7. 프로그래밍 워크샵

이 장에서는 주식관리 프로그램을 통해 네트워크 프로그래밍, 멀티 스래드 프로그래밍 그리고 윈도우 프로그래밍의 개념을 익히도록 합니다.

주요 내용입니다.

-

7.1. Stock Market

7.1.1. Stock Market 개요

본 과정의 목적

- Java와 JDBC를 사용한 주식관리 프로그램 제작과정을 통하여 자바 및 JDBC 사용법과 MVC 디자인 패턴에 대한 기술들을 익힌다.

학습 목표

- 자바를 사용하여 Application을 개발한다.
- MVC(Model-View-Controller) Design Pattern, Exception Handling, JDBC등을 사용하여 Application을 개발한다.

본 과정의 버전

- JAVA 5
- JDBC Spec 4.0

설치 툴

- J2SDK 1.5.0_02
- 데이터베이스: Oracle 9i
- 편집 도구: Eclipse 3.0.2(반드시 JDK 1.4 이상이 설치되어 있어야 함)

환경변수

- PATH
- JAVA_HOME

7.1.2. Eclipse에서 프로젝트 생성 방법

7.1.2.1. 프로젝트 생성

```
[File]->[New]->[Project]
   [New Project] 창
      [Java][Java Project] 선택
      [Next]
   [Project settings] 창
      Project Name: StockMarket(원하는 이름으로 하셔도 됩니다)
      [Next]
   [Java Settings] 창
      [Source] 탭에서는 제외필터를 설정할 수 있습니다.
          [Allow output folders for source folders] 체크 안 함
      [Libraries] 탭에서(오라클 JDBC드라이버)
          [Add External JARs]버튼 누르고 ojdbc14.jar 선택 후 [열기]
      Default out folders: StockMarket
      [Finish]
   [Confirm Perspective Switch] 메시지박스 나오면 [Yes] 선택
```

프로젝트 백업 및 리스토어

```
   백업 :
      [File] -> [Export] -> Zip file
   리스토어 :
      같은 이름의 프로젝트를 생성한 후,
      [File] -> [Import] -> Zip file 선택, Into folder : /
```

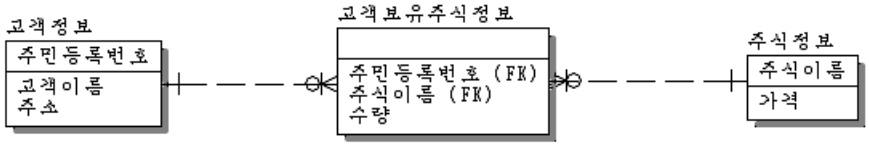
java 파일 가져오기(나중에 커넥션 풀 사용 할 때 하시면 됩니다.)

```
[File] -> [Import]
   [Select] 창
      [File system] 선택, [Next]
   [File system] 창에서
      From directory : Import할 폴더가 포함되어 있는 폴더를 선택
          Import 할 폴더 선택
      Into folder : StockMarket 선택
      [Finish]
```

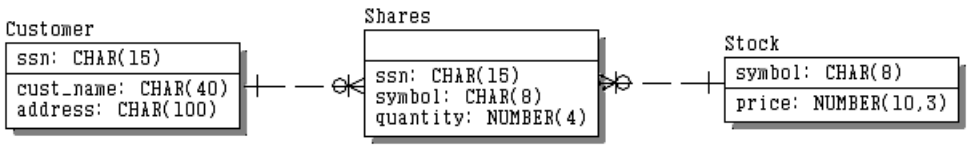
7.1.3. SotckMarket 데이터베이스 테이블

테이블 명	설 명
Customer	고객의 정보를 저장한다.
Shares	고객이 보유한 주식의 정보를 저장한다.
Stock	주식의 정보를 저장한다.

7.1.3.1. Logical Diagram



7.1.3.2. Physical Diagram



7.1.3.3. 테이블 생성 쿼리문

```
CREATE TABLE customer (
  ssn       CHAR(15)  PRIMARY KEY,
  cust_name    CHAR(40)  NOT NULL,
  address  CHAR(100)    NOT NULL );

CREATE TABLE stock (
  symbol       CHAR(8)   PRIMARY KEY,
  price        NUMBER(10, 3)    NOT NULL );

CREATE TABLE shares (
  ssn       CHAR(15)  NOT NULL,
  symbol       CHAR(8)   NOT NULL,
  quantity NUMBER(4)    NOT NULL );
```

7.1.3.4. Sample Data 입력용 쿼리문

```
INSERT INTO customer VALUES( '111-111', 'Heo', 'Seoul');
INSERT INTO customer VALUES( '111-112', 'Kim',  'Jeju');
INSERT INTO customer VALUES( '111-113', 'Lee', 'Busan');
INSERT INTO customer VALUES( '111-114', 'Hong', 'Seoul');
INSERT INTO customer VALUES( '111-115', 'Park',  'Gwangju');
INSERT INTO customer VALUES( '111-116', 'Sun', 'Incheon');
INSERT INTO customer VALUES( '111-117', 'You', 'Seoul');
INSERT INTO customer VALUES( '111-118', 'Seo', 'Seoul');

INSERT INTO stock VALUES ('SUN', 68.25);
INSERT INTO stock VALUES ('CyAs', 22.625);
INSERT INTO stock VALUES ('DUKE', 6.25);
INSERT INTO stock VALUES ('ABStk', 18.5);
INSERT INTO stock VALUES ('JSVCo', 9.125);
INSERT INTO stock VALUES ('TMAs', 82.375);
INSERT INTO stock VALUES ('BWInc', 11.375);
INSERT INTO stock VALUES ('GMEnt', 44.625);
INSERT INTO stock VALUES ('PMLtd', 203.375);
INSERT INTO stock VALUES ('JDK', 33.5);

COMMIT;
```

7.2. 사용된 패턴들

7.2.1. Design Pattern에 대한 정의

디자인 패턴(Design Pattern)은 좋은 객체지향 소프트웨어를 구현할 수 있도록 전문가의 설계 경험들을 정립해서 소프트웨어 개발의 기준으로 만들어 놓은 개념이나 철학(Paradigm)을 의미한다.

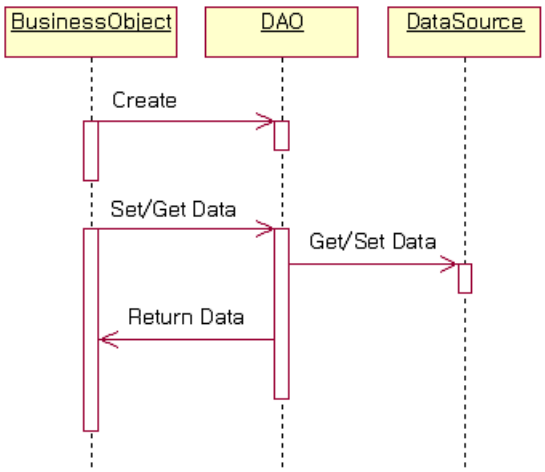
7.2.2. MVC(Model-View-Controller)패턴

7.2.3. Command 패턴(Behavioral Patterns)

Command 패턴은 인터페이스를 이용해서 내부에서 실행되는 객체가 무언지 모르는 상태에서 동일한 과정으로 메서드를 수행하고 싶을 때 사용하는 패턴이다. 다시 말하면, Command 패턴은 실행하고 싶은 어떤 작업을 메서드를 호출하는 처리로서 표현하는 것이 아니라, 명령을 나타내는 클래스의 인스턴스로 표현할 수 있도록 해준다. 명령을 클래스로 표현하면 명령의 집합을 보존해 둘 수 있다. 그러므로 똑같은 명령을 재실행 할 수도 있고, 여러 개의 명령을 모은 것을 새로운 명령으로서 재사용 할 수도 있다.

7.2.4. Data Access Object(DAO) 패턴

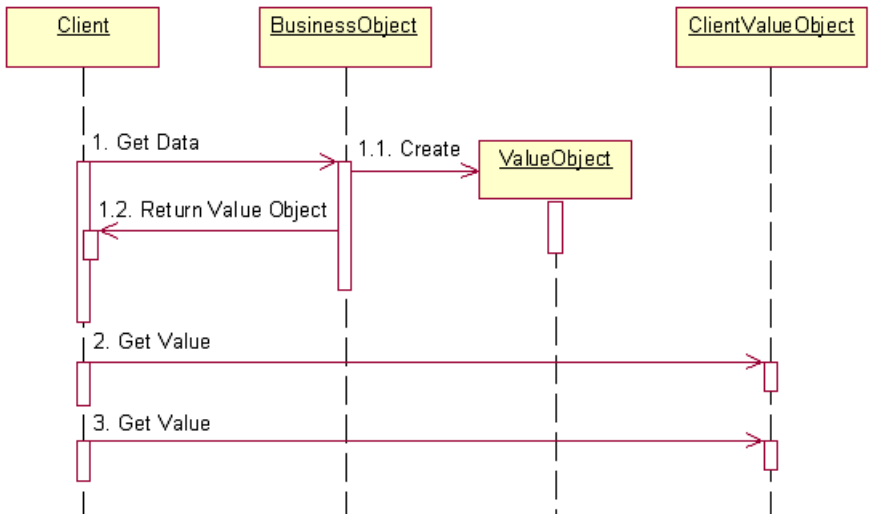
Data Access Object는 데이터 소스에 접근하는 모든 부분을 추상화시키거나 은폐시킨다. 그리고, Data Access Object는 데이터를 가져오거나 저장하기 위해 데이터 소스와 커넥션(Connection)을 관리한다.



이 코스에서는 Database.java 파일이 이에 속하게 된다.

7.2.5. Value Object(VO) 패턴

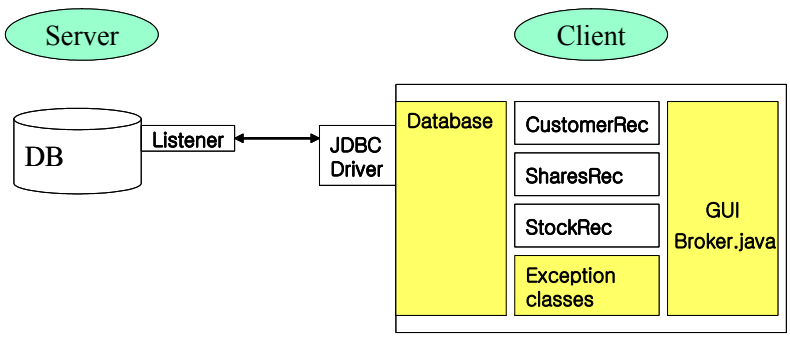
Value Object는 비즈니스 데이터(business data)를 은닉화(Encapsulate)시킨다. 한번의 메서드 호출로 value object를 보내거나 가져올 수 있다. 클라이언트가 비즈니스 데이터를 얻으려고 Business Object를 요청했을 때, Business Object는 value object를 생성하고 attribute value에 적용시키고 나서 클라이언트에게 value를 전달해 준다.



이 코스에서는 CustomerVO.java, SharesVO.java, StockVO.java파일이 이에 속하게 된다.

7.3. 2-Tier application

7.3.1. 2-Tier 애플리케이션



StockMarket Application 디렉토리 구조

TestDatabase.java(테스트파일) Broker.java(클라이언트) ProtocolHandler(미들서버)			
broker	명령과 관련된 클래스		
broker	database	Value Object 와 Data Access Object 클래스	
broker	database	exception	예외 클래스
broker	database	connpool	Connection Pool 클래스

7.3.2. 이미 작성되거나 배치(deploy) 되어 있어야 하는 파일들

역 할	파 일 명	설 명	비 고
Value Object	broker.database.CustomerVO.java broker.database.SharesVO.java broker.database.StockVO.java	데이터베이스 테이블을 나타내는 클래스	파일제공
예외 클래스	broker.database.exception.DuplicateIDException.java broker.database.exception.RecordNotFoundException.java broker.database.exception.InvalidTransactionException.java	예외 클래스	파일제공

7.3.3. 작성해야 하는 파일들

역 할	파 일 명	설 명	비 고
DAO	broker.database.Database.java	Data Access Object	

* 제공한 파일들을 가져오기 하세요.
실행 후 화면에는 고객의 정보와 주식의 정보가 나타나야 하며, [Apply]버튼을 눌러 [111-121 GilDong 서울] 정보가 추가되도록 하여야 한다.

