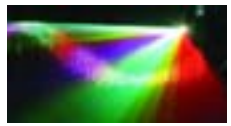
Supplement to
The Design and Implementation of Multimedia Software

Multi-Threaded Programs

Prof. David Bernstein

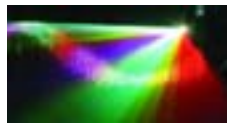
James Madison University

users.cs.jmu.edu/bernstdh

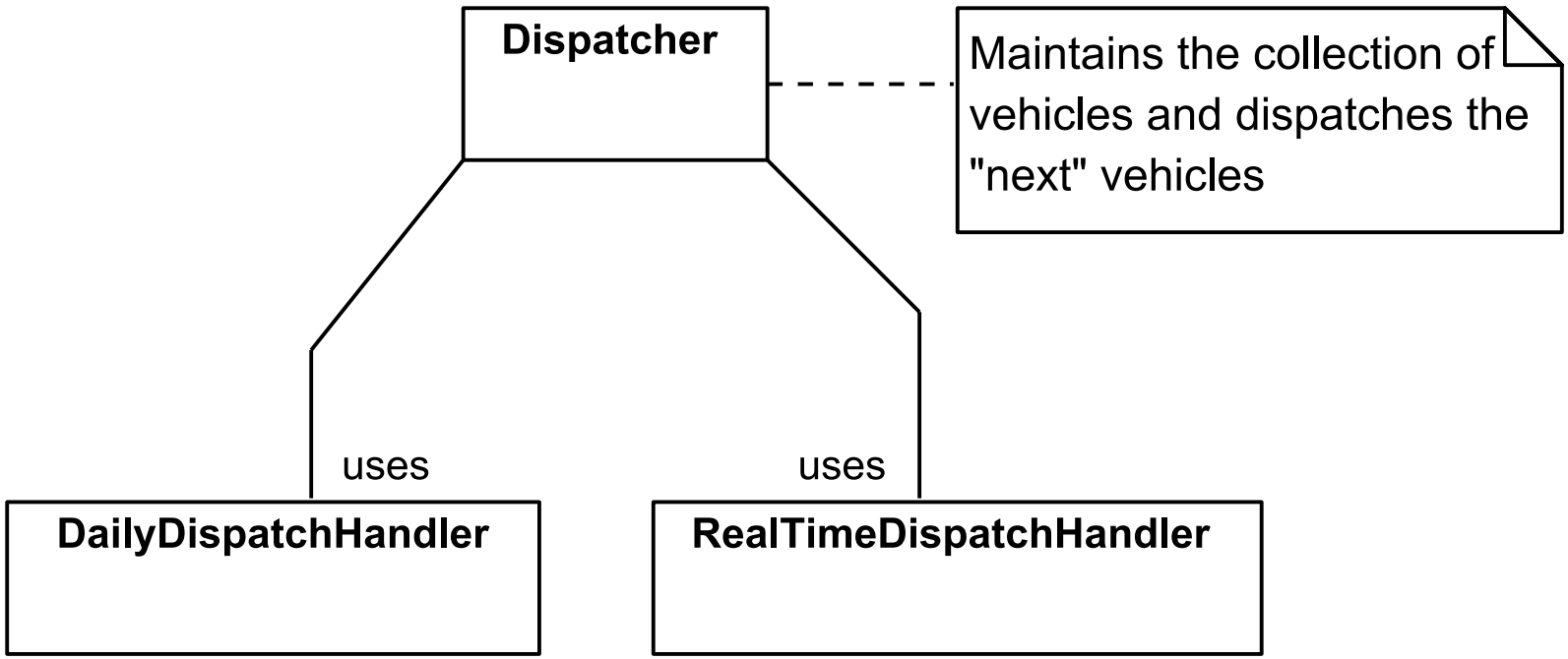


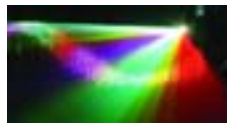
About this Chapter

- Almost everyone that uses a computer today is familiar with the concept of *multi-tasking* – "running more than one application at a time"
- This chapter is about *multi-threading* – allowing each application to "perform more than one task at a time"



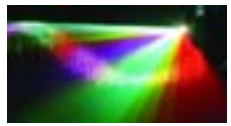
A Simple Emergency Vehicle Dispatching System





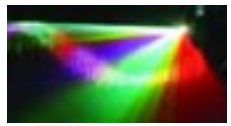
Dispatcher – Available Vehicles

```
import java.util.LinkedList;


public class Dispatcher
{
    protected int                       numberOfVehicles;
    protected LinkedList<Integer>       availableVehicles;


    public Dispatcher(int n)
    {
       int     i;

       numberOfVehicles = n;
       availableVehicles = new LinkedList<Integer>();

       for (i=0; i < n; i++)
       {
          makeVehicleAvailable(i);
       }
    }


}
```



