Array class

docs

Note: Most examples have been written to run in irb. We changed a few for *.rb work. Some of the documented code here is refering to the pickax book. see https://pragprog.com/book/ruby/programming-ruby or use the free online version http://ruby-doc.com/docs/ProgrammingRuby/

Array < Object means inherits from

Arrays are ordered, integer-indexed collections of any object. Array indexing starts at 0, as in C or Java. A negative index is assumed to be relative to the end of the array; that is, an index of −1 indicates the last element of the array, −2 is the next to last element in the array, and so on.

Mixes in Enumerable https://ruby-doc.org/core-2.2.3/Enumerable.html

```
puts Array.include? Enumerable
#=> true
puts Enumerable.public_instance_methods(false).sort
#=>
[:all?, :any?, :chunk, :collect, :collect_concat, :count, :cycle, :detect, :drop,
 :drop_while, :each_cons, :each_entry, :each_slice, :each_with_index,
 :each_with_object, :entries, :enum_with_index, :find, :find_all, :find_index,
 :first, :flat_map, :grep, :group_by, :include?, :inject, :map, :max, :max_by,
 :member?, :min, :min_by, :minmax, :minmax_by, :none?, :one?, :partition,
 :reduce, :reject, :reverse_each, :select, :slice_before, :sort, :sort_by, :take,
 :take_while, :to_a, :with_object, :zip]

Enumerable.public_instance_methods(false).sort.each {|methods| p methods }
```

Creating Arrays

A new array can be created by using the literal constructor []. Arrays can contain different types of objects. For example, the array below contains an Integer, a String and a Float:

```
ary = [1, "two", 3.0]
puts ary.inspect
#=> [1, "two", 3.0]
```

An array can also be created by explicitly calling ::new with zero, one (the initial size of the Array) or two arguments (the initial size and a default object).

```
ary                                                   = Array.new
puts ary.inspect                                      #=> []
```

```
ary                                       = Array.new(3)
puts ary.inspect                          #=> [nil, nil, nil]
ary                                       = Array.new(3, true)
puts ary.inspect                          #=> [true, true, true]
```

Note that the second argument populates the array with references to the same object. Therefore, it is only recommended in cases when you need to instantiate arrays with natively immutable objects such as Symbols, numbers, true or false.

To create an array with separate objects a block can be passed instead. This method is safe to use with mutable objects such as hashes, strings or other arrays:

```
ary                                       = Array.new(4) { Hash.new }
puts ary.inspect                          #=> [{}, {}, {}, {}]
```

This is also a quick way to build up multi-dimensional arrays:

```
empty_table                               = Array.new(3) { Array.new(3) }
puts empty_table.inspect
#=> [[nil, nil, nil], [nil, nil, nil], [nil, nil, nil]]
```

An array can also be created by using the Array() method, provided by Kernel, which tries to call #to_ary, then #to_a on its argument.

```
ary                                       = Array({:a => "a", :b => "b"})
puts ary.inspect                          #=> [[:a, "a"], [:b, "b"]]
```

Example Usage

In addition to the methods it mixes in through the Enumerable module, the Array class has proprietary methods for accessing, searching and otherwise manipulating arrays.

Some of the more common ones are illustrated below.

Accessing Elements

Elements in an array can be retrieved using the #[] method. It can take a single integer argument (a numeric index), a pair of arguments (start and length) or a range. Negative indices start counting from the end, with -1 being the last element.

```
arr                                       =   [1, 2, 3, 4, 5, 6]
puts arr[2].inspect                       #=> 3
puts arr[100].inspect                     #=> nil
puts arr[-3].inspect                      #=> 4
puts arr[2, 3].inspect                    #=> [3, 4, 5]
puts arr[1..4].inspect                    #=> [2, 3, 4, 5]
puts arr[1..-3].inspect                                 #=> [2, 3, 4]
```

Another way to access a particular array element is by using the #at method

```
puts arr.at(0).inspect                                  #=> 1
```

The slice method works in an identical manner to #[].

The special methods #first and #last will return the first and last elements of an array, respectively.

```
puts arr.first.inspect                                  #=> 1
puts arr.last.inspect                                   #=> 6
```

To return the first n elements of an array, use #take

```
puts arr.take(3).inspect                                #=> [1, 2, 3]
```

#drop does the opposite of take, by returning the elements after n elements have been dropped:

```
puts arr.drop(3).inspect                                #=> [4, 5, 6]
```

To raise an error for indices outside of the array bounds or else to provide a default value when that happens, you can use #fetch.

```
arr                                        = ['a', 'b', 'c', 'd', 'e', 'f']

puts arr.fetch(100, "oops").inspect    #=> "oops"
puts arr.fetch(100).inspect            #=> IndexError: index 100 outside of array
```

Obtaining Information about an Array

Arrays keep track of their own #length at all times.

```
# To query an array about the number of elements it contains,
# use length, count or size.

browsers = ['Chrome', 'Firefox', 'Safari', 'Opera', 'IE']
puts browsers.length.inspect                            #=> 5
puts browsers.count.inspect                             #=> 5
puts browsers.size.inspect                              #=> 5

  # To check whether an array contains any elements at all

puts browsers.empty?                                    #=> false
```

```
  # To check whether a particular item is included in the array

puts browsers.include?('Konqueror')                        #=> false
```

