Lecture 4

JAVA (46-935)

Somesh Jha

Inner Classes

- Flashback to the *AbstractTermStructure* class.

- There was a *SlowYieldVolObject* which computed:
  - ($i = 0$) Difference between *computed yield* and the *market yield*
  - (otherwise) Difference between *computed volatility* and the *market volatility*.

- *Clumsy!* The object used by the *NewtonRaphson* solver belongs in the *TermStructure* class.

- Nobody else uses it.

Inner Classes (Contd)

- Put a *YieldVolObject* class inside the *TermStructure* class.

- Nobody else except *TermStructure* class can use the *YieldVolObject*.

- These are called *member classes*.

- There are other kind of inner classes. (Read Chapter 5).

Code Fragment

```
/**
   Abstract class for building a BDT type interest-rate
   model.
   @author Somesh Jha
   */

package interestRate;

import mathUtil.*;

public abstract class TermStructure {


  private static final boolean DEBUG=false;

  //time horizon
  int T;

  //Used by the Newton-Raphson solver
  YieldVolObject slowYieldVolObj;
  NewtonRaphson slowSolver;

  //parameters of the BDT model
  double r[];
  double k[];

  //bond yields and yield volatilities
  //at time 0
  double yield[];
  double volatilities[];

  //nodes[i] points to link list of
  //nodes with time i
  LinkList nodes[];
```

```
  class YieldVolObject extends AbstractFunctionObject {


    //Yield and volatality are computed for the
    //bond of that maturity
    public int maturity;


    public YieldVolObject() {

      //call the constructor for the super class
      super(2);
    }//end of constructor


    //If i==0 calculate the yield and otherwise
    //calculate the vol. Use values as value
    //of r[t] and k[t]
    public double evaluate(int i, double val[]) {
      r[maturity-1]=val[0];
      k[maturity-1]=val[1];

      if (i==0) {
double tempYield =slowYield(0,0,maturity);
return(tempYield-yield[maturity-1]);
    }
      else {
double tempVol = slowLogVol(0,0,maturity);
return(tempVol-volatilities[maturity-1]);
      }
    }//end of evaluate

  }//end of YieldVolObject
```

Points to notice

- Notice that the class *YieldVolObject* is defined inside the *TermStructure* class.

- Notice that *YieldVolObject* has complete access to data of the *TermStructure* class.

- Nobody can access *YieldVolObject* from outside.

JAVA packages

- We have or will cover some classes from the following packages:
  - `java.applet` (Classes concerned with applets)
  - `java.awt` (Classes concerned with GUIs)
  - `java.io` (Classes concerned with I/O)
  - `java.util` (Classes concerned with various utilities)
  - `java.net` (Classes concerned with networking)

- Whole list of packages given on page 86-89 in the book.

JAVA I/O package

- We covered various kind of I/O streams.
- How to read and write to a file.
- A very wide variety of I/O provided by JAVA.

JAVA utlities package

- `StingTokenizer` is in the `java.util` package.

- Allows us to break a line into tokens for parsing.

- Consider the following fragment of code.

```
String line=''hhh:xxx:ccc'';
StringTokenizer tokenizer = new StringTokenizer(line,'':'');
String firstToken = tokenizer.nextToken();
String secondToken = tokenizer.nextToken();
```

Other interesting classes

- `HashTable`
- `Date`
- `Random`
- `Vector`
- `Stack`
- Read about them on page 527.

java.lang **package**

- This package is loaded up by *default*.
- `String`, `Math`, `System`, `Double` are all in this package.
- `Object` and `Exception` are also defined in this package.
- Read about it on page 442.

What is a thread?

- A *Thread* is a like a program.

- Two threads run independently like separate programs.

- Difference is that threads run within a program and hence can share variables.

Concurrent Programming

- Threads enable concurrent programming.

- Two separate tasks can be going on in parallel in a single program.

- Let us say you enter the *Bloomberg* website.

