

## Multithreading

Horstmann ch.9

## Multithreading

- Basics
  - thread state: runnable, blocked
  - start, sleep, interrupt
- Synchronization
  - race condition
  - object lock: synchronization
  - deadlock avoidance: wait, notifyall
- Animation
  - example: mergesort

## 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

## Running Threads

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

## Running Threads

```
public class MyRunnable implements Runnable {
    public void run() {
        thread action
    }
}
```

...

```
Runnable r = new MyRunnable();
Thread t = new Thread(t);
t.start();
```

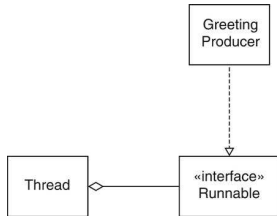
## Thread Example

- 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`

## Thread Example



## Class GreetingProducer

```

public class GreetingProducer implements Runnable {
    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;
}
  
```

## Class ThreadTest

```

public class ThreadTest {
    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();
    }
}
  
```

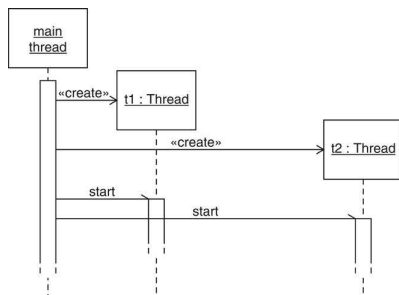
## Thread Example

- Note: output not exactly interleaved

```

1: Hello, World!
1: Goodbye, World!
2: Hello, World!
2: Goodbye, World!
3: Hello, World!
3: Goodbye, World!
4: Hello, World!
4: Goodbye, World!
5: Hello, World!
5: Goodbye, World!
6: Hello, World!
6: Goodbye, World!
7: Hello, World!
7: Goodbye, World!
8: Goodbye, World!
8: Hello, World!
9: Goodbye, World!
9: Hello, World!
10: Goodbye, World!
10: Hello, World!
  
```

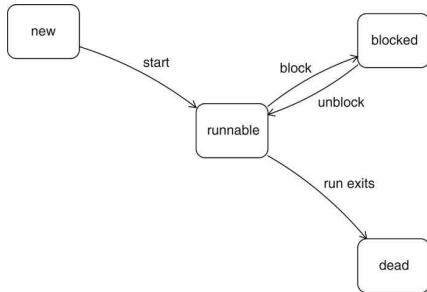
## Starting Two Threads



## Thread States

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

## Thread States



## Blocked Thread State

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

## 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

## Terminating Threads

- Thread terminates when `run` exits
- Sometimes necessary to terminate running thread
- Don't use deprecated `stop` method
- Interrupt thread by calling `interrupt`
- Thread must cooperate and exit its `run`
- 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: 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

## Producer Thread

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

all threads use same queue

## Consumer Thread

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

### • Expected Program Output

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

### • Why is Output Corrupted?

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

## Circular Array Implementation



- Efficient implementation of bounded queue
- Avoids inefficient shifting of elements
- Circular: head, tail indexes wrap around
- In circular array implementation, failure of remove precondition corrupts queue!

## add method in Queue

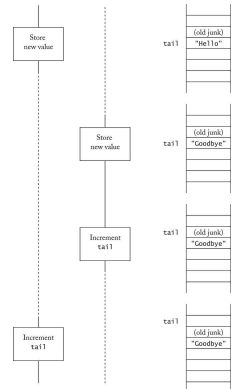
```
public void add(Object anObject) {
    if (debug) System.out.print("add");
    elements[tail] = anObject;
    if (debug) System.out.print(".");
    tail++;
    if (debug) System.out.print(".");
    size++;
    if (tail == elements.length) {
        if (debug) System.out.print(".");
        tail = 0;
    }
    if (debug) System.out.println("head=" + head +
        ",tail=" + tail + ",size=" + size);
}
```

## removeFirst method in Queue

```
public Object removeFirst() {
    if (debug) System.out.print("removeFirst");
    Object r = 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;
}
```

## 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++;`



## Object Locks

- Each object has a lock
- Thread can lock, unlock object
- No other thread can lock object until unlocked
- Use synchronized method
- Calling synchronized method locks implicit parameter
- No other thread can call another synchronized method on same object
- Place thread-sensitive code inside synchronized methods

```
public synchronized void add(Object anObject) { ... }
```

## Scenario with Synchronized Methods

- First thread calls add and executes `elements[tail] = anObject;`
- First thread at end of time slice
- Second thread tries to call `add`, blocks
- First thread executes `tail++;`
- First thread completes add, unlocks queue
- Second thread unblocked
- Second thread calls `add`

## Visualizing Locks

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



## Deadlocks

- Not enough to synchronize `add, removeFirst`
- `if (!queue.isFull()) queue.add(...);`  
can still be interrupted
- Must move test inside `add` method
- What should `add` do if queue is full?
- Consumer will remove objects
- Producer needs to wait
- Producer can't go to sleep inside locked object
- Person sleeping in phone booth, waiting for technician?

