```ruby
# This is a comment

=begin
This is a multiline comment
No-one uses them
You shouldn't either
=end

# First and foremost: Everything is an object.

# Numbers are objects

3.class #=> Fixnum

3.to_s #=> "3"


# Some basic arithmetic
1 + 1 #=> 2
8 - 1 #=> 7
10 * 2 #=> 20
35 / 5 #=> 7
2**5 #=> 32
5 % 3 #=> 2

# Bitwise operators
3 & 5 #=> 1
3 | 5 #=> 7
3 ^ 5 #=> 6

# Arithmetic is just syntactic sugar
# for calling a method on an object
1.+(3) #=> 4
10.* 5 #=> 50

# Special values are objects
nil # equivalent to null in other languages
true # truth
false # falsehood

nil.class #=> NilClass
true.class #=> TrueClass
false.class #=> FalseClass

# Equality
1 == 1 #=> true
2 == 1 #=> false

# Inequality
1 != 1 #=> false
2 != 1 #=> true

# apart from false itself, nil is the only other 'falsey' value
```

```ruby
!nil   #=> true
!false #=> true
!0     #=> false

# More comparisons
1 < 10 #=> true
1 > 10 #=> false
2 <= 2 #=> true
2 >= 2 #=> true

# Combined comparison operator
1 <=> 10 #=> -1
10 <=> 1 #=> 1
1 <=> 1 #=> 0

# Logical operators
true && false #=> false
true || false #=> true
!true #=> false

# There are alternate versions of the logical operators with much lower
# precedence. These are meant to be used as flow-control constructs to chain
# statements together until one of them returns true or false.

# `do_something_else` only called if `do_something` succeeds.
do_something() and do_something_else()
# `log_error` only called if `do_something` fails.
do_something() or log_error()


# Strings are objects

'I am a string'.class #=> String
"I am a string too".class #=> String

placeholder = 'use string interpolation'
"I can #{placeholder} when using double quoted strings"
#=> "I can use string interpolation when using double quoted strings"

# Prefer single quoted strings to double quoted ones where possible
# Double quoted strings perform additional inner calculations

# Combine strings, but not with numbers
'hello ' + 'world'  #=> "hello world"
'hello ' + 3 #=> TypeError: can't convert Fixnum into String
'hello ' + 3.to_s #=> "hello 3"

# Combine strings and operators
'hello ' * 3 #=> "hello hello hello "

# Append to string
'hello' << ' world' #=> "hello world"

# print to the output with a newline at the end
```

```
puts "I'm printing!"
#=> I'm printing!
#=> nil

# print to the output without a newline
print "I'm printing!"
#=> I'm printing! => nil

# Variables
x = 25 #=> 25
x #=> 25

# Note that assignment returns the value assigned
# This means you can do multiple assignment:

x = y = 10 #=> 10
x #=> 10
y #=> 10

# By convention, use snake_case for variable names
snake_case = true

# Use descriptive variable names
path_to_project_root = '/good/name/'
path = '/bad/name/'

# Symbols (are objects)
# Symbols are immutable, reusable constants represented internally by an
# integer value. They're often used instead of strings to efficiently convey
# specific, meaningful values

:pending.class #=> Symbol

status = :pending

status == :pending #=> true

status == 'pending' #=> false

status == :approved #=> false

# Arrays

# This is an array
array = [1, 2, 3, 4, 5] #=> [1, 2, 3, 4, 5]

# Arrays can contain different types of items

[1, 'hello', false] #=> [1, "hello", false]

# Arrays can be indexed
# From the front
array[0] #=> 1
array.first #=> 1
```

```ruby
array[12] #=> nil

# Like arithmetic, [var] access
# is just syntactic sugar
# for calling a method [] on an object
array.[] 0 #=> 1
array.[] 12 #=> nil

# From the end
array[-1] #=> 5
array.last #=> 5

# With a start index and length
array[2, 3] #=> [3, 4, 5]

# Reverse an Array
a=[1,2,3]
a.reverse! #=> [3,2,1]

# Or with a range
array[1..3] #=> [2, 3, 4]

# Add to an array like this
array << 6 #=> [1, 2, 3, 4, 5, 6]
# Or like this
array.push(6) #=> [1, 2, 3, 4, 5, 6]

# Check if an item exists in an array
array.include?(1) #=> true