Concurrent Programming (contd)

- The following tasks can be going on in parallel:
  - Various indexes being displayed in a box.
  - Hot news ticker.
  - Asking the user to enter a stock symbol.

- All these separate tasks could be handled by separate threads.

A small example

```
package threadRelated;


public class MyThread extends Thread {


  public MyThread(String name) {

    super(name);
  }//end of MyThread


  public void run() {

    for(int i=0; i < 10; i++) {
      System.out.println(i+" "+getName());
      try {
sleep((int)Math.random()*1000);
      } catch (InterruptedException e) {};

    }
    System.out.println("DONE! "+getName());
  }//end of run

}//end of MyThread
```

Constructor

- A thread class extends the JAVA class `Thread` defined in the package `java.lang`.

- The constructor takes the name of thread as an argument.

- What does `super(name)` do?

run **method**

- Whenever a thread is started, `run` method is called.

- The `run` method in this case goes through the loop `10` times.

- `sleep((int)Math.random()*1000)` *suspends* the thread for a random time.

- What is `Math.random()`?

- `getName()` gets the name of the thread.

Main Program

```
package threadRelated;

public class testMyThread {

  static public void main(String argv[]) {


    MyThread thread1 = new MyThread("put");
    MyThread thread2 = new MyThread("call");


    thread1.start();
    thread2.start();

  }
}
```

Main Program (Contd).

- There are two threads: `thread1` and `thread2`.
- First thread has name `put` and the second one `call`.
- The `start` method on the thread starts the thread.

When does a thread stop?

- A thread stops when either of the following events happen:
  - `run` method exits.
  - `stop` method is called on the thread.

Output of the program

```
0 put
0 call
1 call
1 put
2 put
2 call
3 put
3 call
4 put
5 put
4 call
5 call
6 put
6 call
7 put
7 call
8 put
8 call
9 put
9 call
DONE! put
DONE! call
```

Synchronization

- Suppose that there are two threads $T1$ and $T2$ that share two variables i and j.

- Suppose each thread increments i and then increments j by the value of i.

- We can have two sample executions shown on the next slide.

Two executions

```
Initial value
i=1 j=1

First exceution
i = i+1 (T1 executes)
j = j+i (T2 executes)
i = i+1 (T2 executes)
j = j+i (T1 exceutes)

Second execution
i = i+1 (T1 executes)
j = j+i (T1 executes)
i = i+1 (T2 executes)
j = j+i (T2 executes)
```

Inconsistency!

- *First Exceution*
  $T1$ observes $j = 6$.

- *Second Exceution*
  $T1$ observes $j = 3$.

- Need to control access to shared variables.

- *Answer:* `Synchronization`

Toy Trading System

- Each trader enters the transaction he/she just made.

- A *producer* thread reads the transaction record and puts in a queue.

- *Consumer* threads read records from the queue and log it.

Toy Trading Example (Fig)

