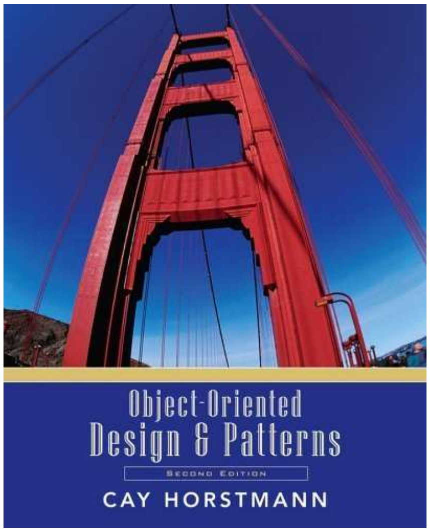
next

Object-Oriented Design & Patterns

Cay S. Horstmann

Chapter 9

Multithreading



next

previous | start | next

Chapter Topics

- Thread Basics
- Thread Synchronization
- Animations

previous | start | next

previous | start | next

Threads

- Thread: program unit that is executed independently
- Multiple threads run simultaneously
- Virtual machine executes each thread for short time slice
- Thread scheduler activates, deactivates threads
- Illusion of threads running in parallel
- Multiprocessor computers: threads actually run in parallel

previous | start | next

previous | start | next

Running Threads

- Define class that implements `Runnable`
- `Runnable` has one method
  `void run()`
- Place thread action into `run` method
- Construct object of runnable class
- Construct thread from that object
- Start thread

previous | start | next

previous | start | next

Running Threads

```
public class MyRunnable implements Runnable{   public void run()   {      thread action
   }
}
...
Runnable r = new MyRunnable();
Thread t = new Thread(t);
t.start();
```

previous | start | next

previous | start | next

Thread Example

- Run two threads in parallel
- Each thread prints 10 greetings

  ```
  for (int i = 1; i <= 10; i++){   System.out.println(i + ": " + greeting);   Thread.sleep(100);}
  ```

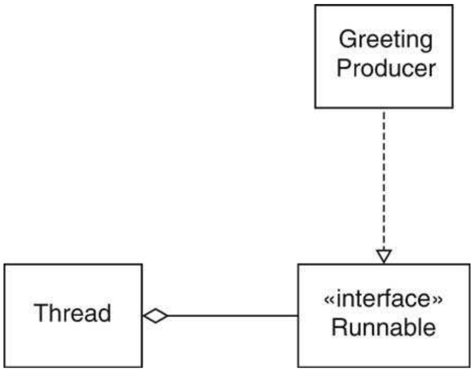
- After each printout, sleep for 100 millisec
- All threads should occasionally yield control
- `sleep` throws `InterruptedException`

previous | start | next

previous | start | next

Thread Example

- Ch9/greeting/GreetingProducer.java
- Ch9/greeting/ThreadTester.java



previous | start | next

```
/**
   An action that repeatedly prints a greeting.
*/
public class GreetingProducer implements Runnable
{
   /**
      Constructs the producer object.
      @param aGreeting the greeting to display
   */
   public GreetingProducer(String aGreeting)
   {
      greeting = aGreeting;
   }

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

   private String greeting;

   private static final int REPETITIONS = 10;
   private static final int DELAY = 100;
}
```

```
/**
   This program runs two threads in parallel.
*/
public class ThreadTester
{
   public static void main(String[] args)
   {
      Runnable r1 = new GreetingProducer("Hello, World!");
      Runnable r2 = new GreetingProducer("Goodbye, World!");

      Thread t1 = new Thread(r1);
      Thread t2 = new Thread(r2);

      t1.start();
      t2.start();
   }
}

