Assignment Preparation

Precision Of Language

When asked to explain a piece of code, remember to always use precise vocabulary.

- Do not say ***method*** but ***method definition*** or ***method invocation / method call***
- Do not say *"result"* but talk about *"output"*, *"return value"* and *"mutations"*
- Be consistent when talking about variable assignment, a value is assigned to a variable, not the other way around.
- Use *"evaluates to true"* or *"is truthy"* when discussing expressions that evaluate to `true` in a boolean context
- Do not use "is true" or "is equal to true" unless you really are talking about the boolean `true`
- Be concise, don't beat around the bush too much
- Divide your answers in paragraphs
- Do not say `a` is `"hello"` but `a` is initialized and the value "string" is assigned to it.
- Make sure to distinguish between method parameters and arguments: a method is defined with parameters but it is called with arguments

Vocabulary to keep in mind

```
def foo(string)
  ...
end
```

We are defining a method called `foo` which takes 1 parameter named `string`.

```
foo("hello")
```

We are calling the method `foo` and pass the `String` `"hello"` as argument.

```
var = "hello"
```

We are initializing a local variable called `var` to the String `"hello"`.

```
puts "hello"
```

We are calling the `puts` method and pass it the `String` `"hello"` as an argument. This invocation will output `"hello"` and return `nil`.

```
i = 10

loop do
  i -= 1
  break if i == 0
end
```

We are calling the method `loop` and pass it a `do..end` block as an argument.

Inside of the block, we are reassigning the local variable `i` to the return value of the `Integer#-` method called on the local variable `i` with the integer `1` passed to it as an argument.

We break out of the loop with the keyword `break` if the value of the object that local variable `i` is referencing is equal to `0`.

Method definition vs. method invocation

A **method definition** is the place where we write out our method and what we want it to do.

```
def my_example_method
  puts "this is a cool method!"
end
```

A **method invocation** or **method call** is the place where we actually make use of this method.

```
my_example_method() # => "this is a cool method!
```

Truthiness

Be clear about the distinction between *truthy* values and the boolean `true` and subsequently, *falsey* values and the boolean `false`.

In Ruby, **every value apart** from `false` and `nil` evaluates to `true` in a boolean context. In other words, every value apart from `false` and `nil` is considered *truthy*.

This is **not** the same as saying that *truthy* values equal to the boolean value `true` and *falsey* values are equal to the boolean value `false`.

Return values of method invocations and blocks

Ruby methods **always** return the evaluated result of the last expression in both blocks and method definitions unless an explicit return comes before it.

Local variable scope

There are two major areas where we encounter local variable scoping rules:

- Method invocations with blocks (`do..end` or `{ }`)
- Method definitions

Local variable scope in method invocations with blocks

They might not look like it but **blocks are actually arguments passed to method invocations**.
In other words, we define blocks by passing them to a method invocation.

Technically any method can be called with a block, but the block is only executed if the method is defined in a particular way.

Example of a method definition not using a passed block:

```
def greetings
  puts "Goodbye"
end

word = "Hello"

greetings do
  puts word
```

```ruby
end

# Outputs 'Goodbye'
```

Example of a method definition using the passed block:

```ruby
def greetings
  yield
  puts "Goodbye"
end

word = "Hello"

greetings do
  puts word
end

# Outputs 'Hello'
# Outputs 'Goodbye'
```

Blocks also **create a new scope for local variables**. When we pass a block to a method invocation, we get the following:

- An **inner scope** created and inside of the block
- An **outer scope**, existing around and outside of the block

A variable's scope is determined by where it is initialised.

When it comes the combination of inner and outer scope, there are certain rules that come into play.

Outer scope variables can be accessed by inner scope

```ruby
a = 1          # outer scope variable

loop do        # the block following the invocation of the `loop` method creates an inner scope
  puts a       # => 1
  a = a + 1    # "a" is re-assigned to a new value
  break        # necessary to prevent infinite loop
end

puts a         # => 2  "a" was re-assigned in the inner scope
```

Inner scope variables cannot be accessed in outer scope

```ruby
loop do       # the block following the invocation of the `loop` method creates an inner scope
  b = 1
  break
end

puts b        # => NameError: undefined local variable or method `b' for main:Object
```

Peer scopes do not conflict

```ruby
2.times do
  a = 'hi'
  puts a      # 'hi' <= this will be printed out twice due to the loop
end

loop do
  puts a      # => NameError: undefined local variable or method `a' for main:Object
  break
end

puts a        # => NameError: undefined local variable or method `a' for main:Object
```

Variable shadowing

Variable shadowing is a Ruby mechanism that comes into play when we have a block that is passed to a method invocation with a parameter that has the same name as a local variable in the outer scope.

```ruby
number = 10

[1, 2, 3].each do |number|
  number = 2
end
```

What variable shadowing does, is preventing access to the outer scope variable.

