Chapter 1

Ruby

> A regretful honey maker could be called a rue-bee
>
> Kliff

This is a programming language chapter so it has two (2) main things: talk about some properties that the Ruby programming language has and the syntax the language has. If you want to code along, all you need is a working version of Ruby and a text editor. You can check to see if you have Ruby installed by running `ruby -version`. At the time of writing, I am using Ruby 3.0.4. You can also download an interactive ruby shell called `irb`.

1.1 Introduction

Unlike the previous languages you have seen in 131/132 and 216, Ruby does not use a compiler so no machine code is generated. This means that compile-time checks do not exist. This basically means that every check or error is done during run-time. I'll expand more on this in a bit but for now, lets write our first Ruby program.

```
1 # hello_world.rb
2 puts "Hello World"
```

Despite this being very simple, we already learned five (5) things.

- Single line comments are started with the pound or hashtag symbol
- no semicolons to denote end of statement
- `puts` is used to print things out to stdout (`print` if you don't want a newline at the end
- Parenthesis are technically optional when calling functions (but good style says to only leave out if there are no arguments or if the function is `puts`, `require`, or `include`)

- ruby file name conventions are `lowercase_and_underscore.rb`
- Strings exists in the language (most languages do, but some do not)

Now to run our ruby program we can just use

```
ruby hello_world.rb
```

Congrats, you have just made your first program in Ruby!

1.2 Typing

Now we just said that Ruby does not use compile-time checks and all checks and errors are done at run-time. Let's see this in action.

```
# program1.rb
variable1 = 5
variable1 = "hello"
variable2 = 4
variable3 = variable1 + variable2
puts varable3
```

Here is a simple program that sets some variables, adds two things together, and then prints the results. This looks weird, but first lets run and see what happens, and then we can look at the syntax.

```
ruby program1.rb
hw.rb:5:in `+': no implicit conversion of Integer into String (TypeError)
    from hw.rb:4:in `<main>'
```

Looks like we have an error! Seems like `variable1` and `variable2` have different types so we can't use `'+'` on them. Which is interesting for 2 main reasons

- We did not get an error on line 3 when we change `variable1` from an `Integer` value to a `String` value
- there is no automatic conversion of integers to strings like we saw in Java

The first point is really important, but before we talk about it, let's just modify our code so that it runs without any errors by deleting line 3.

```
# program1-1.rb
variable1 = 5
variable2 = 4
variable3 = variable1 + variable2
puts varable3
```

Now if we run our code we get a different error. We get **`'undefined local variable or method `varable3' for main:Object (NameError)'`**. Notice that because we check everything during run-time, this error was not picked up until ruby was about to execute it. **Many bugs from beginner Ruby programmers is due to misspelled variable names**. Here is a more visual demonstration of run-time erring.

```
12 # program1-2.rb
13 variable1 = 5
14 variable2 = 4
15 variable3 = variable1 + variable2
16 print variable1
17 print " + "
18 print variable2
19 print " = " # gross. We will learn to convert later
20 puts varable3   # still misspelled
```

If we run the above code, we get **'5 + 4 = '** printed out and then we get the same **'NameError'** as before. A error free program would look like

```
21 # program1-3.rb
22 variable1 = 5
23 variable2 = 4
24 variable3 = variable1 + variable2
25 puts variable3
```

Now that we fixed this issue, we can talk about why we didn't get an error on line 3 in `program1.rb`.

Notice that in the above code we set values to variables but we didn't define the type like we did in Java and C. This is because Ruby uses **Dynamic type checking**. Dynamic type checking is a form of type checking which is typically contrasted to **Static type checking**. **Type checking** is an action that is used for a **Type system**, which determines how a language assigns a type to a variable. Ultimately: How does a a language know if a variable is an `int` or a `pointer`? It does so by type checking.

For the most part, you probably only used static type checking since both C and Java are statically typed. **Static typing** means that the type of a variable, construct, function, etc is known at compile time. Should we then use a type in an incorrect manner (as we did in `program1.rb`, then the compiler will raise an error and compilation will be aborted. Contrasted to **Dynamic typing** which means that the type is only calculated at run-time. Consider the following:

```
1 # err.c
2 void main(){
3     int x;
4     x = "hello"
5 }
```

Should we try to compile this code with the `-Werror`[1] compile flag, we will get the following: **`'error: assignment to 'int' from 'char *' makes integer from pointer without a cast'`**. This is because during compilation, the compiler marks the x variable as having a type of int yet is being assigned a pointer.

