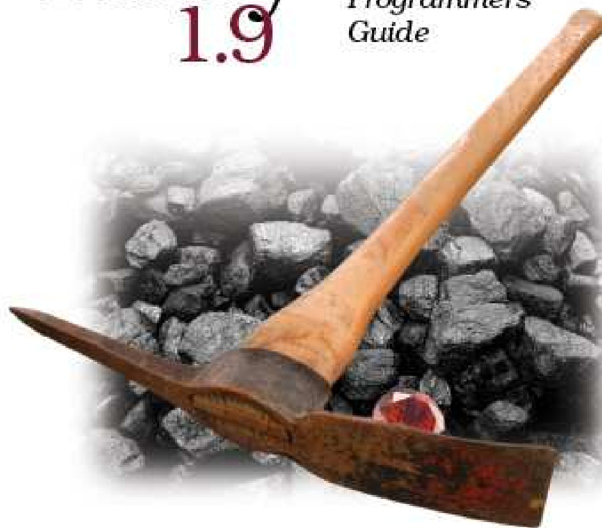
The Pragmatic Programmers

10th Anniversary

Updated for Ruby 1.9.2

Programming Ruby 1.9

The Pragmatic Programmers' Guide



Dave Thomas

with Chad Fowler and Andy Hunt

The Facets of Ruby Series

Extracted from:

Programming Ruby 1.9

The Pragmatic Programmers' Guide

This PDF file contains pages extracted from Programming Ruby 1.9, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2010 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Programming Ruby 1.9

The Pragmatic Programmers' Guide

Dave Thomas

with Chad Fowler
Andy Hunt

The Pragmatic Bookshelf
Raleigh, North Carolina   Dallas, Texas

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at http://www.pragprog.com.

Copyright © 2010 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-10: 1-934356-08-5

ISBN-13: 978-1-934356-08-1

Printed on acid-free paper.

3.0 printing, November 2010

Version: 2010-11-5

Class Array < Object

Relies on: each, <=>

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:

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, 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, zip

Class methods

[ ]

Array[⟨ obj ⟩*] → *an_array*

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

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

new

Array.new → *an_array*
Array.new (*size*=0, *obj*=nil) → *an_array*
Array.new(*array*) → *an_array*
Array.new(*size*) { |i| ... } → *an_array*

Returns a new array. In the first form, the new array is empty. In the second, it is created with *size* copies of *obj* (that is, *size* references to the same *obj*). The third form creates a copy of the array passed as a parameter (the array is generated by calling to_ary on the parameter). 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.

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

# only one instance of the default object is created
a = Array.new(2, Hash.new)
a[0]['cat'] = 'feline'
a          # => [{"cat"=>"feline"}, {"cat"=>"feline"}]
a[1]['cat'] = 'Felix'
a          # => [{"cat"=>"Felix"}, {"cat"=>"Felix"}]
```

CLICK HERE to purchase this book now.

```
a = Array.new(2) { Hash.new }  # Multiple instances
a[0]['cat'] = 'feline'
a          # => [{"cat"=>"feline"}, {}]

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

copy = Array.new(squares)      # initialized by copying
squares[5] = 25
squares    # => [0, 1, 4, 9, 16, 25]
copy       # => [0, 1, 4, 9, 16]
```

try_convert Array.try_convert(*obj*) → *an_array* or nil

If *obj* is not already an array, attempts to convert it to one by calling its to_ary method. Returns nil if no conversion could be made. *1.9*

```
class Stooges
  def to_ary
    [ "Larry", "Curly", "Moe" ]
  end
end
Array.try_convert(Stooges.new) # => ["Larry", "Curly", "Moe"]
Array.try_convert("Shemp")     # => nil
```

Instance methods

& *arr* & *other_array* → *an_array*

Set Intersection—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 setlike behavior, see the library class Set on page 837.

