

# Multithreading

# Threads and Concurrency

---

- A **thread** is a single sequential flow of operations with a definite beginning and an end
  - It is also called a *lightweight* process
  - Java has built-in support for **concurrent programming** by running **multiple threads** concurrently within a **single program**
- Two basic units of execution in concurrent programming:
  - A **process** has a self-contained execution environment.
    - A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space (variables)
  - **Threads** exist within a process — every process has at least one.
    - Threads **share** the process's resources, including memory (variables) and open files (somewhat risky but easier to program; processes require *Inter Process Communication*, IPC, to cooperate)
    - Fewer resources are required to create a new thread than a process

# Why Multithreading ?

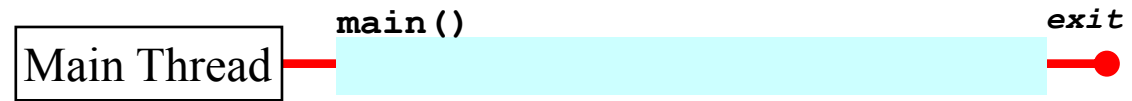
---

- **Multitasking** is the ability of a computer to perform many tasks concurrently
  - *Pre-emptive multitasking systems*
    - Tasks are given time-slices of the CPU(s) and will be forced to yield control to other tasks once their allocation is used up
    - eg: UNIX/Linux, Win32/64, Mac OS X
  - *Co-operative multitasking systems*
    - Each task must *voluntarily* yield control to other tasks
    - This has the drawback that a run-away or uncooperative task may hang the entire system
    - eg: Windows 3.1, Mac OS 9
- Multithreading
  - *Intra-program concurrency*: multithreading provides a way to have more than one thread executing in the same process while allowing every thread access to the same memory address space

- **Multithreading is extremely useful in practice**
  - **Fair Warning: multithreading can get very complex**

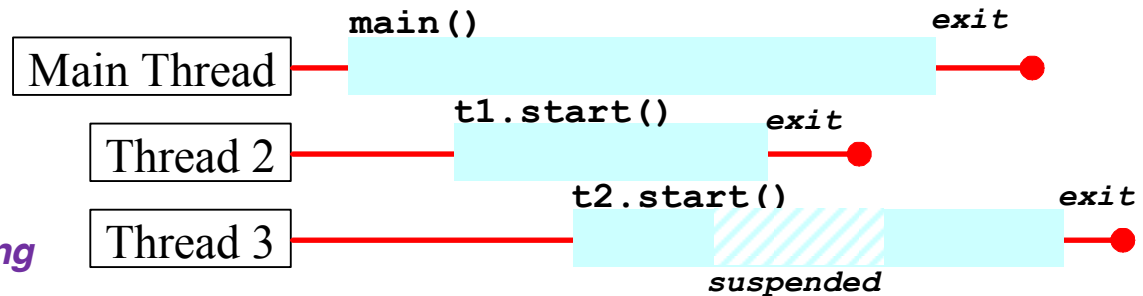
# Multitasking and Threads

Single thread program



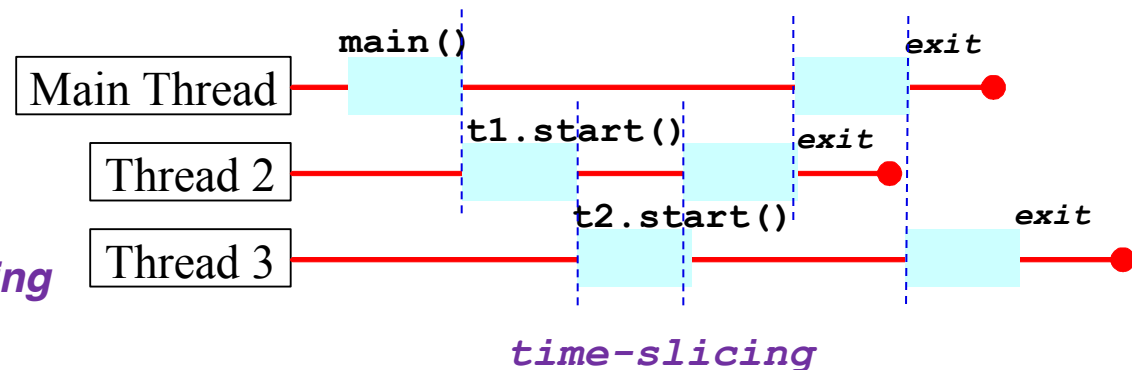
Multiple threads on multiple CPUs

*Simultaneous multitasking*



Multiple threads on a single CPU

*Time-sliced multitasking*



# When To Use Threads

---

- All Java GUI programs are multithreaded
  - You have to live with threads
- Threads for computationally intensive tasks
  - You want more responsive UI and do not want your program to freeze
  - The long-running task is executed in a separate thread so that the event dispatch thread remains free to process UI events,
- Tasks are parallelizable
  - It is natural to use a thread for each task
  - Imagine the characters etc. in games
- You need to handle multiple concurrent I/O channels
  - Downloading several images/serving many concurrent clients
  - Any I/O channels can block
- You have a multicore CPU
  - Subtasks are performed on individual CPU cores so that the overall task is finished more quickly
  - Multicore CPUs are common today

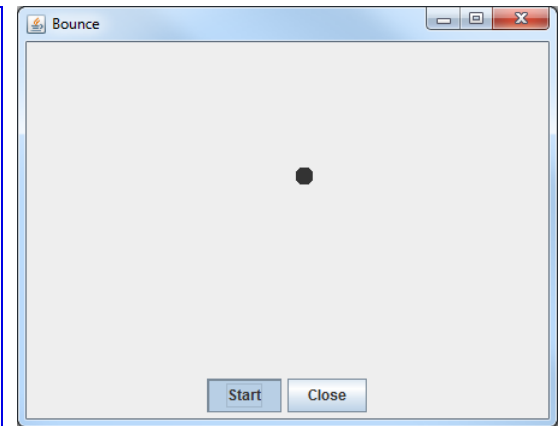
# Risks of Threads

---

- Safety hazards
  - Programs that behave correctly in a single-threaded environment may not in the absence of synchronization in a multithreaded environment
  - Safety: “nothing bad ever happens”
- Liveness hazards
  - Use of threads introduces liveness failures that do not occur in single-threaded programs
  - Liveness: “something good eventually happens”
- Performance hazards
  - Performance issues subsume a broad range of problems
    - Poor service time, responsiveness, throughput, resource consumption, or scalability
  - Performance: “good things happen quickly”

# Bouncing Example: Single Threaded

```
Ball ball = new Ball();  
panel.add(ball);  
  
for (int i = 1; i <= STEPS; i++) {  
    ball.move(...);  
    ...  
    Thread.sleep(DELAY);  
    // just pause; does not create  
    // a new thread  
}
```



You cannot interact with the program until the ball finished bouncing (it finished its 1000 bounces)

# Creating Threads

## by Implementing the Runnable Interface

```
// Custom task class
public class TaskClass
    implements Runnable {
    ...
    public TaskClass(...) {
        ...
    }

    // Implement the run method
    // in Runnable
    public void run() {
        // Tell system how to run
        // custom thread
        ...
    }
    ...
}
```

```
// Client class
public class Client {
    ...
    public static void main(String[] args){
        ...
        // Create an instance of TaskClass
        Runnable task = new TaskClass(...);

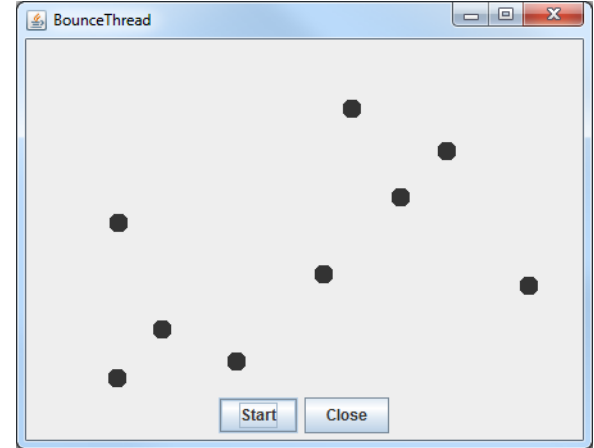
        // Create a thread
        Thread thread = new Thread(task );

        // Start a thread
        thread.start();
        ...
    }
    ...
}
```