```

previous | start | next

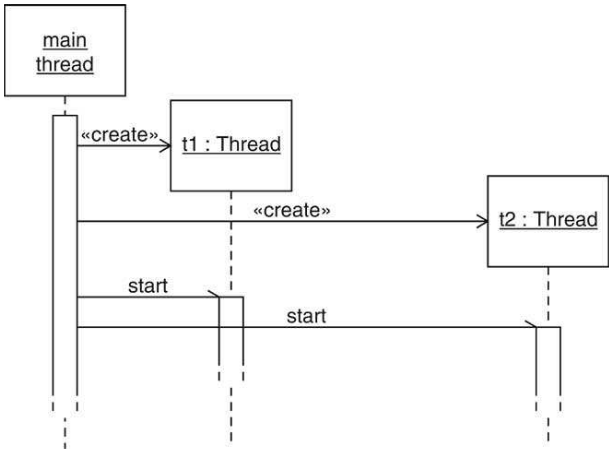
Thread Example

- Note: output not exactly interleaved

previous | start | next

previous | start | next

Starting Two Threads



previous | start | next

previous | start | next

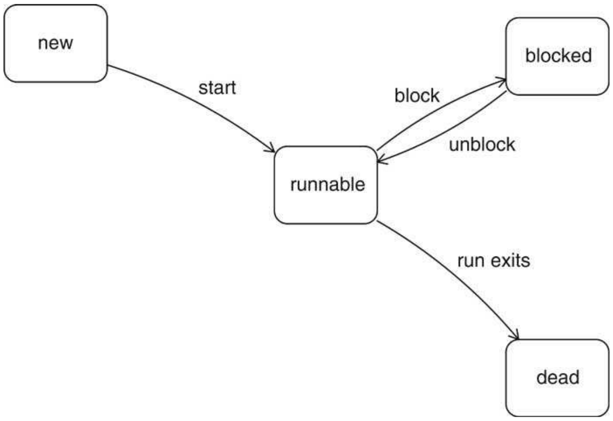
Thread States

- Each thread has
  - thread state
  - priority
- Thread states:
  - new (before `start` called)
  - runnable
  - blocked
  - dead (after `run` method exits)

previous | start | next

previous | start | next

Thread States



previous | start | next

previous | start | next

Blocked Thread State

- Reasons for blocked state:
  - Sleeping
  - Waiting for I/O
  - Waiting to acquire lock (later)
  - Waiting for condition (later)
- Unblocks only if reason for block goes away

previous | start | next

previous | start | next

Scheduling Threads

- Scheduler activates new thread if
  - a thread has completed its time slice
  - a thread has blocked itself
  - a thread with higher priority has become runnable
- Scheduler determines new thread to run
  - looks only at runnable threads
  - picks one with max priority

previous | start | next

Terminating Threads

- Thread terminates when `run` exits
- Sometimes necessary to terminate running thread
- Don’t use deprecated `stop` method
- Interrupt thread by calling `interrupt`
- Calling `t.interrupt()` doesn’t actually interrupt `t`; just sets a flag
- Interrupted thread must sense interruption and exit its `run` method
- Interrupted thread has chance to clean up

Sensing Interruptions

- Thread could occasionally call `Thread.currentThread().isInterrupted()`
- `sleep`, `wait` throw `InterruptedException` when thread interrupted
- `. . .` and then the interruption status is cleared!
- More robust: Sleep occasionally, catch exception and react to interruption
- Recommendation: Terminate `run` when sensing interruption

Sensing Interruptions

```
public class MyRunnable implements Runnable{   public void run()   {      try      {         while (...)         {            do work                        Thread.sleep(...);         }         }      catch (InterruptedException e) {}      clean up
   }
}
```

Thread Synchronization

- Use bounded queue from chapter 3
- Each producer thread inserts greetings
- Each consumer thread removes greetings
- Two producers, one consumer

previous | start | next

Producer Thread

```
int i = 1; while (i <= greetingCount) {    if (!queue.isFull())    {       queue.add(i + ": " + greeting);       i++;    }    Thread.sleep((int)(Math.random() * DELAY)); }
```

previous | start | next

previous | start | next

Consumer Thread

```
int i = 1; while (i <= greetingCount) {    if (!queue.isEmpty())    {       Object greeting = queue.remove();       System.out.println(greeting);       i++;    }    Thread.sleep((int)(Math.random() * DELAY)); }
```

previous | start | next

previous | start | next

Expected Program Output

```
1: Hello, World! 1: Goodbye, World! 2: Hello, World! 3: Hello, World! ... 99: Goodbye, World! 100: Goodbye, World!
```

previous | start | next

previous | start | next

Why is Output Corrupted?