Now how did gcc know that there was a type issue here? The first conclusion would be that we declared x as an `int` explicitly on line 3. This is called **manifest or explicit typing** where we explicitly declare the type of any variable we create. This is in contrast to **latent or implicit typing** where we don't have to do this as we saw in ruby. It is important to note: **manifest typing is not the same as static typing**. We will see this in OCaml as the language is statically typed, but uses latent typing for its variables.

Back to Dynamic type checking, let's look at the following:

```
1 # checking.rb
2 def add(a,b)
3     puts a + b
4 end
5
6 add(1,2)
7 add("hello"," world")
```

To begin, this is how you create a function in Ruby. Functions begin with the `def` keyword and end with the `end` keyword. We will go into ruby code examples later so for now just know this is a function that takes two arguments and then prints the the result of `a + b`. Since Ruby is dynamically typed, we don't assign types to the parameters a and b until we run our code, and not until we actually use the values. This allows us to call `add` with both `Integers` and `Strings`. Much like before it is important to note that **latent typing is not the same as dynamic typing**.

In any case, you may be wondering Ruby knows the types of variables if we don't explicitly declare their types. The process of deciding a type for an expression is called **Type inference** and we will go more in depth with this in the OCaml section, but there are many ways that type inference can be done, and in fact you already saw one way with our `err.c`

[1] Canonically it will just raise an error and still compile

program. We did not say that `"hello"` was a `char *` yet gcc knew because of the syntax of the datatype. The same holds true for ruby. Integers are numbers without decimal points. Floats are numbers with decimal points. Strings are anything put in quotes.[2] You can check this with the `.class` method. Did I mention that Ruby is object oriented?

1.3 Object Oriented Programming

You should be familiar with Object Oriented Programming (OOP) because of Java, however, unlike Java, everything is an object in Ruby. Lets test this out.

```
1 # oop.rb
2 puts 3.class
3 puts "Hello".class
4 puts 4.5.class
```

You can see that everything, including primitives, are object oriented. Also notice that like java, when calling an object's methods, we use the dot syntax. That means that earlier when we called `a + b` in `add.rb`, it was actually doing something like `a.+(b)`. Don't believe me? Consider the following:

```
1 # program2.rb
2 m = 3.methods
3 puts methods.include?(:+)
4 puts 3.+(4)
```

The Line 2 just gets the methods which the object **`'3'`** has as an array. The Line 3 will then print out **`'true'`** because we are asking if the array of methods includes the **`':+'`** method. This is actually a symbol, but we will talk about that later. We can then call the **`'+'`** method on 3 adding it to 4 and we get **`'7'`** as output. Pretty weird right?

Other properties of OOP also exist in Ruby. As we saw before, objects have methods, and this is the primary way that objects interact with each other. Recall that Objects are instances of Classes so each Object has it's own state. This also means that all values are references to objects (so be careful how you check to see if two values are equal). Additionally, Ruby has an inheritance structure similar to Java. In Ruby, all classes are derived from the `Object` class.

```
1 # oop.rb
2 puts 3.class
3 puts 3.class.ancestors
4
5 a = "hello"
6 b = a
7 puts a.equal?(b)         #true
8 puts a.equal?("hello")   #false
9 puts "hello" == "hello" #true
```

[2] We will see how ruby does this when we talk about parsing. In particular Project 4

Object oriented programming in Ruby also means that we need some sort of value to represent the absence of an object. In java it was called **'null'**, in Ruby, we call it **'nil'**. `nil` actually is an object itself and has methods which you can use.

```
1 # nil.rb
2 puts nil.methods
3 puts nil.to_s
```

1.3.1 Class Creation

Let's make our first class

```
1  # square.rb
2  class Square
3    def initialize(size)
4      @size = size
5    end
6
7    def area
8      @size*@size
9    end
10 end
11
12 s = Square.new(5)
13 puts s.area
```

There is a lot here, let's break it down.

