Precision of language

Vocabulary

Variable assignment

```
a = "hello"
```

The variable `a` is initialised with a reference to the string object `"hello"`.

Method invocation

```
arr = [1, 2, 3]

new_array = arr.each { |num| puts num }
```

The `each` method is called on `arr` and passed a block as an argument...

Variables

References / variables as pointers / mutability

Variables do not contain objects themselves. Rather variables contain ***references*** to objects held in address space in memory.

Variables ***point*** to address spaces in memory that contain objects.

Variable assignment and reassignment

Variable assignment

In an assignment with the variable on the left and the object on the right, the variable ***receives a reference to the object***.

In an assignment from one variable to another, the variable on the left receives a ***copy of the reference*** stored in the variable on the right, with the result that ***both variables now contain references to the same object***.

Reassignment

In a **reassignment** of an existing variable on the left and a new object on the right, the variable receives a reference to a different object. The reassignment does not mutate the original object but instead ***creates a new object and changes what the variable references.***

The variable is disconnected from the original object that it referred to previously, and reassigning/reusing the variable has no effect on any other variables containing references to the original object.

object_id

Before reassignment: Calling the `object_id` method on either variable demonstrates that both variables not only reference a string with the same value, but are in fact a reference to the ***same*** string object.

After reassignment: Calling `object_id` on either variable demonstrates that the variables no longer contain references to the same string object but instead different objects.

Mutability

Mutable objects can be modified after they are created. Destructive/mutating operations performed on mutable objects modify the original object without creating a new one, while non-destructive operations do not modify the original object and instead return new objects.

Immutable objects cannot be modified after they are created. Operations performed on immutable objects create new objects rather than modifying the original.

Immutable objects, like numbers, cannot be modified in place...

Mutation vs reassignment

Mutation refers to changing the value of an object in place, without creating a new object. When a mutating method is performed on an object, any variables pointing to the mutated object will also see this change.

/ ... the mutation can be examined through any of its references

Non-mutating operations, like reassignments, do not change the value of an object, but rather return a new object and change what a variable references, breaking the connection to the original object and any of its references. Non-destructive operations do not have an effect on the original object.

Language precision

Reassignment

```
str = "hello"
abc = str
str = "goodbye"

puts str # => "goodbye"
puts abc # => "hello"
```

The variable `str` is initialised with a reference to the string object `"hello"`.

- In the assignment from one variable to another, `str` receives a copy of the reference stored in `abc`.
- `str` is reassigned with a reference to a new string object `"goodbye"`—the reassignment creates a new object and changes what the variable references.
- `str` is disconnected from the original object `"hello"`, which has no effect on any other variables containing references to the original object. `abc` continues to reference `"hello"`.

(Calling the `object_id` method on either variable demonstrates that both variables not only reference a string with the same value, but are in fact a reference to the *same* string object.)

```
# object_d method calls BEFORE reassignment
puts str.object_id # unique id
puts abc.object_id # same id as str

# object_id method calls AFTER reassignment
puts str.object_id # unique id
puts abc.object_id # another unique id - different from str
```

Mutation

```
str = "hello"
abc = str
str << ", world"

puts str # => "hello, world"
puts abc # => "hello, world"
```

- The destructive method `<<` (shovel operator) is called on `str` which modifies/mutates the original string object `"hello"` by concatenating its argument, `"world"`, to the object, resulting in `"hello, world"`.
- Even though the mutating method is called on `str`, the changes can be seen through any of its references, including `abc`. Both variables continue to point to the same string object.

Variable scope

Variable scope refers to where a variable is accessible in a program. "Local" pertains to the fact that variables have limited scope i.e. they are only visible in limited parts of the program.

Variable isolation

Variables defined within one scope cannot be directly accessed from another scope.

(In method definitions, this isolation prevents methods from accessing or modifying variables from their surrounding context. To use an outer variable inside a method, it must explicitly be passed as an argument.)

Local variable scope in relation to blocks

Variables initialised in the surrounding scope / outer code can be accessed and modified from within the block—reassignment of outer variables is possible from within blocks.

Variables initialised inside the block cannot be accessed by outer code.

