Programação Orientada a Objetos

Programação Concorrente
(Multithreading)

Programação Concorrente

Sumário

- **Introdução**
- **Estados de um Thread: Ciclo de vida de um Thread**
- **Prioridades de Thread Priorities and Thread Scheduling**
- **Creating and Executing Threads**
- **Thread Synchronization**
- **Producer/Consumer Relationship without Synchronization**
- **Synchronization and Deadlocks**
- **Producer/Consumer Relationship with Synchronization**
- **Daemon Threads**
- Runnable **Interface**
- **Callables**
- **ExecutorServices**

5.1 Introdução

- Java provê multithreading em sua biblioteca padrão
  - Multithreading melhora a performance de alguns programas
  - Permite que um programa continue respondendo a pedidos do usuário, enquanto faz atividades demoradas

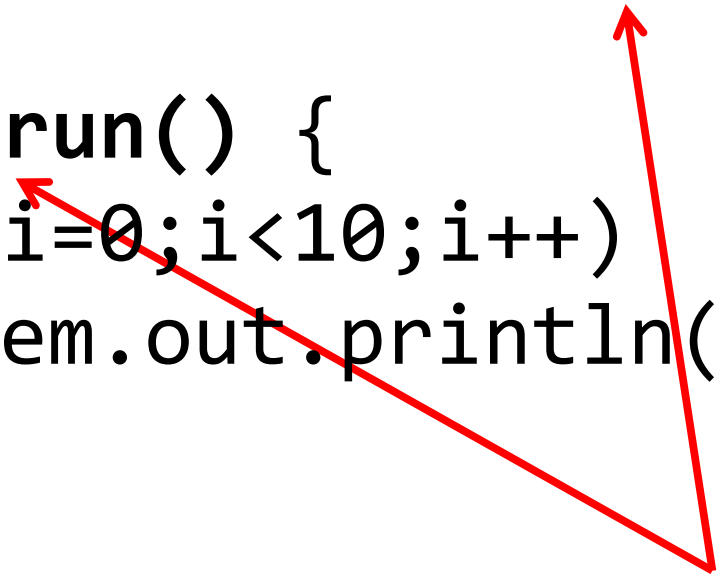
*A **Java Thread** is a lightweight process created by the Java Virtual Machine.*

The actual representation in the operating system depends on the implementation.

`Runnable` → *An interface which represents something that can be executed on a thread.*

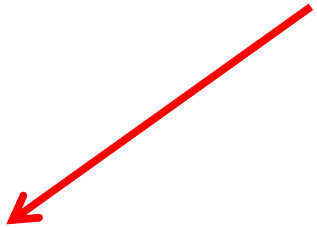
`Thread` → *A class which executes implementations of Runnable on a parallel process.*

```
public class CounterPrinter
                    extends Thread {

    public void run() {
        for(int i=0;i<10;i++)
            System.out.println(i);
    }

}
```

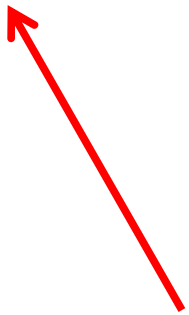


Implementations of Runnable have a method called run() with the logic to be executed on the thread

Create a Thread



```
CounterPrinter p =
    new CounterPrinter();
p.start();
```



Initiate the thread execution

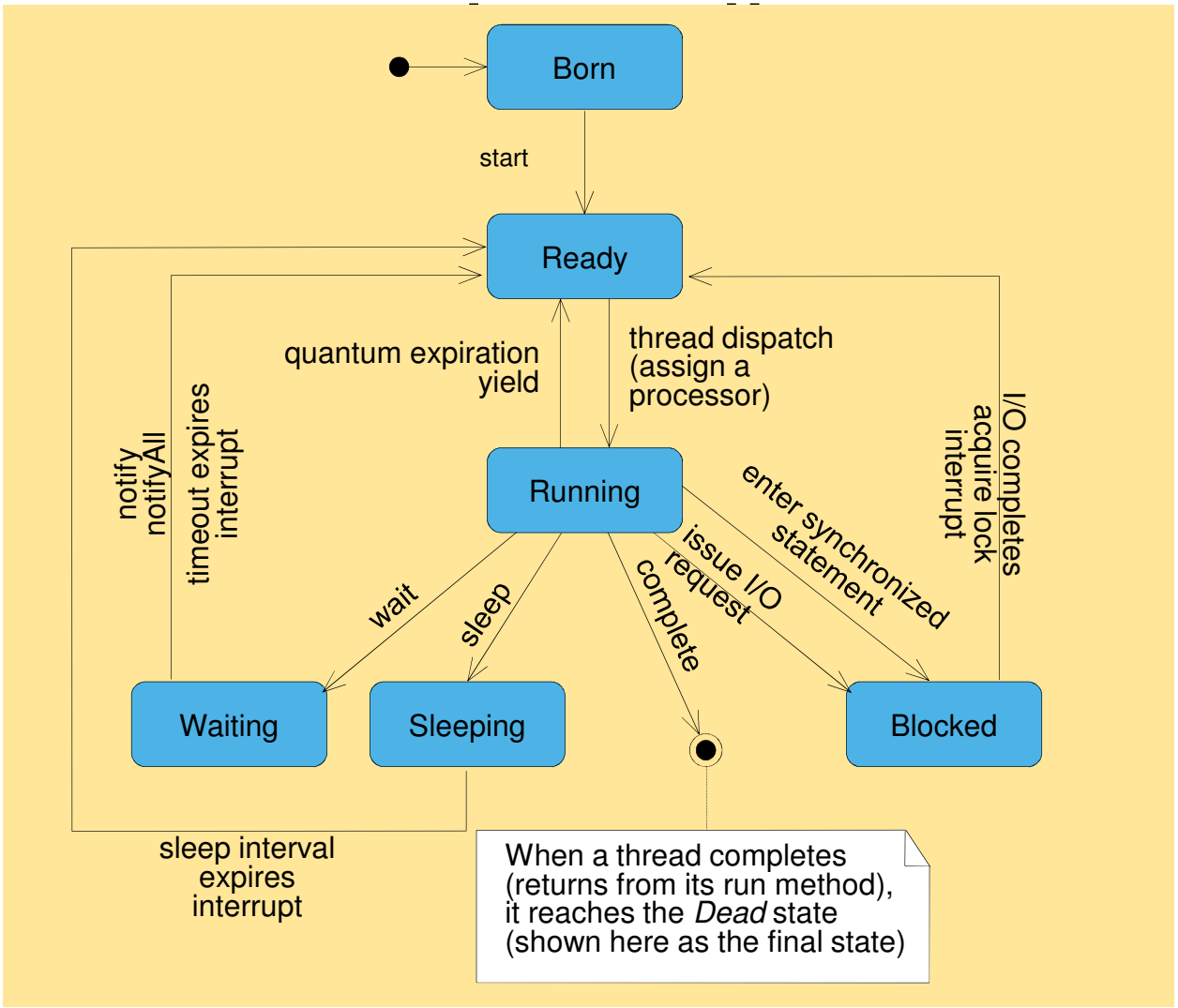
Exercício

- *Crie uma Thread que imprime uma sequencia de números de 1 a 10*
- *Crie um método main que cria e inicia 10 thread da classe*
- *Identifique cada Thread com um nome para saber qual thread está escrevendo. Use o construtor Thread(String) e métodos getName e setName de Thread*

