BankAccountThreadRunner.java

```
 1  /**
 2     This program runs threads that deposit and withdraw
 3     money from the same bank account.
 4  */
 5  public class BankAccountThreadRunner
 6  {
 7     public static void main(String[] args)
 8     {
 9        BankAccount account = new BankAccount();
10        final double AMOUNT = 100;
11        final int REPETITIONS = 100;
12        final int THREADS = 100;
13
14        for (int i = 1; i <= THREADS; i++)
15        {
16           DepositRunnable d = new DepositRunnable(
17              account, AMOUNT, REPETITIONS);
18           WithdrawRunnable w = new WithdrawRunnable(
19              account, AMOUNT, REPETITIONS);
20
```

Continued

BankAccountThreadRunner.java (cont.)

```
21              Thread dt = new Thread(d);
22              Thread wt = new Thread(w);
23
24              dt.start();
25              wt.start();
26          }
27      }
28  }
```

DepositRunnable.java

```
1  /**
2     A deposit runnable makes periodic deposits to a bank account.
3  */
4  public class DepositRunnable implements Runnable
5  {
6     private static final int DELAY = 1;
7     private BankAccount account;
8     private double amount;
9     private int count;
10
11    /**
12       Constructs a deposit runnable.
13       @param anAccount the account into which to deposit money
14       @param anAmount the amount to deposit in each repetition
15       @param aCount the number of repetitions
16    */
17    public DepositRunnable(BankAccount anAccount, double anAmount,
18          int aCount)
19    {
20       account = anAccount;
21       amount = anAmount;
22       count = aCount;
23    }
24
```

Continued

DepositRunnable.java (cont.)

```
25      public void run()
26      {
27         try
28         {
29            for (int i = 1; i <= count; i++)
30            {
31               account.deposit(amount);
32               Thread.sleep(DELAY);
33            }
34         }
35         catch (InterruptedException exception) {}
36      }
37   }
```

WithdrawRunnable.java

```
1  /**
2     A withdraw runnable makes periodic withdrawals from a bank account.
3  */
4  public class WithdrawRunnable implements Runnable
5  {
6     private static final int DELAY = 1;
7     private BankAccount account;
8     private double amount;
9     private int count;
10
11    /**
12       Constructs a withdraw runnable.
13       @param anAccount the account from which to withdraw money
14       @param anAmount the amount to withdraw in each repetition
15       @param aCount the number of repetitions
16    */
17    public WithdrawRunnable(BankAccount anAccount, double anAmount,
18          int aCount)
19    {
20       account = anAccount;
21       amount = anAmount;
22       count = aCount;
23    }
```

Continued

WithdrawRunnable.java (cont.)

```
24
25      public void run()
26      {
27         try
28         {
29            for (int i = 1; i <= count; i++)
30            {
31               account.withdraw(amount);
32               Thread.sleep(DELAY);
33            }
34         }
35         catch (InterruptedException exception) {}
36      }
37   }
```

BankAccount.java

```
1   /**
2       A bank account has a balance that can be changed by
3       deposits and withdrawals.
4   */
5   public class BankAccount
6   {
7      private double balance;
8
9      /**
10        Constructs a bank account with a zero balance.
11     */
12     public BankAccount()
13     {
14        balance = 0;
15     }
16
```

Continued

BankAccount.java (cont.)

```
17      /**
18         Deposits money into the bank account.
19         @param amount the amount to deposit
20      */
21      public void deposit(double amount)
22      {
23         System.out.print("Depositing " + amount);
24         double newBalance = balance + amount;
25         System.out.println(", new balance is " + newBalance);
26         balance = newBalance;
27      }
28
```

Continued

BankAccount.java (cont.)

```
29      /**
30         Withdraws money from the bank account.
31         @param amount the amount to withdraw
32      */
33      public void withdraw(double amount)
34      {
35         System.out.print("Withdrawing " + amount);
36         double newBalance = balance - amount;
37         System.out.println(", new balance is " + newBalance);
38         balance = newBalance;
39      }
40
41      /**
42         Gets the current balance of the bank account.
43         @return the current balance
44      */
45      public double getBalance()
46      {
47         return balance;
48      }
49   }
```

Continued

BankAccount.java (cont.)