# Bouncing Example: Multithreaded

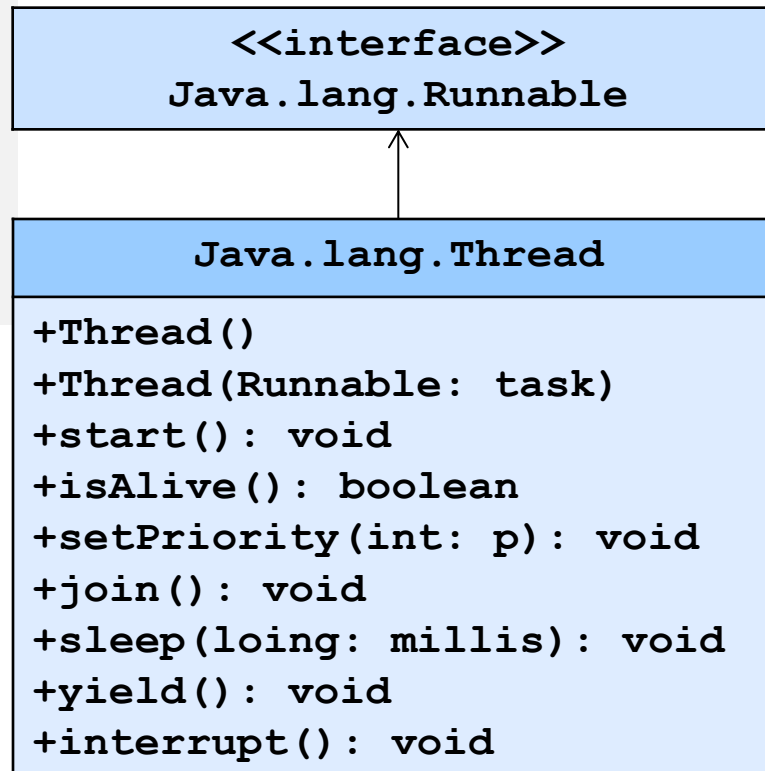
```
class BallRunnable implements Runnable
{
    ...
    public void run() {
        try{
            for (int i = 1; i <= STEPS; i++){
                ball.move();
                ...
                sleep(DELAY);
            }
        } catch (InterruptedException exception) {
        }
    }
}
```



Using Threads to give other tasks a chance

# Thread Class

---



# Creating Threads

## by Extending the Thread Class

```
// Custom thread class
public class CustomThread extends Thread {
    ...
    public CustomThread(...) {
        ...
    }

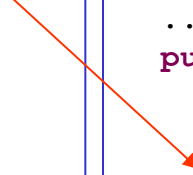
    // Override the run method in Thread
    public void run() {

        // Tell system how to run custom thread
        ...

    }
    ...
}
```

```
// Client class
public class Client {
    ...
    public someMethod() {
        ...
        // Create a thread
        CustomThread thread =
            new CustomThread(...);

        // Start a thread
        thread.start();
        ...
    }
    ...
}
```



### This approach is not recommended:

1. You should decouple the task that is to be run in parallel from the mechanism of running it
2. You can use a thread pool to run many tasks – too expensive to create a separate thread for each

# Creating Threads: Steps

---

- Implementing the **Runnable** Interface

- Implement the **Runnable** interface
- Override the **run()** method.
- Create a thread with **new Thread(runnable)**.
- Start the thread by calling the **start()** method.

Better OO design; single inheritance; consistency; applicable to the high-level thread management APIs

- Extending the **Thread** class

- Subclass the **Thread** class.
- Override the **run()** method.
- Create a thread with **new MyThread(...)**.
- Start the thread by calling the **start()** method.

Not recommended any more; simpler code for simple programs; potential issues

# Example: Unresponsive Flying Label

**Problem: Add a button to control the flying label**

```
public static void main(String[] args) {  
    ...  
    flyingLabel.fly();  
}  
  
public void fly(){  
    for (int i = 0; i<FLYTIME ;i++){  
        if(x > getWidth()){  
            x = -150;  
        }  
        x += 1;  
        label.setBounds(x,166,160,30);  
                                // relocating label  
        Thread.sleep(10); // slow it down  
    }  
}
```

*Move to button event handler*

```
JButton btnStart = new JButton("Start");  
btnStart.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent evt) {  
        stop=false;  
        fly();  
    }  
});  
  
JButton btnStop = new JButton("Stop");  
btnStop.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent evt) {  
        stop=true;  
    }  
});  
  
public void fly(){  
    for (int i = 0; i<FLYTIME ;i++){  
        if(!stop){ // fly only when not stopped  
            if(x > getWidth()){  
                x = -150;  
            }  
            x += 1;  
            label.setBounds(x,166,160,30);  
                                // relocating label  
        }  
        Thread.sleep(10); // slow it down  
    }  
}
```

*What is happening? Why ?*

# Example: Responsive Flying Label

Problem: Use thread to give Event Dispatch Thread a chance

```
JButton btnStart = new JButton("Start");
btnStart.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        stop=false;
        fly();
    }
});
JButton btnStop = new JButton("Stop");
btnStop.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        stop=true;
    }
});
public void fly(){
    for (int i = 0; i<FLYTIME ;i++){
        if(!stop){ // fly only when not stopped
            if(x > getWidth()){
                x = -150;
            }
            x += 1;
            label.setBounds(x,166,160,30);
            // relocating label
        }
        Thread.sleep(10); // slow it down
    }
}
```

*Move to a thread*

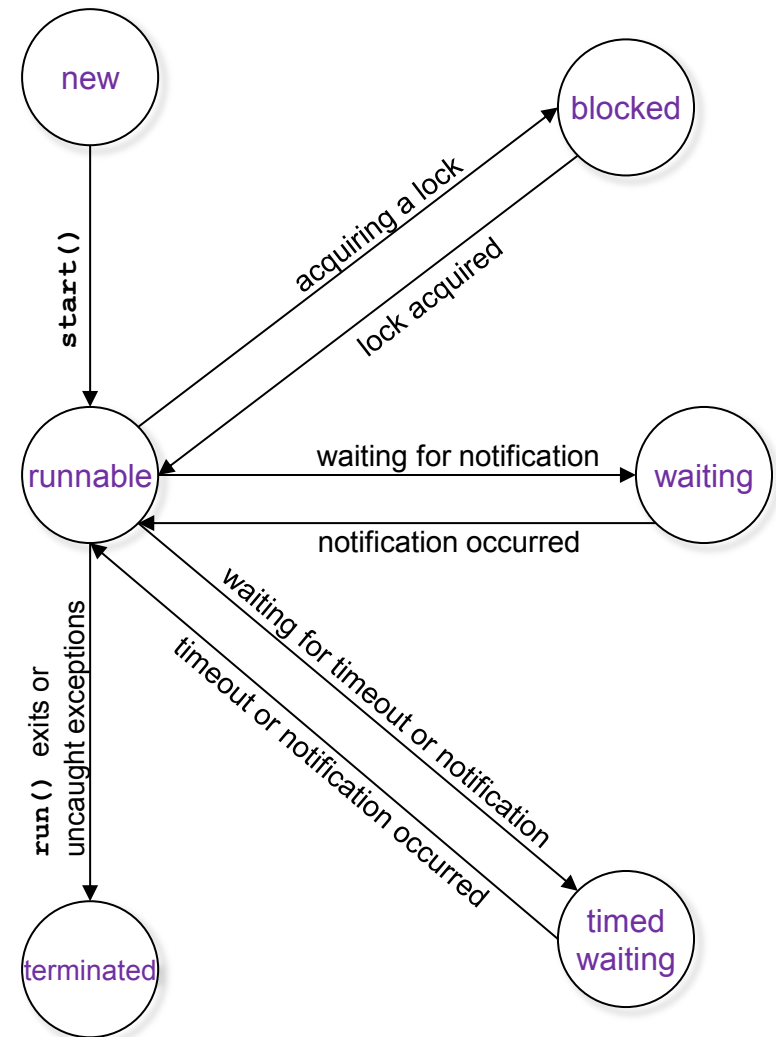
```
JButton btnStart = new JButton("Start");
btnStart.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        stop=false;
        Thread t = new Thread(){ // Create a thread
            public void run() {
                fly();
            }
        };
        t.start();
    }
});
JButton btnStop = new JButton("Stop");
btnStop.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        stop=true;
    }
});
public void fly(){
    for (int i = 0; i<FLYTIME ;i++){
        if(!stop){ // fly only when not stopped
            if(x > getWidth()){
                x = -150;
            }
            x += 1;
            label.setBounds(x,166,160,30);
            // relocating label
        }
        Thread.sleep(10); // slow it down and yield control
    }
}
```

*It is flying but why is it not thread safe ?*

# Thread States

## Call `getState()` method

- **NEW**  
A thread that has not yet started is in this state.
- **RUNNABLE**  
A thread executing in the Java virtual machine is in this state.
- **BLOCKED**  
A thread that is blocked waiting for a monitor lock is in this state.
- **WAITING**  
A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- **TIMED\_WAITING**  
A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- **TERMINATED**  
A thread that has exited is in this state.



# Thread Scheduling and Priority

---

- JVM implements a fixed priority thread-scheduling scheme
- JVM also implements a pre-emptive scheduling scheme in a pre-emptive environment
  - **Starvation**: one thread runs until completion without yielding control to other equal-priority threads
- The thread scheduling and priority is JVM dependent
  - Programs should not rely on the priority
    - Most JVM does not guarantee that the highest-priority thread is being run at all times; It may choose to dispatch a lower-priority thread for some reasons such as to prevent starvation
  - It is a good practice to yield control to other threads via the `sleep()` or `yield()` method
- Accessing priority
  - `get/setPriority(int)` methods
  - Each thread is assigned a default priority of `Thread.NORM_PRIORITY`
  - Some constants for priorities  
`Thread.MIN_PRIORITY`, `Thread.MAX_PRIORITY`, `Thread.NORM_PRIORITY`



# Controlling Threads

---

- Controlling a Thread From within the Thread
  - **Thread.sleep()**
    - voluntarily relinquishes control of the CPU to other threads for a designated interval of time
  - **Thread.yield()**
    - voluntarily relinquishes control of the CPU to other threads of equal (or higher) priority

```
public void fly(){
    for (int i = 0; i<FLYTIME ;i++){
        if(x > getWidth()){
            x = -150;
        }
        x += 1;
        label.setBounds(x,166,160,30);    // relocating label
        sleep(10);                        // slow it down
    }
}
```

# Controlling Threads

---

- Reading a Thread's State From Another Thread
  - There are methods you can invoke from one thread to monitor or respond to the state of another, notably:
    - **isAlive()** method
      - indicates whether the thread to which it is bound has started but not yet completed
    - **Join()** method
      - causes the current thread to wait until the thread it joins has terminated
      - Happens-before relationship

# Controlling Threads

---

- Controlling a thread from another thread: *interrupting threads*
  - An *interrupt* is an indication to a thread that it should stop what it is doing and do something else
    - It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate
  - Sending interrupts using **Thread.interrupt()**
    - For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption
  - The Interrupt Status Flag
    - Internal **boolean** flag: *interrupt status*
    - **Thread.interrupt()** sets this flag
    - Static method **Thread.interrupted()** checks and clears the flag
    - Non-static **isInterrupted()** method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag
  - **InterruptedException**
    - When the **interrupt()** method is called on a blocked thread, the blocking call is terminated by an **InterruptedException**

# Thread Interaction

---

- Methods for inter-thread communication

They are declared as `final` within `Object`, intended to void polling . All three methods can be called only from within a **synchronized** context.

