JAC444 - Introduction to Java for C++ Programmers

Lesson 9: Multi-threaded Programming

Agenda

- Concurrency Concepts
- The Life Cycle of a Thread
- Multi-threaded Programming
- Critical Section

Processes vs. Threads

- A process is an instance of a computer program that is being executed.
  - It runs independently and isolated of other processes.
  - It cannot directly access shared data in other processes.
  - The resources of the process, e.g. memory and CPU time, are allocated to it via the operating system.
- A thread is a sequence of executing instructions that can run independently within a process.
  - Threads organize programs into logically separate paths of execution.
  - A thread can perform task independent of other threads.
  - Threads can share access to common resources.
  - If a thread reads shared data it stores this data in its own memory cache.

Concurrency

- What is concurrency?

  Concurrency is the ability to run several programs or several parts of a program in parallel.

- Java concurrent programming is mostly concerned with threads.
  - A Java application runs in its own process and in a default thread.
  - Within a Java application you can work with several threads to achieve parallel processing or asynchronous behavior.

An Example

- Showcases: BallBouncing2.java

- The goal of thread programming (application level):

  "doing several things at once"

Applications of Multithreading

- An applet that does animation
  - a thread is used to carry out the task of drawing pictures
- The tasks of a multi-threaded server program
  - multiple threads are used to process the requests of multiple clients
- Java servlet technology
  - a servlet container creates multiple threads to process multiple user requests for the service of a Java servlet

Applications of Multithreading

- RMI technology
  - JVM creates one or more threads to execute remote requests
- The concurrent tasks of a web browser program
  - scroll a page
  - download an applet
  - play animation and sound
- The concurrent tasks of a responsive interactive program
  - monitor GUI events
  - calculations as requested
  - I/O

How to create and start a thread?

- Option 1: "extends" the Thread Class
  - override the run( ) method
  - example: ThreadDemo.java, ThreadDemo1.java
- Option 2: "implements" the Runnable interface
  - implement the run( ) method
  - pass the Runnable object into the Thread constructor
  - example: ThreadDemo3.java
- Notice
  - All examples invoke Thread.start in order to start the new thread.
  - The Runnable object can subclass a class other than Thread.

The Thread Class

- Thread class implements the Runnable interface.
- Static methods of the Thread class that control the current thread.
- Some static methods:
  - sleep(long milliseconds)
    - Causes the currently executing thread to sleep
    - state: running => sleeping
    - at expiration time: state: runnable <= sleeping
  - getName( )
  - currentThread( )
    - Returns a reference to the currently executing thread object.

How to stop a thread?

- An *interrupt* is an indication to a thread that it should stop what it is doing and do something else.
- A thread sends an interrupt by invoking interrupt method on the Thread object for the thread to be interrupted.
  - The stop() method is unsafe and has been depecated.
- e.g. InterruptDemo.java

```
Thread t = new Thread(this);
t.start(); // this will call run() method
… …
if (!t.interrupted()) { t.interrupt(); }
```

How to stop a thread?

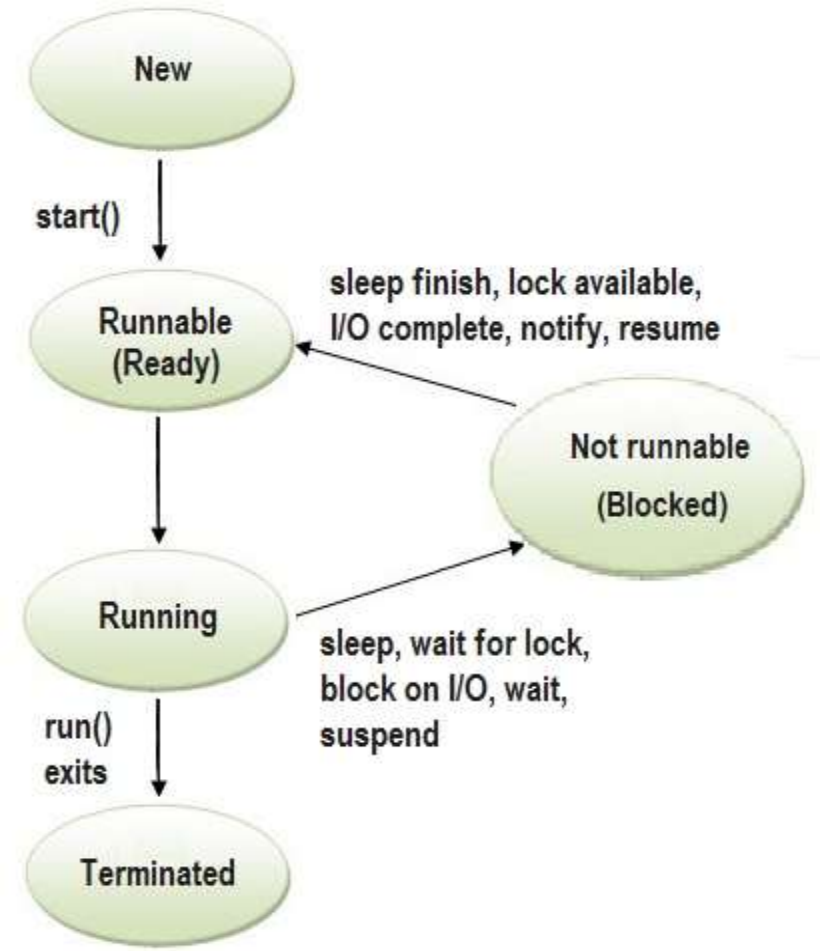
- InterruptedException is thrown when a thread is waiting, sleeping, or otherwise paused for a long time and **another thread interrupts it** using the interrupt( ) method in class Thread (API documentation)
- a checked exception
- e.g. (within run() method)

```
try {
    Thread.sleep(4000);
} catch (InterruptedException e) {
    System.out.println(t.getName() + " interrupted: ");
    return;   }
```

