Practical Sheet : Multi-threading

Application on a single thread

Throughout the following examples we shall be making use of two classes available with this practical. The classes are:

**BounceFrame**: A frame which holds two buttons. When the start button is clicked a ball object is created and its bounce method is invoked. This class holds also a main() method which launches the application.

**Ball**: An object which defines a method bounce() which is invoked by frame. This method draws the ball (through draw()) and moves the ball by invoking move(). In the move method the ball move for 5ms and then goes to sleep, waiting for the scheduler to wake it up.

1) Compile the two classes and write a Launcher which creates a BounceFrame and invoke its show() method.

Note: *The whole application runs on a single thread - hence when we try to stop the ball by clicking the close button this is not possible until the ball stops moving - the ball's movement is occupying all the thread's time.*

Multi-threaded application

In this example we shall try to make the ball execute on a separate thread. This should enable the gui to interact with the user **while** the ball is moving.

1) Subclass Ball from java.lang.Thread and convert its bounce method to run(). Make sure that you encapsulate the method in a try-catch block which captures InterruptedException.

2) In BounceFrame, in the actionListener associated with the Start button, create the Ball thread and activate. Check with documentation on how to activate a subclass of a thread.

Note: *When there is one ball the whole application runs on two threads - this enables the gui to interact with the user WHILE the ball is running. Now click again and again on start - new Balls (and therefore new Threads) are created.*

Thread priorities

In Bounce.java add another button labeled **turbo** (cut and paste code used to create start button). When creating balls using the start button set their priority to normal. In the case of the express button set the balls' priority to Thread.NORM_PRIORITY + 2 and give them
a different color.

Now try to execute the example.

Note: The balls running on prioritized threads execute faster - this is because the threads with low-level priority get the chance to execute only when all of the balls running on prioritized threads are asleep.

**Thread synchronization**

1) Write a Counter object which holds a private int counter. Write an increment() method which is synchronized and which increments the counter attribute. Also in the increment() method, check if it the counter has reached value 5 - if not then put the thread executing the method to sleep and output Putting thread to sleep. If yes notify all sleeping threads and output Waking up all sleeping threads.

1) Write an Incrementer class which extends Thread. In its constructor it takes a reference to a Counter object and a thread id (a simple int) and stores them in its attributes. In its run() method it tries to increment the method of the counter (given to it in its constructor) and after invoking the method it prints out Thread with thread id ...has just incremented the counter.

3) In the Launcher class create a Counter object and 5 Incrementer objects (giving to each a reference to Counter as well as a unique id (say 1,2,3,4,5)). Next, launch each of the Incrementers, one every second.

Try to predict what the output will looks like before executing program.

**Analyzing database connections through a GUI using threads (for advanced students)**

In the following exercise we shall be implementing a simple GUI to launch threads which execute queries on the ConnectionPool, concurrently. The GUI consists of a "Launch new query" button and two text fields The second text field (set to non-editable mode) displays the number of connections which are available from the connection pool.

**Steps for implementation:**

i) Create a class QueryThread which extends Thread. In its constructor it takes a delay
value. Given that the query executes in a short time this delay should let us simulate the execution of time consuming operations which occupy a connection for a period of time. This delay is useful only for display purposes. In the run() method the QueryThread obtains a connection from the connection pool, executes a simple query and executes a sleep (with the provided delay).

ii) Define a ThreadSpawner class which implements the ActionListener interface. In its actionPerformed() method it creates a QueryThread class and launches the thread.

iii) Modify DBConnectionPool such that a PropertyChangedEvent is thrown whenever the number of available connections changes (keep track of the number of available connections in the getConnection and releaseConnection methods). Add also appropriate registration methods (see bound properties section on notes, pg4-33) such that PropertyChangedListeners can be registered. Define also a mechanism to fire the event.

i) Define a class MyTextField which implements a PropertyChangeListener. In this method the textfield takes the new value from the event and sets its text to that value.

ii) Create a GUIFrame class which extends JFrame and which creates the GUI components described above. Make an instance of ThreadSpawner and register it as a listener to the button. Make an instance of MyTextField and register it with DBConnectionPool.

iii) Create a launcher which creates a GUIFrame. Whenever the button is clicked, an ActionEvent is generated. This event is captured by ThreadSpawner which creates a Thread which executes a query. At this point a connection is obtained from the connection pool and a propertychangedEvent is released. This is captured by the textfield which updates its view.