Object-Oriented Programming CSE-703029

Lecture 11: Threads & JDBC

Faculty of Computer Science
Phenikaa University

Today's Topics

- Threads
- Intro to JDBC

Java's Thread Class (From Eckel)

- Classes are one way to encapsulate program code.
- Threads provide another way:
  - Your application or applet runs in its own process, with its own memory space.
  - The operating system allocates CPU cycles to your application's process (multi-tasking).
  - A thread is a subtask, that runs independently in your program's process.
  - **Java allocates** time to your **threads**.

Why Bother With Threads?

- The standard example is responsive user interfaces:
  - One part of your program is CPU-intensive (e.g., computing a new generation of symbolic regression trees).
  - You have a Pause button in your user interface, but clicking it does nothing because program control isn't currently checking for mouse clicks.

How Do Threads Help?

- One solution:
  - Have your Evolver methods constantly check if mouse clicks have come in.
  - But where should you add these checks?
  - What to do if you detect them?
  - This results in convoluted code (needlessly complex).
- A better solution:
  - Run the GUI code and number-crunching code in separate threads.

Thread: java.lang.thread Running

```
public class ClassName extends Thread {
    ClassName cn = new ClassName();
    cn.start();  //start concurrent prog

public void run (){
//override run() of thread
    }
}
```

Thread: java.lang.thread

Concurrency problem: unpredictable values

```
public class threadExample extends Thread{
public static int count = 0 ;
public static void main(String[] args) {
threadExample threadExp = new threadExample();
threadExp.start();
System.out.println("count "+ count);//0
count ++; //1
System.out.println("count after ++"+count); //2
}
public void run(){//overridden run method of thread
count ++; //1
}
}
```

Thread: java.lang.thread
isAlive(): to prevent Concurrency problem

```
public class threadExample extends Thread{
public static int count = 0 ;
public static void main(String[] args) {
threadExample threadExp = new threadExample();
threadExp.start();
while (threadExp.isAlive()){
System.out.println("running...");
}
System.out.println("count "+ count); //1
count ++; //2
System.out.println("count after ++"+count); //2
}
public void run(){//overridden run method of thread
count ++; //1
}
}
```

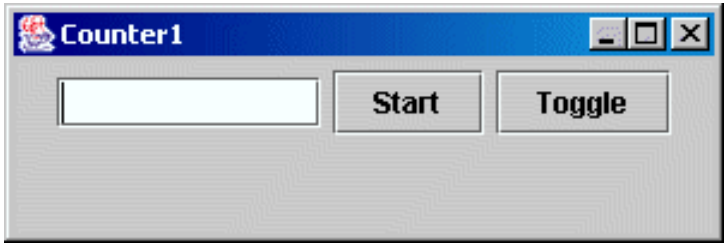
Thread: java.lang.thread

Reference Document:

https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

A Dysfunctional GUI

- Here's the outline:




```
public class Counter1 extends JApplet {
   private int count = 0;
   private boolean runFlag = true;
   private JButton    start = new JButton("Start"),
                      onOff = new JButton("Toggle");
   private JTextField t = new JTextField(10);
   public void init() { // set up the GUI }
   public void go() { // do something when Start is clicked
                      // but stop doing it if Toggle is clicked
   }
   class StartL implements ActionListener {
      public void actionPerformed(ActionEvent e) { go(); }
   }
   class OnOffL implements ActionListener {
      public void actionPerformed(ActionEvent e) { runFlag = !runFlag; }
   }
}
```

A Dysfunctional GUI

- A JApplet subclass, with two buttons and a text field:

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Counter1 extends JApplet {
    private int count = 0;
    private JButton
        start = new JButton("Start"),
        onOff = new JButton("Toggle");
    private JTextField t = new JTextField(10);
    private boolean runFlag = true;
```

A Dysfunctional GUI

- Nested ActionListeners for the buttons:

```
class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        go();
    }
}
class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        runFlag = !runFlag;
    }
}
```

A Dysfunctional GUI

- An init() method to setup the widgets:

```
public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(t);
        start.addActionListener(new StartL());
        cp.add(start);
        onOff.addActionListener(new OnOffL());
        cp.add(onOff);
}
```

A Dysfunctional GUI

- A **go()** method to do something:

