COMPSCI 230 S2C 2013
Software Design and Construction

A Thread's Life

Lecture 2 of Theme C

Lecture Plan for Weeks 10-12

18/5	Introduction to Java threads	Sikora pp. 157-9, Goetz1 pp. 1-6.
21/5	A thread's life	Goetz1 pp. 6-10.
22/5	Where Java threads are used; synchronization	Goetz1 pp. 10-15.
25/5	Locking, blocking, mutex; visibility, consistency.	Goetz1 pp. 15-20.
28/5	Deadlock; performance; programming guidelines.	Goetz1 pp. 20-24.
29/5	Dealing with InterruptedException (intro)	Goetz2 pp. 1-3.
1/6	Executors, tasks, concurrent collections, synchronizers.	Bloch pp. 271-7.
4/6	Concurrency in Swing	Oracle
5/6	Debugging Swing / Revision of this unit	Potochkin

Learning Goals for Today

- Refine your understanding of threading:
  - Make a careful distinction between the support of an operating system (or a computer) for running a thread, and an instance of a Thread object in the execution of a Java program.
- Understand the "lifecycle" of a thread
  - Start to analyse a multi-threaded application, by identifying "where" in the code the state of a thread can change state i.e. are created, become runnable, start to wait, stop waiting, and are terminated.

Ways to Create Threads

- When a Java program is launched, it has a single thread running main().
  - This is called the main thread.
- Any Java thread can create a new thread,
  - By instantiating the Thread object, or
  - By instantiating an object which extends Thread.

Java Threads and OS-supported Threads

- The word "thread" is ambiguous.
  - It may mean an object of type Thread: this is a Java data structure.
    - One field in this structure is its run() method.
  - It may mean a locus of control in a computer system.
    - Goetz calls this the "actual thread".
    - The operating system may provide direct support for multiple threads per process.
    - The JVM time-shares its OS-provided threads among its Thread objects. These objects can "come to life" only when they are paired up with an OS-supported thread.
- By analogy:
  - a Java thread object is like a soul, and
  - an OS-supported thread is like a body,
  - in an OS-defined universe where
    - Souls are repeatedly incarnated in different bodies,
    - Souls inhabit at most one body at any given time, and
    - Bodies persist much longer than souls.

The Life-Cycle of a Thread (at "Birth")

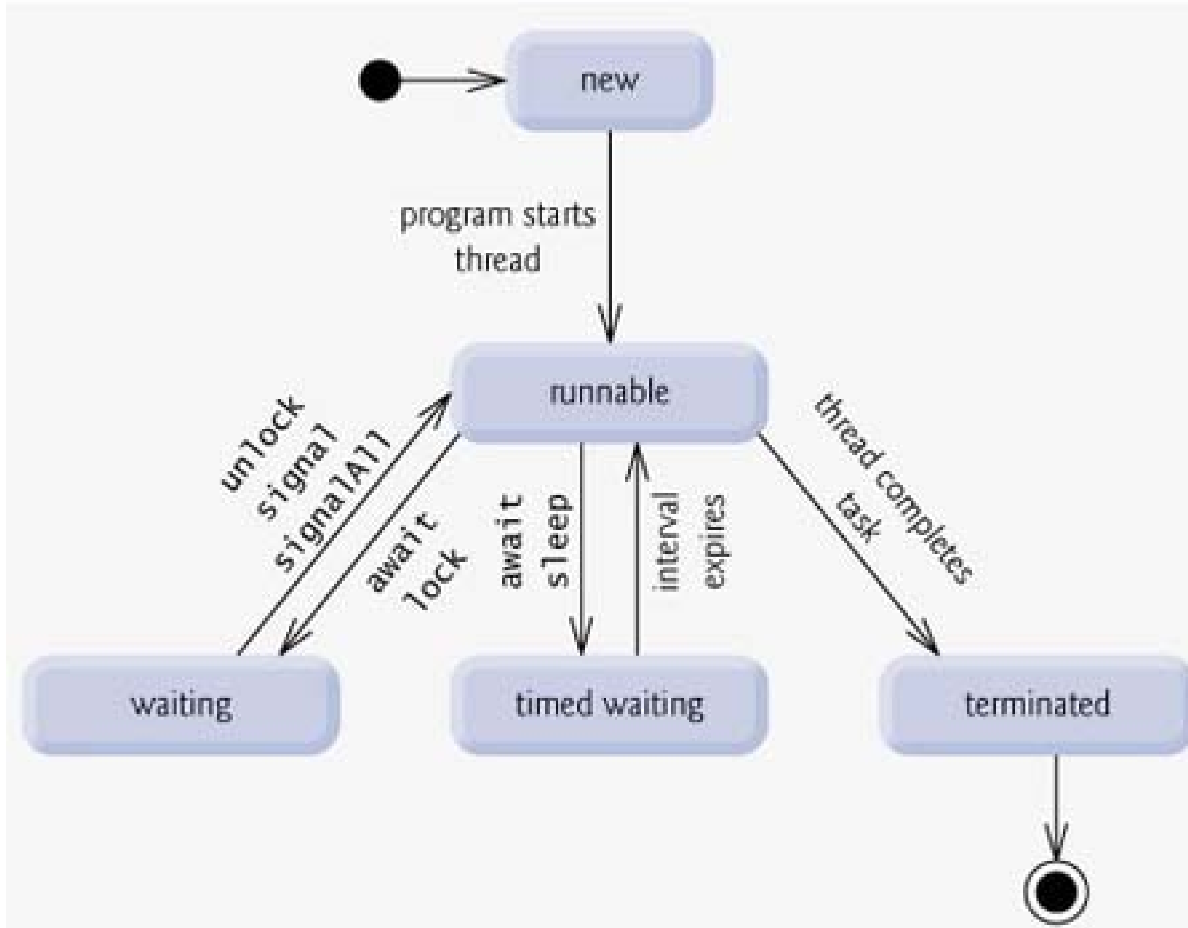
- Instantiation (Thread.State = NEW):
  - A new object *t* of type Thread is created.
    - Analogy: a soul with no body. Its methods and initial state are its karma (षन्चित कर्म).
  - Any running thread can instantiate a new thread.
- Inspiration (RUNNABLE for the first time):
  - Some running thread invokes *t*'s start() method.
  - Now *t* is ready to run()… but it needs a body!
  - Warning: if *t*'s inspiration occurs before its instantiation is complete, then it might start to run() before all of its instance variables are initialised. This will lead to very unpredictable – even dangerous – behaviour. The constructor method for a Thread object should not invoke this.start()!
- First incarnation (actually running for the first time):
  - The JVM has given it a "body" (an OS thread), so it starts to execute its run() method.

