19.9 *Case Study and Lab:* Stock Exchange

In this section we consider a larger, more realistic case study: a miniature stock exchange. A stock exchange is an organization for trading shares in publicly owned companies. In the OTC (Over The Counter) system, stocks are traded electronically through a vast network of securities dealers connected to a computer network. There is no physical "stock exchange" location. In the past few years, thousands of investors have started trading stocks directly from their home computers through Internet-based online brokerage firms.

In this project we will program our own stock exchange and electronic brokerage, which we call *SafeTrade*. What do we mean by "safe"? Not too long ago, some unscrupulous online brokerage firms started encouraging a practice called "day trading," in which traders hold a stock for a few hours or even minutes rather than months or years. As a result, quite a few people lost all their savings and got into debt. Actually, this case study would be more appropriately placed in Chapter 11: in the U.S. code of bankruptcy laws,[uscode] "Chapter 11" deals with reorganization due to bankruptcy. With our *SafeTrade* program, you stay safely offline and out of trouble and don't pay commissions to a broker.

> **We picked this project because it illustrates appropriate uses of many of the interfaces and classes in the Java collections framework. This project is large enough to warrant a meaningful team development effort.**

❖ ❖ ❖

The stock exchange system keeps track of buy and sell orders placed by traders and automatically executes orders when the highest "bid" price to buy stock meets the lowest asking price for that stock. Orders to buy stock at a certain price (or lower) or to sell stock at a certain price (or higher) are called "limit" orders. There are also "market" orders: to buy at the currently offered lowest asking price or to sell at the currently offered highest bid price.

Each stock is identified by its trading symbol. For example, Sun Microsystems is "SUNW" and Microsoft is "MSFT." In the real world, very small stock prices may include fractions of cents, but in *SafeTrade* we only go to whole cents.

SafeTrade's brokerage maintains a list of registered traders and allows them to log in and trade stocks. The program keeps track of all active buy and sell orders for each

stock. A trader can place buy and sell orders, specifying the price for a "limit" order or choosing a "market" order. Each order deals with only one stock. The order for a given stock holds six pieces of information: a reference to the trader who placed it, the stock symbol, the buy or sell indicator, the desired number of shares to be bought or sold, the market or limit indicator, and the price for a limit order. *SafeTrade* acknowledges a placed order by sending a message back to the trader.

When a new order comes in, *SafeTrade* checks if it can be executed and, if so, executes the trade and reports it to both parties by sending messages to both traders. In *SafeTrade*, all orders are "partial" orders. This means that if an order cannot be executed for the total number of shares requested in it, the maximum possible number of shares changes hands and an order for the remaining shares remains active.

A trader can also request a quote for a stock. The quote includes the last sale price, the price and number of shares offered in the current highest bid and lowest "ask" (sell order), the day's high and low price for the stock, and the volume, which is the total number of shares traded during the "day." (In our model, a "day" is one run of the program.)

> **The details of how orders are executed and at what price and the format for a stock quote are described in the *Javadoc* documentation for the `Stock` class.**

SafeTrade does not keep track of the availability of money or shares on the trader's account. If you want, you can add this functionality. For example, you can keep all transactions for a given trader in a list and have a separate field to hold his or her available "cash."

At a first glance, this appears to be a pretty large project. However, it turns out that with careful planning and an understanding of the requirements, the amount of code to be written is actually relatively small. The code is simple and consists of a number of small pieces, which can be handled either by one programmer or by a team of several programmers. We have contributed the main class and a couple of GUI classes.

One of the challenges of a project like this is testing. One of the team members should specialize in QA (Quality Assurance). While other team members are writing code, the QA person should develop a comprehensive test plan. He or she then tests the program thoroughly and works with programmers on fixing bugs.

To experiment with the executable program, set up a project in your IDE with the `SafeTrade.java` and `SafeTrade.jar` files from `JM\Ch19\SafeTrade`. (Actually, `SafeTrade.jar` is a runnable jar file, so you can run *SafeTrade* by just double-clicking on `SafeTrade.jar`.)

`SafeTrade`'s `main` method creates a `StockExchange` and a `Brokerage` and opens a `LoginWindow`. To make program testing easier, `main` also lists several stocks on the `StockExchange` —

```
StockExchange exchange = new StockExchange();
server.listStock("GWP", "GridWorldProductions.com", 12.33);
server.listStock("NSTL", "Nasty Loops Inc.", 0.25);
server.listStock("GGGL", "Giggle.com", 28.00);
server.listStock("MATI", "M and A Travel Inc.", 28.20);
server.listStock("DDLC", "Dulce De Leche Corp.", 57.50);
server.listStock("SAFT", "SafeTrade.com Inc.", 322.45);
```

— and registers and logs in a couple of traders at the `Brokerage`:

```
Brokerage safeTrade = new Brokerage(exchange);
safeTrade.addUser("stockman", "sesame");
safeTrade.login("stockman", "sesame");
safeTrade.addUser("mstrade", "bigsecret");
safeTrade.login("mstrade", "bigsecret");
```

❖ ❖ ❖

Our design process for *SafeTrade* consists of four parts. The first part is structural design, which determines which data structures will be used in the program. The second part is object-oriented design, which determines the types of objects to be defined and the classes and interfaces to be written. The third part is detailed design, which determines the fields, constructors, and methods in all the classes. The fourth part is developing a test plan. We are going to discuss the structural design first, then the classes involved, and after that the detailed design and testing.

