# Arrays as a List...

```ruby
a = [5, 6, 7, 8]
```

# zero indexed
```
a[0]
a[1]
```

# no array bounds error in Ruby
```
a[10]
```

# get the length..
```
a.size
```

# negative indexes: count from the end of the array
```
a[-1]
a[-4]
a[-10]
```

# adding an element beyond the end of the array:
```
# fills in enough spaces with nil
# NOTE: arrays can dynamically grow and shrink

a[10] = 9
```

# of course, dynamically typed
```
a[3] = "hello world"
```

# append another array
```
```

---

# Arrays as a Tuple...

```
triple = [false, "hi", a[0] + 4]
```

# get last element
```
triple[2]
```

---

# Arrays as a Set...

```
s1 = [1, 2, 3]
s2 = [2, 3, 4, 5]
```

# union (distinct elements only)
```
c = s1 | s2
```

# intersection (joint elements only)
```
c = s1 & s2
```

# subtraction
```
c = s2 - s1
```

---

# Arrays as a Stack...

```
a = []
a.push 5
a.push 7
```

# note: returns nil, not an error
```
a.pop
```

---

# Arrays as a Queue...

```ruby
b = a + [true, false]
```

# Initializing Arrays

```ruby
# pick initial size at run-time (x == 20)
x = if a[1] < a[0] then 10 else 20 end
y = Array.new(x)
```

# better: initialized with a block (more on blocks soon)
```
z = Array.new(x) { 0 }
w = Array.new(x) { |i| i*i }
```

---

# Arrays as a Tuple...

```
triple = [false, "hi", a[0] + 4]
```

# get last element
```
triple[2]
```

---

# Arrays as a Set...

```
s1 = [1, 2, 3]
s2 = [2, 3, 4, 5]
```

# union (distinct elements only)
```
c = s1 | s2
```

# intersection (joint elements only)
```
c = s1 & s2
```

# subtraction
```
c = s2 - s1
```

---

# Arrays as a Stack...

```
a = []
a.push 5
a.push 7
```

# note: returns nil, not an error
```
a.pop
```

---

# Arrays as a Queue...

```
```
a.push 11
a.push 3
a.push 5

# take first element
a.shift

# skip to the head of the queue
a.unshift 42

# Array aliasing
# =======================================
a = [1, 2, 3, 4]
d and a both refer to the same array
d = a
da

# e refers to a new array with the same values as a / d
# (could also add an empty array)
# e = a + []
e

evidence
d[0]
a[0] = "new"
d[0]
a[0]

# note, copy is unchanged
e[0]

# Array slices
# =======================================
f = [2, 4, 6, 8, 10, 12, 14]
# slice, starting at element 2 and continuing for 4 elements
f[2, 4]

# can also assign this way, don’t need to use the same number of elements
f[2, 4] = [1, 1]
f

# Array Summary
# ==================================================================================
# *) Arrays are really a generic mapping from numeric indices to values
# *) Arrays in Ruby can be used for many purposes:
#   *) Lists
#   *) Tuples
#   *) Stacks
#   *) Queues
#   *) Sets
#   *) ... consult documentation for even more!.
# *) Very Dynamic
#   *) Many operations allowed that would be errors in other languages, ex:
#       *) Accessing / assigning out of bounds
#       *) Replacing subsets with subsets of a different length

# Array includes many methods that apply iterative functions
# *) Typically take a /block/ that is called for each element
# *) Blocks:
#   *) Almost closures
#   *) Similar to anonymous functions that can be passed to method calls
#     *) Have lexical scope, like closures (block body uses environment where block was defined)
#   *) However, cannot be assigned to variables
#   *) Can take 0 or more arguments
#   *) Used extensively in Ruby code
# *) Loops in Ruby:
#   *) Traditional loops such as /for/ and /while/ do exist
#   *) In Ruby, hardly ever used
#   *) Most "loops" can be implemented using a function that accept a block
#     *) Instead of writing a loop, go find a useful iterator

# puts just prints to console..
[1, 2, 5, 12].each{|i| puts i*i}
a.any? { |x| x > 7 }
a.all? { |x| x > 7 }
# implicit: are elements "true" (i.e., neither false nor nil)
a.all?

# Lexical Scope
#
# i = 7
[4, 6, 8].each { |x| if i > x then puts (x+1) end }

# Hashes in Ruby

# *) similar to arrays, but...
#   *) keys can be anything; strings and symbols common
#   *) unordered, since no natural way to order keys
#   *) different syntax to make them

# Constructing / Basic Lookups
#
# build an empty hash, could also use Hash.new
h1 = {}

# insert records (note: dynamic typing)
  h1["my key"] = "my value"
  h1[true] = 36

# get keys / values
h1["my key"]
h1.keys
h1.values

# other ways of constructing hashes:
  h2 = {"SML"=>1, "Racket"=>2, "Ruby"=>3}
  h2["SML"]

# Symbols are like strings, but faster/more efficient.
# Prefix with ":"#
# Often used with hashes.
  h3 = {:sml => 1, :racket => 2, :ruby => 3}

# Removing records
#
# h1.delete("my key")

# Iteration of key / value pairs
#
# puts like println

# semicolons are like newlines
# Note: two argument block
h2.each { [k, v] print k; print ":"; puts v }

# Hashes: other notes
#
# *) Methods with many possible arguments will sometimes take a hash instead and perform record lookups using symbols

# Ranges in Ruby

# Similar to an array of contiguous numbers, but more efficiently represented
1..1000000
# just has an upper bound and lower bound
# can be iterated over in much the same way as arrays
# very useful for processing over a sequence of contiguous numbers

# Turning into an array
(1..100).to_a

# Support many of the same iterative functions as array

# example using inject (ruby’s name for reduce)
(1..100).inject {|acc, e| acc + e}

# Can be used for array slicing

  a = Array.new(20) { |i| i**2+1.to_s }
  a[3..9]
# Note difference!
  a[3...9]
  a[3, 9]

# Similar methods => Duck typing

# Arrays, hashes, and ranges all have some methods others don’t
# E.g. keys and values
# But also have many of the same methods, particularly iterators
# Great for duck typing

def m a  
a.count { |x| x*x < 50 }
end

# duck typing in m
  m [3, 5, 7, 9]
  m [3..9]
# Separates "how to iterate" from "what to do"
#
# Comparing basic types
#
# Good style to:
# use ranges when you can
# use hashes when non-numeric keys better represent data

# Exploratory Programming Using Reflection
#
# Can determine the "type" (class) of an object by calling its /class/ method
5.class
[1, 2, 3].class

# Can determine an object's methods by calling its /methods/ method
[1, 2, 3].methods
[1, 2, 3].methods - 5.methods # throw out "boring" methods all objects have
[1, 2, 3].methods.sort - 5.methods # sort alphabetically to search easier

# Example Uses
#
# *) Useful for exploring the language while in the REPL
# *) Can branch depending on which methods are defined by an object
# *) Exploratory Programming => Write programs by fiddling around in the REPL and trying stuff out