- Sometimes program gets stuck and doesn’t complete
- Can see problem better when turning debugging on `queue.setDebug(true);`

- Ch9/queue1/ThreadTester.java
- Ch9/queue1/Producer.java
- Ch9/queue1/Consumer.java
- Ch9/queue1/BoundedQueue.java

previous | start | next

```java
/**
   This program runs two threads in parallel.
*/
public class ThreadTester
{
   public static void main(String[] args)
   {
      BoundedQueue<String> queue = new BoundedQueue<String>(10);
      queue.setDebug(true);
      final int GREETING_COUNT = 100;
      Runnable run1 = new Producer("Hello, World!",
            queue, GREETING_COUNT);
      Runnable run2 = new Producer("Goodbye, World!",
            queue, GREETING_COUNT);
      Runnable run3 = new Consumer(queue, 2 * GREETING_COUNT);

      Thread thread1 = new Thread(run1);
      Thread thread2 = new Thread(run2);
      Thread thread3 = new Thread(run3);

      thread1.start();
      thread2.start();
      thread3.start();
   }
}

```

```java
/**
   An action that repeatedly inserts a greeting into a queue.
*/
public class Producer implements Runnable
{
   /**
       Constructs the producer object.
       @param aGreeting the greating to insert into a queue
       @param aQueue the queue into which to insert greetings
       @param count the number of greetings to produce
    */
   public Producer(String aGreeting, BoundedQueue<String> aQueue, int count)
   {
      greeting = aGreeting;
      queue = aQueue;
      greetingCount = count;
  }

   public void run()
   {
      try
      {
         int i = 1;
         while (i <= greetingCount)
         {
            if (!queue.isFull())
            {
               queue.add(i + ": " + greeting);
               i++;
            }
            Thread.sleep((int) (Math.random() * DELAY));
         }
      }
      catch (InterruptedException exception)
      {
      }
   }

   private String greeting;
   private BoundedQueue<String> queue;
   private int greetingCount;

   private static final int DELAY = 10;
}
```

```java
/**
   An action that repeatedly removes a greeting from a queue.
*/
public class Consumer implements Runnable
{
   /**
       Constructs the consumer object.
       @param aQueue the queue from which to retrieve greetings
       @param count the number of greetings to consume
   */
   public Consumer(BoundedQueue<String> aQueue, int count)
   {
      queue = aQueue;
      greetingCount = count;
   }

   public void run()
   {
      try
      {
         int i = 1;
         while (i <= greetingCount)
         {
            if (!queue.isEmpty())
            {
               String greeting = queue.remove();
               System.out.println(greeting);
               i++;
            }
            Thread.sleep((int)(Math.random() * DELAY));
         }
      }
      catch (InterruptedException exception)
      {
      }
   }

   private BoundedQueue<String> queue;
   private int greetingCount;

   private static final int DELAY = 10;
}
```

```java
/**
    A first-in, first-out bounded collection of objects.
*/
public class BoundedQueue<E>
{
   /**
       Constructs an empty queue.
       @param capacity the maximum capacity of the queue
    */
   public BoundedQueue(int capacity)
   {
      elements = new Object[capacity];
      head = 0;
      tail = 0;
      size = 0;
   }

   /**
       Removes the object at the head.
       @return the object that has been removed from the queue
       @precondition !isEmpty()
    */
   public E remove()
   {
      if (debug) System.out.print("removeFirst");
      E r = (E) elements[head];
      if (debug) System.out.print(".");
      head++;
      if (debug) System.out.print(".");
      size--;
      if (head == elements.length)
      {
         if (debug) System.out.print(".");
         head = 0;
      }
      if (debug)
         System.out.println("head=" + head + ",tail=" + tail
            + ",size=" + size);
      return r;
   }

   /**
       Appends an object at the tail.
       @param newValue the object to be appended
```

```
         @precondition !isFull();
      */
   public void add(E newValue)
   {
      if (debug) System.out.print("add");
      elements[tail] = newValue;
      if (debug) System.out.print(".");
      tail++;
      if (debug) System.out.print(".");
      size++;
      if (tail == elements.length)
      {
         if (debug) System.out.print(".");
         tail = 0;
      }
      if (debug)
         System.out.println("head=" + head + ",tail=" + tail
            + ",size=" + size);
   }

   public boolean isFull()
   {
      return size == elements.length;
   }

   public boolean isEmpty()
   {
      return size == 0;
   }

   public void setDebug(boolean newValue)
   {
      debug = newValue;
   }

   private Object[] elements;
   private int head;
   private int tail;
   private int size;
   private boolean debug;
}
```