1. Structural design

Our structural design decisions are summarized in Table 19-3. We are lucky to have a chance to use many of the collections classes discussed in this chapter.

Data	*interface* => class
Registered traders	*Map* => TreeMap<String, Trader>
Logged-in traders	*Set* => TreeSet<Trader>
Mailbox for each trader	*Queue* => LinkedList<String>
Listed stocks	*Map* => HashMap<String, Stock>
"Sell" orders for each stock	*Queue* => PriorityQueue<TradeOrder> (with ascending price comparator)
"Buy" orders for each stock	*Queue* => PriorityQueue<TradeOrder> (with descending price comparator)

Table 19-3. Structural design decisions for *SafeTrade*

We have chosen to hold all registered traders in a `TreeMap`, keyed by the trader's login name. A `HashMap` could potentially work faster, but the response time for a new registration is not very important, as long as it takes seconds, not hours. A `HashMap` might waste some space, and we hope that thousands of traders will register, so we can't afford to waste any space in our database. For similar reasons, we have chosen a `TreeSet` over a `HashSet` to hold all currently logged-in traders.

A trader may log in, place a few orders, and log out. Meanwhile, *SafeTrade* may execute some of the trader's orders and send messages to the trader. But the trader may already not be there to read them. So the messages must be stored in the trader's "mailbox" until the trader logs in again and reads them. This is a perfect example of a queue. In *SafeTrade*, a mailbox for each trader is a `Queue<String>` (implemented as a `LinkedList<String>`).

SafeTrade also needs to maintain data for each listed stock. A stock is identified by its trading symbol, so it is convenient to use a map where the stock symbol serves as the key for a stock and the whole `Stock` object serves as the value. The number of all listed stocks is limited to two or three thousand, and the list does not change very often. *SafeTrade* must be able to find a stock immediately for real-time quotes and especially to execute orders. Traders will get upset if they lose money because their order was delayed. Therefore, a good choice for maintaining listed stocks is a hash table, a `HashMap`.

Finally, *SafeTrade* must store all the buy and sell orders placed for each stock in such a way that it has quick access to the highest bid and the lowest ask. Both adding and

executing orders must be fast. This is a clear case for using priority queues. We need two of them for each stock: one for sell orders and one for buy orders. For sell orders the order with the lowest asking price has the highest priority, while for buy orders the order with the highest bid price has the highest priority. Therefore, we need to write a comparator class and provide two differently configured comparator objects — one for the buy priority queue and one for the sell priority queue.

2. Object-oriented design

Figure 19-20 shows a class diagram for *SafeTrade*. The project involves nine classes and one interface. We have provided three classes and the interface: the application launcher class `SafeTrade`, the GUI classes `LoginWindow` and `TraderWindow`, and the `Login` interface. Your team's task is to write the remaining six classes. The following briefly describes the responsibilities of different types of objects.