This is not something that you want to have happen, so use explicit names!

Variables in method definitions

A block might be have a leaky scope (being able to access and change the outer scope) but **a method's scope is entirely self-contained and separate**. The only way that a method definition can access **LOCAL** variables is by passing them as arguments or by initialising them inside its definition. In other words, there is **no notion of outer of inner scope**.

Pass by Value vs Pass by Reference

There are two ways to handle objects that get passed to methods:

- Pass by Value
- Pass by Reference

Pass By Value

When we say that objects get passed by value, it means we provide the method with a copy of the object we pass to it. Whatever we do with that object inside of the method does not reflect on the outside object since we are altering its copy, not the object itself.

An example of a language that is purely pass by value would be C.

In some ways Ruby also shows "pass by value"-behaviour:

```ruby
def change_name(name)
  name = 'bob' # this reassignment does not change the object outside the method
end

name = 'jim'
change_name(name)
puts name           # => jim
```

Pass By Reference

When we say that objects get passed by reference, it means the method gets passed a reference to the original object, not a copy as is the case with pass by value. This also implies that everything we do with the object inside of the method directly alters the outside object as well.

Again, in some ways Ruby also shows "pass by reference"-behaviour:

```ruby
def cap(str)
  str.capitalize!   # does this affect the object outside the method?
end

name = "jim"
cap(name)
puts name           # => Jim
```

Call by sharing / Pass by value of the reference

As we can see, Ruby exhibits behaviour of both pass by value and pass by reference. Some people call this call by sharing or pass by value of the reference. However, the ground rule that Ruby maintains is **when an operation within the method mutates the caller, it will affect the original object.**

Variables as pointers

What are variables?

Variables (in Ruby) are basically references to objects in memory. These objects hold some sort of state or value and associated behaviour.

```ruby
number = 42
# We tell Ruby to associate the name number with the Integer object whose
# value is 42.

number.even?

# This also means that we can use this variable to make use of any behaviour
# defined in the object's class.
```

Object Id

Every object in Ruby has a unique object id, even literals such as numbers, booleans, nil, …
This unique id can be easily retrieved by calling the `#object_id` method on any object.

```ruby
number = 18
number.object_id

true.object_id
5.object_id
nil.object_id
"abc".object_id
```

Referencing

Since a variable is merely a reference to an object in memory, this also allows us to have different variables referencing the same exact object.

```ruby
blade = "runner"
bar = blade

blade.object_id
# 34

bar.object_id
# 34
```

Reassigment

When two variables point to the same object, we can still make one variable point to another object entirely.

```ruby
blade = "runner"
bar = blade

blade = "vampire"

blade.object_id
```

```
# 78

bar.object_id
# 34
```

Mutability

We can also decide to mutate an object instead, which means changing its value in a certain way. Not all objects in Ruby allow this to happen, though. Some are what we call immutable, the most prevalent being integers, booleans, ranges and nil. They are immutable simply because their classes do not provide any associated behaviour that lets us change their values.

```ruby
meaning_of_life = 42
meaning_of_life.object_id
# 22

meaning_of_life = meaning_of_life * 2
meaning_of_life.object_id
# 89
```

Even though it looks like we mutated the `meaning_of_life` variable, we actually simply reassigned it, which as we saw above, does not change the object but simply points the variable to a new object.

That being said, most of the objects in Ruby are mutable, meaning the class of those objects permits modification of the object's state in some way. Some examples here are strings, arrays, hashes, ...

```ruby
arr = ["a", "b", "c"]
arr.object_id
# 19

arr[0].object_id
# 75

arr[0] = "A"
arr[0].object_id
# 99

arr.object_id
# 19
```

As we stated above, everything in Ruby has an object_id, so while the array `arr` clearly has an object_id, its different elements do too. We can then use a setter method to reassign one of those elements of `arr` to a different value, hereby mutating `arr` but not its elements.

```
arr = ["a", "b", "c"]
arr.object_id
# 19

arr[0].object_id
# 75

arr[0].upcase!
arr[0].object_id
# 75

arr.object_id
# 19
```

Here we go one step further. Since the first element in the `arr` array is a string, which is mutable, we can change its value in place, without reassigning it. This means that we are now mutating `arr[0]` but also `arr` itself.

