MULTITHREADING - III

- Communication & co-operation among threads
- Graphical User Interfaces and multithreading
Communication & Co-ordination:

- Threads often have to coordinate their actions.
  - These co-ordinations are guided by some conventions.
  - Without such ‘conventional’ guidelines, we may end up in the corridor situation between two people, i.e., “livelock”.
- The most common is called the **guarded block**.
  - A guarded block keeps checking for a condition to be true.
    - ... and only then the thread execution resumes.
  - It’s simple enough as a concept, but we must carefully follow a sequence of steps to implement it correctly.
Guarded Blocks

- Let’s say we have a method `guardedJoy()` that must not proceed unless a shared variable `joy` is set to true by another thread.

- Well … it’s a simple concept. We don’t even need threads!

```java
public void guardedJoy() {
    while(!joy) { /* do nothing */ }
    System.out.println("Joy has been achieved!");
}
```
Guarded Blocks

■ Let’s say we have a method \texttt{guardedJoy()} that must not proceed unless a shared variable \texttt{joy} is set to true by another thread.

■ Well … it’s a simple concept. We don’t even need threads!

\begin{verbatim}
public void guardedJoy() {
    while(!joy) { /* do nothing */ }
    System.out.println("Joy has been achieved!");
}
\end{verbatim}

• Of course we can ‘guard’ against a condition with a loop.
• But this wastes processor time, which is exactly what we want to avoid in multithreaded applications!
  • Don't do this!
Guarded Blocks: waiting

- A more efficient guard invokes `Object.wait()` to suspend the current thread.
  - The call to `wait()` does not return until another thread issues a notification that something special may have happened.
  - This “something special” may not be the event our thread is waiting for.

```java
public synchronized void guardedJoy() {
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {
            /* do nothing */
        }
    }
    System.out.println("Joy and efficiency " + "have been achieved!");
}
```

- This guard only loops once for *each special event*
- Which may not be the event we're waiting for.
Guarded Blocks: waiting

- A more efficient guard invokes `Object.wait()` to suspend the current thread.
  - The call to `wait()` does not return until another thread issues a notification that something special may have happened.
  - This “something special” may not be the event our thread is waiting for.

```java
public synchronized void guardedJoy() {
    while (!joy) {
        try {
            wait();
        } catch (InterruptedException e) {
            /* do nothing */
        }
    }
    System.out.println("Joy and efficiency " + "have been achieved!");
}
```

- **Always** invoke `wait` inside a loop that tests for the condition being waited for.
- **Never** assume that the interrupt was for the particular condition you were waiting for.
Guarded Blocks: waiting

■ Just like sleep, an InterruptedException can be thrown by the wait method.

■ So …
  
  - Why is this version of the method synchronized???

```java
public synchronized void guardedJoy() {
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) { /* do nothing */
        }
    }
    System.out.println("Joy and efficiency " + "have been achieved!");
}
```
Guarded Blocks: waiting

- Suppose we are using an object d to call wait().
  - When a thread invokes d.wait(), it must own d’s intrinsic lock.
    - Otherwise an error (IllegalMonitorStateException) gets thrown.
  - That is why it is invoked inside synchronized block.

```
public synchronized void guardedJoy() {
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {
            /* do nothing */
        }
    }
    System.out.println("Joy and efficiency " + "have been achieved!");
}
```

- Thread releases the lock and suspends execution.
- At some point in the future, another thread will acquire this lock and invoke 
  Object.notifyAll()
Guarded Blocks: notification

- The `Object.notifyAll()` method wakes up (i.e., “notifies”) all threads waiting on this object’s intrinsic lock.

```java
public synchronized notifyJoy() {
    joy = true;
    notifyAll();
}
```

- There is another notification method `notify()`
  - wakes up a single thread
    - out of possibly multiple threads waiting on this object’s lock
  - the choice of which thread is chosen is arbitrary
  - this method is useful only in massively parallel applications where a very large number of threads are doing similar tasks.
    - In such an application, you don't care which thread gets woken up.
A *producer-consumer* example

- Data is a series of text messages
  - the producer creates the data
  - and the consumer does something with this data

- The communication uses a shared object
  - the consumer must try to retrieve before producer has created
    - ... so, communication is crucial!