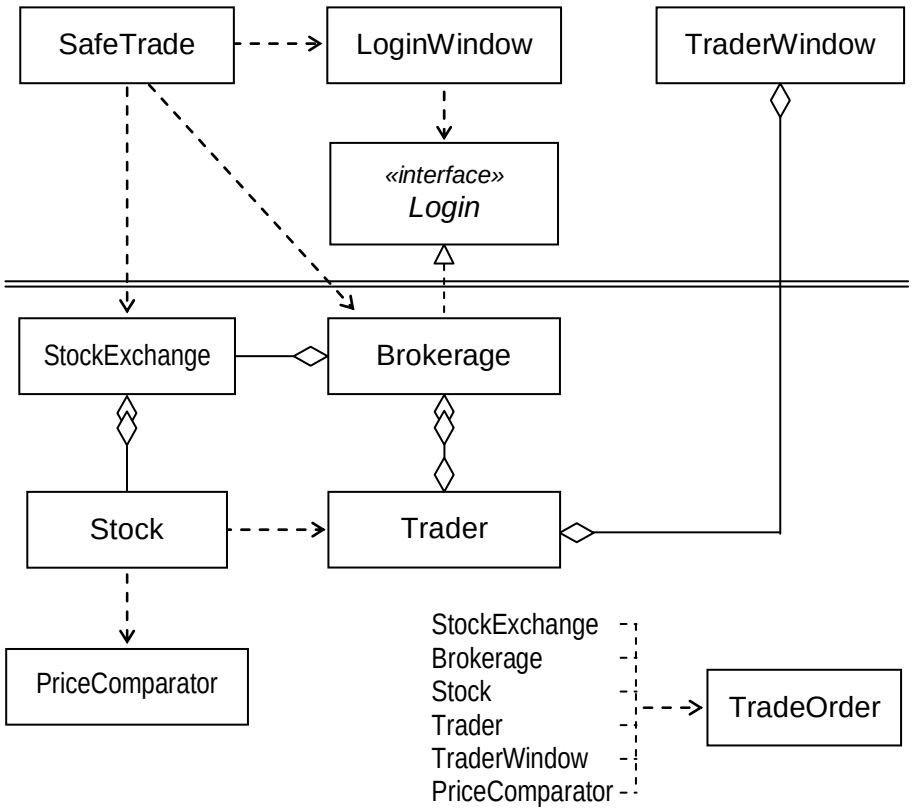

Figure 19-20. Class diagram for the *SafeTrade* program

Our classes:

We have already mentioned `SafeTrade`, the class that has `main` and starts the program.

The `LoginWindow` accepts a user name and a password from a user and can register a new trader.

> **The `Login` interface isolates `LoginWindow` from `Brokerage`, because logging in is a common function: we want to keep the `LoginWindow` class general and reusable in other projects.**

The `TraderWindow` object is a GUI front end for a `Trader` (Figure 19-21). Each `Trader` creates one for itself. The `TraderWindow` collects the data about a quote request or a trade order and passes that data to the `Trader` by calling `Trader`'s methods.

Figure 19-21. A trader window

Your classes:

The `StockExchange` keeps a `HashMap` of listed stocks, keyed by their symbols, and relays quote requests and trade orders to them.

The `Brokerage` keeps a `TreeMap` of registered traders and a `TreeSet` of logged-in traders. It receives quote requests and trade orders from traders and relays them to the `StockExchange`.

A `Stock` object holds the stock symbol, the company name, the lowest and highest sell prices, and the volume for the "day." It also has a priority queue for sell orders and another priority queue for buy orders for that stock.

> **The `Stock` class is more difficult than the other classes, because it includes an algorithm for executing orders.**

A `PriceComparator` compares two `TradeOrder` objects based on their prices; an ascending or a descending comparator is passed as a parameter to the respective priority queue of orders when the priority queue is created.

A `Trader` represents a trader; it can request quotes and place orders with the brokerage. It can also receive messages and store them in its mailbox (a `Queue<String>`) and tell its `TraderWindow` to display them.

A `TradeOrder` is a "data carrier" object used by other objects to pass the data about a trade order to each other. Since all the other classes depend on `TradeOrder`, it makes sense to write it first. This is a simple class with fields and accessor methods that correspond to the data entry fields in a `TraderWindow` (Figure 19-21).

3. Detailed design

The detailed specs for the *SafeTrade* classes have been generated from the Javadoc comments in the source files and are provided in the `SafeTradeDocs.zip` file in `JM\Ch19\SafeTrade`. Open `index.html` to see the documentation.

4. Testing

Proper testing for an application of this size is in many ways more challenging than writing the code. Entering and executing a couple of orders won't do. The QA specialist has to make a list of possible scenarios and to develop a strategy for testing them methodically. In addition, the tester must make a list of features to be tested. The most obvious are, for example: Do the "login" and "add user" screens work? Does the "Get quote" button work? When a trader logs out and then logs in again, are the messages preserved? And so on.

The `Stock` / `PriceComparator` subsystem can be tested separately. Write a simple console application for this. You will also need a *stub* class (a greatly simplified version of a class) for `Trader`, with a simple constructor, which sets the trader's name, and one method `receiveMessage(String msg)`, which prints out the trader's name and `msg`. Test the `StockExchange` class separately, using a stub class for Stock. Test the Brokerage / Trader subsystem separately using a stub class for StockExchange.

Package <Unnamed>

Interface Summary

Login	Specifies methods for registering and logging in users.

Class Summary

Brokerage	Represents a brokerage.
LoginWindow	Provides GUI for registering and logging in users.
PriceComparator	A price comparator for trade orders.
SafeTrade	The main class for the *SafeTrade* application.
Stock	Represents a stock in the SafeTrade project
StockExchange	Represents a stock exchange.
TradeOrder	Represents a buy or sell order for trading a given number of shares of a specified stock.
Trader	Represents a stock trader.
TraderWindow	Provides GUI for a trader.

Class Brokerage

```
java.lang.Object
  └ Brokerage
```

All Implemented Interfaces:

Login

```
public class Brokerage
extends java.lang.Object
implements Login
```

Represents a brokerage.

Constructor Summary

Brokerage(StockExchange exchange) Constructs new brokerage affiliated with a given stock exchange.