7.3.4. [Java응용프로그램-TestDatabase] 실행

7.3.5. Value Object 클래스

7.3.5.1. CustomerVO

broker.database.CustomerVO.java

```
1:  package broker.database;
2:
3:  import java.io.Serializable;
4:  import java.util.*;
5:
6:  public class CustomerVO implements Serializable {
7:
8:     private String ssn;
9:     private String name;
10:    private String addr;
11:    private Vector portfolio;
12:
13:    // Constructors
14:    public CustomerVO (String ssn, String name, String addr,
15:       Vector portfolio) {
16:       this.ssn = ssn;
17:       this.name = name;
18:       this.addr = addr;
19:       this.portfolio = portfolio;
20:    }
21:
22:    public CustomerVO (String ssn, String name, String addr) {
23:       this (ssn, name, addr, null);
24:    }
25:
26:    public CustomerVO (String ssn) {
27:       this (ssn, null, null, null);
28:    }
29:
30:    public CustomerVO () {
31:       this (null, null, null, null);
32:    }
33:
34:
35:
36:
37:    // Accessor methods
38:
39:    public String getSSN () {
40:       return ssn;
41:    }
42:
43:    public String getName () {
```

```
44:        return name;
45:     }
46:
47:     public String getAddr () {
48:        return addr;
49:     }
50:
51:     // Get and return the portfolio for this customer
52:     // This method will return the Vector object that contains
53:     // the portfolio
54:     public Vector getPortfolio () {
55:        return portfolio;
56:     }
57:
58:     // Mutator methods - note you cannot change the ssn
59:
60:     public void setName (String newName) {
61:        name = newName;
62:     }
63:
64:     public void setAddr (String newAddr) {
65:        addr = newAddr;
66:     }
67:
68:     public void setPortfolio (Vector newPortfolio) {
69:        portfolio = newPortfolio;
70:     }
71:
72: }//end class
```

7.3.5.2. SharesVO

broker.database.SharesVO.java

```
1:  package broker.database;
2:  import java.io.Serializable;
3:
4:  // The SharesVO class
5:  // Stores a single instance of a Shares record
6:  //
7:  // These are dynamic objects that belong to customers
8:  public class SharesVO implements Serializable {
9:
10:    private String ssn;
11:    private String symbol;
12:    private int quantity;
13:
14:    // SharesVO constructor
```

```
15:    public SharesVO (String ssn, String symbol, int quantity) {
16:       this.ssn = ssn;
17:       this.symbol = symbol;
18:       this.quantity = quantity;
19:    }
20:
21:    public SharesVO (String ssn) {
22:       this (ssn, "", 0);
23:    }
24:
25:    public SharesVO () {
26:       this ("", "", 0);
27:    }
28:
29:
30:    // Accessor methods
31:    public String getSSN () {
32:       return ssn;
33:    }
34:
35:    public String getSymbol () {
36:       return symbol;
37:    }
38:
39:    public int getQuantity () {
40:       return quantity;
41:    }
42:
43:    // Mutator methods - note no setSSN method
44:    public void setSymbol (String newSymbol) {
45:       symbol = newSymbol;
46:    }
47:
48:    public void setQuantity (int newQuantity) {
49:       quantity = newQuantity;
50:    }
51:
52: }//end class
```

7.3.5.3. StockVO

broker.database.StockVO.java

```
1:  package broker.database;
2:  import java.io.Serializable;
3:
4:  // The StockVO class
5:  // Keeps a single record instance of a stock object
```

```
6:  //
7:  // These are created from a dynamic query against the DB
8:  public class StockVO implements Serializable {
9:
10:    private String symbol;
11:    private float price;
12:
13:    // StockVO constructor - create a instance of a stock
14:    // from the queried data
15:    public StockVO (String symbol, float price) {
16:       this.symbol = symbol;
17:       this.price = price;
18:    }
19:
20:    // StockVO constructor - create a instance of a stock
21:    // with no data
22:    public StockVO () {
23:       symbol = "";
24:       price = 0.0f;
25:    }
26:
27:    // Methods to return the private values of this object
28:    public float getPrice () {
29:       return price;
30:    }
31:
32:    public String getSymbol () {
33:       return symbol;
34:    }
35:
36:    public void setPrice (float newPrice) {
37:       price = newPrice;
38:    }
39:
40:    public void printPrice () {
41:       System.out.print(price);
42:    }
43:
44:    public void printStock () {
45:       System.out.print(symbol);
46:    }
47:
48: }//end class
```

7.3.6. 예외 클래스

7.3.6.1. DuplicateIDException

broker.database.exception.DuplicateIDException.java

```
1:  package broker.database.exception;
2:
3:  public class DuplicateIDException extends Exception{
4:     public DuplicateIDException(String message){
5:       super(message);
6:     }
7:     public DuplicateIDException(){
8:       this("DuplicateIDException is occured...");
9:     }
10: }//end class
```

7.3.6.2. InvalidTransactionIDException

broker.database.exception.InvalidTransactionException.java

```
1:  package broker.database.exception;
2:
3:  public class InvalidTransactionException extends Exception{
4:     public InvalidTransactionException(String message){
5:       super(message);
6:     }
7:     public InvalidTransactionException(){
8:       this("InvalidTransactionException is occured...");
9:     }
10: }//end class
```

7.3.6.3. RecordNotFoundException

broker.database.exception.RecordNotFoundException.java

```
1:  package broker.database.exception;
2:
3:  public class RecordNotFoundException extends Exception {
4:     public RecordNotFoundException(String message) {
5:        super(message);
6:     }
7:     public RecordNotFoundException() {
8:        this("RecordNotFoundException is occured...");
9:     }
10: }//end class
```

7.3.7. Database 애플리케이션 메인

7.3.7.1. broker.database.Database.java

※ 다음 소스코드를 완성하세요.

broker.database.Database.java

```
1:  package broker.database;
2:
3:  import java.sql.Connection;
4:  import java.sql.DriverManager;
5:  import java.sql.ResultSet;
6:  import java.sql.SQLException;
7:  import java.sql.Statement;
8:  import java.util.Vector;
9:
10: import broker.database.connpool.ConnectionPool;
11: import broker.database.exception.DuplicateIDException;
12: import broker.database.exception.InvalidTransactionException;
13: import broker.database.exception.RecordNotFoundException;
14:
15: public class Database{
16:
17: /* mySql인 경우
18:    private static final String DRIVER = "org.gjt.mm.mysql.Driver";
19:    private static final String URL = "jdbc:mysql://127.0.0.1:3306/mydb";
20:    //sl285?useUnicode=true&characterEncoding=euc-kr";
21:    private static final String USER_ID = "scott";
22:    private static final String PASSWORD = "tiger";
23: */
24:
25:
26:    private static final String DRIVER = "oracle.jdbc.driver.OracleDriver";
27: // private static final String URL = "jdbc:oracle:thin:@127.0.0.1:1521:orcl";
28:    private static final String USER_ID = "scott";
29:    private static final String PASSWORD = "tiger";
30:
31:    private Connection con;
32:    private Statement stmt;
33:    private ResultSet rset;
34:
35:    // Driver loading, get Connection, create Statement
36:    public Database(String server) throws ClassNotFoundException, SQLException
    {
37:       Class.forName(DRIVER);
38:       System.out.println("Driver is loaded.");
39:       String url = "jdbc:oracle:thin:@"+server+":1521:orcl";
40:       con = DriverManager.getConnection(url, USER_ID, PASSWORD);
41:       System.out.println("Connection is setting.");
42:       stmt = con.createStatement();
43:       System.out.println("Statement is created.");
44:    }
45:
```

```
46:    public Database() throws ClassNotFoundException, SQLException {
47:       this("127.0.0.1");
48:    }
49:
50:    public void close() throws SQLException{
51:          stmt.close();
52:          con.close();
53:    }
54:
55:    private String makeSQL(String sql){
56:       return "'"+sql+"'";
57:    }
58:
59:    private boolean ssnExist(String ssn){
60:       String query="SELECT * FROM CUSTOMER WHERE SSN="+makeSQL(ssn);
61:       boolean isExist = false;
62:       try{
63:           rset=stmt.executeQuery(query);
64:          isExist = rset.next();
65:       }catch(SQLException sqle){
66:       }
67:       return isExist;
68:    }
69:
70:    public void addCustomer(String name, String ssn, String address) throws
    DuplicateIDException {
71:       if(ssnExist(ssn)) throw new DuplicateIDException();
72:
73:       String query="INSERT INTO CUSTOMER(SSN,CUST_NAME,ADDRESS) VALUES ("+
74:          makeSQL(ssn)+","+makeSQL(name)+","+makeSQL(address)+")";
75:       try{
76:           stmt.executeUpdate(query);
77:       }catch(SQLException sqle){}
78:    }
79:
80:    public void deleteCustomer(String ssn) throws RecordNotFoundException {
81:       if(!ssnExist(ssn)) throw new RecordNotFoundException();
82:
83:       String query1="DELETE FROM SHARES WHERE SSN="+makeSQL(ssn);
84:       String query2="DELETE FROM CUSTOMER WHERE SSN="+makeSQL(ssn);
85:       try{
86:          stmt.executeUpdate(query1);
87:          stmt.executeUpdate(query2);
88:       }catch(SQLException sqle){}
89:    }
90:
91:    public void updateCustomer(String name, String ssn, String address) throws
    RecordNotFoundException {
92:       if(!ssnExist(ssn)) throw new RecordNotFoundException();
93:
```

```
94:       String query="UPDATE CUSTOMER SET CUST_NAME="+makeSQL(name)+
95:          ",ADDRESS="+makeSQL(address)+" WHERE SSN="+makeSQL(ssn);
96:       try{
97:           stmt.executeUpdate(query);
98:
99:       }catch(SQLException sqle){}
100:   }
101:
102:   public CustomerVO getCustomer(String ssn) throws RecordNotFoundException {
103:      if(!ssnExist(ssn)) throw new RecordNotFoundException();
104:
105:      String query="SELECT * FROM CUSTOMER WHERE SSN="+makeSQL(ssn);
106:      CustomerVO cust=null;
107:      String name,address;
108:      try{
109:         rset=stmt.executeQuery(query);
110:         rset.next();
111:         name=rset.getString("CUST_NAME").trim();
112:         ssn=rset.getString("SSN").trim();
113:         address=rset.getString("ADDRESS").trim();
114:
115:         cust=new CustomerVO(ssn,name,address);
116:         cust.setPortfolio(getPortfolio(ssn));
117:      }catch(SQLException sqle){}
118:
119:      return cust;
120:   }
121:
122:   public CustomerVO[] getAllCustomers() {
123:      String query1="SELECT COUNT(*) FROM CUSTOMER";
124:      String query2="SELECT * FROM CUSTOMER";
125:      CustomerVO[] custAll=null;
126:      String name, ssn, address;
127:      SharesVO sr;
128:      int count=0;
129:      try{
130:         rset=stmt.executeQuery(query1);
131:         rset.next();
132:         count=rset.getInt(1);
133:         custAll=new CustomerVO[count];
134:          rset=stmt.executeQuery(query2);
135:
136:          for(int i=0;rset.next();i++){
137:              name=rset.getString("CUST_NAME").trim();
138:             ssn=rset.getString("SSN").trim();
139:             address=rset.getString("ADDRESS").trim();
140:            custAll[i]=new CustomerVO(ssn, name, address);
141:         }
142:         for(int i=0;i<custAll.length;i++){
143:            ssn=custAll[i].getSSN();
```

