1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Day 10: Advanced concurrency (CS)

Tuesday, 14 November

JC2002 Java Programming

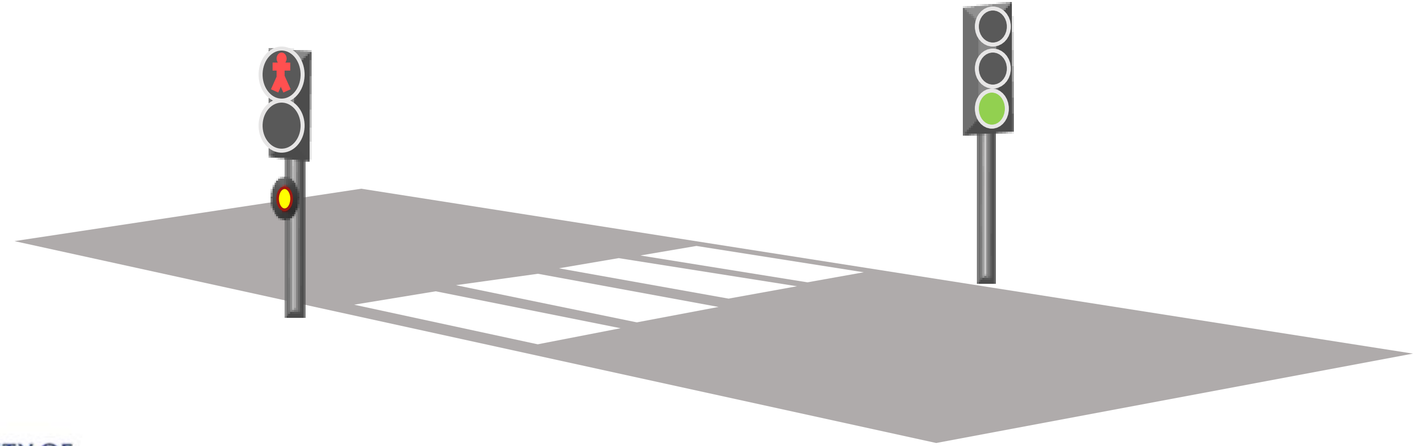
Day 10, Session 1: Timed events and synchronisation

References and learning objectives

- Today's sessions are mostly based on:
  - Deitel, H., ***Java How to Program, Early Objects***, Chapter 23, 2018
  - https://docs.oracle.com/javase/tutorial/uiswing
- After today's session, you should be able to:
  - Implement timed events using multithreading
  - Implement the basic techniques to avoid thread interference and deadlocks in your multithreading applications

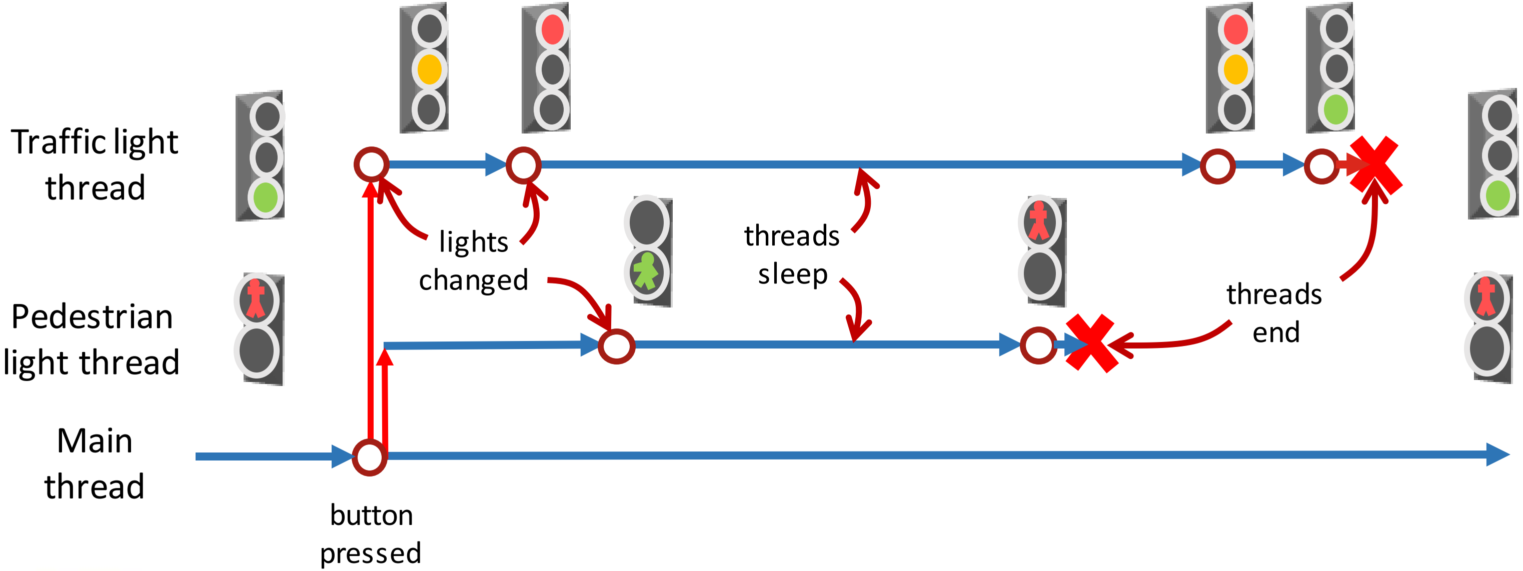
Multithreading for timed events

- Threads can be useful for implementing timed events
- Example: traffic lights, where pedestrian pushes a button to request green light and to initiate the light cycle



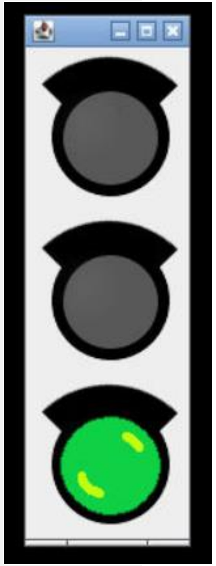
Example: traffic lights

- We could implement pedestrian and traffic light cycles as two threads



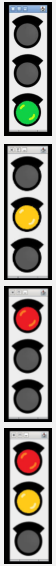
Example: define lights for cars (1)

```
class TrafficLights extends JPanel {

    JLabel topLight, middleLight, bottomLight;
    ImageIcon off, red, yellow, green;

    TrafficLights() {
        off = new ImageIcon("traffic_off.png");
        red = new ImageIcon("traffic_red.png");
        yellow = new ImageIcon("traffic_yellow.png");
        green = new ImageIcon("traffic_green.png");
        topLight = new JLabel();
        topLight.setIcon(off);
        middleLight = new JLabel();
        middleLight.setIcon(off);
        bottomLight = new JLabel();
        bottomLight.setIcon(green);
        GridLayout gridlayout = new GridLayout(3,1);
        setLayout(gridlayout);
        add(topLight);
        add(middleLight);
        add(bottomLight);
    }
```