```
public void go() {
    while(true) {
        try {
            Thread.sleep(100); //current execution to sleep in 100 milisecond
        } catch(InterruptedException e) {
            System.out.println("Interrupted");
        }
        if (runFlag) {
            t.setText(Integer.toString(count++));
            System.out.println(Integer.toString(count));
        }
    }
}
```

A Dysfunctional GUI

- This doesn't work at all! Why?
- When the Start button is clicked,
  - The method **go()** is called.
  - **go()** contains a **while(true)** loop, so **go()** is never exited.
  - Control never gets back to the handler for the Toggle button.

A Better GUI With a Thread

- Put the CPU-intensive activity inside a separate thread.
- The GUI has its own thread, and control comes back to it from time to time (as determined by the Java runtime system).
- Users interact with the GUI, and it responds in what seems like real time.
- (Demo Counter2 now)

A Better GUI With a Thread

```
public class Counter2 extends JApplet {
   private class SeparateSubTask extends Thread {
      private int count = 0;
      private boolean runFlag = true;
      SeparateSubTask() { start(); }
      void invertFlag() { runFlag = !runFlag; }
      public void run() {//override run() of thread
         while (true) {
            try {
               sleep(100);
            } catch(InterruptedException e) {
               System.out.println("Interrupted");
            }
            if (runFlag)
               t.setText(Integer.toString(count++));
         }
      }
   }
```

A Better GUI With a Thread

- The Listeners have to change a little:

```
private SeparateSubTask sp = null;

class StartL implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      if (sp == null)
         sp = new SeparateSubTask();
   }
}
class OnOffL implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      if (sp != null)
         sp.invertFlag();
   }
}
```

A Better GUI With a Thread

- The subtask runs in its own little thread, but the larger program still listens for user mouse clicks.
- Thus mouse clicks on the Toggle button can effectively interrupt the program.

Database

Java Database Connectivity

- Most software development /Applications involves client/server operations.

Which DBMS Should We Use?

- Following its “platform independent” nature, Java support most of Database system.
- JDBC is supposed to be generic (*not* genetic!), so it supports standard SQL.
- DBMS vendors (Oracle, IBM, etc) provide a driver for each product, and these drivers allow for product-specific “customizations”.
- Insofar as possible, we should write “vanilla” Java code, so it’s portable.

The Basic Steps

- We need:

  1. a database

  2. a "**database URL**" that identifies the protocol and the database itself

  3. a **driver** for the protocol

  4. a Java **Connection object** to link our program to the database

- Once this is accomplished, we can create a **Statement** object through which we execute queries.

Example: mySQL

Reference: **Connect to Mysql Server Using VS Code**

1. Database: mySQL server

   https://dev.mysql.com/downloads/mysql/

2. URL for connection:

   jdbc:mysql://localhost:3306/myPatient

   myPatient is the name of database

3. Driver Protocol : Driver class for connectivity:

   com.mysql.cj.jdbc.Driver

4. Connection Object

```
Class.forName("com.mysql.cj.jdbc.Driver");
conn = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/myPatient", "sqluser", "password");
```

Let's try it

```
import java.sql.*;
public class mySQLConn {
   public static void main(String[] args) {
      Connection conn = null;
      try{
         Class.forName("com.mysql.cj.jdbc.Driver");
         conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/myPatient", "sqluser", "password");
         Statement sta = conn.createStatement();
         ResultSet reset = sta.executeQuery("select * from  patientRecord");
         System.out.println("reSEt"+reSet.toString());
         reSet.close();
         sta.close();
         conn.close();
      } catch (Exception e){ System.out.println(e);
      }
   }
}
```

Some Comments

- SQLExceptions may be thrown if the driver can't be found.
- The "JDBC-ODBC bridge driver" is explicitly loaded by the statement
  Class.forName("com.mysql.cj.jdbc.Driver");
  and this registers the driver with Java's driver manager.
- The driver manager is supposed to load all the right drivers, but if you don't use the **Class.forName** statement, you get exceptions.
- Ways of telling the driver manager what to load are given in the **DriverManager** documentation.

More Comments

- Once the JDBC driver is loaded, the form of the URL is determined. For our driver, and this example, it is

  "jdbc:mysql://localhost:3306/myPatient", "sqluser", "password");

  If the database were somewhere across a network, then the URL would be more complicated…
- The mySQL database require a user name or password for root access.
- Finally, the **DriverManager.getConnection** is called to get a **Connection** object, through which all the work is done.

The **Connection** Interface
java.sql

- Representative methods include:
  - Statement createStatement() // executing a static SQL statement and returning the results

Example:

```
Statement sta = conn.createStatement();
ResultSet reSet = sta.executeQuery("select * from patientRecord");
```

The **Connection** Interface java.sql

- Representative methods include:
  - void close()

Example:

reSet.close(); //Releases this ResultSet object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed

sta.close(); //Releases this Connection object's database and JDBC resources immediately instead of waiting for them to be automatically released.

conn.close(); Releases this Connection object's database and JDBC resources immediately instead of waiting for them to be automatically released.

The **Connection** Interface
java.sql

- Representative methods include:
  - DatabaseMetaData getMetaData()

  Comprehensive information about the database as a whole.

  Ref:
  https://docs.oracle.com/javase/8/docs/api/java/sql/DatabaseMetaData.html

The **Connection** Interface
java.sql

- Representative methods include:
  - Statement createStatement() // executing a static SQL statement and returning the results
  - DatabaseMetaData getCatalog()
  - DatabaseMetaData getMetaData()
  - void rollback()
  - void commit()
  - void close()
- Danger! **commit()** is automatic after each **Statement** is executed, unless you explicitly disable it (with **setAutoCommit(false)**).

The **Connection** Interface java.sql.DatabaseMetaData

- Representative methods include:
  - ResultSet getCatalog();

Retrieves the catalog names available in this database.

Ref:

https://docs.oracle.com/javase/8/docs/api/java/sql/DatabaseMetaData.html

DatabaseMetaData

```
ResultSetMetaData md = conn.getMetaData();
System.out.println(md.getNumericFunctions());
```

produces

ABS,ATAN,CEILING,COS,EXP,FLOOR,LOG,MOD,POWER,
RAND,ROUND,SIGN,SIN,SQRT,TAN

The **ResultSet** Interface

A table of data representing a database result set, which is usually generated by executing a statement that queries the database.

- Maintains a cursor pointing to the current row.
- **first()** move the cursor to the first row
- **next()** moves the cursor to the next row, returning true if there is a next row, false if not.
- **getXXX()** methods take either a column label or column Index.

getXXX() Methods

- getBoolean()
- getBlob()
- getByte()
- getClob() (Character Large Object)
- getDouble()
- getObject()
- etc. etc.

Scrollable **ResultSets**

```
Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
//scrollable but not sensitive to change
    ResultSet.CONCUR_UPDATABLE);
//concurrency mode for resultSet object, updatable
    ResultSet rs = stmt.executeQuery("SELECT a, b FROM TABLE2");

// rs will be scrollable, will not show changes made by others,
// and will be updatable
```

Updating Rows

```
rs.absolute(5); // moves the cursor to the given row (5th) of rs
rs.updateString("NAME", "AINSWORTH");
// updates the NAME column of row 5 to be AINSWORTH
rs.updateRow(); // updates the row in the data source

rs.moveToInsertRow(); // moves cursor to the insert row
rs.updateString(1, "AINSWORTH"); // updates the
      // first column of the insert row to be AINSWORTH
rs.updateInt(2, 35); // updates the second column to be 35
rs.updateBoolean(3, true); // updates the third row to true
rs.insertRow();
rs.moveToCurrentRow();
```

Ref. https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html

Databases and Java's **Table**

- The **Table** class provides a "grid" interface that's perfect for showing DB tables.
- Typically we use the **TableModel** interface, which specifies (among others)
  - **getColumnCount()**
  - **getRowCount()**
  - **getValueAt(int, int)**
  - **setValueAt(Object, int, int)**

Ref.

https://docs.oracle.com/javase/8/docs/api/javax/swing/table/TableModel.html

AbstractTableModel

- Just like listener adapters, there is a “filled in” version of TableModel: AbstractTableModel (abstract class)
- AbstractTableModel is the Abstract Class. Thus, use it, 3 methods need to implement:
  - getRowCount()
  - getColumnCount()
  - getValueAt(int, int)

Ref.
https://docs.oracle.com/javase/8/docs/api/javax/swing/table/AbstractTableModel.html

Table and TableModel

- A Table can be created by calling the constructor that takes a TableModel as an argument.
- The Table then knows how to set itself up with the right number of rows and columns.
- This is remarkably easy; a great design!

Ref.

https://docs.oracle.com/javase%2Ftutorial%2Fuiswing%2F%2Fcomponents/table.html#simple

TableModel In Action