Life-Cycle of a Thread (after birth)

- After birth, Java threads are usually in one of the following states:
  - RUNNABLE, BLOCKED, WAITING, and TIMED WAITING.
- There is a fifth state:
  - TERMINATED.
  - This state allows the garbage collector to (attempt to) reclaim any resources left behind by a thread that has exited, and which are (apparently) inaccessible to any non-TERMINATED thread.
  - It also allows the programmer (through the JDI) to inspect the residue of a thread, that is, the final state of its instance variables, and any resources accessible through these variables.
- Thread states are adjusted by the JVM, in response to the thread's requests and also by external events.
  - The Thread.getState() method will reveal a recent state of a thread –
    - This is stale information (especially for this.Thread.getState()), so you should not rely on it for scheduling decisions.
    - It is very helpful for performance-monitoring and debugging.

A State Diagram for Threads

- http://www.tutorialspoint.com/java/java_multithreading.htm :



- This is a Petri net model for the JVM's management of threads.
  - The start node is at the top: any number of tokens can be placed here.
  - The JVM moves a token from one place to the next, in response to the events described on the arcs.

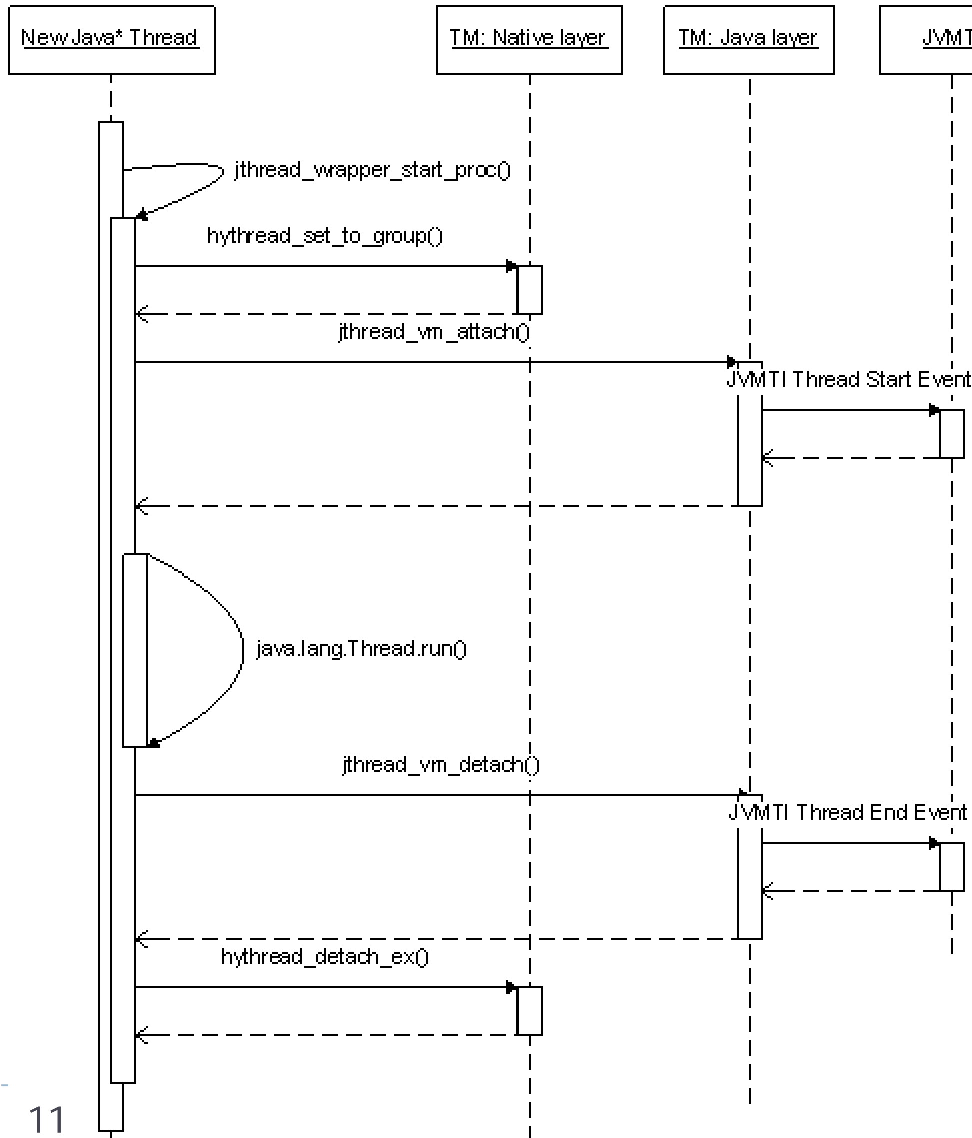
Death of a Thread

- It's not a sad event… usually…
- Normal termination:
  - A thread reaches the end of its run() method.
- Abnormal termination:
  - A thread throws an Exception or Error that isn't caught.
  - Try to catch and handle all exceptions and errors!
  - An abnormally terminated thread may be holding some resources that won't be "recycled" appropriately, e.g. file handles, issued by the operating system, which may prevent other processes from accessing this file until the handle is released.
- Terrible termination (deprecated):
  - Another thread calls stop(). ("Inherently unsafe… causes a thread to unlock all of the monitors that it has locked… " in SE1.4/Java 2)
