Systems Software

COMP20081

Lecture 15 – Java Support for Multithreading

Dr Michalis Mavrovouniotis

School of Science and Technology

ERD 200

Office Hours: Thursday 12:00-14:00

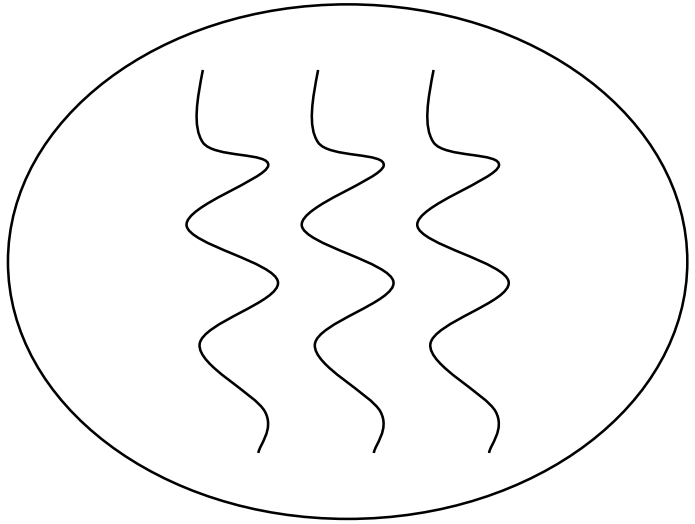
NOTTINGHAM TRENT UNIVERSITY

Recall and Lecture Overview

- Recall
  - Java Sockets
  - Streams and Datagrams
- Overview
  - How can Java support multithreading
  - Java Thread Class
  - Application Examples

Multithreading

- What are threads?
- Advantages and Disadvantages



Concurrency in Java

- Java Virtual Machine (JVM) allows an application to have multiple threads running concurrently
- Java class Thread from the **java.lang** package
- Two ways to create a new thread
  - **Extending** the *Thread* class
  - **Implementing** the *Runnable* interface

Extending the **Thread** Class

- Declare a subclass of Thread, overriding its run()

```
public class ExampleThread extends Thread {
        int parameter;
        ExampleThread(int p) {
            parameter = p;
        }

        public void run() {
            ....// what should the thread do?
        }
}
```

- Allocate and start an instance of the subclass (in the main method) as:

```
ExampleThread t = new ExampleThread(10);
t.start();
```

Implementing the **Runnable** Interface

- First declare a class implementing the Runnable

```
public class ExampleThread implements Runnable {
        int parameter;
        ExampleThread(int p) {
            parameter = p;
        }
        public void run() {
            ....// what should the thread do?
        }
}
```

- Threads can then be allocated and started (in the main class) by:

```
ExampleThread r = ExampleThread(10);
Thread t = new Thread(r);
t.start();
```

Implementing the **Runnable** Interface (Cont'd)

- Sometimes more convenient to implement run() from Java interface Runnable instead of extending from Thread
- Java does not permit multiple inheritance
- Slightly more work to setup the threads
- **Advantage**: we can extend other class while implementing Runnable

Methods of Thread Class

- Main methods of the Thread class
  - start(): causes this thread to begin execution; JVM will call the run() of this thread
  - run(): call the run method of the thread directly
  - sleep(long ms): cease the execution of current thread for ms milliseconds
  - wait(): blocks the thread
  - notify(): notifies other thread to unblock
- Refer to the JavaDocumentation about the Thread class

Java Thread Example - *Extending*

```
public class ThreadID extends Thread {
   int id;
   ThreadID(int _id){
      id = _id;
   }
   public void run(){
      System.out.println("This is Thread " + id);
   }
   public static void main(String[] args){
      System.out.println(“Main thread starts”);
      ThreadID t1 = new ThreadID(1);
      ThreadID t2 = new ThreadID(2);
      ThreadID t3 = new ThreadID(3);
      t1.start();
      t2.start();
      t3.start();
      System.out.println("Main thread ends");
   } }
```

Java Thread Example - *Implementing*

```
public class ThreadID implements Runnable{
   int id;
   ThreadID (int _id){
      id = _id;
   }
   public void run(){
      System.out.println("This is Thread " + id);
   }
   public static void main(String[] args){
      System.out.println(“Main thread starts”);
      ThreadID t1 = new ThreadID(1);
      ThreadID t2 = new ThreadID(2);
      ThreadID t3 = new ThreadID(3);
      Thread thr1 = new Thread(t1);
      Thread thr2 = new Thread(t2);
      Thread thr3 = new Thread(t3);
       thr1.start(); thr2.start(); thr3.start();
      System.out.println("Main thread ends");
   } }
```

