Multithreading

Horstmann Chapter 9

Unit 6
CS 3354
Spring 2017

Jill Seaman

Threads

- What is a process?
  - ✦a self-contained running program with its own address space.
  - ✦processes are controlled by the operating system.
- What is a thread?
  - ✦A thread is an execution stream *within* a process.
- A thread is also called a lightweight process.
  - ✦Has its own execution stack, local variables, and program counter.
  - ✦Very much like a process, but it runs within a process.
- There may be more than one thread in a process.
  - ✦Is called a **multithreaded** process.

Multithreading

- Multithreading:
  - ✦Provides the capability to run tasks in parallel for a process.
  - ✦All threads **share** with each other **resources** allocated to the process.
  - ✦In fact, they compete and may interfere with each other.
- Threads allow the programmer to turn a program into separate, independently running subtasks
- In all cases, thread programming:
  1. Seems mysterious and requires a shift in the way you think about programming
  2. Looks similar to thread support in other languages, so when you understand threads, you understand a common tongue

9.1 Thread Basics (Threads in Java)

- "In general, you'll have some part of your program tied to a particular event or resource, and you don't want that to hold up the rest of your program. So, you create a thread associated with that event or resource and let it run independently of the main program."
- The java.lang.Thread class has all the wiring necessary to create and run threads.
- The run( ) method contains the code that will be executed "simultaneously" with the other threads in a program
- The Java Thread class provides a generic thread that, by default, does nothing.
  - ✦Its run() method is empty, and should be overridden by all subclasses of Thread.

The Runnable Interface

- The java.lang.Runnable Interface
  - ✦This interface should be implemented by any class whose instances are intended to be executed by a thread (but do not want to or cannot subclass Thread).
  - ✦The class must define a method of no arguments called run.
  - ✦A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target.
- Runnable is implemented by the class java.lang.Thread.

Threads in Java

- There are two techniques to implement threads in Java:
  - ✦To subclass Thread and override run().
  - ✦To implement the Runnable interface (by defining run()) and embed class instances in a Thread object.

This allows a class to have a superclass other than Thread, but still implement a thread.

- Once a Thread instance is created, call the start() method to make it run.
  - ✦ This causes the run() method to be executed in a separate thread.
  - ✦ The code following the call to start() will execute concurrently with the thread's run method.

Subclassing Thread: example

```
public class YinYang extends Thread {
     private String word;             // what to say

     public YinYang(String whatToSay) {
         word = whatToSay;
     }

     public void run() {
         for (int i = 0; i < 10; i++) {
             System.out.print(word + " ");
             yield();                 // to give another thread a chance
         }
     }

     public static void main(String[] args) {
         YinYang yin = new YinYang("Yin");    // to create Yin thread
         YinYang yang = new YinYang("Yang");  // to create Yang thread
         yin.start();                         // to start Yin thread
         yang.start();                        // to start Yang thread
     }
 }
```

output:
```
Yin Yang Yang Yang Yin Yang Yin Yang Yin Yang
Yin Yang Yin Yang Yin Yang Yin Yang Yin Yin
```

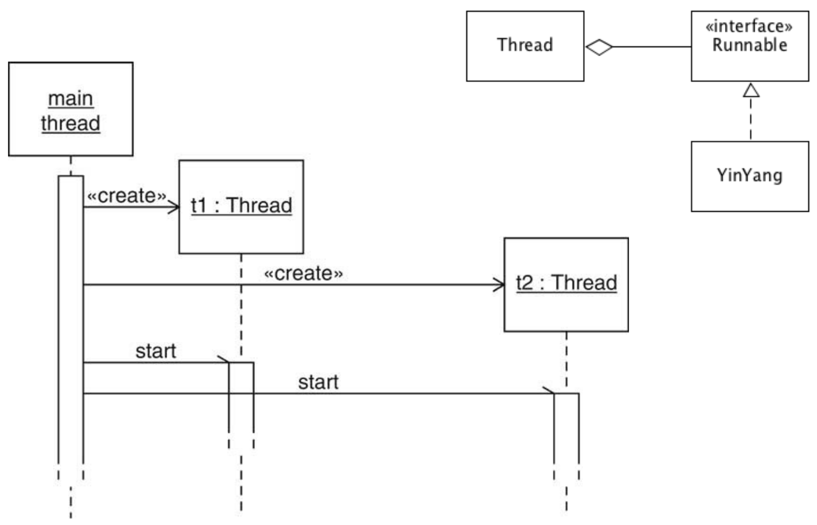
Implementing Runnable: example

```
public class YangYin implements Runnable {
    private String word;             // what to say
    public YangYin(String whatToSay) {
        word = whatToSay;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.print(word + " ");
            Thread.yield();          // to give another thread a chance
        }
    }
    public static void main(String[] args) {
        Runnable rYang = new YangYin("Yang");   // to instantiate YangYin
        Runnable rYin = new YangYin("Yin");     // to instantiate again

        Thread yang  = new Thread(rYang);       // to create Yang thread
        Thread yin  = new Thread(rYin);         // to create Yin thread
        yang.start();                           // to start Yang thread
        yin.start();                            // to start Yin thread
    }
}
```