Adding Items to Arrays

```
arr                                                        =   [1, 2, 3, 4]
arr.push(5)
puts arr.inspect                                           #=> [1, 2, 3, 4, 5]
arr << 6
puts arr.inspect                                           #=> [1, 2, 3, 4, 5, 6]

  # unshift will add a new item to the beginning of an array.

arr.unshift(0)
puts arr.inspect                                           #=> [0, 1, 2, 3, 4, 5, 6]

  # With insert you can add a new element to an array at any position.

arr.insert(3, 'apple')
puts arr.inspect
#=> [0, 1, 2, 'apple', 3, 4, 5, 6]

  # Using the insert method, you can also insert multiple values at once:

arr.insert(3, 'orange', 'pear', 'grapefruit')
puts arr.inspect
#=> [0, 1, 2, "orange", "pear", "grapefruit", "apple", 3, 4, 5, 6]
```

Removing Items from an Array

```
  # The method pop removes the last element in an array and returns it:

arr                                                        =   [1, 2, 3, 4, 5, 6]
puts arr.pop.inspect                                       #=> 6
puts arr.inspect                                           #=> [1, 2, 3, 4, 5]

  # To retrieve and at the same time remove the first item, use shift:

puts arr.shift.inspect                                     #=> 1
puts arr.inspect                                           #=> [2, 3, 4, 5]

  # To delete an element at a particular index:
```

```
puts arr.delete_at(2).inspect                         #=> 4
puts arr.inspect                                      #=> [2, 3, 5]

  # To delete a particular element anywhere in an array, use delete:

arr                                                   =   [1, 2, 2, 3]
puts arr.delete(2).inspect                            #=> 2
puts arr.inspect                                      #=> [1,3]

  # A useful method if you need to remove nil values from an array is compact:

arr = ['foo', 0, nil, 'bar', 7, 'baz', nil]
puts arr.compact.inspect   #=> ['foo', 0, 'bar', 7, 'baz']
puts arr.inspect           #=> ['foo', 0, nil, 'bar', 7, 'baz', nil]
puts arr.compact!.inspect  #=> ['foo', 0, 'bar', 7, 'baz']
puts arr.inspect           #=> ['foo', 0, 'bar', 7, 'baz']

  # Another common need is to remove duplicate elements from an array.

# It has the non-destructive uniq, and destructive method uniq!

arr = [2, 5, 6, 556, 6, 6, 8, 9, 0, 123, 556]
puts arr.inspect                        #=> [2, 5, 6, 556, 6, 6, 8, 9, 0, 123, 556]
puts arr.uniq.inspect                   #=> [2, 5, 6, 556, 8, 9, 0, 123]
puts arr.inspect                        #=> [2, 5, 6, 556, 6, 6, 8, 9, 0, 123, 556]
puts arr.uniq!.inspect                  #=> [2, 5, 6, 556, 8, 9, 0, 123]
puts arr.inspect                        #=> [2, 5, 6, 556, 8, 9, 0, 123]
```

Iterating over Arrays

Like all classes that include theEnumerable module, Array has an each method, which defines what elements should be iterated over and how. In case of Array's each, all elements in the Array instance are yielded to the supplied block in sequence.

```
  # Note that this operation leaves the array unchanged.

arr                                                   =   [1, 2, 3, 4, 5]
arr.each { |a| print a -= 10, " " }                   #=> -9 -8 -7 -6 -5
print "\n"

  # Another sometimes useful iterator is reverse_each which will iterate over the
  # elements in the array in reverse order.

words = %w[first second third fourth fifth sixth]
str = ""
```

```
words.reverse_each { |word| str += "#{word} " }
p str
#=> "sixth fifth fourth third second first "

# The map method can be used to create a new array based on the original array,
# but with the values modified by the supplied block:

arr.map { |a| 2*a }
puts arr.inspect                                        #=> [1, 2, 3, 4, 5]
puts arr.map { |a| 2*a }.inspect                        #=> [2, 4, 6, 8, 10]
arr.map! { |a| a**2 }
puts arr.inspect                                        #=> [1, 4, 9, 16, 25]
```

Selecting Items from an Array

Elements can be selected from an array according to criteria defined in a block. The selection can happen in a destructive or a non-destructive manner. While the destructive operations will modify the array they were called on, the non-destructive methods usually returns a new array with the selected elements, but leave the original array unchanged.

Non-destructive Selection

```
arr                                                     =   [1, 2, 3, 4, 5, 6]
puts arr.select { |a| a > 3 }.inspect                   #=> [4, 5, 6]
puts arr.reject { |a| a < 3 }.inspect                   #=> [3, 4, 5, 6]
puts arr.drop_while { |a| a < 4 }.inspect               #=> [4, 5, 6]
puts arr.inspect                                        #=> [1, 2, 3, 4, 5, 6]
```

Destructive Selection

#select! and #reject! are the corresponding destructive methods to #select and #reject

```
  # Similar to select vs. reject, delete_if and keep_if have the exact opposite
  # result when supplied with the same block:

puts  arr.delete_if { |a| a < 4 }.inspect              #=> [4, 5, 6]
puts arr.inspect                                        #=> [4, 5, 6]

arr                                                     =   [1, 2, 3, 4, 5, 6]
puts arr.keep_if { |a| a < 4 }.inspect                  #=> [1, 2, 3]
puts arr.inspect                                        #=> [1, 2, 3]
```