Method Summary

int	**addUser**(java.lang.String name, java.lang.String password) Tries to register a new trader with a given screen name and password.
void	**getQuote**(java.lang.String symbol, Trader trader) Requests a quote for a given stock from the stock exachange and passes it along to the trader by calling trader's `receiveMessage` method.
int	**login**(java.lang.String name, java.lang.String password) Tries to login a trader with a given screen name and password.
void	**logout**(Trader trader) Removes a specified trader from the set of logged-in traders.
void	**placeOrder**(TradeOrder order) Places an order at the stock exchange.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Brokerage

```
public Brokerage(StockExchange exchange)
```

Constructs new brokerage affiliated with a given stock exchange. Initializes the map of traders to an empty map (a TreeMap), keyed by trader's name; initializes the set of active (logged-in) traders to an empty set (a TreeSet).

Parameters:

`exchange` - a stock exchange.

Method Detail

addUser

```
public int addUser(java.lang.String name,
                   java.lang.String password)
```

Tries to register a new trader with a given screen name and password. If successful, creates a `Trader` object for this trader and adds this trader to the map of all traders (using the screen name as the key).

Specified by:

`addUser` in interface `Login`

Parameters:

`name` - the screen name of the trader.
`password` - the password for the trader.

Returns:

0 if successful, or an error code (a negative integer) if failed:
-1 -- invalid screen name (must be 4-10 chars)
-2 -- invalid password (must be 2-10 chars)
-3 -- the screen name is already taken.

---

login

```
public int login(java.lang.String name,
                 java.lang.String password)
```

Tries to login a trader with a given screen name and password. If no messages are waiting for the trader, sends a "Welcome to SafeTrade!" message to the trader. Opens a dialog window for the trader by calling trader's `openWindow()` method. Adds the trader to the set of all logged-in traders.

Specified by:

`login` in interface `Login`

Parameters:

`name` - the screen name of the trader.
`password` - the password for the trader.

Returns:

0 if successful, or an error code (a negative integer) if failed:
-1 -- screen name not found
-2 -- invalid password
-3 -- user is already logged in.

---

logout

```
public void logout(Trader trader)
```

Removes a specified trader from the set of logged-in traders. The trader may be assumed to logged in already.

Parameters:

`trader` - the trader that logs out.

---

getQuote

```
public void getQuote(java.lang.String symbol,
                     Trader trader)
```

Requests a quote for a given stock from the stock exachange and passes it along to the trader by calling trader's `receiveMessage` method.

Parameters:

`symbol` - the stock symbol.

`trader` - the trader who requested a quote.

---

placeOrder

```
public void placeOrder(TradeOrder order)
```

Places an order at the stock exchange.

Parameters:

`order` - an order to be placed at the stock exchange.

Interface Login

All Known Implementing Classes:

Brokerage

```
public interface Login
```

Specifies methods for registering and logging in users.

Method Summary

int	**addUser**(java.lang.String name, java.lang.String password) Tries to register a new user with a given screen name and password;
int	**login**(java.lang.String name, java.lang.String password) Tries to login a user with a given screen name and password;

Method Detail

addUser

```
int addUser(java.lang.String name,
            java.lang.String password)
```

Tries to register a new user with a given screen name and password;

Parameters:

`name` - the screen name of the user.
`password` - the password for the user.

Returns:

0 if successful, or an error code (a negative integer) if failed:
-1 -- invalid screen name (must be 4-10 chars)

-2 -- invalid password (must be 2-10 chars)
-3 -- the screen name is already taken.

---

login

```
int login(java.lang.String name,
          java.lang.String password)
```

Tries to login a user with a given screen name and password;

Parameters:

`name` - the screen name of the user.
`password` - the password for the user.

Returns:

0 if successful, or an error code (a negative integer) if failed:
-1 -- screen name not found
-2 -- invalid password
-3 -- user is already logged in.

Class LoginWindow

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Window
              └ java.awt.Frame
                  └ javax.swing.JFrame
                      └ LoginWindow
```

All Implemented Interfaces:

java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants

---