Estados de um Thread: Ciclo de vida de um Thread

- Estados de Thread
  - Born state
    - Quando acabou de ser criado
  - Ready state
    - Método start de Thread é chamado
    - Thread está pronto para ser executado
  - Running state
    - Thread é alocado a um processador para execução
  - Dead state
    - Thread completou sua tarefa (run)
  - Waiting State
    - Espera até ser notificado
  - Sleeping State
    - Espera um tempo pré-determinado
  - Blocked State
    - Espera até que um determinado recurso I/O ou código sincronizado seja liberado

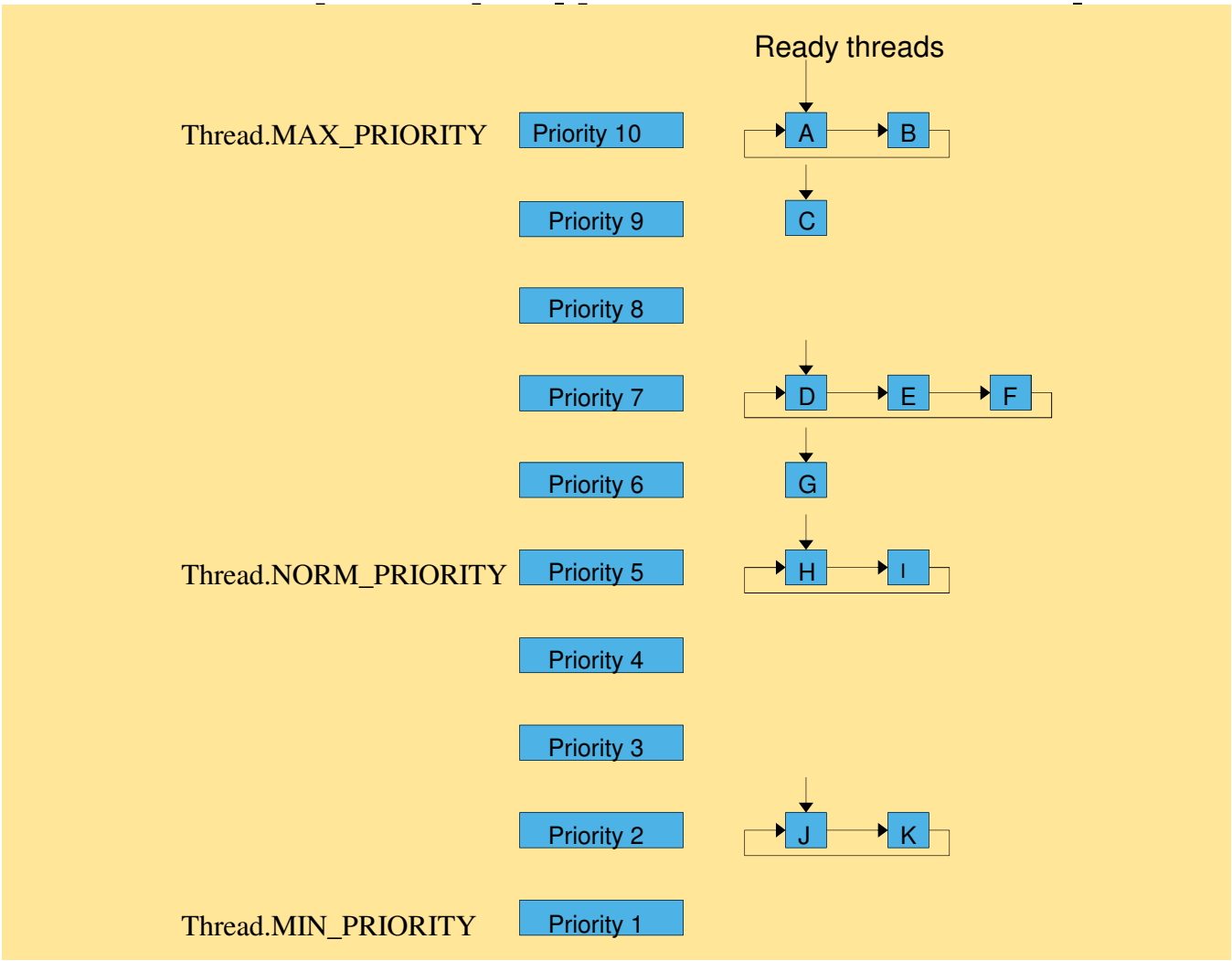
Fig. 5.1 Thread life-cycle



5.3 Thread Priorities and Thread Scheduling

- Java thread priority
  - Priority in range 1-10
- Timeslicing
  - Each thread assigned time on the processor (called a quantum)
  - Keeps highest priority threads running

Fig. 5.2 Thread priority



5.4 Creating and Executing Threads

- Sleep state
  - `Thread` method `sleep` called
  - Thread sleeps for a set time interval then awakens

```
1  // Fig. 5.3: ThreadTester.java
2  // Multiple threads printing at different intervals.
3
4  public class ThreadTester {
5
6     public static void main( String [] args )
7     {
8        // create and name each thread
9        PrintThread thread1 = new PrintThread( "thread1" );
10       PrintThread thread2 = new PrintThread( "thread2" );
11       PrintThread thread3 = new PrintThread( "thread3" );
12
13       System.err.println( "Starting threads" );
14
15       thread1.start(); // start thread1 and place it in ready state
16       thread2.start(); // start thread2 and place it in ready state
17       thread3.start(); // start thread3 and place it in ready state
18
19       System.err.println( "Threads started, main ends\n" );
20
21    } // end main
22
23 } // end class ThreadTester
24
```

create four PrintThreads

call start methods

```
25 // class PrintThread controls thread execution
26 class PrintThread extends Thread {
27    private int sleepTime;
28
29    // assign name to thread by calling superclass constructor
30    public PrintThread( String name )
31    {
32       super( name );
33
34       // pick random sleep time between 0 and 5 seconds
35       sleepTime = ( int ) ( Math.random() * 5001 );
36    }
37
38    // method run is the code to be executed by new thread
39    public void run()
40    {
41       // put thread to sleep for sleepTime amount of time
42       try {
43          System.err.println(
44             getName() + " going to sleep for " + sleepTime );
45
46          Thread.sleep( sleepTime );
47       }
48
```