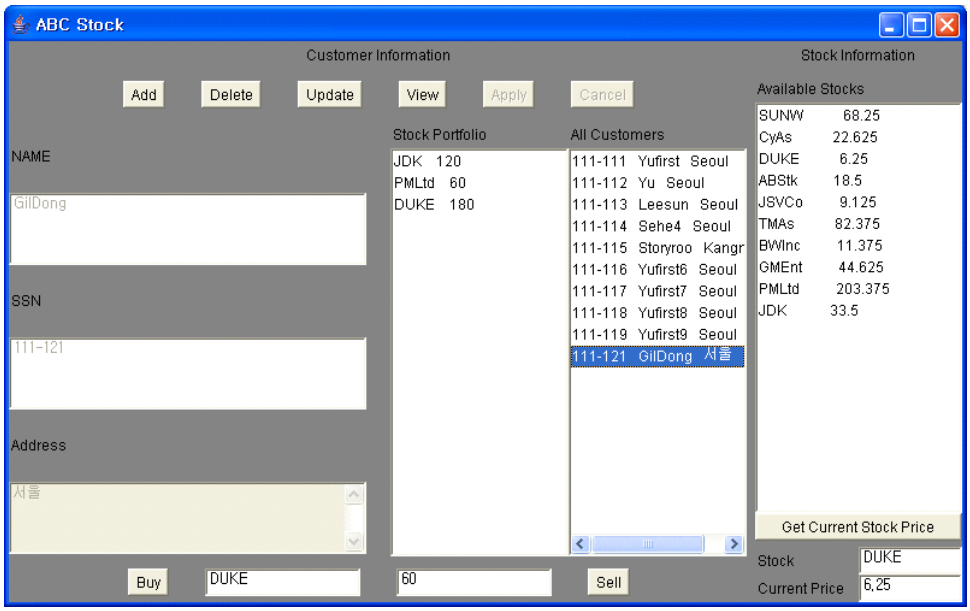
```
144:             try{
145:                custAll[i].setPortfolio(getPortfolio(ssn));
146:             }catch(RecordNotFoundException re){}
147:          }
148:       }catch(SQLException sqle){}
149:
150:       return custAll;
151:    }
152:
153:    public Vector getPortfolio(String ssn) throws RecordNotFoundException {
154:       if(!ssnExist(ssn)) throw new RecordNotFoundException();
155:
156:       String query="SELECT * FROM SHARES WHERE SSN="+makeSQL(ssn);
157:       Vector v=new Vector(1,1);
158:       SharesVO sr;
159:       String symbol;
160:       int quantity;
161:       try{
162:          rset=stmt.executeQuery(query);
163:          while(rset.next()){
164:             symbol=rset.getString("SYMBOL").trim();
165:             quantity=rset.getInt("QUANTITY");
166:             sr=new SharesVO(ssn,symbol,quantity);
167:             v.addElement(sr);
168:          }
169:       }catch(SQLException sqle){}
170:       return v;
171:    }
172:
173:    public StockVO[] getAllStocks() {
174:       String query1="SELECT COUNT(*) FROM STOCK";
175:       String query2="SELECT * FROM STOCK";
176:       int count=0;
177:       StockVO[] stocks=null;
178:       String symbol;
179:       float price=-1.0F;
180:       try{
181:          rset=stmt.executeQuery(query1);
182:          rset.next();
183:          count=rset.getInt(1);
184:          stocks=new StockVO[count];
185:          rset=stmt.executeQuery(query2);
186:          for(int i=0;rset.next();i++){
187:             symbol=rset.getString("SYMBOL");
188:             price=rset.getFloat("PRICE");
189:             stocks[i]=new StockVO(symbol,price);
190:          }
191:       }catch(SQLException sqle){}
192:       return stocks;
193:    }
194:
```

```
195:   public void sellShares(String ssn, String symbol, int quantity)
196:            throws RecordNotFoundException, InvalidTransactionException {
197:      if(!ssnExist(ssn)) throw new RecordNotFoundException();
198:
199:      String query="SELECT QUANTITY FROM SHARES WHERE SSN="+makeSQL(ssn)
200:         +" AND SYMBOL="+makeSQL(symbol);
201:      int realQuantity=0,gap=0;
202:      try{
203:         rset=stmt.executeQuery(query);
204:         if(!rset.next()) throw new InvalidTransactionException();
205:
206:         realQuantity=rset.getInt(1);
207:         gap=realQuantity-quantity;
208:         if(gap>0){
209:            query="UPDATE SHARES SET QUANTITY=" + gap + " WHERE SSN=" +
    makeSQL(ssn) + " AND SYMBOL="+makeSQL(symbol);
210:             stmt.executeUpdate(query);
211:         }else if(gap==0){
212:            query="DELETE FROM SHARES WHERE SSN=" + makeSQL(ssn) + " AND
    SYMBOL=" + makeSQL(symbol);
213:            stmt.executeUpdate(query);
214:         }else{
215:            throw new InvalidTransactionException();
216:         }
217:      }catch(SQLException sqle){
218:         System.out.println("sellShares() method failed...");
219:      }
220:   }
221:
222:   public void buyShares(String ssn, String symbol, int quantity) throws
    RecordNotFoundException {
223:      if(!ssnExist(ssn)) throw new RecordNotFoundException();
224:
225:      String query="SELECT QUANTITY FROM SHARES WHERE SSN="+makeSQL(ssn)+
226:         " AND SYMBOL="+makeSQL(symbol);
227:      int realQuantity=0;
228:      try{
229:         rset=stmt.executeQuery(query);
230:         if(rset.next()){
231:            realQuantity=rset.getInt(1);
232:            query="UPDATE SHARES SET QUANTITY="+(quantity+realQuantity)+ "
    WHERE SSN=" + makeSQL(ssn)+" AND SYMBOL="+makeSQL(symbol);
233:            stmt.executeUpdate(query);
234:         }else{
235:            query="INSERT INTO SHARES(SSN,SYMBOL,QUANTITY) VALUES("+
236:               makeSQL(ssn)+","+makeSQL(symbol)+","+quantity+")";
237:            stmt.executeUpdate(query);
238:         }
239:      }catch(SQLException sqle){}
240:   }
241:
```

```
242:   public void updateStockPrice(String symbol,float price) {
243:      String query="UPDATE STOCK SET PRICE="+price+" WHERE SYMBOL=" +
    makeSQL(symbol);
244:      try{
245:            stmt.executeUpdate(query);
246:      }catch(SQLException sqle){}
247:   }
248:
249:   public float getStockPrice(String symbol){
250:      String query="SELECT PRICE FROM STOCK WHERE SYMBOL="+makeSQL(symbol);
251:      try{
252:         rset=stmt.executeQuery(query);
253:         rset.next();
254:         return rset.getFloat(1);
255:      }catch(SQLException sqle){
256:         return -1.0F;
257:      }
258:   }
259:
260:}//end class
```

7.4. GUI 만들기

다음 화면과 같이 되도록 GUI를 작성하세요.