- Lines 1 and 9 is the outline of the class. The name of the class is Square
- lines 2-4 is the Ruby equivalent to the constructor.
- lines 3 and 7 use `@size` which is a instance variable
- lines 6-8 is a instance method
- Line 11 is the instantiating of the newly made Square class
- line 12 is calling the instance method

The Equivalent Java code if below.

```java
// Square.java
public class Square{
  private int size;
  public initialize(size){
    this.size = size;
  }

  public int area(){
    return size*size;
  }
}

public static void main(String[] args){
  Square s = new Square(5);
  System.out.println(s.area);
}
```

Notice that the instance variable `size` is private which means if we wished to access it, we would need to make getters and setters. We could do this by adding the following

```ruby
# square-1.rb
class Square
# ...
    # getter
    def size
        @size
    end
    #setter
    def size=(s)
        @size = s
    end
end
# ...
```

This is annoying to do for each variable we have so Ruby actually has a built in function to help us: **`attr_accessor`** Consider the following:

```ruby
# square- 2.rb
class Square
    attr_accessor :size
    # ... same as before ...
end
s = Square.new(5)
puts s.area
s.size= 6
puts s.area
puts s.size
```

If you wanted to use `static` or class variables, you just prepend the variable name with **'@@'**. So if you wanted to count how many squares were made, you could do so like so:

```
# square- 3.rb
class Square
    @@count = 0
    attr_accessor :size
    def initialize(size)
        @@count += 1
        @size = size
    end

    def count
        @@count
    end
    # ... same as before ...
end
s = Square.new(5)
s2 = Square.new(6)
puts s.count
```

For class or static variables you need to initialize them and write your own getters and setters. You can also make static methods by defining them in terms of the class. See below.

```
# counter.rb
class Counter
    @@count = 0
    def initialize()
        @@count += 1
    end

    def Counter.counter
        @@count
    end
end
c = Counter.new
c1 = Counter.new
puts Counter.counter
```

One other thing you may have noticed is that we did not use the common **return** keyword. This is because Ruby will return whatever the last line in a function evaluates to. The following 3 methods all do the same thing:

```
# return.rb
def to_s1
    s = "Hello"
    return s
```

```
5  end
6
7  def to_s2
8      s = "Hello"
9      s
10 end
11
12 def to_s3
13     "Hello"
14 end
15  puts to_s1
16  puts to_s2
17  puts to_s3
```

Lastly, we stated earlier that classes are all derived from the `Object` class. This means we must have some form of inheritance. It acts much like Java, the syntax is just different.

```
1  # inheritance.rb
2  class Shape
3    def to_s
4      "I am a shape"
5    end
6  end
7
8  class Square < Shape
9    def to_s
10     super() + " and a square"
11   end
12 end
13 puts Square.new
```

The above created two classes, Shape and `Square`. A `Square` is a subclass of Shape and so it inherits all its methods. If we wish to override our parent's method, we can certainly do so as seen in lines 9-11. If we wish to refer to the parent's method we can do so using the `super` method. In fact we can override any method, but method overloading is not supported. We would get an error should be try to run

```
1  # overload-err.rb
2  class Square
3      def func1(x)
4          puts "func1"
5      end
6
7      def func1(x,y)
8          puts "func2"
9      end
10 end
11 Square.new.func1(2,3) #fine
```

```
12 Square.new.func1(2)   #error
```

But we could do something like the following

```
1  # override.rb
2  class Square
3      def func1(x)
4          puts "func1"
5      end
6
7      def func1(x)
8          puts "func2"
9      end
10 end
11 Square.new.func2(1)
```

This can lead to some pretty interesting behaviour where we can add things to existing Classes. In the following example, I am going to add a new method to the `Integer` class which just returns the double of the value.

```
1  # double.rb
2  puts 3.methods.include?(:double)
3
4  class Integer
5    def double
6      self + self
7    end
8  end
9
10 puts 3.methods.include?(:double)
11 puts 3.double
```

The `self` keyword is similar to java's `this`. It refers to the current object. We can also use this power of overriding methods to break ruby

```
1  # break.rb
2  class Integer
3    def +(x)
4      "Not Today"
5    end
6
7    def -(x)
8      self * x
9    end
10 end
11
12 puts 3+4
13 puts 3-4
```

If we run this in irb, it will crash, but if you save this as a file and run it, you get **'Not Today'** followed by **'12'**.

