Programming for Everybody

7. Refactoring

le wagon

Why refactoring?

to make programmers' life easier Ruby has a lot of *syntax shortcuts* that can help us write code in a faster, cleaner and more efficient way

One-line if / unless

when the block inside a conditional statement (like *if* or *unless*) is a short, simple expression we can write the entire statement in one line only

```
age = 20

if age >= 18
   puts “you can vote!”
end
```

same as

```
puts “you can vote!” if age >= 18 ✓

(if age >= 18 puts “you can vote!” ✗ )
```

the order of the elements matters!

the syntax and order of elements is: expression + if/unless

One-line if/else statement

a quicker and more concise version of a simple if-else statement is the **ternary conditional expression**

it takes three arguments: a condition (followed by a question mark) + some code to execute if the condition is *true* (followed by a colon) + some code to execute if the condition is *false*

condition ? do this if true : do this if false

```
age = 25

puts age >= 18 ? “You can drive" : “You can’t drive”

(prints out “You can drive”)
```

Case statement

a quicker and more concise option for when we're dealing with multiple if and elsifs statements is the **case statement**

```
puts "Which language are you learning?"
language = gets.chomp
```

```
case language
    when "ruby"
        puts "Web apps"
    when "css"
        puts "Style"
    when "html"
        puts "Content"
    else
        puts "Sounds interesting!"
end
```

or

```
case language when "ruby" then puts
"Web apps" when "css" then puts
"Style" when "html" then puts "Content"
else puts "Sounds interesting!" end
```

Implicit return

unlike most programming languages, Ruby's methods will implicitly *return* the result of the last evaluated expression even if we don't specifically type the keyword "return"

```
def sum(a, b)
    return a + b
end
```
✗

```
def sum(a, b)
    a + b
end
```
✓

both print out the same result, but the second is more concise

Upto & downto

if we know the range of numbers we'd like to loop through, instead of a for loop we can use the **.upto** and **.downto** methods

```
for num in 95..100
    print num , " "
end
```
✗

```
95.upto(100) { | num | print num, " " }
```
✓

both print out the same result, but the second is more “Rubyist”

One-line Blocks

when a block (aka the code inside a method) takes just one line we should write the entire method as a one-liner and use curly brackets instead of “do” and “end”

```
[“zoe”, “zack”].each do l name l
    puts name.capitalize
end
```

✗

```
[“zoe”, “zack”].each { l name l puts name.capitalize }
```

✓

both print out the same result, but the second is more “Rubyist”

Adding to an array

to add an element to the end of an array, instead of using the .push method we can simply use **<<** operator (known as *the shovel*)

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

print my_array.push(4)

prints out [1, 2, 3, 4]
```

same as

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

print my_array << 4

prints out [1, 2, 3, 4]
```

Conditional assignment

usually we use the = operator to assign a value to a variable

but if we only want to assign a value to variable if that variable hasn't been assigned a value before, we can use the *conditional assignment operator* (**||=**)

```
teacher = "Mariana"
teacher ||= "John"

puts "Today's teacher is #{teacher}!"

prints out "Today's teacher is Mariana"
```

```
teacher = nil
teacher ||= "John"

puts "Today's teacher is #{teacher}!"

prints out "Today's teacher is John"
```

this trick is useful to assign an alternative value to a variable in case that variable is false, nil or undefined

Thank you! :)

le wagon