MultiThreading in Java

**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:
  ```java
  public interface Runnable
  {
    void run();
  }
  ```
- Place the code for your task into the `run` method of your class:
  ```java
  public class MyRunnable implements Runnable
  {
    public void run()
    {
      Task statements
      ...
    }
  }
  ```
- Create an object of your subclass:
  ```java
  Runnable r = new MyRunnable();
  ```
- Construct a `Thread` object from the `runnable` object:
  ```java
  Thread t = new Thread(r);
  ```
- Call the `start` method to start the thread:
  ```java
  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!
```
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

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
    }
}
```

GreetingRunnable.java
```
import java.util.Date;
/*
 * A runnable that repeatedly prints a greeting.
 */
public class GreetingRunnable implements Runnable {
    static final int REPETITIONS = 10;
    static final int DELAY = 1000;
    private String greeting;
    /**
     * Constructs the runnable object.
     * @param aGreeting the greeting to display
     */
    public GreetingRunnable(String aGreeting) {
        greeting = aGreeting;
    }
    public void run() {
        try {
            Task statements
        } catch (InterruptedException exception) {
            Clean up, if necessary
        }
    }
}
```
GreetingRunnable.java (cont.)

```java
22 public void run(){
23     try {
24         for (int i = 1; i <= REPETITIONS; i++) {
25             Date now = new Date();
26             System.out.println(now + " " + greeting);
27             Thread.sleep(DELAY);
28         }
29     } catch (InterruptedException exception) {
30         }
31 } }
```

To Start the Thread

- Construct an object of your runnable class:
  ```java
  Runnable t = new GreetingRunnable("Hello World");
  ```
- Then construct a thread and call the start method:
  ```java
  Thread t = new Thread(r);
  t.start();
  ```

GreetingThreadRunner.java

```java
1 /**
2 * This program runs two greeting threads in parallel.
3 */
4 public class GreetingThreadRunner {
5     public static void main(String[] args) {
6         GreetingRunnable r1 = new GreetingRunnable("Hello, World!");
7         GreetingRunnable r2 = new GreetingRunnable("Goodbye, World!");
8         Thread t1 = new Thread(r1);
9         Thread t2 = new Thread(r2);
10         t1.start();
11         t2.start();
12     }
13 }
```

Program Run:

```
Mon Dec 28 12:04:45 PST 2009 Hello, World!
Mon Dec 28 12:04:45 PST 2009 Goodbye, World!
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 Hello, World!
Mon Dec 28 12:04:52 PST 2009 Goodbye, 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 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 2

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

```java
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:

```java
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:

```java
public void run() {
    for (int i = 1; i <= REPETITIONS && !Thread.interrupted(); i++) {
        Do work
        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 3

Suppose a web browser uses multiple threads to load the images on a web page. Why should these threads be terminated when the user hits the “Back” button?

**Answer:** If the user hits the “Back” button, the current web page is no longer displayed, and it makes no sense to expend network resources for fetching additional image data.
Self Check 4
Consider the following Runnable:

```java
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);
    }
}
```

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
Here is the run method of DepositRunnable:

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

The WithdrawRunnable class is similar

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:

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

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
- But sometimes you may notice messed-up output, like this:
  Depositing 100.0, withdrawing 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

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

BankAccountThreadRunner.java

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

BankAccountThreadRunner.java (cont.)

