LECTURE 16

MULTI-THREADING 2: SYNCHRONIZATION

Slides by Rick Giles

Race Condition

- Suppose thread 1 is executing

```
x = 1;
y = 1;
a[x][y] = 1;
```

- Suppose thread 2 is executing

```
x = 2;
y = 2;
a[x][y] = 2;
```

Race Condition

- If one executes to completion before the other is started, there is no problem. However, suppose that the execution proceeds as follows:

Thread 1	Thread 2
x = 1;	
	x = 2;
	y = 2;
	a[x][y] = 2;
y = 1;	
a[x][y] = 1;	

The result is a[2][2] is set to 2, and a[2][1] is set to 1.

The race condition resulted in the wrong location being set.

Handling Race Conditions

- Prevent them by allowing only one thread to access the data at a time.
- When one finishes, the other can have access.
- This is called **synchronizing**.
- Access is controlled by having a lock.
  - When one thread has the lock for a resource, no other thread can access the resource until the thread holding the lock has released the lock.
- Java has two independent ways to handle locks:
  - Special lock objects
  - A built-in lock in every object

Special lock objects

```
private Lock ballAccessLock = new ReentrantLock();

…

ballAccessLock.lock();      // grab the lock

            …     // code to access and manipulate the shared resource

ballAccessLock.unlock();  // return the lock
```

The thread that grabbed a lock is said to own the lock.

If a thread A attempts to grab a lock, but it is currently owned by another thread,

then thread A is deactivated. From time-to-time, thread A is reactivated to try the lock again.

It is necessary to guarantee that any thread that grabs a lock eventually releases the lock, even if an exception occurs. The later situation is handed by using the **finally** clause of a try-catch.

synchronized

- Every object has a lock built into it.
- Two approaches to using this lock:
  1. Use the synchronized statement on the object.
  2. Place the synchronized modifier on a method.

synchronized (1)

- Use the synchronized statement on the object.

```
synchronized(ball)
{
        // the lock of the “ball” object is owned by this thread as long as
        //      the statements in the block are being executed
        // the lock is automatically released when the block is exited,
        //      whether normally or abnormally
        // no other thread can obtain the lock for this object
        //      when it is owned by this thread, but unsynchronized access is
        //      allowed by Java (the programmer should preclude this)
}
```

synchronized (2)

- Place the synchronized modifier on a method.

```
public synchronized void moveBallAndRepaint(Double x, Double y)
{
        // the lock for “this” object is held for the duration of the method
}
```

Note that this would obtain the lock of the panel, not the ball.

Bouncing Ball with Synchronization

```
public void shiftBallAndRepaint()
{
        synchronized(ball)
        {
                super.shiftBallAndRepaint();
        }
}

public void moveBallAndRepaint(Double x, Double y)
{
        Double newX;
                … (as before)
        Double newY;
                … (as before)
        synchronized(ball)
        {
                ball.setFrame(newX, newY, ball.getWidth(), ball.getHeight());
        }
        repaint();
}
```

20.3 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)