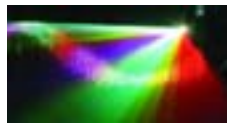
Dispatcher (cont.)

```
public boolean dispatch(String task)
{
   boolean  ok;
   int      vehicle;
   Integer  v;

   ok = false;
   v  = availableVehicles.removeFirst();

   if (v == null) ok = false;
   else
   {
      vehicle = v.intValue();
      sendMessage(vehicle, task);
      ok = true;
   }

   return ok;
}
```



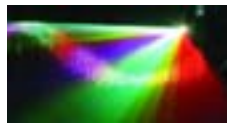
Dispatcher (cont.)

```
private void sendMessage(int vehicle, String message)
{

   // This method would normally transmit the message
   // to the vehicle.  For simplicity, it now writes it
   // to the screen instead.

   System.out.println(vehicle+"\t"+message+"\n");
   System.out.flush();
}
```



Dispatcher (cont.)

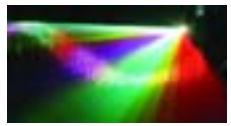
```
public void makeVehicleAvailable(int vehicle)
{
   availableVehicles.addLast(new Integer(vehicle));
}
```



DailyDispatchHandler

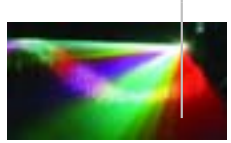
Uses a `Dispatcher` object to handle a set of "regular" dispatches.

```
import java.io.*;
import java.util.*;

public class DailyDispatchHandler
{
    private Dispatcher  dispatcher;
    private String      fileName;

    public DailyDispatchHandler(Dispatcher d, String f)
    {
       dispatcher = d;
       fileName = f;
    }


}
```



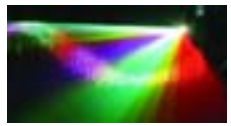
DailyDispatchHandler.processDispatches()

```
public void processDispatches()
{
   BufferedReader    in;
   int               wait;
   long              currentTime, lastTime;
   String            line, message;
   StringTokenizer   st;

   try
   {
      in = new BufferedReader(new FileReader(fileName));

      lastTime = System.currentTimeMillis();

      while ((line = in.readLine()) != null)
      {
         st = new StringTokenizer(line,"\t");
         wait = Integer.parseInt(st.nextToken());
         message = st.nextToken();

         // Wait until the appropriate time before
         // dispatching this vehicle
         //
         while (System.currentTimeMillis()-lastTime
                < wait)
         {
```



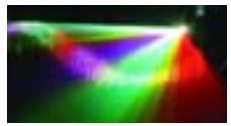
DailyDispatchHandler.processDispatches() (cont.)

```
            // Do nothing
         }

         dispatcher.dispatch(message);
         lastTime = System.currentTimeMillis();
      }
   }
   catch (IOException ioe)
   {
      System.out.println("No daily dispatches "+
                         "in: "+fileName);
   }
   catch (NoSuchElementException nsee)
   {
      System.out.println("Problem in file: "+fileName);
   }
}
```



DailyDispatchHandler.start()

```
public void start()
{
   processDispatches();
}
```



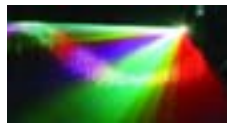
RealTimeDispatchHandler

Uses a `Dispatcher` object to handle a set of real-time dispatches

```
import java.io.*;
import java.util.*;

public class RealTimeDispatchHandler
{
    private Dispatcher  dispatcher;

    public RealTimeDispatchHandler(Dispatcher d)
    {
       dispatcher = d;
    }

    public void processDispatches()
    {
       BufferedReader    in;
       String            message;

       try
       {
          in = new BufferedReader(
                   new InputStreamReader(System.in));

          while ((message = in.readLine()) != null)
          {
             dispatcher.dispatch(message);
```

RealTimeDispatchHandler (cont.)

```
            }
        }
        catch (IOException ioe)
        {
            System.out.println("Problem with the console");
        }
    }

    public void start()
    {
        processDispatches();
    }

}
```



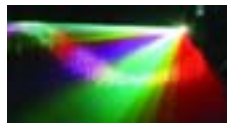
Why We Need Multi-Threading

- The Problem:

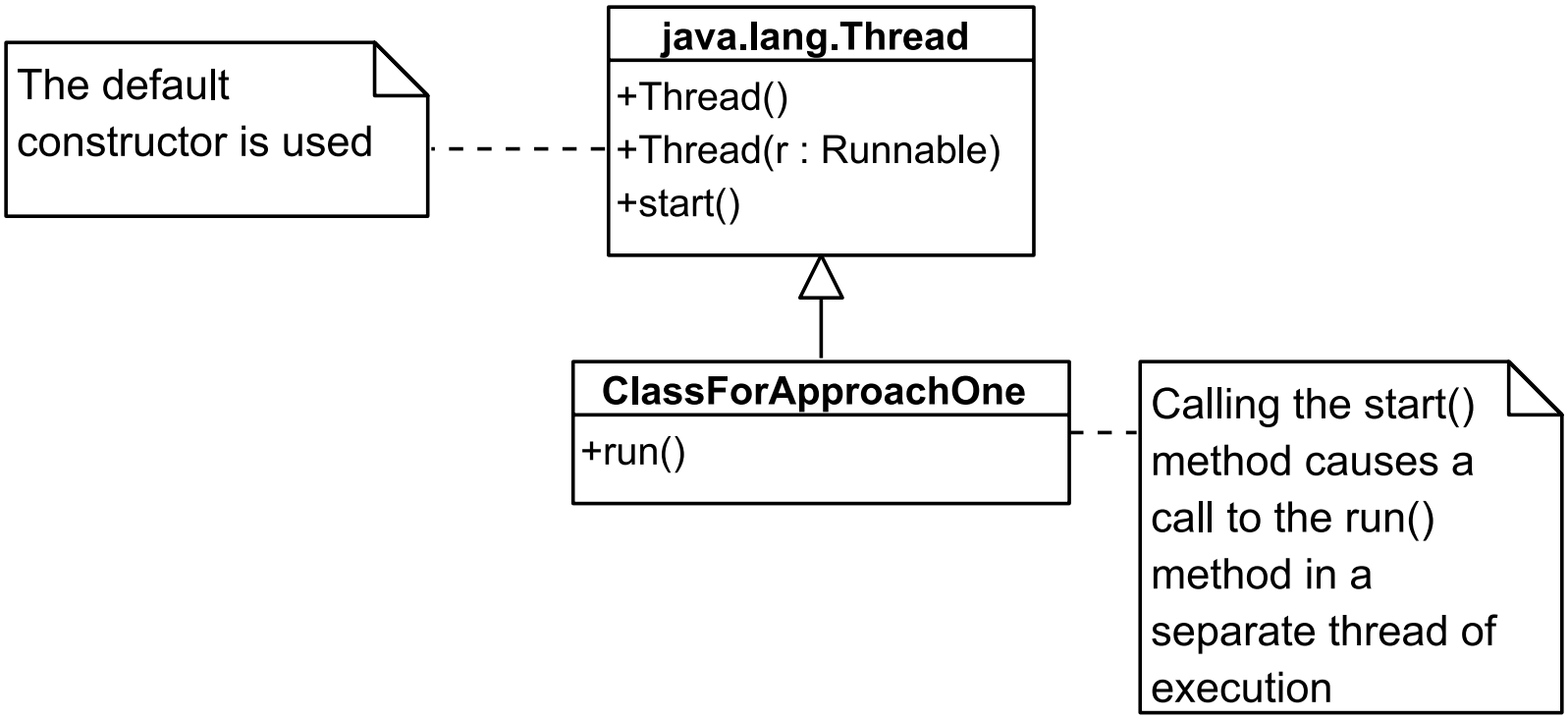
  The `RealTimeDispatchHandler` can't start doing any work until the `DailyDispatchHandler` has completed its job

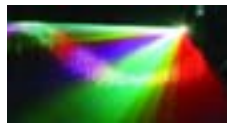
- What's Needed:

  A way for multiple objects to use the `Dispatcher` "at the same time"