```java
24     dt.start();
25     wt.start();
26     }
27     }
```
DepositRunnable.java

```java
public class DepositRunnable implements Runnable {
    private static final int DELAY = 1;
    private BankAccount account;
    private double amount;
    private int count;

    public DepositRunnable(BankAccount anAccount, double anAmount, int aCount) {
        account = anAccount;
        amount = anAmount;
        count = aCount;
    }

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

WithdrawRunnable.java

```java
public class WithdrawRunnable implements Runnable {
    private static final int DELAY = 1;
    private BankAccount account;
    private double amount;
    private int count;

    public WithdrawRunnable(BankAccount anAccount, double anAmount, int aCount) {
        account = anAccount;
        amount = anAmount;
        count = aCount;
    }

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

BankAccount.java

```java
public class BankAccount {
    private double balance;

    public BankAccount() {
        balance = 0;
    }

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

```java
/**
 * Withdrawing 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;
}
```

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

Self Check 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

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

```java
public class BankAccount
{
    private Lock balanceChangeLock;

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

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

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

- If code between calls to `lock` and `unlock` throws an exception, call to `unlock` never happens
To overcome this problem, place a call to `unlock` into a `finally` clause:

```java
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();
    }
}
```

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 a thread so it can try to acquire the lock.

Eventually, a waiting thread can acquire the lock.

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.

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.

A deadlock occurs if no thread can proceed because each thread is waiting for another to do some work first.

**BankAccount example:**

```java
public void withdraw(double amount) {
    balanceChangeLock.lock();
    try {
        while (balance < amount)  // Wait for the balance to grow
            ...  // 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:
  ```java
  public class BankAccount {
      public BankAccount() {
          balanceChangeLock = new ReentrantLock();
          sufficientFundsCondition = balanceChangeLock.newCondition();
          ...
      }
      ...
      private Lock balanceChangeLock;
      private Condition sufficientFundsCondition;
  }
  ```

Condition Objects (cont.)

- 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:
  ```java
  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:
  ```java
  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`
BankAccountThreadRunner.java

```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);

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

    public static class DepositRunnable implements Runnable {
        private Lock balanceChangeLock;
        private Condition sufficientFundsCondition;

        public DepositRunnable(BankAccount account, double amount, int reps) {
            this.balanceChangeLock = account平衡ChangeLock;
            this.sufficientFundsCondition = balanceChangeLock.newCondition();
        }

        public void run() {
            balanceChangeLock.lock();
            try {
                System.out.println("Depositing " + amount);
                double newBalance = balance + amount;
                System.out.println("New balance is " + newBalance);
                balance = newBalance;
                sufficientFundsCondition.signalAll();
            } finally {
                balanceChangeLock.unlock();
            }
        }
    }

    public static class WithdrawRunnable implements Runnable {
        private Lock balanceChangeLock;
        private Condition sufficientFundsCondition;

        public WithdrawRunnable(BankAccount account, double amount, int reps) {
            this.balanceChangeLock = account.balanceChangeLock;
            this.sufficientFundsCondition = balanceChangeLock.newCondition();
        }

        public void run() {
            balanceChangeLock.lock();
            try {
                double balanceLessThanAmount = balance < amount;
                double newBalance = balance - amount;
                System.out.println("Withdrawing " + amount);
                System.out.println("New balance is " + newBalance);
                balance = newBalance;
                sufficientFundsCondition.signalAll();
            } finally {
                balanceChangeLock.unlock();
            }
        }
    }
}
```

BankAccount.java

```java
/**
 * 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;

    public BankAccount() {
        balance = 0;
        balanceChangeLock = new ReentrantLock();
        sufficientFundsCondition = balanceChangeLock.newCondition();
    }

    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();
        }
    }

    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;
            sufficientFundsCondition.signalAll();
        } finally {
            balanceChangeLock.unlock();
        }
    }

    public double getBalance() {
        return balance;
    }
}
```
**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 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 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.

