Threads

Contributed by Amir Kirsh
The operation we want to be threaded:

```java
public class PrintNumbers {
    static public void printNumbers() {
        for(int i=0; i<1000; i++) {
            System.out.println(Thread.currentThread().getId() + ": " + i);
        }
    }
}
```
Threads in Java

**Option 1** – extending class Thread:

```java
public class Thread1 extends Thread {

    @Override
    public void run() {
        System.out.println("Thread1 ThreadId: " + Thread.currentThread().getId());
        // do our thing
        PrintNumbers.printNumbers();
    }
}
```
Threads in Java

**Option 1 – extending class Thread (cont’):**

```java
static public void main(String[] args) {
    System.out.println("Main ThreadId: " +
            Thread.currentThread().getId());
    for(int i=0; i<3; i++) {
        new Thread1().start(); // don't call run!
        // (if you want a separate thread)
    }
    printNumbers();
}
```
Option 2 – implementing Runnable:

```java
public class Thread2 implements Runnable {

    @Override
    public void run() {
        System.out.println("Thread2 ThreadId: " + Thread.currentThread().getId());
        // do our thing
        PrintNumbers.printNumbers();
    }
}
```
Option 2 – implementing Runnable (cont’):

```java
static public void main(String[] args) {
    System.out.println("Main ThreadId: " +
                        Thread.currentThread().getId());
    for(int i=0; i<3; i++) {
        new Thread(new Thread2()).start();
        // again, don't call run!
        // (if you want a separate thread)
    }
    printNumbers();
}
```
Synchronization of threads is needed for in order to control threads coordination, mainly in order to prevent simultaneous operations on data.

For simple synchronization Java provides the synchronized keyword.

For more sophisticated locking mechanisms, starting from Java 5, the package java.concurrent.locks provides additional locking options, see: http://java.sun.com/javase/6/docs/api/java/util/concurrent/locks/package-summary.html
Example 1 – synchronizing methods:

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

The `synchronized` keyword on a method means that if this is already locked anywhere (on this method or elsewhere) by another thread, we need to wait till this is unlocked before entering the method.
Synchronization

Example 2 – synchronizing blocks:

```java
public void addName(String name) {
    synchronized (this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

When **synchronizing a block**, key for the locking should be supplied (usually would be **this**). The advantage of not synchronizing the entire method is **efficiency**.
Example 3 – synchronizing using different locks:

```java
public class TwoCounters {
    private long c1 = 0, c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized (lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized (lock2) {
            c2++;
        }
    }
}
```

You must be absolutely sure that there is no tie between c1 and c2
Synchronization

Example 4 – synchronizing static methods …

Having a static method be synchronized means that ALL objects of this type are locked on the method and can get in one thread at a time.

The lock is the Class object representing this class.

The performance penalty might be sometimes too high – needs careful attention!
Example 4 – synchronizing static methods:

```java
public class Screen {
    private static Screen theScreen;
    private Screen() { /* private c’tor */
    }

    public static synchronized getScreen() {
        if (theScreen == null) {
            theScreen = new Screen();
        }
        return theScreen;
    }
}
```

This is a Singleton example

It is not the most efficient way to implement Singleton in Java
wait(), notify(), notifyAll()

wait() and notify() allows a thread to wait for an event

A call to notifyAll() allows all threads that are on wait() with the same lock to be released

A call to notify() allows one arbitrary thread that is on a wait() with the same lock to be released

Read:
(a) http://java.sun.com/docs/books/tutorial/essential/concurrency/guardmeth.html
(b) http://java.sun.com/javase/6/docs/api/java/lang/Object.html#wait()
wait(), notify(), notifyAll()

Example

public class Drop {

    // Message sent from producer to consumer
    private String message;

    // A flag, True if consumer should wait for
    // producer to send message, False if producer
    // should wait for consumer to retrieve message
    private boolean empty = true;

    ...
class Drop {
  ...
  public synchronized String take() {
    // Wait until message is available
    while (empty) {
      // we do nothing on InterruptedException
      // since the while condition is checked anyhow
      try { wait(); } catch (InterruptedException e) {} 
    }
    // Toggle status and notify on the status change
    empty = true;
    notifyAll();
    return message;
  }
  ...
}
public class Drop {
    ...
    public synchronized void put(String message) {
        // Wait until message has been retrieved
        while (!empty) {
            // we do nothing on InterruptedException
            // since the while condition is checked anyhow
            try { wait(); } catch (InterruptedException e) {}  
        }
        // Toggle status, store message and notify consumer
        empty = false;
        this.message = message;
        notifyAll();
    }
    ...
}