RUBY ARRAYS

Ruby arrays are ordered, integer-indexed collections of any object. Each element in an array is associated with and referred to by an index.

Array indexing starts at 0, as in C or Java. A negative index is assumed 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.

Ruby arrays can hold objects such as String, Integer, Fixnum, Hash, Symbol, even other Array objects. Ruby arrays are not as rigid as arrays in other languages. Ruby arrays grow automatically while adding elements to them.

Creating Arrays:

There are many ways to create or initialize an array. One way is with the *new* class method:

```
names = Array.new
```

You can set the size of an array at the time of creating array:

```
names = Array.new(20)
```

The array *names* now has a size or length of 20 elements. You can return the size of an array with either the size or length methods:

```
#!/usr/bin/ruby

names = Array.new(20)
puts names.size   # This returns 20
puts names.length # This also returns 20
```

This will produce the following result:

```
20
20
```

You can assign a value to each element in the array as follows:

```
#!/usr/bin/ruby

names = Array.new(4, "mac")

puts "#{names}"
```

This will produce the following result:

```
macmacmacmac
```

You can also use a block with new, populating each element with what the block evaluates to:

```
#!/usr/bin/ruby

nums = Array.new(10) { |e| e = e * 2 }

puts "#{nums}"
```

This will produce the following result:

```
024681012141618
```

There is another method of Array, []. It works like this:

```
nums = Array.[](1, 2, 3, 4,5)
```

One more form of array creation is as follows :

```
nums = Array[1, 2, 3, 4,5]
```

The *Kernel* module available in core Ruby has an Array method, which only accepts a single argument. Here, the method takes a range as an argument to create an array of digits:

```
#!/usr/bin/ruby

digits = Array(0..9)

puts "#{digits}"
```

This will produce the following result:

```
0123456789
```

Array Built-in Methods:

We need to have an instance of Array object to call a Array method. As we have seen, following is the way to create an instance of Array object:

```
Array.[](...) [or] Array[...] [or] [...]
```

This will return a new array populated with the given objects. Now, using created object, we can call any available instance methods. For example:

```
#!/usr/bin/ruby

digits = Array(0..9)

num = digits.at(6)

puts "#{num}"
```

This will produce the following result:

```
6
```

Following are the public array methods (Assuming *array* is an array object):

SN	Methods with Description
1	**array & other_array** Returns a new array containing elements common to the two arrays, with no duplicates.
2	**array * int [or] array * str** Returns a new array built by concatenating the int copies of self. With a String argument, equivalent to self.join(str).
3	**array + other_array**

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

4 **array - other_array**

Returns a new array that is a copy of the original array, removing any items that also appear in other_array.

5 **str <=> other_str**

Compares str with other_str, returning -1 (less than), 0 (equal), or 1 (greater than). The comparison is casesensitive.

6 **array | other_array**

Returns a new array by joining array with other_array, removing duplicates.

7 **array << obj**

Pushes the given object onto the end of array. This expression returns the array itself, so several appends may be chained together.

8 **array <=> other_array**

Returns an integer (-1, 0, or +1) if this array is less than, equal to, or greater than other_array.

9 **array == other_array**

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.

10 **array[index] [or] array[start, length] [or]**

array[range] [or] array.slice(index) [or]

array.slice(start, length) [or] array.slice(range)

Returns the element at *index*, or returns a subarray starting at *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 (or starting index) is out of range.

11 **array[index] = obj [or]**

array[start, length] = obj or an_array or nil [or]

array[range] = obj or an_array or nil

Sets the element at *index*, or replaces a subarray starting at *start* and continuing for *length* elements, or replaces a subarray specified by *range*. If indices are greater than the current capacity of the array, the array grows automatically. Negative indices will count backward from the end of the array. Inserts elements if *length* is zero. If *nil* is used in the second and third form, deletes elements from *self*.

12 **array.abbrev(pattern = nil)**

Calculates the set of unambiguous abbreviations for the strings in *self*. If passed a pattern

or a string, only the strings matching the pattern or starting with the string are considered.

13 **array.assoc(obj)**

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 or *nil* if no match is found.

14 **array.at(index)**

Returns the element at index. A negative index counts from the end of self. Returns nil if the index is out of range.

15 **array.clear**

Removes all elements from array.

16 **array.collect { |item| block } [or]**

array.map { |item| block }

Invokes block once for each element of *self*. Creates a new array containing the values returned by the block.

17 **array.collect! { |item| block } [or]**

array.map! { |item| block }

Invokes *block* once for each element of *self*, replacing the element with the value returned by *block*.

18 **array.compact**

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

19 **array.compact!**

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

20 **array.concat(other_array)**

Appends the elements in other_array to *self*.

21 **array.delete(obj) [or]**

array.delete(obj) { block }

Deletes items from *self* 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.

22 **array.delete_at(index)**

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

23 **array.delete_if { |item| block }**

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

24 **array.each { |item| block }**

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

25 **array.each_index { |index| block }**

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

26 **array.empty?**

Returns true if the self array contains no elements.

27 **array.eql?(other)**

Returns true if *array* and *other* are the same object, or are both arrays with the same content.

28

array.fetch(index) [or]

array.fetch(index, default) [or]

array.fetch(index) { |index| block }

Tries to return the element at position *index*. If *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 *block*, passing in *index*. Negative values of *index* count from the end of the array.