Program Run:

```
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
...
Withdrawing 100.0, new balance is 400.0
Depositing 100.0, new balance is 500.0
Withdrawing 100.0, new balance is 400.0
Withdrawing 100.0, new balance is 300.0
```

21.3 Race Conditions

- When threads share a common object, they can conflict with each other
- **Sample program:** multiple threads manipulate a bank account
  - Create two sets of threads:
    - Each thread in the first set repeatedly deposits $100
    - Each thread in the second set repeatedly withdraws $100

Sample Program (1)

- `run` method of `DepositRunnable` class:

```
public void run()
{
   try
   {
      for (int i = 1; i <= count; i++)
      {
         account.deposit(amount);
         Thread.sleep(DELAY);
      }
   }
   catch (InterruptedException exception)
   {
   }
}
```

- Class `WithdrawRunnable` is similar – it withdraws money instead

Sample Program (2)

- Create a `BankAccount` object, where `deposit` and `withdraw` methods have been modified to print messages:

```
public void deposit(double amount)
{
   System.out.print("Depositing " + amount);
   double newBalance = balance + amount;
   System.out.println(", new balance is "
      + newBalance);
   balance = newBalance;
}
```

Sample Program (3)

- Normally, the program output looks somewhat like this:

```
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
Withdrawing 100.0, new balance is 100.0
...
Withdrawing 100.0, new balance is 0.0
```

- The end result should be zero, but sometimes the output is messed up, and sometimes end result is not zero:

```
Depositing 100.0Withdrawing 100.0, new balance is
  100.0, new balance is -100.0
```

Sample Program (4)

- Scenario to explain problem:
  1. A deposit thread executes the lines:

     ```
     System.out.print("Depositing " + amount);
     double newBalance = balance + amount;
     ```

     The `balance` variable is still `0`, and the `newBalance` local variable is `100`
  2. The deposit thread reaches the end of its time slice and a withdraw thread gains control
  3. The withdraw thread calls the `withdraw` method which withdraws $100 from the `balance` variable; it is now `-100`
  4. The withdraw thread goes to sleep

Sample Program (5)