# Hashes are Ruby's primary dictionary with keys/value pairs.
# Hashes are denoted with curly braces:
hash = { 'color' => 'green', 'number' => 5 }

hash.keys #=> ['color', 'number']

# Hashes can be quickly looked up by key:
hash['color'] #=> 'green'
hash['number'] #=> 5

# Asking a hash for a key that doesn't exist returns nil:
hash['nothing here'] #=> nil

# Since Ruby 1.9, there's a special syntax when using symbols as keys:

new_hash = { defcon: 3, action: true }

new_hash.keys #=> [:defcon, :action]

# Check existence of keys and values in hash
new_hash.key?(:defcon) #=> true
new_hash.value?(3) #=> true

# Tip: Both Arrays and Hashes are Enumerable
# They share a lot of useful methods such as each, map, count, and more

# Control structures

if true
  'if statement'
elsif false
  'else if, optional'
else
  'else, also optional'
end

for counter in 1..5
  puts "iteration #{counter}"
end
#=> iteration 1
#=> iteration 2
#=> iteration 3
#=> iteration 4
#=> iteration 5

# HOWEVER, No-one uses for loops.
# Instead you should use the "each" method and pass it a block.
# A block is a bunch of code that you can pass to a method like "each".
# It is analogous to lambdas, anonymous functions or closures in other
# programming languages.
#
# The "each" method of a range runs the block once for each element of the range.
# The block is passed a counter as a parameter.
# Calling the "each" method with a block looks like this:

(1..5).each do |counter|
  puts "iteration #{counter}"
end
#=> iteration 1
#=> iteration 2
#=> iteration 3
#=> iteration 4
#=> iteration 5

# You can also surround blocks in curly brackets:
(1..5).each { |counter| puts "iteration #{counter}" }

# The contents of data structures can also be iterated using each.
array.each do |element|
  puts "#{element} is part of the array"
end
hash.each do |key, value|
  puts "#{key} is #{value}"
end

# If you still need and index you can use "each_with_index" and define an index
# variable
array.each_with_index do |element, index|
  puts "#{element} is number #{index} in the array"
end

counter = 1
while counter <= 5 do
  puts "iteration #{counter}"
  counter += 1
end
#=> iteration 1
#=> iteration 2
#=> iteration 3
#=> iteration 4
#=> iteration 5

# There are a bunch of other helpful looping functions in Ruby,
# for example "map", "reduce", "inject", the list goes on. Map,
# for instance, takes the array it's looping over, does something
# to it as defined in your block, and returns an entirely new array.
array = [1,2,3,4,5]
doubled = array.map do |element|
  element * 2
end
puts doubled
#=> [2,4,6,8,10]
puts array
#=> [1,2,3,4,5]

grade = 'B'

case grade
when 'A'
  puts 'Way to go kiddo'
when 'B'
  puts 'Better luck next time'
when 'C'
  puts 'You can do better'
when 'D'
  puts 'Scraping through'
when 'F'
  puts 'You failed!'
else
  puts 'Alternative grading system, eh?'
end
#=> "Better luck next time"

# cases can also use ranges
grade = 82
case grade
when 90..100
  puts 'Hooray!'
when 80...90
  puts 'OK job'
else
  puts 'You failed!'
```

```
end
#=> "OK job"

# exception handling:
begin
  # code here that might raise an exception
  raise NoMemoryError, 'You ran out of memory.'
rescue NoMemoryError => exception_variable
  puts 'NoMemoryError was raised', exception_variable
rescue RuntimeError => other_exception_variable
  puts 'RuntimeError was raised now'
else
  puts 'This runs if no exceptions were thrown at all'
ensure
  puts 'This code always runs no matter what'
end

# Functions

def double(x)
  x * 2
end

# Functions (and all blocks) implicitly return the value of the last statement
double(2) #=> 4

# Parentheses are optional where the result is unambiguous
double 3 #=> 6

double double 3 #=> 12

def sum(x, y)
  x + y
end

# Method arguments are separated by a comma
sum 3, 4 #=> 7

sum sum(3, 4), 5 #=> 12

# yield
# All methods have an implicit, optional block parameter
# it can be called with the 'yield' keyword

def surround
  puts '{'
  yield
  puts '}'
end

surround { puts 'hello world' }

# {
# hello world
```

```ruby
# }


# You can pass a block to a function
# "&" marks a reference to a passed block
def guests(&block)
  block.call 'some_argument'
end

# You can pass a list of arguments, which will be converted into an array
# That's what splat operator ("*") is for
def guests(*array)
  array.each { |guest| puts guest }
end

# Define a class with the class keyword
class Human