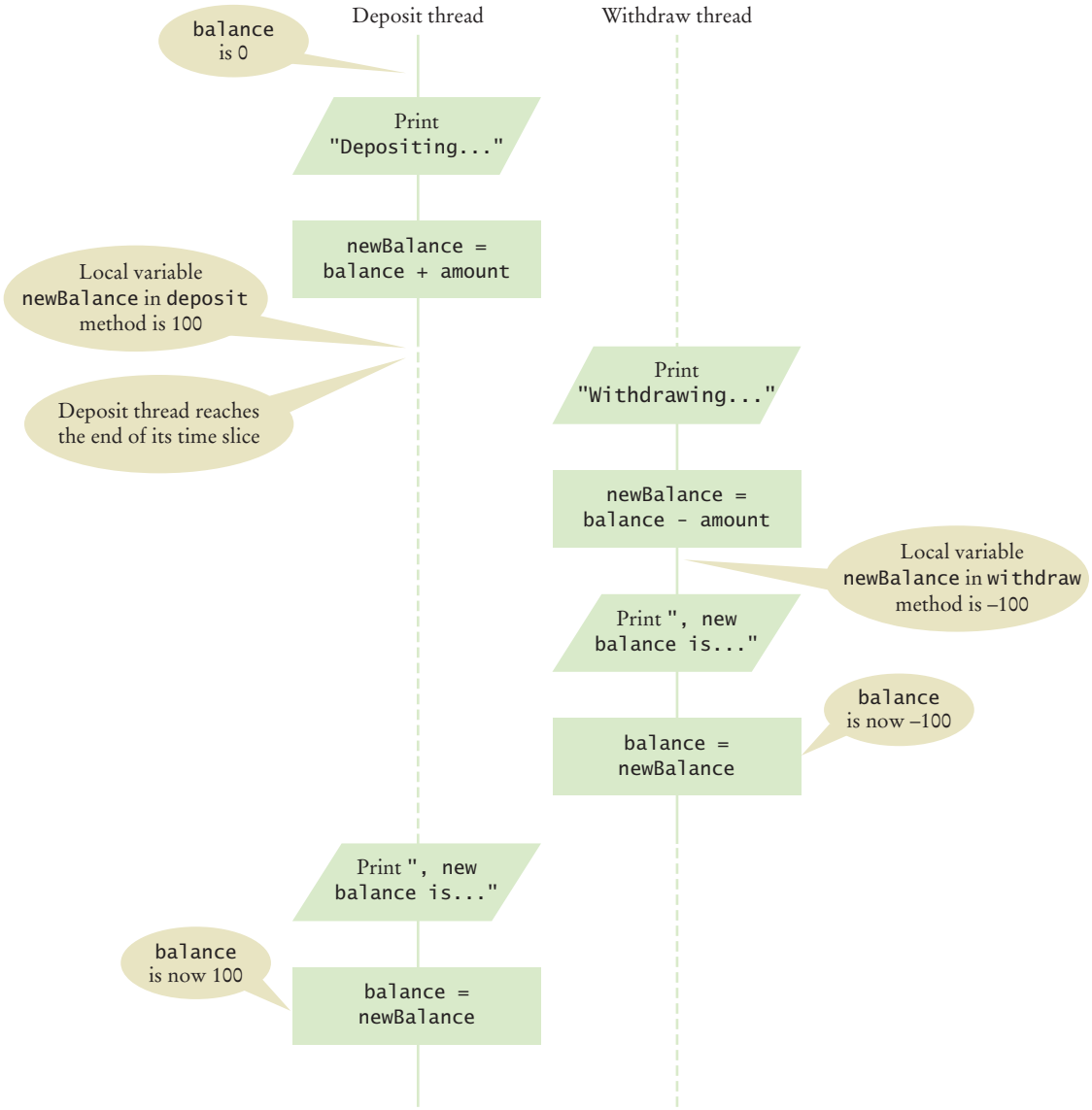
- Scenario to explain problem (cont.):

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
```

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
```

20.4 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

20.5 Avoiding Deadlocks

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

Thread 1:
- requests the lock for object A
- acquires lock on object A
- …
- requests the lock for object B
- waits for the lock on object B

Thread 2:
- requests the lock for object B
- acquires the lock on object B
- …
- requests the lock for object A
- waits for the lock on object A

Both threads now wait indefinitely.

It is the programmer's responsibility to ensure that this never happens.

20.5 Avoiding Deadlocks

- `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
```

20.6 Application: Algorithm Animation

- Animation shows different objects moving or changing as time progresses
- Often achieved by launching one or more threads that compute how parts of the animation change
- Can use Swing `Timer` class for simple animations
- More advanced animations are best implemented with thread
- Algorithm animation helps visualize the steps in the algorithm

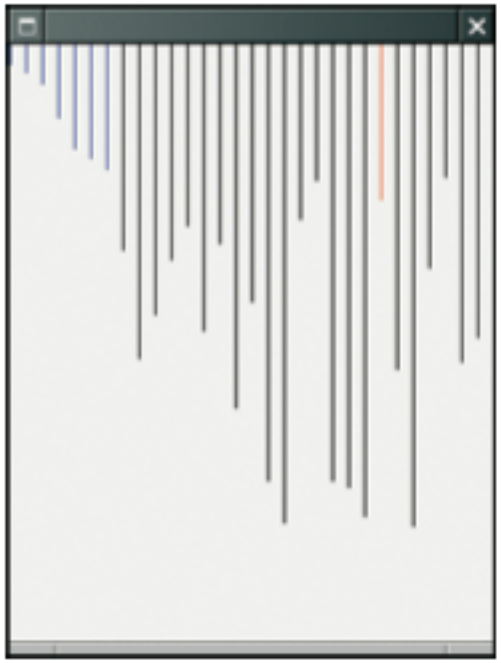
Algorithm Animation

- Runs in a separate thread that periodically updates an image of the current state of the algorithm
- Then pauses so the user can view the image
- After a short time the algorithm thread wakes up and runs to the next point of interest
- Updates the image again and pauses again
- Repeats sequence until algorithm has finished

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*

Selection Sort Algorithm Animation Step



Selection Sort Algorithm Animation: Implementation (1)