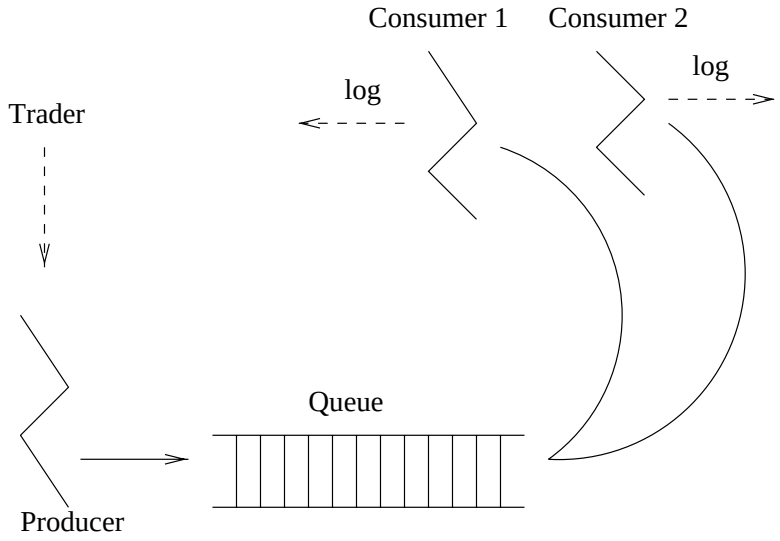

Figure 1: A Toy Trading System

Producer Thread

```
package threadRelated;

import java.io.*;

public class ProducerThread extends Thread {

  Queue myQueue;
  BufferedReader bSystemIn;
  public ProducerThread(String name, Queue q) {
    super(name);
    myQueue = q;
    bSystemIn = new BufferedReader(new InputStreamReader(System.in));
  }


  public void run() {

    try {
      String line;
      while ((line = bSystemIn.readLine()) != null) {
TraderEntry tEntry = new TraderEntry(line);
myQueue.add(tEntry);
      }//end of while
    }
    catch (IOException e) {
      System.err.println("Exception occured "+e.getMessage());
    }

  }//end of run
}
```

Constructor

- Constructor takes:
  - Name of the thread.
  - The queue to put things in.

run **method**

- Reads a line from the screen.

- Makes that line into a trader-entry.

- Adds that trader-entry to the queue.

TraderEntry

```
package threadRelated;

import java.util.*;

public class TraderEntry {

  //put, call,...
  String transactionType;

  //price and amount
  double price, amount;

  static final int NO_OF_ENTRIES=5;

  String counterParty;

  String comments;

  public TraderEntry(String tType, double p,
     double a, String cP,
     String co) {
    transactionType = tType;
    price = p;
    amount = a;
    counterParty = cP;
    comments = co;
  }//end of first constructor

  public TraderEntry(String line) {
    StringTokenizer tokenizer = new StringTokenizer(line);
    if (tokenizer.countTokens() == NO_OF_ENTRIES) {
      transactionType = tokenizer.nextToken();
      price = Double.valueOf(tokenizer.nextToken()).doubleValue();
      amount = Double.valueOf(tokenizer.nextToken()).doubleValue();
```

```
      counterParty = tokenizer.nextToken();
      comments = tokenizer.nextToken();
    }
  }//end of TraderEntry




  public String toString() {

    StringBuffer result=new StringBuffer(transactionType);
    result = result.append(" : ");
    result = result.append(price);
    result = result.append(" : ");
    result = result.append(amount);
    result = result.append(" : ");
    result = result.append(counterParty);
    result = result.append(" : ");
    result = result.append(comments);
    return(result.toString());
  }//end of toString()

}//end of TraderEntry
```

Constructors

- *First Constructor*
  This constructor takes all five arguments explicitly.

- *Second Constructor*
  Second constructor takes a *string* and parses the entries out of that.

toString **method**

- Notice the use of `StringBuffer` in this method.

- This class represents a string of characters.

- A `StringBuffer` object grows as things are appended to it.

ConsumerThread

```
package threadRelated;

public class ConsumerThread extends Thread {

  Queue myQueue;
  public ConsumerThread(String name, Queue q) {
    super(name);
    myQueue = q;
  }


  public void run() {

    while(true) {
      TraderEntry tEntry = (TraderEntry)myQueue.delete();
      System.out.print("Consumer <"+getName()+"> ");
      System.out.println(tEntry);
    }

  }//end of run
}
```

Constructor

- Same as the producer thread.

run **method**

- Goes on forever.
- Each time in the loop it gets an entry from the queue.
- Prints it out on the screen.
- What happens in the following statement?

```
System.out.println(tEntry);
```

Main program

```
package threadRelated;

import java.io.*;

public class testProducerConsumer {

  static public void main(String argv[]) {


    String line;
    Queue entryQueue = new Queue(1000);

    ProducerThread prod = new ProducerThread("producer",entryQueue);
    ConsumerThread consumer1 = new ConsumerThread("consume-1",entryQueue);
    ConsumerThread consumer2 = new ConsumerThread("consume-2",entryQueue);
    prod.start();
    consumer1.start();
    consumer2.start();


  }//end of main
}
```

Main program

- Has one producer thread `prod`.
- Has two consumer threads `consumer1` and `consumer2`.
- Starts the three threads.