  # A class variable. It is shared by all instances of this class.
  @@species = 'H. sapiens'

  # Basic initializer
  def initialize(name, age = 0)
    # Assign the argument to the "name" instance variable for the instance
    @name = name
    # If no age given, we will fall back to the default in the arguments list.
    @age = age
  end

  # Basic setter method
  def name=(name)
    @name = name
  end

  # Basic getter method
  def name
    @name
  end

  # The above functionality can be encapsulated using the attr_accessor method as follows
  attr_accessor :name

  # Getter/setter methods can also be created individually like this
  attr_reader :name
  attr_writer :name

  # A class method uses self to distinguish from instance methods.
  # It can only be called on the class, not an instance.
  def self.say(msg)
    puts msg
  end

  def species
    @@species
  end
end


# Instantiate a class
jim = Human.new('Jim Halpert')

dwight = Human.new('Dwight K. Schrute')

# Let's call a couple of methods
jim.species #=> "H. sapiens"
jim.name #=> "Jim Halpert"
jim.name = "Jim Halpert II" #=> "Jim Halpert II"
jim.name #=> "Jim Halpert II"
dwight.species #=> "H. sapiens"
dwight.name #=> "Dwight K. Schrute"

# Call the class method
Human.say('Hi') #=> "Hi"

# Variable's scopes are defined by the way we name them.
# Variables that start with $ have global scope
$var = "I'm a global var"
defined? $var #=> "global-variable"

# Variables that start with @ have instance scope
@var = "I'm an instance var"
defined? @var #=> "instance-variable"

# Variables that start with @@ have class scope
@@var = "I'm a class var"
defined? @@var #=> "class variable"

# Variables that start with a capital letter are constants
Var = "I'm a constant"
defined? Var #=> "constant"

# Class is also an object in ruby. So class can have instance variables.
# Class variable is shared among the class and all of its descendants.

# base class
class Human
  @@foo = 0

  def self.foo
    @@foo
  end

  def self.foo=(value)
    @@foo = value
  end
end

# derived class
```

```ruby
class Worker < Human
end

Human.foo # 0
Worker.foo # 0

Human.foo = 2 # 2
Worker.foo # 2

# Class instance variable is not shared by the class's descendants.

class Human
  @bar = 0

  def self.bar
    @bar
  end

  def self.bar=(value)
    @bar = value
  end
end

class Doctor < Human
end

Human.bar # 0
Doctor.bar # nil

module ModuleExample
  def foo
    'foo'
  end
end

# Including modules binds their methods to the class instances
# Extending modules binds their methods to the class itself

class Person
  include ModuleExample
end

class Book
  extend ModuleExample
end

Person.foo     # => NoMethodError: undefined method `foo' for Person:Class
Person.new.foo # => 'foo'
Book.foo       # => 'foo'
Book.new.foo   # => NoMethodError: undefined method `foo'

# Callbacks are executed when including and extending a module

module ConcernExample
```

```ruby
  def self.included(base)
    base.extend(ClassMethods)
    base.send(:include, InstanceMethods)
  end

  module ClassMethods
    def bar
      'bar'
    end
  end

  module InstanceMethods
    def qux
      'qux'
    end
  end
end

class Something
  include ConcernExample
end

Something.bar     # => 'bar'
Something.qux     # => NoMethodError: undefined method `qux'
Something.new.bar # => NoMethodError: undefined method `bar'
Something.new.qux # => 'qux'
```

Additional resources

- Learn Ruby by Example with Challenges - A variant of this reference with in-browser challenges.
- An Interactive Tutorial for Ruby - Learn Ruby through a series of interactive tutorials.
- Official Documentation
- Ruby from other languages
- Programming Ruby - An older free edition is available online.
- Ruby Style Guide - A community-driven Ruby coding style guide.
- Try Ruby - Learn the basic of Ruby programming language, interactive in the browser.