... continued

This lesson continues the discussion on Monitor in Ruby.

Monitor Mixin

A mixin, when added to a class, augments it with additional capabilities without using inheritance. In Ruby, a mixin is a code wrapped up in a module that a class can include or extend. The standard library offers a mixin called `MonitorMixin`. We can either include it in our class or extend an object of our class with it to gain monitor functionality.

Let's consider an example class `Person` as follows:

```
class Person
  def initialize()
    @count = 0
    @name = "unnamed"
  end

  def updateName(name)
    @name = name
    @count += 1
  end

  def to_s
    return "I am #{@name} updated #{@count} times."
  end

end
```

The class is trivial and has only two instance fields. One is the name of the person and another is the count, which keeps track of how many times the name gets updated. The method `updateName()` isn't thread-safe. Multiple threads writing to it can step over each other's changes. One way we can fix this is to include the `MonitorMixin` in the class and then use synchronize or mon_enter() and mon_exit() methods to protect the critical sections. This implementation appears in the code widget below:

```
require 'monitor'

class Person
  include MonitorMixin

  def initialize(*args)
    @count = 0
    @name = "unnamed"
    # we must invoke the mixin's initialize method
    super(*args)
  end

  def updateName(name)
    self.mon_enter()
    @name = name
    @count += 1
    self.mon_exit()
  end

  def to_s
      return "I am #{@name} updated #{@count} times."
  end

end

# create a person
nikolaTesla = Person.new

# update the name 3000 times
# using 30 threads
threads = 30.times.map do
  Thread.new do
    100.times do
      nikolaTesla.updateName("Nikola")
    end
  end
end

threads.each(&:join)

# verify the name is shown updated
# 3000 times
puts nikolaTesla
```

Note that in the example above, we implement synchronization inside the instance method `updateName()`. An alternative could be to synchronize outside the class, as shown below in **lines#32-34**:

```
require 'monitor'

class Person
  include MonitorMixin

  def initialize(*args)
    @count = 0
    @name = "unnamed"
    # we must invoke the mixin's initialize method
    super(*args)
  end

  def updateName(name)
    @name = name
    @count += 1
  end

  def to_s
      return "I am #{@name} updated #{@count} times."
  end

end

# create a person
nikolaTesla = Person.new

# update the name 3000 times
# using 30 threads
threads = 30.times.map do
  Thread.new do
    100.times do
      # synchronize outside the class
      nikolaTesla.synchronize {
        nikolaTesla.updateName("Nikola")
      }
    end
  end
end

threads.each(&:join)

# verify the name is shown updated
# 3000 times
puts nikolaTesla
```

Note that each object becomes a monitor in itself. If we created two objects and two different threads invoked `updateName()` method on one object each, then the two threads can proceed as each object is associated with a distinct monitor. Don't confuse the monitor to be a class variable, i.e. access to `updateName()` for all objects of `Person` is not serialized, rather it is serialized for a given object.

Another approach is to `extend()` an object of class `Person`. This is shown in the widget below:

```
require 'monitor'

class Person

  def initialize(*args)
    @count = 0
    @name = "unnamed"
  end

  def updateName(name)
    self.mon_enter()
    @name = name
    @count += 1
    self.mon_exit()
  end

  def to_s
      return "I am #{@name} updated #{@count} times."
  end

end

# create a person
nikolaTesla = Person.new

# extend the person object with MonitorMixin
nikolaTesla.extend(MonitorMixin)

# update the name 3000 times
# using 30 threads
threads = 30.times.map do
  Thread.new do
    100.times do
      nikolaTesla.updateName("Nikola")
    end
  end
end

threads.each(&:join)

# verify the name is shown updated
# 3000 times
puts nikolaTesla
```

Note we extend the object `nikolaTesla` on **line#27** with the monitor mixin. Note that the method `updateName()` is able to use `mon_enter()` and `mon_exit()` methods because we invoke the `updateName()` after extending the object with the monitor mixin. If we attempt to update the name before extending, we'll get an error. You can try it out by commenting **line#27**.