Java Threads and Synchronization

Overview
The Class **Thread**

- Several ways to create threads.
  - Using the class **Thread**
  - Using the interface **Runnable**

- **Runnable** is more complex, but also more flexible (sometimes the simple method is insufficient).

- Here we describe only the simple method that uses the class **Thread**.
Issues

- Two issues:
  - Creation of **threaded code**:
    - Note: in OSP-2, you were only managing threads, i.e., doing only what the OS does.
    - Here we are talking about the programmer's point of view:
      - creation of threaded code, which runs as a bunch of threads.
  - Synchronization:
    - synchronizing different pieces of code in Java.
    - based on the idea of monitors
Threaded Code

- Create a class that **extends** `Thread`. As many classes as the application needs.
  - class `Consumer` extends `Thread`
  - class `Producer` extends `Thread`

- Put the code that is supposed to run as threads inside the method `run()`.
  - This method overrides what is inherited from class `Thread`.
  - Application classes (such as `Consumer` & `Producer`) can have other methods as well.
Threaded Code

- Have another class that drives the application. It creates instances of the threads and starts them.

```java
public class ConsumerProducer {
    private static Vector buffer = new Vector();

    public static void main(String args[]) {
        Consumer c1, c2;
        Producer p1, p2, p3;
        c1 = new Consumer("Bob");
        c2 = new Consumer("Alice");
        p1 = new Producer("Acme");
        ...
        c1.start();
        c2.start();
        p1.start();
        ...
    }
}
```

Code to be called by threads:

```java
public static void put(Object obj) {
    buffer.add(obj);
}

public static Object take() {
    while (buffer.size() == 0) {
    }
    return buffer.remove();
}
```

Driver code:

```java
```

// shared buffer
Threaded Producer & Consumer with Infinite Buffer

public class Producer extends Thread {

    public void run() {
        while (true) {
            ConsumerProducer.put(getNewItem());
        }
    }

    MyItem getNewItem() {
        MyItem item = new MyItem();
        ... put stuff in item ...
        return item;
    }
}

public class Consumer extends Thread {

    public void run() {
        while (true) {
            ConsumerProducer.take();
        }
    }
}

Problems With Our Code

- If `buffer.size() == 0`, `take()` loops - bad.
- In case of concurrent consumers, several can fall through the loop
  ```java
  while (buffer.size() == 0) { }
  ```
  - If a producer puts 1 item in the buffer, the first concurrent consumer executes `remove()`
  but the second will cause an error.
Solution: Java Monitors

- Change the put/take methods as follows:

  ```java
  public synchronized static void put(Object obj) {
    buffer.add(obj);
  }
  
  public synchronized static Object take() {
    while (buffer.size() == 0) {
    }
    return buffer.remove();
  }
  
  Busy wait is still a problem!
  ```
Eliminating Busy Wait: \texttt{wait/notify()}

- Change put/take methods as follows:

```java
public synchronized void put(Object obj) {
    buffer.add(obj);
    ConsumerProducer.class.notify();
}

public synchronized Object take() {
    try {
        if (buffer.size() == 0)   ConsumerProducer.class.wait();
        return buffer.remove();
    } catch (InterruptedException ie) {
        System.err.println("Someone interrupted my work");
    }
}
```

- \texttt{wait/notify()} can operate on any object. Here on \texttt{ConsumerProducer.class}.
- \texttt{notify()} notifies the first waiting thread.
- \texttt{notifyAll()} notifies all waiting threads.
Additional Features

- The previous technique uses ConsumerProducer as a monitor and calls wait() / notify() on this class-object.
  - In general, wait/notify can work on any object.
  - Thus, objects act as conditional variables of monitors.

- Our monitor is rather coarse – the entire class ConsumerProducer.
  - Java lets one declare pretty arbitrary blocks of code as belonging to the same named monitor.