◎ 작성해야 하는 파일들

역 할	파 일 명	설 명	비 고
Main 클래스	Broker.java	GUI를 구현한 메인 클래스	

◎ Broker.java

```
1:  import java.awt.*;
2:  import java.awt.event.*;
3:  import java.util.*;
4:  import java.sql.*;
5:  import java.awt.List;
6:
7:  import broker.database.CustomerVO;
8:  import broker.database.Database;
9:  import broker.database.SharesVO;
10: import broker.database.StockVO;
11: import broker.database.exception.DuplicateIDException;
12: import broker.database.exception.InvalidTransactionException;
13: import broker.database.exception.RecordNotFoundException;
14: /**
15:     <PRE>
16:     * database.class에서 가져온 결과를 GUI에 뿌려주는 기능을 담당하는 class.
17:
18:     * <B>-- 초기 화면 구성 --</B>
19:
20:     * 1)event 처리를 포함한 GUI setting : addListener() method call
21:     * 2)customer list를 보여줌 : showList(CustomerVO,List) method call
22:     * 3)stock list를 보여줌 : showList(StockVO,List) method call
23:     * 4)< button의  초기 활성화 상태 >
24:     *   add,update,delete,buy,sell : 활성화 되어야 함
25:     *   apply,cancel : 비활성화되어야 함
26:     * 5)< textfield와 textarea의 초기 편집상태 >
27:     *   sell Textfield를 제외한 모든 textfield와 textarea는 비편집상태이어야함
28:     *    name, ssn, address를 제외한 다른 textfield는 초기상태를 그대로 유지한다.
29:
30:
31:     * <B>-- 구현되어야 할 기능 --</B>
32:
33:     * 1)customer list 항목 하나를 click하면 port list에 보유주식이 나타나고
34:     *     name,ssn,addr 텍스트필드에 해당항목 정보를 보여줄 것
35:     * 2)stock list 항목 하나를 click하면 buy,stock,price 텍스트필드에 해당정보를
36:     *   보여주고 해당 주식을 살수(buy)있어야 함
37:     * 3)port list 항목 하나를 click 하면 buy,sell 텍스트필드에 해당정보를 보여주고
38:     *   주식을 팔거나(sell) 살수(buy)있어야 함
39:     * 4)add button : 새로운 고객을 customer table에 입력
40:     *              - button(add,delete,update)의 비활성화
41:     *              - button(apply,cancel)의 활성화
42:     *              - textfield(name,ssn,addr)의 편집상태
43:     *         apply button 으로 실행
44:     * 5)delete button : 해당 ssn고객을 customer table에서 지움
45:     *
46:     * 6)update button : 특정 고객정보를 변경함
47:                       - button(add,delete,update,buy,sell)의 비활성화
48:     *                 - button(apply,cancel)의 활성화
49:     *                 - textfield(name,ssn,addr)의 편집상태
```

```
50:     *         apply button 으로실행
51:     * 7)buy,sell button : 주식을 사고 팜
52:     * 8)apply button : 명령을 실행하는 버튼
53:     * 9)stock list는 대략 15초마다 DB Stock table에서 가져온 새로운 정보로 경신되어야함
54:     </PRE>
55:     */
56: public class Broker implements Runnable {
57:    private Framef = new Frame("ABC Stock");
58:    private PanelmainPanel = new Panel(),
59:                    custInfoPanel = new Panel(),
60:                           centerPanel = new Panel(),
61:                                   buttonPanel = new Panel(),
62:                                   cPanel = new Panel(),
63:                                           panelC1 = new Panel(),
64:                                           panelC2 = new Panel(),
65:                                           panelC3 = new Panel(),
66:                                           panelC4 = new Panel(),
67:                           buySellPanel = new Panel(),
68:                    stockInfoPanel = new Panel(),
69:                           availStockPanel = new Panel(),
70:                           stockPricePanel = new Panel();
71:
72:    private LabelcustInfoLabel = new Label("Customer Information",
    Label.CENTER),
73:                 stockInfoLabel = new Label("Stock Information", Label.CENTER);
74:
75:    private Button addButton = new Button("Add"),
76:                 deleteButton = new Button("Delete"),
77:                 updateButton = new Button("Update"),
78:                 viewButton = new Button("View"),
79:                 applyButton = new Button("Apply"),
80:                 cancelButton= new Button("Cancel");
81:
82:
83:    private LabelnameLabel = new Label("NAME"),
84:                 ssnLabel = new Label("SSN"),
85:                 addressLabel= new Label("Address");
86:
87:    private TextField nameField = new TextField(15),
88:                      ssnField  = new TextField(15);
89:    private TextArea addressField = new TextArea(3, 15);
90:
91:    private LabelsharesLabel = new Label("Stock Portfolio"),
92:                 custLabel = new Label("All Customers");
93:    private List  sharesList = new List(),
94:                 custList = new List();
95:
96:    private Button buyButton = new Button("Buy"),
97:                 sellButton = new Button("Sell");
98:
```

```
99:    private TextField symbolField = new TextField("", 15);
100:   private TextField quantityField = new TextField("", 15);
101:
102:   private LabelstockLabel = new Label("Available Stocks");
103:   private List stockList = new List();
104:   private Button   getPriceButton = new Button("Get Current Stock Price");
105:
106:   private LabelstockNameLabel = new Label("Stock"),
107:                priceLabel = new Label("Current Price");
108:   private TextField stockNameField = new TextField(),
109:                   priceField = new TextField();
110:
111:
112:   private boolean mode=true;
113:
114:
115:   private int flag=0;   //insert--1,delete--2,update--3,cancel--0
116:
117:   Database db;
118:   CustomerVO[] cust;
119:   StockVO[] stock;
120:   Vector portfolio;
121:
122:   //@@Constructor
123:   public Broker(){
124:      //데이터베이스 연결
125:      try{
126:         db=new Database("127.0.0.1");
127:      } catch(SQLException sqle){
128:         System.out.println("Broker Constructor : " + sqle);
129:      } catch(ClassNotFoundException cnfe){
130:         System.out.println("Broker Constructor : " + cnfe);
131:      }
132:      System.out.println("Database Conncted");
133:
134:      cust=db.getAllCustomers();
135:      stock=db.getAllStocks();
136:      //createGui() - applet ver에 추가
137:      //addListener     "
138:   }
139:
140:   //@@User Interface 구성
141:   public void createGui(){
142:      f = new Frame("ABC Stock");
143:      f.setBackground(Color.gray);
144:      f.setLayout(new BorderLayout(5, 5));
145:
146:      f.add(custInfoPanel, BorderLayout.CENTER);
147:         custInfoPanel.setLayout(new BorderLayout(5,5));
148:         custInfoPanel.add(custInfoLabel, BorderLayout.NORTH);
```

```
149:          custInfoPanel.add(centerPanel, BorderLayout.CENTER);
150:             centerPanel.setLayout(new BorderLayout(5,5));
151:             centerPanel.add(buttonPanel, BorderLayout.NORTH);
152:                buttonPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 30,
    5));
153:                buttonPanel.add(addButton);
154:                buttonPanel.add(deleteButton);
155:                buttonPanel.add(updateButton);
156:                buttonPanel.add(viewButton);
157:                buttonPanel.add(applyButton);
158:                buttonPanel.add(cancelButton);
159:             centerPanel.add(cPanel, BorderLayout.CENTER);
160:             cPanel.setLayout(new GridLayout(1,2, 20, 20));
161:                cPanel.add(panelC1);
162:                   panelC1.setLayout(new GridLayout(6, 1));
163:                   panelC1.add(nameLabel);
164:                   panelC1.add(nameField);
165:                   panelC1.add(ssnLabel);
166:                   panelC1.add(ssnField);
167:                   panelC1.add(addressLabel);
168:                   panelC1.add(addressField);
169:                cPanel.add(panelC2);
170:                   panelC2.setLayout(new GridLayout(1,2));
171:                   panelC2.add(panelC3);
172:                         panelC3.setLayout(new BorderLayout());
173:                         panelC3.add(sharesLabel, BorderLayout.NORTH);
174:                         panelC3.add(sharesList, BorderLayout.CENTER);
175:                   panelC2.add(panelC4);
176:                         panelC4.setLayout(new BorderLayout());
177:                         panelC4.add(custLabel, BorderLayout.NORTH);
178:                         panelC4.add(custList, BorderLayout.CENTER);
179:
180:          custInfoPanel.add(buySellPanel, BorderLayout.SOUTH);
181:             buySellPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 30, 5));
182:             buySellPanel.add(buyButton);
183:             buySellPanel.add(symbolField);
184:             buySellPanel.add(quantityField);
185:             buySellPanel.add(sellButton);
186:
187:       f.add(stockInfoPanel, BorderLayout.EAST);
188:          stockInfoPanel.setLayout(new BorderLayout(5,5));
189:          stockInfoPanel.add(stockInfoLabel, BorderLayout.NORTH);
190:          stockInfoPanel.add(availStockPanel, BorderLayout.CENTER);
191:             availStockPanel.setLayout(new BorderLayout());
192:             availStockPanel.add(stockLabel, BorderLayout.NORTH);
193:             availStockPanel.add(stockList, BorderLayout.CENTER);
194:             availStockPanel.add(getPriceButton, BorderLayout.SOUTH);
195:          stockInfoPanel.add(stockPricePanel, BorderLayout.SOUTH);
196:             stockPricePanel.setLayout(new GridLayout(2,2));
197:             stockPricePanel.add(stockNameLabel);
```

```
198:             stockPricePanel.add(stockNameField);
199:             stockPricePanel.add(priceLabel);
200:             stockPricePanel.add(priceField);
201:
202:       enableButton(true);
203:       enableField(false);
204:
205:       f.addWindowListener(new WindowAdapter() {
206:          public void windowClosing(WindowEvent e) {
207:             System.exit(0);
208:          }
209:       });
210:       f.setSize(800,500);
211:       f.setLocation(100,100);
212:       f.setVisible(true);
213:    }//end createGui(){
214:
215:    public void addListener(){
216:       ButtonHandler bh = new ButtonHandler();
217:
218:       addButton.addActionListener(bh);
219:       deleteButton.addActionListener(bh);
220:       updateButton.addActionListener(bh);
221:       viewButton.addActionListener(bh);
222:       applyButton.addActionListener(bh);
223:       cancelButton.addActionListener(bh);
224:
225:       buyButton.addActionListener(bh);
226:       sellButton.addActionListener(bh);
227:
228:       getPriceButton.addActionListener(bh);
229:
230:       ItemHandler ih = new ItemHandler();
231:       sharesList.addItemListener(ih);
232:       custList.addItemListener(ih);
233:       stockList.addItemListener(ih);
234:
235:    }//end addListener()
236:
237:    //@@util method ---    1.enableButton()
238:    //               2.emptyText()
239:    //               3.enableField()
240:
241:    /**
242:    <PRE>
243:     * button 활성화에 대한 method임
244:     * 이 메서드는 1번의 3개가 활성화 되면 2번의 2개가 비활성화 되고,
245:     * 2번의 2개가 활성화 되면 1번의 3개가 비활성화 됨
246:     * < 초기 상태 >
247:     * 1)add,delete,update : 활성화
```

```
248:     * 2)apply,cancel : 비활성화
249:     </PRE>
250:     */
251:    public void enableButton(boolean b){
252:       addButton.setEnabled(b);
253:       deleteButton.setEnabled(b);
254:       updateButton.setEnabled(b);
255:       viewButton.setEnabled(b);
256:       applyButton.setEnabled(!b);
257:       cancelButton.setEnabled(!b);
258:    }//end enableButton()
259:
260:    // ssn,name,address의 TextField 값을 ""(blank string)으로clear시킨다.
261:    public void emptyText(){
262:       nameField.setText("");
263:       ssnField.setText("");
264:       addressField.setText("");
265:    }//end emptyText()
266:
267:    /**
268:    <PRE>
269:     * ssn,name,address textfield의 편집/비편집 상태 결정하는 메서드
270:     * true : 모두 편집상태
271:     * false : 모두 비편집상태
272:     </PRE>
273:     */
274:    public void enableField(boolean b){
275:       nameField.setEnabled(b);
276:       ssnField.setEnabled(b);
277:       addressField.setEnabled(b);
278:    }//end enableField()
279:
280:    //@@Thread --- run() method
281:    /**
282:     * while문 사용
283:     * 일정 시간 간격으로 Stock정보와 Customer정보를 뿌려주게 한다.
284:     * sleep()메서드 이용
285:     */
286:    public void run(){
287:       while(true){
288:          stock=db.getAllStocks();
289:          viewStock();
290:          viewCustomer();
291:          try{
292:             Thread.sleep(15*1000);
293:          }catch(InterruptedException e){}
294:       }
295:    }//end run()
296:
297:
```

```
298:   //@@Application --- main() method (applet시 init()으로)
299:   public static void main(String args[]){
300:      Broker b=new Broker();
301:      b.createGui();
302:      b.addListener();
303:      new Thread(b).start();
304:   }//end main()
305:
306:   //버튼들의 이벤트 처리를 담당하는 내부 클래스임
307:   private class ButtonHandler implements ActionListener {
308:     //@@actionListener --- actionPerformed() method
309:      public void actionPerformed(ActionEvent e){
310:         if(e.getSource()==addButton){
311:            flag=1;
312:            enableField(true);
313:            enableButton(false);
314:         }else if(e.getSource()==deleteButton){
315:            flag=2;
316:            enableField(true);
317:            enableButton(false);
318:         }else if(e.getSource()==updateButton){
319:            flag=3;
320:            enableField(true);
321:            enableButton(false);
322:         }else if(e.getSource()==applyButton){
323:            if(flag==1){
324:               recordInsert();
325:            }else if(flag==2){
326:               recordDelete();
327:            }else if(flag==3){
328:               recordUpdate();
329:            }
330:         }else if(e.getSource()==viewButton){
331:            viewCustomer();
332:            viewStock();
333:         }else if(e.getSource()==cancelButton){
334:            flag=0;
335:            enableField(false);
336:            enableButton(true);
337:         }else if(e.getSource()==buyButton){
338:            int quantity = Integer.parseInt(quantityField.getText().trim());
339:            String ssn = ssnField.getText().trim();
340:            String symbol = symbolField.getText().trim();
341:            try{
342:               db.buyShares(ssn,symbol,quantity);
343:               CustomerVO cr=db.getCustomer(ssn);
344:               showList(cr.getPortfolio(),sharesList);
345:            }catch(RecordNotFoundException re){}
346:
347:         }else if(e.getSource()==sellButton){
```

```
348:             int quantity=Integer.parseInt(quantityField.getText().trim());
349:             String ssn=ssnField.getText().trim();
350:             String symbol=symbolField.getText().trim();
351:             try{
352:                db.sellShares(ssn,symbol,quantity);
353:                CustomerVO cr=db.getCustomer(ssn);
354:                showList(cr.getPortfolio(), sharesList);
355:             }catch(RecordNotFoundException re2){
356:             }catch(InvalidTransactionException ie){}
357:          }else if(e.getSource()==getPriceButton){
358:             viewStock();
359:          }
360:       }
361:    }//end ButtonHandler inner class
362:
363:    //리스트들의 이벤트 처리를 담당하는 내부 클래스임
364:    private class ItemHandler implements ItemListener {
365:       //@@itemListener --- itemStateChanged() method
366:       public void itemStateChanged(ItemEvent e){
367:          if (e.getSource()==custList) {
368:             String str=custList.getSelectedItem();
369:             StringTokenizer st=new StringTokenizer(str);
370:             String ssn=st.nextToken();
371:             try{
372:                CustomerVO cr=db.getCustomer(ssn);
373:                ssnField.setText(ssn);
374:                nameField.setText(cr.getName().trim());
375:                addressField.setText(cr.getAddr().trim());
376:                Vector v=cr.getPortfolio();
377:                showList(v,sharesList);
378:             }catch(RecordNotFoundException re){}
379:          }else if (e.getSource()==sharesList) {
380:             String str=sharesList.getSelectedItem();
381:             StringTokenizer st=new StringTokenizer(str);
382:             String symbol=st.nextToken();
383:             String quantity=st.nextToken();
384:             symbolField.setText(symbol);
385:             quantityField.setText(quantity);
386:          }else if (e.getSource()==stockList) {
387:             String str=stockList.getSelectedItem();
388:             StringTokenizer st=new StringTokenizer(str);
389:             String symbol=st.nextToken();
390:             stockNameField.setText(symbol);
391:             symbolField.setText(symbol);
392:             float price=db.getStockPrice(symbol);
393:             priceField.setText(price+"");
394:          }
395:       }
396:    }//end ItemHandler inner class
397:
```

```
398:   //###1.recordInsert()
399:   /**
400:    * 1) name, ssn, address를 TextField로부터 받아들인다.(공백 제거)
401:    * 2) 읽어들인 값이 null이면 ? 아님 addCustomer() 사용 입력
402:    * 3) Button들과 Field들 setEnaled()조정
403:    * 4) 모든 고객 읽어들여서 다시 뿌린다.
404:    */
405:   public void recordInsert(){
406:      String name=nameField.getText().trim();
407:      String ssn=ssnField.getText().trim();
408:      String address=addressField.getText().trim();
409:      if(name.equals("")||ssn.equals("")||address.equals("")){
410:         //===
411:      }else{
412:         try{
413:            db.addCustomer(name,ssn,address);
414:         }catch(DuplicateIDException e){
415:            //dialog or message
416:         }
417:      }
418:      enableButton(true);
419:      enableField(false);
420:      emptyText();
421:      cust=db.getAllCustomers();
422:      showList(cust,custList);
423:   }
424:
425:
426:   //###2.recordUpdate()
427:   /**
428:    * 1) name, ssn, address를 TextField로부터 받아들인다.(공백 제거)
429:    * 2) 읽어들인 값이 null이면 ? 아님 updateCustomer() 사용 입력
430:    * 3) Button들과 Field들 setEnaled()조정
431:    * 4) 모든 고객 읽어들여서 다시 뿌린다.
432:    */
433:   public void recordUpdate(){
434:      String name=nameField.getText().trim();
435:      String ssn=ssnField.getText().trim();
436:      String address=addressField.getText().trim();
437:      if(name.equals("")||ssn.equals("")||address.equals("")){
438:         //===
439:      }else{
440:         try{
441:            db.updateCustomer(name,ssn,address);
442:         }catch(RecordNotFoundException e){
443:            //
444:         }
445:      }
446:      enableButton(true);
447:      enableField(false);
```

```
448:       emptyText();
449:       cust=db.getAllCustomers();
450:       showList(cust,custList);
451:    }
452:
453:
454:    //###3.recordDelete()
455:    /**
456:     * 1) ssn을 TextField로부터 받아들인다. (공백 제거)
457:     * 2) 읽어들인 값이 null이면 ? 아님 deleteCustomer() 사용 입력
458:     * 3) Button들과 Field들 setEnaled()조정
459:     * 4) 모든 고객 읽어들여서 다시 뿌린다.
460:     */
461:    public void recordDelete(){
462:       String ssn=ssnField.getText().trim();
463:       if(ssn.equals("")){
464:          //===
465:       }else{
466:          try{
467:             db.deleteCustomer(ssn);
468:          }catch(RecordNotFoundException e){
469:             //
470:          }
471:       }
472:       enableButton(true);
473:       enableField(false);
474:       emptyText();
475:       cust=db.getAllCustomers();
476:       showList(cust,custList);
477:    }
478:
479:
480:
481:    //###4.showList--->CustomerList
482:    /**
483:     * Customer List에 내용 뿌려주는 메서드
484:     * for문 이용하여 cust.length만큼 toBuild에 add() 한다.
485:     */
486:    public void showList(CustomerVO[] cust, List toBuild){
487:       toBuild.removeAll();
488:       for(int i=0;i<cust.length;i++){
489:    toBuild.add(cust[i].getSSN()+"   "+cust[i].getName()+"
    "+cust[i].getAddr());
490:       }
491:    }
492:
493:
494:
495:
496:    //###5.showList--->PortfolioList
```

```
497:    /*
498:     * 고객 주식정보 List에 내용 뿌려주는 메서드
499:     * for문 이용하여 fortfolio.length만큼 toBuild에 add() 한다.
500:     */
501:    public void showList(Vector portfolio, List toBuild){
502:       toBuild.removeAll();
503:       SharesVO sr=null;
504:       for(int i=0;i<portfolio.size();i++){
505:          sr=(SharesVO)portfolio.elementAt(i);
506:          toBuild.add(sr.getSymbol()+"    "+sr.getQuantity());
507:       }
508:    }
509:
510:
511:    //###6.showList--->StockList
512:    /**
513:     * Stock List에 내용 뿌려주는 메서드
514:     * for문 이용하여 stock.length만큼 toBuild에 add() 한다.
515:     */
516:    public void showList(StockVO[] stock, List toBuild){
517:       toBuild.removeAll();
518:       for(int i=0;i<stock.length;i++){
519:          toBuild.add(stock[i].getSymbol()+"      "+stock[i].getPrice());
520:       }
521:    }
522:
523:
524:    //###7.viewPort
525:    public void viewPort(){
526:    }//이거 뭐하려고 만들었어라?
527:
528:
529:    //###8.viewStock
530:    /**
531:     * 1) getAllStocks()이용하여 모든 stock정보 읽어들여서
532:     * 2) showList()사용하여 List에 뿌려준다.
533:     */
534:    public void viewStock(){
535:       stock=db.getAllStocks();
536:       showList(stock,stockList);
537:    }
538:
539:    //###9.viewCustomer
540:    /**
541:     * 1) getAllCustomres()이용하여 모든 customer정보 읽어들여서
542:     * 2) showList()사용하여 List에 뿌려준다.
543:     */
544:    public void viewCustomer(){
545:       cust=db.getAllCustomers();
546:       showList(cust,custList);
```

```
547:    }
548:
549:}//end class
```

