Tutorial on Java Multithreading

Yida Tao, Yao Zhao

Synchronization

We've learned `Bank Account Management` in the lecture. In this part, we continue to take `a simulated bank` for example. The example is cited from Section 12.4, Core Java Volume I Fundamentals 11th Edition.

In `Bank` class, a double array is used to store a number of bank accounts:

```
public class Bank
{
   private final double[] accounts;
   /**
    * Constructs the bank.
    * @param n the number of accounts
    * @param initialBalance the initial balance for each account
    */
   public Bank(int n, double initialBalance)
   {
      accounts = new double[n];
      Arrays.fill(accounts, initialBalance);
   }

...
```

Here is the code for the `transfer` method of the `Bank` class:

```
   /**
    * Transfers money from one account to another.
    * @param from the account to transfer from
    * @param to the account to transfer to
    * @param amount the amount to transfer
    */
   public void transfer(int from, int to, double amount)
   {
      if (accounts[from] < amount) return;
      System.out.print(Thread.currentThread());
      accounts[from] -= amount;
      System.out.printf(" %10.2f from %d to %d", amount, from, to);
      accounts[to] += amount;
      System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
   }
```

Here is the code for the `Runnable` instances in `UnsynchBankTest`:

```
        Runnable r = () -> {
           try
           {
              while (true)
              {
                 int toAccount = (int) (bank.size() * Math.random());
                 double amount = MAX_AMOUNT * Math.random();
                 bank.transfer(fromAccount, toAccount, amount);
                 Thread.sleep((int) (DELAY * Math.random()));
              }
           }
           catch (InterruptedException e)
           {
           }
        };
```

When this simulation runs, we do not know how much money is in any one bank account at any time. But we do know that the total amount of money in all the accounts should remain unchanged because all we do is move money from one account to another.

In `UnsynchBankTest`:

```
public class UnsynchBankTest
{
    public static final int NACCOUNTS = 100;
    public static final double INITIAL_BALANCE = 1000;
    ...
    public static void main(String[] args)
    {
        var bank = new Bank(NACCOUNTS, INITIAL_BALANCE);
        ...
    }
    ...
}
```

The size of `accounts` in `bank` initials to 100 and the value per account initials to 1000, so the total amount would be always 100000.00.

At the end of each transaction, the transfer method recomputes the total amount and prints it. If the process goes correctly, the output will be always 100000.00.

First, we can try to run the `UnsynchBankTest` in package `unsynch`. This program never finishes. Just press `Ctrl+C` in the commandline or `Stop` button in IDE to kill the program.

Here is a typical printout:

```
...
Thread[Thread-86,5,main]     167.13 from 86 to 17 Total Balance:
100000.00
Thread[Thread-63,5,main]     732.47 from 63 to 3 Total Balance:  100000.00
Thread[Thread-28,5,main]     808.41 from 28 to 19 Total Balance:
100000.00
Thread[Thread-79,5,main]     530.71 from 79 to 7 Total Balance:  100000.00
...
Thread[Thread-7,5,main]     310.60 from 7 to 80 Total Balance:   99339.30
Thread[Thread-11,5,main]     211.68 from 11 to 17 Total Balance:
99339.30
Thread[Thread-11,5,main]     191.81 from 11 to 72 Total Balance:
99339.30
Thread[Thread-72,5,main]     139.57 from 72 to 3 Total Balance:   99339.30
...
Thread[Thread-67,5,main]     543.89 from 67 to 40 Total Balance:
99339.30
Thread[Thread-53,5,main]Thread[Thread-80,5,main]     317.93 from 80 to 31
Total Balance:   98924.86
Thread[Thread-96,5,main]     108.88 from 96 to 76 Total Balance:
98924.86
Thread[Thread-48,5,main]      31.05 from 48 to 16 Total Balance:
98924.86
     660.70 from 24 to 19 Total Balance:   99585.55
     414.45 from 53 to 64 Total Balance:  100000.00
...
```

As you can see, something is wrong. For a few transactions, the bank balance remains at $100,000, which is the correct total for 100 accounts of $1,000 each. But after some time, the balance changes slightly. Why?

We can fix the problem using the following two approaches.

The synchronized Keyword

we can simply declare the `transfer()` method of the `Bank` class as synchronized like this:

```
public class Bank
{
   private final double[] accounts;

    public synchronized void transfer(int from, int to, double amount)
         throws InterruptedException
    {
      while (accounts[from] < amount)
         wait();
      System.out.print(Thread.currentThread());
      accounts[from] -= amount;
      System.out.printf(" %10.2f from %d to %d", amount, from, to);
      accounts[to] += amount;
      System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
      notifyAll();
```

```
    }

    public synchronized double getTotalBalance(){...}

    ...
}
```

Run the program again, and check if the bank balance will become corrupted or not.

Lock

We can also use a lock to protect the transfer method of the Bank class:

```
public class Bank {
    private Lock bankLock = new ReentrantLock();
    ...
    public void transfer(int from, int to, double amount) {
        bankLock.lock();
        try{
            System.out.print(Thread.currentThread());
            accounts[from] -= amount;
            System.out.printf(" %10.2f from %d to %d", amount, from,to);
            accounts[to] += amount;
            System.out.printf(" Total Balance: %10.2f%n",
getTotalBalance());
        }
        finally {
            bankLock.unlock();
        }
    }
}
```

Run the program again, and check if the bank balance will become corrupted or not.

Condition

What we should do if we do not want to transfer money out of an account that does not have the funds to cover the transfer? Note that we cannot use code like:

```
    if (bank.getBalance(from) >= amount) bank.transfer(from, to, amount);
```

When there is not enough money in the account, we can wait until some other thread has added funds.

If we only use `Lock`:

```
public void transfer(int from, int to, double amount) {
    bankLock.lock();
    try{
        while (accounts[from] < amount) {
            // wait
            ...
        }
        // transfer funds
        ...
    }
    finally {
        bankLock.unlock();
    }
}
```

When some thread has entered the `wait` branch, this thread has just gained exclusive access to the `bankLock`, so no other thread has a chance to make a deposit. This is where condition objects come in.

A lock object can have one or more associated condition objects. It is customary to give each condition object a name that evokes the condition that it represents. For example, here we set up a condition object to represent the “sufficient funds” condition.

```
class Bank {
    ...
    private Condition sufficientFunds; ...
    public Bank(){
        ...
        sufficientFunds = bankLock.newCondition();
        ...
    }
    ...
}
```

`sufficientFunds.await();` will make the current thread deactivated and give up the lock, while `sufficientFunds.signalAll();` will reactivate all threads waiting for the condition.

Try to add the `sufficientFunds.await();` and `sufficientFunds.signalAll();` to the proper places in `transfer()` method. Then compare it to `Bank.java` in `synch` package to see if you got it right.

```
NOTE: In general, a call to await should be inside a loop of the form
while (!(OK to proceed))
    condition.await();
```

Thread States

We've learned different thread states in the lecture. In this part, we further provide a sample code `ThreadState.java`; you may execute this code to see how different thread states occur in action.