Multithreading

Chapter Goals

- To understand how multiple threads can execute in parallel
- To learn how to implement threads
- To understand race conditions and deadlocks
- To be able to avoid corruption of shared objects by using locks and conditions
- To be able to use threads for programming animations

Threads

- **Thread:** a program unit that is executed independently of other parts of the program
- The Java Virtual Machine executes each thread in the program for a short amount of time
- This gives the impression of parallel execution

Running a Thread

- Implement a class that implements the `Runnable` interface:

```
public interface Runnable
{
   void run();
}
```

- Place the code for your task into the `run` method of your class:

```
public class MyRunnable implements Runnable
{
   public void run()
   {
      Task statements
      ...
   }
}
```

Running a Thread

- Create an object of your subclass:

```
Runnable r = new MyRunnable();
```

- Construct a `Thread` object from the `runnable` object:

```
Thread t = new Thread(r);
```

- Call the `start` method to start the thread:

```
t.start();
```

Example

A program to print a time stamp and “Hello World” once a second for ten seconds:

```
Mon Dec 28 23:12:03 PST 2009 Hello, World!
Mon Dec 28 23:12:04 PST 2009 Hello, World!
Mon Dec 28 23:12:05 PST 2009 Hello, World!
Mon Dec 28 23:12:06 PST 2009 Hello, World!
Mon Dec 28 23:12:07 PST 2009 Hello, World!
Mon Dec 28 23:12:08 PST 2009 Hello, World!
Mon Dec 28 23:12:09 PST 2009 Hello, World!
Mon Dec 28 23:12:10 PST 2009 Hello, World!
Mon Dec 28 23:12:11 PST 2009 Hello, World!
Mon Dec 28 23:12:12 PST 2009 Hello, World!
```

`GreetingRunnable` Outline

```
public class GreetingRunnable implements Runnable
{
   private String greeting;

   public GreetingRunnable(String aGreeting)
   {
      greeting = aGreeting;
   }

   public void run()
   {
      Task statements
      ...
   }
}
```

Thread Action for `GreetingRunnable`

- Print a time stamp
- Print the greeting
- Wait a second

`GreetingRunnable`

- We can get the date and time by constructing a `Date` object:

  ```
  Date now = new Date();
  ```

- To wait a second, use the sleep method of the `Thread` class:

  ```
  sleep(milliseconds)
  ```

- A sleeping thread can generate an `InterruptedException`
  - *Catch the exception*
  - *Terminate the thread*

Running Threads

- `sleep` puts current thread to sleep for given number of milliseconds:

```
    Thread.sleep(milliseconds)
```

- When a thread is interrupted, most common response is to terminate `run`

Generic run method

```
public void run()
{
   try
   {
      Task statements
   }
   catch (InterruptedException exception)
   {
   }
   Clean up, if necessary
}
```

ch20/greeting/GreetingRunnable.java

```java
import java.util.Date;

/**
   A runnable that repeatedly prints a greeting.
*/
public class GreetingRunnable implements Runnable
{
   private static final int REPETITIONS = 10;
   private static final int DELAY = 1000;

   private String greeting;

   /**
      Constructs the runnable object.
      @param aGreeting the greeting to display
   */
   public GreetingRunnable(String aGreeting)
   {
      greeting = aGreeting;
   }

```

Continued

ch20/greeting/GreetingRunnable.java (cont.)

```java
   public void run()
   {
      try
      {
         for (int i = 1; i <= REPETITIONS; i++)
         {
            Date now = new Date();
            System.out.println(now + " " + greeting);
            Thread.sleep(DELAY);
         }
      }
      catch (InterruptedException exception)
      {
      }
   }
}
```

To Start the Thread

- Construct an object of your `runnable` class:

```
Runnable t = new GreetingRunnable("Hello World");
```

- Then construct a thread and call the `start` method:

```
Thread t = new Thread(r);
t.start();
```

ch20/greeting/GreetingThreadRunner.java

```java
/**
   This program runs two greeting threads in parallel.
*/
public class GreetingThreadRunner
{
   public static void main(String[] args)
   {
      GreetingRunnable r1 = new GreetingRunnable("Hello, World!");
      GreetingRunnable r2 = new GreetingRunnable("Goodbye, World!");
      Thread t1 = new Thread(r1);
      Thread t2 = new Thread(r2);
      t1.start();
      t2.start();
   }
}
```

Continued

ch20/greeting/GreetingThreadRunner.java (cont.)

