Programming for Everybody

7. Refactoring

le wagon

The beauty of Ruby

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 on a single line

The syntax and order of elements is: `expression if boolean`

```
age = 20

if age >= 18
  puts "you can vote!"
end

puts "you can vote!" if age >= 18

puts if age >= 18 "you can vote!"
```

the order of the elements matters!

Ternary operator

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

It's in three parts: 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 vote" : "You can’t vote"

# prints out "You can vote"
```

Case statement

A quicker and more concise option for when we're dealing with multiple `if` and `elsif` statements evaluating the value of the same variable, 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
```

Case statement (cont.)

If the statements are short, we can refactor in single lines

```
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 use the `return` keyword

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

✗

```ruby
def sum(a, b)
    a + b
end
```

✓

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

Exception: we will need to use `return` within a method if we need a result to be returned before its last expression

Conditional assignment

We can use the **=** operator to assign a value to a variable, but if we want to assign a variable *only* if it hasn't already been assigned, we can use the *conditional assignment operator* **||=**

Ex 1:

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

puts "Today's teacher is #{teacher}!"
# prints out "Today's teacher is Solene"
```

Ex 2

```
teacher = nil
teacher ||= "Nawel"

puts "Today's teacher is #{teacher}!"
# prints out "Today's teacher is Nawel"
```

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 braces instead of **`def`** and **`end`**

```
["zoe", "zack"].each do | name |
  puts name.capitalize
end
```

```
["zoe", "zack"].each { |name| 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 (also known as *the shovel*)

```
my_array = [1, 2, 3]
print my_array.push(4)
```

same as

```
my_array = [1, 2, 3]
print my_array << 4
```

```
# Both print out [1, 2, 3, 4]
```

Thank you.