PrintThread extends Thread

Constructor initializes sleepTime

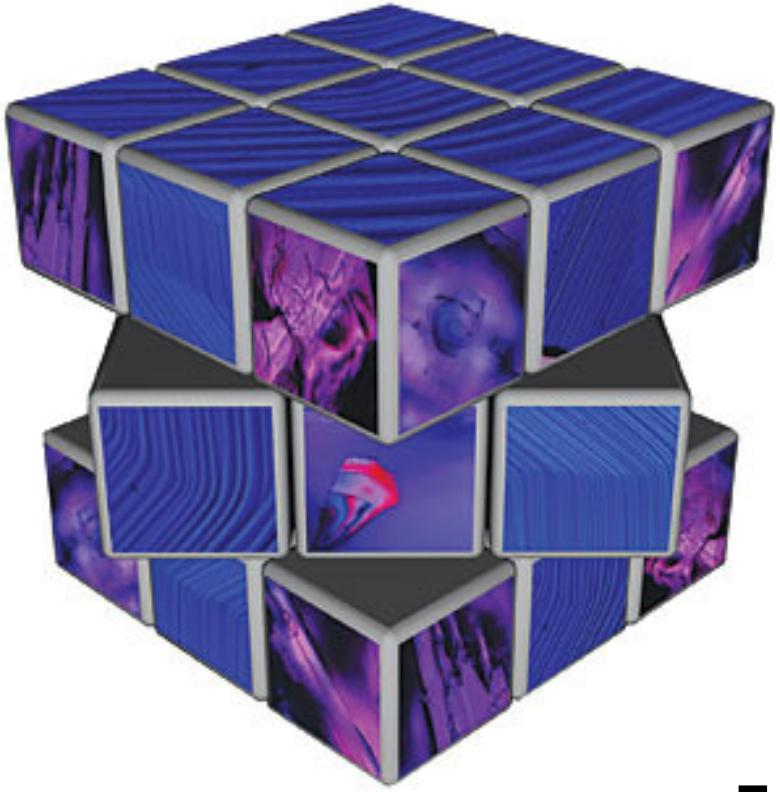
When the thread enters the running state, run is called

```
49          // if thread interrupted during sleep, print stack trace
50          catch ( InterruptedException exception ) {
51             exception.printStackTrace();
52          }
53
54          // print thread name
55          System.err.println( getName() + " done sleeping" );
56
57       } // end method run
58
59    } // end class PrintThread
```

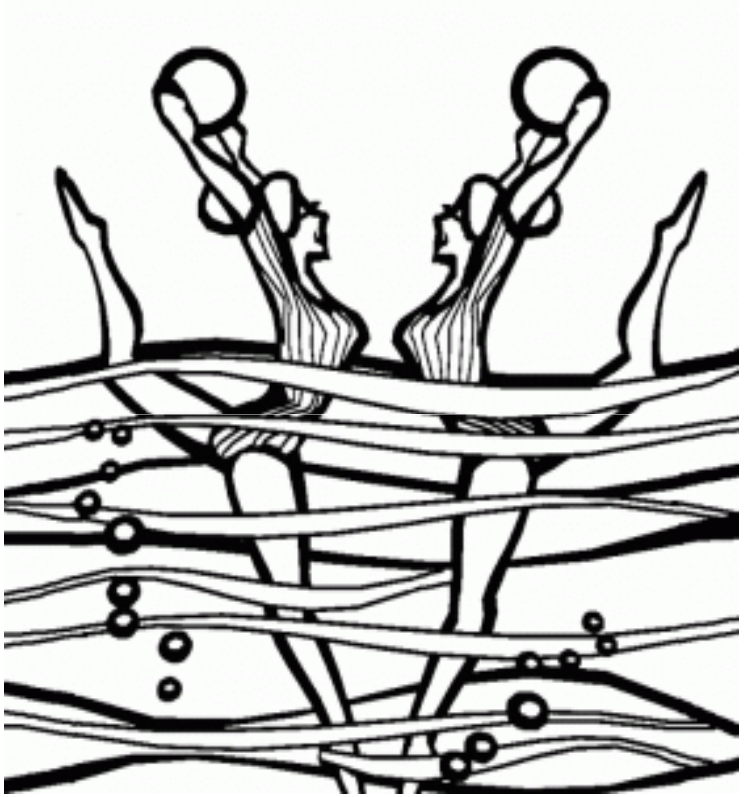
Starting threads
Threads started, main ends

thread1 going to sleep for 1217
thread2 going to sleep for 3989
thread3 going to sleep for 662
thread3 done sleeping
thread1 done sleeping
thread2 done sleeping

```
Starting threads
thread1 going to sleep for 314
thread2 going to sleep for 1990
Threads started, main ends

thread3 going to sleep for 3016
thread1 done sleeping
thread2 done sleeping
thread3 done sleeping
```

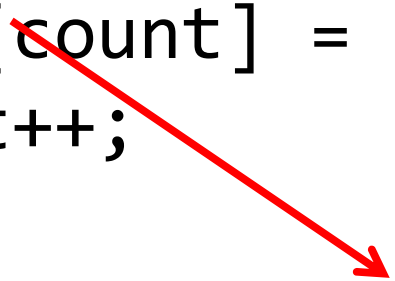


When you use threads, you can't guarantee the order of execution!



When more than one thread need to work together, they need to synchronize to access shared resources.

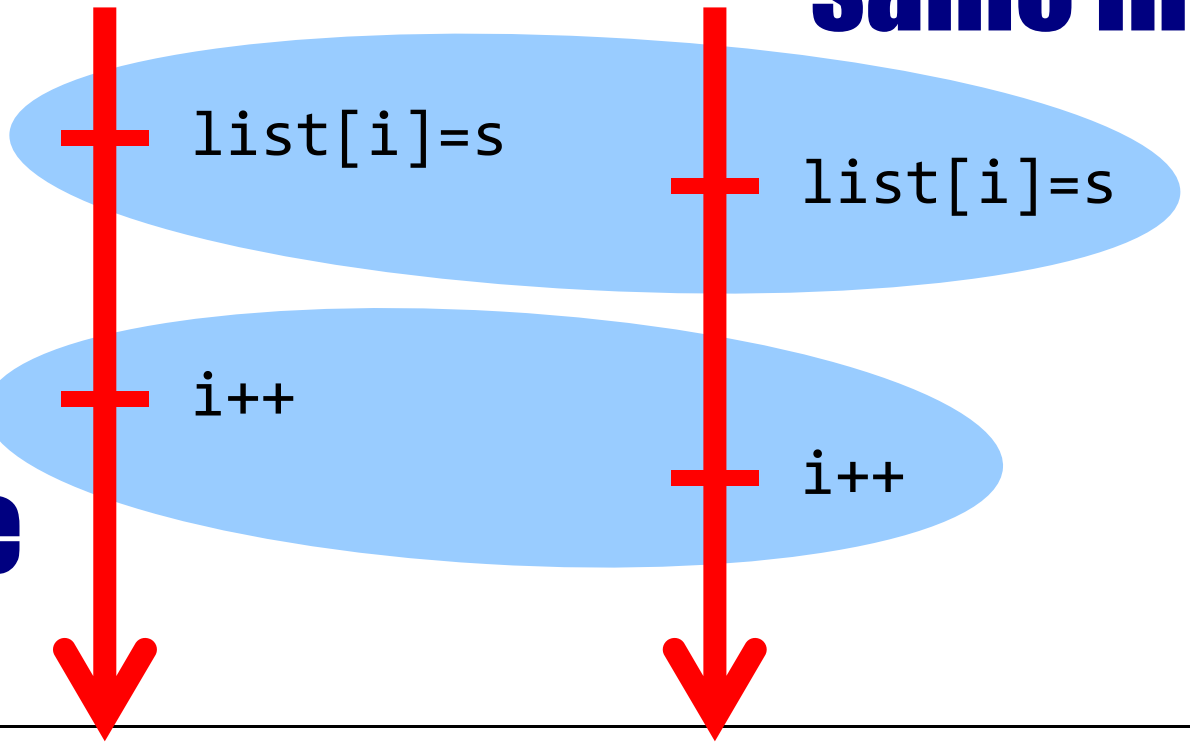
```
public class Storage {
     private String[] list = new
String[100];
     private int count = 0;

     public void add(String str){
          list[count] = str;
          count++;
     }
}
```



