A Simple On-line Shopping System using Java's RMI

Java Remote Method Invocation (RMI) is used to implement the default project, which is a simulation of one client, two stores, and a bank that will be created to fit the given specifications. The client, stores, and bank each naturally lend themselves to being distinct object classes. Methods of these classes can be called remotely using RMI when the objects must communicate. The program must take into account issues of message passing in a distributed environment, such as messages being received out of order and synchronizing communication between several objects concurrently.

These are important issues in operating systems because any given distributed system must have stable communication between its parts. The components must know whom the messages are coming from, taking into account possible fraud (security issues). The layers above the OS rely on messages being passed reliably and in the proper order so that expected results can be achieved. Some task concurrency may play a role so that the client can shop other stores while waiting for results on a previous purchase. This is a useful concept in operating systems so that multiple programs can operate (seemingly) at once to fulfill the user's needs. Lastly, tying all these ideas together is important because an OS is large, with all of these tasks running at once, and we have to know how to integrate them seamlessly.

My learning objectives throughout this project were to become more familiar with Java, to learn the RMI system, and to become a better thinker on topics of distributed and concurrent computing.
**Project Description**

The abstract design of this project for Java was straightforward: each component in the shopping simulation would become its own class. These are the stores, the bank, and the client. Objects of these types could be instantiated as needed and would each run independently in its own space. This allows for multiple clients, stores, and banks, as needed.

Different objects are never assumed to be on the same physical machine. RMI is used whenever two objects need to communicate. This requires that the "RMI Registry" is running on each machine with objects that can be invoked externally. The objects that want to make themselves available for remote invocation simply export themselves to that registry, where remote callers can look up their methods. This is done on a specific port (I chose 8172 arbitrarily but to avoid the default RMI port where other programs may be listening) that must be known by both the local and remote programs.

In this implementation, only one client, one bank, and two stores are created. Extending these is a simple matter and is discussed in the section for each class below.

**The Client**

The user interface is a simple, text-menu-driven environment. Upon startup the client immediately contacts the bank to retrieve the user's account balance. This is to tell the user how much is available for purchases and to do local checking on whether a purchase costs too much (this check for enough money is also done by the bank during each purchase, as I'll discuss later).

The client's bank account number is hard-coded for the sake of simplicity of demonstration. To extend the system to support multiple users, the interface could request this account number from the user at startup, and retain it only for the duration of the program. However, the problem remains of having multiple clients on the same machine. To support this, a "key" can be created to identify the particular instance of the client software running on the local machine. This could be based on the process ID for example, to guarantee that remote objects message the particular client they intended.

After the user retrieves an account balance, a menu of two stores is presented. Choosing either gives the choice of viewing or buying items. Viewing simply contacts the store and has it return an array of ten objects of type "Item." An Item is simply an object name and its cost. The client displays these items along with a number besides each, which can be used to identify the item during a purchase.

Buying an item is the most involved procedure. First the items are "viewed" as above. The user is asked for the number of the item to buy and the client program declines the request if the user's money has run too low. Otherwise the client remembers this purchase in a vector, and the store's "buy" method is invoked with the item to buy, the user's bank account number, and the client's Callback abstraction. "Callback" is the class
that the client descends from, and it defines a "verifyPurchase" method that the bank will use to get back in touch with it later.

The store accepts the request and proceeds to check on the purchase by asking the bank if it is okay. It passes to the bank the client information (account number and customer Callback), item name and cost, and the store's name. If the bank sees the client has enough money, it contacts the customer by invoking the Callback's "verifyPurchase." This asks the client to verify that the item name and cost was really purchased at the named store. Since the client stored all of its purchases in a vector, it can do this check (removing items once okayed) and tell the bank whether or not to grant permission for the purchase.

The bank at last has enough information to return to the store whether the purchase was granted. The client need not be contacted again, since the store already has permission from the bank to make the charge.

It is worth noting that the client need not wait around for the bank to contact it. The user can go off to another store to make purchases, and since it has exported its definition to the RMI Registry, the "verifyPurchase" method is sitting around waiting to be executed. The invocation is made whenever the bank is ready. This matches the need for concurrency in the client program.

The Store

Most of the store's operation has been detailed already. It answers remote calls to return its list of items for sale, as well as handling the aforementioned purchases. All stores inherit from the "Store" interface and thus implement the remote methods "view," "buy," and "getName" (which simply returns the store's name as a string).

The store's design is kept flexible via object inheritance. The StoreImpl ("store implementation") class does the work of the above operations. Specific stores inherent from StoreImpl and then give themselves a name and define their own items for sale. (For example, the class "BobsStore" would descend from the generic StoreImpl and would simply change its name to "Bob's Hardware" and define its item list to include "Nails" for $15, "Hammers" for $100, etc.)

For simplicity, stores were limited to ten or fewer items each, but changing this for a real store would be a simple matter of increasing the array size throughout the class files.

The Bank

The simplest of the major classes, "BankImpl" implements the Bank interface that has only two remote methods: getFunds and validate. getFunds is passed an account number by the client and returns the balance on that account. "validate" is used by stores to query the bank about whether a purchase can be completed, as discussed under the client heading.
The main concern for the bank, and one not addressed in this program due to time constraints, is that of security. The client wants reassurance that his credit card number is not intercepted, so some form of encryption should be used in all transfers of personal information. A second safety check on the card such as a password or card expiration date would also be expected of this application.

**Class Diagrams**

These are all of the objects in the program listing their private and public methods. Arrows labeled "extends" imply that one class inherits from another.

The following page shows the program flow through a purchase. The messages passed between objects are shown as arrows from one column, depicting an object's operations, to another object's column.
Customer C

- C requests funds
- View Items
  - C requests Items
- Print 10 Items
  - Store S
    - view()
      - S returns array of this store's 10 Items
      - 10 Items
- Buy Items
  - Perform steps under "View items"
- C requests Items
  - Got Item list
  - Ask user which Item to buy
    - Deny if too little cash (done on client side)
  - Else buy
    - C returns true if made this purchase, else false.
      - If true, deduct cash.

Store S

- bank.getFunds()
- int funds

Bank B

- Look up account # and return its funds
- bank.validate(store name, account #, item, cost, Callback for Customer)
- C returns true if made this purchase, else false.
  - If true then deduct cost from C's account & return true to S
  - TRUE or FALSE
  - If True then purchase is complete. Else do not make it.
  - TRUE or FALSE
  - Customer Callback (store name, item name, cost)
  - Verify that account exists & has enough credit
  - Verify C made this purchase
Conclusions

This project met my goals for developing my thinking about remote computing and concurrency. I found myself considering "what if this computer failed or became unreachable?" for each component in the system. This led to additional error checking and gave relevance to Java's demands for "catching" or "throwing" all exceptions.

Concurrency is a necessary tool with today's multitasking OS's so it was no stretch of necessity that demanded its inclusion in this program. Every decent Web browser offers multi-window shopping. This turned out to be a snap using RMI because the concurrent execution of methods we've made available via the registry are run behind-the-scenes. The only consideration is that this method does not change variables we are working with elsewhere in the program at the time it is called. This is overcome by placing locks on any variables being changed that could be in use elsewhere in the concurrently executing program.

This project lent itself well to object-oriented programming, and the usual networking headaches were kept minimal due to RMI handling so many network details. With an improved user interface, secure transactions, and a few quick changes to support more and larger stores, this could be a practical set of programs for on-line shopping.