- `wait()`

causes the current thread to wait until either another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified amount of time has elapsed

- Always call `wait()` in a loop that checks the condition being waited on

- `notify()/notifyAll()`

wakes up a single thread or all threads that are waiting on this object's monitor

- Always ensure that you satisfy the waiting condition before calling `notify()` or `notifyAll()`

- Swing programming notes:

- You must not suspend the EDT thread with a `wait()` or any other blocking call
  - It is not defined how you `notify()` to the EDT

# Example: Producer/Consumer Problem

```
class Q {
    int n;
    synchronized int get() {
        System.out.println("Got: " + n);
        return n;
    }
    synchronized void put(int n) {
        this.n = n;
        System.out.println("Put: " + n);
    }
}
```

```
class Producer implements Runnable {
    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}
```

```
class Consumer implements Runnable {
    public void run() {
        while(true) {
            q.get();
        }
    }
}
```

Incorrect implementation

Output

Put: 1  
Got: 1  
Got: 1  
Got: 1  
Put: 2  
Put: 3  
Put: 4  
Put: 5  
Put: 6  
Put: 7  
Got: 7

```
class Q {
    int n;
    boolean valueSet = false;
    synchronized int get() {
        if(!valueSet)
            try {
                wait();
            } catch (InterruptedException e) {}
        System.out.println("Got: " + n);
        valueSet = false;
        notify();
        return n;
    }
}
```

```
synchronized void put(int n) {
    if(valueSet)
        try {
            wait();
        } catch (InterruptedException e) {}
    this.n = n;
    valueSet = true;
    System.out.println("Put: " + n);
    notify();
}
```

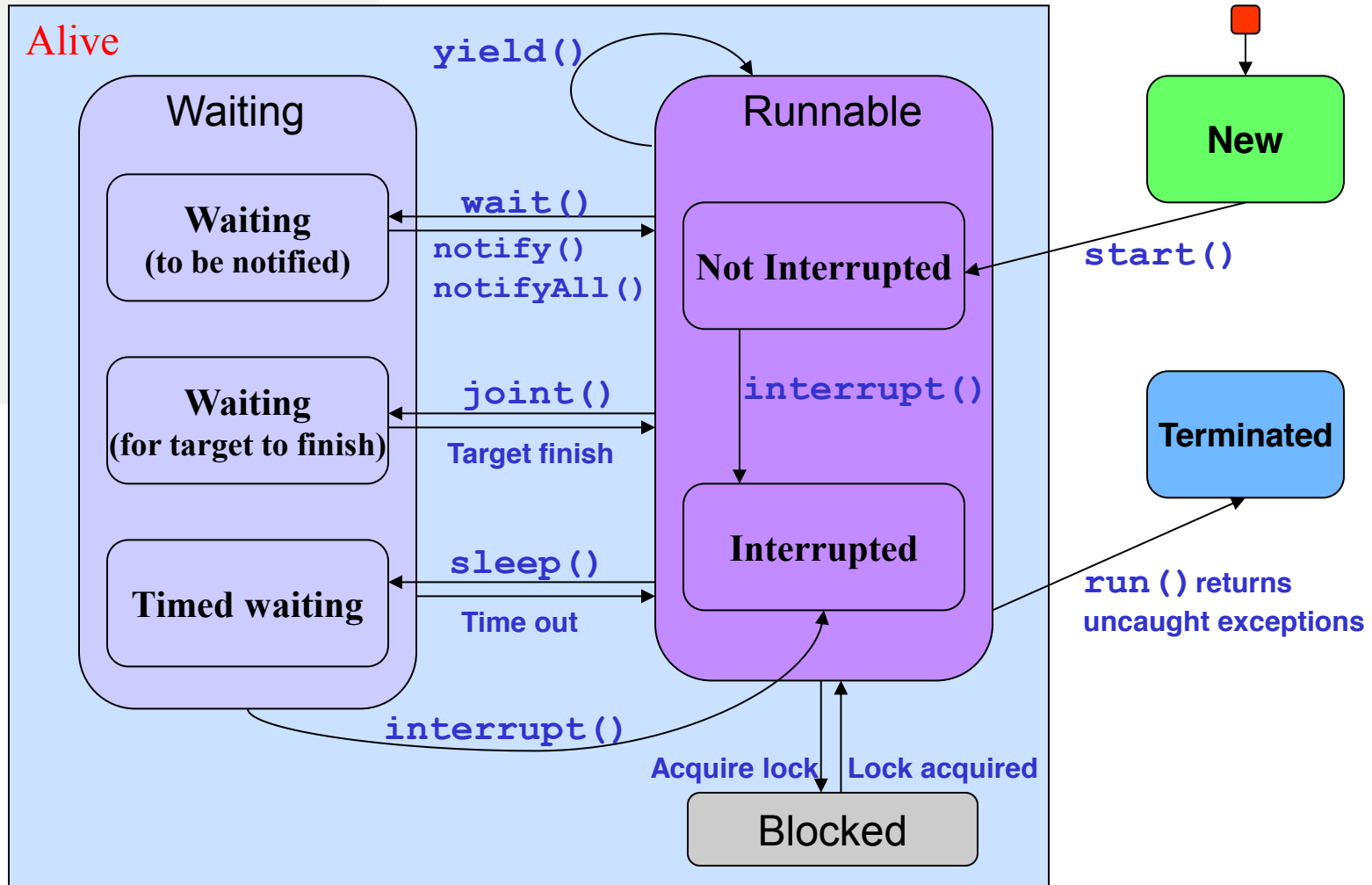
. . .

Correct implementation

Output

Put: 1  
Got: 1  
Put: 2  
Got: 2  
Put: 3  
Got: 3  
Put: 4  
Got: 4  
Put: 5  
Got: 5

# Life Cycle of Threads



# Thread Groups

---

- Construct a thread group using the **ThreadGroup** constructor:

```
ThreadGroup g = new ThreadGroup("thread group");
```

- Place a thread in a thread group using the **Thread** constructor:

```
Thread t = new Thread(g, "This thread");
```

- To find out how many threads in a group are currently running, use the **activeCount()** method:

```
System.out.println("The number of threads: " + g.activeCount());
```

# Example: Need for Synchronization

- Threads work asynchronously and may produce unexpected output if they are not coordinated

```
public class NumberThreadDemo {  
    public static void main( String [] args ) {  
        System.out.println( "Starting Main" );  
        for ( int i = 1 ; i <= 5 ; i++ ) {  
            Thread numberThread = new Thread(new NumberTask(i)) ;  
            numberThread.start() ;  
        }  
        System.out.println( "Ending Main" ) ;  
    }  
}
```

```
class NumberTask implements Runnable {  
    int count ;  
    public NumberTask( int count ) {  
        this.count = count ;  
    }  
    public void run() {  
        System.out.println( "Count : " + count ) ;  
    }  
}
```

```
> java NumberThreadDemo  
Starting Main  
Count : 1  
Count : 4  
Count : 2  
Ending Main  
Count : 3  
Count : 5
```



# Bank Example: Unsynchronized

```
public void transfer(int from, int to, int amount)
    throws InterruptedException
{
    if (accounts[from] < amount) return;
    System.out.print(Thread.currentThread());
    accounts[from] -= amount;
    System.out.printf(" %10.2f from %d to %d", amount, from, to);
    accounts[to] += amount;
    ntransacts++;
    System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
}
```

```
public void run()
{
    try{
        while (true){
            int toAccount = (int) (bank.size() * Math.random());
            double amount = maxAmount * Math.random();
            bank.transfer(fromAccount, toAccount, amount);
            sleep((int) DELAY * Math.random());
        }
    } catch (InterruptedException e) {}
}
```