What happen when 10 threads try to add 10 strings on the same instance of Storage?

```
list[52] = Thread3-5
list[53] = null
list[54] = Thread6-9
```

Put on the same index!

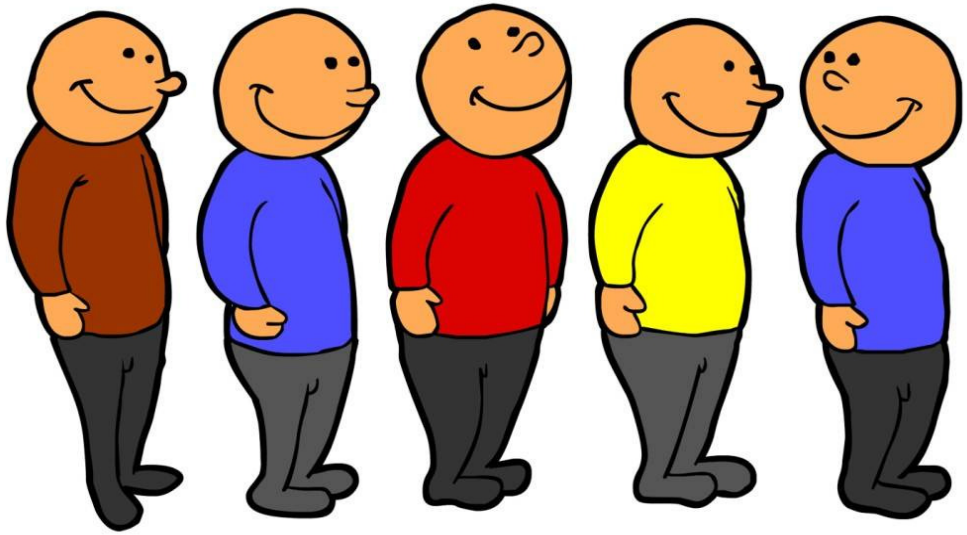


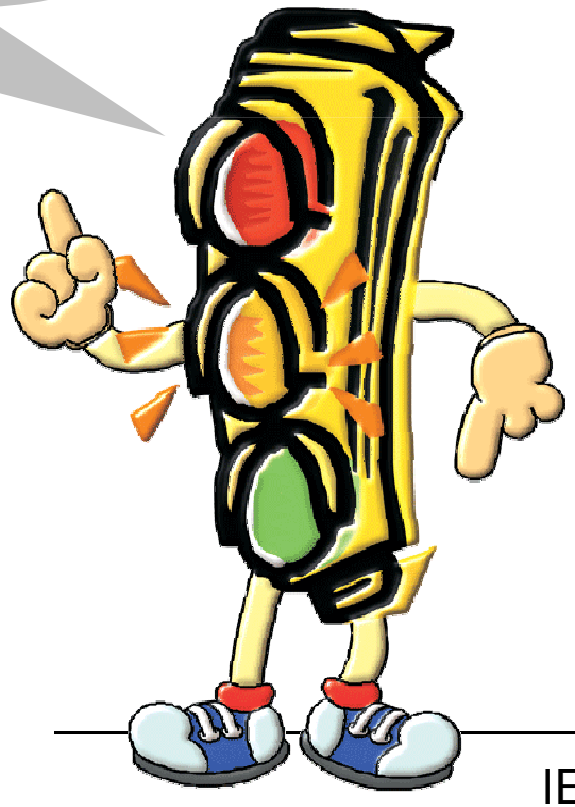
Jump one position!

```
public class Storage {
     private String[] list = new
String[100];
     private int count = 0;

     public void add(String str){
          list[count] = str;
          count++;
     }
}
```

This method execution should be atomic!

You can only enter in this method when the other Thread finishes!





5.6 Producer/Consumer Relationship without Synchronization

- Buffer
  - Shared memory region
- Producer thread
  - Generates data to add to buffer
  - Calls `wait` if consumer has not read previous message in buffer
  - Writes to empty buffer and calls `notify` for consumer
- Consumer thread
  - Reads data from buffer
  - Calls `wait` if buffer empty
- Synchronize threads to avoid corrupted data

Exemplo de Saída: SharedBuffer

```
Consumer reads -1
Producer writes 1
Consumer reads 1
Consumer reads 1
Consumer reads 1
Consumer read values totaling: 2.
Terminating Consumer.
Producer writes 2
Producer writes 3
Producer writes 4
Producer done producing.
Terminating Producer.
```

Producer writes 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer reads 4
Consumer read values totaling: 13.
Terminating Consumer.

```
Producer writes 1
Consumer reads 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer read values totaling: 10.
Terminating Consumer.
```

```java
// Buffer.java
// Buffer interface specifies methods called by Producer and Consumer.

public interface Buffer {
   public void set( int value );  // place value into Buffer
   public int get();              // return value from Buffer
}
```

```
1  // Fig. 5.5: Producer.java
2  // Producer's run method controls a thread that
3  // stores values from 1 to 4 in sharedLocation.
4
5  public class Producer extends Thread {
6     private Buffer sharedLocation; // reference to shared object
7
8     // constructor
9     public Producer( Buffer shared )
10    {
11        super( "Producer" );
12        sharedLocation = shared;
13    }
14
15    // store values from 1 to 4 in sharedLocation
16    public void run()
17    {
18       for ( int count = 1; count <= 4; count++ ) {
19          int sum=0;
20          // sleep 0 to 3 seconds, then place value in Buffer
21          try {
22             Thread.sleep( ( int ) ( Math.random() * 3001 ) ); sum+=co
23             sharedLocation.set( count );
24          }
25
```

