5COSC001W - OBJECT ORIENTED PROGRAMMING
Lecture 10: Java Concurrency — Part 2

Dr Dimitris C. Dracopoulos

Interrupting Threads

- An interrupt is an indication to a thread that it should stop.
- It is up to the program to decide whether to stop the thread or do something else.
- A thread sends an interrupt by invoking method `interrupt()` on another `Thread` object for the thread to be interrupted.
- Programs can call method `Thread.interrupted()` periodically (e.g. part of a loop) to check if they received an interrupt. Calling this twice, returns `false` the second time, i.e. the first call clears the *interrupted status* flag. The static method *isInterrupted()* does not clear the status.

```
if (interrupted()) {
    // do something - e.g. terminate thread?
}
```

Example:

```java
class T1 extends Thread {
    public void run() {
        try {
            while (true) {
                System.out.println("Thread is working");

                Thread.sleep(100000);
            }
        }
        catch (InterruptedException ex) {
            System.out.println("Caught interrupt! Exiting...");
        }
    }
}
```

```java
class InterruptedThreadExample  {
    public static void main(String[] args) {
        T1 t = new T1();
        t.start();

        t.interrupt();

        // sleep 1 sec (main thread)
        try {
            Thread.sleep(10000);
        }
        catch (InterruptedException e) {
            System.out.println("main method interrupted!");
        }

        System.out.println("after sleep");
    }
}
```

High Level Concurrency Classes

Starting from Java 5.0 new higher level concurrency classes were introduced based on third party classes previously developed by Doug Lea.

Package:

`java.util.concurrent`

- Standardised Extensible Frameworks
- Classes providing useful functionality which are tedious or difficult to implement

Thread Pools

- Creating a very large number of threads could lead to inefficiency as there is a cost for creating every single thread.
- This cost can be reduced by using a *thread pool*.
- A thread pool creates a fixed number of threads and keeps them alive.
- When a `Runnable` object is added to the pool, the next idle thread executes it.

```
Runnable r1 = new GreetingRunnable("Hello");
Runnable r2 = new GreetingRunnable("Goodbye");
ExecutorService pool = Executors.newFixedThreadPool(MAX_THREADS);
pool.execute(r1);
pool.execute(r2);
```

- If many runnables are submitted but there are not enough in the pool, then the runnables are put in a queue until a thread is available.

Synchronising Code using Higher Level Objects

Similarly to the `synchronized` keyword in low level thread programming, `java.util.concurrent` provides a number of high level `Lock` classes for synchronisation.

A lock object is added to a class whose methods access shared resources, as follows:

```
lockObject.lock();
try {
    Manipulate the shared resource.
}
finally {
    lockObject.unlock();
}
```

```java
public class BankAccount {
    private Lock balanceChangeLock;
    . . .
    public BankAccount() {
        balanceChangeLock = new ReentrantLock();
        . . .
    }
}
public void deposit(double amount) {
   balanceChangeLock.lock();
   try {
      double newBalance = balance + amount;
      System.out.println(", new balance is " + newBalance);
      balance = newBalance;
   }
   finally {
      balanceChangeLock.unlock();
   }
}
```

- A thread which holds a `Reentrant` lock can call the `lock` method on a lock object that already owns. The thread gives up ownership of the lock if the `unlock` method is called as many times as the `lock` method.

How to Avoid Deadlocks

A deadlock occurs when a thread has obtained the lock `A` and it is waiting for a lock `B` to be released. At the same time, a different thread has obtained the lock for `B` and it is waiting for lock `A` to be released before resuming (circular dependency, i.e. a catch-22 situation).

- Higher level thread programming uses a **condition object**.
- Condition objects allow a thread to temporarily release a lock, so that another thread can proceed.
- Each condition object belongs to a specific lock object which can be obtained with the `newCondition` method of the `Lock` interface.

Calling `await()` and `signalAll()` on Condition Objects

Similarly with `wait` and `notifyAll` (low level synchronisation) we have `await()` and `signalAll` on high level lock conditions.

- When a thread calls `await()` on a *condition* object it is deactivated goes in a waiting list associated with that condition object and releases the lock.
- The `signalAll()` wakes up all threads waiting on that condition and they compete to get the lock. One of them will get it and resume its execution from where it was left when it called `await()`.

```java
public class BankAccount {
    private Lock balanceChangeLock;
    private Condition sufficientFundsCondition;
    . . .
    public BankAccount() {
        balanceChangeLock = new ReentrantLock();
        sufficientFundsCondition = balanceChangeLock.newCondition();
        . . .
    }

    public void withdraw(double amount) {
        balanceChangeLock.lock();
        try {
            while (balance < amount) {
                sufficientFundsCondition.await();
            }
            . . .
        }
        finally {
            balanceChangeLock.unlock();
        }
    }
}
```

```java
public void deposit(double amount) {
    balanceChangeLock.lock();
    try {
        . . .
        sufficientFundsCondition.signalAll();
    }
    finally {
        balanceChangeLock.unlock();
    }
}
```

The result is that multiple threads can run without a deadlock and without ever reaching a negative balance.

