which are appended to the array after the call to **`map`** begins will not be visited by *callbackfn*. If existing elements of the array are changed, or deleted, their value as passed to *callbackfn* will be the value at the time **`map`** visits them; elements that are deleted are not visited.

When the **`map`** method is called with one argument, the following steps are taken:

1. Let *A* be a new array created as if by the expression **`new Array()`**.
2. Let *n* be 0.
3. Let *E* be this object
4. Call the [[Get]] method of *E* with the argument "**`length`**".
5. Call ToUint32(Result(4)).
6. If Result(5) is 0 go to step 18.
7. If Type(*callbackfn*) is not a function, throw a **TypeError** exception.
8. Let *k* be 0.
9. Call ToString(*k*).
10. If *E* does not have a property named by Result(9), go to step 20.
11. Call the [[Get]] method of *E* with argument Result(9).
12. Call *callbackfn* with arguments Result(11), *k*, and *E*.
13. Call ToString(*n*).
14. Call the [[Put]] method of *A* with the argument Result(12) and Result(13).
15. Increase *n* by 1.
16. Increase *k* by 1.
17. If *k* is less than Result(5) go to step 9.
18. Return *A*.

The **`length`** property of the **`map`** method is **1**.

NOTE

*The **map** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **map** function can be applied successfully to a host object is implementation-dependent.*

15.4.4.20 Array.prototype.filter (callbackfn)

callbackfn should be a function that accepts three arguments and returns the boolean value **true** or **false**. **filter** calls the provided callback, as a function, once for each element in the array, in ascending order, and constructs a new array of all the values for which *callbackfn* returns **true**. *callbackfn* is called only for indexes of the array which have assigned values; it is not called for indexes which have been deleted or which have never been assigned values.

callbackfn is called with three arguments: the value of the element, the index of the element, and the Array object being traversed.

filter does not mutate the array on which it is called.

The range of elements processed by **filter** is set before the first call to *callbackfn*. Elements which are appended to the array after the call to **filter** begins will not be visited by *callbackfn*. If existing elements of the array are changed, or deleted, their value as passed to *callbackfn* will be the value at the time **filter** visits them; elements that are deleted are not visited.

When the **filter** method is called with one argument, the following steps are taken:

1. Let *A* be a new array created as if by the expression **new Array()**.
2. Let *n* be 0.
3. Let *E* be this object
4. Call the [[Get]] method of *E* with the argument "**length**".
5. Call ToUint32(Result(4)).
6. If Result(5) is 0 go to step 20.
7. If Type(*callbackfn*) is not a function, throw a **TypeError** exception.
8. Let *k* be 0.
9. Call ToString(*k*).
10. If *E* does not have a property named by Result(9), go to step 18.
11. Call the [[Get]] method of *E* with argument Result(9).
12. Call *callbackfn* with arguments Result(10), *k*, and *E*
13. Call ToBoolean(Result(12)).
14. If Result(13) is **false** go to step 18.
15. Call ToString(*n*).
16. Call the [[Put]] method of *A* with the argument Result(11) and Result(15).
17. Increase *n* by 1.
18. Increase *k* by 1.
19. If *k* is less than Result(5) go to step 9.
20. Return *A*.

The **length** property of the **filter** method is **1**.

NOTE

*The **filter** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **filter** function can be applied successfully to a host object is implementation-dependent.*

15.4.5 Properties of Array Instances

Array instances inherit properties from the Array prototype object and also have the following properties.

15.4.5.1 [[Put]] (P, V)

Array objects use a variation of the [[Put]] method used for other native ECMAScript objects (8.6.2.2).

Assume *A* is an Array object and *P* is a string.

When the [[Put]] method of *A* is called with property *P* and value *V*, the following steps are taken:

1. Call the [[CanPut]] method of *A* with name P.
2. If Result(1) is **false**, return.
3. If *A* doesn't have a property with name *P*, go to step 7.
4. If P is **"length"**, go to step 12.
5. Set the value of property *P* of *A* to *V*.
6. Go to step 8.
7. Create a property with name *P*, set its value to *V* and give it empty attributes.
8. If *P* is not an array index, return.
9. If ToUint32(*P*) is less than the value of the **length** property of *A*, then return.
10. Change (or set) the value of the **length** property of *A* to ToUint32(*P*)+1.
11. Return.
12. Compute ToUint32(*V*).
13. If Result(12) is not equal to ToNumber(*V*), throw a **RangeError** exception.
14. For every integer *k* that is less than the value of the **length** property of *A* but not less than Result(12), if *A* itself has a property (not an inherited property) named ToString(*k*), then delete that property.
15. Set the value of property *P* of *A* to Result(12).
16. Return.

15.4.5.2 length

The **length** property of this Array object is always numerically greater than the name of every property whose name is an array index.

The **length** property has the attributes { DontEnum, DontDelete }.