Producer extends Thread

This is a shared object

Method run is overridden

The thread goes to sleep, then the buffer is set

```
26            // if sleeping thread interrupted, print stack trace
27            catch ( InterruptedException exception ) {
28               exception.printStackTrace();
29            }
30
31         } // end for
32
33         System.err.println( getName() + " done producing totaling “+sum +
34            "\nTerminating " + getName() + ".");
35
36      } // end method run
37
38   } // end class Producer
```

```
1  // Fig. 5.6: Consumer.java
2  // Consumer's run method controls a thread that loops four
3  // times and reads a value from sharedLocation each time.
4
5  public class Consumer extends Thread {
6     private Buffer sharedLocation; // reference to shared object
7
8     // constructor
9     public Consumer( Buffer shared )
10    {
11       super( "Consumer" );
12       sharedLocation = shared;
13    }
14
15    // read sharedLocation's value four times and sum the values
16    public void run()
17    {
18       int sum = 0;
19
20       for ( int count = 1; count <= 4; count++ ) {
21
22          // sleep 0 to 3 seconds, read value from Buffer and add to sum
23          try {
24             Thread.sleep( ( int ) ( Math.random() * 3001 ) );
25             sum += sharedLocation.get();
26          }
27
```

Consumer extends Thread

This is a shared object

Method run is overridden

The thread goes to sleep, then the buffer is read

```
28            // if sleeping thread interrupted, print stack trace
29            catch ( InterruptedException exception ) {
30               exception.printStackTrace();
31            }
32         }
33
34         System.err.println( getName() + " read values totaling: " + sum +
35            ".\nTerminating " + getName() + ".");
36
37      } // end method run
38
39   } // end class Consumer
```

```
1  // UnsynchronizedBuffer.java
2  // UnsynchronizedBuffer represents a single shared integer.
3
4  public class UnsynchronizedBuffer implements Buffer {
5     private int buffer = -1; // shared by producer and consumer
6
7     // place value into buffer
8     public void set( int value )
9     {
10       System.err.println( Thread.currentThread().getName() +
11          " writes " + value );
12
13       buffer = value;
14    }
15
16    // return value from buffer
17    public int get()
18    {
19       System.err.println( Thread.currentThread().getName() +
20          " reads " + buffer );
21
22       return buffer;
23    }
24
25 } // end class UnsynchronizedBuffer
```

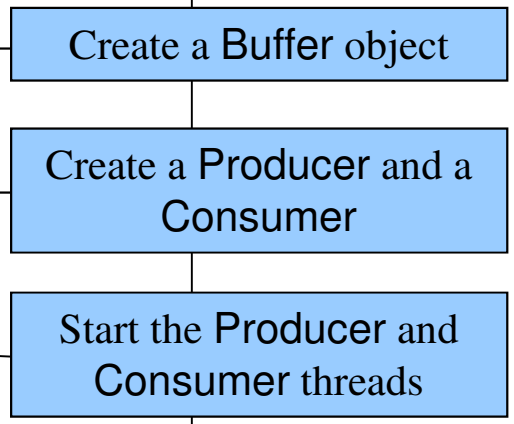
This class implements the Buffer interface

The data is a single integer

This method sets the value in the buffer

This method reads the value in the buffer

```
1  // SharedBufferTest.java
2  // SharedBufferTest creates producer and consumer threads.
3
4  public class SharedBufferTest {
5
6      public static void main( String [] args )
7      {
8          // create shared object used by threads
9          Buffer sharedLocation = new UnsynchronizedBuffer();
10
11         // create producer and consumer objects
12         Producer producer = new Producer( sharedLocation );
13         Consumer consumer = new Consumer( sharedLocation );
14
15         producer.start();  // start producer thread
16         consumer.start();  // start consumer thread
17
18     } // end main
19
20 } // end class SharedCell
```



5.5 Thread Synchronization

- Java uses monitors for thread synchronization
- The `sychronized` keyword
  - Every `synchronized` method of an object has a monitor
  - One thread inside a `synchronized` method at a time
  - All other threads block until method finishes
  - Next highest priority thread runs when method finishes

```
public class Storage {
     private String[] list = new String[100];
     private int count = 0;

     public synchronized void add(String
str){
          list[count] = str;
          count++;
     }
}
```

A synchronized method uses the instance as a semaphore to allow only one thread to get in at the same time.

```
for(int i=0;i<10;i++){
      synchronized(Storage.instance){
            Storage.instance.add(name+"-
"+i);
      }
}
```

An alternative are synchronized blocks in which you need to inform the instance used as the semaphore

Wait and Notify

Paulo André C

Sometimes a thread need to wait for other thread to finish its execution to start working.

Threads need to talk!

`obj.wait()`

The thread wait for a notification to be called on the instance.

Need to be called on a synchronized(obj) block or on a synchronized method from obj.

`obj.notify()`

The thread notify one thread that previously call the method wait() on obj.

Need to be called on a synchronized(obj) block or on a synchronized method from obj.

`obj.wait(time)`

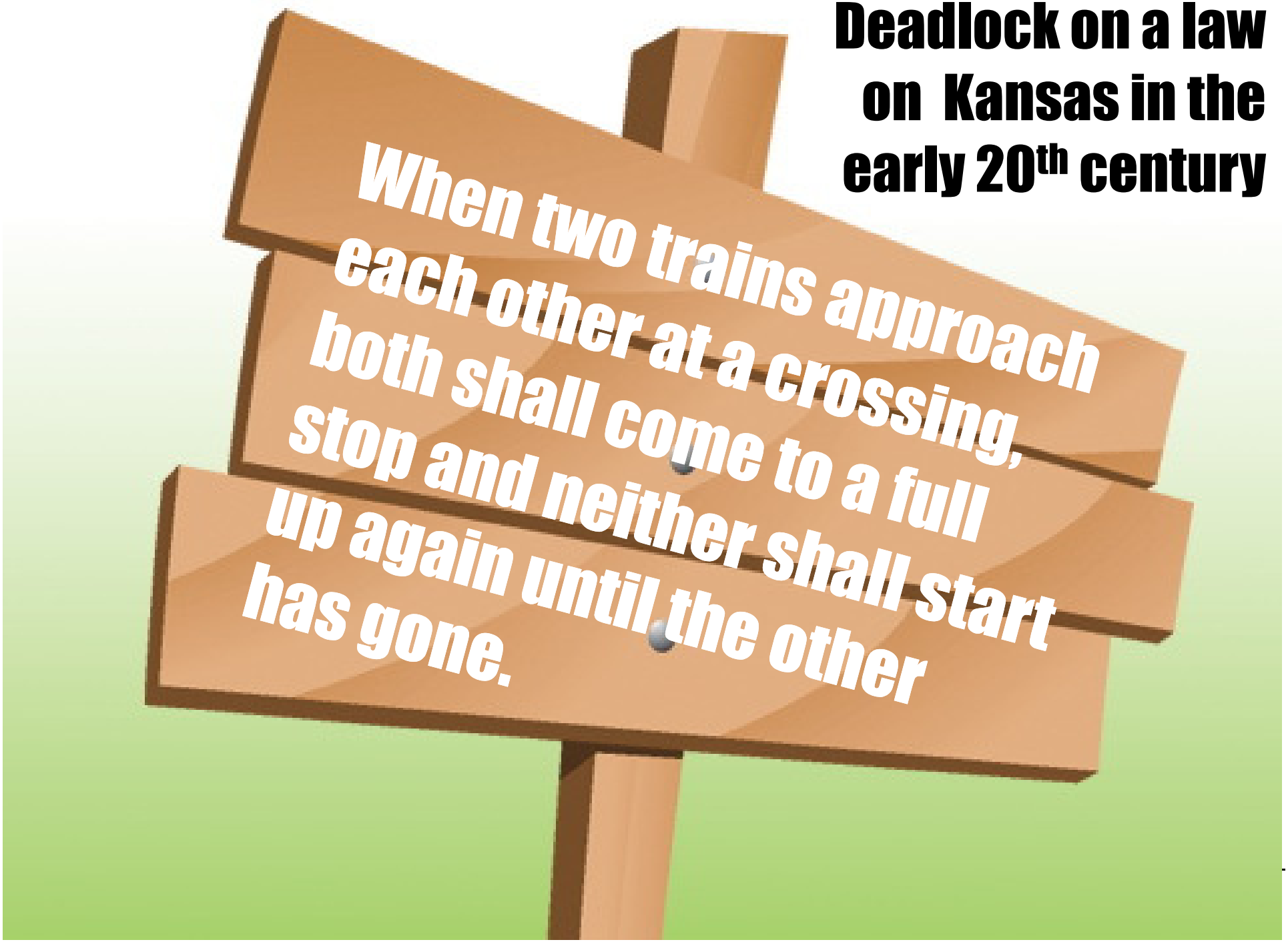
The thread waits for the notification, but wakes up after a time amount.