```
public class LoginWindow
extends javax.swing.JFrame
```

Provides GUI for registering and logging in users.

See Also:

Serialized Form

Nested Class Summary

Nested classes/interfaces inherited from class javax.swing.JFrame
javax.swing.JFrame.AccessibleJFrame

Nested classes/interfaces inherited from class java.awt.Frame
java.awt.Frame.AccessibleAWTFrame

Nested classes/interfaces inherited from class java.awt.Window

java.awt.Window.AccessibleAWTWindow

Nested classes/interfaces inherited from class java.awt.Container
java.awt.Container.AccessibleAWTContainer

Nested classes/interfaces inherited from class java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Field Summary

Fields inherited from class javax.swing.JFrame
accessibleContext, EXIT_ON_CLOSE, rootPane, rootPaneCheckingEnabled

Fields inherited from class java.awt.Frame
CROSSHAIR_CURSOR, DEFAULT_CURSOR, E_RESIZE_CURSOR, HAND_CURSOR, ICONIFIED, MAXIMIZED_BOTH, MAXIMIZED_HORIZ, MAXIMIZED_VERT, MOVE_CURSOR, N_RESIZE_CURSOR, NE_RESIZE_CURSOR, NORMAL, NW_RESIZE_CURSOR, S_RESIZE_CURSOR, SE_RESIZE_CURSOR, SW_RESIZE_CURSOR, TEXT_CURSOR, W_RESIZE_CURSOR, WAIT_CURSOR

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface javax.swing.WindowConstants
DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE, HIDE_ON_CLOSE

Fields inherited from interface java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

LoginWindow(java.lang.String title, Login server)
Constructs a new login window.

Method Summary

Methods inherited from class javax.swing.JFrame

addImpl, createRootPane, frameInit, getAccessibleContext, getContentPane, getDefaultCloseOperation, getGlassPane, getJMenuBar, getLayeredPane, getRootPane, isDefaultLookAndFeelDecorated, isRootPaneCheckingEnabled, paramString, processWindowEvent, remove, setContentPane, setDefaultCloseOperation, setDefaultLookAndFeelDecorated, setGlassPane, setIconImage, setJMenuBar, setLayeredPane, setLayout, setRootPane, setRootPaneCheckingEnabled, update

Methods inherited from class java.awt.Frame

addNotify, finalize, getCursorType, getExtendedState, getFrames, getIconImage, getMaximizedBounds, getMenuBar, getState, getTitle, isResizable, isUndecorated, remove, removeNotify, setCursor, setExtendedState, setMaximizedBounds, setMenuBar, setResizable, setState, setTitle, setUndecorated

Methods inherited from class java.awt.Window

addPropertyChangeListener, addPropertyChangeListener, addWindowFocusListener, addWindowListener, addWindowStateListener, applyResourceBundle, applyResourceBundle, createBufferStrategy, createBufferStrategy, dispose, getBufferStrategy, getFocusableWindowState, getFocusCycleRootAncestor, getFocusOwner, getFocusTraversalKeys, getGraphicsConfiguration, getInputContext, getListeners, getLocale, getMostRecentFocusOwner, getOwnedWindows, getOwner, getToolkit, getWarningString, getWindowFocusListeners, getWindowListeners, getWindowStateListeners, hide, isActive, isAlwaysOnTop, isFocusableWindow, isFocusCycleRoot, isFocused, isLocationByPlatform, isShowing, pack, postEvent, processEvent, processWindowFocusEvent, processWindowStateEvent, removeWindowFocusListener, removeWindowListener, removeWindowStateListener, setAlwaysOnTop, setBounds, setCursor, setFocusableWindowState, setFocusCycleRoot, setLocationByPlatform, setLocationRelativeTo, show, toBack, toFront

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalPolicy, getInsets, getLayout, getMaximumSize, getMinimumSize, getMousePosition, getPreferredSize, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paint, paintComponents, preferredSize, print, printComponents, processContainerEvent, remove, removeAll, removeContainerListener, setComponentZOrder, setFocusTraversalKeys, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, setFont, transferFocusBackward, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains, contains, createImage, createImage, createVolatileImage, createVolatileImage, disable, disableEvents, dispatchEvent, enable, enable, enableEvents, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getBackground, getBounds, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusListeners,

getFocusTraversalKeysEnabled, getFont, getFontMetrics, getForeground, getGraphics, getHeight, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocation, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getSize, getTreeLock, getWidth, getX, getY, gotFocus, handleEvent, hasFocus, imageUpdate, inside, isBackgroundSet, isCursorSet, isDisplayable, isDoubleBuffered, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isMaximumSizeSet, isMinimumSizeSet, isOpaque, isPreferredSizeSet, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, prepareImage, prepareImage, printAll, processComponentEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent, processInputMethodEvent, processKeyEvent, processMouseEvent, processMouseMotionEvent, processMouseWheelEvent, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, repaint, requestFocus, requestFocus, requestFocusInWindow, requestFocusInWindow, reshape, resize, resize, setBackground, setBounds, setComponentOrientation, setDropTarget, setEnabled, setFocusable, setFocusTraversalKeysEnabled, setForeground, setIgnoreRepaint, setLocale, setLocation, setLocation, setMaximumSize, setMinimumSize, setName, setPreferredSize, setSize, setSize, setVisible, show, size, toString, transferFocus, transferFocusUpCycle

Methods inherited from class java.lang.Object

clone, equals, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.awt.MenuContainer

getFont, postEvent

Constructor Detail

LoginWindow

```
public LoginWindow(java.lang.String title,
                   Login server)
```

Constructs a new login window.

Parameters:

`title` - title bar text.
`server` - an object that keeps track of all the registered and logged-in users.

Class PriceComparator

```
java.lang.Object
  └ PriceComparator
```

All Implemented Interfaces:

java.util.Comparator<TradeOrder>

