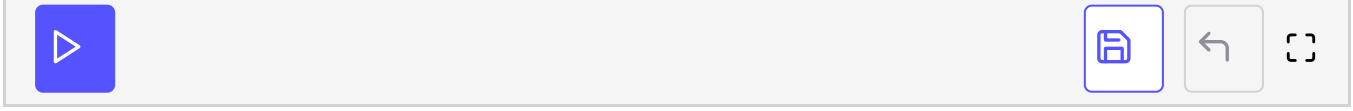
... continued

This lesson continues the discussion on instantiating threads.

> ### Ping Pong Program
>
> We'll write a program that will print ping and pong in strict alternation on the console. Our program will consist of two threads, one to write each string. The challenge is to coordinate the two threads to take turns one after another when printing to the console.
>
> We'll use a boolean `flag` variable to remember which thread should go next. While one thread prints on the console, we'll need a way to stop the other thread from moving forward. One and an inefficient way is to busy-wait in a while loop and constantly poll for a change in the while loop's condition. For now, we'll stick to this strategy and improve upon it in later lessons.
>
> The code appears in the code widget below:

```
flag = true
keepRunning = true

# prints ping on the console
pingThread = Thread.new do

  while keepRunning

    while flag == true
      # busy-wait
    end

    puts "ping"
    flag = true
  end
end


# prints pong on the console
pongThread = Thread.new do
```

```
  while keepRunning

    while flag == false
      # busy-wait
    end

    puts "pong"
    flag = false

  end
end



# run simulation for 10 seconds
sleep(10)
keepRunning = false
pingThread.join()
pongThread.join()
```



As an exercise, consider the highlighted lines 13 and 14 in the snippet above. If we flip their order, will the code continue to work correctly? The ping thread's code block would become:

```
pingThread = Thread.new do

  while keepRunning

    while flag == true
      # busy-wait
    end

    # flip the order of the two lines
    flag = true
    puts "ping"

  end
end
```

We encourage you to take a minute and reason out why or why not the suggested change break the program before reading on.

The subtlety to understand here is that a thread can get context switched at any time by the OS. In this case, the ping thread can get context

switched right after setting the `flag` variable to true and before printing

ping on the console. Or in other words, a context switch happens between the two lines. If the second thread gets scheduled next, it will find the flag set to true, break out of the while loop and print pong in succession, thus violating the constraint of our program.

One can appreciate the complexity of writing correct multithreaded programs with this example. A seemingly innocuous change can break the program!

Folks with Java or C# background would know that a similarly written ping-pong program in either of the languages will be broken on account of cache coherence and possible reordering of statements by the runtime or compiler. However, the standard MRI Ruby doesn't such optimizations and the program works correctly.