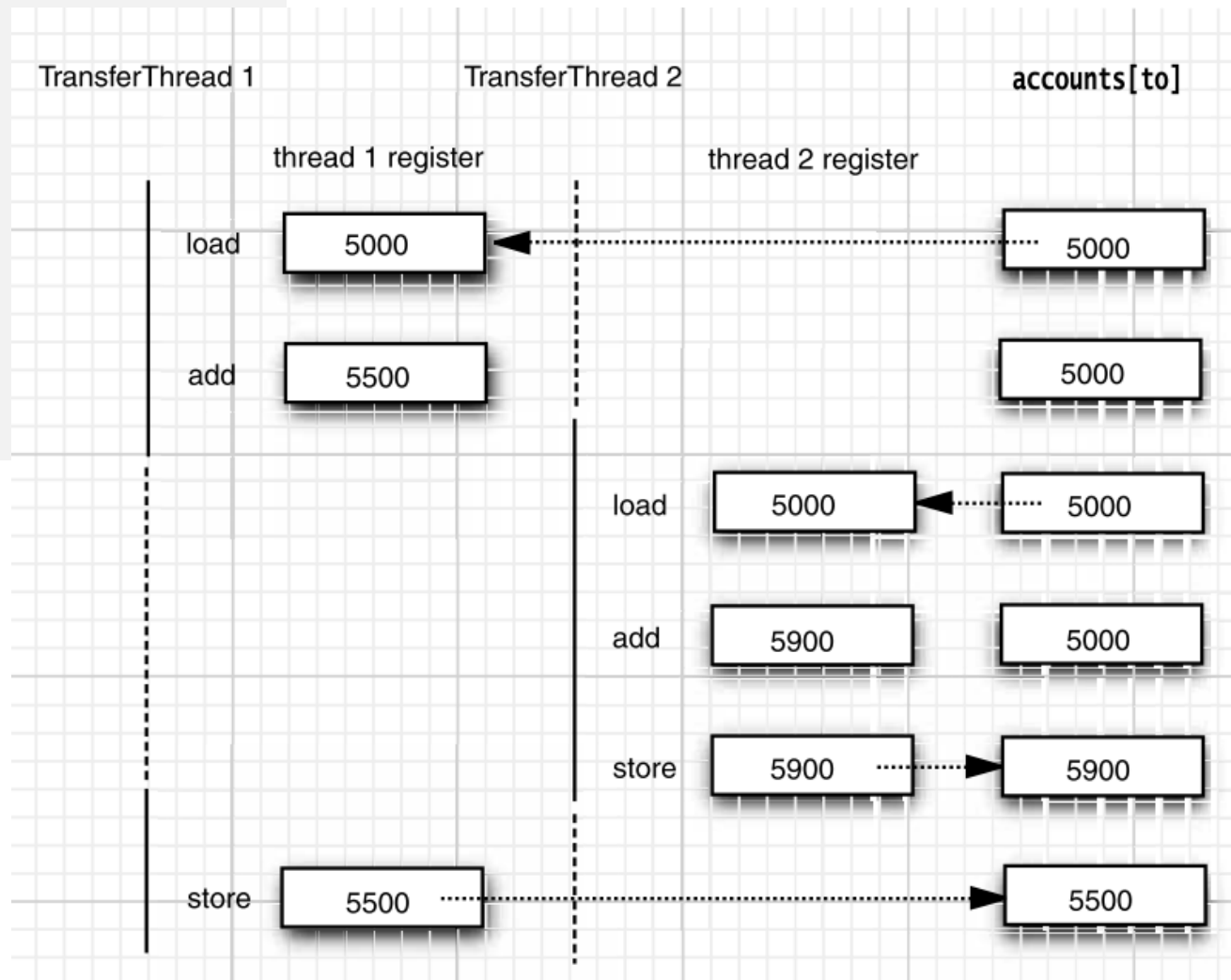
# Race Condition and Thread Safety

- Race Condition
  - Two or more threads access to the same object and each updates the state of the object that result in **corrupted objects**
- Thread Safety
  - A class is **thread-safe** if an object of the class does not cause a race condition in the presence of multiple threads

Output of the Unsynchronized Bank

```
. . .
Thread[Thread-42,5,main]    295.38 from 42 to 1  Total Balance: 100000.00
Thread[Thread-72,5,main]    151.17 from 72 to 15 Total Balance: 100000.00
Thread[Thread-84,5,main]    140.33 from 84 to 66 Total Balance: 100000.00
Thread[Thread-13,5,main]    543.10 from 13 to 36 Total Balance: 100000.00
. . .
Thread[Thread-22,5,main]    289.85 from 22 to 72 Total Balance: 99576.68
Thread[Thread-85,5,main]    37.27 from 85 to 38 Total Balance: 99576.68
Thread[Thread-11,5,main]    718.28 from 11 to 1  Total Balance: 99576.68
. . .
Thread[Thread-57,5,main]    332.41 from 57 to 46 Total Balance: 99323.76
Thread[Thread-70,5,main]    253.12 from 70 to 67 Total Balance: 99323.76
```

# Race Condition Explained



# Synchronization

very useful, update variables in same function in different thread

- Two kinds of errors

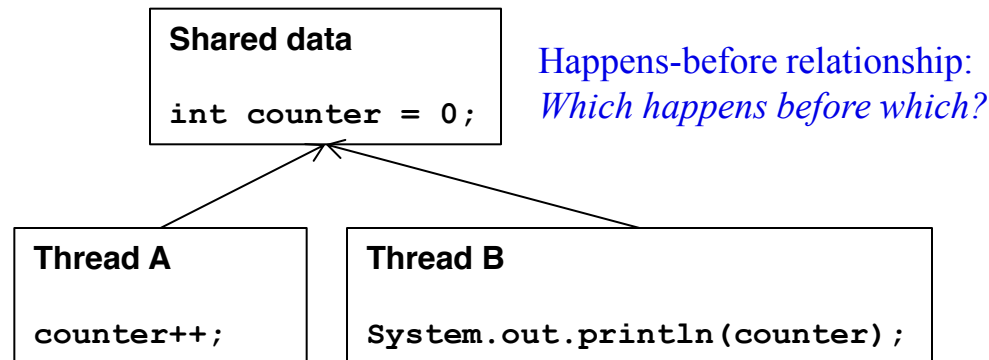
- Thread interference

- Errors introduced when multiple threads access shared data
    - Operation *Interleave*:

Interference happens when two operations, running in different threads, but acting on the same data, interleave. This means that the two operations consist of *multiple steps*, and the sequences of steps overlap.

- Memory consistency errors

- Errors that result from inconsistent views of shared memory
    - Memory consistency errors occur when different threads have inconsistent views of what should be the same data



# synchronized Keyword

- To avoid race conditions, Java uses the keyword **synchronized** to prevent more than one thread from simultaneously entering a certain part of the program, known as the *critical section* or *critical region*
  - All access to delicate data should be synchronized
  - Delicate data protected by synchronized should be private

- **Synchronized method:**

```
public synchronized void xMethod() {  
    // method body  
}
```

- **Synchronized block (statements):**

```
synchronized (expr) {    // lock the object  
    // statements;  
}
```

where **expr** must evaluate to an object reference

# Example: Why Synchronized Block

---

```
public void addName(String name) {  
    synchronized(this) {  
        lastName = name;  
        nameCount++;  
    }  
    nameList.add(name);  
}
```

- The `addName()` method needs to synchronize changes to `lastName` and `nameCount`, but also needs *to avoid synchronizing invocations of other objects' methods* (`nameList.add()`)
- Invoking other objects' methods from synchronized code can create problems
- Synchronized statements are also useful for *improving concurrency with fine-grained synchronization* to avoid unnecessary blocking

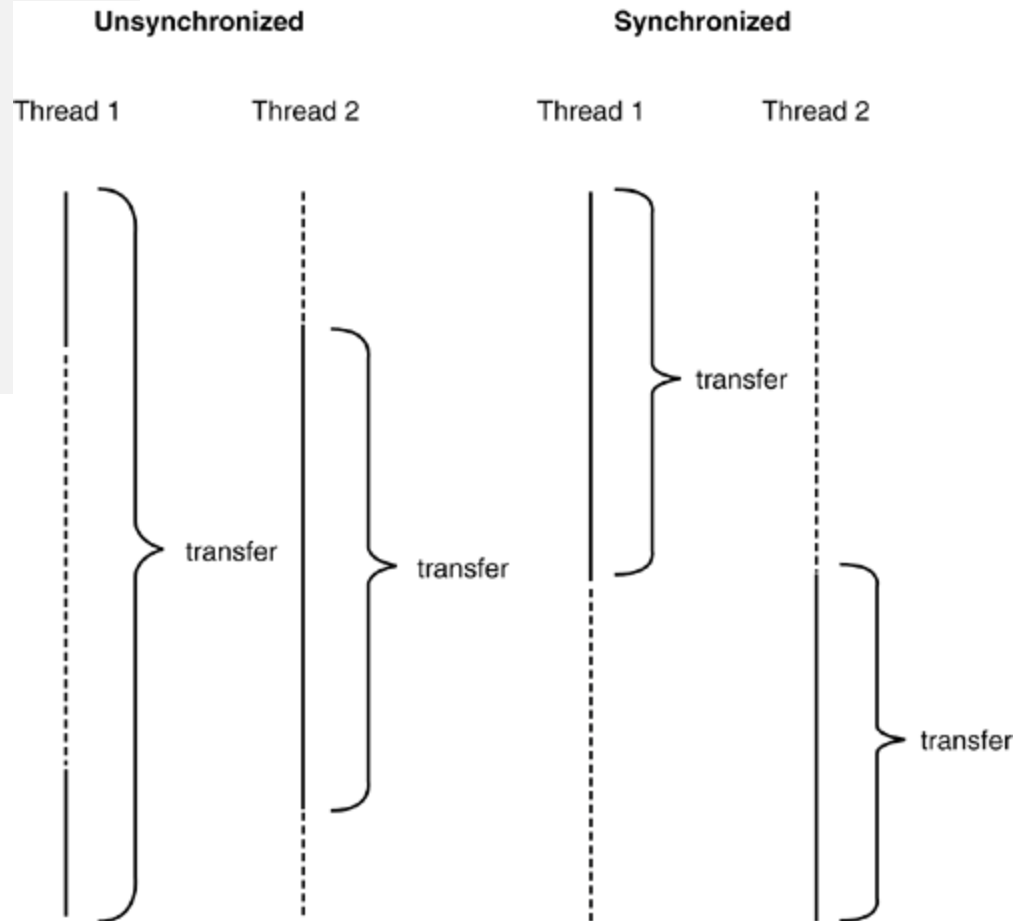
# Bank Example: Synchronized

```
public synchronized void transfer(int from, int to, int amount)
    throws InterruptedException
{
    if (accounts[from] < amount) return;
    System.out.print(Thread.currentThread());
    accounts[from] -= amount;
    System.out.printf(" %10.2f from %d to %d", amount, from, to);
    accounts[to] += amount;
    ntransacts++;
    System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
}
```