Public Class Methods

```
puts Array.constants                                    #=> [:ClassMethods]
```

[](*args) → an_array

Array[obj*] → an_array

Returns a new array populated with the given objects. Equivalent to the operator form Array.[](. . .).

```
puts Array.[]( 1, 'a', /^A/ ).inspect                 #=> [1, "a", /^A/]
puts Array[ 1, 'a', /^A/ ].inspect                    #=> [1, "a", /^A/]
puts [ 1, 'a', /^A/ ].inspect                         #=> [1, "a", /^A/]
```

Array.new → new array

Array.new (size=0, obj=nil) → new array

Array.new(array) → new array

Array.new(size) {| i | block } → new array

Returns a new array.

In the first form, if no arguments are sent, the new array is empty.

```
puts Array.new.inspect                                #=> []
```

In the second it is created with size copies of obj (that is, size references to the same obj). When a size and an optional obj are sent, an array is created with size copies of obj.

```
puts Array.new(2).inspect                             #=> [nil, nil]
puts Array.new(5, "A").inspect                        #=> ["A", "A", "A", "A", "A"]
```

The third form creates a copy of the array passed as a parameter (the array is generated by calling to_ary on the parameter).

```
first_array                                           = ["Matz", "Guido"]
second_array                                          = Array.new(first_array)
puts first_array.equal? second_array                  #=> false (it's not a copy)
```

In the last form, an array of the given size is created. Each element in this array is calculated by passing the element’s index to the given block and storing the return value.

```
ary = Array.new(5){ |index| index ** 2 }
puts ary.inspect                                      # => [0, 1, 4, 9, 16]

squares                                               = Array.new(5) {|i| i*i}
puts squares.inspect                                  #=> [0, 1, 4, 9, 16]
```

new array initialized by copying:

```
copy                                                    = Array.new(squares)
```

now add element using block. see above:

```
squares[5]                                              = 25
```

new element added:

```
puts squares.inspect                                    #=> [0, 1, 4, 9, 16, 25]
```

not effecting our copy previously made:

```
puts copy.inspect                                       #=> [0, 1, 4, 9, 16]
```

Common gotchas

Array elements

When sending the second parameter, the same object will be used as the value for all the array elements:

Only one instance of the default object is created using the second form

```
a = Array.new(2, Hash.new)
puts a.inspect                                          #=> [{}, {}]
a[0]['cat']                                             = 'feline'
puts a.inspect                #=> [{"cat"=>"feline"}, {"cat"=>"feline"}]
a[1]['cat']                                             = 'Felix'
puts a.inspect                #=> [{"cat"=>"Felix"}, {"cat"=>"Felix"}]
```

Since all the Array elements store the same hash, changes to one of them will affect them all.

Multiple instances may be created using the last form of passing the block. If multiple copies are what you want, you should use the block version which uses the result of that block each time an element of the array needs to be initialized:

```
a                                                       = Array.new(2) { Hash.new }
puts a.inspect                                          #=> [{}, {}]
a[0]['cat']                                             = 'feline'
puts a.inspect                                          #=> [{"cat"=>"feline"}, {}]
a[1]['cat']                                             = 'Felix'
puts a.inspect
#=> [{"cat"=>"feline"}, {"cat"=>"Felix"}]
```

try_convert(obj) → array or nil

Tries to convert obj into an array, using to_ary method. Returns the converted array or nil if obj cannot be converted for any reason. This method can be used to check if an argument is an array.

```
puts Array.try_convert([1]).inspect                #=> [1]
puts Array.try_convert("1").inspect                #=> nil

if tmp = Array.try_convert(arg)                    # the argument is an array
elsif tmp = String.try_convert(arg)                # the argument is a string
end
```

Set Intersection

arr & other_array → new array

Returns a new array containing elements common to the two arrays, with no duplicates. The rules for comparing elements are the same as for hash keys. If you need set like behaviour, see the library class Set on page 710.

```
puts ([ 1, 1, 3, 5 ] & [ 1, 2, 3 ]).inspect        #=> [1, 3]
```

Repetition

arr * int → new array

arr * str → a_string

With an argument that responds to to_str, equivalent to arr.join(str). Otherwise, returns a new array built by concatenating int copies of arr.

```
puts ([ 1, 2, 3 ] * 3).inspect                     #=> [1, 2, 3, 1, 2, 3, 1, 2, 3]
puts ([ 1, 2, 3 ] * "").inspect                    #=> "123"
```

Concatenation

arr + other_array → new array

Returns a new array built by concatenating the two arrays together to produce a third array.

```
puts ([ 1, 2, 3 ] + [ 4, 5 ]).inspect              #=> [1, 2, 3, 4, 5]
a                                                  = [ "a", "b", "c" ]
c                                                  = a + [ "d", "e", "f" ]
puts c.inspect
#=> [ "a", "b", "c", "d", "e", "f" ]
puts a.inspect                                     #=> [ "a", "b", "c" ]
```