```
[ 1, 1, 3, 5 ] & [ 1, 2, 3 ] # => [1, 3]
```

***** *arr* * *int* → *an_array*
arr * *str* → *a_string*

Repetition—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*.

```
[ 1, 2, 3 ] * 3    # => [1, 2, 3, 1, 2, 3, 1, 2, 3]
[ 1, 2, 3 ] * "--" # => "1--2--3"
```

+ *arr* + *other_array* → *an_array*

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

```
[ 1, 2, 3 ] + [ 4, 5 ] # => [1, 2, 3, 4, 5]
```

CLICK HERE to purchase this book now.

- *arr* - *other_array* → *an_array*

Array Difference—Returns a new array that is a copy of the original array, removing any items that also appear in *other_array*. If you need setlike behavior, see the library class Set on page 837.

```
[ 1, 1, 2, 2, 3, 3, 4, 5 ] - [ 1, 2, 4 ] # => [3, 3, 5]
```

<< *arr* << *obj* → *arr*

Append—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.

```
[ 1, 2 ] << "c" << "d" << [ 3, 4 ] # => [1, 2, "c", "d", [3, 4]]
```

<=> *arr* <=> *other_array* → -1, 0, +1, or nil

Comparison—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. nil is returned if the argument is not comparable to *arr*.

```
[ "a", "a", "c" ]    <=> [ "a", "b", "c" ]  # => -1
[ 1, 2, 3, 4, 5, 6 ] <=> [ 1, 2 ]           # => 1
[ 1, 2, 3, 4, 5, 6 ] <=> "wombat"           # => nil
```

== *arr* == *obj* → *true* or *false*

Equality—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*.

```
[ "a", "c" ]    == [ "a", "c", 7 ]   # => false
[ "a", "c", 7 ] == [ "a", "c", 7 ]   # => true
[ "a", "c", 7 ] == [ "a", "d", "f" ] # => false
```

[ ] *arr*[*int*] → *obj* or nil
arr[*start*, *length*] → *an_array* or nil
arr[*range*] → *an_array* or nil

Element Reference—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

Click Here to purchase this book now.

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" ]
a[2] +  a[0] + a[1] # => "cab"
a[6]                # => nil
a[1, 2]             # => ["b", "c"]
a[1..3]             # => ["b", "c", "d"]
a[4..7]             # => ["e"]
a[6..10]            # => nil
a[-3, 3]            # => ["c", "d", "e"]
# special cases
a[5]                # => nil
a[5, 1]             # => []
a[5..10]            # => []
```

[ ]=

arr[*int*] = *obj* → *obj*
arr[*start*, *length*] = *obj* → *obj*}
arr[*range*] = *obj* → *obj*

Element Assignment—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 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. (Prior to Ruby 1.9, assigning nil with the second and third forms of element assignment could delete the corresponding array elements; now it simply assigns nil to them.) See also Array#push and Array#unshift. *1.9*

```
a = Array.new                  # => []
a[4] = "4";                  a # => [nil, nil, nil, nil, "4"]
a[0] = [ 1, 2, 3 ];          a # => [[1, 2, 3], nil, nil, nil, "4"]
a[0, 3] = [ 'a', 'b', 'c' ]; a # => ["a", "b", "c", nil, "4"]
a[1..2] = [ 1, 2 ];          a # => ["a", 1, 2, nil, "4"]
a[0, 2] = "?";               a # => ["?", 2, nil, "4"]
a[0..2] = "A", "B", "C";     a # => ["A", "B", "C", "4"]
a[-1]   = "Z";               a # => ["A", "B", "C", "Z"]
a[1..-1] = nil;              a # => ["A", nil]
```

|

arr | *other_array* → *an_array*

Set Union—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 837.

```
[ "a", "b", "c" ] | [ "c", "d", "a" ] # => ["a", "b", "c", "d"]
```

assoc *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 *asso*ciated 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 ]
a.assoc("letters") # => ["letters", "a", "b", "c"]
a.assoc("foo")     # => nil
```

at *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#[].

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

clear *arr*.clear → *arr*

Removes all elements from *arr*.

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

combination *arr*.combination(*size*) → *enumerator*
arr.combination(*size*) { | array | … } → *arr*

Constructs all combinations of the elements of *arr* of length *size*. If called with a block, passes each combination to that block; otherwise, returns an enumerator object. An empty result is generated if no combinations of the given length exist. See also Array#permutation. *1.9*