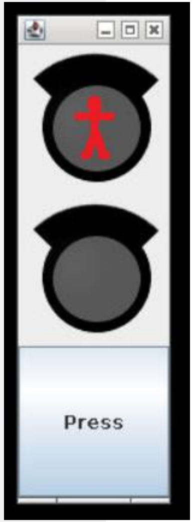
Example: define lights for cars (2)

```java
    public void setGreen() {
        topLight.setIcon(off);
        middleLight.setIcon(off);
        bottomLight.setIcon(green);
    }
    public void setYellow() {
        topLight.setIcon(off);
        middleLight.setIcon(yellow);
        bottomLight.setIcon(off);
    }
    public void setRed() {
        topLight.setIcon(red);
        middleLight.setIcon(off);
        bottomLight.setIcon(off);
    }
    public void setRedAndYellow() {
        topLight.setIcon(red);
        middleLight.setIcon(yellow);
        bottomLight.setIcon(off);
    }
}
```



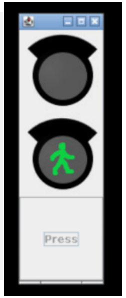
Example: define lights for pedestrians (1)

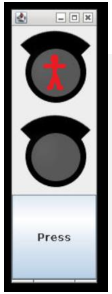
```java
class PedestrianLights extends JPanel
                       implements ActionListener {
    JLabel topLight, bottomLight;
    JButton button;
    ImageIcon off, wait, go;
    String status;
    TrafficLights trafficLights;
    PedestrianLights(TrafficLights tl) {
        trafficLights = tl;
        status = new String("wait");
        off = new ImageIcon("traffic_off.png");
        wait = new ImageIcon("traffic_wait.png");
        go = new ImageIcon("traffic_go.png");
        topLight = new JLabel();
        topLight.setIcon(wait);
        bottomLight = new JLabel();
        bottomLight.setIcon(off);
        button = new JButton("Press");
        GridLayout gridlayout = new GridLayout(3,1);
        setLayout(gridlayout);
        add(topLight);
        add(bottomLight);
        add(button);
```



Example: define lights for pedestrians (2)

```
        button.addActionListener(this);
    }
    public void setGo() {
        topLight.setIcon(off);
        bottomLight.setIcon(go);
        status = new String("go");
    }
```

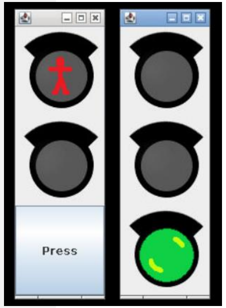




```
    public void setWait() {
        topLight.setIcon(wait);
        bottomLight.setIcon(off);
        status = new String("wait");
    }
```

Example: create and show GUI

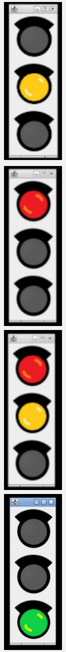
```
public class TrafficLightExample {
  private static void createAndShowGUI() {
    JComponent trafficLights = new TrafficLights();
    trafficLights.setOpaque(true);
    JComponent pedestrianLights = new PedestrianLights((TrafficLights)trafficLights);
    pedestrianLights.setOpaque(true);
    JFrame plFrame = new JFrame("Pedestrian Lights");
    plFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JFrame tlFrame = new JFrame("Traffic Lights");
    tlFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    plFrame.add(pedestrianLights);
    plFrame.pack();
    plFrame.setLocation(50,50);
    plFrame.setVisible(true);
    tlFrame.add(trafficLights);
    tlFrame.pack();
    tlFrame.setLocation(300,50);
    tlFrame.setVisible(true);
  }
```



```
  public static void main(String[] args) {
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        createAndShowGUI();
      }
    });
  }
}
```

Example: run cycle for cars

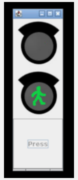
```java
class PedestrianLights extends JPanel
                       implements ActionListener {
...
    public void startCycle() {
        Thread trafficThread = new Thread(new Runnable() {
            @Override
            public void run()
            {
                try {
                    trafficLights.setYellow();
                    Thread.sleep(2000);
                    trafficLights.setRed();
                    Thread.sleep(5000);
                    trafficLights.setRedAndYellow();
                    Thread.sleep(1000);
                    trafficLights.setGreen();
                    button.setEnabled(true);
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
```

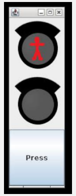


Example: run cycle for pedestrians

```
   Thread pedestrianThread = new Thread(new Runnable() {
     @Override
     public void run()
     {
       try {
         button.setEnabled(false);
         Thread.sleep(3000);
         setGo();
         Thread.sleep(3000);
         setWait();
      }
      catch (InterruptedException e) {
         e.printStackTrace();
      }
    }
  });
  trafficThread.start();
  pedestrianThread.start();
}
```

Button is disabled to avoid new cycle to start before the old has stopped

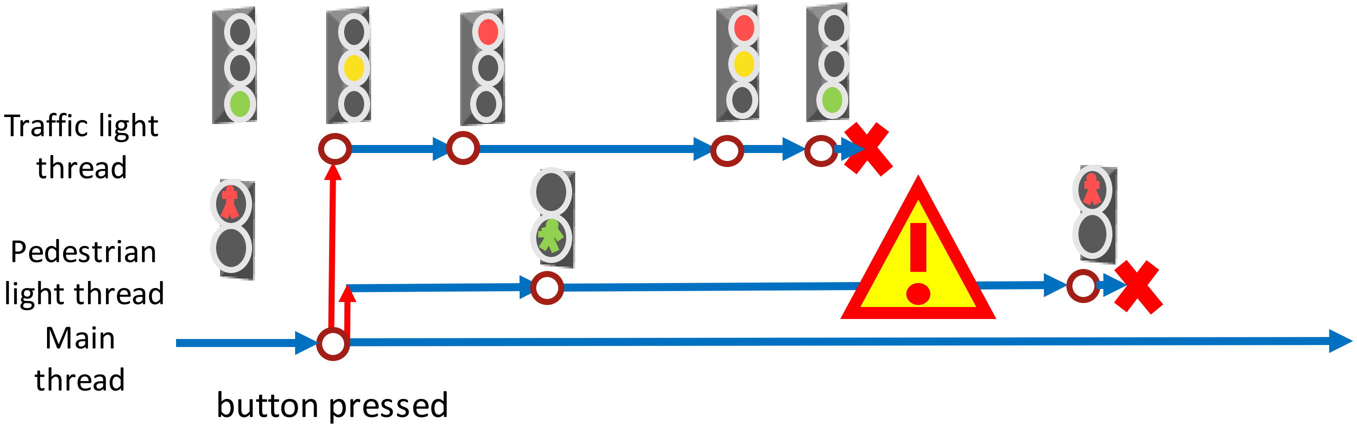




```
  public void actionPerformed(ActionEvent e) {
    startCycle();
  }
}
```