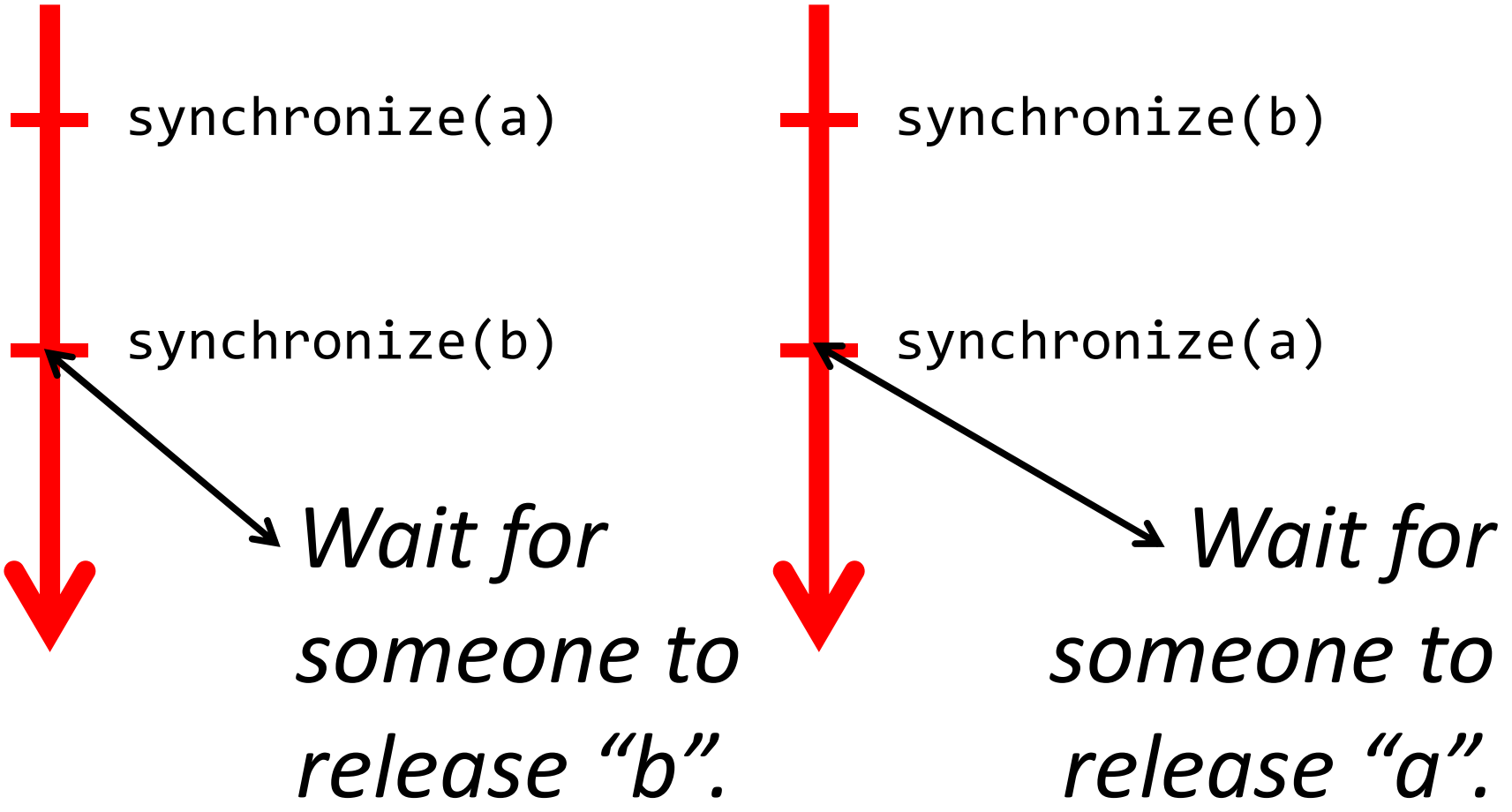
`obj.notifyAll()`

All objects that are waiting are notified and are eligible for execution.

Deadlock

Deadlock on a law on Kansas in the early 20th century





They are going to wait forever!

```
Lock l = new ReentrantLock();
```

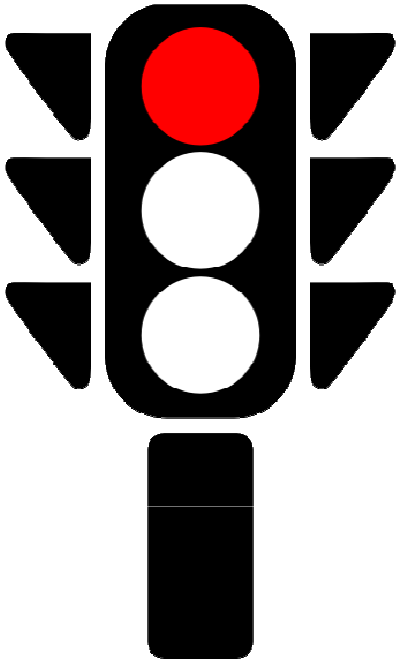
Lock can be used to control the access of several threads to a resource.

Equivalent

```
l.lock();
doSomething();
l.unlock();
```

```
synchronized(l){
    doSomething();
}
```

```
Condition c = lock.newCondition();
```



Condition substitutes the monitor methods, like Lock replaces the use of synchronized blocks and statements.

