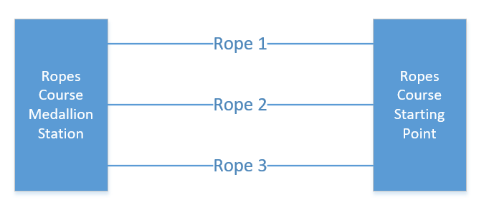
# COMP 10183 Assignment 1 Fall 2025 Prepared by: Karen Laurin & Dave Gilbert

# Overview

In this assignment you will be experimenting with creating and maintaining threads and using mutual exclusion to protect critical sections of the code.

# The Problem

You will be simulating an outdoor adventure race. The racers are tasked with crossing a gorge on a rope course. There are 16 racers, who must all cross the gorge. Once all the members have crossed the gorge their race completion time should be recorded. The goal for the group is to collectively minimize the total time it takes to get all of them across.

The gorge consists of three rope crossing points. There is no way to pass another racer on the rope and each rope can only support one racer for safety, so a racer may only enter a rope if it is empty.

If an adventure racer is waiting to use some rope, and another racer gets the rope ahead of them, that is ok. There is no need to try and implement fairness as in first-come-first served. The collective score is the total time for all racers to complete, given the limit that only one racer can use one rope at a time. The total time, or time for the last racer to complete the crossing safely, is all we are concerned about for this example.

Ideally you should keep all ropes busy at all times, however if most of your racers end up chosing to use rope 1 and rope 2, and rope 3 becomes idle for part of the race, this is acceptable. In the real world no racer would line up to cross a rope when there was an idle rope available, but for this first assignment keeping all ropes 100% busy for the entire race will be treated as optional, so long as most of the ropes are busy most of the time your performance will be ok.

# The Design

The design for this simulation must follow the basic UML class diagram given below. You may add any variables or methods you feel are necessary to solve the problem, although you should submit 3 classes structured as follows:

A diagram of a diagram

Description automatically generated

For all threads to see the three RopeCrossing objects, these three objects as static class fields in your main thread (the RaceSimulation class). Racers will need to communicate with the RopeCrossing objects, and vice versus, so declare objects as static in the RaceSimulation class and make them accessible from there through getters. UML diagrams specify class details, not method implementaiton details, RopeCrossing and Racer objects are not local method variables.

# Concurrency and Performance

In the main method, 16 racers will arrive at the beginning of execution. Each racer needs **time to set up their gear, which can randomly take between 800 and 1200 milliseconds**. Once a racer is ready, they can begin crossing.

Each racer must select a rope to cross. Your racers may wait until a rope is available before selecting and crossing that rope or you may put each racer in a queue, preselecting the rope. How you queue up your racers is up to you, the first method is better, but more difficult.

The racer will take **between 1000 and 2000 milliseconds to cross the gorge** on the rope, which will be determined randomly.

16 racers concurrently set up their equipment: Average time = 1 sec

16 racers use 3 ropes to cross the gorge. Average time = 16 \* 1.5 secs / 3 = 8 secs.

Total race time should be about (8s + 1s) ~= 9 secs. If you take 1-2 seconds longer than this, likely you have cut some corners in your design, and for the first assignment we will allow that. If your simulation runs in a lot less than 9 seconds, i.e. 6 or 4 or <0.5 seconds, you likely have a major problem. Likewise, if your simulation runs in 15 seconds or longer, it probably isn’t running the threads in parallel. Either you aren’t using threads, or you are locking too much.

# Sample Output

Your program should track and display the time for each racer to setup their gear, a message when they start and finish crossing, time to cross and total race time.

racer 09 ready to cross - 862 ms for gear settup

racer 15 ready to cross - 879 ms for gear settup

racer 08 ready to cross - 900 ms for gear settup

> racer 09 started crossing Forward Rope ONE

> racer 08 started crossing Forward Rope THREE

racer 10 ready to cross - 930 ms for gear settup

racer 01 ready to cross - 932 ms for gear settup

> racer 01 started crossing Forward Rope TWO

racer 12 ready to cross - 976 ms for gear settup

racer 03 ready to cross - 986 ms for gear settup

racer 04 ready to cross - 1005 ms for gear settup

racer 07 ready to cross - 1022 ms for gear settup

racer 11 ready to cross - 1032 ms for gear settup

racer 14 ready to cross - 1105 ms for gear settup

racer 00 ready to cross - 1145 ms for gear settup

racer 02 ready to cross - 1151 ms for gear settup

racer 06 ready to cross - 1179 ms for gear settup

racer 05 ready to cross - 1184 ms for gear settup

racer 13 ready to cross - 1191 ms for gear settup

# racer 08 finished crossing Forward Rope THREE in 1459 ms

> racer 02 started crossing Forward Rope THREE

# racer 01 finished crossing Forward Rope TWO in 1556 ms

> racer 04 started crossing Forward Rope TWO

# racer 09 finished crossing Forward Rope ONE in 1792 ms

> racer 00 started crossing Forward Rope ONE

# racer 02 finished crossing Forward Rope THREE in 1381 ms

> racer 05 started crossing Forward Rope THREE

# racer 04 finished crossing Forward Rope TWO in 1501 ms

> racer 07 started crossing Forward Rope TWO

# racer 00 finished crossing Forward Rope ONE in 1548 ms

> racer 03 started crossing Forward Rope ONE

# racer 05 finished crossing Forward Rope THREE in 1971 ms

> racer 11 started crossing Forward Rope THREE

# racer 07 finished crossing Forward Rope TWO in 1862 ms

> racer 10 started crossing Forward Rope TWO

# racer 03 finished crossing Forward Rope ONE in 1743 ms

> racer 06 started crossing Forward Rope ONE

# racer 11 finished crossing Forward Rope THREE in 1014 ms

> racer 14 started crossing Forward Rope THREE

# racer 10 finished crossing Forward Rope TWO in 1468 ms

> racer 13 started crossing Forward Rope TWO

# racer 06 finished crossing Forward Rope ONE in 1888 ms

> racer 12 started crossing Forward Rope ONE

# racer 14 finished crossing Forward Rope THREE in 1413 ms

# racer 13 finished crossing Forward Rope TWO in 1313 ms

# racer 12 finished crossing Forward Rope ONE in 1064 ms

> racer 15 started crossing Forward Rope ONE

# racer 15 finished crossing Forward Rope ONE in 1545 ms

Total Execution Time = 10.572 sec

# Help With Implementation

You only need threads, synchronized methods (or statements), and atomic variables to implement a solution.