Peer blocks

Variables initialised inside a block cannot be accessed by outer code, nor can they be accessed by peer blocks.

```
name = "ruby"

1.times do |i|
  greeting = "hello"
  name_2 = "cat"
  puts "#{greeting} #{name}"
end

loop do
  puts name
  puts "#{greeting} {name}"
  break
end

puts name_2
```

Nested blocks

In nested blocks, the innermost block can access variables initialised in its surrounding scope, including its most immediate surrounding scope and the outermost scope.

The second-level block can only access variables in its surrounding scope (the first-level scope), but it cannot access variables from the nested scope.

The outermost (first-level) scope cannot access variables inside nested scopes.

```
# nested blocks

first_level_var = "I can be accessed by all scopes!"

[1, 2, 3].each do |n|
  second_level_var = "I can be accessed by the nested scope!"

  loop do
    third_level_var = "I cannot be accessed by outer scopes!"

    puts first_level_var  # nested scope has access to first-level scope
    puts second_level_var # nested scope has access to second-level scope
    puts n                # nested scope has access to block parameter
    break
  end

  puts first_level_var # second-level can access first-level scope
  puts third_level_var # error - cannot access nested scope
end

puts second_level_var, third_level_var # error - cannot access either block scopes
```

Conditionals

Conditional statements do **not** create a new scope for local variables: variables initialised inside conditionals can be accessed from within the conditional statement, as well as from outside of the conditional statement.

```
a = 1
b = 2

if a < 2
  message = "you can see this"
end

puts message
```

Language precision

Local variable scope of a block

```
name = "ruby"
count = 0

loop do
  greeting = "hello"
  puts name  # block can access outer variable
```

```
  count += 1 # outer variable can be reassigned from within the block
  break if count == 3
end

puts count # => 3 # modified from within the block
puts greeting # NameError - outer scope cannot access variables initialised inside block
```

- In each iteration of the block code, the variable `greeting` is initialised with a reference to the string object `"hello"`.
- The `puts` method call outputs the value of the outer variable `name`—the block has access to variables initialised in the outer code, which means that outer variables can be modified from within the block.
- `count += 1` is a reassignment operation of the outer variable `count`, which increments the value of `count` by `1`.

`puts count` outputs `3`—`count` was reassigned to `3` from within the block.

`puts greeting` throws an error message because `greeting`, which was initialised inside the block, cannot be accessed by the outer code.

Variable shadowing

Variable shadowing occurs when a block parameter has the same name as a variable in an outer scope.

When this happens, the ***block parameter shadows the outer variable, preventing access to the variable to the outer variable***. The outer variable cannot be modified from within the block.

Variable shadowing only occurs with block parameters, not variables within a block that have the same name as an outer variable. In these cases, reassignment would occur.

Language precision

```
val = 1

arr = [1, 2, 3].map do |val|
  puts val
  val += 1
end

puts val
p arr
```

- The block parameter `val` shadows the outer variable of the same name `val`, preventing access to the outer variable from within the block.
- Inside the block, `val` refers to each element of the array, not the outer variable. The outer `val` remains unchanged.
- `puts val` outputs the value of `val` that was initialised in the outer scope
  - —this is the only variable with the name `val` that is available to the `puts` call

---

Local variable scope in relation to method definitions

Method definitions create a **self-contained scope**: variables initialised within the method definition cannot be accessed by the outer scope,

... and variables initialised in the outer scope cannot be accessed or modified from within the method definition, ***unless they are passed in as arguments and mutating methods are called on those corresponding objects***.

Language precision

Variable scope of a method definition

```
# outer scope cannot access variables initialised inside the method definition

def some_method
  another_name = "ruby"
end

puts another_name
```

- Inside the `some_method` method definition, the local variable `another_name` is initialised with a reference to the string object `"ruby"`.
- `puts another_name` throws an error because the outer scope does not have access to a variable called `another_name`.
- `another_name` is only visible to the method definition's scope—that is, the scope that it is defined in.

```
# variables initialised in the outer scope cannot be directly accessed from inside the
method definition

name = "ruby"

def some_method
  puts name # NameError - cannot access the outer variable name
end

some_method
```

The variable `name` is initialised with a reference to the string object `"ruby"`.

