Blocks

Like methods, but without a name

Blocks are one of the things programmers absolutely love about Ruby. They are an extremely powerful feature that allows us to write very flexible code. At the same time they read very well, and they are used all over the place.

So, what is a block?

A block, essentially, is the same thing as a method, except it does not have a name, and does not belong to an object.

I.e. a block is an anonymous piece of code, it can accept input in form of arguments (if it needs any), and it will return a value, but it does not have a name.

Moreover, blocks can only be created by the way of passing them to a method when the method is called.

A block is a piece of code that accepts arguments, and returns a value. A block is always passed to a method call.

Let's jump right in:

```
5.times do
  puts "Oh, hello from inside a block!"
end
```

As you can see **`times`** is a method that is defined on numbers: `5.times` calls the method **`times`** on the number `5`.

Now, when this method is called the only thing passed is a **block:** that is the anonymous piece of code between **`do`** and **`end`**. There are no objects passed as arguments to the method **`times`**, instead it just passes a block.

The method `times` is implemented in such a way that it simply calls (executes) the block 5 times, and thus, when you run the code, it will print out the message `"Oh, hello from inside a block!"` 5 times.

The code almost reads like an English sentence *Five times do output this message*, right? It does, and that's one of the reasons why Rubyists love using blocks.

One of the things that seem rather hard to grasp about blocks is that

- First, they are anonymous chunks of code.
- Second, they are passed to methods just like other objects.
- And third, they still can be called (just like methods), from inside the method that it was passed to.

Does that make sense?

Imagine you are the object that represents the number `5`. You are a number and you do know your own value.

Now I hand you a piece of paper saying: `Please print the following on the screen: "Oh, hello!"`, and I ask you to execute this instruction as many times as the value that you know.

You'd go ahead and follow the instructions on the paper, and thus print out the message. You repeat this 5 times, because `5` is the value that you know.

This is pretty much how the method `times` on numbers works, and how blocks work: `times` takes the block (the instructions), and runs it as many times as the value of the number.

To summarize: Methods can not only accept input in the form of objects passed as arguments. They can also accept this one special piece of input, which is an anonymous block of code. And they can then call (execute) this block of code in order to do useful things with it.

Alternative block syntaxes

Next to the syntax shown before, using **`do`** and **`end`**, Ruby comes with an alternative syntax, which uses curly braces for defining a block.

These two statements do exactly the same:

```
5.times do
  puts "Oh, hello!"
end

5.times { puts "hello!" }
```

Both statements define a block, which is passed to the method `times`. And both blocks contain a single line of code.

Blocks can be defined enclosing code in **`do`** and **`end`**, or curly braces **`{}`**.

So, when do you use one or the other syntax?

In the Ruby community there's the convention to use curly braces if you have a single line block and it fits nicely on the same line (as, in our example, it does).

Whenever you need to have more than one line in your block, then you use the syntax using **`do`** and **`end`**. Sometimes people also use the `do` and `end` syntax when they feel it makes the code more readable.

Use curly braces **`{}`** for blocks, when the code fits on one line.

Block arguments

Blocks make a lot of sense on methods that are defined on collections like arrays and hashes.

Let's have a look at some examples with arrays.

In our previous example that used the method `times` our block did not accept an argument. A block that accepts an argument looks like this:

```
[1, 2, 3, 4, 5].each do |number|
  puts "#{number} was passed to the block"
end
```

And, again, this is the same as:

```
[1, 2, 3, 4, 5].each { |number| puts "#{number} was passed to the block" }
```

It is unknown to us why Matz has chosen to not enclose the argument list of a block with round parentheses just like method argument lists. Instead, Ruby wants us to use vertical bars (we call them "pipes").

So, for blocks**, `do |number|`** is the same that is **`def add_two (number)`** for a method definition, except that the method wants a name while a block is anonymous:

`|number|` and **`(number)`** both are argument lists. The first one is used for blocks, the second one for methods.

Block arguments are listed between pipes `|`, instead of parentheses.

Now, when you run the code example above, you'll see the message printed out for each of the numbers contained in the array.

Does that make sense? Again, our code almost reads like an English sentence:

With this array for each of its elements, naming it `number`, output the following message.

The method `each` is defined on arrays, and it does just this:

It takes each of the elements in the array and calls the block, passing the element as an argument. The block can then do whatever you want it to do with the element. In our case we interpolate it into a string and print it out to the screen.

Block return values

Remember that we said a block returns a value just like methods do? So far, in our two examples above, we did not do anything with the return values of the block.

Here's an example that does that:

```
p [1, 2, 3, 4, 5].collect { |number| number + 1 }
```

This will take the array of numbers, and transform it into *another* array.

It does this by calling the **method `collect`** on the original array, which calls the given block for each of the elements, and collects each of the return values returned by the block. The resulting array is then returned by the method `collect`, and printed to the screen.