Program Run:

```
Mon Dec 28 12:04:46 PST 2009 Hello, World!
Mon Dec 28 12:04:46 PST 2009 Goodbye, World!
Mon Dec 28 12:04:47 PST 2009 Hello, World!
Mon Dec 28 12:04:47 PST 2009 Goodbye, World!
Mon Dec 28 12:04:48 PST 2009 Hello, World!
Mon Dec 28 12:04:48 PST 2009 Goodbye, World!
Mon Dec 28 12:04:49 PST 2009 Hello, World!
Mon Dec 28 12:04:49 PST 2009 Goodbye, World!
Mon Dec 28 12:04:50 PST 2009 Hello, World!
Mon Dec 28 12:04:50 PST 2009 Goodbye, World!
Mon Dec 28 12:04:51 PST 2009 Hello, World!
Mon Dec 28 12:04:51 PST 2009 Goodbye, World!
Mon Dec 28 12:04:52 PST 2009 Goodbye, World!
Mon Dec 28 12:04:52 PST 2009 Hello, World!
Mon Dec 28 12:04:53 PST 2009 Hello, World!
Mon Dec 28 12:04:53 PST 2009 Goodbye, World!
Mon Dec 28 12:04:54 PST 2009 Hello, World!
Mon Dec 28 12:04:54 PST 2009 Goodbye, World!
Mon Dec 28 12:04:55 PST 2009 Hello, World!
Mon Dec 28 12:04:55 PST 2009 Goodbye, World!
```

Thread Scheduler

- **Thread scheduler:** runs each thread for a short amount of time (a **time slice**)
- Then the scheduler activates another thread
- There will always be slight variations in running times - especially when calling operating system services (e.g. input and output)
- There is no guarantee about the order in which threads are executed

Self Check 20.1

What happens if you change the call to the `sleep` method in the `run` method to `Thread.sleep(1)`?

> **Answer:** The messages are printed about one millisecond apart.

Self Check 20.2

What would be the result of the program if the `main` method called

```
r1.run();
r2.run();
```

instead of starting threads?

Answer: The first call to `run` would print ten “Hello” messages, and then the second call to `run` would print ten “Goodbye” messages

Terminating Threads

- A thread terminates when its `run` method terminates
- Do not terminate a thread using the deprecated `stop` method
- Instead, notify a thread that it should terminate:

```
    t.interrupt();
```

- `interrupt` does not cause the thread to terminate – it sets a boolean variable in the thread data structure

Terminating Threads

- The `run` method should check occasionally whether it has been interrupted
  - *Use the `interrupted` method*
  - *An interrupted thread should release resources, clean up, and exit:*

```
public void run()
{
   for (int i = 1;
      i <= REPETITIONS && !Thread.interrupted();
      i++)
   {
      Do work
   }
   Clean up
}
```

Terminating Threads

- The `sleep` method throws an `InterruptedException` when a sleeping thread is interrupted
  - *Catch the exception*
  - *Terminate the thread :*

```
public void run()
{
   try
   {
      for (int i = 1; i <= REPETITIONS; i++)
      {
            Do work
            Sleep
      }
   }
   catch (InterruptedException exception)
   {
         Clean up
   }
}
```

Terminating Threads

- Java does not force a thread to terminate when it is interrupted
- It is entirely up to the thread what it does when it is interrupted
- Interrupting is a general mechanism for getting the thread’s attention

Self Check 20.4

Consider the following `Runnable`:

```
public class MyRunnable implements Runnable
{
   public void run()
   {
      try
      {
         System.out.println(1);
         Thread.sleep(1000);
         System.out.println(2);
      }
      catch (InterruptedException exception)
      {
         System.out.println(3);
      }
      System.out.println(4);
   }
}
```

Self Check 20.4 (cont.)

Suppose a thread with this `Runnable` is started and immediately interrupted.

```
Thread t = new Thread(new MyRunnable());
t.start();
t.interrupt();
```

What output is produced?