```
public class PriceComparator
extends java.lang.Object
implements java.util.Comparator<TradeOrder>
```

A price comparator for trade orders.

Constructor Summary

PriceComparator() Constructs a price comparator that compares two orders in ascending order.
PriceComparator(boolean asc) Constructs a price comparator that compares two orders in ascending or descending order.

Method Summary

int	**compare**(TradeOrder order1, TradeOrder order2) Compares two trade orders.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from interface java.util.Comparator
equals

Constructor Detail

PriceComparator

```
public PriceComparator()
```

Constructs a price comparator that compares two orders in ascending order. Sets the private `boolean ascending` flag to `true`.

PriceComparator

```
public PriceComparator(boolean asc)
```

Constructs a price comparator that compares two orders in ascending or descending order. The order of comparison depends on the value of a given parameter. Sets the private `boolean ascending` flag to `asc`.

Parameters:

`asc` - if true, make an ascending comparator; otherwise make a descending comparator.

Method Detail

compare

```
public int compare(TradeOrder order1,
                   TradeOrder order2)
```

Compares two trade orders.

Specified by:

`compare` in interface `java.util.Comparator<TradeOrder>`

Parameters:

`order1` - the first order
`order2` - the second order

Returns:

0: if both orders are market orders.
-1: if `order1` is market and `order2` is limit.
1: if `order1` is limit and `order2` is market.
The difference in prices in cents: if both `order1` and `order2` are limit orders. In this last case, the difference returned is cents1 - cents2 or

cents2 - cents1, depending on whether this is an ascending or descending comparator (`ascending` is `true` or `false`). You will need to **round** your answer otherwise floating point conversion to integer can cause errors. Be careful when rounding positive and negative values...

Class TradeOrder

```
java.lang.Object
  └ TradeOrder
```

```
public class TradeOrder
extends java.lang.Object
```

Represents a buy or sell order for trading a given number of shares of a specified stock.

Constructor Summary

TradeOrder(Trader trader, java.lang.String symbol, boolean buyOrder, boolean marketOrder, int numShares, double price) Constructs a new TradeOrder for a given trader, stock symbol, a number of shares, and other parameters.

Method Summary

double	**getPrice**() Returns the price per share for this trade order (used by a limit order).
int	**getShares**() Returns the number of shares to be traded in this trade order.
java.lang.String	**getSymbol**() Returns the stock symbol for this trade order.
Trader	**getTrader**() Returns the trader for this trade order.
boolean	**isBuy**() Returns true if this is a buy order; otherwise returns false.
boolean	**isLimit**() Returns true if this is a limit order; otherwise returns false.
boolean	**isMarket**() Returns true if this is a market order; otherwise returns false.

Class SafeTrade

```
java.lang.Object
  └ SafeTrade
```

```
public class SafeTrade
extends java.lang.Object
```

The main class for the *SafeTrade* application.

Constructor Summary

SafeTrade()

Method Summary

static void	main(java.lang.String[] args)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

SafeTrade

```
public SafeTrade()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
```

Class Stock

```
java.lang.Object
  └ Stock
```

```
public class Stock
extends java.lang.Object
```

Represents a stock in the SafeTrade project

Field Summary

static java.text.DecimalFormat	**money**

Constructor Summary

Stock(java.lang.String symbol, java.lang.String name, double price) Constructs a new stock with a given symbol, company name, and starting price.

Method Summary

protected void	**executeOrders**() Executes as many pending orders as possible.
java.lang.String	**getQuote**() Returns a quote string for this stock.
void	**placeOrder**(TradeOrder order) Places a trading order for this stock.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

money

```
public static java.text.DecimalFormat money
```

Constructor Detail

Stock

```
public Stock(java.lang.String symbol,
             java.lang.String name,
             double price)
```

Constructs a new stock with a given symbol, company name, and starting price. Sets low price, high price, and last price to the same opening price. Sets "day" volume to zero. Initializes a priority qieue for sell orders to an empty `PriorityQueue` with a `PriceComparator` configured for comparing orders in ascending order; initializes a priority qieue for buy orders to an empty `PriorityQueue` with a `PriceComparator` configured for comparing orders in descending order.

Parameters:

`symbol` - the stock symbol.
`name` - full company name.
`price` - opening price for this stock.

Method Detail

getQuote

```
public java.lang.String getQuote()
```

Returns a quote string for this stock. The quote includes: the company name for this stock; the stock symbol; last sale price; the lowest and highest day prices; the lowest price in a sell order (or "market") and the number of shares in it (or "none" if there are no sell orders); the highest price in a buy order (or "market") and the number of shares in it (or "none" if there are no buy orders). For example:

```
Giggle.com (GGGL)
Price: 10.00  hi: 10.00  lo: 10.00  vol: 0
Ask: 12.75 size: 300  Bid: 12.00 size: 500
```

Or:

```
Giggle.com (GGGL)
Price: 12.00  hi: 14.50  lo: 9.00  vol: 500
Ask: none  Bid: 12.50 size: 200
```

Returns:
the quote for this stock.

---

placeOrder