1.4 Code blocks

Okay, I'll be upfront. I lied earlier when I said everything is an object. Afaik there is only one feature of Ruby which is not object oriented: codeblocks. If you took a look at Section 1.6 you will know that we can create an Array the following way:

```
1 a = Array.new(3,"Item")
```

However, there is another way that you can initialize an array with a default value.

```
1 a = Array.new(3){"Item"}
```

This is an example of a codeblock. Codeblocks are typically surrounded in curly braces({}) but can also be surrounded with do ... end. Codeblocks are not objects so you cannot assign variables to them, nor can you call methods on them. Additionally you cannot pass them into functions as parameters, nor can you return a codeblock as a return value. However there is one important thing to take away from codeblocks: that we can treat code as data. Consider the following:

```
1 def func
2     if block_given?
3         yield
4     end
5 end
6 func1 {puts "hello"}
```

The `yield` keyword on line 3 tells ruby to pass control to the codeblock associated with the function. We can see this more clearly in the following example:

```
1  def func1
2      yield 5
3  end
4
5  def func2(i)
6      y = i+1
7      puts y
8  end
9
10 func1 {|i| puts i + 1}
11 func2(5)
```

Right off the bat, we should acknowledge that codeblocks can take in paramaters when yielded to, as seen on line 2. The syntax for accepting arguments can be shown on line 10, where you suround the arguments in pipbars (|). Anyway, the two function calls on line 10 and 11 have similar behavior., The difference is that when using the codeblock, the parameter 5 is kept constant with the code being executed being variable on all calls of

func1, whereas func(2) can take in a varaible parameter, but the code executed will always be the same. Let's see this more clearly:

```
# similar
func1 {|i| puts i + 1}
func2(5)

#not similar
func1{|i| puts i %2}
func2(6)

func1{|i| Array.new(i)}
func2(3)
```

If you squint, you can consider this similar to passing in a function pointer in C and then calling said function. Notice that control is passed to the codeblock on a yield and then returned when the codeblock finishes executing.

```
# codeblock not executed unless yield is called
def func3
    puts "hello"
end
func3 {puts "World"}

# control passed when yield is called
def func4
    yield 1,2
    yield 3,4
end
func4{|a,b| puts a + b}
```

Again, I will reiterate that codeblocks are not objects which means no passing them in as parameters or returning them.

```
# cannot do
def func5(i)
    yield i
    return { puts "hello"} #cannot do
end

func5({puts "hello"}) # error
```

THere is however a workaround. They are called Procs. Procs create this thing called a closure which we talk about in the OCaml sections. For now, just know that Procs allow us to store codeblocks inside an object.

```
p = Proc.new {puts "Hello"}
puts p.class
```

Because procs are objects and not a codeblock, you cannot yield to them, but there are methods you can call from a proc. To execute the code stored in a proc, we can use the `.call` method. Much like a codeblock, the body of a proc is not executed until the `call` method is called.

```
def func6(p)
    p.call
    p.call
end
p = Proc.new {puts "hello"}
func6(p)
```

Procs can also take multiple arguments, and afaik, unlike codeblocks, can be nested in eachother. For example

```
def func7(x)
    p = Proc.new {|y| Proc.new {|z| x + y + z}}
    return p
end
a = func7(1)
b = a.call(2)
c = b.call(3)
puts c
```

Because Procs are objects, we can do some fun things:

```
def map(arr,func)
    for value in arr
        puts func.call(value)
    end
end
map([1,2,3,4],Proc.new{|i| i + 1})

def execute(arr)
    for func in arr
        func.call
    end
end
funcs = [Proc.new{puts "hello"}, Proc.new{puts "Bye"}, Proc.new{
    puts "C you"}]
execute(funcs)
```

1.5 Modules

Now that we talked about the one thing that is not object oriented, let's go back and talk about one issue with object oriented programming in Ruby (and Java): inheritance restrictions. In these languages, we have the feature of inheritance, but we can only have one

parent class which is not entirely feasible. In java we got around this with interfaces. In ruby, we can use modules.