You should have 1 thread for each racer, and also 1 thread for each rope. The rope thread coordinates when racers are allowed to proceed across the rope. The racer thread decides when it is done setting up and does the work of crossing the rope. Communication between threads can be achieved with global variables stored in the RaceSimulation class.

If you have multiple threads accessing the same variable and writing to that variable you must guard it either with a synchronized method or statement, or the variable must be atomic.

In our code the *"work"* of crossing the rope will be done via a call to Thread.sleep() as will setup time. Use the following methods:

*/\*\*  
 \* Wait an appropriate amount of time for the racer to cross, avg time = 1.5 sec  
 \* @param ropeName name of the rope crossing  
 \* @param racerName name of the racer  
 \* @param direction direction, always == 1 for A1, 1 or 2 for A2  
 \*/*public void cross(String ropeName, String racerName, int direction) {  
  
 Random random = new Random();  
 int low = 1000;  
 int high = 2000;  
  
 int waitTime = random.nextInt(high-low) + low;  
  
 String dir = direction == 1 ? "Forward " : "Return ";  
 System.*out*.println("> " + name + " started crossing " + dir + ropeName);  
  
 try {  
 Thread.*sleep*(waitTime);  
 } catch(Exception e) {}  
  
 System.*out*.println("# " + racerName + " finished crossing " + dir + ropeName +" in "  
 + waitTime + " ms");  
}  
  
*/\*\*  
 \* Wait an appropriate amount of time for the racer to setup, average time = 1 sec  
 \* @param rName - racer name  
 \*/*public void setup(String rName) {  
  
 Random random = new Random();  
 int low = 800;  
 int high = 1200;  
int waitTime = random.nextInt(high - low) + low;  
  
 try {  
 Thread.*sleep*(waitTime);  
 } catch (Exception e) {}  
 System.*out*.println(rName + " ready to cross - " + waitTime + " ms for gear settup");  
}

There are several tricky problems to solve here. Once the racer thread has finished set up, it must wake, and then pick a rope to cross on. You might pre-assign ropes at the beginning of the program. Put 5 racers on rope 1, 5 on rope 2, and 6 racers on rope 3 at the start. This solution will typically waste resources, and often run longer than 9 seconds. A better solution only assigns racers to ropes once the rope is available.

The other hard problem is that once a racer has finished setting up, it must wait until its rope is available before it starts crossing. For this assignment you will use busy waiting. In the middle of your racer thread's run() method, before it crosses a rope, you will need a loop like this:

while(!startCrossing) {  
 try {  
 Thread.*sleep*(1);  
 } catch (Exception e) {}  
}

In this example **"startCrossing" is a boolean variable** that is some how set by the rope. The racer thread is waiting for a *"signal",* that it is time to cross. You **\*must\*** sleep a little bit while you wait, otherwise you will burn all CPU resources and the program probably won't work. In computing *busy waiting is considered to be an anti-pattern - i.e. a very bad idea*. At this time in the course it is all we know how to do. See: <https://en.wikipedia.org/wiki/Busy_waiting>

In this course, one of our main goals is to teach you proper alternatives to this weak solution, although you will likely need it for this first assignment. In future assignments you will not be allowed to call Thread.sleep().

# Submission

You will submit your java code in a zip folder to the submission folder on MyCanvas. Please **don't zip up the entire project**, you just need to include all of the .java files in the zip folder.

# Future Work

Assignment 2 in this course is an extension of Assignment 1. In Assignment 2, racers are grouped into small teams, they must cross the gorge 2 times, and you are not allowed to call sleep or do \*any\* busy waiting at all. Using Thread.sleep() to manage parallel programs is an extremely bad idea, and we will focus on alternatives to busy waiting throughout the course.

# Grading

You will be evaluated on the following:

* Progress report – 20%
  + Meet with your professor in the lecture before the assignment is due
  + You will have 1 minute to present your code and ask questions
  + You must be present to get credit.
    - If you haven’t started the work, you get 1 / 4 for showing up
    - Full score: about 1/2 of the code finished and you ask good questions
* Code Structure, Formatting, and Documentation – 40%
  + Code is readable
  + Statement of authorship is included at the top of all files
  + Javadoc and commenting are complete
    - All class attributes must be documented
    - All methods including main must be documented
  + Structure grade = 0 if lambda functions or anonymous classes are used.
  + May use synchronized methods/statements, atomic variables, and busy waiting.
  + For this assignment code may not use ANY other synchronization techniques.
    - Use of any other techniques will result in a zero for code structure.
  + Each class should be in a separate file.
    - Just include the .java files in your .zip folder, no other files please.
  + Code structure follows UML diagram, 3 classes, ropes and racers are class attributes of RaceSimulation, they may be static. They may not be local variables.
* Code Execution and Performance – 40%
  + Submission correctly solves the above problem
  + Should complete in about 9 seconds
  + Output should be similar to sample