---

**Algorithm Animation**

- Runs in a separate thread that periodically updates an image of the current state of the algorithm
- It then pauses so the user can see the change
- After a short time the algorithm thread wakes up and runs to the next point of interest
- It updates the image again and pauses again

---

**Selection Sort Algorithm Animation**

- Items in the algorithm’s state
  - The array of values
  - The size of the already sorted area
  - The currently marked element
- This state is accessed by two threads:
  1. One that sorts the array, and
  2. One that repaints the frame
- To visualize the algorithm
  - Show the sorted part of the array in a different color
  - Mark the currently visited array element in red
A Step in the Animation of the Selection Sort Algorithm

Selection Sort Algorithm Animation: Implementation

- Use a lock to synchronize access to the shared state
- Add a component instance variable to the algorithm class and augment the constructor to set it
- That instance variable is needed for
  - Repainting the component, and
  - Finding out the dimensions of the component when drawing the algorithm state

public class SelectionSorter
{
    private JComponent component;
    public SelectionSorter(int[] anArray, JComponent aComponent)
    {
        a = anArray;
        sortStateLock = new ReentrantLock();
        component = aComponent;
    }
    ...
}

Selection Sort Algorithm Animation: Implementation

- At each point of interest, algorithm needs to pause so user can observe the graphical output
- We need a pause method that repaints component and sleeps for a small delay:
  public void pause(int steps)
  throws InterruptedException
  {
      component.repaint();
      Thread.sleep(steps * DELAY);
  }
  - Delay is proportional to the number of steps involved
  - pause should be called at various places in the algorithm

Selection Sort Algorithm Animation: Implementation

- We add a draw method to the algorithm class
- draw draws the current state of the data structure, highlighting items of special interest
- draw is specific to the particular algorithm
- In this case, draws the array elements as a sequence of sticks in different colors
  - The already sorted portion is blue
  - The marked position is red
  - The remainder is black

  public void draw(Graphics2D g2)
  {
      sortStateLock.lock();
      try
      {
          int deltaX = component.getWidth() / a.length;
          for (int i = 0; i < a.length; i++)
          {
              if (i == markedPosition)
                  g2.setColor(Color.RED);
              else if (i <= alreadySorted)
                  g2.setColor(Color.BLUE);
              else
                  g2.setColor(Color.BLACK);
              g2.draw(new Line2D.Double(i * deltaX, 0, i * deltaX, a[i]));
          }
      }
  }
Selection Sort Algorithm Animation: draw (cont.)

```java
finally {
    sortStateLock.unlock();
}
```

Selection Sort Algorithm Animation: Pausing

- Update the special positions as the algorithm progresses
- Pause the animation whenever something interesting happens
- Pause should be proportional to the number of steps that are being executed
- In this case, pause one unit for each visited array element
- Augment `minimumPosition` and `sort` accordingly

```java
public int minimumPosition(int from) throws InterruptedException {
    int minPos = from;
    for (int i = from + 1; i < a.length; i++) {
        sortStateLock.lock();
        try {
            if (a[i] < a[minPos]) minPos = i;
            markedPosition = i;
        } finally {
            sortStateLock.unlock();
        }
        pause(2); // two array elements were inspected
    }
    return minPos;
}
```

Selection Sort Algorithm Animation: paintComponent

- `paintComponent` calls the draw method of the algorithm object:

```java
public class SelectionSortComponent extends JComponent {
    private SelectionSorter sorter;

    public void paintComponent(Graphics g) {
        if (sorter == null) return;
        Graphics2D g2 = (Graphics2D) g;
        sorter.draw(g2);
    }
    ...
}
```

Selection Sort Algorithm Animation: startAnimation

```java
public void startAnimation() {
    int[] values = ArrayUtil.randomIntArray(30, 300);
    sorter = new SelectionSorter(values, this);
    class AnimationRunnable implements Runnable {
        public void run() {
            try {
                sorter.sort();
            } catch (InterruptedException exception) {
            }
        }
    }
    AnimationRunnable r = new AnimationRunnable();
    ....
}
```
Selection Sort Algorithm Animation: startAnimation (cont.)

```java
Runnable r = new AnimationRunnable();
Thread t = new Thread(r);
t.start();
```

