Ruby Recap: Building Blocks

The Ruby Programming Language

Let's recap what we learned about Ruby from the prep work.

The Ruby Programming Language

You all did the prep work, right?

RIGHT?

The Ruby Programming Language

Ruby is a programming language that favors simplicity of writing and understanding code, while still remaining very **powerful**.

You can do a lot of things in Ruby, but we will use it to write our Web applications.

The Ruby Programming Language

“I hope to see Ruby help every programmer in the world to be productive, and to enjoy programming, and to be happy. That is the primary purpose of Ruby language.”

Yukihiro "Matz" Matsumoto

Running your Ruby code

You should be running your code in the terminal with the ruby command.

```
$ ruby your_program.rb
```

Printing to the screen

The most common way you will be printing messages to the screen is puts:

```
puts "Well hello there"
```

You could also use print.

```
print "Hello to you as well"
```

Printing to the screen

Who can tell me the difference between the two?

```
puts "Well hello there"
```

```
print "Hello to you as well"
```

Comments

You make comments in the code with the # symbol.

```
# This line will be ignored
puts "This line is going to be executed"
```

Comments

```ruby
# This line will be ignored
puts "This line is going to be executed"
```

Use them to take notes or temporarily remove code from your program's execution without actually deleting anything.

Values & variables

You assign variables with the *equal sign*:

```
favorited_food = "Pizza"
```

Variables are like boxes that store values.

Values & variables

Once you store a value in a variable you can use it as if it was that value.

```
favorited_food = "Pizza"

puts "My favorite food is:"
puts favorited_food
```

Values & variables

ALL variables in Ruby are mutable. You can change their value whenever you want.

```
favorited_food = "Pizza"
puts "My favorite food is:"
puts favorited_food

favorited_food = "Sushi"
puts "My favorite food is:"
puts favorited_food
```

Values & variables

Variables can store any kind of values. There are a few basic types we will talk about today.

It's important to remember that anywhere you could use a regular value, you could substitute that with a variable that contains the same kind of value.

Values & variables

In Ruby, *every single value is an object*. You will learn more about object oriented programming in another section, but for now keep in mind that every value has **methods** attached to it.

```
puts 42.zero?

puts "thugs".include?("hugs")
```

Strings

First, text values. They are known to programmers as **strings**.

Just surround any text by quotations. Both single quotes and double quotes work.

```
str = "a string"

str2 = 'another string'
```

Strings

Strings have a size, the amount of characters in the string.

```
str = "a string"
puts str.size # => 8

str2 = 'another string'
puts str2.size # => 14
```

Strings

Strings can be stitched together with the *plus sign*:

```
name = "hal"
puts "My name is" + name + ". It's
nice to meet you"

puts "You can concatenate" + "any
amount of strings" + "together."
```

We call this string **concatenation**.

Strings

You only really have to do this when there's a variable involved.
This would be better written as one large string.

```
puts "You can concatenate" + "any
amount of strings" + "together."

puts "You can concatenate any amount
of strings together."
```

Strings

Often it's better to use string **interpolation** instead of concatenation.

Interpolation is when you insert a value into a larger string. In Ruby you use the `#{}` in a string.

```
name = "hal"
puts "My name is " + name + ". It's
nice to meet you"

puts "My name is #{name}. It's nice to
meet you"
```

Strings

String interpolation only works with double quote strings:

```ruby
name = "hal"
puts "My name is #{name}."
#=> Double quotes: My name is Hal.

puts 'My name is #{name}.'
#=> Single quotes: My name is #{name}.
```

Strings

Don't be the person to try to interpolate with single quotes.

```
#=> Just don't...

puts 'Single quotes: My name is #{name}.'
```

Strings

Aside from string interpolation, what other differences are there between single quotes and double quotes?

```
puts "Hey\sthere."
puts 'Hey\sthere.'

puts"Look under me\n^^^^^^^\nLook above me"
puts 'Look under me\n^^^^^^^\nLook above me'
```

Numbers

There are also number values. As expected you can do math with them:

```
add = 39 + 84
subt = 567 - 40
mult = 6 * 7
division = 42 / 7
```

Numbers

You can see that if the result was supposed to be a decimal number it eat up the decimal part.

```
puts 40 / 50
#=> 0

puts 3 / 2
#=> 1
```

Numbers

But if you make either of those numbers a decimal number it works.