```
public void placeOrder(TradeOrder order)
```

Places a trading order for this stock. Adds the order to the appropriate priority queue depending on whether this is a buy or sell order. Notifies the trader who placed the order that the order has been placed, by sending a message to that trader. For example:

```
New order:  Buy GGGL (Giggle.com)
200 shares at $38.00
```

Or, for market orders:

```
New order:  Sell GGGL (Giggle.com)
150 shares at market
```

Executes pending orders by calling `executeOrders`.

Parameters:
`order` - a trading order to be placed.

---

executeOrders

```
protected void executeOrders()
```

Executes as many pending orders as possible.

1. Examines the top sell order and the top buy order in the respective priority queues.
    i. If both are limit orders and the buy order price is greater or equal to the sell order price, continue to step 2 to process the order (or a part of it) at the sell order price.
    ii. If one order is limit and the other is market, continue to step 2 to process the order (or a part of it) at the limit order price
    iii. If both orders are market, continue to step 2 to process the the order (or a part of it) at the last sale price.
    iv. If none of the above (i.e. both were limit orders and the buy order price is less than the sell order price) then no orders can be executed.
2. Figures out how many shares can be traded, which is the smallest of the numbers of shares in the two orders.

3. Subtracts the traded number of shares from each order; Removes each of the orders with 0 remaining shares from the respective queue.
4. Updates the day's low price, high price, and volume.
5. Sends a message to each of the two traders involved in the transaction. For example:

```
You bought: 150 GGGL at 38.00 amt 5700.00
```

   Note: The dollar amounts should be formatted to two decimal places (eg. 12.40, not 12.4)

6. Repeats steps 1-5 for as long as possible, that is as long as there is any movement in the buy / sell order queues. (The process gets stuck when the top buy order and sell order are both limit orders and the ask price is higher than the bid price.)

Class StockExchange

```
java.lang.Object
  └ StockExchange
```

```
public class StockExchange
extends java.lang.Object
```

Represents a stock exchange. A `StockExchange` keeps a `HashMap` of stocks, keyed by a stock symbol. It has methods to list a new stock, request a quote for a given stock symbol, and to place a specified trade order.

Constructor Summary

`StockExchange``()` Constructs a new stock exchange object.

Method Summary

`java.lang.String`	**`getQuote`**`(java.lang.String symbol)` Returns a quote for a given stock.
`void`	**`listStock`**`(java.lang.String symbol, java.lang.String name, double price)` Adds a new stock with given parameters to the listed stocks.
`void`	**`placeOrder`**`(TradeOrder order)` Places a trade order by calling `stock.placeOrder` for the stock specified by the stock symbol in the trade order.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Class Trader

```
java.lang.Object
  └ Trader
```

All Implemented Interfaces:
java.lang.Comparable<Trader>

```
public class Trader
extends java.lang.Object
implements java.lang.Comparable<Trader>
```

Represents a stock trader.

Constructor Summary

Trader(Brokerage brokerage, java.lang.String name, java.lang.String pswd) Constructs a new trader, affiliated with a given brockerage, with a given screen name and password.

Method Summary

int	**compareTo**(Trader other) Compares this trader to another by comparing their screen names case blind.
boolean	**equals**(java.lang.Object other) Indicates whether some other trader is "equal to" this one, based on comparing their screen names case blind.
java.lang.String	**getName**() Returns the screen name for this trader.
java.lang.String	**getPassword**() Returns the password for this trader.
void	**getQuote**(java.lang.String symbol) Requests a quote for a given stock symbol from the brokerage by calling brokerage's getQuote.

Class TraderWindow

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Window
              └ java.awt.Frame
                  └ javax.swing.JFrame
                      └ TraderWindow
```

All Implemented Interfaces:

java.awt.event.ActionListener, java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, java.util.EventListener, javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants

---

```
public class TraderWindow
extends javax.swing.JFrame
implements java.awt.event.ActionListener
```

Provides GUI for a trader.

See Also:

Serialized Form

Nested Class Summary

Nested classes/interfaces inherited from class javax.swing.JFrame
javax.swing.JFrame.AccessibleJFrame

Nested classes/interfaces inherited from class java.awt.Frame
java.awt.Frame.AccessibleAWTFrame

Nested classes/interfaces inherited from class java.awt.Window
java.awt.Window.AccessibleAWTWindow

Nested classes/interfaces inherited from class java.awt.Container
java.awt.Container.AccessibleAWTContainer

Nested classes/interfaces inherited from class java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Field Summary

Fields inherited from class javax.swing.JFrame
accessibleContext, EXIT_ON_CLOSE, rootPane, rootPaneCheckingEnabled

Fields inherited from class java.awt.Frame
CROSSHAIR_CURSOR, DEFAULT_CURSOR, E_RESIZE_CURSOR, HAND_CURSOR, ICONIFIED, MAXIMIZED_BOTH, MAXIMIZED_HORIZ, MAXIMIZED_VERT, MOVE_CURSOR, N_RESIZE_CURSOR, NE_RESIZE_CURSOR, NORMAL, NW_RESIZE_CURSOR, S_RESIZE_CURSOR, SE_RESIZE_CURSOR, SW_RESIZE_CURSOR, TEXT_CURSOR, W_RESIZE_CURSOR, WAIT_CURSOR

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface javax.swing.WindowConstants
DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE, HIDE_ON_CLOSE

Fields inherited from interface java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

TraderWindow(Trader trader)
Constructs a new trading window for a trader.

