Concurrency

Introduction

- Concurrency is the ability of an application to perform or appear to perform multiple tasks at once.
- Made possible by:
  - Multiple processors
  - Multiple execution cores
  - OS enabled time slicing on single-processor, single-core hardware
- Support for concurrency in Java:
  - Basic concurrency support
  - High-level concurrency API since Java 5.0

Threads

- Application -> Processes -> Threads
  - Each process has a private set of resources, in particular a separate memory space.
  - Threads share process resources such as memory and open files.
- Java provides the **Thread** class.

Defining and starting a Thread using Runnable

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

Defining and starting a Thread by subclassing Thread

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

Example: Threads - Threading

Synchronized code

- Two synchronization idioms:
  - Synchronized methods
  - Synchronized statements
- Synchronized method example:

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

Intrinsic Locks

- Synchronization is built around the concept of **intrinsic locks** a.k.a. **monitor locks** (monitor for short).
- Every object has a monitor associated to it.
- Exclusive access to an object’s synchronized methods is possible by owning the object’s monitor. Once the synchronized method completes, the monitor gets released.
- Only one Thread can own an object’s monitor at a given time.

Synchronized statements

Synchronized statements must specify the object that provides the monitor.

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

Synchronized statements

```
public class MsLunch {

    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

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

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

Pausing a Thread

- Invoking **Thread.sleep(long)** causes the current thread to suspend execution for a specified period.
- Not guaranteed to be precise.
- Sleeping can be terminated by interrupts.

Example: Threads - PausingAThread

Interrupts

- An interrupt is an indication that a Thread should stop what it’s doing and do something else. It’s up to the programmer to decide how a thread responds to an interrupt.
- Interruption is achieved by invoking the **Thread.interrupt()** method on the Thread that needs to be interrupted.

Responding to interrupts

- Periodically invoking **Thread.interrupted()** which returns true if an interrupt has been received.

```
for (int i = 0; i < importantInfo.length; i++) {
    if (Thread.interrupted()) {
        // We've been interrupted: no more messages.
        return;
    }
    // Print a message.
    System.out.println(importantInfo[i]);
}
```

and ...

Responding to interrupts

- Additionally act upon an **InterruptedException**:

```
for (int i = 0; i < importantInfo.length; i++) {
    if (Thread.interrupted()) {
        // We've been interrupted: no more messages.
        return;
    }
    // Pause for 4 seconds
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        // We've been interrupted: no more messages.
        return;
    }
    // Print a message.
    System.out.println(importantInfo[i]);
}
```

Example: Threads - Interrupts

Joins

The **join()** method allows one thread to wait for the completion of another.

Example:

If t is a Thread object whose thread is currently executing,

```
t.join() or t.join(long);
```

cause the current thread to pause execution until t's thread terminates.

Example: Join

Guarded block

- Threads often have to coordinate their actions. One of the most common idioms is a **guarded block** where a thread needs to wait until certain condition is met.
- A non-efficient implementation would look like this:

```
public void guardedJoy() {
    // Simple loop guard. Wastes processor time. Don't do this!
    while(!joy) {}
    System.out.println("Joy has been achieved!");
}
```

Guarded block - Efficient implementation

```
public synchronized void guardedJoy() {
    // This guard only loops once for each special event, which may not be the event we're
waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}
// …
public synchronized notifyJoy() {
    joy = true;
    notifyAll();
}
```

Example: Threads - GuardedBlock

Exercises

Exercise: NumberPrinter

- Create an application that prints out the numbers starting from one (1) up until a specified number. The print process should be interrupted if it fails to complete in a given time interval.
- Specify the number and the time interval as arguments to the application.

Exercise: StopWatch

- Create a stopwatch component that prints out each passing second until terminated by a command from the console captured from the standard input.
- Additional commands should allow the stopwatch to be paused and resumed.