previous | start | next

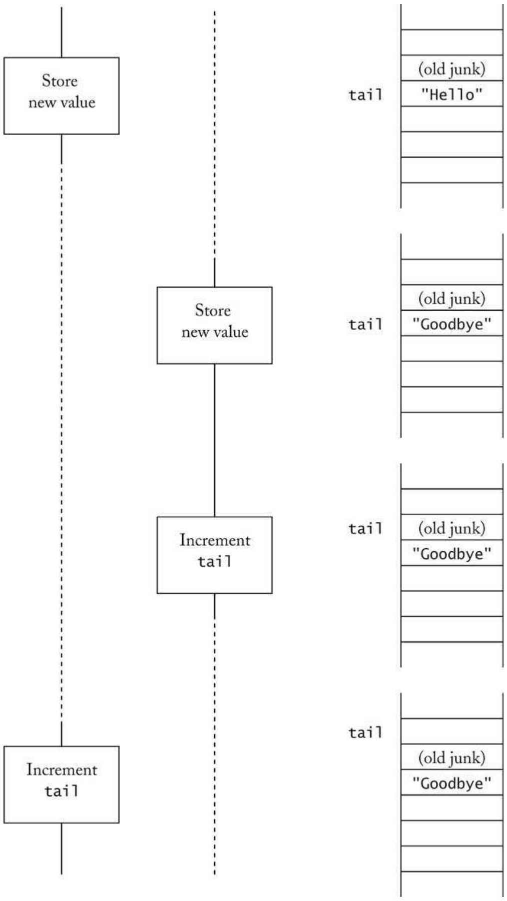
Race Condition Scenario

- First thread calls `add` and executes `elements[tail] = anObject;`
- First thread at end of time slice
- Second thread calls `add` and executes `elements[tail] = anObject;` `tail++;`
- Second thread at end of time slice
- First thread executes `tail++;`

previous | start | next

previous | start | next

Race Condition Scenario



previous | start | next

previous | start | next

Locks

- Thread can *acquire* lock
- When another thread tries to acquire same lock, it blocks
- When first thread *releases* lock, other thread is unblocked and tries again
- Two kinds of locks
  - Objects of class implementing `java.util.concurrent.Lock` interface type, usually `ReentrantLock`
  - Locks that are built into every Java object

previous | start | next

previous | start | next

Reentrant Locks

```
aLock = new ReentrantLock();. . .aLock.lock();try{   protected code}finally{   aLock.unlock();}
```

previous | start | next

previous | start | next

Scenario with Locks

1. First thread calls `add` and acquires lock, then executes `elements[tail] = anObject;`
2. Second thread calls `add` and tries to acquire lock, but it is blocked
3. First thread executes `tail++;`
4. First thread completes `add`, releases lock
5. Second thread unblocked
6. Second thread acquires lock, starts executing protected code

previous | start | next

previous | start | next

Deadlocks

- Not enough to synchronize `add`, `remove`
- `if (!queue.isFull()) queue.add(...);` can still be interrupted
- Must move test inside `add` method

```
public void add(E newValue){   queueLock.lock();   try   {      while (queue is full)         wait for more space      . . .   }   finally { qeueLock.unlock(); }}
```

- Problem: nobody else can call `remove`

previous | start | next

previous | start | next

Avoiding Deadlocks

- Use *condiiton* object to manage "space available" condition

```
private Lock queueLock = new ReentrantLock();private Condition spaceAvailableCondition   = queueLock.newCondition();
```

