SE 424: Distributed Systems **Recitation 3b**
Semester 1 5785 **18 Nov 2024**
Lecturer: Michael J. May **Kinneret College**

RMI Remote Banking Tool

In this recitation we'll develop a more interesting use of RMI using two levels of remote objects: a main remote object which acts as an index for a set of other remote objects. As part of this example, we will introduce two new concepts for using RMI:

- The use of a gateway object to send pointers to other objects
- Transferring objects via remote method calls.

The application is a remote banking service in which the client can connect to the server, list the accounts available, and perform actions on the accounts such as deposit, withdrawal, and printing balance.

The arrangement of the parts is sketched out in Figure 1.

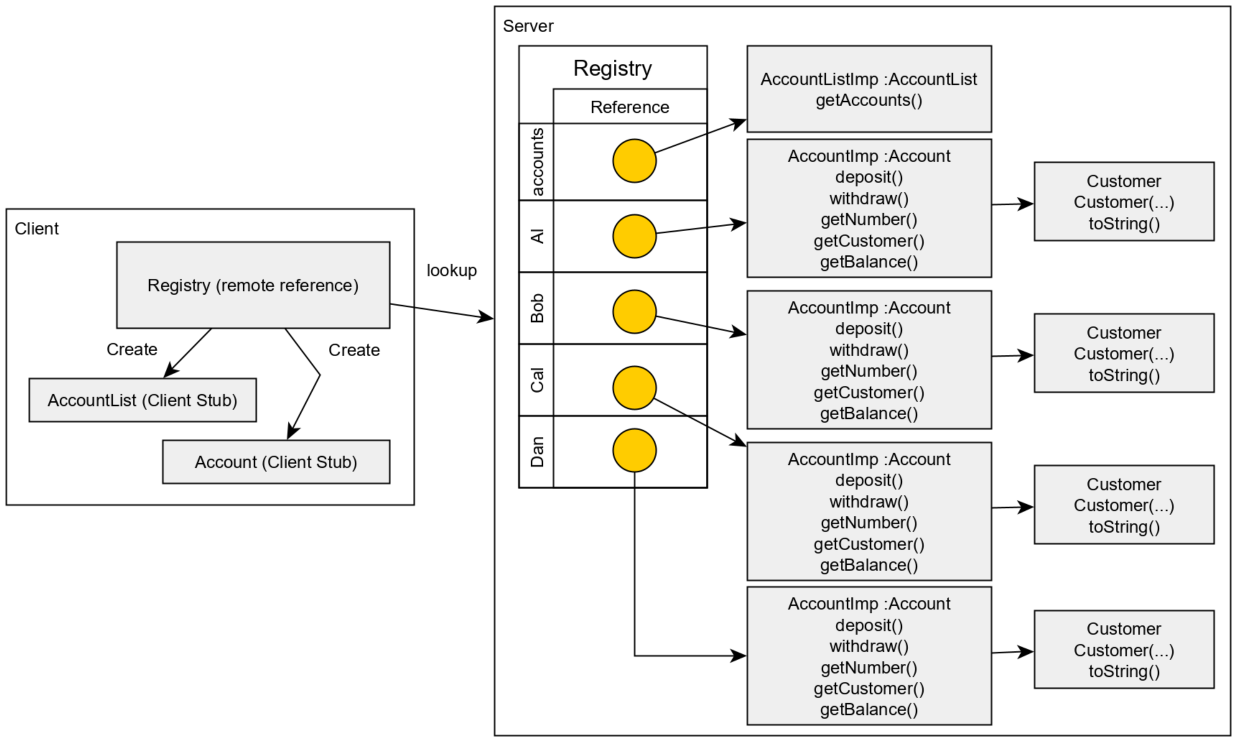


Figure 1: Bank Server and Client sketch

1 Class: Customer

A `Customer` object contains information about the customer who owns the bank account. The class implements `Serializable` so it can be sent over the network. This is required to enable the passing of objects via remote function calls.