In other words, the method `collect` uses the block as a transformer. It takes each element of the array, passes it to the block in order to transform it to something else, and then keeps all the transformed values in a *new* array that the method `collect` then eventually returns.

Note that the method `collect` has an alias, which is **`map`**. These are exactly the same methods. Many programmers prefer `map` over `collect` because it is shorter, and also more commonly used in other languages. However, in our study groups we use `collect` more often, because it simply expresses more clearly what the method does.

Use the method `collect` to transform an array into another array.

Here's another example that uses the return value of the block, can you guess what it does?

```
p [1, 2, 3, 4, 5].select { |number| number.odd? }
```

In this case, the method **`select`** uses the block in a different way: as a filter, or criterion, to select values out of the array, and then return a new array with the selected values.

Let's walk through this step by step, under the microscope:

- We create an array `[1, 2, 3, 4, 5]`.
- We then call the method `select` on it, and pass our block as a filter criterion.
- Now, the method `select` calls our block for each of the numbers.
- It first calls the block passing the number `1`.
- We now are inside the block, and a local variable `number` is assigned the object that was passed as an argument, which is the number `1`.
- Inside our block we now call the method `odd?` on this number, and of course, because `1` is odd, this will return `true`.
- Since this is the only, and thus, last statement in the body of our block, our block also returns `true` to the method `select`. `select` therefore will *keep* ("select") the number `1`.
- It then calls the block again, this time passing the number `2`. Of course, because this is not an odd number, the method **`odd?`** and therfore our block will return `false` back the the method `select`. Therefore it *discards* this element.
- It keeps doing this for each of the remaining elements in the array, and eventually has this array: `[1, 3, 5]`
- The method `select` then returns this array and Ruby will pass it to the method `p`, which prints the array out to the screen.

Thus, the code above prints out `[1, 3, 5]`.

Use the method `select` to select a new array with values that match a criteria defined by the block.

Here's another example of a method that uses the block as a criterion:

```
p [1, 2, 3, 4, 5].detect { |number| number.even? }
```

Again, **`detect`** will pass each of the elements of the array to the block, one by one, and check the return value of the block. However, as soon as the block returns something truthy (something that is "equivalent to true"), the method `detect` will return the current object itself. Therefore, this will print out `2`: the first number in the array that is even.

Iterators

Methods on arrays and hashes that take a block are also called iterators.

We say they *iterate over the array*, meaning that these methods take each element of the array and do something with it.

In Ruby iterators are “chainable”, adding functionality on top of each other.

That means that, if you do not pass a block to an iterator method, such as **`each`**, **`collect`**, **`select`**, then you’ll get an *iterator object* back. You can then call more methods on these iterator objects, and finally pass a block. Like so:

```
numbers = [1, 2, 3, 4, 5].collect.with_index do |number, index|
  number + index
end
p numbers
```

This will print out:

```
[1, 3, 5, 7, 9]
```

What’s going on here?

The **method `with_index`** can be called on any iterator object. All it does is pass the index of the element within the array to the block, as a second block argument, in addition to the element itself.

Inside of the block we can then use it, and add the index to the number itself.

So for the first iteration it will call the block with `1` and `0`, since `0` is the first “position”, that is, index. It therefore returns `1`. For the second iteration it calls the block with `2` and `1`, and returns `3`, and so on.

Therefore the method call eventually returns the array `[1, 3, 5, 7, 9]`.

Iterators in Ruby are chainable.

Inversion of control

In Ruby there are a lot more methods that accept blocks, and they do very different things. However, they have one thing in common:

By accepting a block, from you as a programmer, the method can pass control to you.

This design is an example for the principle of *inversion of control*, and it's the real reason why Rubyists love blocks so much.

What does that mean?

In short it means that Matz, the creator of Ruby, put a tool in place that can be used to allow methods to pass control to its users (i.e. you as a programmer).

"Control" in this context refers to the question who gets to decide how things work.

In older languages, where there was no such tool, people either had to implement lots of very specific methods, and *guess* what users might need in the future. Or they'd decide to just not implement any of these methods at all.

For example, in Ruby, we don't have to define lots of methods like `select_odd`, `select_even`, `select_lesser_than`, `select_greater_than` and so on, … where each of these methods would be useful for one very specific usecase.

Instead, the class `Array` only has to implement one single, very generic method for arrays: `select`. Since Ruby has blocks, the method can allow you (as a programmer) to specify the criterion yourself: by passing a piece of code, in the form of block to the method.

That way Ruby lets you, as a programmer, take over control, and specify what is used as a criterion to select elements.

One of the reasons we mention this is because we think this is a nice example of a pretty abstract principle applied to software design. There are lots of other principles like these, and they'll make more and more sense to you over time. Programming languages and code, from this perspective, is a subject of design, and thus art, as well as social and cultural questions … much rather than strictly logical or technical ones.