```
class DataModel extends AbstractTableModel {
    Connection c;
    Statement s;
    ResultSet r;
    int rowCount;
    public DataModel() throws SQLException, ClassNotFoundException {
        c = DriverManager.getConnection(
          "jdbc:mysql://localhost:3306/myPatient", "sqluser", "password");
      ResultSet r;
      Statement s = conn.createStatement(
        r. TYPE_SCROLL_INSENSITIVE,
        r. CONCUR_READ_ONLY);
      ResultSet r = sta.executeQuery("select * from  patientRecord");
      r.last();
      rowCount = r.getRow();
      r.first();
    }
```

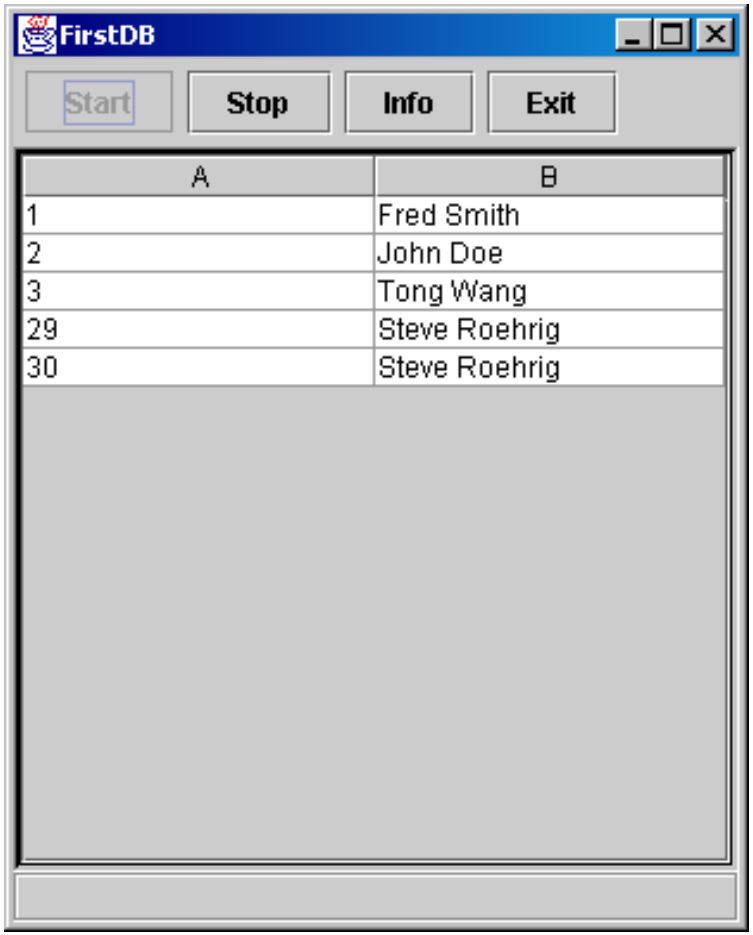
TableModel In Action

```
        public int getColumnCount() { return 2; }
        public int getRowCount() { return rowCount; }
        public Object getValueAt(int row, int col) {
            String st = null;
            try {
                r.absolute(row+1);
                st = r.getString(col+1);
            }
            catch(SQLException e){}
              return st;
        }
public boolean isCellEditable(int row, int col) {
            return false;
}
```

Build An Applet

```
public class FirstDB extends JApplet {
    public void init() {
        Container cp = getContentPane();
        JTable table = null;
        try {
            table = new JTable(new DataModel());
        }
        catch(SQLException e) {}
        catch(ClassNotFoundException e) {}
        cp.add(new JScrollPane(table));
    }
    public static void main(String[] args) {
        Console.run(new FirstDB(), 350, 200);
    }
}
```

Here's The Result



The 613th Slide

- That’s all, folks!