Answer: The run method prints the values 1, 3, and 4. The call to `interrupt` merely sets the interruption flag, but the `sleep` method immediately throws an `InterruptedException`.

Race Conditions

- When threads share a common object, they can conflict with each other
- Sample program: multiple threads manipulate a bank account

ch20/unsynch/BankAccountThreadRunner.java

```
/**
   This program runs threads that deposit and withdraw
   money from the same bank account.
*/
public class BankAccountThreadRunner
{
   public static void main(String[] args)
   {
      BankAccount account = new BankAccount();
      final double AMOUNT = 100;
      final int REPETITIONS = 100;
      final int THREADS = 100;

      for (int i = 1; i <= THREADS; i++)
      {
         DepositRunnable d = new DepositRunnable(
            account, AMOUNT, REPETITIONS);
         WithdrawRunnable w = new WithdrawRunnable(
            account, AMOUNT, REPETITIONS);

         Thread dt = new Thread(d);
         Thread wt = new Thread(w);

```

Continued

ch20/unsynch/BankAccountThreadRunner.java (cont.)

```
         dt.start();
         wt.start();
      }
   }
}
```

ch20/unsynch/DepositRunnable.java

```java
/**
   A deposit runnable makes periodic deposits to a bank account.
*/
public class DepositRunnable implements Runnable
{
   private static final int DELAY = 1;
   private BankAccount account;
   private double amount;
   private int count;

   /**
      Constructs a deposit runnable.
      @param anAccount the account into which to deposit money
      @param anAmount the amount to deposit in each repetition
      @param aCount the number of repetitions
   */
   public DepositRunnable(BankAccount anAccount, double anAmount,
         int aCount)
   {
      account = anAccount;
      amount = anAmount;
      count = aCount;
   }

```

Continued

ch20/unsynch/DepositRunnable.java (cont.)

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

ch20/unsynch/WithdrawRunnable.java

```
/**
    A withdraw runnable makes periodic withdrawals from a bank account.
*/
public class WithdrawRunnable implements Runnable
{
   private static final int DELAY = 1;
   private BankAccount account;
   private double amount;
   private int count;

   /**
      Constructs a withdraw runnable.
      @param anAccount the account from which to withdraw money
      @param anAmount the amount to withdraw in each repetition
      @param aCount the number of repetitions
   */
   public WithdrawRunnable(BankAccount anAccount, double anAmount,
         int aCount)
   {
      account = anAccount;
      amount = anAmount;
      count = aCount;
   }

```

Continued

ch20/unsynch/WithdrawRunnable.java (cont.)

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

ch20/unsynch/BankAccount.java

```java
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount
{
   private double balance;

   /**
      Constructs a bank account with a zero balance.
   */
   public BankAccount()
   {
      balance = 0;
   }

```

Continued

ch20/unsynch/BankAccount.java (cont.)

```
   /**
      Deposits money into the bank account.
      @param amount the amount to deposit
   */
   public void deposit(double amount)
   {
      System.out.print("Depositing " + amount);
      double newBalance = balance + amount;
      System.out.println(", new balance is " + newBalance);
      balance = newBalance;
   }

```

Continued

ch20/unsynch/BankAccount.java (cont.)

```java
   /**
      Withdraws money from the bank account.
      @param amount the amount to withdraw
   */
   public void withdraw(double amount)
   {
      System.out.print("Withdrawing " + amount);
      double newBalance = balance - amount;
      System.out.println(", new balance is " + newBalance);
      balance = newBalance;
   }

   /**
      Gets the current balance of the bank account.
      @return the current balance
   */
   public double getBalance()
   {
      return balance;
   }
}
```

ch20/unsynch/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
```

Sample Application

- Create a `BankAccount` object
- Create two sets of threads:
    - *Each thread in the first set repeatedly deposits $100*
    - *Each thread in the second set repeatedly withdraws $100*
- `deposit` and `withdraw` 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 Application

- The result should be zero, but sometimes it is not
- 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
```

- But sometimes you may notice messed-up output, like this:

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

Scenario to Explain Non- zero Result: Race Condition

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

Continued

Scenario to Explain Non- zero Result: Race Condition

5. The deposit thread regains control and picks up where it left off; it executes:

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

The balance is now 100 instead of 0 because the deposit method used the OLD balance