output:
```
Yin Yin Yang Yin Yang Yin Yang Yang Yin Yang Yin
Yang Yang Yin Yang Yin Yang Yin Yang Yin
```

Class Diagram and Sequence Diagram



Scheduling Threads

- Each thread runs for a short amount of time (a time slice).
- Then the scheduler activates another thread
- The thread scheduler gives no guarantee about the order in which threads are executed (so the yin/yang output is not perfectly interleaved).

- The scheduler activates a new thread when:
  - the running thread finishes its time slice or
  - the running thread blocks itself (it's sleeping or waiting for some other event to occur)

Thread methods

- run()
  - ✦The code that will be run concurrently (in its own thread)
- start()
  - ✦Causes the run method to execute in a separate thread, continues execution (immediately returns control to caller).
- yield()
  - ✦Causes the currently executing thread object to temporarily pause and allow other threads to execute.
- getName()
  - ✦Returns this thread's name (set in the constructor).

Thread methods

- sleep(long milllis)
  - ✦Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds
- join()
  - ✦Causes the calling thread to wait for this thread to complete before proceeding.
- interrupt()
  - ✦Called from outside the thread to interrupt a thread that is paused via sleep(), or join().
  - ✦InterruptedException is generated in the sleep/join
  - ✦Calls to sleep/join must be in a try/catch block

```
public final void join() throws InterruptedException
```

Interrupt() example: Sleeper

```
class Sleeper extends Thread {
    private int duration;
    public Sleeper(String name, int sleepTime) {
        super(name);
        duration = sleepTime;
        start();              //starts itself
    }
    public void run() {
        try {
            sleep(duration);   //sleeps for a bit
        } catch (InterruptedException e) {
            System.out.println(getName() + " was interrupted.");
            return;
        }
        System.out.println(getName() + " has awakened");
    }
}
```

Interrupt() example: Joiner

```
class Joiner extends Thread {
    private Sleeper sleeper;
    public Joiner(String name, Sleeper sleeper) {
        super(name);
        this.sleeper = sleeper;
        start();             //starts itself
    }
    public void run() {
        try {
            sleeper.join();    //waits for sleeper to wake up
        } catch (InterruptedException e) {
            System.out.println(getName() + " was interrupted. ");
            return;
        }
        System.out.println(getName() + " join completed");
    }
}
```

Interrupt() example: JoiningTester

```
public class JoiningTester {
    public static void main(String[] args) {
      Sleeper
        sleepy = new Sleeper("Sleepy", 1500),
        grumpy = new Sleeper("Grumpy", 1500);
      Joiner
        dopey = new Joiner("Dopey", sleepy),
        doc = new Joiner("Doc", grumpy);
      // grumpy.interrupt();  or doc.interrupt();
    }
}
```

No interrupt():

```
Sleepy has awakened
Grumpy has awakened
Dopey join completed
Doc join completed
```

grumpy.interrupt():

```
Grumpy was interrupted.
Doc join completed
Sleepy has awakened
Dopey join completed
```

doc.interrupt():

```
Doc was interrupted.
Sleepy has awakened
Grumpy has awakened
Dopey join completed
```

9.2 Thread synchronization

- We now have the possibility of two or more threads trying to use the same limited resource (i.e. a queue) at once.
- Each Producer thread inserts a (numbered) greeting into a queue.
- A Consumer thread removes greetings from the same queue, and outputs them to the screen.
- Our example will have two Producers and one Consumer.
- Each Producer inserts n copies of its greeting into the queue, so the Consumer needs to remove 2*n greetings.
- The BoundedQueue is the standard implementation that stores a queue in a “circular array” (it wraps back to the front of the array).