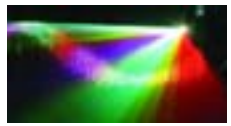
Specializing the Thread Class



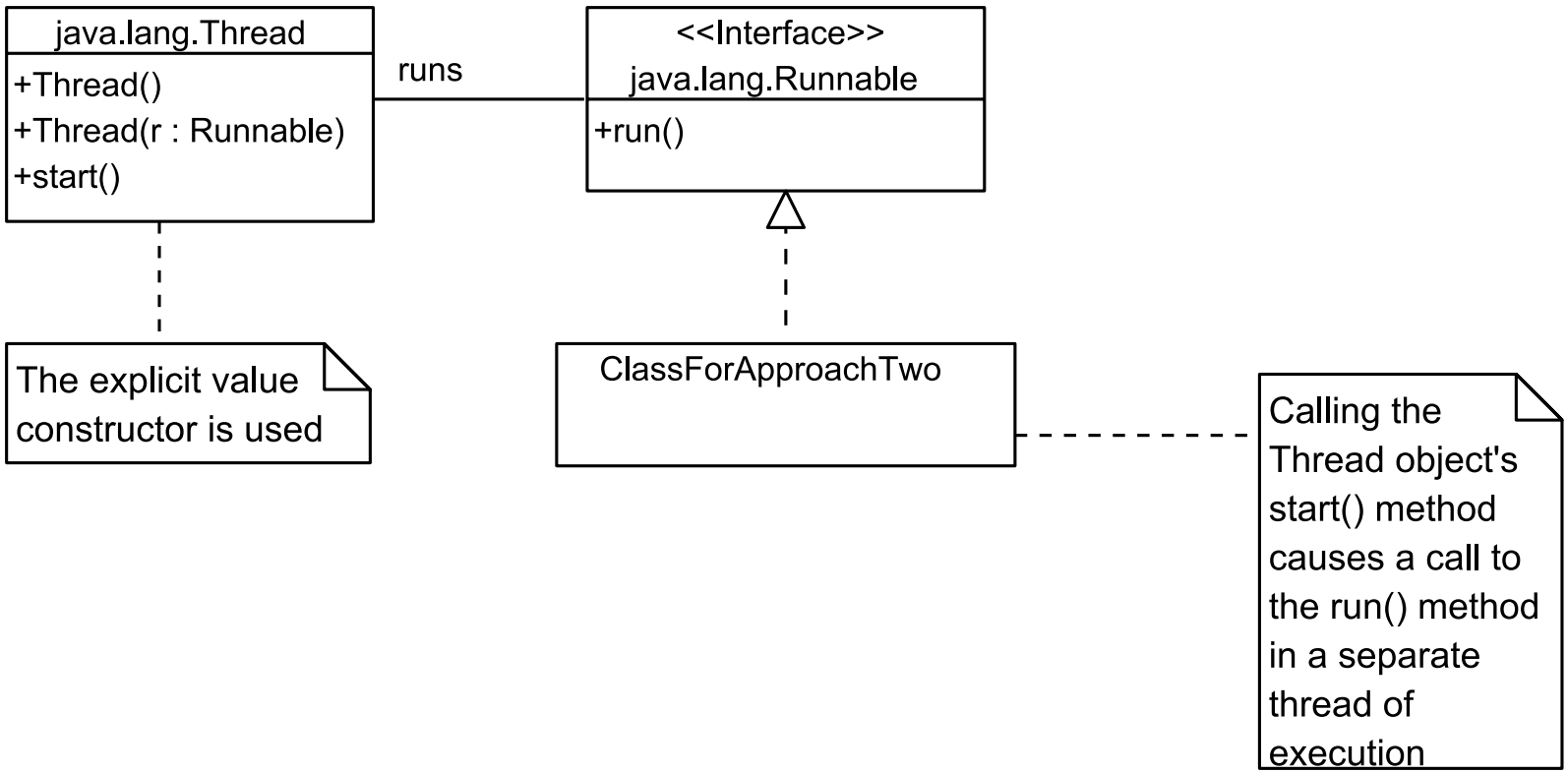


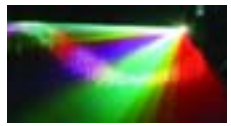
Specializing the `Thread` Class - Shortcomings

- Leads people to believe that the `ClassForApproachOne` object is a "thread of execution"
- A class that extends `Thread` can't extend another class



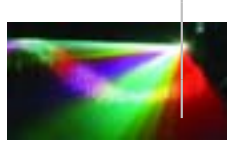
Implementing the `Runnable` Interface





The DailyDispatchHandler Revisited

```
import java.io.*;
import java.util.*;

public class DailyDispatchHandler implements Runnable
{
    private Dispatcher  dispatcher;
    private String      fileName;
    private Thread      controlThread;

    public DailyDispatchHandler(Dispatcher d, String f)
    {
       dispatcher = d;
       fileName = f;
    }



    public void run()
    {
       BufferedReader    in;
       int               wait;
       String            line, message;
       StringTokenizer   st;

       try
       {
          in = new BufferedReader(new FileReader(fileName));
```

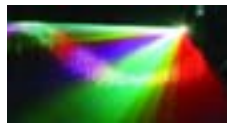


The DailyDispatchHandler Revisited (cont.)

```
      while ((line = in.readLine()) != null)
      {
         st = new StringTokenizer(line,"\t");
         wait = Integer.parseInt(st.nextToken());
         message = st.nextToken();

         try
         {
            // Sleep the appropriate amount of time
            // before dispatching the vehicle.  Other
            // threads can execute while this one
            // is sleeping.
            //
            controlThread.sleep(wait);
         }
         catch (InterruptedException ie)
         {
            // Do nothing
         }

         dispatcher.dispatch(message);
      }
   }
   catch (IOException ioe)
   {
      System.out.println("No daily dispatches "+
                         "in: "+fileName);
```