The Life Cycle of a Thread

Thread States

- New (i.e. a new thread)
- Runnable/ready
- Running
- Not runnable
  - Sleeping
  - Waiting
  - Blocked
- Terminated
  - exit from the run( ) method
  - uncaught exception



The Life Cycle of a Thread

- State transitions triggered by the methods
  - sleep( ): running => sleeping
  - wait( ) : running => waiting
  - notify( ), notifyAll( ) : waiting => runnable
  - yield( ) : running => runnable
  - interrupt( ): running => runnable
  - I/O: running => blocked
  - completion of run( ): running => terminated
- Do not use anymore stop(), suspend(), resume() methods.

Multi-threaded Programming

- Unrelated threads
- Related threads that do not interact (i.e. no data sharing)
  - examples: MultipleThreads.java,
    FastFoodService.java,
    FastFoodServiceGUI.java
- Threads that share data and avoid interference
- Threads that pass data back and forth
  - the famous example:
    the producer-consumer scenario

The Problem of Race Condition/Data Corruption

- Multiple threads that share a data object
  - access of the shared object by one thread is interfered by other threads. Example:

- The problem at the system level:
  - the timing of thread scheduling and be predicted
  - example: TwoThreads.java
  - fundamental issue: Which thread runs faster?

The Problem of Race Condition/Data Corruption

- The problem at the application level: data corruption

Race Condition	Example Bank Account
getResource();	*a = A.getBalance();*
modifyResource();	*a += deposit;*
setResource();	*A.setBalance(a);*

- Example:
  - AnotherBank.java

Solution: Use Synchronized Blocks/Methods

- Solution:
  - Critical Sections – Synchronized Blocks/Methods
- Critical Section Definition.
  - Any part of the code in a program with the property that only one thread can execute it at any given time is called critical section.
- Critical sections are called sometimes Monitors.
- Only one condition is called Mutual Exclusion or the ability to lock.

Synchronization and Blocking

- Synchronization:
  - access of the shared data object by other threads are blocked until the current thread has completed its access
- Blocking mechanism:
  - acquisition and release of a lock on the shared object (implemented by the JVM)

Synchronized Blocks and Methods

- Synchronized Method:
  - Example:

```
public class  SynchronizedCounter {
      private int c = 0;
      public synchronized  void increment() { c++; }
      public synchronized  void decrement() { c--; }
      public synchronized  int value() { return c; }
}
```

  - Example: AnotherBankSync.java
  - Example: AnotherBankNotify.java

Synchronized Blocks and Methods

- Synchronized Block:
  - Example

```
//Use Collecions.synzhonizedList method
List list = Collections.synchronizedList(new ArrayList());
...

//Use iterator on the synchronized list
synchronized (list) {
    Iterator iterator = list.iterator();
    while (iterator.hasNext())
    ...
    iterator.next();
    ...
}
```

Volatile Variables

- If a variable is declared with the volatile keyword then it is guaranteed that
  - Any thread that reads the field will see the most recently written value.
  - The volatile keyword will not perform any mutual exclusive lock on the variable.

Resourceful Links

- Java Tutorial on Concurrency
- Synchronization in Java

Thank You!