Synchronisation in traffic light example

- When traffic light threads run independently and timings are wrong, there is a risk that the light for cars turns green to early
- We should make sure the lights are not in conflict with each other

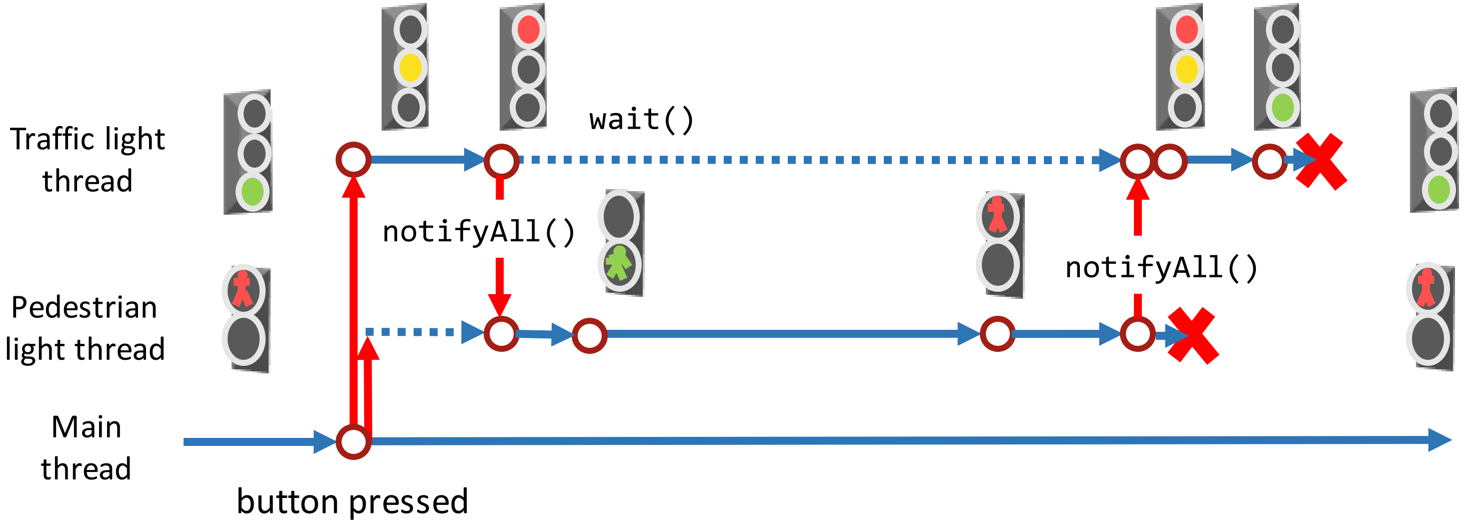


Using wait() and notify()

- Access to any object of class or subclass of `Object` (basically any object) from a thread can be controlled by **`Object.wait()`** and **`Object.notifyAll()`**
  - Method `wait()` suspends the current thread until another thread issues a notification for the same object so that it can resume
  - Method `notifyAll()` sends a notification to all the other threads so that they can resume
  - Method `notify()` is similar to `notifyAll()`, but only notifies one thread chosen randomly

Example: synchronised traffic lights

- We could synchronize threads so that traffic lights for cars always waits until pedestrian lights have been turned red again



Traffic lights with synchronisation (1)

```
...
    public void startCycle() {
        TrafficLightThread tlCycle = new TrafficLightThread(this,tl);
        PedestrianLightThread plCycle = new PedestrianLightThread(this);
        tlCycle.start();
        plCycle.start();
    }

    public void actionPerformed(ActionEvent e) {
        disableButton();
        startCycle();
    }
}
```

Note that the threads are synchronized, and therefore `plCycle` starts in waiting state

We define custom subclasses of `Thread` for traffic lights and pedestrian lights

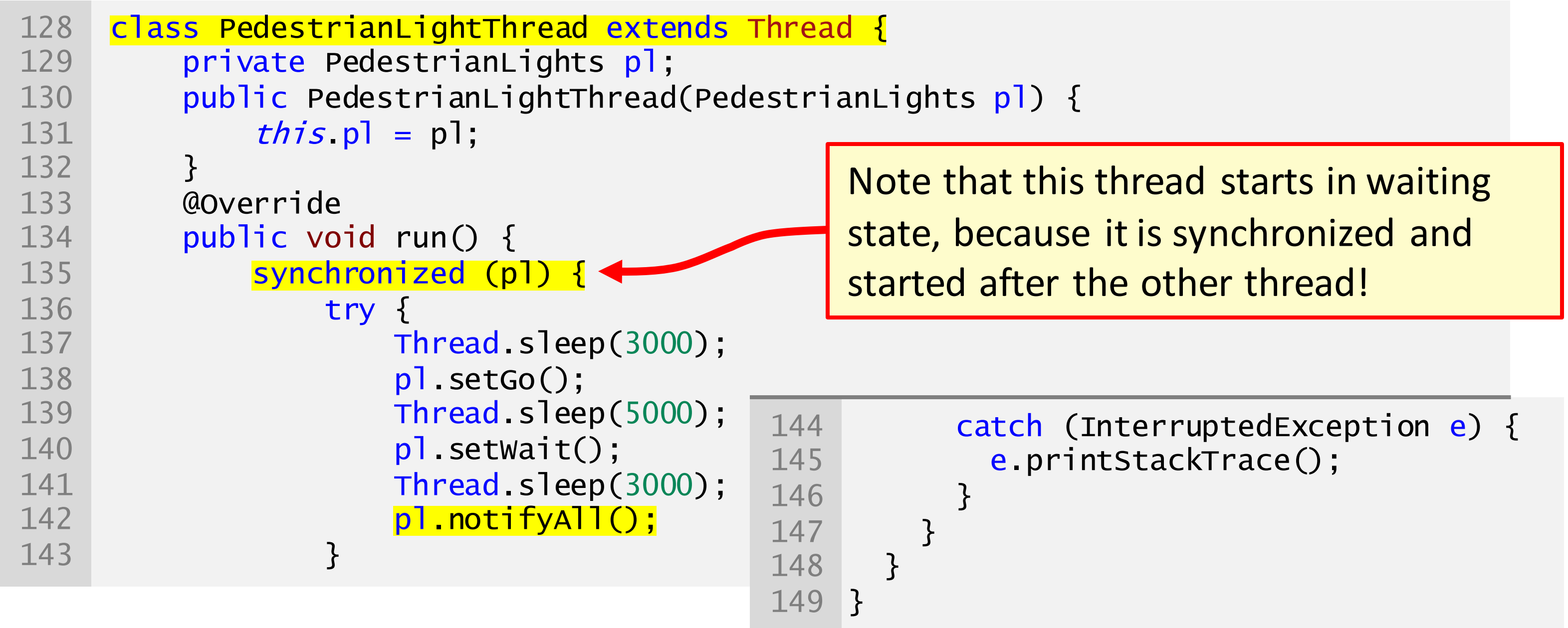
Traffic lights with synchronisation (2)

```
class TrafficLightThread extends Thread {
  private PedestrianLights pl;
  private TrafficLights tl;
  public TrafficLightThread(PedestrianLights pl, TrafficLights tl) {
    this.pl = pl;   this.tl = tl;
  }
  @Override
  public void run() {
    synchronized (pl) {
      try {
        tl.setYellow();
        Thread.sleep(2000);
        tl.setRed();
        pl.notifyAll();
        pl.wait();
        tl.setRedAndYellow();
        Thread.sleep(1000);
        tl.setGreen();
        pl.enableButton();
      }
```