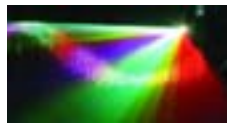
The DailyDispatchHandler Revisited (cont.)

```
        }
        catch (NoSuchElementException nsee)
        {
           System.out.println("Problem in file: "+fileName);
        }
    }

}
```



The DailyDispatchHandler Revisited (cont.)

```
public void start()
{
   if (controlThread == null)
   {
      controlThread = new Thread(this);
      controlThread.start();
   }
}
```



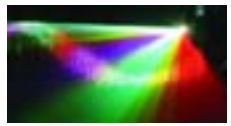
"At the Same Time"

- Computers with Multiple CPUs or Cores:

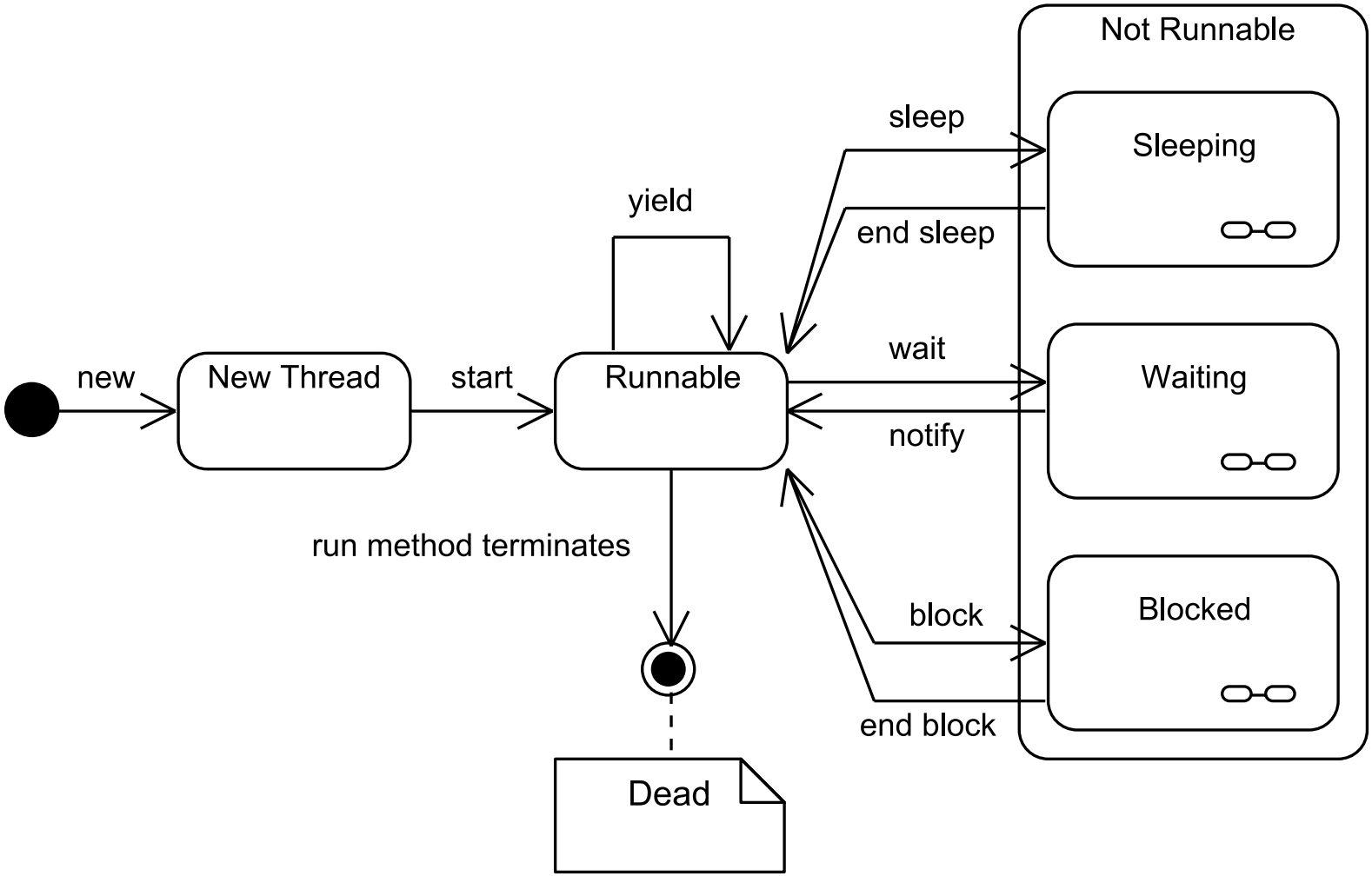
  Multiple threads can be executed at the same time