Array Difference

arr - other_array → new array

Returns a new array that is a copy of the original array, removing any items that also appear in other_array. If you need set like behaviour, see the library class Set on page 710.

```
x                                                  = [ 1, 1, 2, 2, 3, 3, 4, 5 ]
y                                                  = [1, 2, 4 ]
puts ( x - y).inspect                              #=> [3, 3, 5]
```

Append

arr << obj → arr

Pushes the given object on to the end of this array. This expression returns the array itself, so several appends may be chained together. See also Array#push.

```
puts ([ 1, 2 ] << "c" << "d" << [ 3, 4 ]).inspect  #=> [1, 2, "c", "d", [3, 4]]
```

Comparison

arr <=> other_array → −1, 0, +1

Returns an integer −1, 0, or +1 if this array is less than, equal to, or greater than other_array. Each object in each array is compared (using <=>). If any value isn't equal, then that inequality is the return value. If all the values found are equal, then the return is based on a comparison of the array lengths. Thus, two arrays are "equal" according to Array#<=> if and only if they have the same length and the value of each element is equal to the value of the corresponding element in the other array.

```
puts ([ "a", "a", "c" ]    <=> [ "a", "b", "c" ]).inspect #=> -1
puts ([ 1, 2, 3, 4, 5, 6 ] <=> [ 1, 2 ]).inspect          #=> 1
```

Equality

arr == obj → true or false

Two arrays are equal if they contain the same number of elements and if each element is equal to (according to Object#==) the corresponding element in the other array. If obj is not an array, attempt to convert it using to_ary and return obj==arr.

```
puts ([ "a", "c" ]    == [ "a", "c", 7 ]).inspect   #=> false
puts ([ "a", "c", 7 ] == [ "a", "c", 7 ]).inspect   #=> true
puts ([ "a", "c", 7 ] == [ "a", "d", "f" ]).inspect #=> false
```

Element Reference

arr[int] → obj or nil

arr[start, length] → an_array or nil

arr[range] → an_array or nil

Returns the element at index int, returns a subarray starting at index start and continuing for length elements, or returns a subarray specified by range. Negative indices count backward from the end of the array (–1 is the last element). Returns nil if the index of the first element selected is greater than the array size. If the start index equals the array size and a length or range parameter is given, an empty array is returned. Equivalent to Array#slice.

```
a                                          = [ "a", "b", "c", "d", "e" ]
puts (a[2] + a[0] + a[1]).inspect          #=> "cab"
puts a[6].inspect                          #=> nil
puts a[1, 2]                               #=> ["b", "c"]
puts a[1..3]                               #=> ["b", "c", "d"]
puts a[4..7]                               #=> ["e"]
puts a[6..10]                              #=> nil
puts a[3, 3]                               #=> ["d", "e"]
```

special cases:

```
puts a[5].inspect                          #=> nil
puts a[5, 1].inspect                       #=> []
puts a[5..10].inspect                      #=> []
```

Element Assignment

arr[int] = obj → obj

arr[start, length] = obj → obj

arr[range] = obj → obj

Sets the element at index int, replaces a subarray starting at index start and continuing for length elements, or replaces a subarray specified by range. If int is greater than the current capacity of the array, the array grows automatically. A negative int will count backward from the end of the array. Inserts elements if length is zero. If obj is nil, deletes elements from arr. If obj is an array, the form with the single index will insert that array into arr, and the forms with a length or with a range will replace the given elements in arr with the array contents. An IndexError is raised if a negative index points past the beginning of the array. See also Array#push and Array#unshift.

```
a       = Array.new;        puts a.inspect #=> [] # a is an empty array
a[4]    = "4";              puts a.inspect #=> [nil, nil, nil, nil, "4"]
a[0]    = [ 1, 2, 3 ];      puts a.inspect #=> [[1, 2, 3], nil, nil, nil, "4"]
a[0, 3] = [ 'a', 'b', 'c' ];puts a.inspect #=> ["a", "b", "c", nil, "4"]
a[1..2] = [ 1, 2 ];         puts a.inspect #=> ["a", 1, 2, nil, "4"]
a[0, 2] = "?";              puts a.inspect #=> ["?", 2, nil, "4"]
a[0..2] = "A";              puts a.inspect #=> ["A", "4"]
```

```
a[1]    = "Z";              puts a.inspect #=> ["A", "Z"]
a[1..1] = nil;              puts a.inspect #=> ["A", nil]
```

Set Union

arr | other_array → an_array

Returns a new array by joining this array with other_array, removing duplicates. The rules for comparing elements are the same as for hash keys. If you need setlike behavior, see the library class Set on page 710.

```
puts ([ "a", "b", "c" ] | [ "c", "d", "a" ]).inspect #=> ["a", "b", "c", "d"]
```

arr.assoc(obj) → an_array or nil

Searches through an array whose elements are also arrays comparing obj with the first element of each contained array using obj.== . Returns the first contained array that matches (that is, the first associated array) or nil if no match is found. See also Array#rassoc.

```
s1                                    = [ "colors", "red", "blue", "green" ]
s2                                    = [ "letters", "a", "b", "c" ]
s3                                    = "foo"
a                                     = [ s1, s2, s3 ]
puts a.assoc("letters").inspect     #=> ["letters", "a", "b", "c"]
puts a.assoc("foo").inspect         #=> nil
```