## Output of the Synchronized Bank

```
. . .
Thread[Thread-36,5,main]    732.12 from 36 to 17 Total Balance: 100000.00
Thread[Thread-4,5,main]    120.50 from 4 to 29 Total Balance: 100000.00
Thread[Thread-12,5,main]   725.23 from 12 to 62 Total Balance: 100000.00
Thread[Thread-59,5,main]   135.03 from 59 to 95 Total Balance: 100000.00
Thread[Thread-22,5,main]   822.45 from 22 to 70 Total Balance: 100000.00
Thread[Thread-1,5,main]    28.52 from 1 to 14 Total Balance: 100000.00
Thread[Thread-16,5,main]   254.19 from 16 to 58 Total Balance: 100000.00
Thread[Thread-30,5,main]   585.52 from 30 to 66 Total Balance: 100000.00
. . .
```

# Unsyncronized and Synchronized Threads





# Example: Safe Flying Label

## Problem: Prevent race conditions

```

JButton btnStart = new JButton("Start");
btnStart.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        stop=false;
        Thread t = new Thread(){ // Create a thread
            public void run() {
                fly();
            }
        };
        t.start();
    }
});
JButton btnStop = new JButton("Stop");
btnStop.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        stop=true;
    }
});
public void fly(){ Synchronizing the method
    for (int i = 0; i<FLYTIME ;i++){
        if(!stop){ // fly only when not stopped
            if(x > getWidth()){
                x = -150;
            }
            x += 1;
            label.setBounds(x,166,160,30);
            // relocating label
        }
        Thread.sleep(10); // slow it down
    }
}
```

```

JButton btnStart = new JButton("Start");
btnStart.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        stop=false;
        Thread t = new Thread(){ // Create a thread
            public void run() {
                fly();
            }
        };
        t.start();
    }
});
JButton btnStop = new JButton("Stop");
btnStop.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        stop=true;
    }
});
public synchronized void fly(){
    for (int i = 0; i<FLYTIME ;i++){
        if(!stop){ // fly only when not stopped
            if(x > getWidth()){
                x = -150;
            }
            x += 1;
            label.setBounds(x,166,160,30);
            // relocating label
        }
        Thread.sleep(10); // slow it down and yield the control
    }
}
```

# Intrinsic Locks and Synchronization

---

- Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock* or *monitor*
- Intrinsic locks play a role in both aspects of synchronization:
  - *Enforcing exclusive access* to an object's state
  - *Establishing happens-before relationships* that are essential to visibility
- Every object has an intrinsic lock associated with it
  - A thread that needs exclusive and consistent access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it's done with them
  - A thread is said to *own* the intrinsic lock between the time it has acquired the lock and released the lock
  - As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock
- When a thread invokes a synchronized method or executes a synchronized block, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns
- Calling a static method locks the class object

# Liveness

---

- Liveness:
  - A concurrent application's ability to execute in a timely manner
- *Deadlock*
  - Two or more threads are blocked forever, waiting for each other
- *Starvation*
  - A thread is unable to gain regular access to shared resources and is unable to make progress
  - This happens when shared resources are made unavailable for long periods by "greedy" threads
- *Livelock*
  - A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result.
  - As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work

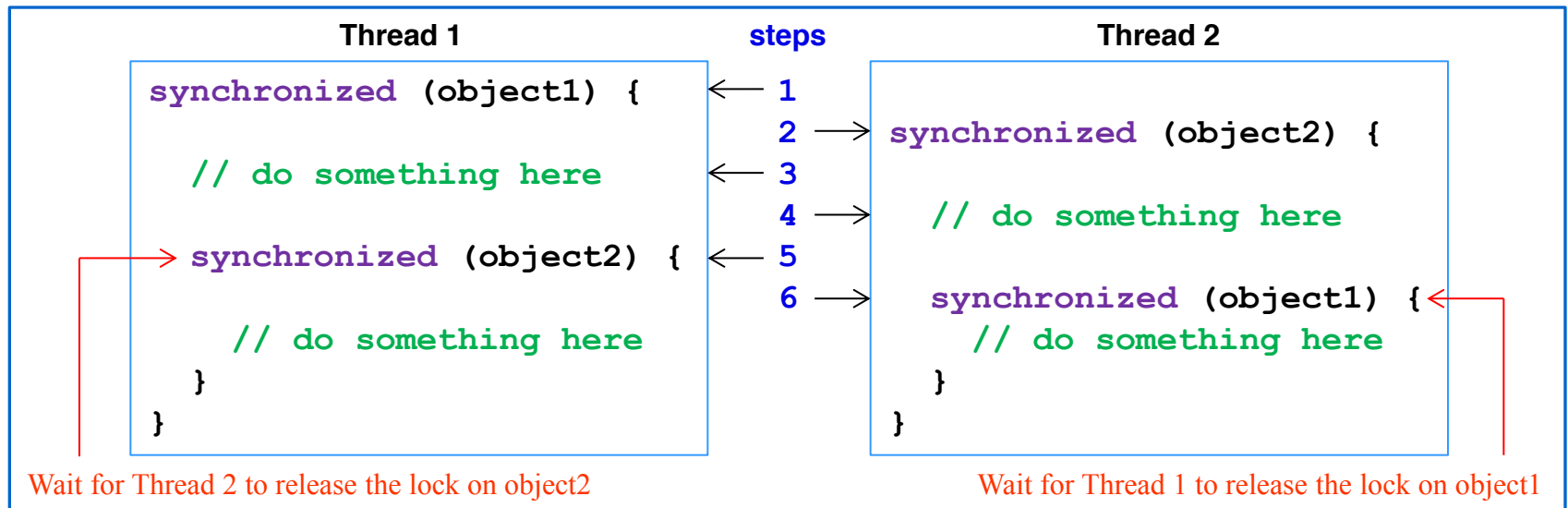
# Liveness Failures

---

- Contention
  - A runnable thread never gets a chance to run
  - Solution: periodically invoke `sleep()` or `yield()`
- Dormancy
  - A thread that is blocked never becomes runnable
  - Solution: be sure that each thread can be blocked by the `wait()` method will be awakened by another thread that invokes the `notify()` or `notifyAll()` method
- Deadlock
  - Two or more threads block each other and none can make progress
  - Solution: make a global decision about the order in which the locks will be obtained and adhere to that order throughout the program. Release the locks in the reverse order from which they were obtained
- Premature Termination
  - A thread is terminated before it should be, impeding the progress of other threads

# Deadlock

- This occurs when one thread is waiting for a lock held by another thread, but the other thread is waiting for a lock already held by the first thread
- A general rule of thumb for avoiding a deadlock is as follows
  - If you have multiple objects to which you want synchronized access to, make a global decision about the order in which the locks will be obtained and adhere to that order throughout the program. Release the locks in the reverse order from which they were obtained



# How to Avoid Deadlock

---

- Resource ordering
  - Assign an order to all the objects whose locks must be acquired and ensure that each thread acquires the locks in that order

For the example in the diagram on the previous slide:

Suppose that the objects are ordered as **object1** and **object2**. Acquire a lock on **object1** first, then on **object2**.