```
public class SelectionSorter
{
   private JComponent component;

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

Selection Sort Algorithm Animation: Implementation (2)

- 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 (3)

- 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*

Selection Sort Algorithm Animation: `draw` Method (1)

```
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);
         }
```

Continued

Selection Sort Algorithm Animation: `draw` Method (2)

```
            else
            {
               g2.setColor(Color.BLACK);
            }
            g2.draw(new Line2D.Double(i * deltaX, 0,
               i * deltaX, a[i]));
         }
      }
      finally
      {
         sortStateLock.unlock();
      }
}
```

Selection Sort Algorithm Animation: Pausing (1)

- 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

Selection Sort Algorithm Animation: `minimumPosition` Method (1)

```
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;
         // For animation
         markedPosition = i;
      }
      finally
      {
         sortStateLock.unlock();
```

Continued

Selection Sort Algorithm Animation: `minimumPosition` Method (2)

```
         }
         pause(2);
      }
      return minPos;
   }
```

Selection Sort Algorithm Animation: `paintComponent` Method

- Component's `paintComponent` calls the `draw` method of the algorithm object:

```
public class SelectionSortComponent extends JComponent
{
   private SelectionSorter sorter;
   . . .
   public void paintComponent(Graphics g)
   {
      sorter.draw(g);
   }
}
```

Selection Sort Algorithm Animation: `SelectionSortComponent` Constructor

- Constructs `SelectionSorter` object which supplies a new array and the `this` reference to the component that displays the sorted values:

```
public SelectionSortComponent()
{
   int[] values = ArrayUtil.randomIntArray(30, 300);
   sorter = new SelectionSorter(values, this);
}
```

Selection Sort Algorithm Animation: `startAnimation` Method (1)

- Constructs a thread that calls the sorter's sort method:

```
public SelectionSortComponent()
{
   int[] values = ArrayUtil.randomIntArray(30, 300);
   sorter = new SelectionSorter(values, this);
}
```

SelectionSortViewer.java

```
1  import java.awt.BorderLayout;
2  import javax.swing.JButton;
3  import javax.swing.JFrame;
4
5  public class SelectionSortViewer
6  {
7     public static void main(String[] args)
8     {
9        JFrame frame = new JFrame();
10
11       final int FRAME_WIDTH = 300;
12       final int FRAME_HEIGHT = 400;
13
14       frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
15       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16
17       final SelectionSortComponent component
18             = new SelectionSortComponent();
19       frame.add(component, BorderLayout.CENTER);
20
21       frame.setVisible(true);
22       component.startAnimation();
23    }
24 }
```

SelectionSortComponent.java

```
1  import java.awt.Graphics;
2  import javax.swing.JComponent;
3
4  /**
5     A component that displays the current state of the selection sort algorithm.
6  */
7  public class SelectionSortComponent extends JComponent
8  {
9     private SelectionSorter sorter;
10
11    /**
12       Constructs the component.
13    */
14    public SelectionSortComponent()
15    {
16       int[] values = ArrayUtil.randomIntArray(30, 300);
17       sorter = new SelectionSorter(values, this);
18    }
19
20    public void paintComponent(Graphics g)
21    {
```

Continued

SelectionSortComponent.java (cont)

```
22          sorter.draw(g);
23       }
24
25       /**
26          Starts a new animation thread.
27       */
28       public void startAnimation()
29       {
30          class AnimationRunnable implements Runnable
31          {
32             public void run()
33             {
34                try
35                {
36                   sorter.sort();
37                }
38                catch (InterruptedException exception)
39                {
40                }
41             }
42          }
43
44          Runnable r = new AnimationRunnable();
45          Thread t = new Thread(r);
46          t.start();
47       }
48    }
```