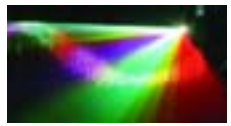
- Computers with One CPU/Core:

  Multiple threads take turns



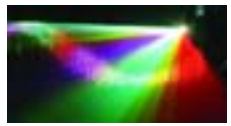
The Thread Lifecycle in Java





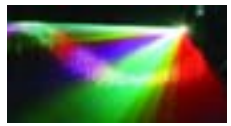
Types of Threads

- The *daemon status* can be changed using the `setDaemon()` method
- Daemon threads are normally used for background/helper activities
- A thread should be marked as a daemon only if it can be safely destroyed at any time (i.e., only if it is safe to stop executing code in that thread at any time)



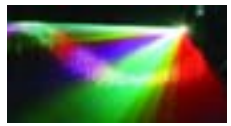
Interruption

- Every `Thread` object has an `interrupt()` method that can be used to set its *interrupt status* to `true`
- This method is used to ask a thread to stop what it is currently doing
- If you want to write a method that can be "cancelled" you need to include code that checks the interrupt status periodically using the `isInterrupted()` method in the `Thread` class



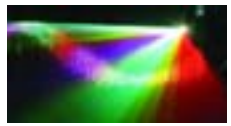
A Tracing Example

```
public class SlasherDriver
{
    public static void main(String[] args)
    {
        Slasher     plus, slash;

1       slash = new Slasher();
6       slash.setCount(3);
8       slash.start();
10      plus = new Slasher("+");
14      plus.setCount(2);
16      plus.start();
    }
}


public class Slasher implements Runnable
{
    private int         count;
    private String      symbol;
    private Thread      controlThread;

    public Slasher()
    {
2       this("/");
    }
```



A Tracing Example (cont.)

```
    public Slasher(String symbol)
    {
3 11    this.symbol = symbol;
4 12    count = 0;
5 13    controlThread = new Thread(this);
    }

    public void run()
    {
A C E G  for (int i=0; i<count; i++)  a c e
        {
B D F       System.out.print(symbol);  b d
        }
    }

    public void setCount(int count)
    {
7 15    this.count = count;
    }

    public void start()
    {
9 17    controlThread.start();
    }
}
```



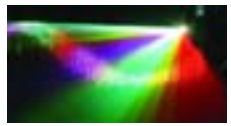
A Potential Problem

- The Issue:

  Conflicts can arise when more than one thread is using the same object

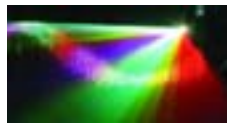
- An Example:

  Suppose two threads are both using the same `Dispatcher` object and there is only one vehicle in the queue



A Potential Problem (cont.)

```
public boolean dispatch(String task)
{
   boolean  ok;
   int      vehicle;
   Integer  v;

   ok = false;
   if (availableVehicles.size() > 0)
   {
      v = availableVehicles.removeFirst();
      vehicle = v.intValue();
      sendMessage(vehicle, task);
      ok = true;
   }
   else
   {
      ok = false;
   }


   return ok;
}
```



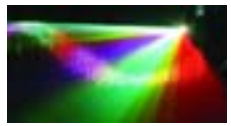
Race Conditions

- Defined:

  Code that causes the correctness of a computation to depend on the relative timing of different threads

- This Example:

  A *check-then-act* condition

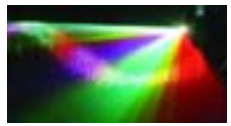


Race Conditions (cont.)

An Example with a *Read-Modify-Write* Condition:

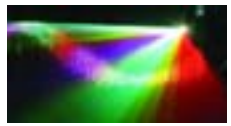
```
public int getNextIndex()
{
    return ++index;
}
```

The expression `++index` actually performs three operations – load the value, increment the value, and store the value



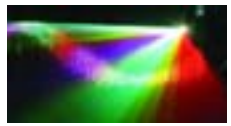
"Concurrency Protection" Objects

- In Java, every object (and class) has a *monitor* (or an *intrinsic lock*)
- When a thread of execution reaches a *synchronized* block or method it attempts to acquire the relevant monitor
- A thread of execution can only enter a synchronized method/block if it can acquire the relevant monitor and only one thread can acquire a monitor at a time



A Synchronized Method

```
public synchronized boolean dispatch(String task)
{
   boolean  ok;
   int      vehicle;
   Integer  v;

   ok = false;
   v  = availableVehicles.removeFirst();

   if (v == null) ok = false;
   else
   {
      vehicle = v.intValue();
      sendMessage(vehicle, task);
      ok = true;
   }

   return ok;
}
```



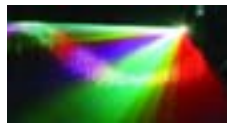
Is the Problem Fixed?

