... continued

This lesson continues the discussion on working with Fibers in Ruby.

Yielding from Fibers

In the previous lesson, we used fibers much like function calls but the true potential of fibers is truly unleashed, when used with `Fiber.yield`. Don't confuse `Fiber.yield` with `yield`, the latter is used to execute arbitrary blocks of code. The `Fiber.yield` serves two purposes:

- Returns control back to the invoker of the `resume()` method on the fiber.
- Return a value to the invoker.
- Receive a value from the invoker.

We'll see examples of each. Consider the following example first:

```
fib = Fiber.new do
  puts "Control passed to fiber 1"
  Fiber.yield
  puts "Control passed to fiber 2"
  Fiber.yield
end


fib.resume()
puts "Control returned to main thread"
fib.resume()
puts "Control returned to main thread"
```

The executions sequence of the above snippet is as follows:

1. When `fib.resume()` is executed on **line#10**, the execution of the fiber starts from **line#3**.

2. The statement `Fiber.yield` on **line#4** when executed returns the control to the main thread on **line#11**. Before returning control back to the main thread, a snapshot of the fiber's stack frame is saved.

3. Executing **line#12** returns control back to the fiber. The snapshot of the stack frame is restored and the fiber continues execution starting from **line#5**. In contrast, the stack frame of a method call is deallocated when the method returns and all local variables are lost.

4. The fiber resumes execution, prints the message and returns control back to the main thread when executing **line#6**.

Returning Values from Fibers

We can return values from a fiber as follows:

```
Fiber.yield <ReturnVal>
```

We tweak the previous example to return an integer from the yield statements. The main thread receives the values as shown below:

```ruby
fib = Fiber.new do
  start = 9
  Fiber.yield start
  start += 1
  Fiber.yield start
end


puts fib.resume()
puts "Control returned to main thread"
puts fib.resume()
puts "Control returned to main thread"
```

As a further clarification, consider the following two fibers:

```
fib = Fiber.new do
end
```

and

```
fib = Fiber.new do
  Fiber.yield()
end
```

Are the two fibers equivalent? They may seem similar in functionality as they don't do anything but their executions aren't. Realize that the first fiber will run to completion on the first invocation of `resume()` whereas the second fiber will run to completion by invoking `resume()` twice. Both are presented below:

```
fib = Fiber.new do
end

fib.resume()

# uncommenting the below line will result in an error
# fib.resume()
```

```
fib = Fiber.new do
  Fiber.yield
end

fib.resume()
fib.resume()

# uncommenting the below line will result in an error
# fib.resume()
```

Passing Values to Fiber

We can also pass arguments to a fiber, however, the way we pass

We can also pass arguments to a fiber, however, the way we pass arguments the first time a fiber is resumed versus every other time is different. Consider the snippet below

```
printer = Fiber.new do |msg|
  puts msg
end

printer.resume("hello world")
```

In the above snippet, the `msg` argument is sent as a block argument to the fiber's code block when we `resume()` the fiber for the first time. When the next time we `resume()` the fiber, the argument passed in the `resume()` method will be received by the fiber as the value of the `Fiber.yield` expression. Consider the following snippet:

```
printer = Fiber.new do |msg|
  puts msg
  newMsg = Fiber.yield
  puts newMsg
end

printer.resume("hello")
printer.resume("world")
```