Methods `notifyAll()` and `wait()` must be inside `synchronized` block to avoid exception

```
   catch (InterruptedException e) {
    e.printStackTrace();
   }
  }
 }
}
```

Traffic lights with synchronisation (3)



```java
class PedestrianLightThread extends Thread {
    private PedestrianLights pl;
    public PedestrianLightThread(PedestrianLights pl) {
        this.pl = pl;
    }
    @Override
    public void run() {
        synchronized (pl) {
            try {
                Thread.sleep(3000);
                pl.setGo();
                Thread.sleep(5000);
                pl.setWait();
                Thread.sleep(3000);
                pl.notifyAll();
            }
```

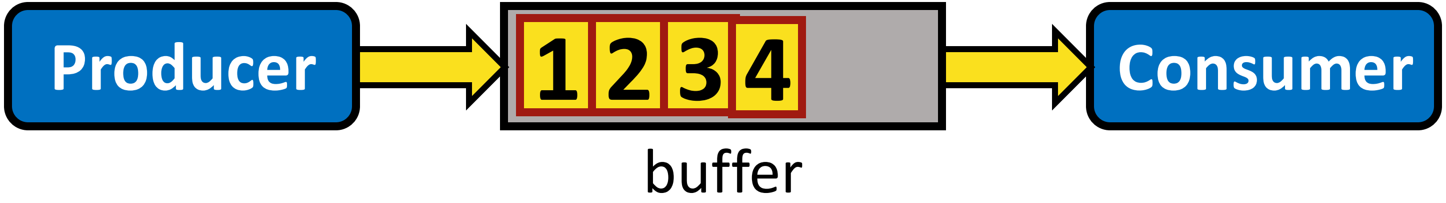
```java
      catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Note that this thread starts in waiting state, because it is synchronized and started after the other thread!

UNIVERSITY OF ABERDEEN

Producer-consumer model

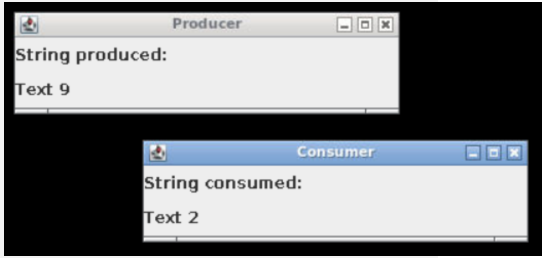
- A common example of the importance of synchronization is *producer-consumer problem*
  - *Producer* thread produces data and adds it in buffer
  - *Consumer* thread consumes data and removes it from buffer
  - Without synchronization, threads may try to add and remove data at the same time, leading to problems



Producer-consumer example: GUI

```
public class ProducerConsumerExample {
  private static void createAndShowGUI() {
    ArrayList<String> buffer = new ArrayList<>();
    JPanel producerPanel = new JPanel();
    producerPanel.setOpaque(true);
    JLabel prodInfo = new JLabel("String produced:");
    JLabel prodLabel = new JLabel();
...
    JPanel consumerPanel = new JPanel();
    consumerPanel.setOpaque(true);
    JLabel consInfo = new JLabel("String consumed:");
    JLabel consLabel = new JLabel();
...
    Producer prodThread = new Producer(buffer, prodLabel);
    prodThread.start();
    Consumer consThread = new Consumer(buffer, consLabel);
    consThread.start();
  }
...
```

Create buffer



String produced:

Text 9

String consumed:

Text 2

Create and start producer and consumer threads

Producer thread

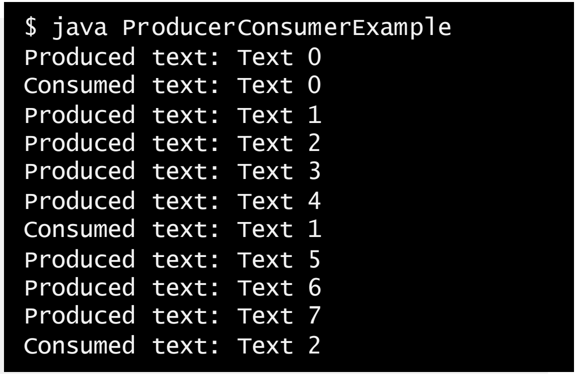
```java
class Producer extends Thread {
  private ArrayList<String> buffer;
  private JLabel label;
  public Producer(ArrayList<String> buffer, JLabel label) {
    this.buffer = buffer;
    this.label = label;
  }
  @Override
  public void run() {
    for(int i = 0; i < 100; i++) {
      synchronized (buffer) {
        String text = new String("Text “ + i);
        System.out.println("Produced text: " + text);
        buffer.add(text);
        label.setText(text);
      }
      try {
        Random r = new Random();
        Thread.sleep(r.nextInt(800));
      }
```

Simulate random intervals between produced text items

```java
      catch(InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Consumer thread

```java
class Consumer extends Thread {
  private ArrayList<String> buffer;
  private JLabel label;
  public Consumer(ArrayList<String> buffer, JLabel label) {
    this.buffer = buffer;
    this.label = label;
  }
  @Override
  public void run() {
    while(true) {
      synchronized (buffer) {
        if(!buffer.isEmpty()) {
          String text = buffer.remove(0);
          System.out.println("Consumed text: " + text);
          label.setText(text);
        }
      }
      try {
        Thread.sleep(1000);
      }
      catch(InterruptedException e) {
          e.printStackTrace();
        }
      }
    }
}
```



```
$ java ProducerConsumerExample
Produced text: Text 0
Consumed text: Text 0
Produced text: Text 1
Produced text: Text 2
Produced text: Text 3
Produced text: Text 4
Consumed text: Text 1
Produced text: Text 5
Produced text: Text 6
Produced text: Text 7
Consumed text: Text 2
```

Simulate one second processing time for consumed items

Questions, comments?

