Programming for Everybody

5. Methods & Blocks

le wagon

The sort built-in method

The `sort` method is one of Ruby's many built-in methods

It sorts the elements within a collection, from A - Z or from smaller to larger

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

# Will print 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 the others

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

That's how it decides the order in which the elements are to be displayed

Writing our own methods

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

Methods are **reusable** pieces of code, written to perform a repeatable and specific task

They are mathematical functions that can take one or multiple **parameters** (variables), and **arguments** (values), to compute calculations with said values, and then return a result

Why methods?

They are reusable

They help keeping the code organized by separating the different tasks of the app:

A specific method executes a specific task

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:

The **header** with the **def** (short for “define”) keyword, the **name** of the method and any **parameters** the method takes

The **body** includes the lines of code which determine the procedures the method carries out

The **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

Once the method is defined, we have to **call** it by using its **name:** this triggers the program to look for a method with that name, and then execute the code inside of it

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

my_method(2, 6)

# Will print out 12
```

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

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

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 method returns, the value we get (as the *caller*) becomes available and can thus be used

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

output = double(6) # output holds the value 12
output += 2 # 12 + 2, stored back into output
puts output # Will print 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 do |friend|
    puts "Hi, #{friend}!"
  end
end

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

# Will print out:
Hi, Ian!
Hi, Zoe!
Hi, Zenas!
Hi, Eleanor!
```

vs.

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

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

# Will print 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**, and are an argument of a method. (That's what we've been doing with `.each` 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 | name |
  puts reversed_name = name.reverse
end

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

Thank you.