Let's write our first module and then we can see how to go about using it:

```
module Doubler
    def Doubler.base
        2
    end

    def double
        self + self
    end
end
```

This module has both a static and an instance method. The syntax for this module is similar to Ruby's Class creation. We create static methods with the `<Classname>.<method>` syntax and create instance methods using the common `def...end` keywords. There are a few things to note about Modules that make them different from classes:

- Modules cannot be instantiated
- Modules use the `module` keyword instead of the `class` keyword
- cannot be extended like a class

That's it. Pretty simple. We cannot extend modules but we can still overwrite them; But we are getting ahead of ourselves. Let's just first see how we use them.

```
class Integer
    include Doubler
end
puts 10.double
```

Here we are adding the `Doubler` module to the `Integer` class so any integer now has access to the `doubler` method. We can also do things like

```
puts Doubler.base
puts Doubler.class
puts Doubler.instance_methods
```

But because we cannot instantiate we cannot do

```
Doubler.new
Doubler.double
```

The only thing that may be confusing with Modules is when it comes to overwritten methods. Consider the following:

```
module M1
    def bye
        "Goodbye"
    end
```

```
end

module M2
    def bye
        "Bye"
    end
end

class C
    include M1
    include M2
end

puts C.new.bye
```

In this case, we load modules in the order we include them so M2 has the last instance of defining bye so M2's bye method will be called. Had we swapped the order:

```
class C
    include M2
    include M1
end
```

Then M1's bye method would be called instead. If we had an instance method in the C class, then we would call C's bye method.

```
class C
    include M2
    include M1

    def bye
        "C ya"
    end
end
```

Typically the order in which something is called is by first looking at self, then the self's modules, then the parent's instance methods, then the parent's modules, then the grandparent's instance methods, then the grandparent's modules, etc. We can see that here:

```
module M1
    def bye
        "Goodbye"
    end
end

module M2
    def bye
        "Bye"
    end
```

```
11 end
12
13 class C
14     include M1
15 end
16
17 class D < C
18     include M2
19 end
20 puts D.new.bye
```

If you are ever unsure of the order, you can always use the `.ancestors` method.

```
1 puts D.new.class.ancestors
2 # [D,M2,C,M1,Object,Kernel,BasicObject]
```

There is one important thing to note: once something is loaded, it will not be loaded again.

```
1 class C
2     include M2
3     include M1
4 end
5
6 class D < C
7     include M2
8 end
9 puts D.new.bye
```

Some Modules we have kinda seen before, namely the `Comparable` and `Enumerable` modules. Any Class that includes the `Comparable` module supports `<.>„<=,>=,=` operators. Classes that include the `Enumerable` allow things like `map` and `select`.

1.6 Data Types and Syntax

That is pretty much all you need to know about Ruby for this course for now. All that's left is to go over syntax of data types and other things.

1.6.1 Numbers

There are two common types of numbers: `Integers` and `Floats`. An `Integer` is a positive or negative integer value without a decimal point. A `Float` is a positive or negative value with a decimal point and at least one digit on either side of said point. When performing operations between the same types, the resulting value is the same type. When performing an operation which involves a `Float`, the resulting value is typically also a `Float`. For some reason, Ruby also allows you to use an underscore as a separator. Maybe for readability?

```
1 # numbers.rb
2 -1 + 1 # addition between Integers
```

```
6.5 % 1.2 # modulus between Floats

2. # not valid for floats
.1 # also not valid

3.0/2 # will result in a Float

1_000_000 == 1000000 # true
```

1.6.2 Stings and Symbols

Strings in ruby are anything in-between double or single quotes. Since things are Objects in Ruby, Strings follow structural equality, but not physical equality. You can nest single and double quotes if you want to print one or the other.

```
# strings.rb
"String 1"
'String 1'
'String' == "String"      # true
"string".equal?("string") # false
```

Symbols on the other hand are special strings, but only one of each symbol exits meaning they are physically equal. Since they are physically equal, they are also structurally equal. A symbol can be any valid string but is written with a `':'` in the front. You can add quotes if you have a multi-word symbol. We have seen symbols when using `attr_accessor` and `.methods`.

```
# symbol.rb
:"String 1"
:'String 1'
:"string".equal?(:"string") # true
:"string".equal?(:'string') # true
:"string".equal?(:string)   # true
```