1.1 Member data

The Customer class has the following fields:

```
private String tz, name, address, city;
```

1.2 Methods

The class implements standard accessor and mutator methods for the fields.

It has two constructors, one for building the object from fields and a copy constructor for convenience when creating accounts.

To help with printing, it has a toString() method to print out the customer details.

The constructors and toString method are shown below.

```
public Customer (String tz, String n, String add, String c)
{
 this.name = n;
 this.tz = tz;
 this.address = add;
 this.city = c;
}

public Customer (Customer c)
{
 this.name = c.name;
 this.tz = c.tz;
 this.address = c.address;
 this.city = c.city;
}

public String toString()
{
 return name + " (" + tz + ") " + address + ", " + city;
}
```

2 Interface: Account

There `Account` interface defines an interface to objects that include information about each bank account. The bank account includes a `Customer` object, the account number, and the current balance. The `Account` interface extends `Remote` and exports the following functions:

- `boolean deposit (double amount) throws RemoteException;`
- `boolean withdraw (double amount) throws RemoteException;`
- `int getNumber () throws RemoteException;`
- `Customer getCustomer() throws RemoteException;`
- `double getBalance() throws RemoteException;`

3 Class: AccountImp

The implementation of the interface is called `AccountImp` and must extend `UnicastRemoteObject`. It offers the following constructor:

```
public AccountImp(Customer c, int num, double amount)
{
 this.customer = new Customer(c);
 this.accountNum = num;
 this.balance = amount;
}
```

In addition to the constructor, AccountImp must offer implementations of the methods that Account defines. The methods must do the following:

deposit Add money to the account. The amount is provided as a parameter. Returns true if the deposit worked. False otherwise.

withdraw Remove money from the account. The balance of the account may not go below 0 due to the withdrawal. The amount is provided as a parameter. Returns true if the withdrawal worked. False otherwise.

getNumber Returns the account number.

getCustomer Returns the Customer object for the account.

getBalance Returns the current balance of the account

4 Interface: AccountList

Since there may be many bank accounts, the bank publishes a gateway object that allows the client to see what accounts are available. We will create an interface called `AccountList` that publishes the following method:

- `ArrayList<Integer> getAccounts() throws RemoteException`

The interface must extend the `Remote` interface.

5 Class: AccountListImp

We will write a class `AccountListImp` that extends `UnicastRemoteObject` and implements `AccountList`.