  - Another thread throws a ThreadDeath error (and this thread doesn't catch it).

Joining Threads

- This probably sounds like a marriage, but it's something that happens after a thread's death!
- When a thread calls `t.join()`, it will block until `t` terminates.
  - This is usually understood to be an assurance that whatever `t` was doing in its `run()` is completed.
  - But the actual situation is more complicated…
    - Any external activity that `t` started (e.g. a disk-write) may not be completed by the time `t` terminates.
    - Any main-memory updates that `t` started (e.g. by writing to an unsynchronised and non-volatile object) may not be completed by the time `t` terminates.
    - Any updates that `t` completed (e.g. by writing to a synchronised or volatile object) before it reached the end of its `run()` will be complete when `t.join()` returns.

Another way to visualise threads



- This is a "swim lane" diagram.
- Threads start at the top.
- They move downwards, sending messages to other lanes, as indicated by arrows.
- A thread must wait for an incoming arrow before proceeding any farther down its path.
- Object.wait() will cause a thread to wait until some other thread invokes the notify() method of this object.
- Object.wait(timeout) allows a thread to proceed without a notify(), after the specified length of time.

Source: http://harmony.apache.org/subcomponents/drlvm/TM.html

Learning Goals for Today

- Refine your understanding of threading:
  - Make a careful distinction between the support of an operating system (or a computer) for running a thread, and an instance of a Thread object in the execution of a Java program.
- Understand the "lifecycle" of a thread
  - Start to analyse a multi-threaded application, by identifying "where" in the code the state of a thread can change state i.e. are created, become runnable, start to wait, stop waiting, and are terminated.