arr.at(int) → obj or nil

Returns the element at index int. A negative index counts from the end of arr. Returns nil if the index is out of range. See also Array#[]. Array#at is slightly faster than Array#[], as it does not accept ranges, and so on.

```
a                                           = [ "a", "b", "c", "d", "e" ]
puts a.at(0).inspect                        #=> "a"
puts a.at(1).inspect                        #=> "b"
```

arr.clear → arr

Removes all elements from arr.

```
a                                           = [ "a", "b", "c", "d", "e" ]
puts a.clear.inspect                        #=> []
```

arr.collect {| obj | block } → arr

Invokes block once for each element of arr, replacing the element with the value returned by block. See also Enumerable#collect.

```
a                                           = [ "a", "b", "c", "d" ]
```

```
puts a.collect {|x| x + "!" }.inspect                 #=> ["a!", "b!", "c!", "d!"]
puts a.inspect                                        #=> [ "a", "b", "c", "d" ]
```

arr.collect! {| obj | block } → arr

Invokes block once for each element of arr, replacing the element with the value returned by block. See also Enumerable#collect.

```
a                                                     = [ "a", "b", "c", "d" ]
puts a.collect! {|x| x + "!" }.inspect                #=> ["a!", "b!", "c!", "d!"]
puts a.inspect                                        #=> ["a!", "b!", "c!", "d!"]
```

arr.compact → an_array

Returns a copy of arr with all nil elements removed.

```
puts [ "a", nil, "b", nil, "c", nil ].compact.inspect #=> ["a", "b", "c"]
```

arr.compact! → arr or nil

Removes nil elements from arr. Returns nil if no changes were made.

```
puts [ "a", nil, "b", nil, "c" ].compact!).inspect #=> ["a", "b", "c"]
puts [ "a", "b", "c" ].compact!.inspect               #=> nil
```

arr.concat(other_array) → arr

Appends the elements in other_array to arr.

```
puts [ "a", "b" ].concat( ["c", "d"] ).inspect     #=> ["a", "b", "c", "d"]
```

arr.delete(obj) → obj or nil

arr.delete(obj) { block } → obj or nil

Deletes items from arr that are equal to obj. If the item is not found, returns nil. If the optional code block is given, returns the result of block if the item is not found.

```
a                                                     = [ "a", "b", "b", "b", "c" ]
puts a.delete("b").inspect                            #=> "b"
puts a.inspect                                        #=> ["a", "c"]
puts a.delete("z").inspect                            #=> nil
puts a.delete("z") { "not found" }.inspect            #=> "not found"
```

arr.delete_at(index) → obj or nil

Deletes the element at the specified index, returning that element, or nil if the index is out of range. See also

Array#slice!.

```
a                                                   = %w( ant bat cat dog )
puts a.delete_at(2).inspect                         #=> "cat"
puts a.inspect                                      #=> ["ant", "bat", "dog"]
puts a.delete_at(99).inspect                        #=> nil
```

arr.delete_if {| item| block } → arr

Deletes every element of arr for which block evaluates to true.

```
a                                                   = [ "a", "b", "c" ]
puts a.delete_if {|x| x >= "b" }.inspect            #=> ["a"]
```

arr.each {| item| block } → arr

Calls block once for each element in arr, passing that element as a parameter.

```
a                                                   = [ "a", "b", "c" ]
a.each {|x| print x, " -- "}                        #=> a -- b -- c --
```

arr.each_index {| index | block } → arr

Same as Array#each but passes the index of the element instead of the element itself.

```
a                                                   = [ "a", "b", "c" ]
puts a.each_index {|x| print x, " -- "}.inspect    #=> 0 -- 1 -- 2 --
```

arr.empty? → true or false

Returns true if arr array contains no elements.

```
puts [].empty?.inspect                              #=> true
puts [ 1, 2, 3 ].empty?.inspect                     #=> false
```

arr.eql?(other) → true or false

Returns true if arr and other are the same object or if other is an object of class Array with the same length and content as arr. Elements in the arrays are compared using Object#eql?. See also Array#<=>.

```
puts [ "a", "b", "c" ].eql?(["a", "b", "c"]).inspect #=> true
puts [ "a", "b", "c" ].eql?(["a", "b"]).inspect      #=> false
puts [ "a", "b", "c" ].eql?(["b", "c", "d"]).inspect #=> false
```

arr.fetch(index) → obj

arr.fetch(index, default) → obj

arr.fetch(index) {| i | block } → obj

Tries to return the element at position index. If the index lies outside the array, the first form throws an IndexError exception, the second form returns default, and the third form returns the value of invoking the block, passing in the index. Negative values of index count from the end of the array.

```
a                                          = [ 11, 22, 33, 44 ]
puts a.fetch(1).inspect                    #=> 22
puts a.fetch(3).inspect                    #=> 44
puts a.fetch(3,'cat').inspect              #=> 44
puts a.fetch(3) {|i| i*i }.inspect         #=> 44
puts a.fetch(4,'cat').inspect              #=> "cat"
puts a.fetch(4) {|i| i*i }.inspect         #=> 16
```