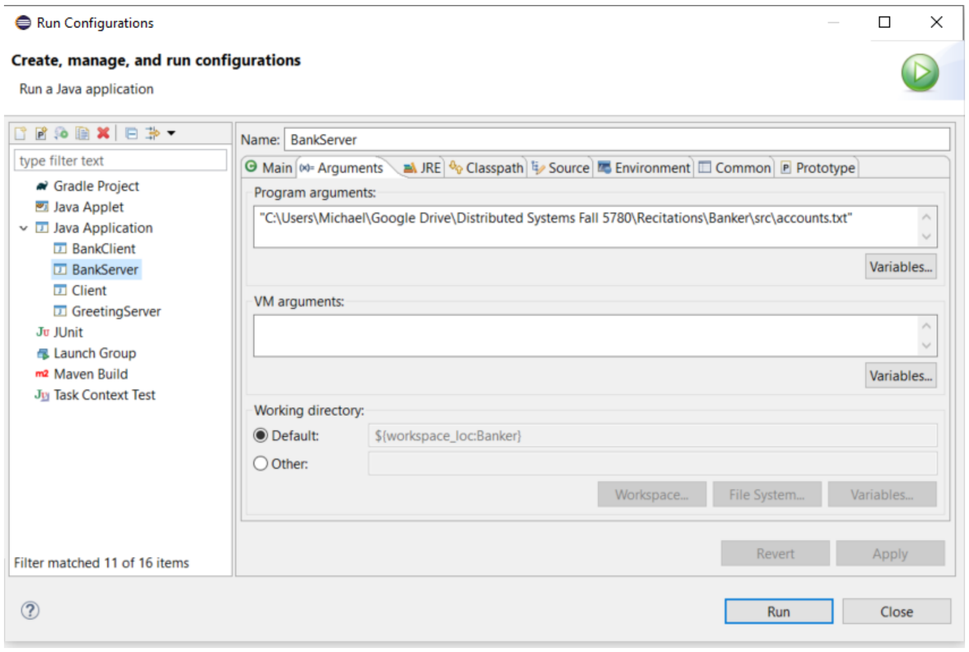
5.1 Member data

The class must have an internal ArrayList that contains the account numbers of each account in the list.

5.2 Methods

The class must offer the following methods:

- AccountListImp: constructor for the list.
- getAccounts: Returns an ArrayList of the account numbers
- addAccount(int): Add a new account number to the list



Name: BankServer

Program arguments:

```
"C:\Users\Michael\Google Drive\Distributed Systems Fall 5780\Recitations\Banker\src\accounts.txt"
```

VM arguments:

Working directory:
[✓ Default:] ${workspace_loc:Banker}
Other:

Figure 2: Bank Server Parameters

6 Class: TheServer

The remote server loads accounts from a file passed to it as part of its program parameters (see Figure 2). The bank accounts are formatted as follows:

[tz];[name];[address];[city];[accountNumber];[initialBalance]

A sample configuration file is:

```
01234;Al;Habanim 1234;Tiberias;456;0
04567;Bob;Habanim 4567;Tiberias;900;10
89999;Cal;Hagalil 1;Tiberias;789;7
90000;Dan;Hamigdal 2;Afula;100;100
```

Each bank account is instantiated from the file. The server then publishes the AccountListImp object at the `accounts` name as well as each account using its name (*e.g.* account 4567 is registered at name `4567` and account 4568 is registered at name `4568`). Based on the sample configuration file above, the server will publish the following objects:

accounts The `AccountListImp` object with the list of all account numbers available. Publish it using the `AccountList` interface.

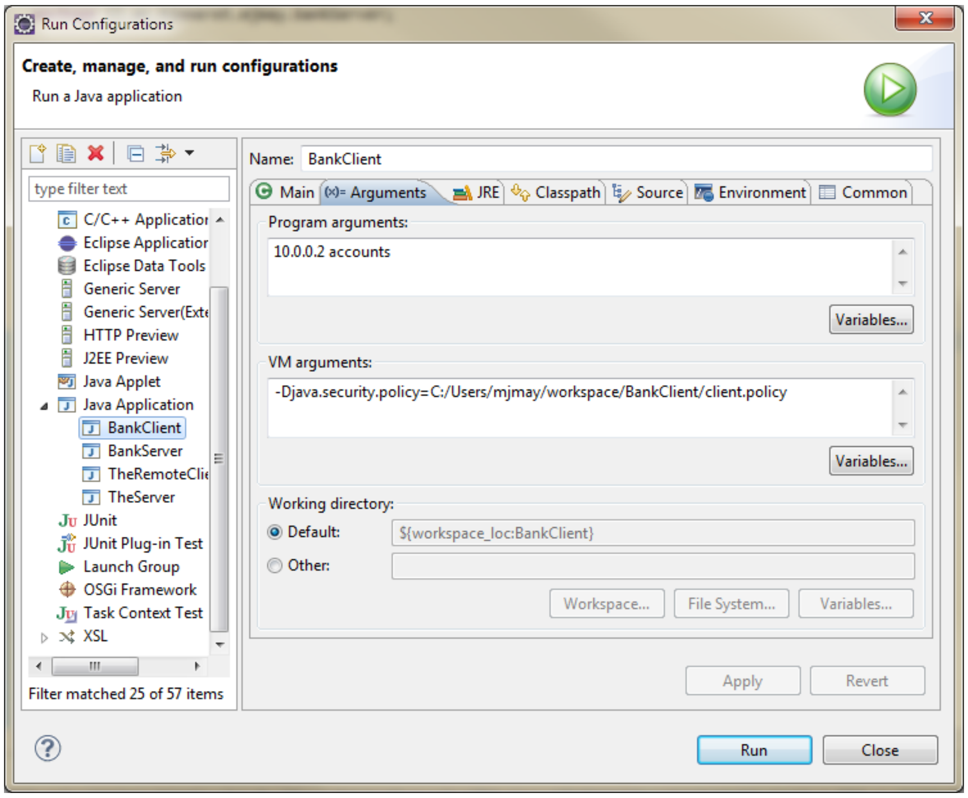
1234 The `AccountImp` object for bank account 1234. Publish it using the `Account` interface.