The `some_method` method is called.

- Inside the method definition, `puts name` is evaluated but returns a NameError message stating that there is an undefined local variable or method—i.e. there is no variable (or method) called `name` that is visible inside the method definition.
- This is because the outer variable `name` is not visible to the method definition.

```
# variables initialised in the outer scope can only be accessed by the method definition if
they are passed as arguments

name = "ruby"

def some_method(name)
  puts name
end

some_method(name)
```

- The `some_method` method is called and `name` is passed as an argument.

- Inside the method definition, the method parameter `name` is bound to the same string object referenced by the outer variable `name`.
- `puts param` outputs the value of the local variable `param` (which is a reference to the same string object that the outer variable `name` references).
- Although the method parameter has the same name as the outer variable `name`, they are entirely separate—they just happen to have the same name.
- This is not variable shadowing because a method definition creates its own scope with its own local variables.

Scope of constants

Constants are variables whose values should remain unchanged through execution of the program—they should be treated as immutable objects.

(Ruby does allow for modifying constants though it does issue a warning!)
(use `freeze` to prevent modification)

Constants defined at the top level of the program are globally accessible.

- define a constant with a capital letter --> by convention constants are written in `ALL_CAPITALS`

Scope:

- **Lexical scope** --> constants are available within the scope they are defined and any nested blocks
- Unlike local variables, constants can be accessed outside the block in which they are defined.

Code examples

Example 1: Basic scope of constants vs local variables inside blocks

```
FAVOURITE_ANIMAL = "elephant"

loop do
  puts "I love #{FAVOURITE_ANIMAL}!"
  FAVOURITE_COLOUR = "green" # define a constant inside a block
  break
end

puts FAVOURITE_ANIMAL # => "elephant"
puts FAVOURITE_COLOUR # => "green" (accessible outside of block)

# Compare with local variable behaviour

loop do
  another_cool_animal = "sloth"
  break
end

puts another_cool_animal # NameError: undefined local variable
```

Example 2: Constants in methods and blocks

```
MINIMUM_LENGTH = 8

def password_long_enough?(password)
  password.length > MINIMUM_LENGTH # constant accessible in method
```

```ruby
end

my_password = "password123"
puts password_long_enough?(my_password)

loop do
  INVALID_PASSWORDS = [ "rubysucks" ] # constant defined in block
  break
end

puts !(password_long_enough?(my_password) && INVALID_PASSWORDS.include?(my_password)) ?
"Valid password!" : "Invalid password!"
```

Methods

Method definition vs method invocation

Method definition - a method is defined with the keyword `def`; method definitions establish what parameters it accepts, what it does, and what value it returns

- In a method definition, the parameters are indicated through the list of variables in parentheses after the method name. Parameters can be required or optional.

Method invocation - when a method is called or executed

- In a method invocation, the values supplied to a method are the corresponding arguments. These arguments are assigned to the parameters defined in a method definition.

```
Method definition - defines the behaviour of a method
Method invocation - executes the behaviour established by the method definition
```

Language precision

```ruby
def greeting
  puts "Hello"
end

greeting
```

- The `greeting` method is called. Inside the method definition, `puts "Hello"` outputs `"Hello"` and returns `nil` (since `puts` returns `nil`) which is the subsequent return value of the method. The return value of the method is the last evaluated expression in the method definition.

```ruby
def greeting(name)
  puts "Hello #{name}"
end

greeting("Ruby")
```

- The `greeting` method is called and passed the string `"Ruby"` as an argument. Inside the method definition, the method parameter `name` is bound to the string object `"Ruby"` that was passed as an argument.
- The `puts` method call outputs the evaluated result of the string interpolation, `"Hello Ruby"`, which is the subsequent return value of the method. The return value of the method is the last evaluated

expression in the method definition.

Parameters vs arguments

Parameters - the list of variables in a method definition that specify what inputs the method accepts/receives. They act as placeholders to be filled when a method invocation is executed.

Arguments - the actual (reference-) values passed to a method when it is called, which are assigned/bound to the method definition's parameters.