1.6.3 Arrays

Arrays use the very common bracket syntax for both creation and indexing. Unlike in most languages, arrays can be heterogeneous which is nice. Ruby arrays also support dynamic sizing and set operations. Any value not initialized because `nil`. One important thing to note is that when dealing with n-Dimensional arrays, you must always have the previous dimension declared.

```
# arr.rb
# creating
arr = []
arr = [1,2,3,4]
arr = [1,2.0,"hello"]
```

```ruby

arr = Array.new(3) # [nil,nil,nil]
arr = Array.new(3,"a") # ["a","a","a"]

# indexing
a = [1,2,3,4]
puts a[0]   # 1
puts a[-1]  # 4

# dynamic sizing
arr = []
arr[4] = 5
puts arr # [nil,nil,nil,nil,5]

# set stuff
a = [1,2,3,4,5]
b = [4,5,6,7,8]
puts a+b # [1,2,3,4,5,4,5,6,7,8]
puts a|b # [1,2,3,4,5,6,7,8]
puts a&b # [4,5]
puts a-b # [1,2,3]

#adding and removing
a = [1,2,3]
a.push(4)
puts a # [1,2,3,4]
a.pop
puts a # [1,2,3]
a.unshift(0)
puts a # [0,1,2,3]
a.shift
puts a # [1,2,3]
a.delete_at(1)
puts a # [1,3]
a.delete(3)
puts a # [1]

a2d = [][] # error
a2d = []
a2d[0] = []
puts a2d # [[]]

# you can also use a code block
a2d = Array.new(3){Array.new(3)} # create a 3x3 matrix
puts a2d #[[nil,nil,nil],[nil,nil,nil],[nil,nil,nil]]
```

Unlike some languages, Hashes are built into Ruby. This means you don't have to make your own hashing mechanism or hash function (though you should if you are doing this for security purposes). Ruby uses the common curly brace syntax for creation and the bracket syntax to index. If a key does not exist in the hash, it is automatically mapped to nil. You can change the default hash if you want. Hashes in ruby are very much like arrays, except instead of mapping numbers (or indexes) to values, you can map anything to anything. That is to say, keys do not have the be the same amongst each other and the same for values. Keys and values also do not have to have the same type. Like Arrays, when dealing with n-Dimensional arrays, you must always have the previous dimension declared.

```ruby
# arr.rb
# creating and indexing
h = {}
h = {"key" => :value,}
h = Hash.new
puts h['key'] # nil
h = Hash.new(:default)
puts h['key'] # :default

# adding
h = {}
h['key1'] = :value1
puts h # {'key1'=>:value1}
h.delete('key1')
puts h # {}

# Multi-dimensional Hashes
h = {}
h[0] = {}
h[0][0] = 4
h2 = {}
h2[0][0] = 4 # error
```

1.6.4 Control Flow

The most simple version of control flow is the `if` statement. You should know what an `if` statement is by now so I won't discuss what they are or how they work. Instead lets talk about the bigger class of statements: control statements. Control statements control the flow of program execution; More specifically they alter which command comes next. There are several in Ruby: `if, while, for, until, do while` the main ones, but most people just use the first 3. For those that have a boolean check, **`'true'`** is anything that is not **`'false'`** or **`'nil'`**. **`'nil'`** is like null, it is used for initialized fields. however, **`'nil'`** is an object itself of the `NilClass`. **`'true'`** and **`'false'`** are also objects of `TrueClass` and `FalseClass` respectively. Note:**`FalseClass` and `NilClass` do not evaluate to false**. Consider the following:

```ruby
# conditional.rb
count = 1
while count >= 0
    if 3 > 4 then # then is optional
        puts hello
    elsif nil
        puts "nil is true"
    else
        if count == 0
            puts FalseClass == false
        end
        puts NilClass == false
    end
    count -= 1
end
```

You should run this code to see what happens but but here are 3 important things

- on line 5, `hello` is an undefined variable but Ruby never catches this. Since Ruby is dynamically typed, this bug goes unnoticed.
- instead of `elseif` or `else if`, ruby uses `elsif`. Why, I have no idea. This is a common bug
- the `end` keyword is commonly used whenever you would otherwise use } in other languages

You can read more at the Ruby Docs.