7.5. Connection Pool

7.5.1. 이미 작성되거나 배치(deploy) 되어 있어야 하는 파일들

역 할	파 일 명	설 명	비 고
Value Object	broker.database.CustomerVO.java broker.database.SharesVO.java broker.database.StockVO.java	데이터베이스 테이블을 나타내는 클래스	파일 제공
예외 클래스	broker.database.exception.DuplicateIDException.java broker.database.exception.RecordNotFoundException.java broker.database.exception.InvalidTransactionException.java	예외 클래스	파일 제공
커넥션 풀	broker.database.connpool.ConnectionPool.java broker.database.connpool.ConnNotAvailException.java broker.database.connpool.ShuttingDownExcpetion.java broker.database.connpool.WrongPoolException.java	커넥션 풀과 예외 클래스	파일 제공

7.5.2. 작성해야 하는 파일들

역 할	파 일 명	설 명	비 고
DB연결	Database.java	DAO 클래스	수정

* 제공한 파일들을 가져오기 하세요.

7.5.3. broker.database.connpool.ConnectionPool.java

```
1:  package broker.database.connpool;
2:
3:  import java.sql.DriverManager;
4:  import java.sql.Driver;
5:  import java.sql.Connection;
6:  import java.sql.SQLException;
7:  import java.util.List;
8:  import java.util.LinkedList;
9:  import java.util.Map;
10: import java.util.HashMap;
11: import java.util.Iterator;
12:
13:
14: /**
15:  * This class defines a pooling mechanism for JDBC Connection objects.
16:  *
17:  * <P>
18:  * This class provides the user with a great deal of flexibility.
19:  * The user can:
20:  * <UL>
21:  *   <LI>specify the [min,max] range of number of connections in the pool
22:  *   <LI>acquire connections with three mechanisms: (a) no wait (throws
23:  *       a {@link ConnNotAvailException} if no connection is immediately
    available),
24:  *       (see {@link #getConnectionNoWait()})
25:  *       (b) wait indefinitely (at least until the pool is shutdown), or
26:  *       (see {@link #getConnectionWait()})
27:  *       (c) wait for a specified amount of time
28:  *       (see {@link #getConnectionWait(int)})
29:  *   <LI>specify a "courtesy shutdown wait period"
30:  *       (see {@link MaintenanceRunner#shutdownCourtesyTimeout})
31:  * </UL>
32:  * </P>
33:  *
34:  * <P>
35:  * This class uses a "closed synchronization" idiom.  All of the non-trivial
36:  * public methods are synchronized and none of the private methods are.
37:  * </P>
38:  *
39:
40:
41:  * <P>Revision History:
42:  * <UL>
43:  *   <LI>v1.4 [21-Jan-2001, BB] Fixed the {@link MaintenanceRunner} behavior
44:  *       to keep the minimum number of connections in the free list.
45:  *   <LI>v1.3 [20-Jan-2001, BB] Dropped the singular <TT>getConnection</TT>
46:  *       method, but promoted the {@link #getConnectionNoWait()} and the
47:  *       {@link #getConnectionWait()} methods to public access and added the
```

```
48:  *      {@link #getConnectionWait(int)} method to round out the functionality.
49:  *   <LI>v1.2 [19-Jan-2001, BB] Implemented the {@link MaintenanceRunner}
50:  *      inner class and {@link #shutdown()} method and fixed several bugs
51:  *   <LI>v1.1 [Nov-2000, BB] Scraped most of the old code as it used old
52:  *      Collections classes and had a poorly designed "maintenance" mechanism.
53:  *   <LI>v1.0 [Oct-2000, BB] First cut; based on code from the SL-310 course
54:  *      from Sun Educational Services.
55:  * </UL>
56:  *
57:  */
58: public class ConnectionPool {
59:
60:   /**
61:    * This constant defines the default minimum number of connections.
62:    */
63:   private static final int DEFAULT_MIN_CONN = 5;
64:
65:   /**
66:    * This constant defines the default maximum number of connections.
67:    */
68:   private static final int DEFAULT_MAX_CONN = 20;
69:
70:   //
71:   // basic data attributes (properties)
72:   //
73:
74:   /**
75:    * This read-only property records the fully-qualified JDBC Driver class.
76:    */
77:   private String jdbcDriverClass;
78:
79:   /**
80:    * This read-only property records the database connection URL for this pool.
81:    */
82:   private String jdbcURL;
83:
84:   /**
85:    * This read-only property records the database connection user-name for this
    pool.
86:    */
87:   private String jdbcUserName;
88:
89:   /**
90:    * This read-only property records the database connection password for this
    pool.
91:    */
92:   private String jdbcPassword;
93:
94:   /**
95:    * This read-only property records the minimum number of connections in this
```

```
    pool.
96:    */
97:   private int minimumConnections;
98:
99:   /**
100:   * This read-only property records the maximum number of connections allowed in
    this pool.
101:   */
102:  private int maximumConnections;
103:
104:  //
105:  // internal implementation attributes
106:  //
107:
108:  /**
109:   * This stores the available database connections.
110:   */
111:  private transient List freeConnections = new LinkedList();
112:
113:  /**
114:   * This stores the connections currently being used by the client.
115:   * The mapping is Connection-&LT;Long, where the Long is start time
116:   * that the connection was allocated to the client.
117:   */
118:  private transient Map usedConnections = new HashMap();
119:
120:
121:
122:
123:
124:  /**
125:   * This Boolean attribute is set when the {@link #shutdown()} method is called.
126:   * It is used to tell this object that no new connections should be granted.
127:   * It also tells the {@link MaintenanceRunner} thread to begin the shutdown
128:   * process.
129:   */
130:  private transient boolean shuttingDown = false;
131:
132:  /**
133:   * This Boolean attribute is set when the {@link MaintenanceRunner} thread
134:   * has completed the shutdown process.
135:   */
136:  private transient boolean shutdownComplete = false;
137:
138:  /**
139:   * This holds the thread running the MaintenanceRunner object.
140:   */
141:  private transient Thread maintThread = null;
142:
143:  //
```

```
144:  // Constructors
145:  //
146:
147:  /**
148:   * Constructor for the ConnectionPool.
149:   * <P>
150:   *   It is up to the client using the ConnectionPool to register the
151:   *   necessary JDBC drivers before using this class.
152:   * </P>
153:   *
154:   * @param jdbcDriverClass  The JDBC fully-qualified Driver class name
155:   * @param jdbcURL  The JDBC connect string. e.g. 'jdbc:pointbase:SL314'
156:   * @param jdbcUserName  Database login name. e.g. 'Scott'
157:   * @param jdbcPassword  Database password. e.g. 'Tiger'
158:   * @param minimumConnections  Minimum number of connections to start with.
159:   * @param maximumConnections  Maximum number of connections in dynamic pool.
160:   * @param shutdownCourtesyTimeout  The "courtesy shutdown wait period"
161:   *
162:   * @exception ClassNotFoundException  Thrown if the jdbcDriverClass is not
163:   *    found in the CLASSPATH
164:   * @exception InstantiationException  Thrown if the jdbcDriverClass is not
165:   *    a concrete class
166:   * @exception IllegalAccessException  Thrown if the jdbcDriverClass or its
167:   *    initializer is not accessible
168:   * @exception SQLException  Thrown on a JDBC exception
169:   */
170:  public ConnectionPool(String jdbcDriverClass, String jdbcURL,
171:                        String jdbcUserName, String jdbcPassword,
172:                        int minimumConnections, int maximumConnections,
173:                        int shutdownCourtesyTimeout)
174:    throws ClassNotFoundException, InstantiationException,
175:           IllegalAccessException, SQLException {
176:
177:    // Initialize basic attributes
178:    this.jdbcDriverClass    = jdbcDriverClass;
179:    this.jdbcURL           = jdbcURL;
180:    this.jdbcUserName       = jdbcUserName;
181:    this.jdbcPassword       = jdbcPassword;
182:    this.minimumConnections = minimumConnections;
183:    this.maximumConnections = maximumConnections;
184:
185:    // Register the JDBC driver with the DriveManager
186:    Driver driver = (Driver) Class.forName(jdbcDriverClass).newInstance();
187:    DriverManager.registerDriver(driver);
188:
189:    // Initialize implementation attributes
190:    for (int i=0; i < minimumConnections; i++) {
191:      freeConnections.add(makeConnection());
192:    }
193:
```

```
194:    // Initialize the CP maintenance thread
195:    maintThread = new Thread(new MaintenanceRunner(shutdownCourtesyTimeout));
196:    maintThread.start();
197:  }
198:
199:  /**
200:   * Constructor for the ConnectionPool.
201:   * <P>
202:   *   It is up to the client using the ConnectionPool to register the
203:   *   necessary JDBC drivers before using this class.
204:   * </P>
205:   *
206:   * @param jdbcDriverClass JDBC fully-qualified Driver class name
207:   * @param jdbcURL tJDBC connec tstring. e.g. 'jdbc:pointbase:SL314'
208:   * @param jdbcUserName    Database login name. e.g. 'Scott'
209:   * @param jdbcPassword    Database password. e.g. 'Tiger'
210:   * @param minimumConnections  Minimum number of connections to star twith.
211:   * @param maximumConnections  Maximum number of connections in dynamic pool.
212:   *
213:   * @exception ClassNotFoundException  Thrown if the jdbcDriverClass is not
214:   *    found in the CLASSPATH
215:   * @exception InstantiationException  Thrown if the jdbcDriverClass is not
216:   *    a concrete class
217:   * @exception IllegalAccessException  Thrown if the jdbcDriverClass or its
218:   *    initializer is no taccessible
219:   * @exception SQLException  Thrown on a JDBC exception
220:   */
221:  public ConnectionPool(String jdbcDriverClass, String jdbcURL,
222:                        String jdbcUserName, String jdbcPassword,
223:                        int minimumConnections, int maximumConnections)
224:    throws ClassNotFoundException, InstantiationException,
225:           IllegalAccessException, SQLException {
226:    this(jdbcDriverClass, jdbcURL, jdbcUserName, jdbcPassword,
227:         minimumConnections, maximumConnections,
228:         MaintenanceRunner.DEFAULT_SHUTDOWN_TIMEOUT);
229:  }
230:
231:  /**
232:   * Constructor for the ConnectionPool.
233:   * <P>
234:   *   It is up to the client using the ConnectionPool to register the
235:   *   necessary JDBC drivers before using this class.
236:   * </P>
237:   * @param jdbcDriverClass JDBC fully-qualified Driver class name
238:   * @param jdbcURL tJDBC connec tstring. e.g. 'jdbc:pointbase:SL314'
239:   * @param jdbcUserName    Database login name. e.g. 'Scott'
240:   * @param jdbcPassword    Database password. e.g. 'Tiger'
241:   *
242:   * @exception ClassNotFoundException  Thrown if the jdbcDriverClass is not
243:   *    found in the CLASSPATH
```

```
244:   * @exception InstantiationException  Thrown if the jdbcDriverClass is not
245:   *    a concrete class
246:   * @exception IllegalAccessException  Thrown if the jdbcDriverClass or its
247:   *    initializer is no taccessible
248:   * @exception SQLException  Thrown on a JDBC exception
249:   */
250:    public ConnectionPool(String jdbcDriverClass, String jdbcURL,
251:                          String jdbcUserName, String jdbcPassword)
252:    throws ClassNotFoundException, InstantiationException,
253:           IllegalAccessException, SQLException {
254:    this(jdbcDriverClass, jdbcURL, jdbcUserName, jdbcPassword,
255:        DEFAULT_MAX_CONN, DEFAULT_MIN_CONN,
256:        MaintenanceRunner.DEFAULT_SHUTDOWN_TIMEOUT);
257:  }
258:
259:
260:  //
261:  // Accessors for basic properties
262:  //
263:
264:  /**
265:   * This accessor returns the JDBC URL for this connection pool.
266:   */
267:  public String getJdbcDriverClass() {
268:    return jdbcDriverClass;
269:  }
270:
271:  /**
272:   * This accessor returns the JDBC URL for this connection pool.
273:   */
274:  public String getJdbcURL() {
275:    return jdbcURL;
276:  }
277:
278:  /**
279:   * This accessor returns the JDBC user-name for this connection pool.
280:   */
281:  public String getJdbcUserName() {
282:    return jdbcUserName;
283:  }
284:
285:  /**
286:   * This accessor returns the JDBC password for this connection pool.
287:   */
288:  public String getJdbcPassword() {
289:    return jdbcPassword;
290:  }
291:
292:  /**
293:   * This accessor returns the minimum number of connections for this pool.
```

```
294:   */
295:  public int getMinimumConnections() {
296:    return minimumConnections;
297:  }
298:
299:  /**
300:   * This accessor returns the maximum number of connections for this pool.
301:   */
302:  public int getMaximumConnections() {
303:    return maximumConnections;
304:  }
305:
306:  //
307:  // Public methods
308:  //
309:
310:  /**
311:   * This method immediately returns the next available connection.
312:   * I twill never wait for a free connection.  Instead, i twill
313:   * throw a ConnNotAvailException if no connection is immediately available.
314:   *t
315:   * @return Connection the next available JDBC connection
316:   * @exception ConnNotAvailException  Thrown if no connection is available
317:   * @exception ShuttingDownException  Thrown if the connection pool has been
    ordered
318:   * to shutdown
319:   */
320:  public synchronized Connection getConnectionNoWait()
321:        throws ConnNotAvailException, ShuttingDownException {
322:    Connection result = null;