1495 UNIVERSITY OF ABERDEEN

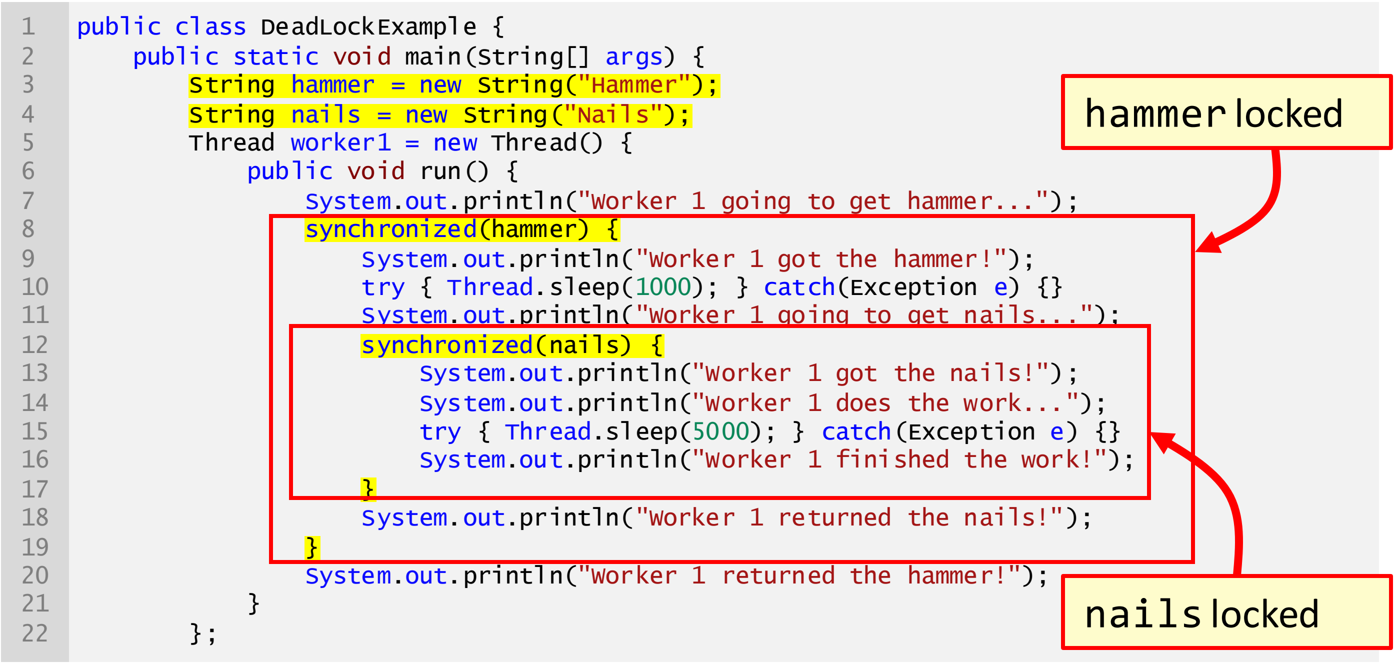
JC2002 Java Programming

Day 10, Session 2: Liveness and high-level concurrency

Liveness

- The ability of an application to execute in a timely manner is known as its *liveness*
- Liveness can be compromised by *deadlocks*, *starvation* and *livelocks*
    - *Deadlocks* happen when two threads are blocking each other
    - *Starvation* happens when low priority thread cannot access shared resources, because they are reserved by high priority "greedy" threads
    - *Livelock* is similar with deadlock, but the threads are not blocked indefinitely, they are just too slow to respond to each other

Deadlock example: worker 1

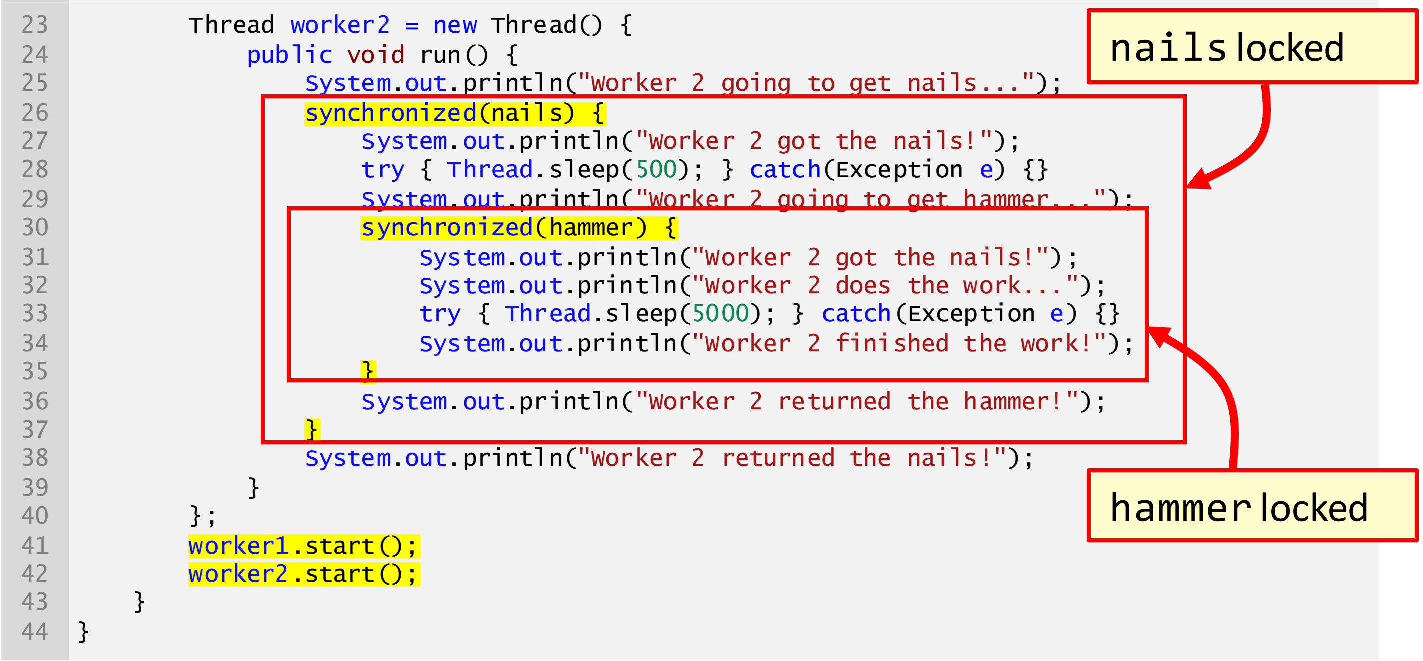


```java
public class DeadLockExample {
    public static void main(String[] args) {
        String hammer = new String("Hammer");
        String nails = new String("Nails");
        Thread worker1 = new Thread() {
            public void run() {
                System.out.println("Worker 1 going to get hammer...");
                synchronized(hammer) {
                    System.out.println("Worker 1 got the hammer!");
                    try { Thread.sleep(1000); } catch(Exception e) {}
                    System.out.println("Worker 1 going to get nails...");
                    synchronized(nails) {
                        System.out.println("Worker 1 got the nails!");
                        System.out.println("Worker 1 does the work...");
                        try { Thread.sleep(5000); } catch(Exception e) {}
                        System.out.println("Worker 1 finished the work!");
                    }
                    System.out.println("Worker 1 returned the nails!");
                }
                System.out.println("Worker 1 returned the hammer!");
            }
        };
```