arr.fill(obj) → arr

arr.fill(obj, start h , length i) → arr

arr.fill(obj, range) → arr

arr.fill {| i | block } → arr

arr.fill(start h , length i) {| i | block } → arr

arr.fill(range) {| i | block } → arr

The first three forms set the selected elements of arr (which may be the entire array) to obj. A start of nil is equivalent to zero. A length of nil is equivalent to arr.length. The last three forms fill the array with the value of the block. The block is passed the absolute index of each element to be filled.

```
a                                          = [ "a", "b", "c", "d" ]
puts a.fill("x").inspect                   #=> ["x", "x", "x", "x"]
puts a.fill("z", 2, 2).inspect             #=> ["x", "x", "z", "z"]
puts a.fill("y", 0..1).inspect             #=> ["y", "y", "z", "z"]

puts a.fill {|i| i*i}.inspect              #=> [0, 1, 4, 9]
puts a.fill(1, 2) {|i| i/2}.inspect        #=> [0, 0, 1, 9]
puts a.fill(1..2) {|i| i+100}.inspect      #=> [0, 101, 102, 9]
```

arr.first → obj or nil

arr.first(count) → an_array

Returns the first element, or the first count elements, of arr. If the array is empty, the first form returns nil, and the second returns an empty array.

```
a                                          = [ "q", "r", "s", "t" ]
puts a.first.inspect                       #=> "q"
puts a.first(1).inspect                            #=> ["q"]
puts a.first(3).inspect                            #=> ["q", "r", "s"]
```

arr.flatten → an_array

Returns a new array that is a one-dimensional flattening of this array (recursively). That is, for every element that is an array, extract its elements into the new array.

```
s                                        = [ 1, 2, 3 ]
puts s.inspect                           #=> [1, 2, 3]
t                                        = [ 4, 5, 6, [7, 8] ]
puts t.inspect                           #=> [4, 5, 6, [7, 8]]
a                                        = [ s, t, 9, 10 ]
puts a.inspect                           #=> [[1, 2, 3], [4, 5, 6, [7, 8]], 9, 10]
puts a.flatten.inspect                   #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
puts a.inspect                           #=> [[1, 2, 3], [4, 5, 6, [7, 8]], 9, 10]
```

arr.flatten! → arr or nil

Same as Array#flatten but modifies the receiver in place. Returns nil if no modifications were made (i.e., arr contains no subarrays).

```
a                                                  = [ 1, 2, [3, [4, 5] ] ]
puts a.flatten!.inspect                            #=> [1, 2, 3, 4, 5]
puts a.flatten!.inspect                            #=> nil
puts a.inspect                                     #=> [1, 2, 3, 4, 5]
```

arr.include?(obj) → true or false

Returns true if the given object is present in arr (that is, if any object == obj), false otherwise.

```
a                                                  = [ "a", "b", "c" ]
puts a.include?("b").inspect                       #=> true
puts a.include?("z").inspect                       #=> false
```

arr.index(obj) → int or nil

Returns the index of the first object in arr that is == to obj. Returns nil if no match is found.

```
a                                                  = [ "a", "b", "c" ]
puts a.index("b").inspect                          #=> 1
puts a.index("z").inspect                          #=> nil
```

arr.indices(i1, i2, ... iN) → an_array

Deprecated; use Array#values_at.

arr.insert(index, obj+) → arr

If index is not negative, inserts the given values before the element with the given index. If index is −1, appends the values to arr. Otherwise inserts the values after the element with the given index.

```
a                                  = %w{ a b c d }
puts a.insert(2, 99).inspect       #=> ["a", "b", 99, "c", "d"]
puts a.insert(2, 1, 2, 3).inspect  #=> ["a", "b", 99, "c", 1, 2, 3, "d"]
puts a.insert(1, "e").inspect      #=> ["a", "b", 99, "c", 1, 2, 3, "d", "e"]
```

arr.join(separator=$,) → str

Returns a string created by concatenating each element of the array to a string, separating each by separator.

```
puts [ "a", "b", "c" ].join.inspect               #=> "abc"
puts [ "a", "b", "c" ].join(",").inspect          #=> "a,b,c"
```

arr.last → obj or nil

arr.last(count) → an_array

Returns the last element, or last count elements, of arr. If the array is empty, the first form returns nil, the second an empty array.

```
puts [ "w", "x", "y", "z" ].last.inspect          #=> "z"
puts [ "w", "x", "y", "z" ].last(1).inspect       #=> ["z"]
puts [ "w", "x", "y", "z" ].last(3).inspect       #=> ["x", "y", "z"]
```

arr.length → int

Returns the number of elements in arr. See also Array#nitems.

```
puts [ 1, nil, 3, nil, 5 ].length.inspect         #=> 5
```

arr.map! {| obj | block } → arr

Synonym for Array#collect!.

```
a                                                 = [ "a", "b", "c", "d" ]
puts a.map! {|x| x + "!" }).inspect               #=> ["a!", "b!", "c!", "d!"]
puts a.inspect                                    #=> ["a!", "b!", "c!", "d!"]
```

arr.nitems → int

Returns the number of non-nil elements in arr. See also Array#length.

```
puts [ 1, nil, 3, nil, 5 ].nitems.inspect         #=> 3
```

arr.pack (template) → binary_string

Packs the contents of arr into a binary sequence according to the directives in template (see Table 27.1 on the page before). Directives A, a, and Z may be followed by a count, which gives the width of the resulting field. The remaining directives also may take a count, indicating the number of array elements to convert. If the count is an asterisk (*), all remaining array elements will be converted. Any of the directives “sSiIlL” may be followed by an underscore (_) to use the underlying platform’s native size for the specified type; otherwise, they use a platform-independent size. Spaces are ignored in the template string. Comments starting with # to the next newline or end of string are also ignored. See also String#unpack on page 602.

```
a                                                  = [ "a", "b", "c" ]
n                                                  = [ 65, 66, 67 ]
puts a.pack("A3A3A3").inspect                      #=> "a  b  c  "
puts a.pack("a3a3a3").inspect                      #=> "a\000\000b\000\000c\000\000"
puts n.pack("ccc").inspect                         #=> "ABC"
```