Decimal numbers are known as **floats** in programming (for *floating point*).

```
puts 40 / 50.0
#=> 0.8

puts 3.0 / 2
#=> 1.5
```

Numbers

Remember, you can do math partially or entirely with variables, so long as those variables have numbers in them.

```
three = 3
fifty_two = 52

puts three * fifty_two
```

Arrays

Now on to **arrays**. Remember, an array is just a list (order matters) of other kinds of values.

```
sweets = [] # empty array
sweets = [ "cookies" ] # array with one element
sweets = [ "cookies", "ice cream", "pie", "crème brûlée" ]
```

Arrays

You can access elements in the array by their assigned number, also known as the **index**.

```
sweets = [ "cookies", "ice cream", "pie", "crème brûlée"]
#            0            1          2            3

puts sweets[0] #=> cookies
puts sweets[3] #=> crème brûlée
puts sweets[99] #=> nil
puts sweets[-1] #=> crème brûlée
puts sweets[0..2] #=> [“cookies”, “ice cream”, “pie”]
puts sweets[1..-1] #=> [“ice cream”, “pie”, “crème brûlée”]
```

Arrays

Like strings, arrays can tell you their size:

```
sweets = [ "cookies", "ice cream", "pie", "crème brûlée"]

puts sweets.size
#=> 4
```

Arrays

You can add things to an array after the fact with the **push** method or the << operator.

```
sweets = [ "cookies", "ice cream", "pie", "crème brûlée"]
sweets << "Reese's Minis"
sweets.push("Heath bar")

puts sweets.size
#=> 6

puts sweets[4]
#=> Reese's Minis
```

Arrays

There's nothing stopping you from having an array with mixed types in there. This is *rarely* useful though.

```
number = 74
message = "hello"

stuff = [ "some strings", 45, "numbers as well", number,
message ]
```

Arrays

Keep in mind that there's nothing stopping you from trying to access an index that doesn't exist.

You will get nil.

```
sweets = [ "cookies", "ice cream", "pie", "crème brûlée" ]

puts sweets[9000]
#=> nil
```

Hashes

Hashes are like arrays except you access its contents by a name, known as the **key**, and has no implicit order.

It's like a super variable that contains a bunch of variables.

```
pizza = {}
pizza = {
  :cheese => "Mozzarella",
  :sauce => "Marinara"
}
```

Hashes

To access one of the values inside the hash, you use a syntax similar to that of arrays, except you use the *key* instead of an index.

```
pizza = {
   :cheese => "Mozarella",
   :sauce => "Marinara"
}
puts pizza[:sauce]
#=> Marinara
```

Hashes

Why would you use hashes? Sometimes a list of values don't have an inherent order so using an array doesn't make a ton of sense.

```
the_godfather_translations = [
  "The Godfather",
  "El padrino",
  "Il padrino",
  "La Baptopatro",
  "Der Pate"
]
```

Hashes

How do you know which of these translations is in Esperanto?

It's the_godfather_translations[3]. Not very intuitive.

```
the_godfather_translations = [
  "The Godfather", #0
  "El padrino", #1
  "Il padrino", #2
  "La Baptopatro", #3
  "Der Pate" #4
]
```

Hashes

It's better if these values were labeled for easy and intuitive access. Let's use a hash.

```
the_godfather_translations = {
  :english => "The Godfather",
  :spanish => "El padrino",
  :italian => "Il padrino",
  :esperanto => "La Baptopatro",
  :german => "Der Pate"
}

puts the_godfather_translations[:esperanto]
```

Hashes

Using keys to access specific values like this makes a lot more sense.

```
the_godfather_translations = {
  :english => "The Godfather",
  :spanish => "El padrino",
  :italian => "Il padrino",
  :esperanto => "La Baptopatro",
  :german => "Der Pate"
}
puts the_godfather_translations[:esperanto]
puts the_godfather_translations[:english]
puts the_godfather_translations[:italian]
```

Symbols

Used as identifiers. Most common use is as hash keys.

```
:i_am_a_symbol
```

Symbols for hash keys

It's a best practice to use symbols for hash keys. It's so common, that we have special syntax for it.

```ruby
the_godfather_translations = {
  :english => "The Godfather",
  :spanish => "El padrino",
  :italian => "Il padrino",
  :esperanto => "La Baptopatro",
  :german => "Der Pate"
}
```

```ruby
the_godfather_translations = {
  english: "The Godfather",
  spanish: "El padrino",
  italian: "Il padrino",
  esperanto: "La Baptopatro",
  german: "Der Pate"
}
```