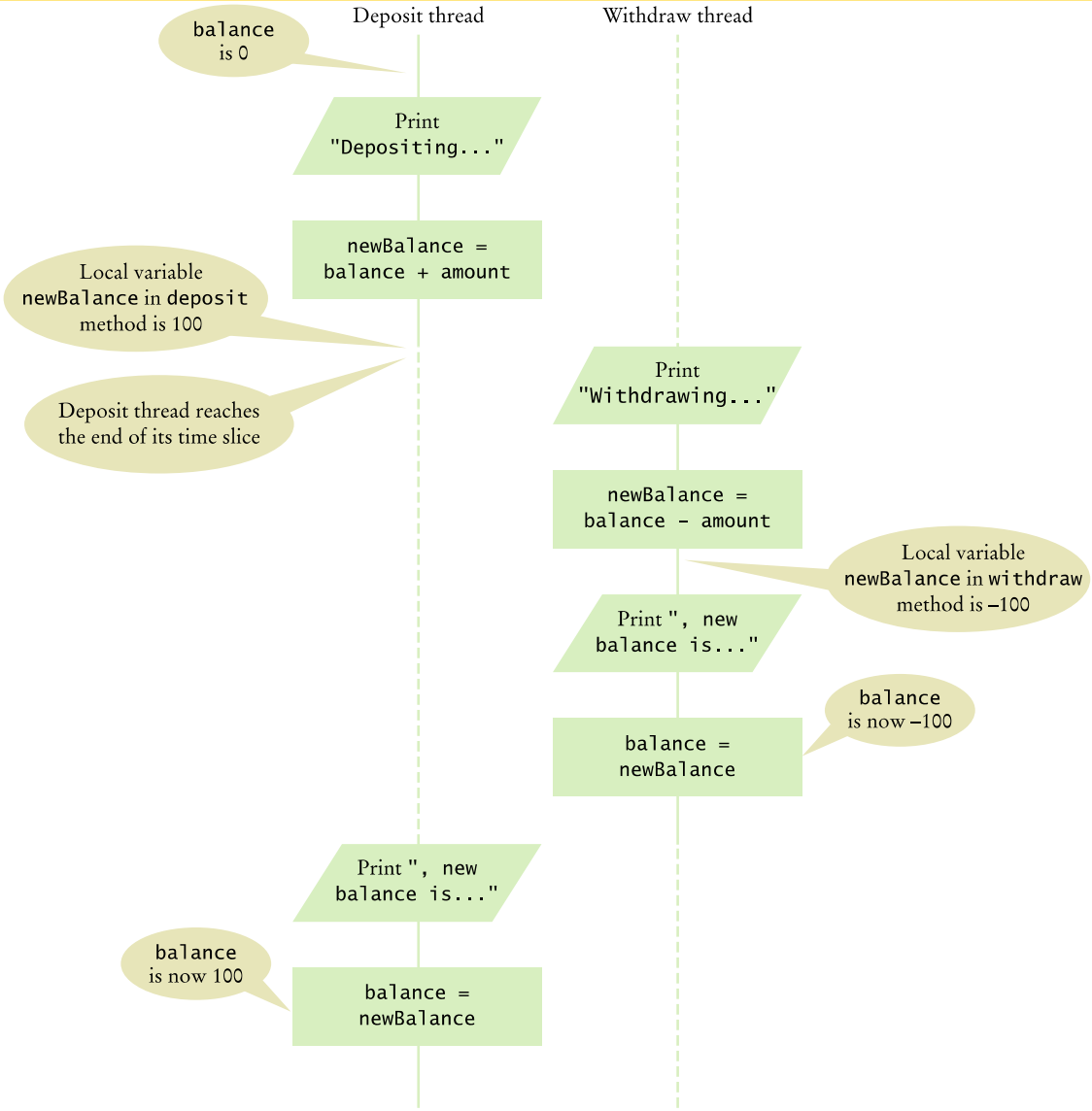
- Scenario to explain problem (cont.):

5. The deposit thread regains control and picks up where it was iterrupted. It now executes:

```
System.out.println(", new balance is " + newBalance);
balance = newBalance;
```

The `balance` variable is now `100` instead of `0` because the `deposit` method used the *old* balance to compute the value of its local variable `newBalance`

Corrupting the Contents of the `balance` Variable



Race Condition

- Occurs if the effect of multiple threads on shared data depends on the order in which they are scheduled
- It is possible for a thread to reach the end of its time slice in the middle of a statement
- It may evaluate the right-hand side of an equation but not be able to store the result until its next turn:

```
public void deposit(double amount)
{
   balance = balance + amount;
   System.out.print("Depositing " + amount
      + ", new balance is " + balance);
}
```

- Race condition can still occur:

```
balance = the right-hand-side value
```

21.4 Synchronizing Object Access

- To solve problems such as the one just seen, use a *lock object*
- **Lock object:** used to control threads that manipulate shared resources
- In Java library: `Lock` interface and several classes that implement it
  - `ReentrantLock`: most commonly used lock class
  - Locks are a feature of Java version 5.0
  - Earlier versions of Java have a lower-level facility for thread synchronization

Synchronizing Object Access (2)

- Typically, a `Lock` object is added to a class whose methods access shared resources, like this:

```
public class BankAccount
{
   private Lock balanceChangeLock;
   . . .
   public BankAccount()
   {
      balanceChangeLock = new ReentrantLock();
      . . .
   }
   . . .
}
```

Synchronizing Object Access (3)