```
a = [ "a", "b", "c" ]
a.combination(1).to_a # => [["a"], ["b"], ["c"]]
a.combination(2).to_a # => [["a", "b"], ["a", "c"], ["b", "c"]]
a.combination(3).to_a # => [["a", "b", "c"]]
a.combination(4).to_a # => []
```

collect! *arr*.collect! { | obj | … } → *arr*

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

CLICK HERE to purchase this book now.

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

compact

arr.compact → *an_array*

Returns a copy of *arr* with all nil elements removed.

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

compact!

arr.compact! → *arr* or nil

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

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

concat

arr.concat(*other_array*) → *arr*

Appends the elements in *other_array* to *arr*.

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

count

arr.count(*obj*) → *int*
arr.count { |obj| ... } → *int*

Returns the count of objects in *arr* that equal *obj* or for which the block returns a true value. Returns an Enumerator if neither an argument nor a block is given (which seems strange...). Shadows the corresponding method in Enumerable. *1.9*

```
[1, 2, 3, 4].count(3)              # => 1
[1, 2, 3, 4].count {|obj| obj > 2 } # => 2
```

cycle

arr.cycle { |obj| ... } → nil or *enumerator*
arr.cycle(*times*) { |obj| ... } → nil or *enumerator*

Returns nil if *arr* has no elements; otherwise, passes the elements, one at a time to the block. When it reaches the end, it repeats. The number of times it repeats is set by the parameter. If the parameter is missing, cycles forever. Returns an Enumerator object if no block is given. *1.9*

```
[1,2,3].cycle(3)      # => #<Enumerator: [1, 2, 3]:cycle(3)>
[1,2,3].cycle(3).to_a # => [1, 2, 3, 1, 2, 3, 1, 2, 3]

columns = [ 1, 2, 3 ]
data    = %w{ a b c d e f g h }
```

```
columns.cycle do |column_number|
  print data.shift, "\t"
  break if data.empty?
  puts if column_number == columns.last
end
```

produces:

```
a       b       c
d       e       f
g       h
```

delete

arr.delete(*obj*) → *obj* or nil
arr.delete(*obj*) { . . . } → *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" ]
a.delete("b")                   # => "b"
a                               # => ["a", "c"]
a.delete("z")                   # => nil
a.delete("z") { "not found" }   # => "not found"
```

delete_at

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

delete_if

arr.delete_if { |item| . . . } → *arr*

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

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

each

arr.each { |item| . . . } → *arr*

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

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

produces:

```
a -- b -- c --
```

CLICK HERE to purchase this book now.

each_index *arr*.each_index { | index | ... } → *arr*

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

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

produces:

```
0 -- 1 -- 2 --
```

empty? *arr*.empty? → *true* or *false*

Returns true if *arr* array contains no elements.

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

eql? *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#<=>.