29

array.fill(obj) [or]

array.fill(obj, start [, length]) [or]

array.fill(obj, range) [or]

array.fill { |index| block } [or]

array.fill(start [, length]) { |index| block } [or]

array.fill(range) { |index| block }

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

30

array.first [or]

array.first(n)

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

31 **array.flatten**

Returns a new array that is a one-dimensional flattening of this array (recursively).

32 **array.flatten!**

Flattens *array* in place. Returns *nil* if no modifications were made. (array contains no subarrays.)

33 **array.frozen?**

Returns true if *array* is frozen (or temporarily frozen while being sorted).

34 **array.hash**

Compute a hash-code for array. Two arrays with the same content will have the same hash code

35 **array.include?(obj)**

Returns true if *obj* is present in *self*, false otherwise.

36 **array.index(obj)**

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

37 **array.indexes(i1, i2, ... iN) [or]**

array.indices(i1, i2, ... iN)

This methods is deprecated in latest version of Ruby so please use Array#values_at.

38 **array.indices(i1, i2, ... iN) [or]**

array.indexes(i1, i2, ... iN)

This methods is deprecated in latest version of Ruby so please use Array#values_at.

39 **array.insert(index, obj...)**

Inserts the given values before the element with the given *index* (which may be negative).

40 **array.inspect**

Creates a printable version of array.

41 **array.join(sep=$,)**

Returns a string created by converting each element of the array to a string, separated by *sep*.

42 **array.last [or] array.last(n)**

Returns the last element(s) of *self*. If array is *empty*, the first form returns *nil*.

43 **array.length**

Returns the number of elements in *self*. May be zero.

44

array.map { |item| block } [or]

array.collect { |item| block }

Invokes *block* once for each element of *self*. Creates a *new* array containing the values returned by the block.

45 **array.map! { |item| block } [or]**

array.collect! { |item| block }

Invokes *block* once for each element of *array*, replacing the element with the value returned by block.

46 **array.nitems**

Returns the number of non-nil elements in *self*. May be zero.

47 **array.pack(aTemplateString)**

Packs the contents of array into a binary sequence according to the directives in aTemplateString. 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 is still may be followed by an underscore (_) to use the underlying platform's native size for the specified type; otherwise, they use a platformindependent size. Spaces are ignored in the template string. (See templating Table below)

48 **array.pop**

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

49 **array.push(obj, ...)**

Pushes (appends) the given obj onto the end of this array. This expression returns the array itself, so several appends may be chained together.

50 **array.rassoc(key)**

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.

51 **array.reject { |item| block }**

Returns a new array containing the items *array* for which the block is not *true*.

52 **array.reject! { |item| block }**

Deletes elements from *array* for which the block evaluates to *true*, but returns *nil* if no changes were made. Equivalent to Array#delete_if.

53 **array.replace(other_array)**

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

54 **array.reverse**

Returns a new array containing array's elements in reverse order.

55 **array.reverse!**

Reverses *array* in place.

56 **array.reverse_each {|item| block }**

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

57 **array.rindex(obj)**

Returns the index of the last object in array == to obj. Returns *nil* if no match is found.

58 **array.select {|item| block }**

Invokes the block passing in successive elements from array, returning an array containing those elements for which the block returns a *true* value.

59 **array.shift**

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

60 **array.size**

Returns the length of *array* (number of elements). Alias for length.

61

array.slice(index) [or] array.slice(start, length) [or]

array.slice(range) [or] array[index] [or]

array[start, length] [or] array[range]

Returns the element at *index*, or returns a subarray starting at *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* (or starting index) are out of range.

62

array.slice!(index) [or] array.slice!(start, length) [or]

array.slice!(range)

Deletes the element(s) given by an *index* (optionally with a length) or by a *range*. Returns the deleted object, subarray, or *nil* if *index* is out of range.

63 **array.sort [or] array.sort { | a,b | block }**

Returns a new array created by sorting self.

64 **array.sort! [or] array.sort! { | a,b | block }**

Sorts self.

65 **array.to_a**

Returns *self*. If called on a subclass of *Array*, converts the receiver to an Array object.

66 **array.to_ary**

Returns self.

67 **array.to_s**

Returns self.join.

68 **array.transpose**

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

69 **array.uniq**

Returns a new array by removing duplicate values in *array*.

70 **array.uniq!**

Removes duplicate elements from *self*. Returns *nil* if no changes are made (that is, no duplicates are found).

71 **array.unshift(obj, ...)**

Prepends objects to the front of array, other elements up one.

72 **array.values_at(selector,...)**

Returns an array containing the elements in self corresponding to the given *selector* (one or more). The selectors may be either integer indices or ranges.

73 **array.zip(arg, ...) [or]**

array.zip(arg, ...){ | arr | block }

Converts any arguments to arrays, then merges elements of *array* with corresponding elements from each argument.

Array pack directives:

Following table lists pack directives for use with Array#pack.

Directive	Description
@	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 RFC 2045).
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.
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 \fnm.
X	Back up a byte.
x	Null byte.
Z	Same as a, except that null is added with *.

Example:

Try following example to pack various data.

```
a = [ "a", "b", "c" ]
n = [ 65, 66, 67 ]
puts a.pack("A3A3A3")   #=> "a  b  c  "
```

```
puts a.pack("a3a3a3")   #=> "a\000\000b\000\000c\000\000"
puts n.pack("ccc")       #=> "ABC"
```

This will produce the following result:

```
a  b  c
abc
ABC
```