Multithreading

Overview

In this lab, you will refactor an existing application so that it performs time-consuming tasks on background threads.

Source folders

Student project:	`StudentMultithreading`
Solution project:	`SolutionMultithreading`

Roadmap

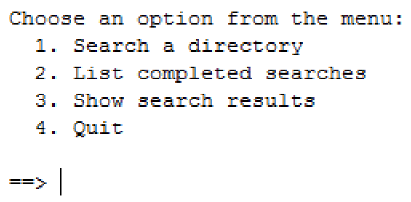
There are 3 exercises in this lab, of which the last exercise is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Introduce multithreading into the application
2. Ensure thread safety
3. Wait for threads to terminate gracefully

Familiarization with the application

Open the student project, and expand the `student.multithreading` package. The package contains a single class named `Main`, which runs entirely in a single thread.

Run the application. You are presented with a menu of options, as follows:



```
Choose an option from the menu:
  1. Search a directory
  2. List completed searches
  3. Show search results
  4. Quit

==> |
```

At the prompt, type `1` and press ENTER. You will be asked to enter a directory name - enter a directory name that doesn't contain too many files! The application then performs a recursive search for all the files and sub-directories in the specified directory. It performs this search on the main application thread, so the application is blocked until the search is complete.

When the search finally completes, the application displays the main menu again. If you choose option 2, you should see your directory name listed (because the application has completed the search for that directory). To see the results of that search, choose option 3; the application will display all the files and sub-directories that it found in your directory (recursively).

Try another search (i.e. choose option 1 again). This time enter the name of a directory that contains a large number of files and/or sub-directories. This search might take a long time to complete, and the application is blocked until the search has finished.

When you're comfortable with how the application works, take a look at the code in the `Main` class. There are lots of comments in the code, so you should be able to follow the logic fairly easily. If there's anything you don't understand, feel free to ask the instructor.

Exercise 1: Introduce multithreading into the application

The application is unacceptable because it forces the user to wait until a search finishes before the user can do anything else. A better approach would be to use multithreading

To run code in a separate thread, you must define a class that implements the `Runnable` interface. So, add a new class named `DirectorySearcher` and implement it as follows:

- The class must implement the `Runnable` interface (obviously).
- The class needs some instance variables so it can "remember" what it's meant to be doing. We suggest the following instance variables:
    - A `String` instance variable called `directoryName`, which remembers the name of the directory to search.
    - A `List<File>` instance variable called `thisResult`, to accumulate all the files and sub-directories for this search. Create an empty list initially.
    - A `Map<String, List<File>>` instance variable called `allResults`, which will hold the results of all searches completed so far (this will be passed in from the main code – see the constructor info below…)
- Write a constructor. The constructor requires two parameters, which will be passed in from the main application code:
    - The name of the directory to search.
    - The map into which the thread can store its results on completion of this search.
- Copy the `searchDirectory()` method from the `Main` class into the `DirectorySearcher` class (because the search operation will now be performed in your background thread class). Refactor the method as follows:
    - It doesn't need to be `static` any more (it was only declared `static` originally because all the methods in the `Main` class were `static` for simplicity).
    - It doesn't need a `List<String>` parameter (the `DirectorySearcher` class can use the `thisResult` instance variable instead).
- Write a `run()` method to perform the work for this thread. Implement it as follows:
    - Invoke `searchDirectory()`, passing in a `File` object to represent the directory to search.
    - After `searchDirectory()` has completed, insert the result into the map of all results (similar to the existing code in the `Main` class's `doSearch()` method).
- In the `Main` class, refactor `doSearch()` to perform the search in a background thread (i.e. create a `DirectorySearcher` object and start it in a separate thread).

Run the application again. Now, when you do a search, the search should take place in a background thread. This means you can kick off multiple searches simultaneously ☺.

Exercise 2: Ensure thread safety

The application isn't thread-safe at the moment, because multiple threads might insert results into the `allResults` map simultaneously. As currently implemented, the application uses a `HashMap`, which is a non-thread-safe collection class.

It's extremely important to ensure thread safety in your applications. There are various ways to achieve thread safety in this situation… the simplest approach is to use a thread-safe map class, e.g. `ConcurrentHashMap`. Refactor the `Main` class to do this.

You probably won't observe any differences when you run the application, but at least you won't get any nasty thread concurrency surprises later on either!

Exercise 3 (If time permits): Wait for threads to terminate gracefully

Improve the application so that it allows all running threads to complete first when the application shuts down. Hints:

- Keep a list of all threads you create in the application.
- When the user quits the application, loop through the thread list and invoke `join()` to wait for the thread to finish. Optionally, specify a timeout on each `join()` operation (to avoid potentially having to wait a very long time for a thread to finish!)