SelectionSortViewer.java

```java
import java.awt.BorderLayout;
import java.awt.Button;
import java.awt.Frame;
import java.awt.event.ActionEvent;

public class SelectionSortViewer
{
  public static void main(String[] args)
  {
    JFrame frame = new JFrame();
    final int FRAME_WIDTH = 300;
    final int FRAME_HEIGHT = 400;
    frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    final SelectionSortComponent component = new SelectionSortComponent();
    frame.add(component, BorderLayout.CENTER);
    frame.setVisible(true);
    component.startAnimation();
  }
}
```

SelectionSortComponent.java

```java
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.geom.Line2D;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class SelectionSortComponent extends JPanel
{
  private static final int DELAY = 100;
  private int[] values = ArrayUtil.randomIntArray(30, 300);
  private Lock sortStateLock;

  public SelectionSortComponent()
  {
    sorter = new SelectionSorter(values, this);
  }

  public void paintComponent(Graphics g)
  {
    Graphics2D g2 = (Graphics2D)g;
    sorter.draw(g2);
  }

  public void startAnimation()
  {
    class AnimationRunnable implements Runnable
    {
      public void run()
      {
        try
        {
          sorter.sort();
        }
        catch (InterruptedException exception)
        {
        }
    }

    Runnable r = new AnimationRunnable();
    Thread t = new Thread(r);
    t.start();
  }
}
```
SelectionSorter.java (cont.)

```java
/**
 * Constructs a selection sorter.
 * @param anArray the array to sort
 * @param aComponent the component to be repainted when the animation pauses
 */
public SelectionSorter(int[] anArray, JComponent aComponent)
{
    a = anArray;
    sortStateLock = new ReentrantLock();
    component = aComponent;
}
```

SelectionSorter.java (cont.)

```java
/**
 * Sorts the array managed by this selection sorter.
 */
public void sort()
throws InterruptedException
{
    for (int i = 0; i < a.length - 1; i++)
    {
        int minPos = minimumPosition(i);
        sortStateLock.lock();
        try
        {
            swap(minPos, i);
            // For animation
            alreadySorted = i;
        }
        finally
        {
            sortStateLock.unlock();
            pause(2);
        }
    }
}
```

SelectionSorter.java (cont.)

```java
private int minimumPosition(int from)
throws InterruptedException
{
    int minPos = from;
    for (int i = from + 1; i < a.length; i++)
    {
        sortStateLock.lock();
        try
        {
            if (a[i] < a[minPos]) minPos = i;
            // For animation
            markedPosition = i;
        }
        finally
        {
            sortStateLock.unlock();
            pause(2);
        }
    }
    return minPos;
}
```

SelectionSorter.java (cont.)

```java
private void swap(int i, int j)
{
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

SelectionSorter.java (cont.)

```java
/**
 * Draws the current state of the sorting algorithm.
 * @param g2 the graphics context
 */
public void draw(Graphics2D g2)
{
    sortStateLock.lock();
    try
    {
        int deltaX = component.getWidth() / a.length;
        for (int i = 0; i < a.length; i++)
        {
            if (i == markedPosition)
                g2.setColor(Color.RED);
            else if (i <= alreadySorted)
                g2.setColor(Color.BLUE);
            else
                g2.setColor(Color.BLACK);
            g2.draw(new Line2D.Double(i * deltaX, 0,
                                       i * deltaX, a[i]));
        }
    }
    finally
    {
        sortStateLock.unlock();
    }
}
```

SelectionSorter.java (cont.)

```java
/**
 * Pauses the animation.
 * @param steps the number of steps to pause
 */
public void pause(int steps)
throws InterruptedException
{
    component.repaint();
    Thread.sleep(steps * DELAY);
}
```
Self Check 11
Why is the draw method added to the SelectionSorter class and not the SelectionSortComponent class?

Answer: The draw method uses the array values and the values that keep track of the algorithm's progress. These values are available only in the SelectionSorter class.

Self Check 12
Would the animation still work if the startAnimation method simply called sorter.sort() instead of spawning a thread that calls that method?

Answer: Yes, provided you only show a single frame. If you modify the SelectionSortViewer program to show two frames, you want the sorters to run in parallel.