hammer locked

nails locked

Deadlock example: worker 2

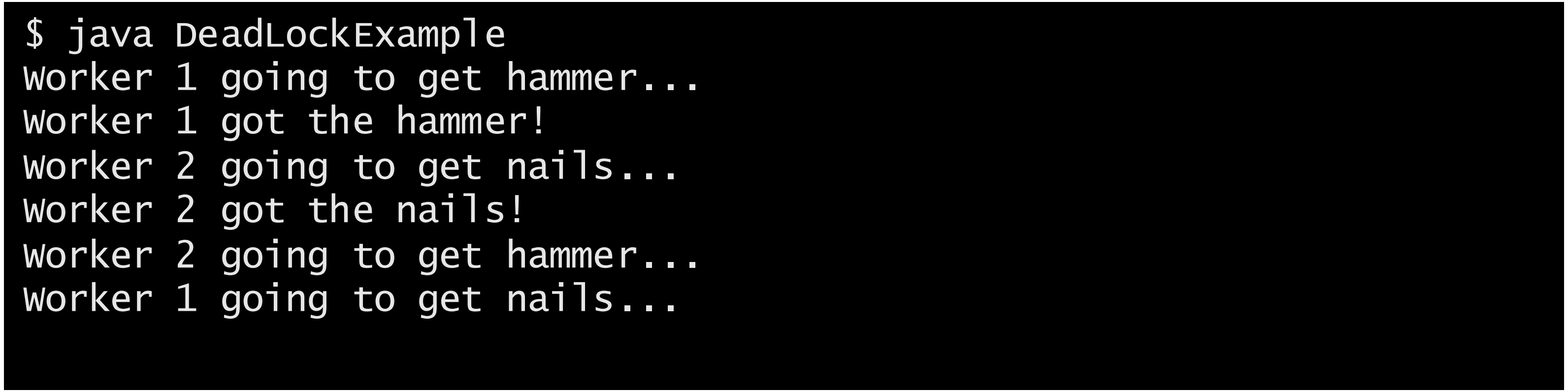


```java
    Thread worker2 = new Thread() {
      public void run() {
        System.out.println("Worker 2 going to get nails...");
        synchronized(nails) {
          System.out.println("Worker 2 got the nails!");
          try { Thread.sleep(500); } catch(Exception e) {}
          System.out.println("Worker 2 going to get hammer...");
          synchronized(hammer) {
            System.out.println("Worker 2 got the nails!");
            System.out.println("Worker 2 does the work...");
            try { Thread.sleep(5000); } catch(Exception e) {}
            System.out.println("Worker 2 finished the work!");
          }
          System.out.println("Worker 2 returned the hammer!");
        }
        System.out.println("Worker 2 returned the nails!");
      }
    };
    worker1.start();
    worker2.start();
  }
}
```

nails locked

hammer locked

Deadlock example: output



```
$ java DeadLockExample
Worker 1 going to get hammer...
Worker 1 got the hammer!
Worker 2 going to get nails...
Worker 2 got the nails!
Worker 2 going to get hammer...
Worker 1 going to get nails...
```

- The program is deadlocked, because worker 1 has the hammer and worker 2 has the nails, and neither worker can proceed…

Avoiding deadlocks

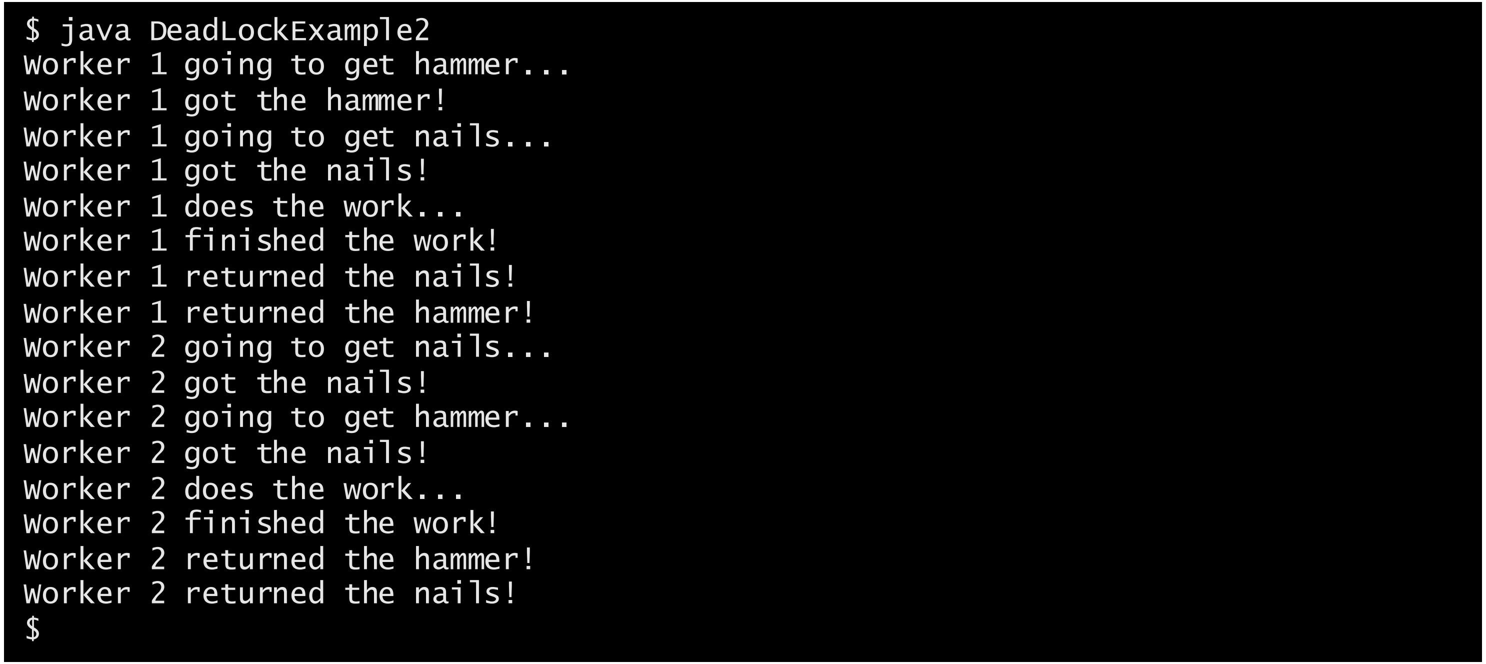
- Avoid nested locks (`synchronization` blocks inside each other)
- Avoid unnecessary locks: only lock objects you really need
- Instead of locking objects via synchronization, use *immutable objects* whenever possible
  - An object is *immutable* if its state cannot be changed after constructed
  - Do not provide setter methods, define all fields final and private
- Invoke **`t.join()`** method of Thread `t` to make other threads to start after `t` has finished
  - Timed version `join(m)` waits at most *m* milliseconds for thread to die