There is a direct correspondence between the syntax of the variables list in a method definition and the syntax of the arguments list in a method invocation.

More

- parameters create local variables that are only available within the method's scope
- parameters receive values of the corresponding arguments when a method is called
- the number of arguments passed should match the number of parameters (unless optional or default parameters are defined)

Default parameters

In a method definition, the parameters are the list of variables that indicate what inputs a method accepts. Parameters can be required or optional. Arguments are the values passed to a method, which are assigned to the parameters when a method is called.

Default parameters allow methods to be defined with arguments that have preset values. If the caller does not supply a value for that parameter, the default value will be used instead.

Default parameters are evaluated when a method is called.

Language precision

```ruby
def greeting(name = "friend")
  puts "Hello, #{name}!"
end

person = "Ruby"

greeting(person)
greeting
greeting(nil)
```

The variable `person` is initialised with a reference to the string object `"Ruby"`.

- The `greeting` method definition specifies a default parameter, which allows arguments to be defined with preset values if no argument is supplied when the method is called.
- In the first `greeting` method invocation, `person` is passed as an argument and the parameter `name` is bound to the same string object `"Ruby"` that `person` references—essentially, the method receives a reference to the object. `puts` outputs the evaluated result of the string interpolation: `"Hello, Ruby!"`
- In the second method invocation, no argument is supplied, so the parameter `name` is assigned the default value of the string `"friend"`. The `puts` call outputs: `"Hello, friend!"`.
- In the third method call, `nil` is passed as an argument, which overrides the default parameter. The output is `"Hello, !"`.

(All three method invocations return `nil`—which is the evaluated result of the last expression in the method body—since `puts` returns `nil`)

Object passing

Pass by value - When an argument is passed to a method by value, the method receives a *copy* of the value. Any operations performed on the value within the method have no effect on the original object.

Pass by reference - When an argument is passed to a method by reference, the method receives a *reference* to the object. Any operations performed on the reference within the method can affect the original object.

Pass by reference value / pass by object reference

Ruby exhibits pass by reference value behaviour: when an argument is passed to a method, ***the method receives a reference to the object***, but what happens to the original object depends on what types of operations are performed on the object.

Mutating methods vs non-mutating methods

Mutating operations ***modify the original object in place***, rather than creating a new object. ***These changes can be seen through any of the object's references***.

Non-mutating operations ***return new objects and change what a variable references***. They do not affect the original object; the original object remains the same.

(related: variable scope of a method definition)

Method definitions create their own scope: variables initialised outside of the method cannot be accessed or modified unless they are passed as arguments to the method and mutating operations are performed on the corresponding objects.

Language precision

Pass by reference value

**Pass by reference value behaviour exhibited by Ruby

- The method receives a reference to the same object that the outer variable references.

Variable scope of a method definition

- Method definitions create their own scope: variables initialised within the method definition cannot be accessed by the outer code, and variables initialised in the outer code cannot be directly accessed or modified from within the method definition unless they are passed in as arguments and mutating operations are performed on those corresponding objects.

Mutating operations inside of a method definition

Mutating/destructive operations inside the method

- Any mutating operations performed on the object will modify the object in place, rather than create a new object.
- This means that the mutations to the original object can be seen in any of its references, including outer variables that contain references to modified object.

Code example

```ruby
def mutate_string(str)
  str.upcase!
end

greeting = "hello"
new_greeting = mutate_string(greeting)

puts greeting     # HELLO
puts new_greeting # HELLO
```

```ruby
# Original state
greeting --> "hello"

# Beginning of `mutate_string` method invocation
greeting --> "hello" <-- str

# During method execution
greeting --> "HELLO" <-- str

# After `mutate_string` method call
greeting --> "HELLO" <-- new_greeting
```

`greeting` is initialised with a reference to the string object `"hello"`. `new_greeting` is initialised with a reference to the return value of `mutate_string(greeting)`.