Corrupting the Contents of the balance Variable

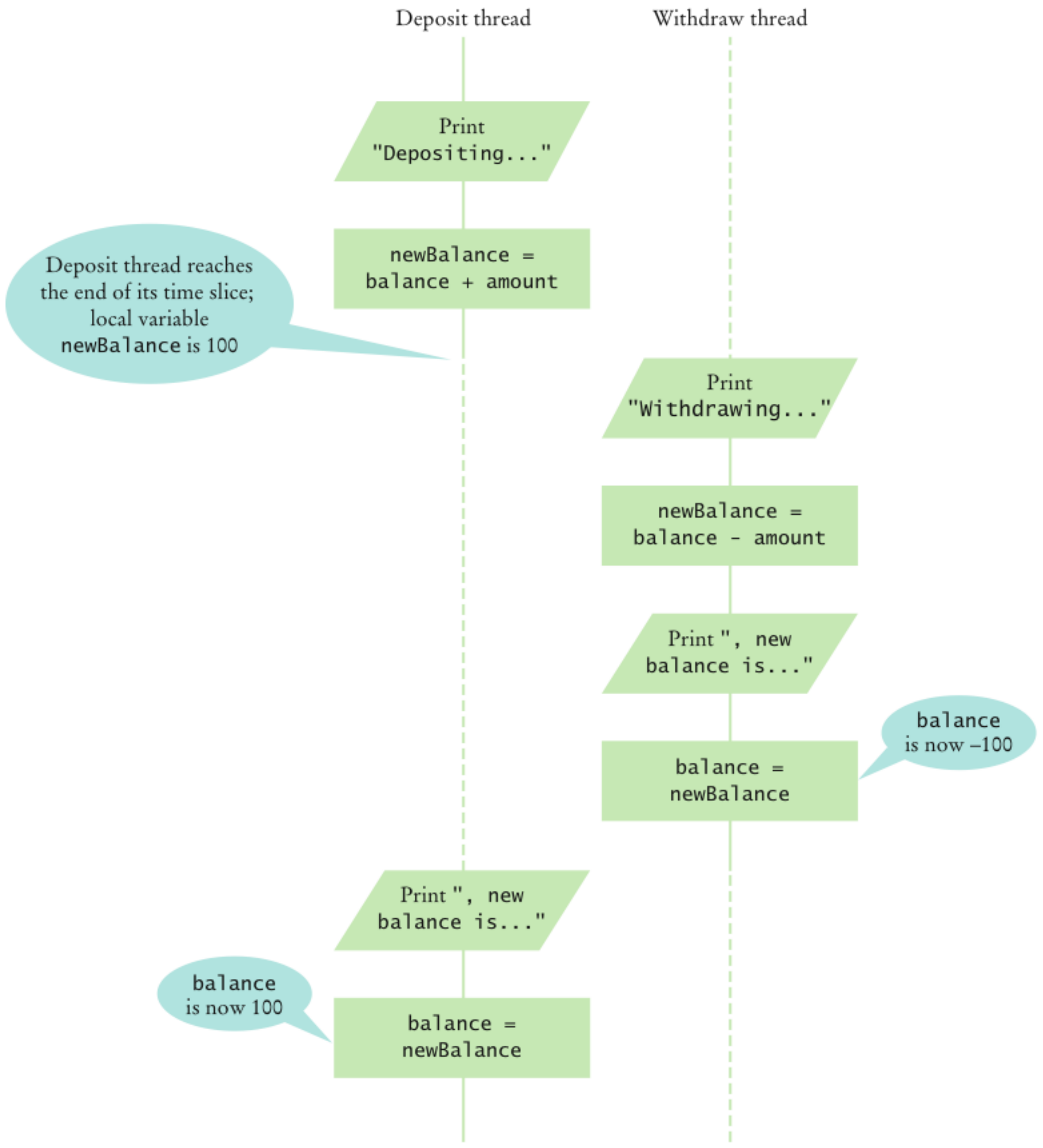
Deposit thread
Withdraw thread
Print "Depositing..."
newBalance = balance + amount
Deposit thread reaches the end of its time slice; local variable newBalance is 100
Print "Withdrawing..."
newBalance = balance - amount
Print ", new balance is..."
balance is now −100
balance = newBalance
Print ", new balance is..."
balance is now 100
balance = newBalance