Once **Thread 1** acquires a lock on **object1**, **Thread 2** has to wait for a lock on **object1**. So **Thread 1** will be able to acquire a lock on **object2** and no deadlock will occur

# High Level Concurrency Objects

---

- The low-level APIs from `java.lang.Thread` packages are adequate for very basic tasks
- Higher-level building blocks are needed for more advanced tasks
  - This is especially true for massively concurrent applications that fully exploit today's multiprocessor and multi-core systems
- High-level concurrency features introduced with version 5.0 are implemented in the new `java.util.concurrent` packages

# Lock Objects

- **Lock** objects work very much like the implicit locks used by synchronized code
- **Lock** objects also support a wait/notify mechanism, through their associated **Condition** objects
- A synchronized instance method implicitly acquires a lock on the instance before it executes the method
- **Lock** objects enable explicit use of locks and more control for coordinating threads

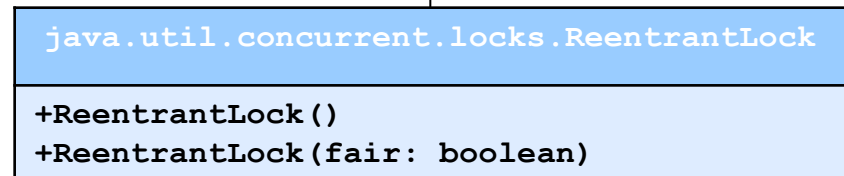
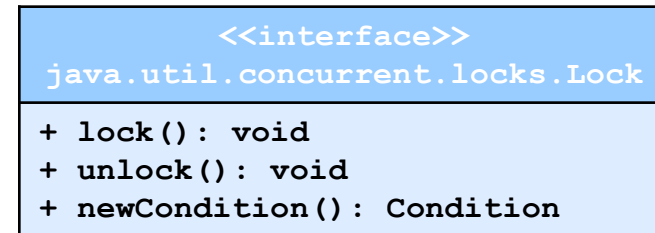
Basic routine:

```
Lock myLock = new ReentrantLock();

myLock.lock(); // a ReentrantLock
try{

    // critical region

} finally {
    myLock.unlock();
}
```





# Condition Objects

- **Condition** object (or also called *condition variable*) is used to manage threads that have acquired a lock but cannot do useful work
  - A thread often enters a critical section, only to discover that it can't proceed until a condition is fulfilled
- A lock object can have one or more associated condition objects
- **await()** deactivates thread and gives up the lock
  - unlike **wait()**
- In general, a call to **await()** should be inside a loop of the form

```
while (!(ok to proceed))  
    condition.await();
```

```
java.util.concurrent.locks.Condition
```

```
+void await()  
+void signalAll()  
+void signal()
```

# Example: Condition Objects

```
private Lock bankLock;  
private Condition sufficientFunds;  
  
. . .  
bankLock = new ReentrantLock();  
sufficientFunds = bankLock.newCondition();  
. . .  
  
public void transfer(int from, int to, double amount) throws InterruptedException  
{  
    bankLock.lock();  
    try {  
        while (accounts[from] < amount)  
            sufficientFunds.await();  
        System.out.print(Thread.currentThread());  
        accounts[from] -= amount;  
        System.out.printf(. . .);  
        accounts[to] += amount;  
        System.out.printf(. . .);  
        sufficientFunds.signalAll();  
    } finally {  
        bankLock.unlock();  
    }  
}
```

# The suspend, resume and Stop Methods

---

- Deprecated since 1.2
- Should be replaced with `wait()` and `notify()` or `Lock` and `Condition`
  - `Suspend()` is deadlock-prone (`suspend()` does not release the held lock)
  - Allows on a thread to have direct control over another thread's code execution
  - `Stop()` releases the lock but can leave shared data in an inconsistent state

**Do not use deprecated methods**

There are good reasons for being deprecated

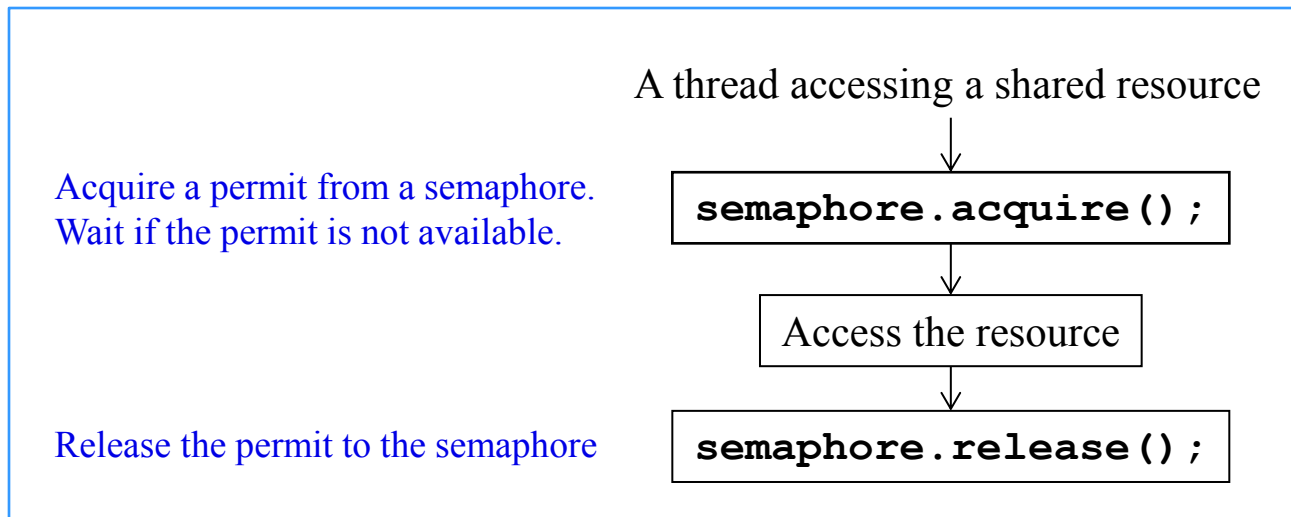
# Synchronizers

- Synchronizers help manage a set of collaborating threads
  - If you have a set of collaborating threads that follows one of these behavior patterns, you should simply reuse the appropriate library class instead of trying to come up with a handcrafted collection of locks and conditions

Class	What it does	When to use
<b>CyclicBarrier</b>	Allows a set of threads to wait until a redefined count of them has reached a common barrier, and then optionally executes a barrier action.	When a number of threads need to complete before their results can be used.
<b>CountDownLatch</b>	Allows a set of threads to wait until a count has been decremented to 0.	When one or more threads need to wait until a specified number of events have occurred.
<b>Exchanger</b>	Allows two threads to exchange objects when both are ready for the exchange.	When two threads work on two instances of the same data structure, one by filling an instance and the other by emptying the other.
<b>Semaphore</b>	Allows a set of threads to wait until permits are available for proceeding.	To restrict the total number of threads that can access a resource. If permit count is one, use to block threads until another thread gives permission.
<b>SynchronousQueue</b>	Allows a thread to hand off an object to another thread.	To send an object from one thread to another when both are ready, without explicit synchronization.

# Semaphores

- Semaphores can be used to restrict the number of threads that access a shared resource.
  - Before accessing the resource, a thread must acquire a permit from the semaphore
  - After finishing with the resource, the thread must return the permit back to the semaphore



# Executors

- Separate thread management and creation from the rest of the application in large-scale applications
  - So far, the task (**Runnable** object) being done by a new thread is closely connected with the thread itself (**Thread** object)
  - It is expensive to construct a new thread because it interacts with OS

- **Executor** interface

```
Runnable r = new Runnable();
```

```
Executor e = ...;      replacing →      Thread t = new Thread(r);  
e.execute(r);          t.start();
```

**execute()** may create a new thread or more likely to use an existing worker thread to run **r** or place **r** in a queue to wait for a worker thread to become available

- **ExecutorService** interface
  - **submit()** accepts **Runnable** and **Callable** objects
- **ScheduledExecutorService** interface
  - **schedule()** executes **Runnable** and **Callable** objects after a specified delay

# Thread Pools

---

- Using worker threads minimizes the overhead due to thread creation
  - Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead
- Using a thread pool to throttle the number of concurrent threads
- Create thread pools
  - `newFixedThreadPool()` factory method to create a fixed thread pool
  - `newCachedThreadPool()` method creates an executor with an expandable thread pool
  - `newSingleThreadExecutor()` method creates an executor that executes a single task at a time

# Fork/Join

---

- The fork/join framework is an implementation of the **ExecutorService** interface that helps you take advantage of multiple processors
- Since 1.7
- The fork/join framework distributes tasks to worker threads in a thread pool
  - The fork/join framework is distinct because it uses a *work-stealing* algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy
- To use fork/join framework
  - rework the code to split the work into recursive tasks and wrap it as a **RecursiveTask** or **RecursiveAction** classes that are subclasses of **ForkJoinTask**, then pass it to **invoke()** method of the **ForkJoinPool** instance