- Code that manipulates shared resource is surrounded by calls to `lock` and `unlock`:

```
balanceChangeLock.lock();
Manipulate the shared resource.
balanceChangeLock.unlock();
```

- If code between calls to `lock` and `unlock` throws an exception, call to `unlock` never happens

Synchronizing Object Access (4)

- To overcome this problem, place call to `unlock` into a `finally` clause:

```
balanceChangeLock.lock();
try
{
   Manipulate the shared resource.
}
finally
{
   balanceChangeLock.unlock();
}
```

Synchronizing Object Access (5)

- Code for `deposit` method:

```
public void deposit(double amount)
{
   balanceChangeLock.lock();
   try
   {
      System.out.print("Depositing " + amount);
      double newBalance = balance + amount;
      System.out.println(", new balance is "
         + newBalance);
      balance = newBalance;
   }
   finally
   {
      balanceChangeLock.unlock();
   }
}
```

Synchronizing Object Access (6)

- When a thread calls `lock`, it owns the lock until it calls `unlock`
- A thread that calls `lock` while another thread owns the lock is temporarily deactivated
- Thread scheduler periodically reactivates thread so it can try to acquire the lock
- Eventually, waiting thread can acquire the lock

21.5 Avoiding Deadlocks

- A deadlock occurs if no thread can proceed because each thread is waiting for another to do some work first
- `BankAccount` example:

```
public void withdraw(double amount)
{
   balanceChangeLock.lock();
   try
   {
      while (balance < amount)
         Wait for the balance to grow
      ...
   }
   finally
   {
      balanceChangeLock.unlock();
   }
}
```

Avoiding Deadlocks (2)

- How can we wait for the balance to grow?
- We can't simply call `sleep` inside `withdraw` method; thread will block all other threads that want to use `balanceChangeLock`
- In particular, no other thread can successfully execute `deposit`
- Other threads will call `deposit`, but will be blocked until `withdraw` exits
- But `withdraw` doesn't exit until it has funds available
- DEADLOCK

Condition Objects (1)

- To overcome problem, use a **condition object**
- Condition objects allow a thread to temporarily release a lock, and to regain the lock at a later time
- Each condition object belongs to a specific lock object

Condition Objects (2)

- You obtain a condition object with `newCondition` method of `Lock` interface:

```
public class BankAccount
{
   private Lock balanceChangeLock;
   private Condition sufficientFundsCondition;
   . . .
   public BankAccount()
   {
      balanceChangeLock = new ReentrantLock();
      sufficientFundsCondition =
         balanceChangeLock.newCondition();
      . . .
   }
}
```

Condition Objects (3)

- It is customary to give the condition object a name that describes condition to test; e.g. “sufficient funds”
- You need to implement an appropriate test

Condition Objects (4)

- As long as test is not fulfilled, call `await` on the condition object:

```
public void withdraw(double amount)
{
   balanceChangeLock.lock();
   try
   {
      while (balance < amount)
      {
         sufficientFundsCondition.await();
      }
      . . .
   }
   finally
   {
      balanceChangeLock.unlock();
   }
}
```

Condition Objects (5)

- Calling `await`
  - Makes current thread wait
  - Allows another thread to acquire the lock object
- To unblock, another thread must execute `signalAll` *on the same condition object* :

```
sufficientFundsCondition.signalAll();
```

- `signalAll` unblocks all threads waiting on the condition
- `signal` randomly picks just one thread waiting on the object and unblocks it
- `signal` can be more efficient, but you need to know that *every* waiting thread can proceed
- **Recommendation:** always call `signalAll`

BankAccount.java

```
1  import java.util.concurrent.locks.Condition;
2  import java.util.concurrent.locks.Lock;
3  import java.util.concurrent.locks.ReentrantLock;
4
5  /**
6     A bank account has a balance that can be changed by
7     deposits and withdrawals.
8  */
9  public class BankAccount
10 {
11    private double balance;
12    private Lock balanceChangeLock;
13    private Condition sufficientFundsCondition;
14
15    /**
16       Constructs a bank account with a zero balance.
17    */
18    public BankAccount()
19    {
20       balance = 0;
21       balanceChangeLock = new ReentrantLock();
22       sufficientFundsCondition = balanceChangeLock.newCondition();
23    }
```