Figure 1 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
```

Self Check 20.5

Give a scenario in which a race condition causes the bank balance to be -100 after one iteration of a deposit thread and a withdraw thread.

Answer: There are many possible scenarios. Here is one:

- *The first thread loses control after the first `print` statement.*
- *The second thread loses control just before the assignment `balance = newBalance`.*
- *The first thread completes the `deposit` method.*
- *The second thread completes the `withdraw` method.*

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: `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

- 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

- 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

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

```
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

- 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

Self Check 20.7

If you construct two `BankAccount` objects, how many lock objects are created?

> **Answer:** Two, one for each bank account object. Each lock protects a separate balance variable.

Self Check 20.8

What happens if we omit the call `unlock` at the end of the `deposit` method?

> **Answer:** When a thread calls `deposit`, it continues to own the lock, and any other thread trying to deposit or withdraw money in the same bank account is blocked forever.

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

- 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

- 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 (cont.)

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

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

Condition Objects

- It is customary to give the condition object a name that describes condition to test
- You need to implement an appropriate test

Condition Objects (cont.)

- 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

- 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`

ch20/synch/BankAccountThreadRunner.java

```
/**
   This program runs threads that deposit and withdraw
   money from the same bank account.
*/
public class BankAccountThreadRunner
{
   public static void main(String[] args)
   {
      BankAccount account = new BankAccount();
      final double AMOUNT = 100;
      final int REPETITIONS = 100;
      final int THREADS = 100;

      for (int i = 1; i <= THREADS; i++)
      {
         DepositRunnable d = new DepositRunnable(
            account, AMOUNT, REPETITIONS);
         WithdrawRunnable w = new WithdrawRunnable(
            account, AMOUNT, REPETITIONS);

         Thread dt = new Thread(d);
         Thread wt = new Thread(w);

```

Continued

ch20/synch/BankAccountThreadRunner.java (cont.)

```
            dt.start();
            wt.start();
         }
      }
   }
```

ch20/synch/BankAccount.java

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount
{
   private double balance;
   private Lock balanceChangeLock;
   private Condition sufficientFundsCondition;

   /**
      Constructs a bank account with a zero balance.
   */
   public BankAccount()
   {
      balance = 0;
      balanceChangeLock = new ReentrantLock();
      sufficientFundsCondition = balanceChangeLock.newCondition();
   }

```

ch20/synch/BankAccount.java (cont.)

```
   /**
      Deposits money into the bank account.
      @param amount the amount to deposit
   */
   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;
         sufficientFundsCondition.signalAll();
      }
      finally
      {
         balanceChangeLock.unlock();
      }
   }

```

Continued

ch20/synch/BankAccount.java (cont.)

```
   /**
      Withdraws money from the bank account.
      @param amount the amount to withdraw
   */
   public void withdraw(double amount)
         throws InterruptedException
   {
      balanceChangeLock.lock();
      try
      {
         while (balance < amount)
            sufficientFundsCondition.await();
         System.out.print("Withdrawing " + amount);
         double newBalance = balance - amount;
         System.out.println(", new balance is " + newBalance);
         balance = newBalance;
      }
      finally
      {
         balanceChangeLock.unlock();
      }
   }

```

Continued

ch20/synch/BankAccount.java (cont.)

```
   /**
      Gets the current balance of the bank account.
      @return the current balance
   */
   public double getBalance()
   {
      return balance;
   }
}
```

Continued

ch20/synch/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
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
```

Self Check 20.9

What is the essential difference between calling `sleep` and `await`?

> **Answer:** A sleeping thread is reactivated when the sleep delay has passed. A waiting thread is only reactivated if another thread has called `signalAll` or `signal`.

Self Check 20.10

Why is the `sufficientFundsCondition` object an instance variable of the `BankAccount` class and not a local variable of the `withdraw` and `deposit` methods?

Answer: The calls to `await` and `signal/signalAll` must be made *to the same object*.