Method Summary

void	**actionPerformed**(java.awt.event.ActionEvent e) Processes GUI events in this window.
void	**showMessage**(java.lang.String msg) Displays a message in this window's text area.

Methods inherited from class javax.swing.JFrame

addImpl, createRootPane, frameInit, getAccessibleContext, getContentPane, getDefaultCloseOperation, getGlassPane, getJMenuBar, getLayeredPane, getRootPane, isDefaultLookAndFeelDecorated, isRootPaneCheckingEnabled, paramString, processWindowEvent, remove, setContentPane, setDefaultCloseOperation, setDefaultLookAndFeelDecorated, setGlassPane, setIconImage, setJMenuBar, setLayeredPane, setLayout, setRootPane, setRootPaneCheckingEnabled, update

Methods inherited from class java.awt.Frame

addNotify, finalize, getCursorType, getExtendedState, getFrames, getIconImage, getMaximizedBounds, getMenuBar, getState, getTitle, isResizable, isUndecorated, remove, removeNotify, setCursor, setExtendedState, setMaximizedBounds, setMenuBar, setResizable, setState, setTitle, setUndecorated

Methods inherited from class java.awt.Window

addPropertyChangeListener, addPropertyChangeListener, addWindowFocusListener, addWindowListener, addWindowStateListener, applyResourceBundle, applyResourceBundle, createBufferStrategy, createBufferStrategy, dispose, getBufferStrategy, getFocusableWindowState, getFocusCycleRootAncestor, getFocusOwner, getFocusTraversalKeys, getGraphicsConfiguration, getInputContext, getListeners, getLocale, getMostRecentFocusOwner, getOwnedWindows, getOwner, getToolkit, getWarningString, getWindowFocusListeners, getWindowListeners, getWindowStateListeners, hide, isActive, isAlwaysOnTop, isFocusableWindow, isFocusCycleRoot, isFocused, isLocationByPlatform, isShowing, pack, postEvent, processEvent, processWindowFocusEvent, processWindowStateEvent, removeWindowFocusListener, removeWindowListener, removeWindowStateListener, setAlwaysOnTop, setBounds, setCursor, setFocusableWindowState, setFocusCycleRoot, setLocationByPlatform, setLocationRelativeTo, show, toBack, toFront

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalPolicy, getInsets, getLayout, getMaximumSize, getMinimumSize, getMousePosition, getPreferredSize, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paint, paintComponents, preferredSize, print, printComponents, processContainerEvent, remove, removeAll, removeContainerListener, setComponentZOrder, setFocusTraversalKeys, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, setFont, transferFocusBackward, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains, contains, createImage, createImage, createVolatileImage, createVolatileImage, disable, disableEvents, dispatchEvent, enable, enable, enableEvents, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getBackground, getBounds, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getFontMetrics, getForeground, getGraphics, getHeight, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocation, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getSize, getTreeLock, getWidth, getX, getY, gotFocus, handleEvent, hasFocus, imageUpdate, inside, isBackgroundSet, isCursorSet, isDisplayable, isDoubleBuffered, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isMaximumSizeSet, isMinimumSizeSet, isOpaque, isPreferredSizeSet, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, prepareImage, prepareImage, printAll, processComponentEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent, processInputMethodEvent, processKeyEvent, processMouseEvent, processMouseMotionEvent, processMouseWheelEvent, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, repaint, requestFocus, requestFocus, requestFocusInWindow, requestFocusInWindow, reshape, resize, resize, setBackground, setBounds, setComponentOrientation, setDropTarget, setEnabled, setFocusable, setFocusTraversalKeysEnabled, setForeground, setIgnoreRepaint, setLocale, setLocation, setLocation, setMaximumSize, setMinimumSize, setName, setPreferredSize, setSize, setSize, setVisible, show, size, toString, transferFocus, transferFocusUpCycle

Methods inherited from class java.lang.Object

clone, equals, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.awt.MenuContainer

getFont, postEvent

Constructor Detail

TraderWindow

public **TraderWindow**(Trader trader)

Constructs a new trading window for a trader.

Parameters:

`trader` - a trader that will own this window.

Method Detail

showMessage

```
public void showMessage(java.lang.String msg)
```

Displays a message in this window's text area.

Parameters:

`msg` - the message to be displayed.

---

actionPerformed

```
public void actionPerformed(java.awt.event.ActionEvent e)
```

Processes GUI events in this window.

Specified by:

`actionPerformed` in interface `java.awt.event.ActionListener`

Parameters:

`e` - an event.