- The Good:

  The entire method now behaves as if it were atomic

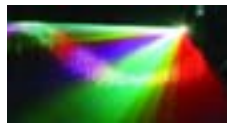
- The Bad:

  There are other methods in the class that change state (i.e., both the `dispatch()` method and the `makeVehicleAvailable()` method change the attribute `availableVehicles`)



Another Synchronized Method

```
public synchronized void makeVehicleAvailable(int vehicle)
{
   availableVehicles.addLast(new Integer(vehicle));
}
```



Liveness Failures

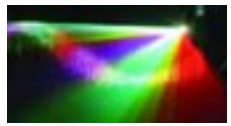
- Defined:

  A state in which an application/algorithm is unable to make any progress

- Types:

  Deadlock - when two or more threads are waiting on conditions that can't be satisfied

  Livelock - a thread can't make progress because it repeatedly attempts an operation that fails (e.g., you first, no you first)



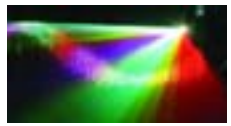
A Possible Performance Failure

- The Problem:

  The `sendMessage()` methods is being executed in either the `DailyDispatchHandler` object's thread or in the main thread and the thread might block

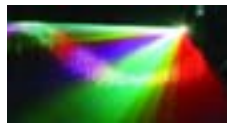
- A Better Approach:

  Have the `dispatch()` method in the `Dispatcher` class return almost immediately and have the messaging code execute in another thread



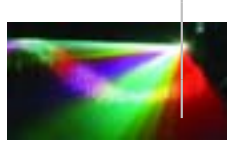
Adding a "Task Queue"

```
public void dispatch(String task)
{
   // Add a task to the queue
   tasks.add(tasks.size(), task);
}
```



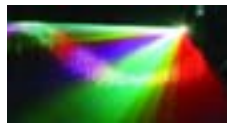
Adding a Thread

```
import java.util.*;


public class Dispatcher implements Runnable
{
    private          int              numberOfVehicles;
    private          List<Integer>    availableVehicles;
    private          List<String>     tasks;
    private          Thread           dispatchThread;


    public Dispatcher(int n)
    {
       int     i;

       numberOfVehicles = n;

       availableVehicles =
        Collections.synchronizedList(new LinkedList<Integer>());

       tasks =
        Collections.synchronizedList(new LinkedList<String>());

       for (i=0; i < n; i++) makeVehicleAvailable(i);

       // Start the thread
       dispatchThread = new Thread(this);
       dispatchThread.start();
```



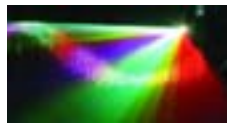
Adding a Thread (cont.)

```
    }

    private void sendMessage(int vehicle, String message)
    {

       // This method would normally transmit the message
       // to the vehicle.  For simplicity, it now writes it
       // to the screen instead.

       System.out.println(vehicle+"\t"+message+"\n");
       System.out.flush();
    }

}
```



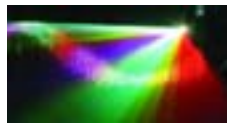
Modifying run()

```
public void run()
{
   while (true)
   {
      processPendingDispatches();

      try
      {
         dispatchThread.sleep(1000);
      }
      catch (InterruptedException ie)
      {
         // Shouldn't be interrupted.  If it is,
         // just continue.
      }
   }
}
```



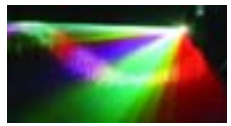
Modifying processPendingDispatches()

```
private void processPendingDispatches()
{
   int      vehicle;
   Integer  v;
   String   task;

   while ((availableVehicles.size()>0) && tasks.size()>0)
   {
      v = availableVehicles.remove(
                              availableVehicles.size()-1);

      task = tasks.remove(tasks.size()-1);

      vehicle = v.intValue();
      sendMessage(vehicle, task);
   }
}
```



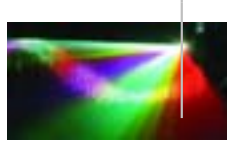
Considering this Design

- It is troubling to be putting the `dispatchThread` object to sleep for an arbitrary amount of time
- It would be better for the `dispatchThread` to wait until there are either new tasks to be dispatched or new vehicles to dispatch them to