Continued

BankAccount.java (cont.)

```
24
25      /**
26         Deposits money into the bank account.
27         @param amount the amount to deposit
28      */
29      public void deposit(double amount)
30      {
31         balanceChangeLock.lock();
32         try
33         {
34            System.out.print("Depositing " + amount);
35            double newBalance = balance + amount;
36            System.out.println(", new balance is " + newBalance);
37            balance = newBalance;
38            sufficientFundsCondition.signalAll();
39         }
40         finally
41         {
42            balanceChangeLock.unlock();
43         }
44      }
45
```

Continued

BankAccount.java (cont.)

```
46      /**
47         Withdraws money from the bank account.
48         @param amount the amount to withdraw
49      */
50      public void withdraw(double amount)
51            throws InterruptedException
52      {
53         balanceChangeLock.lock();
54         try
55         {
56            while (balance < amount)
57            {
58               sufficientFundsCondition.await();
59            }
60            System.out.print("Withdrawing " + amount);
61            double newBalance = balance - amount;
62            System.out.println(", new balance is " + newBalance);
63            balance = newBalance;
64         }
```

Continued

BankAccount.java (cont.)

```
65          finally
66          {
67             balanceChangeLock.unlock();
68          }
69       }
70
71       /**
72          Gets the current balance of the bank account.
73          @return the current balance
74       */
75       public double getBalance()
76       {
77          return balance;
78       }
79    }
```

Continued

BankAccountThreadRunner.java

```
1  /**
2     This program runs threads that deposit and withdraw
3     money from the same bank account.
4  */
5  public class BankAccountThreadRunner
6  {
7     public static void main(String[] args)
8     {
9        BankAccount account = new BankAccount();
10       final double AMOUNT = 100;
11       final int REPETITIONS = 100;
12       final int THREADS = 100;
13
14       for (int i = 1; i <= THREADS; i++)
15       {
16          DepositRunnable d = new DepositRunnable(
17             account, AMOUNT, REPETITIONS);
18          WithdrawRunnable w = new WithdrawRunnable(
19             account, AMOUNT, REPETITIONS);
```

Continued

BankAccountThreadRunner.java (cont.)

```
20
21              Thread dt = new Thread(d);
22              Thread wt = new Thread(w);
23
24              dt.start();
25              wt.start();
26          }
27      }
28  }
```

BankAccountThreadRunner.java (cont.)

Program Run:

```
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
...
Withdrawing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
Withdrawing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
```

Review: Running Threads

- A thread is a program unit that is executed concurrently with other parts of the program.
- The `start` method of the `Thread` class starts a new thread that executes the `run` method of the associated `Runnable` object.
- The `sleep` method puts the current thread to sleep for a given number of milliseconds.
- When a thread is interrupted, the most common response is to terminate the `run` method.
- The thread scheduler runs each thread for a short amount of time, called a time slice.

Review: Terminating Threads

- A thread terminates when its run method terminates.
- The `run` method can check whether its thread has been interrupted by calling the `interrupted` method.

Review: Race Conditions

- A race condition occurs if the effect of multiple threads on shared data depends on the order in which the threads are scheduled.

Review: Synchronizing Object Access

- By calling the `lock` method, a thread acquires a `Lock` object. Then no other thread can acquire the lock until the first thread releases the lock.

Review: Avoiding Deadlocks

- A deadlock occurs if no thread can proceed because each thread is waiting for another to do some work first.
- Calling `await` on a condition object makes the current thread wait and allows another thread to acquire the lock object.
- A waiting thread is blocked until another thread calls `signalAll` or `signal` on the condition object for which the thread is waiting.