High Level Locks vs Low Level Locks

- Low level locks are simply built-in locks associated with each Java object.
- Acquiring the low level lock involves using `synchronized` methods or `synchronized` blocks of code.
- Every Java object has one low level lock and a *single* condition associated with it.
- Using high level lock classes, you can create multiple locks per object and multiple *conditions* on a single lock.

Concurrency in Swing

A well-written Swing program uses concurrency to create a user interface that never "freezes", i.e. it is always responsive to user interaction.

Three types of threads used in Swing programs:

- *Initial thread*: the thread that executes the initial thread code.
- The *event dispatch thread*: where all event-handling code is executed.
  Most code that interacts with the Swing framework must also execute on this thread. This is because most Swing object methods are not "thread safe" and *thread interference* or *memory consistency* problems can occur.
- *Worker threads*: these execute time-consuming background tasks.

The SwingWorker Class

```
abstract class SwingWorker<T,V>
```

3 threads involved in the life cycle of a `SwingWorker`

- *Current thread*: The `execute()` method of a `SwingWorker` object is called on this thread.
  - It schedules SwingWorker for the execution on a worker thread and returns immediately.
  - A thread can wait for a `SwingWorker` to complete by calling the `get` method which gives back the result `T` returned by `doInBackground()`.
- *Worker thread*: The `doInBackground()` method is called on this thread. This is where all background activities should happen.
- *Event Dispatch Thread*: All Swing related activities occur on this thread. SwingWorker invokes the `done()` method on this thread, after `doInBackground()` completes. The `process()` is also called and any `PropertyChangeListeners` are notified.

The `SwingWorker` Class (cont'ed)

- `T`: is the result type returned by this SwingWorker's `doInBackground()` and `get()` methods.
- `V`: is the type used for intermediate results used by the `publish` and `process` methods.

Most useful Methods of `SwingWorker`

- `doInBackground()`: executed in the worker thread, gives back a `T` object as a result.
- `execute()`: starts the worker thread executing the `doInBackground()`.
- `done()`: executed in the event dispatch thread after the `doInBackground()` finishes.
- `get()`: executed by other threads, it waits if necessary for the task (`doInBackground()` to complete and received the `T` object as a result.
- `publish(V... chunks)`: sends data chunks to the `process(java.util.List<V>)` method. This is called from the `doInBackground` method to deliver intermediate results for processing on the Event Dispatch Thread via the `process()` method.
- `process(List<V> chunks)`: Receives data chunks from the `publish` method asynchronously on the Event Dispatch Thread.

Example 1 for SwingWorker

```java
final JLabel label;
class MeaningOfLifeFinder extends SwingWorker<String, Object> {
   public String doInBackground() {
      return findTheMeaningOfLife();  // time consuming task
   }

   protected void done() {
      try {
         label.setText(get());
      } catch (Exception ignore) {}
   }
}

// to start the thread:
(new MeaningOfLifeFinder()).execute();
```

Example 2 - SwingWorker

This program tests the fairness of `java.util.Random` by generating a series of random boolean values in a background task. This is equivalent to flipping a coin; To report its results, the background task uses an object of type `FlipPair`.

```
private static class FlipPair {
    private final long heads, total;
    FlipPair(long heads, long total) {
        this.heads = heads;
        this.total = total;
    }
}
```

`total` is the total number of random numbers. `heads` the number of heads.

```java
private class FlipTask extends SwingWorker<Void, FlipPair> {
    protected Void doInBackground() {
    long heads = 0;
    long total = 0;
    Random random = new Random();
    while (!isCancelled()) {
        total++;
        if (random.nextBoolean()) {
            heads++;
        }
        publish(new FlipPair(heads, total));
    }
    return null;
}
```

```java
protected void process(List<FlipPair> pairs) {
    FlipPair pair = pairs.get(pairs.size() - 1);
    headsText.setText(String.format("%d", pair.heads));
    totalText.setText(String.format("%d", pair.total));
    devText.setText(String.format("%.10g",
            ((double) pair.heads)/((double) pair.total) - 0.5));
}
```