7.1 Literal constructors

Some built-in classes have ‘literal constructors’. This means that instances from this classes can be created with a special notation instead of a call to new(). These classes are:

String -> Quotation marks -> eg “new string”  
Symbol -> Leading colon -> eg :symbol  
Array -> Square brackets -> eg: [1,2,3,4]  
Hash -> Curly braces -> eg { “New York” => “NY”, “Oregon” => “OR”}  
Range -> Two or three dots -> eg 0..9 or 0…10  
Regexp -> Forward slashes -> eg /{[a-z]+}/  
Proc (lambda) -> Dash, arrow, parentheses, braces -> eg ->(x,y) { x \* y }

7.2 Syntactic Sugar

Things Ruby lets you do to make your code look nicer

Ruby has some syntactic sugar that make method calls look like an operator (see table 7.2 on page 189)

For instance: 1+ 2 instead of 1.+(2)

7.3 Dangerous/Destructive methods

So-called ‘bang’ methods are methods followed by an exclamation(!) point. They should have a no bang method counterpart. Most often the bang methods are considered ‘dangerous’ or ‘destructive’ because they change the receiver.

For instance:

str = “Hello”

str.upcase  
=> “Hello”  
str

=> “Hello”

str.upcase!  
=> “HELLO”  
str  
=> “HELLO”

7.4 To\_\* conversion methods

Built-in methods whose names consists of to\_ plus an indicator of a class to which the method converts an object:

to\_s (to string)

every Ruby object responds to to\_s, meaning it has a way of representing itself as a string.  
to\_sym (to symbol)  
to\_a (to array)

provides an array-like representation of objecs

counterpart is the \* (unarray or ‘splat’ method) which unwrappes the elements of an array into a ‘bare list’ (literal objects separated by comma’s).

to\_i (to integer)

Ruby doesn’t automatically convert from string to integer, if you want a number from a string, you have to convert it explicitly. The to\_i conversion value from string that have no reasonable integer equivalent will always be 0:

eg “Hello”.to\_i => 0

to\_f (to float) converts an integer to a floating point equivalent

7.5 Boolean states and objects, nil

true and false can either be objects or a state

7.5.1 as states (true and false as expressions):

every expression in Ruby is either true or false in a Boolean sense. They pass or don’t pass an if-test:

eg if “string” => true (strings are true)

100 > 50 => true

empty classes and methods are false because they evaluate to nil which has the Boolean value of false.

7.5.2 as objects (true and false as instances of TrueClass and FalseClass)

true and false are special objects as they are the only instances of respectively TrueClass and FalseClass.

The only objects that have a Boolean value of false are false and nil.

7.5.3 The special object nil

Special object as it is the only instance of the class NilClass

nil is sometimes difficult to grasp. You can think of it as an object that serves the purpose of representing absence and a state of being undetermined.

7.6 Object comparison techniques

You can compare Ruby objects for equality or for order.

7.6.1 Equality tests

tell you whether two objects are exactly the same object.

Positive equality-test methods: ==, eql? And equal?

eg.

a = Object.new

b = Object.new

a == b

=> false

7.6.2 Comparisons and the Comparable module

comparison methods: ==, !=, >, <, >= and <=

by including the Comparable method in your class and defining the spaceship operator method <=> you can use all of these comparison methods in your class.

Inside the ⬄ method you define what you mean by less than (return value -1-, equal to (return value 1) and greater than (return value 0)

Runtime inspections of objects’ capabilities

~inspection : objects showing string representations of themselves

~reflection : asking objects about the methods they can execute

7.7.1 Listing an object’s methods

through the methods method

You can call this method on a class, module and object (when you call .methods on an object you also get it’s singleton methods)

7.7.2 Querying class and module objects

you can ask a class or module for it’s instance methods:

String.instance\_methods.sort

Fine-grained queries, for instance which instance methods ( without the inherited ones) are defined for a particular class:

String.instance\_methods(false).sort

You can also query an object or class for its private, protected and public methods.