4567 The `AccountImp` object for bank account 4567. Publish it using the `Account` interface.

89999 The `AccountImp` object for bank account 89999. Publish it using the `Account` interface.

90000 The `AccountImp` object for bank account 90000. Publish it using the `Account` interface.

All of the objects must be published using the `Registry.rebind()` method.



Name: BankClient

Program arguments:

```
10.0.0.2 accounts
```

VM arguments:

```
-Djava.security.policy=C:/Users/mjmay/workspace/BankClient/client.policy
```

Working directory: [✓ Default:] ${workspace_loc:BankClient} Other:

Figure 3: Bank Client Parameters

7 Client Details

The client program behaves as follows:

1. It accepts the IP address of the remote host and the name of the remote `AccoutList` object in its program parameters as shown in Figure 3.
2. It retrieves the account list and shows the list of the accounts available (just the account numbers).
3. Once it shows the accounts available, it allows the user to enter an account number from the list to "log in" to that account.
4. It then retrieves the account object selected and enables the user to perform the following operations:

   Current Balance Shows the current balance.

   Customer Details Prints customer details.

   Deposit It allows the user to submit a non-negative real number to deposit. It shows a success or failure message after attempting to perform the deposit. It prints the balance before and after the operation.

   Withdraw It allows the user to submit a non-negative real number to withdraw. It shows a success or failure message after attempting to perform the withdraw. It prints the balance before and after the operation.

   Account Number Prints the current account number.

5. The user can switch to another account if he wants or quit.

A trace from the client is shown in Figure 4.

8 What to do

1. Build the server as described in the document here.
2. Build the client as described in the document here.

8.1 Experiments

When you have successfully written the code for the assignment, perform the following experiments:

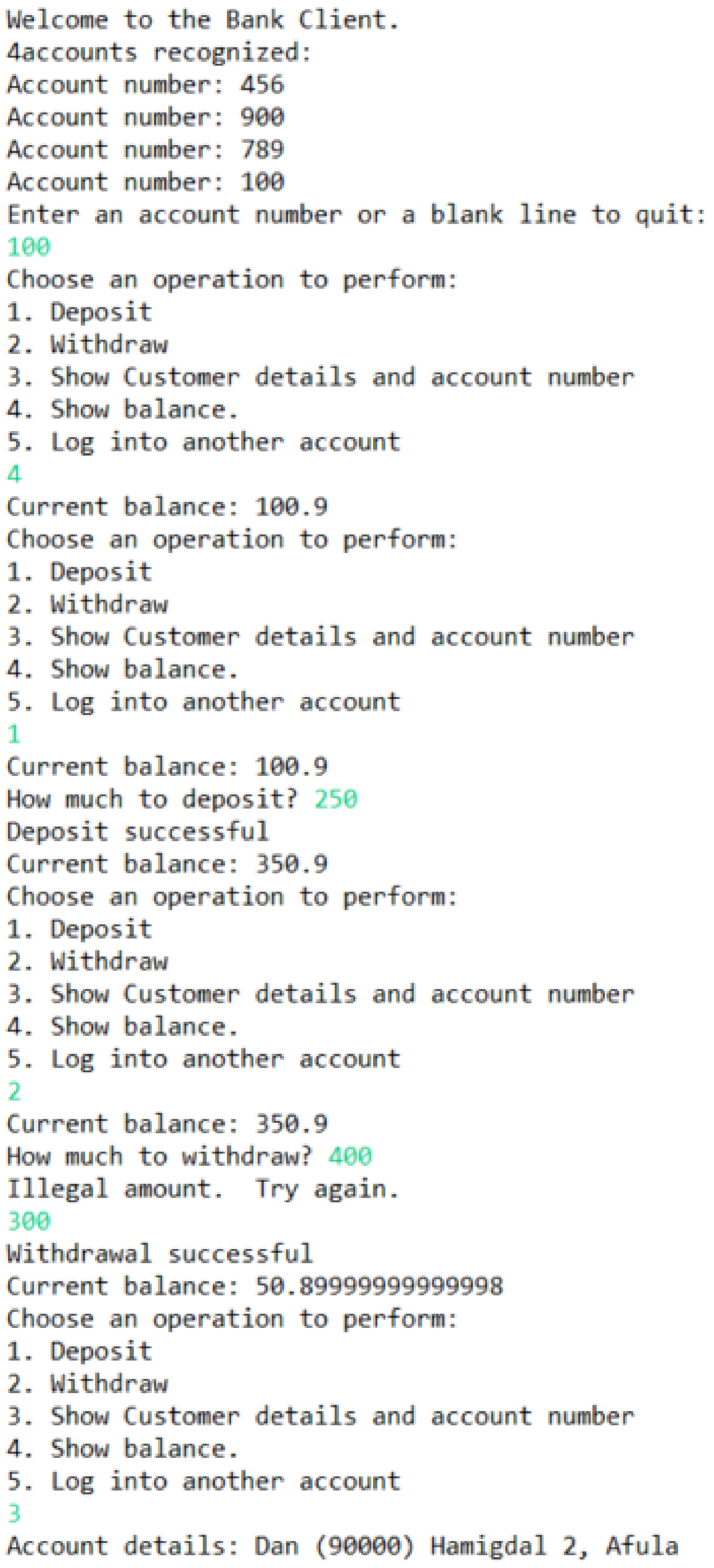
(a) Log in to the same account from two different computers simultaneously. Are the changes made by one machine visible immediately to the other machine? Can you tell what kind of concurrency control is being used?

(b) Add a function at the client that retrieves the `Customer` information from an account and allows the user to modify it. Do the modifications affect the copy stored on the server? What does that tell you about how Java handles the passing of object types via RMI?

8.2 Extensions

If you complete the server and client, add the following extensions:

1. When the client performs changes on the account balances, the changes will be recorded in memory. Add support for saving the updates to the bank account details to the accounts file.
2. Modify the client program to support adding new accounts on the bank server. That implies creating new `AccountImp` objects, updating the `AccountList` object, and registering them as needed.



```
Welcome to the Bank Client.
4accounts recognized:
Account number: 456
Account number: 900
Account number: 789
Account number: 100
Enter an account number or a blank line to quit:
100
Choose an operation to perform:
1. Deposit
2. Withdraw
3. Show Customer details and account number
4. Show balance.
5. Log into another account
4
Current balance: 100.9
Choose an operation to perform:
1. Deposit
2. Withdraw
3. Show Customer details and account number
4. Show balance.
5. Log into another account
1
Current balance: 100.9
How much to deposit? 250
Deposit successful
Current balance: 350.9
Choose an operation to perform:
1. Deposit
2. Withdraw
3. Show Customer details and account number
4. Show balance.
5. Log into another account
2
Current balance: 350.9
How much to withdraw? 400
Illegal amount.  Try again.
300
Withdrawal successful
Current balance: 50.89999999999998
Choose an operation to perform:
1. Deposit
2. Withdraw
3. Show Customer details and account number
4. Show balance.
5. Log into another account
3
Account details: Dan (90000) Hamigdal 2, Afula
```

Figure 4: Bank client trace