## Avoiding Deadlocks

- Call `wait` in `add` method:

```
public synchronized void add(Object anObject)
    throws InterruptedException {
    while (isFull()) wait();
}
```

- Waiting thread is blocked
- Waiting thread is added to *wait set* of object

## Avoiding Deadlocks

- To unblock waiting thread, some other thread calls `notifyAll`
- Must be called *on same object*
- Call when state changes

```
public synchronized Object removeFirst() {
    ...
    notifyAll();
}
```

- All waiting threads removed from wait set, unblocked

## Avoiding Deadlocks

- `wait` and `notifyAll` called in both `add` and `removeFirst`:

```
public synchronized void add(Object anObject) throws
    InterruptedException {
    while (size == elements.length) wait();
    elements[tail] = anObject;
    tail++; size++;
    if (tail == elements.length) tail = 0;
    notifyAll();
}
```

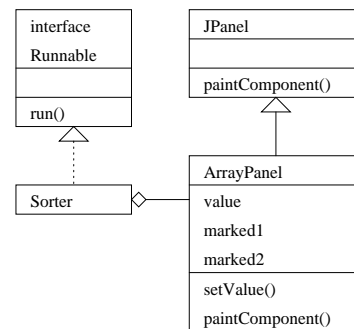
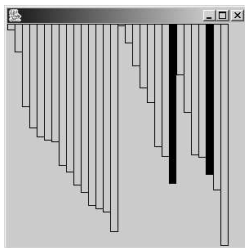
needed when a Consumer thread calls `wait` in `removeFirst`

## Algorithm Animation

- Use thread to make progress in algorithm
- Display algorithm state
- Example: `Animate MergeSorter` (similar to `java.util.Arrays.sort`)
- Pause inside compare method
- Pass custom comparator

```
Comparator comp = new
    Comparator() {
    public void compare(Object o1, Object o2) {
        draw current state
        pause thread
        return comparison result
    };
}
```

## Algorithm Animation



```

public class Sorter implements Runnable {
    public Sorter(Object[] values, ArrayPanel panel) {
        this.values = values; this.panel = panel;
    }
    public void run() {
        Comparator comp = new
        Comparator() {
            public int compare(Object o1, Object o2) {
                panel.setValues(values, o1, o2);
                try { Thread.sleep(DELAY); }
                catch (InterruptedException exception) {
                    Thread.currentThread().interrupt();
                }
                return ((Integer) o1).compareTo(o2);
            }
        };
        MergeSorter.sort(values, comp);
        panel.setValues(values, null, null);
    }
    private Object[] values;
    private ArrayPanel panel;
    private static final int DELAY = 100;
}

```

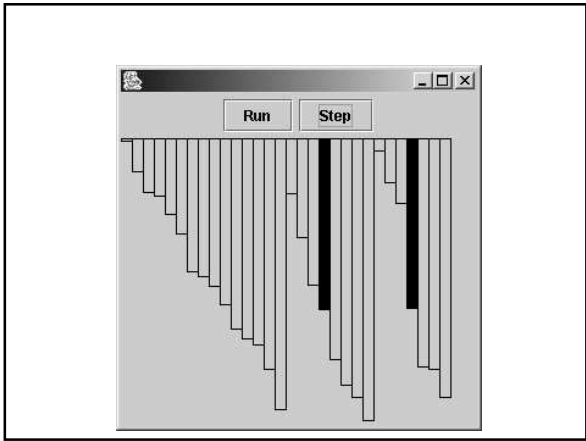
array being sorted

### Pausing and Running the Animation

- Want to pause animation until button is clicked
- Need to coordinate UI thread, animation thread
- Easier to understand coordination with custom object
- Both threads query, manipulate *gate*

**Gate**

- Works like a security gate
- Sorter thread = car approaching gate
- Passes through if gate open, pauses otherwise
- Animation thread = security guard
- Step button opens gate to let one car through
- Run button deactivates gate--always open



### Pausing and Running the Animation

- compare method checks gate calls wait (sorter thread)

```

if (gate.isActive()) gate.waitForOpen();

```

- Button callbacks set gate state

```

runButton.addActionListener(
    ... gate.setActive(false); ...
);
stepButton.addActionListener(
    ... gate.setOpen(true); ...
);

```

calls notifyAll (event dispatching thread)

### Class Gate

```

public class Gate {
    public boolean isActive() { return active; }
    public synchronized void setActive(boolean newValue) {
        active = newValue;
        notifyAll();
    }
    public synchronized void setOpen(boolean newValue) {
        active = true;
        opened = newValue;
        notifyAll();
    }
    public synchronized void waitForOpen()
        throws InterruptedException {
        while (active && !opened) wait();
        opened = false;
    }
    private boolean active;
    private boolean opened;
}

```