- Call `await` when condition is not fulfilled:

```
public void add(E newValue){   . . .   while (size == elements.length)      spaceAvailableCondition.await();   . . .}
```

previous | start | next

previous | start | next

Avoiding Deadlocks

- Waiting thread is blocked
- Condition object manages set of threads that wait for the condition to change
- To unblock, another thread must call `signalAll` on the same condition object
- Call when state changes

```
public E remove(){   . . .   E r = elements[head];   . . .   spaceAvailableCondition.signalAll(); // Unblock waiting threads   return r;}
```

- `All waiting threads removed from wait set, unblocked`
- `Ch9/queue2/BoundedQueue.java`

previous | start | next

```
import java.util.concurrent.locks.*;

/**
   A first-in, first-out bounded collection of objects.
*/
public class BoundedQueue<E>
{
   /**
       Constructs an empty queue.
       @param capacity the maximum capacity of the queue
   */
   public BoundedQueue(int capacity)
   {
      elements = new Object[capacity];
      head = 0;
      tail = 0;
      size = 0;
   }

   /**
       Removes the object at the head.
       @return the object that has been removed from the queue
   */
   public E remove() throws InterruptedException
   {
      queueLock.lock();
      try
      {
         while (size == 0)
            valueAvailableCondition.await();
         E r = (E) elements[head];
         head++;
         size--;
         if (head == elements.length)
            head = 0;
         spaceAvailableCondition.signalAll();
         return r;
      }
      finally
      {
         queueLock.unlock();
      }
   }

   /**
      Appends an object at the tail.
      @param newValue the object to be appended
   */
   public void add(E newValue) throws InterruptedException
   {
      queueLock.lock();
      try
      {
         while (size == elements.length)
            spaceAvailableCondition.await();
         elements[tail] = newValue;
         tail++;
         size++;
         if (tail == elements.length)
            tail = 0;
         valueAvailableCondition.signalAll();
      }
      finally
      {
         queueLock.unlock();
      }
   }

   private Object[] elements;
   private int head;
   private int tail;
   private int size;

   private Lock queueLock = new ReentrantLock();
   private Condition spaceAvailableCondition
         = queueLock.newCondition();
   private Condition valueAvailableCondition
         = queueLock.newCondition();
}
```

previous | start | next

Object Locks

- Each object has a lock
- Calling a `synchronized` method acquires lock of implicit parameter
- Leaving the `synchronized` method releases lock
- Easier than explicit `Lock` objects

```
public class BoundedQueue<E>{   public synchronized void add(E newValue) { . . . }   public synchronized E remove() { . . . }   . . .}
```

previous | start | next

previous | start | next

Object Locks

- Each implicit lock has one associated (anonymous) condition object
- `Object.wait` blocks current thread and adds it to wait set
- `Object.notifyAll` unblocks waiting threads

```
public synchronized void add(E newValue)   throws InterruptedException{   while (size == elements.length) wait();   elements[tail] = anObject;   . . .   notifyAll(); // notifies threads waiting to remove elements}
```

- Ch9/queue3/BoundedQueue.java

previous | start | next

```
/**
    A first-in, first-out bounded collection of objects.
*/
public class BoundedQueue<E>
{
   /**
       Constructs an empty queue.
       @param capacity the maximum capacity of the queue
   */
   public BoundedQueue(int capacity)
   {
      elements = new Object[capacity];
      head = 0;
      tail = 0;
      size = 0;
   }

   /**
       Removes the object at the head.
       @return the object that has been removed from the queue
   */
   public synchronized E remove()
         throws InterruptedException
   {
      while (size == 0) wait();
      E r = (E) elements[head];
      head++;
      size--;
      if (head == elements.length)
         head = 0;
      notifyAll();
      return r;
   }

   /**
       Appends an object at the tail.
       @param newValue the object to be appended
   */
   public synchronized void add(E newValue)
         throws InterruptedException
   {
      while (size == elements.length) wait();
      elements[tail] = newValue;
      tail++;
      size++;
      if (tail == elements.length)
         tail = 0;
      notifyAll();
   }

   private Object[] elements;
   private int head;
   private int tail;
   private int size;
}
```

