# Additional exercises

## Concurrent Programming: Basic Thread management

### Basic thread handling

#### Project MessagePrinter

Define a class named *MessagePrinterRunnable* that implements the Runnable interface. The class should have a String attribute to store a message. This message should be set through the constructor of the class. Implement the run method to print the message attribute to the console. In the main method of your application, create and start two threads:

- The first thread should be created by instantiating the Thread class directly, passing an instance of MessagePrinterRunnable with a custom message to its constructor.
- The second thread should be created using a lambda expression to define the Runnable object, and it should also print a custom message.

Run your application and observe the output. Both messages should be printed to the console, but the order in which they appear may vary each time you run the program.

### Basic thread information

#### Thread Task Manager

You have been tasked with implementing a part of a monitoring system for an application server that manages multiple threads. Your job is to write a Java program that performs the following actions:

1. **Named Thread Creation**: Create a class called Task that implements the `Runnable` interface. Within the `run()` method, assign a unique name to the current thread using `Thread.currentThread().setName()`, and then print out that name. Print also the unique identifier of the current thread using getId() and the beginning of the method's execution. Add a method `processTask` to simulate some kind of task using a Thread.slepp. Write a message when the task is finished.

2. **Thread State Monitoring**: Start three instances of this class on separate threads in the Main class. After starting each thread, your program must print the state of the thread using `getState()`. Ensure that you print the thread's state at two different moments:

   - Immediately after calling `start()`.
   - After a brief pause (for example, using `Thread.sleep(100)`), to observe a possible change in state.

3. **Thread Termination Verification**: Finally, after starting all threads, your program should periodically check whether the threads have completed their execution creating a **new Thread** to do this verification that will have a loop waiting to all the threads to finish. Use both the `TERMINATED` state and the `isAlive()` method to print messages indicating whether the threads have finished.

## Finishing and interrupting threads

1. **Thread Monitoring Exercise**: Create a project called `ThreadMonitoringSystem` and inside implement a class named `ThreadMonitor` with the main method in it that is responsible for monitoring the state of a thread. Your program will create a thread that performs a simple repetitive task ( `SimpleTask` class), such as printing a line of text. Use a boolean attribute to control the execution of this thread. The `ThreadMonitor` class should periodically check if the thread has reached a certain number of iterations ( `getIterationCount method` in `SimpleTask` class), and if so, request that the thread finish by setting the boolean variable to `true` . Show in `SimpleTask` and in `main` messages when starting and finishing the thread.

2. **Interruption Handling Exercise**: Create a project named `InterruptionHandler` with a class called `MainHandler` with the `main` method in it. Create a thread that performs a blocking operation, such as waiting with `Thread.sleep` in an infinite loop ( `Thread.sleep(Long.MAX_VALUE)` ). The main application should be able to interrupt this thread multiple times (for instance, every 2 seconds) and the thread should be capable of handling these interruptions by printing a message with the time of the interruption, and then continuing its execution. After a specific number of interruptions, the thread should be able to cleanly finish its execution using a boolean to stop correctly.

## Groups and daemons

### Exercise 1: Thread Group for File Processing

#### Project WordCountingGroup

1. Define a class called `FileProcessor` that implements `Runnable` . This class should read a file, count the number of words in it, and print the count. It will have an attribute *File* with the file that will be initialized in the constructor.
2. Instantiate a `ThreadGroup` called "FileProcessorGroup".
3. In the main method, for each file in the directory provided by the user, instantiate a thread from your class, adding it to the "FileProcessorGroup", and start the thread.
4. After starting all threads, monitor the active count of the group. Every second print a message with the files pending to process and when all threads have completed, print a message indicating that all files have been processed.

### Exercise 2: Daemon Thread for Directory Monitoring

#### Project MonitoringFiles

1. Create a Java class called `DirectoryMonitor` that implements `Runnable` interface. It will have an attribute *File* for the directory to check and another attribute with a *Set of Strings* with the names of the files in the directory.
2. In the contructor of `DirectoryMonitor` class you will receive a String with the directory to check and you will initialized the attributes based on the information received: the directory and the Set with the names of the files of this directory.
3. In the run method you will check if the directory currently have the same files that the ones stored in the Set. If a new file is detected you will print a message similar to "added File:nameFile.extension to directory nameDirectory", and add this file to the Set.
4. This daemon thread should check a specified directory every 10 seconds to see if any new files have been added.
5. In the main method, create a Thread based on `DirectoryMonitor` class and stablish it as deamon.
6. Add new files to the directory to test the daemon created.