1. Unless : reverse if else.. ie first eval the else condition

unless i==9

print i

Same as if(i<9)

print i

2. Until:

reverse while

until i==9

print i

3. loop

create a loop using iterator

loop do

statements

break if ...

end

Also can use next similar to continue in java to skip entries.

4. for i in 1...5

goes from 1 to 4

two dots like 1..5

goes from 1 to 5

5. methods that return true or false end with ? In ruby

6. putting an exclamation mark after a method causes inplace string replace.ie, content of variable itself is modified.

7. Closing and opening braces can be replaced with do and end ..

8. For each loop.

array.each do |x|

....

end

9. Times keyword to repeat tasks..

14.times do

print "times"

end

10.Hash is the concept of maps in ruby. Hash.new creates empty map . And it encloses it in {}

Values are assigned usign => operator

iterate over hash using each needs two variables. |X,Y|..

If you have a hash with a default value, and you try to access a non-existent key, you get that default value.

h = Hash.new("nothing here")

puts h

# {}

puts h["kitty"]

# nothing here

frequencies = Hash.new(0) // Declare hash and give default value of zero

11. to\_s to convert to string ..

12. Methods are started as

def name(params)

....

end

13. [1, 2, 3, 4, 5].each { |i| puts i }

Here each is a function. And we are passing a block to it.. instead of braces can use do and end

14.**combined comparison operator**

The combined comparison operator looks like this: <=>. It returns 0 if the first **operand** (item to be compared) equals the second, 1 if first operand is *greater* than the second, and -1 if the first operand is *less* than the second.

15. def what\_up(greeting, \*bros)

bros is varargs here.. called as splat arguments

16. Cusotm sort.. an example of passing a block to a function

fruits.sort! do |f,s|

s<=>f

end

17. default values to arguments

def alphabetize(arr , rev=false)

18. In a method last line is implict return

a=[1,2]

so def fn

a

end

return a

but def fn

a

if ...

end

end

wotn return a by default. To overcome this. Write return a or just empty a as last statement..

19. Null is represneted by nil. So accesing an invalid key from hash returns nil.

Nil and false are the two non true values in ruby

20. **Symbols**

You can think of a Ruby symbol as a sort of name. It's important to remember that symbols **aren't** strings:

while there can be multiple different strings that all have the same value, there's only *one copy* of any particular symbol at a given time.

Symbols cannot have spaces..

they're primarily used either as hash keys or for referencing method names.

Symbols make good hash keys for a few reasons:

1. They're immutable, meaning they can't be changed once they're created;
2. Only one copy of any symbol exists at a given time, so they save memory;
3. Symbol-as-keys are faster than strings-as-keys because of the above two reasons.

The .to\_s and .to\_sym methods convert between symbols and strings

Besides using .to\_sym, you can also use **.intern**. This will internalize the string into a symbol and works just like.to\_sym:

From ruby 1.9, symbols inside hash can have colon at end and no need for rocket symbol

21. Select method to filter values from hash..

good\_movies = movie\_ratings.select { |k,v|

v>3

}

22..each\_key and.each\_value, to iterator over keys and values alone

23. Case -> when -> end is similar to switch in java

24. One line if and unelss.. eg: puts “a” if true..

If cannot be first part.. here we dont need end..

25. Ternary

boolean ? Do this if true: Do this if false

26. Case statment. Can use when.. then do shorten

27. **conditional assignment operator**:||=

set a variable only if its not been assigned so far

28. Ruby's methods will return the result of the last evaluated expression

Below both are same.. implicit return

def add(a,b)

return a + b

end

def add(a,b)

a + b

end

29. Upto and downto.

95.upto(100) and downto for reverse

30 .respond\_to? takes a symbol and returns true if an object can receive that method and false otherwise. For example,

[1, 2, 3].respond\_to?(:push)

31. 4.next will return5 . So .respond\_to next is a check for integer

32. [1, 2, 3] << 4

Smae as push 4 to array. Same for stings.

"Yukihiro " << "Matsumoto"

33. String interpolation is the preferred way for printing non string vlaues. Else if all string then + and << can concatenate strings..

34.A Ruby block is just a bit of code that can be executed. Block syntax uses either do..end or curly braces ({})

35.The collect method takes a block and applies the expression in the block to every element in an array

my\_nums = [1, 2, 3]

my\_nums.collect { |num| num \*\* 2 }

# ==> [1, 4, 9]

If we look at the value ofmy\_nums, though, we'll see it hasn't changed:

This is because .collect returns a *copy* of my\_nums, but doesn't change (or **mutate**) the originalmy\_nums array. If we want to do that, we can use .collect! with an exclamation point

36.Why do some methods accept a block and others don't? It's because methods that accept blocks have a way of transferring control from the calling method to the block and back again. We can build this into the methods we define by using the **yield** keyword

37. Blocks *are not* objects, and this is one of the very few exceptions to the "everything is an object" rule in Ruby.

Because of this, blocks can't be saved to variables and don't have all the powers and abilities of a real object. For that, we'll need... **procs**

You can think of a proc as a "saved" block: just like you can give a bit of code a name and turn it into a method, you can name a block and turn it into a proc. Procs are great for keeping your code **DRY**, which stands for**D**on't **R**epeat **Y**ourself. With blocks, you have to write your code out each time you need it; with a proc, you write your code once and can use it many times!

