Monitors in Java

Model and Examples
Monitors in Java

Basic Ideas:

• every object has a single lock and a waiting queue
• the wait() operation suspends the executing thread on the object’s waiting queue and releases the object’s lock
• the notify() operation awakens exactly one thread suspended on the object’s waiting queue but does not release the lock
• the notifyAll() operation awakens all threads suspended on the object’s waiting queue but does not release the lock
• wait, notify, and notifyAll must be in synchronized methods
• synchronized methods acquire the lock before executing
• awakened threads must reacquire the lock before continuing
Monitors in Java

- **acquire**
- **lock**
- **release**
- **lock wait queue**
- **call**
- **return**
- **Synchronized method**
- **object**
Monitors in Java

- **Synchronized method**
- **lock**
- **acquire**
- **release**
- **wait**
- **waiting queue**
- **object**
- **call**

Diagram:

```
+------------------+
|                  |
|  acquire         |
|                  |
+------------------+
|                  |
|  lock            |
|                  |
+------------------+
|                  |
|  release         |
|                  |
+------------------+
|                  |
|  wait()          |
|                  |
+------------------+
|                  |
|  Synchronized    |
|  method          |
|                  |
+------------------+
|                  |
|  object          |
|                  |
+------------------+
|                  |
|  waiting queue   |
|                  |
+------------------+
```
notify()
The diagram illustrates the `notifyAll()` method in the context of synchronization. It shows the `acquire` method being called on a lock, which then awakens the waiting queue. The `notify()` method is also shown, indicating a synchronized method call.
Semaphore code (from Semaphore.java)

// file Semaphore.java
// note: some details missing
public class Semaphore
{
    private int count;
    public Semaphore(int initial)
    {
        count = initial;
    }
    synchronized public void P()
    {
        count = count - 1;
        if(count < 0) wait();
    }
    synchronized public void V()
    {
        count = count + 1;
        if (count <= 0) notify();
    }
}
Code from ReadersWriters.java

//file ReadersWriters.java
public class ReadersWriters
{
priivate boolean writing;
priivate int readers;
publiC ReadersWriters()
{
   writing = false;
   readers = 0;
}
publiC synchronized void startRead()
{
   while (writing)
   try { wait(); }
   catch (InterruptedException ioe) {} 
   readers = readers + 1;
}
publiC synchronized void endRead()
{
   readers = readers - 1;
   if (readers == 0) notify();
}
publiC synchronized void startWrite()
{
   while (writing || readers > 0)
   try { wait(); }
   catch (InterruptedException ioe) {} 
   writing = true;
}
publiC synchronized void endWrite()
{
   writing = false;
   notifyAll();
}
Threads in Java

A Java thread is an independent activity executing concurrently with all other threads. Two methods for controlling threads are:

- \texttt{start()} : begin the activity of the thread
- \texttt{stop()} : halt permanently the activity of the thread
Threads in Java

Two ways of defining a thread to perform application specific activities:

- extend Thread class and define a run() method
- create a class that implements the Runnable interface
Extending the Thread Class

Defined like this:

```java
public class Reader extends Thread
{
   // …
   public void run()
   {
      // what to do
   }
}
```

Used like this:

```java
Reader reader = new Reader();
reader.start();
```
Implementing Runnable

Defined like this:

```java
public class Reader implements Runnable {
    // …
    public void run()
    {   // what to do
        }
    }
}
```

Used like this:

```java
Reader reader = new Reader();
Thread thread = new Thread(reader);
thread.start();
```