Conditionals

Here are the basic ways that things can be compared in Ruby.

```
puts "Is favorite_food 'hot dogs'?"
puts favorite_food == "hot dogs"

puts "Is number greater than 9000?"
puts number > 9000

puts "Is number less than 9999?"
puts number < 9999
```

Conditionals

Their opposites:

```
puts "Is favorite_food NOT 'hot dogs'?"
puts favorite_food != "hot dogs"

puts "Is number less than or equal to 9000?"
puts number <= 9000

puts "Is number less than 9999?"
puts number >= 9999
```

Conditionals

Remember that for equality it's **two equal signs**. It's a common beginner mistake to only use one.

```
# Variable assignment
puts i = 1

# Equality comparison
puts i == 1

# Equivalence (another story)
puts i === 1
```

Conditionals

You can use && (and) to make a condition more **specific** by adding more **constraints**.

```
puts "Is your favorite food pizza AND your favorite
drink tea?"
puts favorite_food == "pizza" && favorite_drink ==
"coffee"

puts "Is number greater than 9000 AND less than 9999?"
puts number > 9000 && number < 9999
```

Conditionals

You can use `||` (or) to make a condition more **general** by adding more **options**.

```
puts "Is your favorite food pizza OR sushi?"
puts favorite_food == "pizza" || favorite_food ==
"sushi"

puts "Is the password's length too short OR is it equal
to 'password'?"
puts password.size < 8 || password == "password"
```

Conditionals

Conditionals are primarily to be used inside **if..else** statements.

```
if favorite_food == "pizza"
  puts "Yes, exactly."
else
  puts "What do you like to eat, then? Sandwiches?"
end
```

Conditionals

Remember, you don't always need an `else`.

```
if favorite_food == "pizza"
  puts "Yes, exactly."
end

if favorite_dessert == "cookies"
  puts "I like your style."
end
```

Conditionals

There's also **elsif** to handle additional cases. You can add as many as you want.

```
if favorite_food == "pizza"
  puts "Pizza is amazing"
elsif favorite_food == "hot dogs"
  puts "Hot dogs taste good"
elsif favorite_food == "chili"
  puts "Chili can be dangerous"
elsif favorite_food == "steak"
  puts "Steak can be expensive"
else
  puts "Uh... #{favorite_food} can be okay. I guess."
end
```

Conditionals

Don't forget your **end**.

Indent your `if..else` statements to make it easy to identify when an `end` is missing.

Conditionals

Every value in Ruby has an inherit *truthiness* or *falsiness*. That means you can use values as conditionals.

```
if 42
  puts "42 is truthy!"
end

if nil
  puts "nil is falsy, so this message won't show up."
end
```

Conditionals

The only values that are considered *falsy* are `nil` and `false`.

Everything else is considered *truthy*.

Conditionals

Some examples:

```ruby
if "hello world"
  puts "'hello world' is truthy"
end

if 42
  puts "42 is truthy"
end

if 0
  puts "0 is truthy"
end
```

Conditionals

More examples:

```ruby
if ""
  puts "The empty string is truthy"
end

if false
  puts "false is falsy"
end

if nil
  puts "nil is falsy"
end
```

User input

You can receive input from the user with `gets`.

```
puts "What's your favorite food?"

favorite_food = gets
```

User input

To avoid getting the `\n` from the user hitting the return or enter key, use the `chomp` method.

```
puts "What's your favorite food?"

favorite_food = gets
#=> "pizza\n"

favorite_food = gets.chomp
#=> "pizza"
```

User input

The `chomp` method isn't some `gets` magic. It's a method that all strings have.

```
str = "pizza\n"

puts str.chomp
#=> "pizza"
```

Writing and reading files

You can also write and read files in Ruby.

The `IO.read` method will return the contents of a file as a *string* if you specify which file you are talking about.

```
menu = IO.read("menu.txt")

puts menu
```

Writing and reading files

To write to a file you use IO.write.

You not only need to specify which file to write to, but also the contents you want the file to have.

```
IO.write("menu.txt", "PIZZA")
```

Writing and reading files

Keep in mind that the file's previous contents will be completely overwritten.

After running this code, menu.txt will only contain *PIZZA*.

```
IO.write("menu.txt", "PIZZA")
```

Ruby Recap: Conclusion

That was a lot of stuff, but don't worry.
Everything will become more clear as you practice using these concepts.

In the next 8 weeks, you will get **a lot** of practice.