Example of avoiding deadlocks with join()

```java
public class DeadLockExample2 {
    public static void main(String[] args) {
        String hammer = new String("Hammer");
        String nails = new String("Nails");
...
...
        try {
            worker1.start();
            worker1.join();
            worker2.start();
        }
        catch(InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Same as previous example

Avoiding deadlocks with join(): output



```
$ java DeadLockExample2
Worker 1 going to get hammer...
Worker 1 got the hammer!
Worker 1 going to get nails...
Worker 1 got the nails!
Worker 1 does the work...
Worker 1 finished the work!
Worker 1 returned the nails!
Worker 1 returned the hammer!
Worker 2 going to get nails...
Worker 2 got the nails!
Worker 2 going to get hammer...
Worker 2 got the nails!
Worker 2 does the work...
Worker 2 finished the work!
Worker 2 returned the hammer!
Worker 2 returned the nails!
$
```

High level concurrency

- Concurrency explained so far on this course is based on low-level API useful for basic tasks, but not suitable for more advanced tasks
- Package `java.util.concurrent` offers more advanced features:
  - *Lock* objects for more sophisticated synchronization features
  - *Executors* defining high level API for launching and managing threads
  - *Concurrent collections* for managing and synchronizing large collections of data
  - *Atomic variables* for atomic operations without synchronization

Lock objects

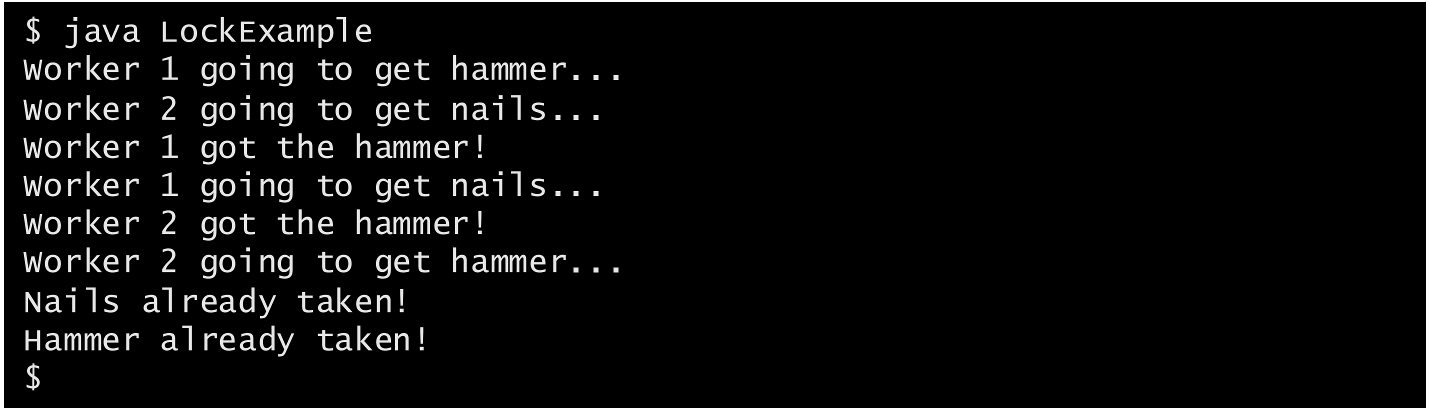
- The main advantage of the lock objects is their ability to back out of an attempt to acquire a lock
- Method **`tryLock()`** can be used to try to lock a lock object, it returns false if locking is not possible (someone acquired lock already)
- It is also possible to use timed version of **`tryLock(m)`**, that waits for the given timeout `m` (in milliseconds) before giving up
- And other advanced features (out of the scope of this course)

Avoiding deadlocks with Lock objects

```java
import java.util.concurrent.locks.ReentrantLock;
public class LockExample {
    public static void main(String[] args) {
        ReentrantLock hammerLock = new ReentrantLock();
        ReentrantLock nailLock = new ReentrantLock();
        Thread worker1 = new Thread() {
            public void run() {
                System.out.println("Worker 1 going to get hammer...");
                if(!hammerLock.tryLock()) {
                    System.out.println("Hammer already taken!");
                    return;
                }
                System.out.println("Worker 1 got the hammer!");
                try { Thread.sleep(1000); } catch(Exception e) {}
...             // ... Try locking nails in the same way
                System.out.println("Worker 1 got the nails!");
                System.out.println("Worker 1 does the work...");
                try { Thread.sleep(5000); } catch(Exception e) {}
                System.out.println("Worker 1 finished the work!");
                nailLock.unlock();
                System.out.println("Worker 1 returned the nails!");
...
```

Thread `worker2` implemented in a similar fashion

Lock example: output