323:
324:    // If the pool is being shutdown, throw an exception
325:    if ( shuttingDown ) {
326:      // Otherwise, throw a ShuttingDownException
327:      throw new ShuttingDownException();
328:    }
329:
330:    // Attempt to ge ta connection
331:    result = getConnectionInternal();
332:
333:
334:    // Throw an exception if no connection is available
335:    if ( result == null ) {
336:      throw new ConnNotAvailException();
337:    }
338:    // Return it
339:    return result;
340:  }//end getConnectionNoWait()
341:
342:  /**
```

```
343:   * This method returns the next available connection.
344:   * <P> I twill wait indefinitely for a free connection, unless the
345:   *     {@link #shuttingDown} flag is set.
346:   * <P> Note: this method is always called within a synchronized public method.
347:   * @return Connection the next available JDBC connection
348:   * @exception ShuttingDownException  Thrown if the connection pool has been
    ordered
349:   * to shutdown
350:   */
351:  public synchronized Connection getConnectionWait()
352:        throws ShuttingDownException {
353:    Connection result = null;
354:
355:    // wait until a free connection is available (or until the pool
356:    // is being shu tdown).
357:    while (    (! shuttingDown)
358:              && ((result = getConnectionInternal()) == null) ) {
359:      try { this.wait(); } catch (InterruptedException ex) { }
360:    }
361:
362:    // If the pool is being shutdown, throw an exception
363:    if ( shuttingDown ) {
364:      // Otherwise, throw a ShuttingDownException
365:      throw new ShuttingDownException();
366:    }
367:
368:    // Return it
369:    return result;
370:
371:  }//end getConnectionWait()
372:
373:
374:
375:
376:  /**
377:   * This method returns the next available connection and will wait
378:   * a specified amount of time to get one.
379:   * @return Connection the next available JDBC connection
380:   * @exception ConnNotAvailException  Thrown if no connection is available
381:   * @exception ShuttingDownException  Thrown if the connection pool has been
    ordered
382:   * to shutdown
383:   */
384:  public synchronized Connection getConnectionWait(int timeout)
385:        throws ShuttingDownException, ConnNotAvailException {
386:    Connection result = null;
387:
388:    // If the pool is being shutdown, throw an exception
389:    if ( shuttingDown ) {
390:      throw new ShuttingDownException();
```

```
391:    }
392:
393:    // Attempt to ge ta connection.
394:    result = getConnectionInternal();
395:    if ( result != null ) {
396:      return result;
397:    }
398:
399:    // If no tfree connection is ye tavailable,
400:    // then wait until a free connection is available
401:    try { this.wait(timeout); } catch (InterruptedException ex) { }
402:
403:    // Check again if the pool is being shutdown
404:    if ( shuttingDown ) {
405:      throw new ShuttingDownException();
406:    }
407:
408:    // Attempt to ge ta connection one last time
409:    result = getConnectionInternal();
410:    if ( result == null ) {
411:      // Throw an exception if no connection is available
412:      throw new ConnNotAvailException();
413:    } else {
414:      // Otherwise, return it
415:      return result;
416:    }
417:  }//end getConnectionWait(int timeout)
418:
419:  /**
420:   * Frees a connection.
421:   */
422:  public synchronized void releaseConnection(Connection connection)
423:    throws WrongPoolException, SQLException {
424:    if ( usedConnections.containsKey(connection) ) {
425:      usedConnections.remove(connection);
426:      if ( ! connection.isClosed() ) {
427:        if ( shuttingDown ) {
428:          try {
429:            connection.close();
430:          } catch (SQLException sex) { }
431:        } else {
432:          freeConnections.add(connection);
433:        }
434:      } else {
435:        // do nothing, let the connection object be GCed
436:      }
437:    } else {
438:      throw new WrongPoolException();
439:    }
440:    this.notifyAll();
```

```
441:  }//end releaseConnection(Connection connection)
442:
443:
444:  /**
445:   * This method tells the connection pool to shutdown.
446:   * This method signals the {@link MaintenanceRunner} thread to perform
447:   * the shutdown operations.  The completion of tha tprocess is signaled
448:   * by the {@link #shutdownComplete} attribute.
449:   */
450:  public synchronized void shutdown() {
451:    shuttingDown = true;
452:    this.notifyAll();
453:    while ( ! shutdownComplete ) {
454:      try { this.wait(); } catch (InterruptedException ex) { }
455:    }
456:  }
457:
458:
459:
460:  /**
461:   * Returns the age of a connection.
462:   * @return long the time (in milliseconds) since i twas handed out
463:   * to an application (-1 if no tbeing used)
464:   */
465:  public synchronized long getAge(Connection connection) {
466:    Long start_time = (Long) usedConnections.get(connection);
467:    if ( start_time == null ) {
468:      return -1;
469:    } else {
470:      return ( System.currentTimeMillis() - start_time.longValue() );
471:    }
472:  }
473:
474:  /**
475:   * Returns the number of connections in the dynamic pool.t
476:   */
477:  public synchronized int getSize() {
478:    return usedConnections.size() + freeConnections.size();
479:  }
480:
481:  /**
482:   * Returns the number of connections in use.t
483:   */
484:  public synchronized int getUseCount() {
485:    return usedConnections.size();
486:  }
487:
488:
489:  //
490:  // Private methods
```

```
491:  //
492:
493:  /**
494:   * This method returns the next available (or new) connection.
495:   * <P> Note: this method is always called within a synchronized public method.
496:   * @return Connection the next available JDBC connection (or <TT>null</TT> if
    no
497:   * connection is available)
498:   */
499:  private Connection getConnectionInternal() {
500:    Connection result = null;
501:
502:    if ( ! freeConnections.isEmpty() ) {
503:      // get the firs titem on the free list
504:      result = (Connection) freeConnections.remove(0);
505:    } else if ( usedConnections.size() < maximumConnections ) {
506:      // try to make a new connection
507:      try {
508:        result = makeConnection();
509:      } catch (SQLException sex) {
510:        sex.printStackTrace(System.err);
511:      }
512:    }
513:
514:    // Store i tin used connections map
515:    if ( result != null ) {
516:      // get the time of this request
517:      Long start_time = new Long(System.currentTimeMillis());
518:      // pu tthis connection in the used map
519:      usedConnections.put(result, start_time);
520:    }
521:
522:    // Return it
523:    return result;
524:
525:  }//end getConnectionInternal()
526:
527:
528:  /**
529:   * This method creates a new JDBC connection for this pool.
530:   * @return Connection a newly created JDBC connection object
531:   */
532:  private Connection makeConnection() throws SQLException {
533:    if ( jdbcUserName.length() > 0 ) {
534:      return DriverManager.getConnection(jdbcURL, jdbcUserName, jdbcPassword);
535:    } else {
536:      return DriverManager.getConnection(jdbcURL);
537:    }
538:  }//end makeConnection()
```

```
1:    /**
2:     * This inner class is used to perform clean-up on a connection pool.
3:     * An object of this class runs in a separate thread (a ta low priority)
4:     * and closes any extra connections.
5:     */
6:    class MaintenanceRunner implements Runnable {
7:
8:      /**
9:       * This constant defines the default length of time (10 seconds) that the
10:     * {@link MaintenanceRunner} thread will wait for used connections to
11:     * be released.
12:     */
13:    private static final int DEFAULT_SHUTDOWN_TIMEOUT = 10 * 1000;
14:
15:    /**
16:     * This attribute defines the timeout policy/duration that the shutdown
17:     * process waits for clients to close "used connections".
18:     * <UL>
19:     *  <LI><TT>-1</TT>: don't wait, close all open connections immediately
20:     *  <LI><TT>0</TT>: wait indefinitely, allow the clients to close connections
    at their leisure
21:     *  <LI><TT>&GT;0</TT>: wait for a specified amount of time (in milliseconds)
22:     * </UL>
23:     */
24:    private int shutdownCourtesyTimeout;
25:
26:    /**
27:     * This constructor allows the client to initialize
28:     * the {@link #shutdownCourtesyTimeout} value.
29:     */
30:    public MaintenanceRunner(int timeout) {
31:      this.shutdownCourtesyTimeout = timeout;
32:    }
33:
34:    /**
35:     * This constructor initializes the {@link #shutdownCourtesyTimeout}
36:     * to a defaul t(positive) value.
37:     */
38:    public MaintenanceRunner() {
39:      this(DEFAULT_SHUTDOWN_TIMEOUT);
40:    }
41:
42:    /**
43:     * The thread process.  This method synchronizes on the pool object so
44:     * that it can perform <TT>wait</TT> operations.  Most of the time this
45:     * thread will be waiting in the {@link #maintLoop()} method.  Once the
46:     * {@link #shuttingDown} flag is set, then this thread will enter the
47:     * {@link #shutdown()} process.  This method sets the {@link
    #shutdownComplete}
48:     * when i tis done; at which time the {@link #shutdown()} method returns and
```

```
49:      * the <TT>run</TT> method return and the thread will halt.
50:      */
51:     public void run() {
52:       synchronized (ConnectionPool.this) {
53:         maintLoop();
54:         shutdown();
55:       }
56:     }
57:
58:     /**
59:      * This method maintains the ideal number of connections in the
60:      * pool during normal operations (that is, before the pool is shutdown).
61:      */
62:     private void maintLoop() {
63:       while ( ! shuttingDown ) {
64:
65:         // Shink the free collection, if there are too many
66:         for ( int i = getSize();
67:               // allow the minimum in the free collection a tany time
68:               freeConnections.size() > minimumConnections;
69:               i-- ) {
70:           Connection conn = (Connection) freeConnections.remove(0);
71:           try { conn.close(); } catch (SQLException sex) {}
72:         }
73:
74:         // Grow the free collection, if there are no tenough
75:         // This can occur if the clien tprocesses are (errantly) closing
76:         // the connections before releasing them.
77:         for ( int i = getSize(); (i < minimumConnections); i++ ) {
78:           try {
79:             Connection conn = makeConnection();
80:             freeConnections.add(conn);
81:           } catch (SQLException sex) {}
82:         }
83:
84:         // wait for the next time to perform maintenance
85:         try { ConnectionPool.this.wait(); } catch (InterruptedException ex) { }
86:       }
87:     }//end mainLoop()
88:
89:     /**
90:      * This method implements the "shutting down" process.  It's job is to
91:      * close the connections in the pool.
92:      * <P><B>Note:</B>
93:      * <UL>
94:      *  <LI>The ConnectionPool object is responsible for rejecting requests
95:      *      for connections once the {@link #shuttingDown} flag is set.
96:      *  <LI>This process will allow the clients a "courtesy timeout" if there
97:      *      are still connections being used.  (see {@link
    #shutdownCourtesyTimeout})
```

```
98:      * </UL>
99:      */
100:    private void shutdown() {
101:      Iterator connections;
102:
103:      if ( (shutdownCourtesyTimeout > -1) && ! usedConnections.isEmpty() ) {
104:        // Play fair: give the client's an opportunity to finish their work
105:        if ( shutdownCourtesyTimeout == 0 ) {
106:          // wait indefinitely
107:          while ( ! usedConnections.isEmpty() ) {
108:            try {
109:              ConnectionPool.this.wait(shutdownCourtesyTimeout);
110:            } catch (InterruptedException ex) { }
111:          }
112:        } else {
113:          // Otherwise, wait for the specified amoun tof time
114:          if ( ! usedConnections.isEmpty() ) {
115:            try {
116:              ConnectionPool.this.wait(shutdownCourtesyTimeout);
117:            } catch (InterruptedException ex) { }
118:          }
119:        }//end else
120:      }//end if
121:
122:      // Release free connections
123:      connections = freeConnections.iterator();
124:      while ( connections.hasNext() ) {
125:        Connection conn = (Connection) connections.next();
126:        try { conn.close(); } catch (SQLException sex) {}
127:      }
128:
129:      // Release used connections
130:      connections = usedConnections.keySet().iterator();
131:      while ( connections.hasNext() ) {
132:        Connection conn = (Connection) connections.next();
133:        try { conn.close(); } catch (SQLException sex) {}
134:      }
135:
136:      // Repor tback to the thread tha tbegan the shutdown process
137:      // tha ti thas been completed.
138:      shutdownComplete = true;
139:      ConnectionPool.this.notifyAll();
140:    }
141:  }//end shutdown()
142:
143:}//end class
```