The BoundedQueue

- dequeue is called remove, enqueue is called add.

```
public class BoundedQueue<E> {
    private Object[] elements;
    private int front, rear, numItems;
    public BoundedQueue(int capacity)    {
        elements = new Object[capacity];
        front = numItems = 0; rear = -1;
    }
    public E remove() {
        E r = (E) elements[front];
        front = (front+1)%elements.length;
        numItems--;
        return r;
    }
    public void add(E newValue) {
        rear = (rear+1)%elements.length;
        elements[rear] = newValue;
        numItems++;
    }
    public boolean isFull() {  return numItems == elements.length; }
    public boolean isEmpty(){  return numItems == 0;  }
}
```

Producer Thread

- Producer adds greetings to the queue:

```
public class Producer implements Runnable {
    private String greeting;
    private BoundedQueue<String> queue;  //reference to the shared queue
    private int greetingCount;
    private static final int DELAY = 10;
    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())   {             //avoid queue overflow
               queue.add(i + ": " + greeting);
               i++;
            }
            Thread.sleep((int) (Math.random() * DELAY));
         }
      }
      catch (InterruptedException exception){   }
   }
}
```

Consumer Thread

- Consumer removes greetings from the queue:

```
public class Consumer implements Runnable  {
   private BoundedQueue<String> queue; //reference to the shared queue
   private int greetingCount;
   private static final int DELAY = 10;
   public Consumer(BoundedQueue<String> aQueue, int count) {
      queue = aQueue;  greetingCount = count;
   }
   public void run() {
      try {
         int i = 1;
         while (i <= greetingCount) {
            if (!queue.isEmpty()) {        //avoid queue underflow
               String greeting = (String)queue.remove();
               System.out.println(greeting);
               i++;
            }
            Thread.sleep((int)(Math.random() * DELAY));
         }
      }
      catch (InterruptedException exception){   }
   }
}
```

Thread Driver

- ThreadTester: 2 producers, 1 consumer

```
public class ThreadTester
{
   public static void main(String[] args)
   {
      BoundedQueue<String> queue = new BoundedQueue<String>(10);
      final int COUNT = 10;
      Runnable run1 = new Producer("Yin", queue, COUNT);
      Runnable run2 = new Producer("Yang", queue, COUNT);
      Runnable run3 = new Consumer(queue, 2 * COUNT);

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

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

Expected output

- Successful test run: 10 Yins and 10 Yangs interleaved:

```
tag181906:queue1 jillseaman$ java -cp bin ThreadTester
1: Yin
1: Yang
2: Yang
3: Yang
4: Yang
2: Yin
5: Yang
3: Yin
6: Yang
4: Yin
5: Yin
7: Yang
6: Yin
7: Yin
8: Yang
9: Yang
10: Yang
8: Yin
9: Yin
10: Yin
```

Unexpected actual output

- Failed test run: Didn't complete and **9: Yin** is missing

```
tag181906:queue1 jillseaman$ java -cp bin ThreadTester
1: Yin
1: Yang
2: Yin
2: Yang
3: Yin
3: Yang
4: Yang
4: Yin
5: Yin
5: Yang
6: Yin
7: Yin
6: Yang
8: Yin
7: Yang
8: Yang
10: Yin
9: Yang
10: Yang
```

Expected/Desired interleaving of threads

- Desired scenario:
  Thread1 adds, then Thread2 adds

```
Thread1 executes queue.add(): rear=(rear+1)%elements.length;
Thread1 executes queue.add(): elements[rear] = "1: Yin";
Thread2 executes queue.add(): rear=(rear+1)%elements.length;
Thread2 executes queue.add(): elements[rear] = "1: Yang";
```

rear=-1   rear=0

0	1: Yin
1	<garbage>
2	
3	
4	

rear=1

0	1: Yin
1	1:Yang
2	
3	
4	

Undesired interleaving of threads

- Possible problem scenario:
  Update to rear and assignment to array are interleaved:

```
Thread1 executes queue.add(): rear=(rear+1)%elements.length;
Thread2 executes queue.add(): rear=(rear+1)%elements.length;
Thread1 executes queue.add(): elements[rear] = "1: Yin";
Thread2 executes queue.add(): elements[rear] = "1: Yang";
```

rear=-1   rear=0   rear=1

0	<garbage>
1	1: Yin
2	
3	
4	

0	<garbage>
1	1:Yang
2	
3	
4	

- The first queue.remove will get garbage and 1: Yin is lost

9.2.2 Race Conditions

- The code from the book is on the class website as queue1.zip if you want to play with it (turn on debugging: queue.setDebug(true);)
- The undesired interleaving on the previous slide is an example of a *race condition*.
- "A race condition occurs if the effect of multiple threads on shared data depends on the order in which the threads are scheduled."
- The program behavior is now non-deterministic (different results each time it is run) and this problem is difficult to detect and fix.
- To fix the race condition, you must ensure only one thread manipulates the shared data at any given moment.

9.2. Locks

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

Reentrant Locks

- Use this pattern to ensure a block of code is executed by only one thread at a time:

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

- The finally clause ensures that the lock is unlocked even when an exception is thrown in the protected code.

The BoundedQueue with locks

```
public class BoundedQueue<E> {
   private Object[] elements;
   private int front, rear, numItems;
   private ReentrantLock queueLock = new ReentrantLock();

   //only showing remove and add, other methods unchanged
   public E remove() {
        queueLock.lock();
        try {
            E r = (E) elements[front];
            front = (front+1)%elements.length;
            numItems--;
            return r;
        } finally {  queueLock.unlock();  }
    }
    public void add(E newValue) {
        queueLock.lock();
        try {
            rear = (rear+1)%elements.length;
            elements[rear] = newValue;
            numItems++;
        } finally { queueLock.unlock();  }
    }
}
```

Lock prevents undesired interleaving:

- Possible problem scenario resolved:

```
Thread1 calls add and acquires lock (queueLock.lock())
Thread1 executes rear=(rear+1)%elements.length;

Thread2 calls add on the queue but cannot acquire the lock and is blocked

Thread1 executes elements[rear] = “1: Yin”;
Thread1 executes numItems++;
Thread1 executes queueLock.unlock();

Thread2 is unblocked.
Thread2 acquires lock (queueLock.lock())
Thread2 executes rear=(rear+1)%elements.length;
Thread2 executes elements[rear] = “1: Yang”;
Thread2 executes numItems++;
Thread2 executes queueLock.unlock();
```

- This is now the same as the successful scenario

9.2.4 Avoiding Deadlocks
Another problem scenario:

- This code is in the producer:

```
if (!queue.isFull()) {
   queue.add(i + ": " + greeting);
   i++;
}
```

```
Thread1 executes queue.isFull() and it’s false
Thread2 executes queue.isFull() and it’s still false
Thread2 executes queue.add(…) (and now the queue is full)
Thread1 executes queue.add(…) (queue overflow!!)
```

- The isFull test should be moved inside of add:
  Don’t do the add if it’s full (maybe throw an exception?)

Perform isFull test inside add:

- New version of add:

```
public void add(E newValue) {
   queueLock.lock();
   try {
      while (numItems == elements.length)
         //wait for more space
      rear = (rear+1)%elements.length;
      elements[rear] = newValue;
      numItems++;
   } finally { queueLock.unlock();  }
}
```

- How to wait? Sleep for a bit?
- Problem: no one else can perform remove because this thread holds the lock.

Deadlocks

- A deadlock occurs if no thread can proceed because each thread is waiting for another to do some work first.
- The thread in the add method holds the lock and is waiting for someone to do a remove.
- Another thread is waiting in remove method for someone to release the lock

- One way to resolve this in Java is using a Condition.

Avoiding Deadlocks

- Each lock can have one or more associated `Condition` objects.

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

- Calling `await()` on a condition object temporarily releases the associated lock and blocks the current thread.
- The current thread is added to a set of threads that are waiting for the condition.
- These threads are unblocked and made runnable when another thread executes the `signalAll()` method on the same condition object. They all then contend for the lock, the one that gets it will come out of the await() call, and continue executing.

The BoundedQueue with a lock and conditions

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

   public E remove() throws InterruptedException {
        queueLock.lock();
        try {
            while (numItems == 0)
                valueAvailableCondition.await();
            E r = (E) elements[front];
            front = (front+1)%elements.length;
            numItems--;
            spaceAvailableCondition.signalAll();
            return r;
        } finally {  queueLock.unlock();  }
    }
    public void add(E newValue) throws InterruptedException{
        queueLock.lock();
        try {
            while (numItems == elements.length)
                spaceAvailableCondition.await();
            rear = (rear+1)%elements.length;
            elements[rear] = newValue;
            numItems++;
            valueAvailableCondition.signalAll();
        } finally { queueLock.unlock();  }
    }
}
```