```
$ java LockExample
Worker 1 going to get hammer...
Worker 2 going to get nails...
Worker 1 got the hammer!
Worker 1 going to get nails...
Worker 2 got the hammer!
Worker 2 going to get hammer...
Nails already taken!
Hammer already taken!
$
```

- Items already taken are detected and deadlock avoided!

Executor interfaces

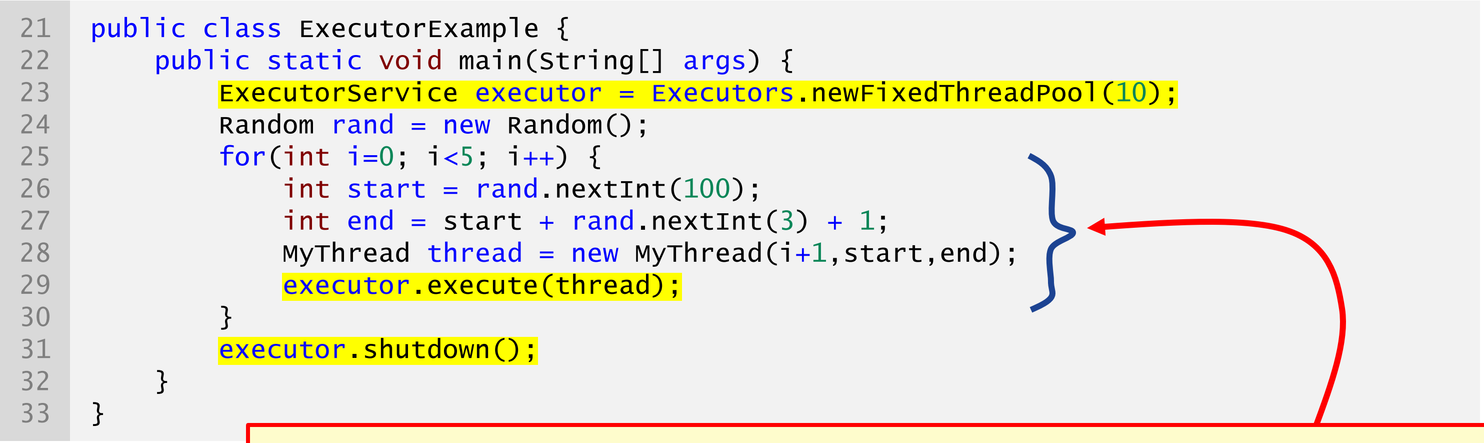
- The executor interface in `java.util.concurrent` package provides methods for launching and managing tasks (e.g., threads)
- Assuming `r` is a `Runnable` and `e` is an **`Executor`** object, it is possible to replace `(new Thread(r)).start();` with `e.execute(r);`
- Most of the executor implementations are designed to handle *thread pools* that consist of several *worker threads*
  - Advantage in large scale applications, such as web servers that need to coordinate a large number of threads in a scalable manner

Simple executor example (1)

```
import java.util.concurrent.*;
import java.util.*;
class MyThread implements Runnable {
    int threadNum, start, end;
    MyThread(int num, int start, int end) {
        this.threadNum = num; this.start = start; this.end = end;
    }
    public void run() {
        try {
            for(int i = start; i <= end; i++) {
                System.out.printf("Thread #%d, step %d\n", threadNum,i);
                Random rand = new Random();
                Thread.sleep(rand.nextInt(1000));
            }
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Implement custom thread in a normal way

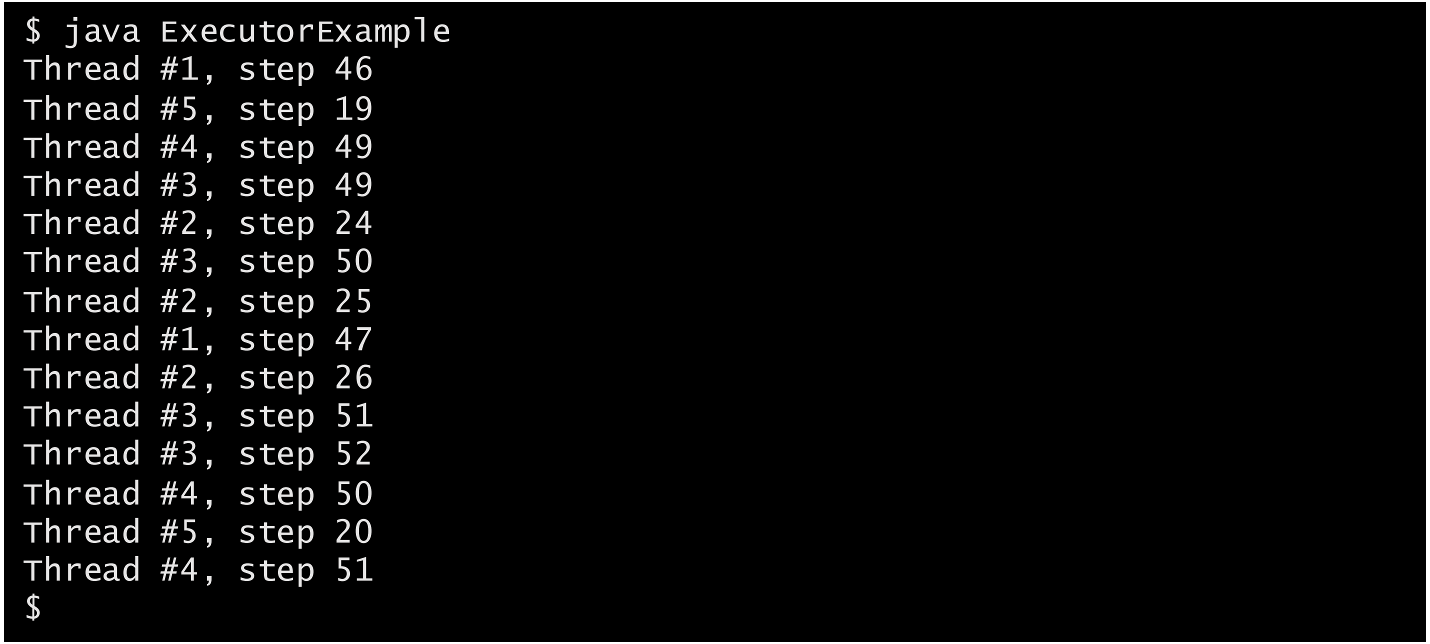
Simple executor example (2)



```java
public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(10);
        Random rand = new Random();
        for(int i=0; i<5; i++) {
            int start = rand.nextInt(100);
            int end = start + rand.nextInt(3) + 1;
            MyThread thread = new MyThread(i+1,start,end);
            executor.execute(thread);
        }
        executor.shutdown();
    }
}
```

Create five threads with different random characteristics, and execute them via the executor object

Simple executor example: output



```
$ java ExecutorExample
Thread #1, step 46
Thread #5, step 19
Thread #4, step 49
Thread #3, step 49
Thread #2, step 24
Thread #3, step 50
Thread #2, step 25
Thread #1, step 47
Thread #2, step 26
Thread #3, step 51
Thread #3, step 52
Thread #4, step 50
Thread #5, step 20
Thread #4, step 51
$
```

Final remarks on advanced concurrency

- Concurrency is a very complex topic, especially when multicore platforms are concerned
- For most programmers, the low-level API is sufficient, but for more advanced applications dealing with a lot of of data and threads, the high-level API from `java.util.concurrent` is a necessity
- For more details:
  - https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html
  - **Book:** Brian Goetz et al.: *Java Concurrency in Practice* (Addison-Wesley)

Summary

- Concurrent programming is often used to implement timed events
  - Synchronisation and methods `wait()` and `notify()` can be used to allow only one thread to access resources simultaneously and make threads to wait for another thread
- Multithreading requires programmer to consider problems with thread interference and deadlocks
  - Thread interference can be avoided using locks, but this may lead to deadlocks and other liveness problems
  - Some advice was given to avoid deadlocks

Questions, comments?

1495
UNIVERSITY OF
ABERDEEN