7.5.4. 예외 클래스

7.5.4.1. ConnNotAvailException

broker.database.connpool.ConnNotAvailException.java

```
1:  package broker.database.connpool;
2:
3:  public class ConnNotAvailException extends RuntimeException {
4:  }
```

7.5.4.2. ShuttingDownExcpetion

broker.database.connpool.ShuttingDownExcpetion.java

```
1:  package broker.database.connpool;
2:
3:  public class ShuttingDownException extends RuntimeException {
4:  }
```

7.5.4.3. WrongPoolException

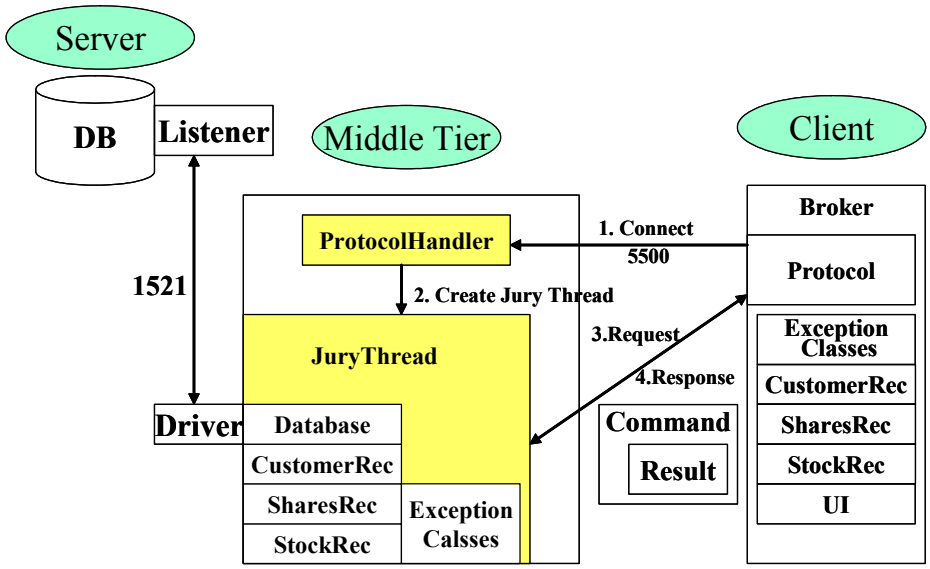
broker.database.connpool.WrongPoolException.java

```
1:  package broker.database.connpool;
2:
3:  public class WrongPoolException extends Exception {
4:  }
```

7.5.5. Database.java 수정

```
1:      //생략...
2:     import broker.database.connpool.ConnectionPool;
3:
4:     //생략...
5:
6:     private static final int initialCons = 1;
7:     private static final int maxCons = 1;
8:
9:     private ConnectionPool cp = null;
10:
11:    public Database(String server)
12:           throws ClassNotFoundException, SQLException {
13:        String url = "jdbc:oracle:thin:@"+server+":1521:orcl";
14:        try{
15:           cp = new ConnectionPool(DRIVER, url, USER_ID, PASSWORD, initialCons,
    maxCons);
16:        }catch(Exception e) {
17:           System.out.println("Connection Pool Failed...");
18:        }
19:        con = cp.getConnectionNoWait();
20:        System.out.println("Connection is setting.");
21:        stmt = con.createStatement();
22:        System.out.println("Statement is created.");
23:     }
24:
25:   //아래 부분 생략
```

7.6. 3-Tier Application



7.6.1. 이미 작성되거나 배치(deploy) 되어 있어야 하는 파일들

역할	파 일 명	설 명	비 고
Value Object	broker.database.CustomerVO.java broker.database.SharesVO.java broker.database.StockVO.java	데이터베이스 테이블을 나타내는 클래스	파일 제공
예외 클래스	brker.database.exception.DuplicateIDException.java broker.database.exception.RecordNotFoundException.java broker.database.exception.InvalidTransactionException.java	예외 클래스	파일 제공
커넥션 풀	broker.database.connpool.ConnectionPool.java broker.database.connpool.ConnNotAvailException.java broker.database.connpool.ShuttingDownExcpetion.java broker.database.connpool.WrongPoolException.java	커넥션 풀	파일 제공

7.6.2. 작성해야 하는 파일들

역 할	파 일 명	설 명	비 고
클라이언트	Broker.java	메인 클래스	수정
결과저장	broker.Result.java	client-middle 을 오고가는 class(Command안에 포함됨)	
Command	broker.Command.java	client-middle server 오고가는 클래스	
Protocol	broker.Protocol.java	Database.java 와 동일 메서드 존재(서버의 JuryThread와 통신)	
미들서버	broker.JuryThread.java	클라이언트의 Protocol과 통신하는 쓰래드	
미들서버	ProtocolHandler.java	미들서버 메인 클래스	

◎ 미들서버 실행: **ProtocolHandler.java**

◎ 클라이언트 실행: **Borker.java**

※ **Eclipse**에서  클릭 하면 실행되고 있는 프로세스의 출력 창을 볼 수 있음

※ 서버 종료는  클릭 하면 종료됨

7.6.3. broker.Result.java

```
1:  package broker;
2:
3:  import java.util.*;
4:  import java.io.*;
5:
6:  public class Result extends Vector implements Serializable{
7:     private int status = -1;//결과상태변수
8:
9:     public Result(){
10:       super(1,1);//Vecotor(1,1)호출 의미함
11:       //new Vectoer(1,1);
12:    }
13:
14:    //결과의 상태 값을 읽음 -> client 사용
15:    public int getStatus(){
16:       return status;
17:    }
18:    //결과 상태 값을 세팅함-> 미들서버에서 세팅
19:    public void setStatus(int value){
20:       status = value;
21:    }
22:    //결과의 크기 -> client 가 사용
23:    public int getNumRows(){
24:       return this.size();
25:    }
26:
27: }//end Result class
```

7.6.4. broker.Command.java

```
1:  package broker;
2:
3:  import java.io.*;
4:
5:  class Command implements Serializable{
6:     private int commandValue;//명령어 상수값
7:     private Result results;//결과값을 담을 변수
8:     private String [] args = {""};//이름 주소 주민등록번호등의 인수를 담는 배열
9:
10:    public static final intBUYSHARES = 10; //buyShares()
11:    public static final int SELLSHARES = 20;//sellShares()
12:    public static final int READSTOCKLIST = 30;//getAllStocks()
13:    public static final int READASTOCK = 40;//
14:    public static final int READCUSTLIST = 50;//getAllCustomers()
15:    public static final int VIEWACUSTOMER = 60;//getCustomer() getPortfolio()
16:    public static final int ADDACUSTOMER = 70;//addCustomer()
17:    public static final int UPDATECUSTOMER = 80;//updateCustomer()
18:    public static final int DELETECUSTOMER = 90;//deleteCustomer()
19:
20:    //Protocol 에서 호출
21:    public Command (int comm){
22:       commandValue = comm;
23:       results = new Result();//결과를 담을 객체생성
24:    }
25:
26:    //미들서버에서 사용할 메서드:명령상수값은 얻기 위함
27:    public int getCommandValue(){
28:       return commandValue;
29:    }
30:
31:    //인수를 세팅 -> client 가 사용하는 메서드 ->Protocol에서 사용함
32:    //Protocol에서 화면에 있는 값을 getText() ->setArg(인수)
33:    public void setArgs(String [] params){
34:       args = params;
35:    }
36:
37:    //미들서버에서 사용하는 메서드
38:    public String [] getArgs(){
39:       return args;
40:    }
41:
42:    //client가 화면에 뿌리기 전에 호출하는 메서드
43:    public Result getResult(){
44:       return results;
45:    }
46:
47: }//end Command class
```