The BoundedQueue with a lock and conditions Other changes

- isFull and isEmpty methods are removed, but could be added back if needed.
- Calls to isFull/isEmpty are removed from Consumer and Producer (these are no longer needed).
- Calling `await()` requires throwing or catching InterruptedException.
  - add and remove call await(), have `throws InterruptedException`
  - add and remove are called from Producer run() and Consumer run() methods, which already catch InterruptedException.

Conditions prevent deadlock:

- Now this code is in the producer, and isFull check is in add:

```
while (i <= greetingCount) {
   queue.add(i + ": " + greeting); //moved isFull test to add
   i++;
}
```

```
Thread1 executes queue.add(), acquires lock and queue is full
Thread1 releases lock, blocks itself, waits for spaceAvailableCondition
Thread2 executes queue.remove(), acquires lock, and removes an element
Thread2 executes numItems–;
Thread2 executes spaceAvailableCondition.signalAll(), releases lock
Thread1 is now runnable, acquires lock, and queue is not full
Thread1 executes queue.add(), releases lock
```

9.2.5 Object Locks

- Lock and Condition were added in Java 5.0
- Before that you had to use Object locks:
  - Every object has an object lock
  - Calling a method tagged as synchronized acquires lock of the object the method is called on.
  - Leaving the synchronized method releases lock.
  - Easier than explicit Lock objects, but not as flexible nor as safe

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

Object Locks: conditions

- 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, all contend for the lock, the one that gets the lock comes out of the wait method and proceeds.

```
public synchronized void add(E newValue) throws InterruptedException {
   while (numItems == elements.length)
      wait();
   rear = (rear+1)%elements.length;
   elements[rear] = newValue;
   numItems++;
   notifyAll();
}
```

The BoundedQueue with Object lock and condition

```
public class BoundedQueue<E> {
    private Object[] elements;
    private int front, rear, numItems;

    public BoundedQueue(int capacity)    {
        elements = new Object[capacity];
        front = numItems = 0; rear = -1;
    }
    public synchronized E remove() throws InterruptedException {
        while (numItems == 0)
            wait();
        E r = (E) elements[front];
        front = (front+1)%elements.length;
        numItems--;
        notifyAll();
        return r;
    }
    public synchronized void add(E newValue) throws InterruptedException {
        while (numItems == elements.length)
            wait();
        rear = (rear+1)%elements.length;
        elements[rear] = newValue;
        numItems++;
        notifyAll();
    }
}
```

Object Locks and Conditions

- wait and notifyAll belong to the Object class, and not the Thread class.
- BoundedQueue previously used two conditions to monitor whether the queue was full or empty.
- With Object locks, there is only one condition. When the queue changes (after an add or remove), all waiting threads are awakened.
- The newly awoken thread that gets the lock will need to check its condition to see if it can proceed, or go back to waiting.

Are locks/synchronization required for accessors?

- suppose we add this method to the BoundedQueue class:

```
public int getSize() { return numItems; }   // Not threadsafe
```

- If one thread updates the numItems field, the change may not be visible in another thread. (i.e. if two threads are executed by different processors).
- Each thread may cache its current value of numItems, and copy it to main memory later. Acquiring/releasing locks causes the value to be copied to main memory.
- Perhaps, one thread keeps adding elements to the queue, but the other always sees the size as 0.