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

fetch *arr*.fetch(*index*) → *obj*
arr.fetch(*index*, *default*) → *obj*
arr.fetch(*index*) { | i | ... } → *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 ]
a.fetch(1)            # => 22
a.fetch(-1)           # => 44
a.fetch(-1, 'cat')    # => 44
a.fetch(4, 'cat')     # => "cat"
a.fetch(4) {|i| i*i } # => 16
```

CLICK HERE to purchase this book now.

fill

arr.fill(*obj*) → *arr*
arr.fill(*obj*, *start* ⟨ , *length* ⟩) → *arr*
arr.fill(*obj*, *range*) → *arr*
arr.fill { |i|... } → *arr*
arr.fill(*start* ⟨ , *length* ⟩) { |i|... } → *arr*
arr.fill(*range*) { |i|... } → *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" ]
a.fill("x")             # => ["x", "x", "x", "x"]
a.fill("z", 2, 2)       # => ["x", "x", "z", "z"]
a.fill("y", 0..1)       # => ["y", "y", "z", "z"]
a.fill {|i| i*i}        # => [0, 1, 4, 9]
a.fill(-3) {|i| i+100} # => [0, 101, 102, 103]
```

find_index

arr.find_index(*obj*) → *int* or nil
arr.find_index { |item|... } → *int* or nil
arr.find_index → *enumerator*

Returns the index of the first object in *arr* that is == to *obj* or for which the block returns a true value. Returns nil if no match is found. See also Enumerable#select and Array#rindex. *1.9*

```
a = [ "a", "b", "c", "b" ]
a.find_index("b")                 # => 1
a.find_index("z")                 # => nil
a.find_index {|item| item > "a"} # => 1
```

flatten

arr.flatten(*level* = -1) → *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, extracts its elements into the new array. The level parameter controls how deeply the flattening occurs. If less than zero, all subarrays are expanded. If zero, no flattening takes place. If greater than zero, only that depth of subarray is expanded. *1.9*

```
s = [ 1, 2, 3 ]           # => [1, 2, 3]
t = [ 4, 5, 6, [7, 8] ] # => [4, 5, 6, [7, 8]]
a = [ s, t, 9, 10 ]       # => [[1, 2, 3], [4, 5, 6, [7, 8]], 9, 10]
a.flatten(0)              # => [[1, 2, 3], [4, 5, 6, [7, 8]], 9, 10]
a.flatten                 # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
a.flatten(1)              # => [1, 2, 3, 4, 5, 6, [7, 8], 9, 10]
a.flatten(2)              # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

CLICK HERE to purchase this book now.

flatten! *arr*.flatten!(*level* = -1) → *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). *1.9*

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

frozen? *arr*.frozen? → *true* or *false*

Returns true if *arr* is frozen or if it is in the middle of being sorted. *1.9*

index *arr*.index(*obj*) → *int* or nil
arr.index { |item| ... } → *int* or nil

Synonym for Array#find_index. *1.9*

insert *arr*.insert(*index*, ⟨ *obj* ⟩+ }) → *arr*

If *index* is not negative, inserts the given values before the element with the given index. If *index* is negative, adds the values after the element with the given index (counting from the end).

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

join *arr*.join(*separator*=$,) → *str*

Returns a string created by converting each element of the array to a string and concatenating them, separated each by *separator*.

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

keep_if *arr*.keep_if { |obj| ... } → *array* or *enumerator*

Modifies *arr* by removing all elements for which *block* is false (see also Enumerable#select and Array.select!). Returns an Enumerator object if no block is given. *1.9.2*

```
a = [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
a.keep_if {|element| element < 6 } # => [1, 2, 3, 4, 5]
a                                  # => [1, 2, 3, 4, 5]
a.keep_if(&:odd?)                  # => [1, 3, 5]
a                                  # => [1, 3, 5]
```

CLICK HERE to purchase this book now.

Directive	Meaning
@	Move to absolute position
A	Sequence of bytes (space padded, count is width)
a	Sequence of bytes (null padded, count is width)
B	Bit string (descending bit order)
b	Bit string (ascending bit order)
C	Unsigned byte
c	Byte
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; by default adds linefeeds every 60 characters; "m0" suppresses linefeeds
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
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°
X	Back up a byte
x	Null byte
Z	Same as “a,” except a null byte is appended if the * modifier is given

° 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).

Figure 27.1: Template characters for Array.pack

CLICK HERE to purchase this book now.

last

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, and the second returns an empty array. (first is defined by Enumerable.)

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

length

arr.length → *int*

Returns the number of elements in *arr*.

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

map!

arr.map! { |obj| ... } → *arr*

Synonym for Array#collect!.

pack

arr.pack (*template*) → *binary_string*

Packs the contents of *arr* into a binary sequence according to the directives in *template* (see Figure 27.1, on the preceding page). 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 s S i I l L may be followed by an underscore (_) or bang (!) 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 716. *1.9*