# Concurrent Collections

---

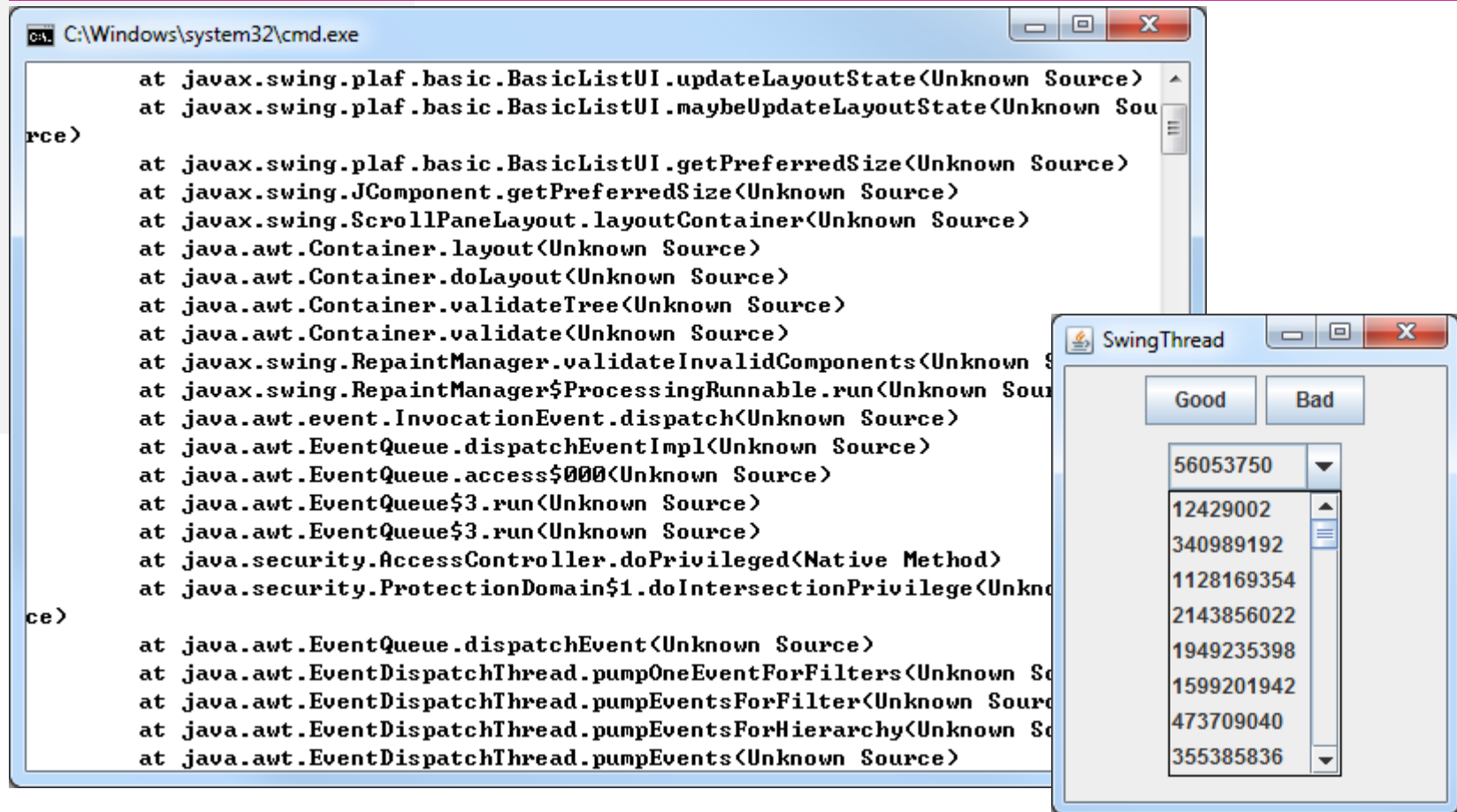
- Easier to use a thread-safe implementation than supplying a lock to protect shared data
  - All of these collections help avoid Memory Consistency Errors by defining a happens-before relationship between an operation that adds an object to the collection with subsequent operations that access or remove that object.
- Concurrent collections:
  - **BlockingQueue** defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue
  - **ConcurrentMap** is a subinterface of **Map** that defines useful atomic operations. Making these operations atomic helps avoid synchronization.
    - The standard general-purpose implementation of **ConcurrentMap** is **ConcurrentHashMap**, which is a concurrent analog of **HashMap**.
  - **ConcurrentNavigableMap** is a subinterface of **ConcurrentMap** that supports approximate matches
    - The standard general-purpose implementation of **ConcurrentNavigableMap** is **ConcurrentSkipListMap**, which is a concurrent analog of **TreeMap**.
- Thread-safe collections: **Vector**, **Stack**, **Hashtable**

# Threads and Swing

---

- A Swing application runs on multiple threads
  - It has three types of threads:
    - An initial thread, or the Main thread, which runs the `main()` method, starts the building of GUI, and exits.
    - An **Event-Dispatching Thread** (EDT)
      - All the event-handling, painting and screen refreshing codes runs in a single EDT
    - Some Background Worker threads for compute-intensive task and IO.
- Swing is *not thread-safe*
  - Why not a thread-safe Swing?
    - Difficult and inefficient implementation
    - Difficult extension
    - Risk of deadlock
    - Minimal benefit
- The majority of methods of Swing classes are not synchronized
- If you try to manipulate user interface elements from multiple threads, your user interface may become corrupted

# Example: Corrupted GUI



**Bad button:** not synchronized and both the worker thread and EDT access combo box

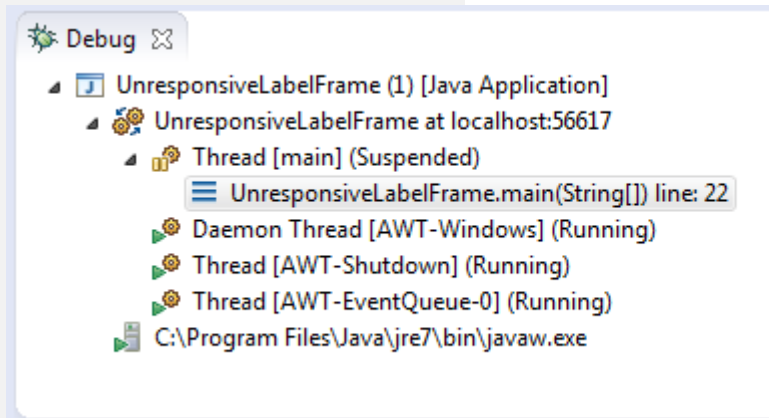
**Good button:** combo box updating task is forwarded to EDT

# Example: Corrupted GUI

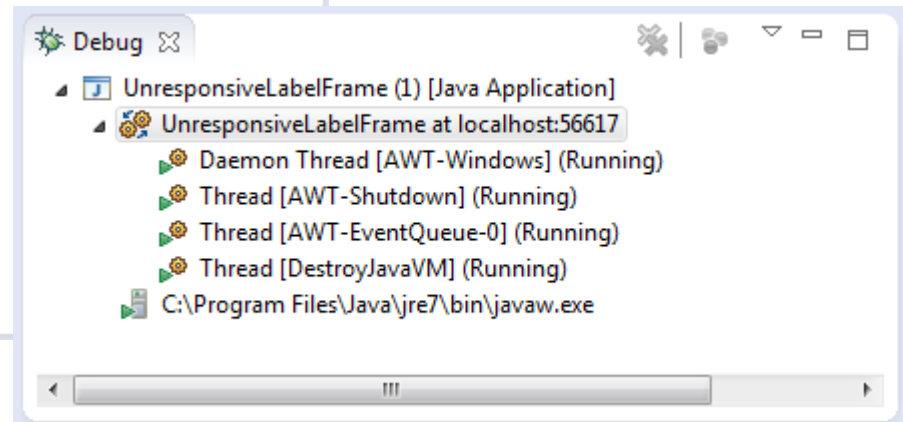
```
public void run() { //Bad worker thread
    try {
        while (!interrupted()) {
            int i = Math.abs(generator.nextInt());
            if (i % 2 == 0)
                combo.insertItemAt(new Integer(i), 0);
            else if (combo.getItemCount() > 0)
                combo.removeItemAt(i % combo.getItemCount());
            sleep(1);
        }
    } catch (InterruptedException exception) {}
}
```

```
public void run() { //Good worker Thread
    try {
        while (!interrupted()) {
            EventQueue.invokeLater(new Runnable(){
                public void run() {
                    int i = Math.abs(generator.nextInt());
                    if (i % 2 == 0)
                        combo.insertItemAt(new Integer(i), 0);
                    else if (combo.getItemCount() > 0)
                        combo.removeItemAt(i % combo.getItemCount());
                }
            });
            Thread.sleep(1);
        }
    } catch (InterruptedException exception) {}
}
```

# Thread Trace: Unresponsive Flying Label

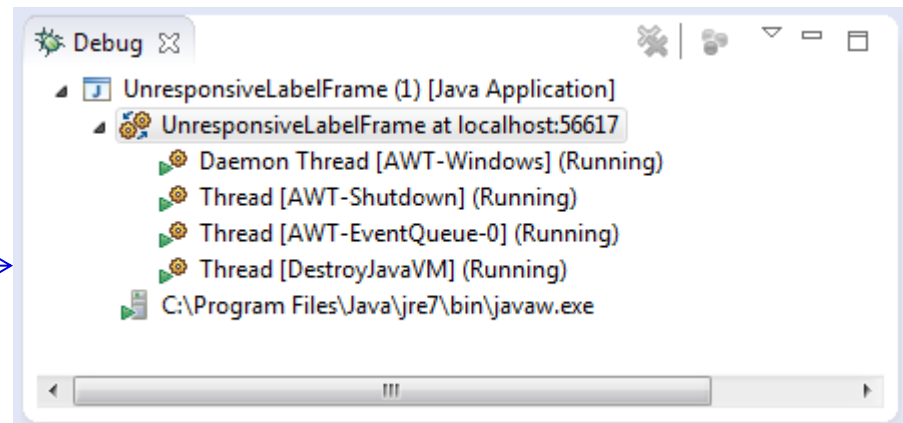


Before Start button pressed  
*before main() exits*



Before Start button pressed  
*after main() exits*

After Start button pressed



# Thread Trace: Observations

- The `main()` method starts in the "main" thread.
  - A new thread "AWT-Windows" (Daemon thread) is started when the constructor "`new UnresponsiveLabel()`" executes (because of the "`extends JFrame`").
  - After executing "`setVisible(true)`", another two threads are created - "AWT-Shutdown" and "AWT-EventQueue-0" (i.e., the EDT).
- The "main" thread exits after the `main()` method completes.
- A new thread called "DestroyJavaVM" is created.

