codecademy

Refactoring

Ruby Case Statement

In Ruby, a `case` statement is a more concise alternative to an `if/else` statement that contains many conditions.

```
tv_show = "Bob's Burgers"

case tv_show
  when "Archer"
    puts "I don't like the voice of
Archer."
  when "Bob's Burgers"
    puts "I love the voice of Bob
Belcher."
  else
    puts "I don't know who voices this
cartoon."
end

# => I love the voice of Bob Belcher.

#In this example, a case statement is used
to check for multiple possible values of
tv_show. Since tv_show is "Bob's Burgers",
the second when is evaluated to true. If
none of the conditions were met, Ruby
would evaluate the else statement.
```

Ruby .respond_to?

In Ruby, `.respond_to?` takes a symbol representing a method name and returns `true` if that method can be called on the object and `false` otherwise.

```
puts "A".respond_to?(:push)
# => false
# Here, the following Ruby code will return false since .push can't be called on a String object.

puts "A".respond_to?(:next)
# => true
# Here, however, the following Ruby code will return true since .next can be called on a String object. Calling .next on the letter "A" would return the letter "".
```

Ruby Short-Circuit Evaluation

When Ruby evaluates expressions containing boolean operators, it uses *short-circuit evaluation*. With `||`, if the expression on the left evaluates to true, it will return `true`. Otherwise, it will check if the expression on the right evaluates to true. If so, the expression returns `true`; otherwise, it will return `false`.
With `&&`, both the expression on the left and the expression on the right have to evaluate to true in order to return `true`. If either expression is false, it will return `false`

```
a = true
b = false
c = true

puts a || b
#Output => true
puts b || a
#Output => true
puts a && c
#Output => true
puts a && b
#Output => false
```

Ruby Ternary Operator

In Ruby, a *ternary* operator is a more concise alternative to an `if/else`. It consists of a *conditional*, followed by `?` and an expression to be evaluated if the conditional is `true`, and then `:` and an expression to evaluate if the conditional is `false`.

```
tacos_eaten = 12

puts tacos_eaten >= 5 ? "Sir, you've had enough!" : "Keep eating tacos!"

# => Sir, you've had enough!
```

Ruby .upto and .downto Methods

In Ruby, the `.upto` and `.downto` methods are used to iterate over a specific range of values.

```
"B".upto("F") { |letter| print letter, " " }
# => B C D E F

5.downto(0) { |num| print num, " " }
# => 5 4 3 2 1 0

#In both examples, Ruby iterates over specified ranges using the initial value, a .downto or .upto method, and a final value. Each element is passed into the block following the .upto or .downto method.
```

Ruby Conditional Assignment Operator

In Ruby, a *conditional assignment operator* (`||=`) assigns a real value to a variable only when its current value is `false` or `nil` . Otherwise, Ruby keeps its original value.

```
boyfriend = nil

boyfriend ||= "Jimmy Jr."

boyfriend ||= "Josh"

puts boyfriend
# => "Jimmy Jr."

# In this example, since the original value of boyfriend is set to nil which is nothing, Ruby assigns it a value of "Jimmy Jr." on the following line. Once boyfriend holds this real value, another reassignment is overlooked by Ruby and the previous value holds.
```

Ruby .push Method Alternative

In Ruby, an alternative to the `.push` method is the concatenation operator `<<` which can be used to add an element to the end of an array or a string.

```
array = [1, 2, 3]
array << 4
print array
#Output => [1, 2, 3, 4]

puts "Hello," << " welcome to Codecademy."
#Output => Hello, welcome to Codecademy.
```

Ruby “if” Statement Short Expression

In Ruby, the `if` statement can be expressed in a single line in the case of a short expression. This single line would consist of an expression followed by the `if` keyword and finally an expression that evaluates to either `true` or `false`.

```
num = 6

if num % 2 == 0
  puts "This number is even!"
end

#Refactored, this can be stated in
a single line as demonstrated below:
puts "This number is even!" if num % 2 ==
0
```

Ruby Implicit Return

In Ruby, the `return` keyword in a method can be omitted making it an *implicit return*, in which Ruby automatically returns the result of the last evaluated expression.

```
def product(x,y)
  x * y
end

product(5, 4)
# => 20
#In this example, Ruby evaluates the
product method and returns 20 even though
the return keyword was omitted.
```