The Passenger Problem

- There is an airplane with two doors (front and rear)
- Passengers can enter but not leave
- To determine the number of passengers in the airplane *at any time* a concurrent system is implemented which is connected with some sensors at each door.
- We simplify the problem for our experiment
  - Simulating 100 passengers entering each door assuming that we expect 200 passengers.
  - Count passengers as they enter the airplane
- When the experiment finishes, the system should show that there are 200 passengers in the airplane
- The nature is concurrency: two doors behave in parallel and the order of their events is unpredictable

Design of the Problem

- Each door is handled by a thread. The two door threads will run in parallel.
- A global variable represents the current number of passenger in the airplane.
- It is updated by a door thread when a passenger enters the front or rear door

Airplane and Door classes

```
public class Airplane {
    public int count = 0;

    void increment(){
        count = count + 1;
        System.out.println("There are " + count + " passengers");
    }

    public static void main(String[] args){
        Airplane ap = new Airplane();
        Door front = new Door(ap);
        Door rear = new Door(ap);
        Thread t1 = new Thread(front);
        Thread t2 = new Thread(rear);
        t1.start(); t2.start();
    }
} //end class
```

```
public class Door implements Runnable{
    Airplane ap;
    Door(Airplane _ap) {
        ap = _ap;
    }

    public void run(){
        for(int i = 1; i <= 100; i++) {
            try{
                ap.increment();
                Thread.sleep(500); //0.5sec every arrival
            } catch (InterruptedException e) { } //do nothing
        }
    }
} //end class
```

Object Sharing

- A shared object can be accessed by multiple threads
  - Typically data items
- When threads access shared data they can:
  - Read
  - Write
  - or Both

Synchronization

- We have to synchronize the shared object
- Java uses the **synchronized** keyword in the definition of the shared method

```
synchronized void increment(){
        count = count + 1;
        System.out.println("There are " + count + " passengers");
}
```

Implementing Multithreaded Server

- The server is a program that provides services to other programs
- A number of clients request for services form a server
- The server may not be on the same computer as clients
- When a server is required to serve many clients, we need an architecture in which the server does not completely serve one client and then wait for new ones
- Instead the server should always remain available and can accept several clients in parallel
- Threads are then employed to achieve this goal. The server does not perform the tasks on its own but delegate this to another class, typically defined as “**Handler”** class

Outline Code for Server and Handler

```
public class Handler implements Runnable {
   public Handler(Socket client) {
      // some code here
    } //constructor
    public void run() {
      //an entire task can be implemented here
     } //thread method
}
```

```
public class MultiThreadedServer  {
public static void main(String[] args)  {
      ServerSocket server = new ServerSocket(9090);
      while(true){
            System.out.println("Waiting…");
            Socket client = server.accept();
            System.out.println("Connected" + client.getInetAddress());
            //assign each client to a thread
            Handler t = new Handler(client);
            Thread th = new Thread(t);
            th.start();
       }
    }
}
```

TimeServer Example

```
import java.net.*; import java.util.*; import java.io.*;

public class TimeServer {
   public static void main(String[] args) throws IOException {
      ServerSocket server = new ServerSocket(9090);
      while(true){
            System.out.println("Waiting...");
            //establish connection
            Socket client = server.accept();
            System.out.println("Connected " + client.getInetAddress());
            //create IO streams
            DataOutputStream outToClient = new DataOutputStream(client.getOutputStream());
            //return date
            Date date = new Date();
            outToClient.writeUTF(date.toString()); //send date to client
      }
   }
}
```

TimeServer Example (cont'd)

```
import java.net.*; import java.util.*; import java.io.*;

public class MultiThreadedTimeServer {
   public static void main(String[] args) throws IOException {
      ServerSocket server = new ServerSocket(9090);
      while(true){
            System.out.println("Waiting...");
            //establish connection
            Socket client = server.accept();
            System.out.println("Connected " + client.getInetAddress());
            //assign the client to a the handler
            TimeHandler th = new TimeHandler(client);
            Thread t = new Thread(th);
            t.start();
      }
   }
}
```

TimeServer Example (cont'd)

```
import java.net.*; import java.util.*; import java.io.*;

public class TimeHandler implements Runnable {
    Socket client;
    DataOutputStream outToClient;

    public TimeHandler(Socket _client) throws IOException {
            client = _client;
            outToClient =  new DataOutputStream(client.getOutputStream());
    } //constructor

    public void run() {
       try {
                //return date
                Date date = new Date();
                outToClient.writeUTF(date.toString()); //send date to client
       } catch (IOException e) { }
   }  //end of run
} //end of class
```

Summary

- Discussed the concept of thread
- Create a thread in Java
  - Extending Thread class
  - Implementing Runnable interface
- Shared objects and Synchronization
- Multithreaded Server Application