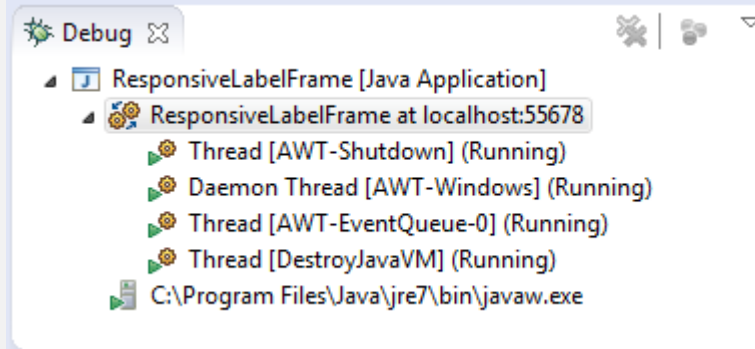
At this point, there are 4 threads running - "AWT-Windows", "AWT-Shutdown" and "AWT-EventQueue-0" and "DestroyJavaVM".

- Clicking the START button invokes the `actionPerformed()` in the EDT.

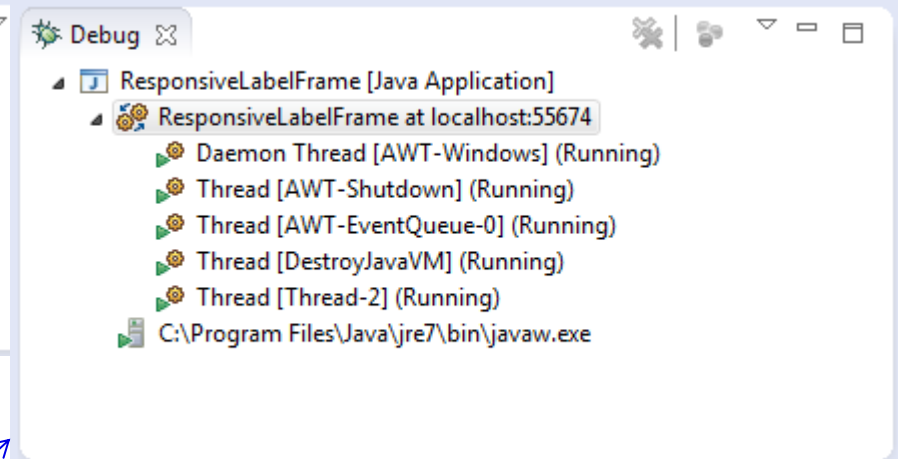
The EDT is now fully-occupied with the compute-intensive fly-loop. In other words, while the flying is taking place, the EDT is *busy* and unable to process any event (eg: clicking the STOP button or the window-close button) and refresh the display - until the flying completes and EDT becomes available. As the result, the display freezes until the fly-loop completes.

The flying is not visible because the screen display is not timely refreshed.

# Thread Trace: Responsive Flying Label

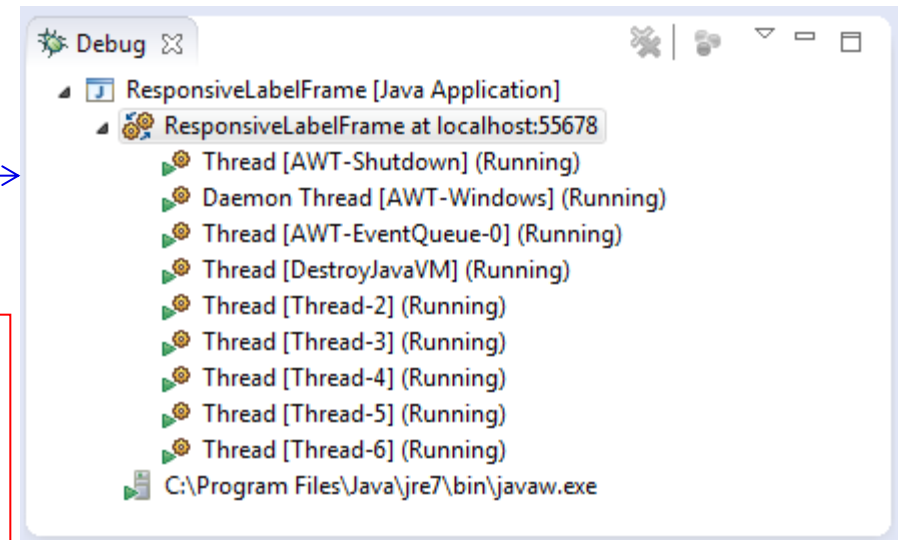


Before Start button pressed  
*after main() exits*



After Start button pressed once

After Start button pressed 5 times



A new thread is created each time when the Start button is pressed.  
Synchronization is required to serialize the operations to prevent race conditions.

# GUI Responsiveness and Threads

---

- GUI Event Dispatch Thread (EDT)
  - This is to ensure that one event handler finishes execution before the next handler starts, and that painting is not interrupted by events
  - If the event-dispatching thread is starved by another compute-intensive task, the user interface "freezes", and the program becomes unresponsive to user interaction
- Two Swing programming rules
  - **Single Thread Rule:**
    - All codes accessing the GUI components should be run on the EDT only for thread safety
    - As many of these components are not guaranteed to be thread-safe, accessing them from the same thread avoids the multithreading issues.
  - Time-consuming and blocking-IO tasks should not be run on the EDT
    - *Any task that requires more than 30 to 100 milliseconds should not be run on the EDT*
    - Otherwise, users will sense a pause between their input and the UI response.



# SwingUtilities.invokeLater()

---

- The `invokeLater(Runnable)` and `invokeAndWait(Runnable)` methods schedule the `Runnable` task in the EDT
  - Use `SwingUtilities.invokeLater(Runnable)` to create the GUI components on the EDT, instead of using the main thread
    - To avoid threading issues between the main thread (which runs the `main()` method) and the EDT
    - Often implemented as an anonymous inner class
  - `invokeAndWait()`, which waits until the event-dispatching thread has executed the specified codes, and returns
    - For applets, it is recommended to run the GUI construction codes (in `init()`) via `invokeAndWait()`.
      - This is to avoid problems caused by `init()` exits before the completion of GUI construction
- The `javax.swing.SwingUtilities.invokeLater()` is just a cover for `java.awt.EventQueue.invokeLater()`
  - You can always use any one of them

# Template to Create Swing Program

---

```
private static void createAndShowGUI() {  
    //Create and set up the window  
    // JFrame frame = new JFrame();  
    // JFrame frame = new Subclass_of_Jframe();  
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    frame.setTitle(".....");  
    frame.setSize(WINDOW_WIDTH, WINDOW_HEIGHT);  
    //Display the window.  
    frame.pack();  
    frame.setVisible(true);  
}
```

```
public static void main(String[] args) {  
    //Schedule a job for the event-dispatching thread:  
    //creating and showing this application's GUI.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            createAndShowGUI();  
        }  
    });  
}
```

# Example: Flying High Label

**Problem: Control a single thread to animate the label**

*Worker thread for computation*

```
class FlyingEngine implements Runnable{

    private JLabel label;
    private int x;
    private Semaphore flyGate;
    private boolean flying;

    public FlyingEngine(JLabel label){
        this.label=label;
        this.flyGate = new Semaphore(1);
        this.flying = false;
    }

    public void setFlying(){//for Start button

        flying = true;
        flyGate.release();
    }

    public void setStop(){ //for Stop button

        flying = false;
    }
}
```

*Use a semaphore to sync the worker thread and EDT*

```
public void run(){
    x = label.getBounds().x;
    while(true){
        if(flying){
            if(x > W_W){
                x = -L_W;
            }
            x += 1;
            try {
                Single thread rule
                EventQueue.invokeLater(new Runnable(){
                    public void run(){
                        label.setBounds((int)x,H,L_W,L_H);
                    }
                });
                Thread.sleep(10);
            } catch (InterruptedException e) {}
        } else{
            try {
                flyGate.acquire();
            } catch (InterruptedException e) {}
        }
    }
}
```

*Continued*

# Using Threads in Swing Applications

---

- Fire up a new thread if
  - An action takes a long time
    - eg. SQL queries
    - Repetitive or timing operations in a loop
  - An action blocks on input or output
    - IO operations throw exceptions, downloading from Internet takes unpredictably long time
- Use a Timer, if
  - You need to wait for a specific amount of time, don't sleep in the event dispatch thread
- The work that you do in your threads should not process GUI
- Exceptions to the Single Thread Rule
  - Construct GUI to realize the components in `main()` or `init()` thread before the first call to `setVisible(true)`
  - Thread-safe Swing methods

# Thread-Safe Swing Methods

---

- Thread-safe Swing methods

- `JTextComponent.setText()`
- `TextArea.insert()`
- `TextArea.append()`
- `TextArea.replaceRange()`

- **JComponent** methods that can be called from any thread

They are intended to be called by the application program to trigger redisplay or recalculation of display by the EDT which actually do not perform any real work in the caller thread, but send a request to the EDT

- `repaint()`
- `revalidate()/invalidate()`

- Listener methods are always thread-safe

No guarantees about the exact temporal relationship between adding a listener and the first event that the listener receive

- `addXxxListener()`
- `removeXxxListener()`

# SwingWorker

---

- Typical UI activities of a background task:
  - After each work unit, update the UI to show progress
  - After the work is finished, make a final change to the UI
- **SwingWorker** class makes it easy to implement such a task
- Since 1.6
- How-to:
  - Override `doInBackground()` method
  - Call `publish()` method to communicate work progress