- We are going to define an object of type ‘Drop’
  - this is the shared object used by the producer and consumer threads
The ‘Drop’ shared object

```java
public class Drop {
    private String message; // Message sent from producer to consumer.
    private boolean empty = true; // True if consumer should wait for producer to send message
        // False if producer should wait for consumer to retrieve message.

    public synchronized String take() {
        // Wait until message is available.
        while (empty)
            try { wait(); } catch (InterruptedException ignore) { /* do nothing */ }
        empty = true; // toggle status
        notifyAll(); // notify producer that status has changed
        return message;
    }

    public synchronized void put(String message) {
        // Wait until message has been retrieved.
        while (!empty)
            try { wait(); } catch (InterruptedException ignore) { /* do nothing */ }
        empty = false; // toggle status
        this.message = message; // store message
        notifyAll(); // notify consumer that status has changed.
    }
}
```
The producer thread

```java
public class Producer implements Runnable {

    protected static final String DONE = "done";
    private Drop drop;

    public Producer(Drop drop) { this.drop = drop; }

    public void run() {
        Random random = new Random();
        List<String> messages = Arrays.asList("Mares eat oats",
                                            "Does eat oats",
                                            "Little lambs eat ivy",
                                            "A kid will eat ivy too");

        messages.forEach((String m) -> {
            drop.put(m);
            try { Thread.sleep(random.nextInt(5000)); }
            catch (InterruptedException ignore) { /* do nothing */ }
        });
        drop.put(DONE);
    }
}
```
The consumer thread

```java
public class Consumer implements Runnable {

    private Drop drop;

    public Consumer(Drop drop) { this.drop = drop; }

    public void run() {
        Random random = new Random();
        for (String m = drop.take(); !m.equals(Producer.DONE); m = drop.take()) {
            System.out.format("MESSAGE RECEIVED: %s\n", m);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException ignore) { /* do nothing */ }
        }
    }
}
```
The main thread

```java
public class ProducerConsumerExample {

    public static void main(String[] args) {
        Drop drop = new Drop();
        (new Thread(new Producer(drop))).start();
        (new Thread(new Consumer(drop))).start();
    }
}
```

- Lock objects (like our `ReentrantLock`) work very much like the intrinsic locks used in synchronization
- Lock objects also support guarded blocks through wait and notify mechanisms
  - *but the implementation is different*
- Lock objects implement wait and notify through `Condition` object.
- The methods are:
  - `await()`, to wait until a condition is ‘signaled’
  - `signal()`, to wake up one waiting thread, and
  - `signalAll()`, to wake up all waiting threads.

Additional Note
Thread joins: another kind of waiting

- The `join()` method allows one thread to wait for the completion of another.
  - If `t` is a Thread object whose thread is currently executing,
  - `t.join();`

- Causes the current thread to pause execution
  - until `t`’s thread terminates

- Overloading `join()` allows us to specify a waiting period

- Just like `sleep()`
  - `join()` is OS-dependent for its timing
    - so you shouldn’t assume that it will wait for exactly as long as you specify
  - `join()` responds to interrupts by exiting with `InterruptedException`
Let’s look at a painting example (in light of homework 2):
- even though the homework itself doesn’t involve multithreading

**Problem**
- What if the state (i.e., the value of an object) changes while the component is in the process of painting?
- How can we perform a long operation on the event dispatcher thread (e.g., reading or writing a file) will block the whole UI
  - Because no other event can be dispatched until this operation completes.
JavaFX Threads

The system runs 2 or more of these threads at any given time:

- **JavaFX application thread**
  - This is the primary thread used by JavaFX application developers. Any “live” scene, which is a scene that is part of a window, must be accessed from this thread.

- **Prism rendering thread**
  - This thread handles the rendering separately from the event dispatcher. It allows the $i^{th}$ frame to be rendered while the $(i + 1)^{st}$ frame is being processed.

- **Media thread**
  - This thread runs in the background and synchronizes the latest frames through the scene graph by using the JavaFX application thread.
The javafx.concurrent package

- [https://docs.oracle.com/javase/8/javafx/JFXIP.pdf](https://docs.oracle.com/javase/8/javafx/JFXIP.pdf)
  - Very detailed
  - Read only the relevant portions on an “as-needed” basis

- The GUI of a JavaFX application is not thread-safe
  - Can only be modified from the JavaFX application thread.
  - Any long-running task will make the UI unresponsive.
    - These tasks are executed by background threads while the application thread processes user events.

- The recommended way is to use the javafx.concurrent package.
  - takes care of multithreaded code that interacts with the UI
  - ensures that interactions happen on the correct thread
The javafx.concurrency package

- One interface: Worker
- Two implemented classes: Task & Service

The Worker interface
  - provides APIs used by the background workers to communicate with the UI

The Task class
  - implementation of the java.util.concurrent.FutureTask class
  - enables developers to implement asynchronous tasks in JavaFX applications.

The Service class
  - executes tasks.
The Worker

- Defines an object to perform some work on one or more background threads

- The Worker object is *observable*
  - *i.e.*, it can be used from the JavaFX application thread.

- The Worker object has multiple states:
  - READY
  - SCHEDULED
  - RUNNING
  - SUCCEEDED

  The “value” contains the result of this Worker object.

  The “exception” property is set to the exception that occurred.
The Task

- This class needs to be extended by the programmer.
  - Your implementation must override the `call()` method to do the background work and return the result.
  - The `call()` method is invoked on the background thread
    - Can only manipulate states that are safe to change from a background thread.
    - *E.g.*, if you try to change an active scene graph, a runtime exception will be thrown.
    - Inside the `call()` method, you can use the `updateProgress()`, `updateMessage()`, and `updateTitle()` methods to update the values of the corresponding properties on the application thread.
    - If the task was canceled, the return value of `call()` is ignored.
The Task

■ It is within the scope of the Java concurrency libraries because
  - inherits from `java.util.concurrent.FutureTask`
  - which implements Runnable

■ A task can be started in one of the following ways:
  1. Start a thread with the task as parameter:
     ```java
     Thread t = new Thread(task);
     t.setDaemon(true); // JVM can exit while thread is still running
     t.start();
     ```
  2. By using the `ExecutorService` API:
     ```java
     ExecutorService.submit(task);
     ```