`mutate_string(greeting)` is evaluated:

- The method `mutate_string` is called and `greeting` is passed as an argument.
- Inside the method definition, ***the method parameter*** `str` ***is bound to the same string object that*** `greeting` ***references***—essentially, the reference to the string object `"hello"` is passed as an argument to the method.
- The destructive `upcase!` method is called on `str` which ***modifies the original object itself*** by changing the string to uppercase character—`"hello"` is mutated to `"HELLO"`.
- Even though the mutating method is called on `str` from within the method definition, ***the mutation to the original object can be viewed through any of its references***, including the outer variable `greeting`.
- Method definitions create their own scope: variables initialised outside the method definition cannot be accessed or modified ***unless they are passed in as arguments and mutating operations are performed on those corresponding objects***.

(The return value of the method is the same as the evaluated result of the last expression in the method body, the string object `"HELLO"`, which is passed back to its calling code, `mutate_string(greeting)`, which is subsequently assigned to `new_greeting`.)

Non-mutating/non-destructive operations inside a method definition

- Any non-mutating operations performed on the object will ***create new object***, rather than modifying the original object in place.
- This includes ***reassignments***: a new object is created and ***the variable's reference changes***
  - The local variable inside the method is likely to be reassigned to the new object.
  - This ***disconnects the reference to the original object*** from the local variable being reassigned.
- This means that there is ***no effect on outer variables that contain references to the original object***; they will continue to point to the original object which remains unchanged.

- This demonstrates how Ruby *appears* to pass by value: particularly for immutable objects, it *appears* that method receives a copy of the value of the original object, therefore any operations performed thereafter do not have an effect on the original object or any of its references outside the method definition.

Code example

```ruby
def mutate_string(str)
  str = str.upcase
end

greeting = "hello"
new_greeting = mutate_string(greeting)

puts greeting     # output: hello
puts new_greeting # output: HELLO
```

```ruby
# Original state
greeting --> "hello"

# Beginning of `mutate_string` method invocation
greeting --> "hello" <-- str

# During method execution
greeting --> "hello"
str      --> "HELLO"

# After `mutate_string` method call
greeting     --> "hello"
new_greeting --> "HELLO"
```

`greeting` is initialised with a reference to the string object `"hello"`. `new_greeting` is initialised with a reference to the return value of `mutate_string(greeting)`.

`mutate_string(greeting)` is evaluated:

- The method `mutate_string` is called and `greeting` is passed as an argument.
- Inside the method definition, ***the method parameter*** `str` ***is bound to the same string object that*** `greeting` ***references***—essentially, the reference to the string object `"hello"` is passed as an argument to the method.
- The non-destructive `upcase` method is called on `str` which creates a new object, `"HELLO"`. The local variable `str` is reassigned with a reference to this new object.
- The ***reference to the original object is disconnected from the local variable***. The non-mutating operation has *no effect* on the original object. The outer variable `greeting` continues to point to the original object.
- Method definitions create their own scope: variables initialised outside the method definition cannot be accessed or modified ***unless they are passed in as arguments and mutating operations are performed on those corresponding objects***.
- Although the method receives a reference to the original object, the non-mutating operations inside the method have no effect on the original object or its references.

(The return value of the method is the same as the evaluated result of the last expression in the method body, the string object `"HELLO"`, which is passed back to its calling code, `mutate_string(greeting)`, which is subsequently assigned to `new_greeting`.)

Both non-mutating and mutating operations

The order of operations inside the method matters!

- If a non-mutating operation is performed on the object and that value is captured in a variable, which is then operated on by mutating methods, the original object remains unchanged.
  - The reference to the original object is disconnected from the local variable.
  - Any mutating operations onwards only modify the new object.
- If a mutating operation is performed on the original object, any of its references will see those changes ***UNTIL*** a non-mutating operation is performed on the object inside the method definition.
  - This creates a new object and changes what the local variable references.
  - Any mutating operations performed on the local variable onwards will have no effect on the original object or its references outside the method definition.
  - Any mutating operations before reassignment / non-mutating operations will modify the original object up until that point in execution.

Output vs return

Return - every expression can be evaluated to some result, which is the return value.

When a method is called, every method call evaluates to its return value, which is passed back to the caller. The method's return value is the same value of the last evaluated expression in the method, unless an explicit `return` keyword is executed prior to the last line in the method body.

