Creating Threads

This lesson discusses the various ways in which we can create, run, prioritise and check the status of threads.

Creating Threads

We can create threads in Ruby using the following class methods of the `Thread` class:

- `new()`
- `start()`
- `fork()`

Note that all these methods create and start the thread at the same time.

An example of creating a thread and passing arguments to it using the `new()` method is presented below.

```
# Example to show how to span a thread using Thread.new

Thread.current.name = "mainThread"

# spawn a child thread
thread = Thread.new("Hello", "World") { |arg1, arg2|
  Thread.current.name = "childThread"
  puts("Name #{Thread.current.name} and id #{Thread.current.__id__} says #{arg1} #{arg2}")
}

# wait for the thread to finish
thread.join()

puts("Name #{Thread.current.name} and id #{Thread.current.__id__}")
```

On **line#6**, we use the `Thread.new` class method to instantiate and run our thread. The next two code widgets below replicate the same code but demonstrate creating threads using the other two class methods.

```
# Example to show how to span a thread using Thread.new

Thread.current.name = "mainThread"

# spawn a child thread
thread = Thread.start("Hello", "World") { |arg1, arg2|
  Thread.current.name = "childThread"
  puts("Name #{Thread.current.name} and id #{Thread.current.__id__} says #{arg1} #{arg2}")
}

# wait for the thread to finish
thread.join()

puts("Name #{Thread.current.name} and id #{Thread.current.__id__}")
```

Using `Thread.fork`:

```
# Example to show how to span a thread using Thread.new

Thread.current.name = "mainThread"

# spawn a child thread
thread = Thread.fork("Hello", "World") { |arg1, arg2|
  Thread.current.name = "childThread"
  puts("Name #{Thread.current.name} and id #{Thread.current.__id__} says #{arg1} #{arg2}")
}

# wait for the thread to finish
thread.join()

puts("Name #{Thread.current.name} and id #{Thread.current.__id__}")
```

Subclassing Thread

Subclassing Thread

Another way to create threads is to subclass the `Thread` class and use an instance of the derived class. An example is shown below:

```
class MyThread < Thread
  def initialize
    puts("Inside MyThread initializer")

    # Must call the Thread's class initialize
    super
  end
end


myThread = MyThread.new do
  puts("Running  an instance of a Thread subclass")
end


myThread.join()

puts("Main thread exiting")
```

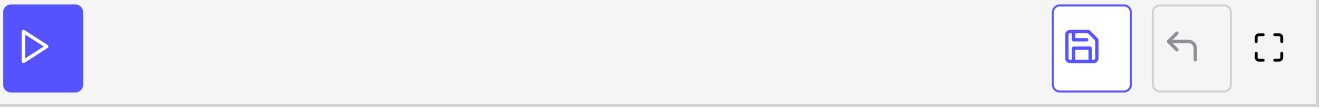
Note that on **line#6**, we are using the keyword `super`, which calls the method with the same name in the parent class, with the same arguments. If you comment out **line#6**, the program will fail.

The other two class methods `start()` and `fork()` behave slightly differently in that they ignore and don't execute the `initialize()` method of the derived class.

Consider the snippet below and note that the print message from the `initialize()` on **line#3** isn't printed.

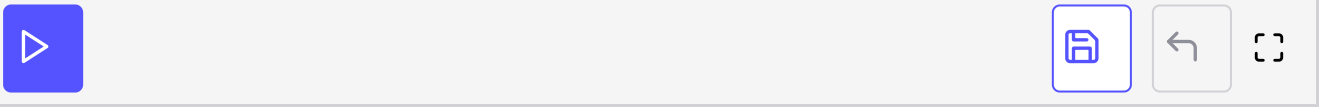
```
class MyThread < Thread
  def initialize
    puts("Inside MyThread initializer")
  end

  def sayHi
```

```
    puts("Hello World. Note initializer isn't executed.")
  end
end


myThread = MyThread.start() {
  myThread.sayHi()
}


myThread.join()

puts("Main thread exiting")
```



The above example is reprinted below with the `fork()` method. **line#4** isn't printed.

```
class MyThread < Thread
  def initialize
    # Doesn't get printed
    puts("Inside MyThread initializer")
  end

  def sayHi
    puts("Hello World. Note initializer isn't executed.")
  end
end


myThread = MyThread.fork do
  myThread.sayHi()
end


myThread.join()

puts("Main thread exiting")
```



Thread Status

Every thread maintains a `status` that depicts its current position. There are multiple states that a thread can be in at different times. The `status`

can be accessed as:

```
Thread.current.status
```

We mention below the possible values of `status` and what they depict.

- **run** - The thread is currently running.
- **sleep** - The thread is currently waiting or blocked.
- **false** - The thread finished execution successfully or was killed.
- **nil** - An unhandled exception was raised.
- **aborting** - The thread is dying.

Below is a working example for checking thread's `status`.

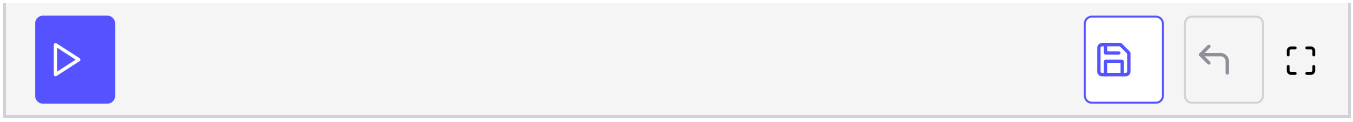
```
t1 = Thread.new { loop {} }
t2 = Thread.new { sleep 5 }
t3 = Thread.new { Thread.exit }
t4 = Thread.new { raise "exception" }

# Let main thread wait for other threads to startup
sleep 1

puts s1 = t1.status    #run
puts s2 = t2.status    #sleep
puts s3 = t3.status    #false
puts s4 = t4.status    #nil
```

Thread Priority

`priority` is the repetitiveness at which a thread is run by the scheduler. Higher `priority` threads are run more frequently than the lower ones. The `priority` can be set by an integer. An example below demonstrates how `priority` works in threads. Two counts are maintained in two different threads and their priorities are being set accordingly.

different threads and their priorities are being set accordingly.

```
count_t1 = 0
count_t2 = 0

t1 = Thread.new do
      loop { count_t1 += 1 }
    end
t1.priority = 2

t2 = Thread.new do
      loop { count_t2 += 1 }
    end
t2.priority = 1

sleep 1
puts count_t1
puts count_t2
```



We can see from the output that the thread with higher `priority` gives a bigger outcome as it is run more frequently than the other one.