Programming for Everybody

5. Methods & Blocks

The *sort* built-in method

the sort method is one of many Ruby’s built in methods

it sorts the elements within a collection both from A - Z or from smaller to bigger numbers

```
names = [“Mary”, “John”, “Zack”]
puts names.sort


(prints out John, Mary, Zack)
```

if we want to reverse the sorting, we just use the **reverse method** after the sort method!

The *sort* built-in method

(cont.)

behind the scenes, the sort method is using the *combined comparison operator* **<=>**

this operator compares each element within a collection against all others

- the result is -1 if the first item compared is less than the second
- the result is 0 if the first item compared equals the second
- the result is 1 if the first operand is greater than the second

that's how the sort method decides de order in which the elements should be displayed

Writing our own methods

methods are also known as *functions* in other languages (ex: JavaScript)

methods are **reusable** lines of code written to perform a repeatable and specific task

they are mathematical functions that can take one or multiple **parameters** and **arguments** (inputs) to compute calculations using those inputs and then return a result

Why methods?

to reuse code

they help keeping the code organised by separating the different parts of the app: specific methods execute specific tasks

this makes the code easier to manage: as it becomes more complex, bigger issues are easier to solve if the whole logic is divided into smaller methods

Method syntax

methods have 3 parts:

header includes the **def** (short for "define") keyword, the **name** of the method and any **parameters** the method may take

body includes the lines of code that determine the procedures the method carries out

end a method is closed using with the **end** keyword

```
def my_method
  puts "Hello"
end
```

```
def my_other_method(x, y)
  puts x * y
end
```

Calling a method

after defining a method we have to call it by **typing its name:** that's what triggers the program to look for a method with that name and then execute the code inside it

```
def my_method(x, y)
    puts x * y
end
```

parameters
the placeholder(s) we put put between the method's parentheses when we **define it**

```
my_method(2, 6)
```

arguments
the elements/values we put between the method's parentheses when we **call it**

```
(prints out 12)
```

Returning

sometimes we don’t want a method to print something to the console, but we just want it to hand us back a value which we can use afterwards -> that’s what the **return** keyword does

when a methods returns, the value we get becomes available within the code and can thus be reused

```
def double(n)
    return n * 2
end

output = double(6)
output += 2
puts output    (prints out 14)
```

not printing, just giving us back the result

Splat

sometimes methods may not know how many arguments they’ll be taking and the solution for that is **splat** -> *

a parameter with the splat operator allows the method to expect one or more arguments

```
def what_up(*friends)
  friends.each { | friend | puts "Hi, #{friend}!" }
end

what_up("Ian", “Zoe", “Zenas”, "Eleanor")

#prints out:
Hi, Ian!
Hi, Zoe!
Hi, Zenas!
Hi, Eleanor!
```

✓

vs.

```
def what_up(friends)
  friends.each { | friend | puts “Hi, #{friend}!" }
end

what_up("Ian", “Zoe", "Zenas", “Eleanor")

#prints out:
wrong number of arguments (given 4, expected 1)
```

✗

Blocks

blocks are chunks of code between curly braces **{}** or between the keywords **do** and **end** that we can associate with method invocations

often a method takes a block as a parameter (that's what .each has been doing this whole time, for instance!)

unlike methods, blocks can only be called **once** and in the **specific context** under which they were created

```
names = [“Zoe”, “John”, “Zack”]

names.each do l name l
  puts reversed_name = name.reverse
end
```

```
names = [“Zoe”, “John”, “Zack”]

names.each { l name l puts
reversed_name = name.reverse }
```

Thank you! :)

le wagon