7.6.5. broker.Protocol.java

```
1:  package broker;
2:
3:  import java.io.IOException;
4:  import java.io.ObjectInputStream;
5:  import java.io.ObjectOutputStream;
6:  import java.net.Socket;
7:
8:  import broker.database.CustomerVO;
9:  import broker.database.StockVO;
10: import broker.database.exception.DuplicateIDException;
11: import broker.database.exception.InvalidTransactionException;
12: import broker.database.exception.RecordNotFoundException;
13:
14: public class Protocol {
15:
16:    private static final int MIDDLE_PORT = 5500;
17:    private Socket socket;
18:    private ObjectInputStream ois;
19:    private ObjectOutputStream oos;
20:    private Command cmd;
21:
22:    //생성자
23:    //socket, ObjectInputStream, ObjectOutputStream을 생성한다.
24:    public Protocol(String serverName) {
25:       try{
26:          socket = new Socket(serverName, MIDDLE_PORT);
27:          ois = new ObjectInputStream(socket.getInputStream());
28:          oos = new ObjectOutputStream(socket.getOutputStream());
29:       }catch (IOException ioe){
30:          System.out.println("Protocol constructor: "+ioe);
31:       }
32:    }
33:
34:    public void close(){
35:       try{
36:          socket.close();
37:       }catch (IOException ioe){
38:          System.out.println("Protocol.close(): "+ioe);
39:       }
40:    }
41:
42:    //Stream으로 부터 읽은 Command객체를 member에 할당하고
43:    //Command의 status 값을 가져온다.
44:    public int getResponse(){
45:       try{
46:          cmd = (Command)ois.readObject();
47:       }catch (ClassNotFoundException cnfe){
48:          System.out.println("Protocol.getResponse(): "+cnfe);
```

```
49:         }catch (IOException ioe){
50:            System.out.println("Protocol.getResponse(): "+ioe);
51:         }
52:         int status = cmd.getResult().getStatus();
53:         return status;
54:      }
55:
56:      //Command object를 stream에 write한다.
57:      public void writeCommand(Command cmd){
58:         try{
59:            oos.writeObject(cmd);
60:            System.out.println("client writeObject().....end");
61:         }catch (IOException ioe){
62:            System.out.println("Protocol.writeObject(): "+ioe);
63:         }
64:      }
```

```
65:     //10 -> BUYSHARES
66:     public void buyShares(String ssn, String symbol, int quantity) throws
    RecordNotFoundException {
67:         cmd = new Command(Command.BUYSHARES);
68:         String [] str = {ssn, symbol, String.valueOf(quantity) };
69:         cmd.setArgs(str);
70:         writeCommand(cmd);//서버가 readObject()기다림
71:         //=========
72:         int status = getResponse();//상태변수
73:         if ( status < 0 ){
74:             System.out.println("Protocol.buyShares(): new
    RecordNotFoundException()");
75:             throw new RecordNotFoundException();
76:         }
77:     }
78:
79:
80:     //20 -> SELLSHARES
81:     public void sellShares(String ssn, String symbol, int quantity) throws
    RecordNotFoundException, InvalidTransactionException {
82:         cmd = new Command(Command.SELLSHARES);
83:         String [] str = { ssn, symbol, String.valueOf(quantity) };
84:         cmd.setArgs(str);
85:         writeCommand(cmd);
86:         int status = getResponse();
87:         switch ( status ){
88:         case -1://해당고객이 없는 경우
89:             System.out.println("Customer not found!!!");
90:             throw new RecordNotFoundException ();
91:
92:         case -2://없는 주식이거니 보유량<매도량일 경우
93:             System.out.println("Invalid Stock Symbol!!!");
94:             throw new InvalidTransactionException ();
95:         }
96:     }
97:
98:     //30 -> READSTOCKLIST
99:     public StockVO[] getAllStocks() {
100:        StockVO [] sr = null;
101:        cmd = new Command(Command.READSTOCKLIST);
102:        writeCommand(cmd);
103:        int status = getResponse();
104:        sr = (StockVO [])cmd.getResult().get(0);
105:        return sr;
106:    }
107:
108:
109:    //40 -> READASTOCK
110:    public float getStockPrice(String symbol) {
111:        float price = 0.0f;
```

```
112:        cmd = new Command(Command.READASTOCK);
113:        String [] str = {symbol};
114:        cmd.setArgs(str);
115:        writeCommand(cmd);
116:        int status = getResponse();
117:        price = ((Float)(cmd.getResult().get(0))).floatValue();
118:        return price;
119:     }
120:
121:
122:     //50 -> READCUSTLIST
123:     public CustomerVO[] getAllCustomers() {
124:        CustomerVO [] cr = null;
125:        cmd = new Command(Command.READCUSTLIST);
126:        writeCommand(cmd);
127:        int status = getResponse();
128:        cr = (CustomerVO [])cmd.getResult().get(0);
129:        return cr;
130:     }
131:
132:
133:
134:     //60 -> VIEWACUSTOMER
135:     public CustomerVO getCustomer(String ssn) throws RecordNotFoundException {
136:        cmd = new Command(Command.VIEWACUSTOMER);
137:        String [] str = {ssn};
138:        cmd.setArgs(str);
139:        writeCommand(cmd);
140:
141:        int status = getResponse();
142:        if ( status < 0 ){//해당 고객이 없을 때
143:           System.out.println("Protocol error:
    getCustomer().....Excpetion");
144:           throw new RecordNotFoundException();
145:        }
146:        CustomerVO cust = (CustomerVO)cmd.getResult().get(0);
147:        return cust;
148:     }
149:
150:
151:     //70 -> ADDACUSTOMER
152:     public void addCustomer(String ssn, String name, String address) throws
    DuplicateIDException {
153:        cmd = new Command(Command.ADDACUSTOMER);
154:        String [] str = {ssn, name, address};
155:        cmd.setArgs(str);
156:        writeCommand(cmd);
157:        int status = getResponse();
158:        if ( status < 0 ){
159:           System.out.println ("Protocol error:
```

```
    addCustomer().....Exception");
160:          throw new DuplicateIDException ();
161:       }
162:    }
163:
164:
165:
166:    //80 -> UPDATECUSTOMER
167:    public void updateCustomer(String ssn, String name, String address) throws
    RecordNotFoundException {
168:       cmd = new Command(Command.UPDATECUSTOMER);
169:       String [] str = {ssn, name, address};
170:       cmd.setArgs(str);
171:       writeCommand(cmd);
172:       int status = getResponse();
173:       if ( status < 0 ){
174:          System.out.println("Protocol error:
    updateCustomer().....Excpetion");
175:          throw new RecordNotFoundException();
176:       }
177:    }
178:
179:
180:    //90 -> DELETECUSTOMER
181:    public void deleteCustomer(String ssn) throws RecordNotFoundException {
182:       cmd = new Command(Command.DELETECUSTOMER);
183:       String [] str = {ssn};
184:       cmd.setArgs(str);
185:       writeCommand(cmd);
186:       int status = getResponse();
187:       if ( status < 0 ){
188:          System.out.println ("Protocol error:
    deleteCustomer().....Excpetion");
189:          throw new RecordNotFoundException ();
190:       }
191:    }
192:
193:}//end class
```

7.6.6. broker.JuryThread.java

```
1:  package broker;
2:
3:  import java.io.IOException;
4:  import java.io.ObjectInputStream;
5:  import java.io.ObjectOutputStream;
6:  import java.net.Socket;
7:
8:  import broker.database.CustomerVO;
9:  import broker.database.Database;
10: import broker.database.StockVO;
11: import broker.database.exception.DuplicateIDException;
12: import broker.database.exception.InvalidTransactionException;
13: import broker.database.exception.RecordNotFoundException;
14:
15: public class JuryThread extends Thread {
16:     public Socket s;
17:     public ObjectOutputStream oos;
18:     public ObjectInputStream ois;
19:     public Database db;
20:     public Command cmd;
21:
22:     //생성자
23:     //argument로 받은 socket과 db를 자신의 member에 할당하고
24:     //stream을 생성(ObjectInputStream, ObjectOutputStream)
25:     public JuryThread (Socket s, Database db){
26:         this.s = s;
27:         this.db = db;
28:         try {
29:             oos = new ObjectOutputStream(s.getOutputStream());
30:             ois = new ObjectInputStream(s.getInputStream());
31:         } catch (IOException e) {
32:             System.out.println("Error in JuryThread creating Object Input or
    Output Stream");
33:         }
34:         System.out.println("start JuryThread....");
35:     }
36:
37:     //무한 loop을 돌면서 client로 부터 Command객체를 read한 후
38:     //Command객체의 commandValue와 args를 가지고 db의 해당 method를 호출한다.
39:     public void run() {
40:         while (true) {
41:             try {
42:                 cmd = (Command)ois.readObject();
43:             } catch (Exception e) {
44:                 System.out.println ("Error reading command value from client");
45:                 return;
46:             }
47:
```

```
48:          String [] args = cmd.getArgs();
49:          Result r = cmd.getResult();
50:
51:          // commandValue값에 따라 db의 method호출
52:          switch (cmd.getCommandValue()) {
53:
54:             //void buyShares(ssn,symbol,quantity)
55:             case Command.BUYSHARES:
56:                try{
57:             db.buyShares(args[0], args[1], Integer.parseInt(args[2]));
58:                   r.setStatus(0);
59:                   }catch(RecordNotFoundException e) {
60:                   System.out.println("Error in SELLSHARES");
61:                   r.setStatus(-1);
62:                }
63:                break;
64:
65:             //void sellShares(ssn,symbol,quantity)
66:             case Command.SELLSHARES:
67:                   try{
68:             db.sellShares(args[0], args[1], Integer.parseInt(args[2]));
69:                         r.setStatus(0);
70:                   }catch(RecordNotFoundException e) {
71:                   System.out.println("Error in SELLSHARES");
72:                   r.setStatus(-1);
73:                }catch(InvalidTransactionException e){
74:                         System.out.println("Error in sellshares");
75:                         r.setStatus(-2);
76:                   }
77:                   break;
78:
79:             //StockVO[] getAllStocks()
80:             case Command.READSTOCKLIST:
81:                StockVO [] allStock = db.getAllStocks();
82:                //결과를 Result객체에 담는다.
83:                r.add(allStock);
84:                r.setStatus(0);
85:                break;
86:
87:             //float getStockPrice(symbol)
88:             case Command.READASTOCK:
89:                float f = db.getStockPrice(args[0]);
90:                r.add(new Float(f));
91:                r.setStatus(0);
92:                   break;
93:
94:             //CustomerVO[] getAllCustomer()
95:             case Command.READCUSTLIST:
96:                CustomerVO [] allCust = db.getAllCustomers();
97:                r.add(allCust);
```

```
98:                  r.setStatus(0);
99:                  break;
100:
101:             //CustomerVO getCustomer(ssn)
102:             case Command.VIEWACUSTOMER:
103:                try{
104:                         CustomerVO cust = db.getCustomer(args[0]);
105:                   r.addElement(cust);
106:                         r.setStatus(0);
107:                }catch(RecordNotFoundException e) {
108:                   r.setStatus(-1);
109:                }
110:                   break;
111:
112:             //void addCustomer(ssn,name,address)
113:             case Command.ADDACUSTOMER:
114:                   try{
115:                   db.addCustomer(args[0],args[1],args[2]);
116:                   r.setStatus(0);
117:                }catch(DuplicateIDException e){
118:                   System.out.println("SSN Exists!!!");
119:                   r.setStatus(-1);
120:                }
121:                break;
122:
123:             //void updateCustomer(ssn,name,address)
124:             case Command.UPDATECUSTOMER:
125:                try{
126:                   db.updateCustomer(args[0], args[1], args[2]);
127:                   r.setStatus(0);
128:                }catch (RecordNotFoundException re){
129:                   System.out.println("Record not Found!!!!");
130:                   r.setStatus(-1);
131:                }
132:                break;
133:
134:             //void deleteCustomer(ssn)
135:             case Command.DELETECUSTOMER:
136:                try{
137:                   db.deleteCustomer(args[0]);
138:                   r.setStatus(0);
139:                }catch (RecordNotFoundException re){
140:                   System.out.println("Record Not Found!!!!!!!!");
141:                   r.setStatus(-1);
142:                }
143:                break;
144:          } // end switch
145:
146:          //setting된 Command객체를 client로 전송
147:          try {
```

```
148:             oos.writeObject(cmd);
149:          } catch (IOException ioe) {
150:             System.out.println ("JuryThread.run(): "+ioe);
151:          }
152:       }// end while
153:    } // end run
154:
155:} //end class
```

7.6.7. ProtocolHandler.java

```
1:  package broker;
2:
3:  import java.io.IOException;
4:  import java.net.ServerSocket;
5:  import java.net.Socket;
6:  import java.sql.SQLException;
7:
8:  import broker.database.Database;
9:
10:
11: public class ProtocolHandler extends Thread {
12:
13:    public ServerSocket server = null;
14:    public Socket socket;
15:    public JuryThread jury;
16:    public Database db;
17:    public int port = 5500;
18:
19:    //생성자
20:    //ServerSocket, Database생성
21:    public ProtocolHandler(String DBServer) {
22:       try {
23:          server = new ServerSocket(port);
24:          db = new Database (DBServer);
25:       } catch(IOException ioe) {
26:          System.out.println("ProtocolHandler constructor: "+ioe);
27:       } catch(ClassNotFoundException cnfe){
28:          System.out.println("ProtocolHandler constructor: "+cnfe);
29:       } catch (SQLException sqle) {
30:          System.out.println ("ProtocolHandler constructor: " +sqle);
31:       }
32:       System.out.println("start ProtocolHandler... service port:"+ port );
33:    }
34:
35:
36:    //무한 loop을 돌면서 Client 요청이 들어오면
37:    // Socket, JuryThread 생성하고 start().
38:    public void run() {
39:       while (true) {
40:          try {
41:             socket = server.accept();
42:             jury = new JuryThread(socket, db);
43:             jury.start();
44:          } catch (IOException e) {
45:             e.printStackTrace();
46:          }
47:       }
48:    }
49:
```

```
50:    public static void main(String args[]) {
51:       ProtocolHandler myHandler = new ProtocolHandler("127.0.0.1");
52:       myHandler.start();
53:    }
54:
55: } //end class
```

7.6.8. broker.Broker.java - 수정

```
1:  //import 추가
2:  import broker.Protocol;
3:  ...
4:  //Database db; --> Protocol db; 로 선언
5:  Protocol db;
6:
7:  //생성자 수정
8:     public Broker(){
9:        System.out.println("브로커 생성자 첫 라인");
10:       db =  new Protocol("127.0.0.1");
11:       System.out.println("브로커 생성자 프로토콜 객체생성");
12:    }
```