previous | start | next

Visualizing Locks

- Object = phone booth
- Thread = person
- Locked object = closed booth
- Blocked thread = person waiting for booth to open

previous | start | next

previous | start | next

Visualizing Locks

previous | start | next

Algorithm Animation

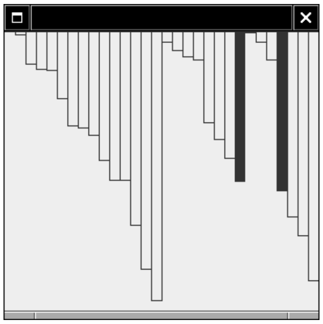
- Use thread to make progress in algorithm
- Display algorithm state
- Example: Animate Ch9/animation/Sorter.java
- Sleeps inside `compare` method
- Pass custom comparator

```
Comparator<Double> comp = new
   Comparator<Double>()
   {
      public void compare(Double d1, Double
d2)
      {
         sleep
         return comparison result
      }
   };
```

```
import java.util.*;

/**
    This class carries out the merge sort algorithm.
*/
public class MergeSorter
{
   /**
       Sorts an array, using the merge sort algorithm.
       @param a the array to sort
       @param comp the comparator to compare array elements
    */
   public static <E> void sort(E[] a, Comparator<? super E> comp)
   {
      mergeSort(a, 0, a.length - 1, comp);
   }

   /**
       Sorts a range of an array, using the merge sort
       algorithm.
       @param a the array to sort
       @param from the first index of the range to sort
       @param to the last index of the range to sort
       @param comp the comparator to compare array elements
    */
   private static <E> void mergeSort(E[] a, int from, int to,
      Comparator<? super E> comp)
   {
      if (from == to) return;
      int mid = (from + to) / 2;
      // Sort the first and the second half
      mergeSort(a, from, mid, comp);
      mergeSort(a, mid + 1, to, comp);
      merge(a, from, mid, to, comp);
   }

   /**
       Merges two adjacent subranges of an array
       @param a the array with entries to be merged
       @param from the index of the first element of the
          first range
       @param mid the index of the last element of the
          first range
       @param to the index of the last element of the
          second range
       @param comp the comparator to compare array elements
    */
   private static <E> void merge(E[] a,
      int from, int mid, int to, Comparator<? super E> comp)
   {
      int n = to - from + 1;
         // Size of the range to be merged

      // Merge both halves into a temporary array b
      Object[] b = new Object[n];

      int i1 = from;
         // Next element to consider in the first range
      int i2 = mid + 1;
         // Next element to consider in the second range
      int j = 0;
         // Next open position in b

      // As long as neither i1 nor i2 past the end, move
      // the smaller element into b
      while (i1 <= mid && i2 <= to)
      {
         if (comp.compare(a[i1], a[i2]) < 0)
         {
            b[j] = a[i1];
            i1++;
         }
         else
         {
            b[j] = a[i2];
            i2++;
         }
         j++;
      }

      // Note that only one of the two while loops
      // below is executed

      // Copy any remaining entries of the first half
      while (i1 <= mid)
      {
         b[j] = a[i1];
         i1++;
         j++;
      }

      // Copy any remaining entries of the second half
      while (i2 <= to)
      {
         b[j] = a[i2];
         i2++;
         j++;
      }

      // Copy back from the temporary array
      for (j = 0; j < n; j++)
         a[from + j] = (E) b[j];
   }
}
```

previous | start | next

Algorithm Animation



- Ch9/animation1/ArrayComponent.java
- Ch9/animation1/AnimationTester.java

previous | start | next

```java
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
   This component draws an array and marks two elements in the
   array.
*/
public class ArrayComponent extends JComponent
{
   public synchronized void paintComponent(Graphics g)
   {
      if (values == null) return;
      Graphics2D g2 = (Graphics2D) g;
      int width = getWidth() / values.length;
      for (int i = 0; i < values.length; i++)
      {
         Double v =  values[i];
         Rectangle2D bar = new Rectangle2D.Double(
            width * i, 0, width, v);
         if (v == marked1 || v == marked2)
            g2.fill(bar);
         else
            g2.draw(bar);
      }
   }

   /**
       Sets the values to be painted.
       @param values the array of values to display
       @param marked1 the first marked element
       @param marked2 the second marked element
    */
   public synchronized void setValues(Double[] values,
      Double marked1, Double marked2)
   {
      this.values = (Double[]) values.clone();
      this.marked1 = marked1;
      this.marked2 = marked2;
      repaint();
   }