Queue shared

- Notice that the queue `entryQueue` is shared between the three threads.

- Need to *synchronize* access to the queue.

- Only one thread should be accessing the queue object at any time.

Queue is empty

- What if a consumer thread wants to get a trader-entry and the queue is empty?
- Consumer thread should go into a *wait* state.
- When a producer thread puts something in the queue, it should *notify* the *waiting* consumer thread.
- Analogous situation happens when the queue is full.

Queue **class**

```
package threadRelated;

public class Queue {

  int size;
  private Object data[];
  int front, back;
  private boolean empty, full;

  public Queue(int size) {
    this.size = size;
    data = new Object[size];
    front=back=0 ;
    empty=true;
    full=false;
  }//end of Queue


  private boolean isEmpty() {
    return(empty);
  }


  private boolean isFull() {
    return(full);
  }


  public synchronized void add(Object obj) {

    while (isFull()) {
      try {
wait();
      }
```

```
      catch (InterruptedException e) {
      }
    }//end of while

    boolean wasEmpty = isEmpty();

    //queue has space
    data[front]=obj;
    if ((front+1)%size == back) full=true;
    front = (front+1)%size;
    empty=false;
    if (wasEmpty) notifyAll();

  }//end of add


  public synchronized Object delete() {
    while (isEmpty()) {
      try {
wait();
      }
      catch (InterruptedException e) {
      }
    }//end of while


    boolean wasFull = isFull();

    Object obj=data[back];
    if ((back+1)%size == front) empty=true;
    back = (back+1)%size;
    full=false;
    if (wasFull) notifyAll();
    return(obj);

  }//end of delete


  public synchronized String toString() {
    String result=" ";
```

```
    if (!isEmpty() ) {
      for(int i=back; i != front; i=(i+1)%size) {
result = result + data[i].toString();
result= result+" \n";
      }
    }//end of if
    return(result);
  }//end of toString

}//end of Queue
```

Queue class

- Implements a circular queue.
- Figure out the logic.
- The variable `size` holds the size of the queue.

delete **method**

- Deletes an object from the end of the queue and returns it.

- Notice the definition.

```
public synchronized Object delete()
```

- This means that a thread has to acquire the *unique lock* associated with this object before it can execute the `delete` method.

delete **method (Contd.)**

- Suppose the `consumer1` thread executes the `delete` method and acquires the *lock*.

- Now suppose the `prod` thread executes the `add` method.

- The thread `prod` *blocks* because the lock associated with the queue object is with the thread `consumer1`.

Waiting

- Suppose thread `consumer2` executes the method `delete` to get a trader-entry.

- Suppose the queue is empty.

- Thread `consumer2` puts itself in the `wait` state using the following fragment of code:

```
while (isEmpty()) {
  try {
wait();
  }
  catch (InterruptedException e) {
  }
}//end of while
```

Who wakes it up?

- Recall that `consumer2` is waiting.

- When the producer thread `prod` puts stuff in the queue and it was empty, it notifies the waiting threads.

- The fragment of code that does this is:

```
if (wasEmpty) notifyAll();
```

Client-Server Programming

- *Server* runs on a known host and a port.

- Has all the `heavy-weight` stuff in it.

  – Databases with historical data.
  – Complicated functionality (pricing and PDE code).

Client

- *Client* is a *light-weight* program that *uses* the server.

- Generally, the client *knows* the host and the port to connect to the server.

- Sockets enable client-server programming in JAVA.

Client-Server

- Most web-based services are client-server programs.

- Large risk-management tools (e.g., *Infinity*) are also client-server programs.

- JAVA makes client-server programming especially easy.