Indexed assigment is mutating

It is important to realise here that while assigment and reassigment are non-mutating, indexed assigment like you would see with strings, arrays, hashes is in fact mutating. **This is because indexed assigment is actually syntactic sugar for a method defined on the calling object's class.**

```
string = "abc"
string.object_id
# 34

string[1] = "x"

string
# axc

string.object_id
# 34
```

Concatenation is mutating

Just as indexed assigment is behaviour a class provides for its object, so is concatenation. And just like indexed assigment, it is mutating. This way of adding elements to collections is used by collections like arrays, hashes and strings (collection of chars).

```
blade = "runner"
blade.object_id
#22

blade << " 2049"

blade
# "runner 2049"
```

```ruby
blade.object_id
#22
```

Setters are mutating

Somewhat similar to indexed assigment, setters are also methods that are used to modify the state of an object. Both look like assigment on a superficial level.

```ruby
person = { name: "Rein" }
person.object_id
#89

person.name = "Van Imschoot"

person.object_id
#89
```

This looks like assigment but it is actually a setter called on the object `person`, modifying its state.

Working with collections

#each

```ruby
[1, 2, 3].each do |num|
  puts num
end

{ "a" => 100, "b" => 200 }.each do |key, value|
  puts key
  puts value
end
```

The `each` method calls the given block once for each element in the collection it is called on, passing that element as a parameter. Afterwards, it returns the collection itself.

#select

```ruby
[1, 2, 3].select do |num|
  num.odd?
end
# returns [1, 3]


{ "a" => 100, "b" => 200, "c" => 300 }.select do |key, value|
  v < 200
end
# returns { "a" => 100}
```

The `select` method calls the given block once for each element in the collection it is called on and **evaluates the return value of the passed block.** When evaluating said return value, `select` only cares about its **truthiness**.

If the return value of the block is "truthy", then the element during that iteration will be selected. If the return value of the block is "falsey" then the element will not be selected.

`select` then returns a new collection containing all of the selected elements.

#map

```ruby
[1, 2, 3].map do |num|
  num * 2
end
# returns [2, 4, 6]

[1, 2, 3].map do |num|
  puts num # puts returns nil
end
# returns [nil, nil, nil]
```

The `map` method calls the given block once for each element in the collection it is called on and **evaluates the return value of the passed block.** It then uses the return value to build a new collection of transformed values. `map` will, as such, always return a collection of the same length of the collection it is called upon.

Sorting of arrays

The sorting of any kind of collection requires the usage of a **complex algorithm**, which isn't important for our purposes right now. What is important to know is that **comparison** lies at the heart of sorting.

When we sort an array we essentially compare the different values which each other by a specific criterion. This criterion can be provided by us, the programmers, or by Ruby itself.

The `<=>` operator

Any type of object that we want to sort in a collection needs to implement a `<=>` method.

The `<=>` method compares two values **of the same type** and returns `-1`, `0` or `1`, depending on whether the first object is less than, equal to, or greater than the second object. If the two objects cannot be compared then nil is returned.

```ruby
2 <=> 1 # => 1
1 <=> 2 # => -1
2 <=> 2 # => 0
'b' <=> 'a' # => 1
```

```
'a' <=> 'b' # => -1
'b' <=> 'b' # => 0
1 <=> 'a' # => nil
```

The return value of the `<=>` method is used by sort to determine the order in which to place the items. If `<=>` returns `nil` to sort then it throws an argument error.

```
['a', 1].sort # => ArgumentError: comparison of String with 1 failed
```

Sorting order

If the type of object you want to sort implements a `<=>` operator, it is important to know how it is actually implemented.

Integers

By default, `Integer` objects will be sorted in ascending order.

```
[3,4,1,2,5].sort # => [1,2,3,4,5]
```

Strings

The `<=>` implementation of `String` objects uses the ASCII table to determine the sorting order.

The ASCII table gives all characters a specific position. It's not required to learn them by heart but it is helpful to know the following general rules:

- Uppercase letters come before lowercase letters
- Digits and (most) punctuation come before letters
- There is an *extended* ASCII table containing accented and other characters - this comes after the main ASCII table

```
["a", "A", "!"].sort # => ["!", "A", "a"]
```

When dealing with multi-character strings, the `String#<=>` method compares them character by character. If two strings are equal but one is longer than the other, the longer string will be considered greater.

```
['arc', 'bat', 'cape', 'ants', 'cap'].sort
# => ["ants", "arc", "bat", "cap", "cape"]
```

Subcollections

Arrays can also be compared and much like the `String#<=>` method compares strings character by character, the `Array#<=>` method compares them element by element.

If two elements are of different types, it will once again result in an `ArgumentError`.

Calling #sort with a block

We can also call sort with a block. This gives us more control over how the items are sorted. The block needs two arguments passed to it (the two items to be compared) and the return value of the block has to be `-1`, `0`, `1` or `nil`.

```
[2, 5, 3, 4, 1].sort do |a, b|
  a <=> b
end
# => [1, 2, 3, 4, 5]

[2, 5, 3, 4, 1].sort do |a, b|
  b <=> a
end
# => [5, 4, 3, 2, 1]
```