Table 27.1. Template characters for Array#pack

Directive	Meaning
@	Moves to absolute position
A	ASCII string (space padded, count is width)
a	ASCII string (null padded, count is width)
B	Bit string (descending bit order)
b	Bit string (ascending bit order)
C	Unsigned char
c	Char
D, d	Double-precision float, native format
E	Double-precision float, little-endian byte order
e	Single-precision float, little-endian byte order
F, f	Single-precision float, native format
G	Double-precision float, network (big-endian) byte order
g	Single-precision float, network (big-endian) byte order
H	Hex string (high nibble first)
h	Hex string (low nibble first)
I	Unsigned integer

i Integer

L Unsigned long

l Long

M Quoted printable, MIME encoding (see RFC2045)

m Base64 encoded string

N Long, network (big-endian) byte order

n Short, network (big-endian) byte order

P Pointer to a structure (fixed-length string)

p Pointer to a null-terminated string

Q, q 64-bit number

1.8 S Unsigned short

s Short

U UTF-8

u UU-encoded string

V Long, little-endian byte order

v Short, little-endian byte order

w BER-compressed integer note. 1

1.8

X Back up a byte

x Null byte

Z Same as A

note: 1 The octets of a BER-compressed integer represent an unsigned integer in base 128, most significant digit first, with as few digits as possible. Bit eight (the high bit) is set on each byte except the last (Self-Describing Binary Data Representation, MacLeod)

arr.pop → obj or nil

Removes the last element from arr and returns it or returns nil if the array is empty.

```
a                                                          = [ "a", "m", "z" ]
puts a.pop.inspect                                         #=> "z"
puts a.inspect                                             #=> ["a", "m"]
```

arr.push(obj*) → arr

Appends the given argument(s) to arr.

```
a                                     = [ "a", "b", "c" ]
puts a.push("d", "e", "f").inspect  #=> ["a", "b", "c", "d", "e", "f"]
```

arr.rassoc(key) → an_array or nil

Searches through the array whose elements are also arrays. Compares key with the second element of each contained array using ==. Returns the first contained array that matches. See also Array#assoc.

```
a = [ [ 1, "one"], [2, "two"], [3, "three"], ["ii", "two"] ]
puts a.rassoc("two").inspect                   #=> [2, "two"]
puts a.rassoc("four").inspect                  #=> nil
```

arr.reject! { block } item → arr or nil

Equivalent to Array#delete_if, but returns nil if no changes were made. Also see Enumerable#reject.

```
a                                                  = [ "a", "b", "c" ]
puts a.reject! {|x| x >= "b" }).inspect          #=> ["a"]
puts a.reject! {|x| x >= "b" }).inspect          #=> nil
```

arr.replace(other_array) → arr

Replaces the contents of arr with the contents of other_array, truncating or expanding if necessary.

```
a                                                  = [ "a", "b", "c", "d", "e" ]
puts a.replace([ "x", "y", "z" ]).inspect          #=> ["x", "y", "z"]
puts a.inspect                                     #=> ["x", "y", "z"]
```

arr.reverse → new array

Returns a new array using arr's elements in reverse order.

```
puts [ "a", "b", "c" ].reverse.inspect             #=> ["c", "b", "a"]
puts [ 1 ].reverse.inspect                         #=> [1]
```

arr.reverse! → arr

Reverses arr in place.

```
a                                                  = [ "a", "b", "c" ]
puts a.reverse!.inspect                            #=> ["c", "b", "a"]
puts a.inspect                                     #=> ["c", "b", "a"]
puts [ 1 ].reverse!.inspect                        #=> [1]
```

arr.reverse_each {| item| block } → arr

Same as Array#each, but traverses arr in reverse order.

```
a                                                   = [ "a", "b", "c" ]
puts a.reverse_each {|x| print x, " " }.inspect     #=> c b a
```

arr.rindex(obj) → int or nil

Returns the index of the last object in arr such that the object == obj. Returns nil if no match is found.

```
a                                                   = [ "a", "b", "b", "b", "c" ]
puts a.rindex("b").inspect                          #=> 3
puts a.rindex("z").inspect                          #=> nil
```

arr.shift → obj or nil

Returns the first element of arr and removes it (shifting all other elements down by one). Returns nil if the array is empty.

```
args                                                = [ "m", "q", "filename" ]
puts args.shift.inspect                             #=> "m"
puts args.inspect                                   #=> ["q", "filename"]
```

arr.size → int

Synonym for Array#length.