```
a = [ "a", "b", "c" ]
n = [ 65, 66, 67 ]
a.pack("A3A3A3") # => "a␣␣b␣␣c␣␣"
a.pack("a3a3a3") # => "a\x00\x00b\x00\x00c\x00\x00"
n.pack("ccc")    # => "ABC"
```

permutation

arr.permutation(*size=arr.size*) → *enumerator*
arr.permutation(*size=arr.size*) { |array| ... } → *arr*}

Constructs all permutations of the elements of *arr* of length *size*. If called with a block, passes each permutation to that block; otherwise, returns an enumerator object. An empty result is generated if no permutations of the given length exist. See also Array#combination. *1.9*

CLICK HERE to purchase this book now.

```
words = {}
File.readlines("/usr/share/dict/words").map(&:chomp).each do |word|
  words[word.downcase] = 1
end

%w{ c a m e l }.permutation do |letters|
  anagram = letters.join
  puts anagram if words[anagram]
end
```

produces:

```
camel
clame
cleam
macle
```

pop *arr*.pop(⟨ n ⟩*) → *obj* or nil

Removes the last element (or the last *n* elements) from *arr*. Returns whatever is removed or nil if the array is empty. *1.9*

```
a = %w{ f r a b j o u s }
a.pop       # => "s"
a           # => ["f", "r", "a", "b", "j", "o", "u"]
a.pop(3)    # => ["j", "o", "u"]
a           # => ["f", "r", "a", "b"]
```

product *arr*.product(⟨ *arrays* ⟩*) → *result_array*
arr.product(⟨ *arrays* ⟩*) ⟨ { | combination | ... } ⟩ → *arr*

Generates all combinations of selecting an element each from *arr* and from any arrays passed as arguments. The number of elements in the result is the product of the lengths of *arr* and the lengths of the arguments (so if any of these arrays is empty, the result will be an empty array). Each element in the result is an array containing n+1 elements, where n is the number of arguments. If a block is present, it will be passed each combination, and *arr* will be returned. *1.9* *1.9.2*

```
suits = %w{ C D H S }
ranks = [ *2..10, *%w{ J Q K A } ]
card_deck = suits.product(ranks).shuffle
card_deck.first(13) # => [["H", 4], ["C", 10], ["D", "J"], ["S", 10], ["H", 9],
                    # .. ["S", 2], ["H", 6], ["C", 6], ["C", 9], ["D", 6], ["D",
                    # .. 3], ["D", 9], ["D", 10]]
```

push *arr*.push(⟨ *obj* ⟩*) → *arr*

Appends the given argument(s) to *arr*.

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

CLICK HERE to purchase this book now.

rassoc *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"] ]
a.rassoc("two")  # => [2, "two"]
a.rassoc("four") # => nil
```

reject! *arr*.reject! { | item | ... } → *arr* or nil

Equivalent to Array#delete_if but returns nil if *arr* is unchanged. Also see Enumerable#reject.

repeated_combination *arr*.repeated_combination(*length*) { | comb | ... } → *arr*
arr.repeated_combination(*length*) → *enum*