Equivalent

`wait()`	`c.await()`
`notify()`	`c.signal()`
`notifyAll()`	`c.signalAll()`

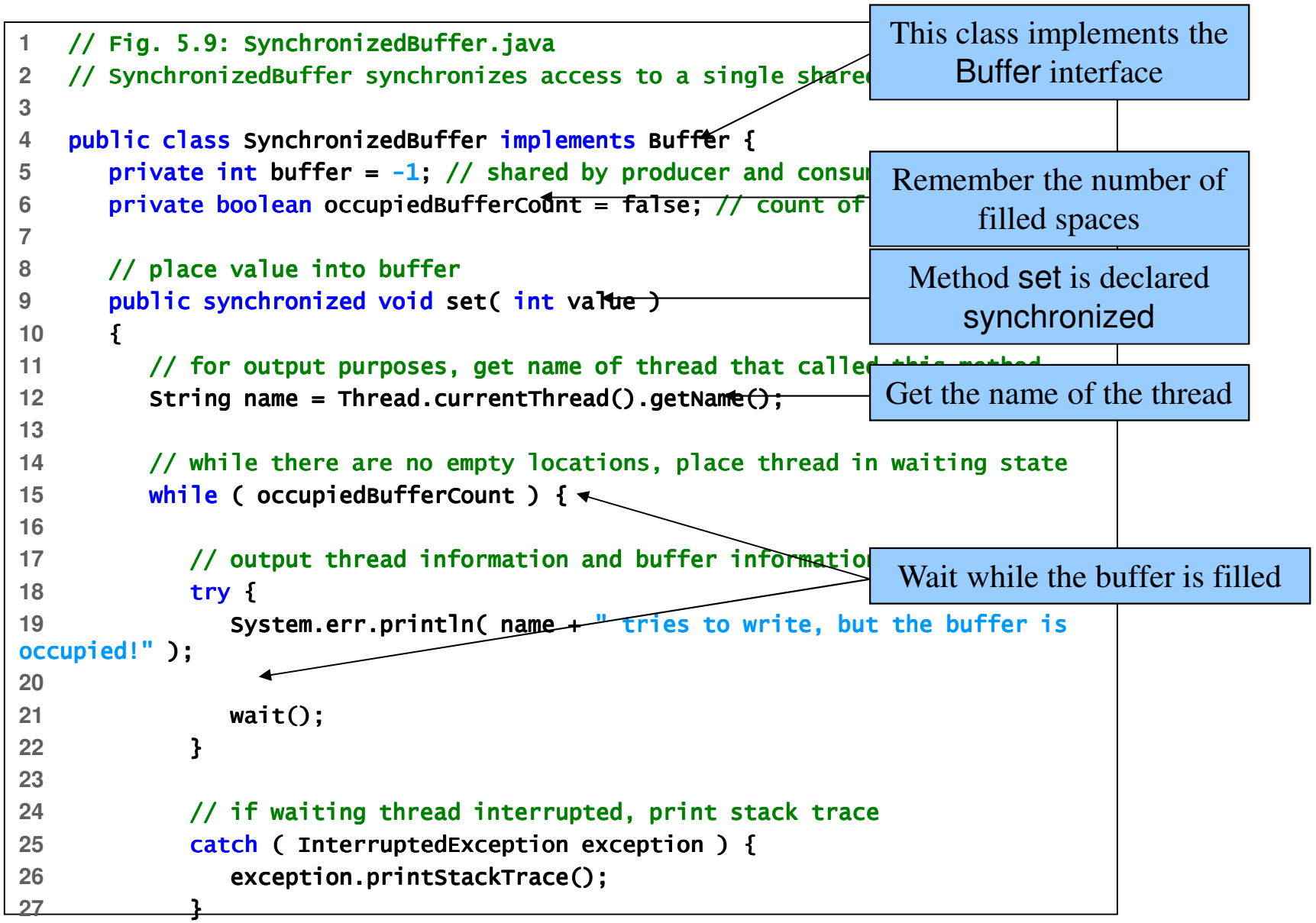
Condições para ocorrência de Deadlock, (Coffman et al. 1971 apud Tanenbaum)

- **Mutual exclusion condition**: Each resource is either currently assigned to exactly one process or it ‘s available.
- **Hold and Wait condition:** A process is currently holding at least one resource and requesting additional resources which are being held by other processes.
- **No Preemption condition:** Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
- **Circular wait condition:** There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

- All four of these conditions must be present for a deadlock to occur. If one of them is not present, no deadlock is possible!

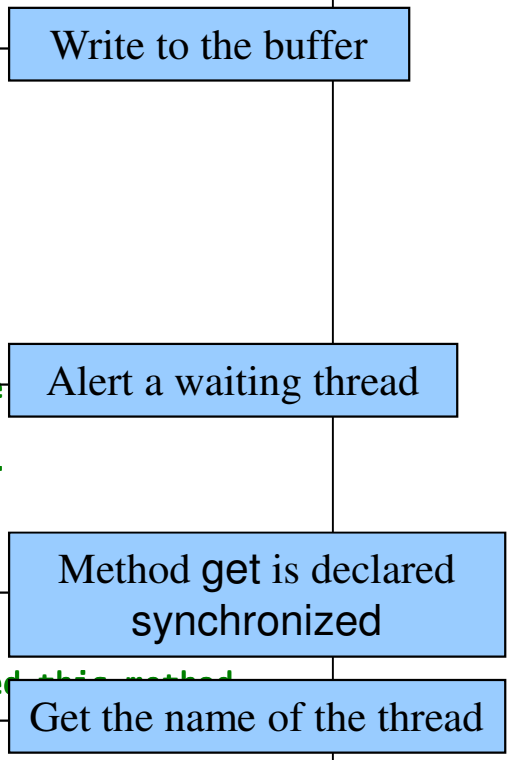
5.7 Producer/Consumer Relationship with Synchronization

- Synchronize threads to ensure correct data
- Exemplo Sincronizado: SynchronizedBuffer




```
1   // Fig. 5.9: SynchronizedBuffer.java
2   // SynchronizedBuffer synchronizes access to a single shared
3
4   public class SynchronizedBuffer implements Buffer {
5      private int buffer = -1; // shared by producer and consu
6      private boolean occupiedBufferCount = false; // count of
7
8      // place value into buffer
9      public synchronized void set( int value )
10     {
11        // for output purposes, get name of thread that called this method
12        String name = Thread.currentThread().getName();
13
14        // while there are no empty locations, place thread in waiting state
15        while ( occupiedBufferCount ) {
16
17           // output thread information and buffer informatio
18           try {
19              System.err.println( name + " tries to write, but the buffer is
occupied!" );
20
21              wait();
22           }
23
24           // if waiting thread interrupted, print stack trace
25           catch ( InterruptedException exception ) {
26              exception.printStackTrace();
27           }
```

```
28
29        } // end while
30
31        buffer = value; // set new buffer value
32
33        // indicate producer cannot store another value
34        // until consumer retrieves current buffer value
35        occupied=true;
36
37
38
39        notify(); // tell waiting thread to enter ready state
40
41     } // end method set; releases lock on SynchronizedBuffer
42
43     // return value from buffer
44     public synchronized int get()
45     {
46        // for output purposes, get name of thread that called this method
47        String name = Thread.currentThread().getName();
48
```



```
49         // while no data to read, place thread in waiting state
50         while ( !occupied) {
51
52            // output thread information and buffer informati
53            try {
54               System.err.println( name + " tries to read, but the buffer is
empty!" );
55
56               wait();
57            }
58
59            // if waiting thread interrupted, print stack trace
60            catch ( InterruptedException exception ) {
61               exception.printStackTrace();
62            }
63
64         } // end while
65
66         // indicate that producer can store another value
67         // because consumer just retrieved buffer value
68         occupied=false;
69         notify(); // tell waiting thread to become ready to execute
70
71         return buffer;
72      } // end method get; releases lock on SynchronizedBuffer
73   } // end class SynchronizedBuffer
```