See GUI Graphics lecture next

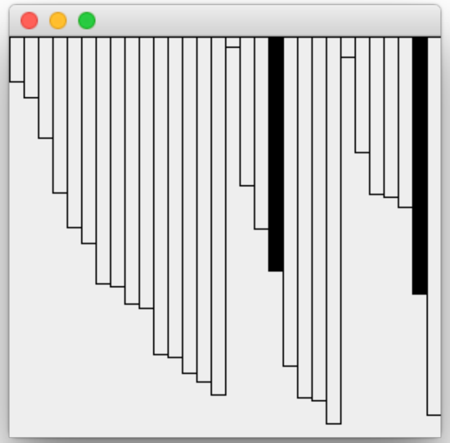
- Place holder for the GUI Graphics lecture

9.3 Animations

- You can use the Swing Timer class for simple animations (no thread programming).
- More advanced animations often require threads.
- We will use animation to visualize the steps of the merge sort algorithm.
- Algorithm will run in a separate thread from the animation.
- Algorithm periodically updates a drawing of its current state, and then sleeps, then runs until the next point of interest.

Algorithm Animation

- This is one step of the output:



- each bar is a number in the array
- bar size is proportional to the value of the number
- black bars are the two values currently being compared.

MergeSorter class, just sorts, given a comparator

```
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;
      mergeSort(a, from, mid, comp);   //sort the first half
      mergeSort(a, mid + 1, to, comp); //sort the second half
      merge(a, from, mid, to, comp);   //merge the results
   }

   //continued…
```

MergeSorter class

```
  /** 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
      Object[] b = new Object[n]; // Merge into a temporary array b
      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++;
      }  // remainder of code finishes merge, copies from b back to a
  }
```

Sorter Thread

- The Sorter is Runnable.
- It contains a reference to the GUI Component that draws the array.
- run() makes an anonymous comparator and passes it with an array of Double, to the MergeSorter.
- The comparator updates the drawing data in the GUI component, then pauses (so array is re-painted) and then returns the result of the comparison.

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

Sorter

```
public class Sorter implements Runnable {
   public Sorter(Double[] values, ArrayComponent panel) {
      this.values = values;
      this.panel = panel;
   }
   public void run(){
      Comparator<Double> comp = new
         Comparator<Double>() {
            public int compare(Double d1, Double d2) {
               panel.setValues(values, d1, d2);
               try {
                  Thread.sleep(DELAY);
               }
               catch (InterruptedException exception)  {
                  Thread.currentThread().interrupt();
               }
               return (d1).compareTo(d2);
            }
         };
      MergeSorter.sort(values, comp);
      panel.setValues(values, null, null);
   }
   private Double[] values;
   private ArrayComponent panel;
   private static final int DELAY = 100;
}
```

ArrayComponent

- A GUI Component that draws an array as a bar graph, and marks two of the elements.

```
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);
      }
   }
  public synchronized void setValues(Double[] values,
      Double marked1, Double marked2) {
      this.values = (Double[]) values.clone();
      this.marked1 = marked1;
      this.marked2 = marked2;
      repaint();
   }
}
```

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

AnimationTester

- Sets up the window/ArrayComponent panel and the array of random values, passes these to the Sorter and starts it.

```
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();
   }
```

Pausing the animation

- let’s add two buttons labeled “Run” and “Step”.
- The “Step” button runs the algorithm until the next step and then pauses the algorithm.
- Need to coordinate UI thread, animation thread.
- We’ll use a shared object, the animation thread asks for permission to proceed, the step button grants it when clicked.
- We’ll use the java.util.concurrent.LinkedBlockingQueue
- Button click adds string "Run" or "Step" to queue
- Animation thread calls take method on the queue, which blocks if no string available.

LinkedBlockingQueue methods

- boolean add(E e)
  Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
- E take()
  Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.
- E peek()
  Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
- It is implemented as a linked list.
- The queue likely never grows above size 1 in this demo.
- This queue is very similar to our “blocking” bounded queue.

Updated compare method

- Waits until a command string is available
- If Run, pauses like the first version, the puts “Run” back in the queue (if “Step” is not there):

```
Comparator<Double> comp = new
   Comparator<Double>()
   {
      public int compare(Double d1, Double d2)
      {
         . . .
         String command = queue.take();
         if (command.equals("Run"))
         {
            Thread.sleep(DELAY);
            if (!"Step".equals(queue.peek()))
               queue.add("Run");
         }
         . . .
         }
   };
```

Sorter

```
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;
}
```

AnimationTester

- Added the buttons and the blocking queue:

```
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");
```

AnimationTester continued

- Action listeners for the buttons:

```
      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;
```