arr.slice(int) → obj

arr.slice(start, length) → an_array

arr.slice(range) → an_array

Synonym for Array#[ ].

```
a                                                      = [ "a", "b", "c", "d", "e" ]
puts (a.slice(2) + a.slice(0) + a.slice(1)).inspect #=> "cab"
puts a.slice(6).inspect                                #=> nil
puts a.slice(1, 2).inspect                             #=> ["b", "c"]
puts a.slice(1..3).inspect                             #=> ["b", "c", "d"]
puts a.slice(4..7).inspect                             #=> ["e"]
puts a.slice(6..10).inspect                            #=> nil
puts a.slice(3, 3).inspect                             #=> ["d", "e"]
```

special cases:

```
puts a.slice(5).inspect                             #=> nil
puts a.slice(5, 1).inspect                            #=> []
puts a.slice(5..10).inspect                           #=> []
```

arr.slice!(int) → obj or nil

arr.slice!(start, length) → an_array or nil

arr.slice!(range) → an_array or nil

Deletes the element(s) given by an index (optionally with a length) or by a range. Returns the deleted object, subarray, or nil if the index is out of range. Inside Array class is equivalent to:

```
def slice!(*args)
  result = self[*args]
  self[*args] = nil
  result
end

a                                                    = [ "a", "b", "c" ]
puts a.slice!(1).inspect                             #=> "b"
puts a.inspect                                       #=> ["a", "c"]
puts a.slice!(1).inspect                             #=> "c"
puts a.inspect                                       #=> ["a"]
puts a.slice!(100).inspect                           #=> nil
puts a.inspect                                       #=> ["a"]
```

arr.sort → new array

arr.sort {| a,b | block } → new array

Returns a new array created by sorting arr. Comparisons for the sort will be done using the <=> operator or using an optional code block. The block implements a comparison between a and b, returning −1, 0, or +1. See also Enumerable#sort_by.

```
a                                                    = [ "d", "a", "e", "c", "b" ]
puts a.sort.inspect                                  #=> ["a", "b", "c", "d", "e"]
puts a.sort {|x,y| y <=> x }.inspect                 #=> ["e", "d", "c", "b", "a"]
puts a.inspect                                       #=> ["d", "a", "e", "c", "b"]
```

arr.sort! → arr

arr.sort! {| a,b | block } → arr

Sorts arr in place (see Array#sort). arr is effectively frozenwhile a sort is in progress.

```
a                                                    = [ "d", "a", "e", "c", "b" ]
puts a.sort!.inspect                                 #=> ["a", "b", "c", "d", "e"]
puts a.inspect                                    #=> ["a", "b", "c", "d", "e"]
```

arr.to_a → arr

array_subclass.to_a → new array

If arr is an array, returns arr. If arr is a subclass of Array, invokes to_ary, and uses the result to create a new array object

```
arg                                               = [ "a", "e", "i", "o" ]
arg.object_id
arg2                                              = arg.to_a
arg2.object_id

class MyArray < Array ; end
ma                                                = MyArray.new arg
puts ma.object_id
puts ma2                                          = ma.to_a
puts ma2.object_id
puts ma2.inspect                                  #=> ["a", "e", "i", "o"]
```

arr.to_ary → arr

Returns arr.

```
arg                                                = [ "a", "e", "i", "o" ]
puts arg.object_id
arg2                                               = arg.to_ary
puts arg2.object_id
puts arg2.inspect                                  #=> ["a", "e", "i", "o"]
```

arr.to_s → str

Returns arr.join.

```
puts [ "a", "e", "i", "o" ].to_s.inspect #=> "[\"a\", \"e\", \"i\", \"o\"]"
```

arr.transpose → an_array

Assumes that arr is an array of arrays and transposes the rows and columns.

```
a                                                  = [[1,2], [3,4], [5,6]]
puts a.transpose.inspect                           #=> [[1, 3, 5], [2, 4, 6]]
```

arr.uniq → an_array

Returns a new array by removing duplicate values in arr, where duplicates are detected by comparing using eql?.

```
a                                                  = [ "a", "a", "b", "b", "c" ]
puts a.uniq.inspect                                #=> ["a", "b", "c"]
```

arr.uniq! → arr or nil

Same as Array#uniq, but modifies the receiver in place. Returns nil if no changes are made (that is, no duplicates are found).

```
a                                                  = [ "a", "a", "b", "b", "c" ]
puts a.uniq!.inspect                               #=> ["a", "b", "c"]
b                                                  = [ "a", "b", "c" ]
puts b.uniq!.inspect                               #=> nil
```

arr.unshift(obj+) → arr

Prepends object(s) to arr.

```
a                                                   = [ "b", "c", "d" ]
puts a.unshift("a")).inspect                        #=> ["a", "b", "c", "d"]
puts a.unshift(1, 2)).inspect                       #=> [1, 2, "a", "b", "c", "d"]
```

arr.values_at(selector*) → an_array

Returns an array containing the elements in arr corresponding to the given selector(s). The selectors may be either integer indices or ranges.

```
a                                       = %w{ a b c d e f }
puts a.values_at(1, 3, 5).inspect       #=> ["b", "d", "f"]
puts a.values_at(1, 3, 5, 7).inspect    #=> ["b", "d", "f", nil]
puts a.values_at(1, 3, 5, 7).inspect    #=> ["f", "d", "b", nil]
puts a.values_at(1..3, 2...5).inspect   #=> ["b", "c", "d", "c", "d", "e"]
```