Output - information printed to the console/screen using methods like `puts`, `prints` or `p`.

In methods, output is a side effect that shows the user information, but it does not affect a method's return value.

> Methods should either return a meaningful value (which can be used in further operations or variable assignments) or produce a side effect, but not both.

Method return values

Implict vs explicit return values

When a method is called, the method evaluates to its return value, which is the same value as the last expression during execution of the method—this value is *implicitly* returned.

If an explicit `return` keyword is reached prior to the last expression, execution of the method terminates, ignoring any lines after `return` in the method body. The evaluated expression following `return` is the return value, which is passed back to the calling code.

Using method return values as arguments

Return values can be used in method chains: this is when a method's return value is immediately used as an argument/input to other methods without storing the value in a variable first.

This creates a chain of method calls where one method's return value is the input for another method call.

Related concepts

Return value of `if` conditionals

The return value of an `if` statement is the last evaluated expression in the executed branch.

- If the condition is truthy, the return value is the last evaluated expression in the `if` branch
- If the condition is falsy and there's an `else` branch, the return value is the last evaluated expression in the `else` branch
- If the condiiton is falsy and there is no `else` branch, the return value is `nil`

Language precision

Passing and using blocks with methods

Block: a chunk of code that can be passed to a method, associated with method invocation (not method definition)

Passing a block to a method is essentially passing a behaviour as an argument to a method.

Things to remember!

- To use a block in method, the block must be passed as an argument to the method, but the method must also be defined to use the block.
  - Use the `yield` keyword inside a method definition to execute the block.
- Blocks have their own return values that can only be used by the method that called the block.
- Variable scope: blocks can access variables in its surrounding scope, but method definitions cannot.

Language precision

Truthiness

In Ruby, every expression is considered truthy except for the two falsy values `false` and `nil`.

Truthiness refers to whether an expression is considered true ("truthy") or considered false ("falsy"). This differs from the boolean objects `true` and `false` whose only purpose is to convey `true` or `false`. In other words, an expression evaluates as true or evaluates as false, but not specifically the boolean values.

In conditional contexts, truthiness is used to evaluate conditional expressions. Ruby does not specifically check for whether the expression evaluates as the boolean object `true`, but rather if it does not evaluate to either falsy values `false` or `nil`.

Important points to consider!

Be careful with variable assignments in conditionals!

Language Precision

Related concepts

Short-circuit evaluation

`&&` and `||` operators exhibit short-circuiting:

- Execution will terminate once it can guarantee a return value
- Expressions after the guaranteed value is reached will not be evaluated

e.g. some predicate methods return `true` or `false` depending on the truthiness of the return values from the block:

`any?`

- if at least one of the return values is truthy, `true` is returned (otherwise, `false` is returned)
- short-circuiting occurs if at least one of the block's return values if truthy

`all?`

- if all of the return values are truthy, `true` is returned (otherwise if at least one of the return values is falsy, `false` is returned)
- short-circuiting occurs if at least one the the block's return values is falsy

Language precision

Iteration

Iteration is the process of repeatedly executed a set of operations until a specific condition is met.

Allows you to process collections of data by performing operations on each element at a time.

each

The `each` method allows for iterating over a collection of data, executing the block code for each iteration, and returns the original collection.

Language precision

Variable scope of blocks

map

The `map` method allows for iterating over a collection of data, and **transforms each element in an array based on the return value of the block**.

`map` **returns a new array containing the transformed values**.

/ ... returns a new array containing the return values from the block

```
# For elements 4 and 5, the block returns nil because puts returns nil
# Map captures the block's return value for each element, so:
# [1, 2, 3, 4, 5].map → [1*2, 2*2, puts 3, puts 4, puts 5] → [2, 4, nil, nil, nil]
```

Language precision

select

The `select` method allows for iterating over a collection of elements, and returns a new array containing the elements for which the block returns a truthy value. If the block's return value is truthy for that iteration, the return array is populated with that iteration's element.

Truthiness in `select`

The return array by `select` is based on the truthiness of the block's return values: if the block's return value is truthy, the current element is selected and added to the return array. If the block's return value is falsy, nothing happens to the return array for that iteration.

Language precision