Procs are easy to define! You just call **Proc.new** and pass in the block you want to save

The & is used to convert the cube proc into a block. We'll do this any time we pass a proc to a method that expects a block

38. The .collect! and.map!methods do the exact same thing

39. Unlike blocks, we can call procs directly by using Ruby's .callmethod.

40. you can *also*convert symbols to procs using that handy little &.

41. Lambdas are defined using the following syntax

lambda { |param| block }

42. proc vs lamda

First, a lambda checks the number of arguments passed to it, while a proc does not. This means that a lambda will throw an error if you pass it the wrong number of arguments, whereas a proc will ignore unexpected arguments and assign nil to any that are missing.

Second, when a lambda returns, it passes control back to the calling method; when a proc returns, it does so immediately, without going back to the calling method.

43.

A block is just a bit of code between do..end or {}. It's not an object on its own, but it can be passed to methods like .eachor .select.

A proc is a saved block we can use over and over.

A lambda is just like a proc, only it cares about the number of arguments it gets and it returns to its calling method rather than returning immediately.

44. initialize is the constructor

In Ruby, we use @ before a variable to signify that it's an**instance variable**

$, @, or @@? This helps mark them as global, instance, and class variables

instance variables begin with an @.

**Class variables** are like instance variables, but instead of belonging to an instance of a class, they belong to the class itself. Class variables always start with *two* @s, like so:@@files.

**Global variables** can be declared in two ways. The first is one that's already familiar to you: you just define the variable outside of any method or class, and voilà! It's global. If you want to make a variable global from inside a method or class, just start it with a $, like so: $matz.

45. class SuperBadError < ApplicationError

Inheritance. Similar to extends in java

46. Ruby has no multiple inheritance

47.

if you want to end a Ruby statement without going to a new line, you can just type a semicolon. This means you can write something like

class Monkey

end

on just one line: class Monkey;end.

48. Methods are public by default in Ruby,

everything after thepublic keyword through the endof the class definition will now be public unless we say otherwise

private methods are just that: they're private to the object where they are  
defined. This means you can only call these methods from other code inside  
the object. Another way to say this is that the method cannot be called with an *explicit receiver*. You've been using receivers all along—these are the objects on which methods are called! Whenever you callobject.method, object is the receiver of the method

In order to access private information, we have to create public methods that know how to get it. This separates the private *implementation* from the public *interface*,

49. We can use attr\_reader to access a variable and attr\_writer to change it

class Person

attr\_reader :name

attr\_writer :name

def initialize(name)

@name = name

end

end

Ruby does something like this for us automatically:

def name

@name

end

def name=(value)

@name = value

end

Like magic, we can read and write variables as we please! We just pass our instance variables (as symbols) to attr\_reader orattr\_writer.That name= might look funny, but you're allowed to put an =sign in a method name. That's just a Ruby convention saying, "hey, this method sets a value!"

We can useattr\_accessor to make a variable readable *and* writeable in one fell swoop.

50. You can think of a **module** as a toolbox that contains a set methods and constants. There are lots and lots of Ruby tools you might want to use, but it would clutter the interpreter to keep them around all the time. For that reason, we keep a bunch of them in modules and only pull in those module toolboxes when we need the constants and methods inside!

You can think of modules as being very much like classes, only modules can't create instances and can't have subclasses. They're just used to store things!

It doesn't make sense to include variables in modules, since variables (by definition) change (or vary). Constants, however, are supposed to always stay the same, so including helpful constants in modules is a great idea.

Ruby doesn't *make* you keep the same value for a constant once it's initialized, but it will warn you if you try to change it. Ruby constants are written in ALL\_CAPS and are separated with underscores if there's more than one word.

51.One of the main purposes of modules is to separate methods and constants into named spaces. This is called (conveniently enough)**namespacing**, and it's how Ruby doesn't confuse Math::PIand Circle::PI.

52. require 'module'

to import modules..

53. Any class that includes a certain module can use those module's methods!

A nice effect of this is that you no longer have to prepend your constants and methods with the module name. Since everything has been pulled in, you can simply write PI instead ofMath::PI.

54. When a module is used to mix additional behavior and information into a class, it's called a **mixin**. Mixins allow us to customize a class without having to rewrite code

mixins could give us the ability to mimic inheriting from more than one class: by mixing in traits from various modules as needed, we can add any combination of behaviors to our classes we like!

55. the extendkeyword mixes a module's methods at the *class* level. This means that *class itself* can use the methods, as opposed to*instances* of the class.

56. we used underscores in our1\_000\_000 (one million). Ruby allows this, and it makes it easier to read big numbers!