```

```java
   private Double[] values;
   private Double marked1;
   private Double marked2;
}
```

```java
import java.awt.*;
import javax.swing.*;

/**
   This program animates a sort algorithm.
*/
public class AnimationTester
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      ArrayComponent panel = new ArrayComponent();
      frame.add(panel, BorderLayout.CENTER);

      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setVisible(true);

      Double[] values = new Double[VALUES_LENGTH];
      for (int i = 0; i < values.length; i++)
         values[i] = Math.random() * panel.getHeight();

      Runnable r = new Sorter(values, panel);
      Thread t = new Thread(r);
      t.start();
   }

   private static final int VALUES_LENGTH = 30;
   private static final int FRAME_WIDTH = 300;
   private static final int FRAME_HEIGHT = 300;
}
```

previous | start

Pausing and Running the Animation

- Want to pause animation until "Run" or "Step" button is clicked
- Need to coordinate UI thread, animation thread
- Try to use built-in thread-safe construct in `java.util.concurrent`
- Trick: Use a blocking queue
- Button click adds string "Run" or "Step" to queue
- Animation thread calls `take` on the queue, blocks if no string inserted
- Ch9/animation2/Sorter.java
- Ch9/animation2/AnimationTester.java

previous | start

```java
import java.util.*;
import java.util.concurrent.*;

/**
   This runnable executes a sort algorithm.
   When two elements are compared, the algorithm
   pauses and updates a panel.
*/
public class Sorter implements Runnable
{
   public Sorter(Double[] values, ArrayComponent panel, BlockingQueue<String> queue)
   {
      this.values = values;
      this.panel = panel;
      this.queue = queue;
   }

   public void run()
   {
      Comparator<Double> comp = new
         Comparator<Double>()
         {
            public int compare(Double d1, Double d2)
            {
               try
               {
                  String command = queue.take();
                  if (command.equals("Run"))
                  {
                     Thread.sleep(DELAY);
                     if (!"Step".equals(queue.peek()))
                        queue.add("Run");
                  }
               }
               catch (InterruptedException exception)
               {
                  Thread.currentThread().interrupt();
               }
               panel.setValues(values, d1, d2);
               return d1.compareTo(d2);
            }
         };
      MergeSorter.sort(values, comp);
      panel.setValues(values, null, null);
   }

   private Double[] values;
   private ArrayComponent panel;
   private BlockingQueue<String> queue;
   private static final int DELAY = 100;
}
```

```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.concurrent.*;

/**
   This program animates a sort algorithm.
*/
public class AnimationTester
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      ArrayComponent panel = new ArrayComponent();
      frame.add(panel, BorderLayout.CENTER);

      JButton stepButton = new JButton("Step");
      final JButton runButton = new JButton("Run");

      JPanel buttons = new JPanel();
      buttons.add(stepButton);
      buttons.add(runButton);
      frame.add(buttons, BorderLayout.NORTH);
      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setVisible(true);

      Double[] values = new Double[VALUES_LENGTH];
      for (int i = 0; i < values.length; i++)
         values[i] = Math.random() * panel.getHeight();

      final BlockingQueue<String> queue = new LinkedBlockingQueue<String>();
      queue.add("Step");

      final Sorter sorter = new Sorter(values, panel, queue);

      stepButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               queue.add("Step");
               runButton.setEnabled(true);
```

```
        }
      });

    runButton.addActionListener(new
      ActionListener()
      {
        public void actionPerformed(ActionEvent event)
        {
          runButton.setEnabled(false);
          queue.add("Run");
        }
      });

    Thread sorterThread = new Thread(sorter);
    sorterThread.start();
  }

  private static final int FRAME_WIDTH = 300;
  private static final int FRAME_HEIGHT = 300;
  private static final int VALUES_LENGTH = 30;
}
```