Creates the set of combinations of length *length* of the elements of *arr*. If *length* is greater than *arr*.size, elements will be allowed to repeat. Passes each combination to the block, or returns an enumerator if no block is given. *1.9.2*

```
a = [1, 2, 3]
a.repeated_combination(2).to_a # => [[1, 1], [1, 2], [1, 3], [2, 2], [2, 3], [3,
                               # .. 3]]
a.repeated_combination(4).to_a # => [[1, 1, 1, 1], [1, 1, 1, 2], [1, 1, 1, 3],
                               # .. [1, 1, 2, 2], [1, 1, 2, 3], [1, 1, 3, 3],
                               # .. [1, 2, 2, 2], [1, 2, 2, 3], [1, 2, 3, 3],
                               # .. [1, 3, 3, 3], [2, 2, 2, 2], [2, 2, 2, 3],
                               # .. [2, 2, 3, 3], [2, 3, 3, 3], [3, 3, 3, 3]]
```

repeated_permutation *arr*.repeated_permutation(*length*) { | comb | ... } → *arr*
arr.repeated_permutation(*length*) → *enum*

Creates the set of permutations of length *length* of the elements of *arr*. If *length* is greater than *arr*.size elements will be allowed to repeat. Passes each permutation to the block, or returns an enumerator if no block given. *1.9.2*

```
a = [:a, :b]
a.repeated_combination(2).to_a # => [[:a, :a], [:a, :b], [:b, :b]]
a.repeated_combination(3).to_a # => [[:a, :a, :a], [:a, :a, :b], [:a, :b, :b],
                               # .. [:b, :b, :b]]
```

replace *arr*.replace(*other_array*) → *arr*

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

CLICK HERE to purchase this book now.

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

reverse

arr.reverse → *an_array*

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

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

reverse!

arr.reverse! → *arr*

Reverses *arr* in place.

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

reverse_each

arr.reverse_each ⟨ { | item | ... } ⟩ } → *arr*

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

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

produces:

```
c b a
```

rindex

arr.rindex(*obj*) → *int* or nil
arr.rindex { | item | ... } → *int* or nil

Returns the index of the last object in *arr* that is == to *obj* or for which the block returns a true value. Returns nil if no match is found. See also Enumerable#select and Array#index. *1.9*

```
a = [ "a", "b", "e", "b", "d" ]
a.rindex("b")                    # => 3
a.rindex("z")                    # => nil
a.rindex {|item| item =~ /[aeiou]/} # => 2
```

rotate

arr.rotate(*places*=1) → *new_array*

Returns a new array containing the elements of *arr* rotated *places* positions (so that the element that originally was at *arr*[*places*] is now at the front of the array. *places* may be negative. *1.9.2*

```
a = [1, 2, 3, 4, 5]
a.rotate(2)  # => [3, 4, 5, 1, 2]
a.rotate(-2) # => [4, 5, 1, 2, 3]
```

CLICK HERE to purchase this book now.

rotate! *arr*.rotate(*places*=1) → *arr*

Rotate *arr* in place. *1.9.2*

sample *arr*.sample(*n*=1) → *an_array* or nil

Returns min(*n*, *arr*.size) random elements from *arr* or nil if *arr* is empty and no argument is given. *1.9*

```
a = [ "a", "b", "c", "d" ]
a.sample    # => "c"
a.sample(3) # => ["a", "d", "b"]
a.sample(6) # => ["b", "c", "d", "a"]
b = []
b.sample    # => nil
```

select! *arr*.select! { |obj| ... } → *array*, nil, or *enumerator*

Modifies *arr* by removing all elements for which *block* is false (see also Enumerable#select and Array#keep_if). Returns nil if no changes were made, returns an Enumerator object if no block is given, or returns *arr*. *1.9.2*

```
a = [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
a.select! {|element| element < 6 } # => [1, 2, 3, 4, 5]
a                                  # => [1, 2, 3, 4, 5]
a.select! {|element| element < 8 } # => nil
a                                  # => [1, 2, 3, 4, 5]
```

shift *arr*.shift(n = 1) → *obj* or nil

Returns the first *n* elements (or the first element with no argument) of *arr* and removes it (shifting all other elements down by one). Returns nil if the array is empty. *1.9*

```
args = [ "-m", "-q", "-v", "filename" ]
args.shift    # => "-m"
args.shift(2) # => ["-q", "-v"]
args          # => ["filename"]
```

shuffle *arr*.shuffle → *an_array*

Returns an array containing the elements of *arr* in random order. *1.9*

```
[ 1, 2, 3, 4, 5 ].shuffle # => [2, 5, 1, 4, 3]
```