SelectionSorter.java

```
1  import java.awt.Color;
2  import java.awt.Graphics;
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5  import javax.swing.JComponent;
6
7  /**
8     This class sorts an array, using the selection sort algorithm.
9  */
10 public class SelectionSorter
11 {
12    // This array is being sorted
13    private int[] a;
14    // These instance variables are needed for drawing
15    private int markedPosition = -1;
16    private int alreadySorted = -1;
17
18    private Lock sortStateLock;
19
20    // The component is repainted when the animation is paused
21    private JComponent component;
22
23    private static final int DELAY = 100;
24
```

Continued

SelectionSorter.java (cont)

```
25    /**
26       Constructs a selection sorter.
27       @param anArray the array to sort
28       @param aComponent the component to be repainted when the animation
29       pauses
30    */
31    public SelectionSorter(int[] anArray, JComponent aComponent)
32    {
33       a = anArray;
34       sortStateLock = new ReentrantLock();
35       component = aComponent;
36    }
37
```

Continued

SelectionSorter.java (cont)

```
38    /**
39       Sorts the array managed by this selection sorter.
40    */
41    public void sort()
42          throws InterruptedException
43    {
44       for (int i = 0; i < a.length - 1; i++)
45       {
46          int minPos = minimumPosition(i);
47          sortStateLock.lock();
48          try
49          {
50             ArrayUtil.swap(a, minPos, i);
51             // For animation
52             alreadySorted = i;
53          }
54          finally
55          {
56             sortStateLock.unlock();
57          }
58          pause(2);
59       }
60    }
61
```

Continued

SelectionSorter.java (cont)

```
62    /**
63       Finds the smallest element in a tail range of the array.
64       @param from the first position in a to compare
65       @return the position of the smallest element in the
66       range a[from] . . . a[a.length - 1]
67    */
68    private int minimumPosition(int from)
69          throws InterruptedException
70    {
71       int minPos = from;
72       for (int i = from + 1; i < a.length; i++)
73       {
74          sortStateLock.lock();
75          try
76          {
77             if (a[i] < a[minPos]) { minPos = i; }
78             // For animation
79             markedPosition = i;
80          }
81          finally
82          {
83             sortStateLock.unlock();
84          }
85          pause(2);
86       }
87       return minPos;
88    }
89
```

Continued

SelectionSorter.java (cont)

```
90    /**
91       Draws the current state of the sorting algorithm.
92       @param g the graphics context
93    */
94    public void draw(Graphics g)
95    {
96       sortStateLock.lock();
97       try
98       {
99          int deltaX = component.getWidth() / a.length;
100         for (int i = 0; i < a.length; i++)
101         {
102            if (i == markedPosition)
103            {
104               g.setColor(Color.RED);
105            }
106            else if (i <= alreadySorted)
107            {
108               g.setColor(Color.BLUE);
109            }
110            else
111            {
112               g.setColor(Color.BLACK);
113            }
114            g.drawLine(i * deltaX, 0, i * deltaX, a[i]);
115         }
116      }
```

Continued

SelectionSorter.java (cont)

```
117          finally
118          {
119             sortStateLock.unlock();
120          }
121       }
122
123       /**
124          Pauses the animation.
125          @param steps the number of steps to pause
126       */
127       public void pause(int steps)
128             throws InterruptedException
129       {
130          component.repaint();
131          Thread.sleep(steps * DELAY);
132       }
133    }
```

Other Java tools for Threading

- Thread Thread.currentThread() // static function
  // returns the current thread

- t.interrupt( ); // sets a flag in the thread to true to indicate
  // an interrupt request
  // if the thread is waiting or sleeping
  // an InterruptedException is thrown

- boolean t.isInterrupted( ) // checks the interrupt flag for thread t

- boolean Thread.interrupted( ) // static method that checks the
  // interrupt flag for the current thread
  // sets the flag to false

Other Java tools for Threading

- Class Object has the method wait( ) that causes the current thread to wait (in this object) until another thread has issued a notify() or notifyAll() for the object.
- Class Object has methods notify( ) and notifyAll( ) to wake up one other (or all other) threads in the object specified as the target.
- A ReentrantLock can be temporarily released :
  - Suppose that there is a need to acquire a resource in a certain state
  - Acquire the resource lock
  - If not in the acceptable state
    - release the lock and enter a special “await” state
    - remains in the “await” state until some other thread issues a “signal” or a “signalAll”
    - try again

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.