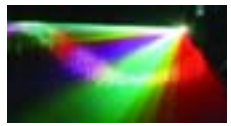
A Better Design

```
import java.util.LinkedList;


public class Dispatcher implements Runnable
{
    private int                     numberOfVehicles;
    private LinkedList<Integer>     availableVehicles;
    private LinkedList<String>      tasks;
    private Thread                  dispatchThread;


    private final Object lock = new Object();


    public Dispatcher(int n)
    {
       int     i;

       numberOfVehicles  = n;
       availableVehicles = new LinkedList<Integer>();
       tasks             = new LinkedList<String>();


       for (i=0; i < n; i++)
       {
          makeVehicleAvailable(i);
       }
```



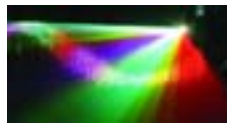
A Better Design (cont.)

```
        dispatchThread = new Thread(this);
        dispatchThread.start();
    }

    private void sendMessage(int vehicle, String message)
    {

        // This method would normally transmit the message
        // to the vehicle.  For simplicity, it now writes it
        // to the screen instead.

        System.out.println(vehicle+"\t"+message+"\n");
        System.out.flush();
    }

}
```



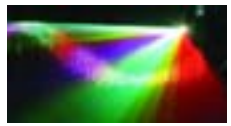
A Better Design (cont.)

```
public void dispatch(String task)
{
   synchronized(lock)
   {
      // Add a task to the queue
      tasks.addLast(task);

      // Start the processing
      lock.notifyAll();
   }
}
```



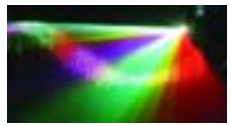
A Better Design (cont.)

```
public void makeVehicleAvailable(int vehicle)
{
   synchronized(lock)
   {
      // Put the vehicle in the queue
      availableVehicles.addLast(new Integer(vehicle));

      // Start the processing
      lock.notifyAll();
   }
}
```



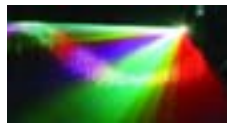
A Better Design (cont.)

```
public void run()
{
   while (true)
   {
      synchronized(lock)
      {
         processPendingDispatches();

         try
         {
            lock.wait();
         }
         catch (InterruptedException ie)
         {
            // Shouldn't be interrupted.  If it is,
            // just continue.
         }
      }
   }
}
```



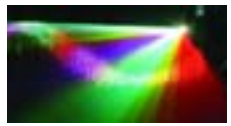
Considering the Improved Design

- The thread that is handling the dispatches never dies since it never "drops out of" the `run()` method
- This problem can be corrected by adding a `boolean` attribute named `keepRunning` and modifying the `run()`



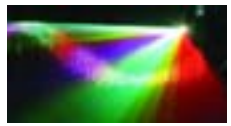
A Still Better Design

```
public void run()
{
   while (keepRunning)
   {
      synchronized(lock)
      {
         processPendingDispatches();

         try
         {
            lock.wait();
         }
         catch (InterruptedException ie)
         {
            // The stop() method was called in
            // another thread
         }
      }
   }

   dispatchThread = null;
}
```



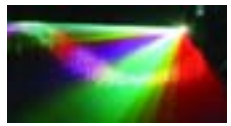
A Still Better Design (cont.)

```
public void stop()
{
   synchronized(lock)
   {
      keepRunning = false;

      // Interrupt the thread in case it
      // is waiting
      dispatchThread.interrupt();
   }
}
```



A Still Better Design (cont.)

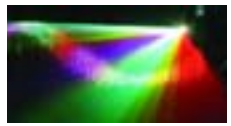
```
public void start()
{
   if (dispatchThread == null)
   {
      keepRunning    = true;
      dispatchThread = new Thread(this);
      dispatchThread.start();
   }
}
```



A Problem Remains

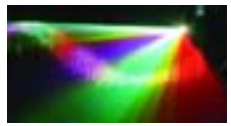
- The `stop()` method is going to be executed in a different thread than the `run()` method and they both use the `keepRunning` attribute

- This causes a problem because Java does not ensure that changes to attributes that are made in one thread propogate to other threads in the way you would expect (i.e., for performance reasons, values can be "cached" in such a way that they are hidden from other threads)



Volatile Attributes

- *Volatile* attributes are, in essence, attributes that are shared across multiple threads
- A "load" of a volatile attribute in any thread always returns the most recent "store" performed in any thread



Volatile Attributes (cont.)

```
private volatile boolean        keepRunning;
```