CLICK HERE to purchase this book now.

shuffle! *arr*.shuffle! → *arr*

Randomizes the order of the elements of *arr*. *1.9*

size *arr*.size → *int*

Synonym for Array#length.

slice

arr.slice(*int*) → *obj*
arr.slice(*start*, *length*) → *an_array*
arr.slice(*range*) → *an_array*

Synonym for Array#[ ].

```
a = [ "a", "b", "c", "d", "e" ]
a.slice(2) + a.slice(0) + a.slice(1) # => "cab"
a.slice(6)                           # => nil
a.slice(1, 2)                        # => ["b", "c"]
a.slice(1..3)                        # => ["b", "c", "d"]
a.slice(4..7)                        # => ["e"]
a.slice(6..10)                       # => nil
a.slice(-3, 3)                       # => ["c", "d", "e"]
# special cases
a.slice(5)                           # => nil
a.slice(5, 1)                        # => []
a.slice(5..10)                       # => []
```

slice!

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.

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

sort!

arr.sort! → *arr*
arr.sort! { | a,b | ... } → *arr*

Sorts *arr* in place (see Enumerable#sort). *arr* is effectively frozen while a sort is in progress.

CLICK HERE to purchase this book now.

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

sort_by!

arr.sort_by! { | a | ... } → *arr*
arr.sort_by! → *enum*

Sorts *arr* in place (see Enumerable#sort_by). *arr* is effectively frozen while a sort is in progress. *1.9.2*

```
a = [ 5, 2, 7, 4, 8, 9 ]
# Sort even numbers before odd, and then by rank
a.sort_by! {|e| [ e & 1, e ] } # => [2, 4, 8, 5, 7, 9]
a                              # => [2, 4, 8, 5, 7, 9]
```

to_a

arr.to_a → *arr*
array_subclass.to_a → *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.

to_ary

arr.to_ary → *arr*

Returns *arr*.

to_s

arr.to_s → *str*

Returns a string representation of *arr*. (Prior to Ruby 1.9, this representation was the same as *arr*.join. Now it is the array as a literal.) *1.9*

```
[ 1, 3, 5, 7, 9 ].to_s # => "[1, 3, 5, 7, 9]"
```

transpose

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

uniq

arr.uniq ⟨ { | element | ... } ⟩ → *an_array*

Returns a new array by removing duplicate values in *arr*, where duplicates are detected by comparing using eql? and hash. If the block is present, the comparisons are made based on the values returned by that block for each element in the array. *1.9.2*

CLICK HERE to purchase this book now.

```
a = %w{ C a a b b A c a }
a.uniq                               # => ["C", "a", "b", "A", "c"]
a.uniq {|element| element.downcase } # => ["C", "a", "b"]
a.uniq(&:upcase)                     # => ["C", "a", "b"]
```

uniq! *arr*.uniq! ⟨ { | element | ... } ⟩ → *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). *1.9.2*

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

unshift *arr*.unshift(⟨ *obj* ⟩+ }) → *arr*

Prepends object(s) to *arr*.

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

values_at *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 }
a.values_at(1, 3, 5)         # => ["b", "d", "f"]
a.values_at(1, 3, 5, 7)      # => ["b", "d", "f", nil]
a.values_at(-1, -3, -5, -7) # => ["f", "d", "b", nil]
a.values_at(1..3, 2...5)     # => ["b", "c", "d", "c", "d", "e"]
```

CLICK HERE to purchase this book now.