Wait while the buffer is empty

Alert a waiting thread

Return the buffer

```java
// SharedBufferTest2.java
// SharedBufferTest2 creates producer and consumer threads.

public class SharedBufferTest2 {

   public static void main( String [] args )
   {
      // create shared object used by threads; we use a SynchronizedBuffer
      // reference rather than a Buffer reference so we can invoke
      // SynchronizedBuffer method displayState from main
      Buffer sharedLocation = new SynchronizedBuffer();

      // create producer and consumer objects
      Producer producer = new Producer( sharedLocation );
      Consumer consumer = new Consumer( sharedLocation );
      producer.start();  // start producer thread
      consumer.start();  // start consumer thread

   } // end main

} // end class SharedBufferTest2
```

Colocando para executar SharedBufferTest2...

```
Producer writes 1
Consumer reads 1
Consumer tries to read, but the buffer is empty.
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Consumer tries to read, but the buffer is empty.
Producer writes 4
Consumer reads 4Producer done producing totaling 10
Terminating Producer.
Consumer read values totaling: 10.
Terminating Consumer.
```

Producer writes 1
Consumer reads 1
Consumer tries to read, but the buffer is empty.
Producer writes 2
Consumer reads 2
Producer writes 3
Producer tries to write, but the buffer is occupied.
Consumer reads 3
Producer done producing totaling 10
Terminating Producer.
Producer writes 4
Consumer reads 4
Consumer read values totaling: 10.
Terminating Consumer.

Exercício

- Crie um novo buffer que seja capaz de armazenar 10 inteiros em uma estrutura First in, First out.
- O produtor e o consumidor devem ser capaz de gerar e consumir 40 números

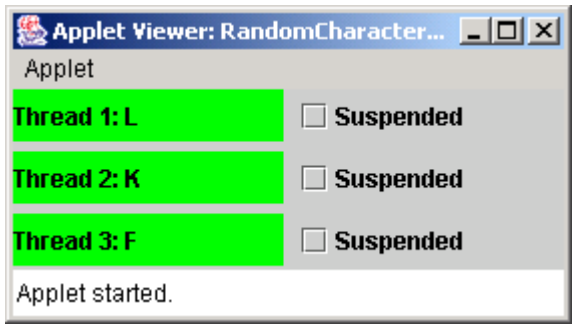
- Faça o produtor "mais rápido" que o consumidor

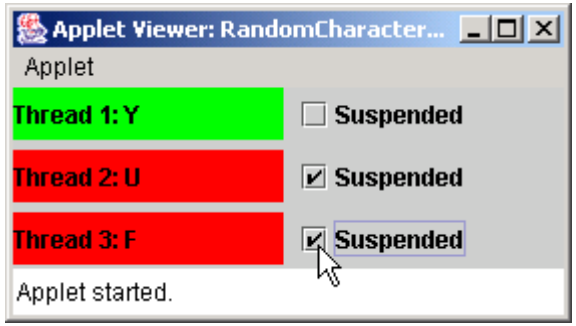
5.9 Daemon Threads

- Run for benefit of other threads
  - Do not prevent program from terminating
  - Garbage collector is a daemon thread
- Set daemon thread with method `setDaemon`
  - `setDaemon(true);`
  - `setDaemon(false);`

5.10 Runnable Interface

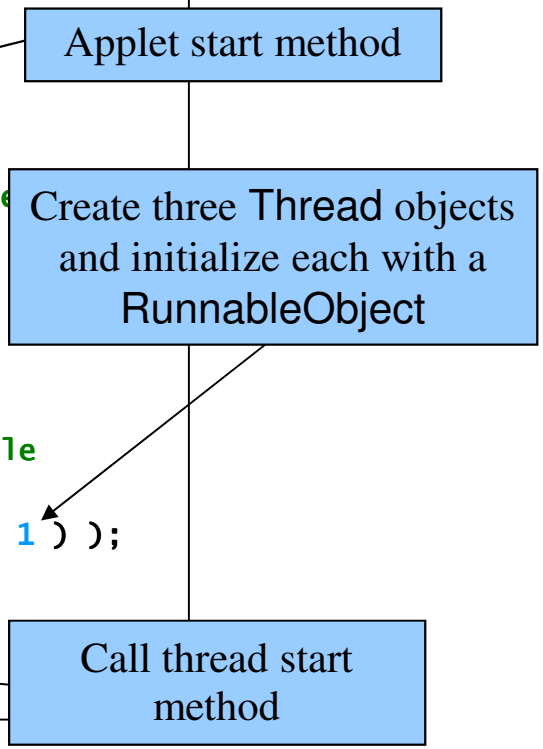
- A class cannot extend more than one class
- Implement Runnable for multithreading support
- Exemplo: RandomCharacters.java





```java
// RandomCharacters.java
// Class RandomCharacters demonstrates the Runnable interface
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RandomCharacters extends JApplet implements ActionListener {
   private String alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   private final static int SIZE = 3;
   private JLabel outputs[];
   private JCheckBox checkboxes[];
   private Thread threads[];
   private boolean suspended[];

   // set up GUI and arrays
   public void init()
   {
      outputs = new JLabel[ SIZE ];
      checkboxes = new JCheckBox[ SIZE ];
      threads = new Thread[ SIZE ];
      suspended = new boolean[ SIZE ];

      Container container = getContentPane();
      container.setLayout( new GridLayout( SIZE, 2, 5, 5 ) );

```

```
26          // create GUI components, register listeners and attach
27          // components to content pane
28          for ( int count = 0; count < SIZE; count++ ) {
29             outputs[ count ] = new JLabel();
30             outputs[ count ].setBackground( Color.GREEN );
31             outputs[ count ].setOpaque( true );
32             container.add( outputs[ count ] );
33
34             checkboxes[ count ] = new JCheckBox( "Suspended" );
35             checkboxes[ count ].addActionListener( this );
36             container.add( checkboxes[ count ] );
37          }
38
39       } // end method init
40
41       // create and start threads each time start is called (i.e., afte
42       // init and when user revists Web page containing this applet)
43       public void start()
44       {
45          for ( int count = 0; count < threads.length; count++ ) {
46
47             // create Thread; initialize object that implements Runnable
48             threads[ count ] =
49                new Thread( new RunnableObject(), "Thread " + ( count + 1 ) );
50
51             threads[ count ].start(); // begin executing Thread
52          }
```

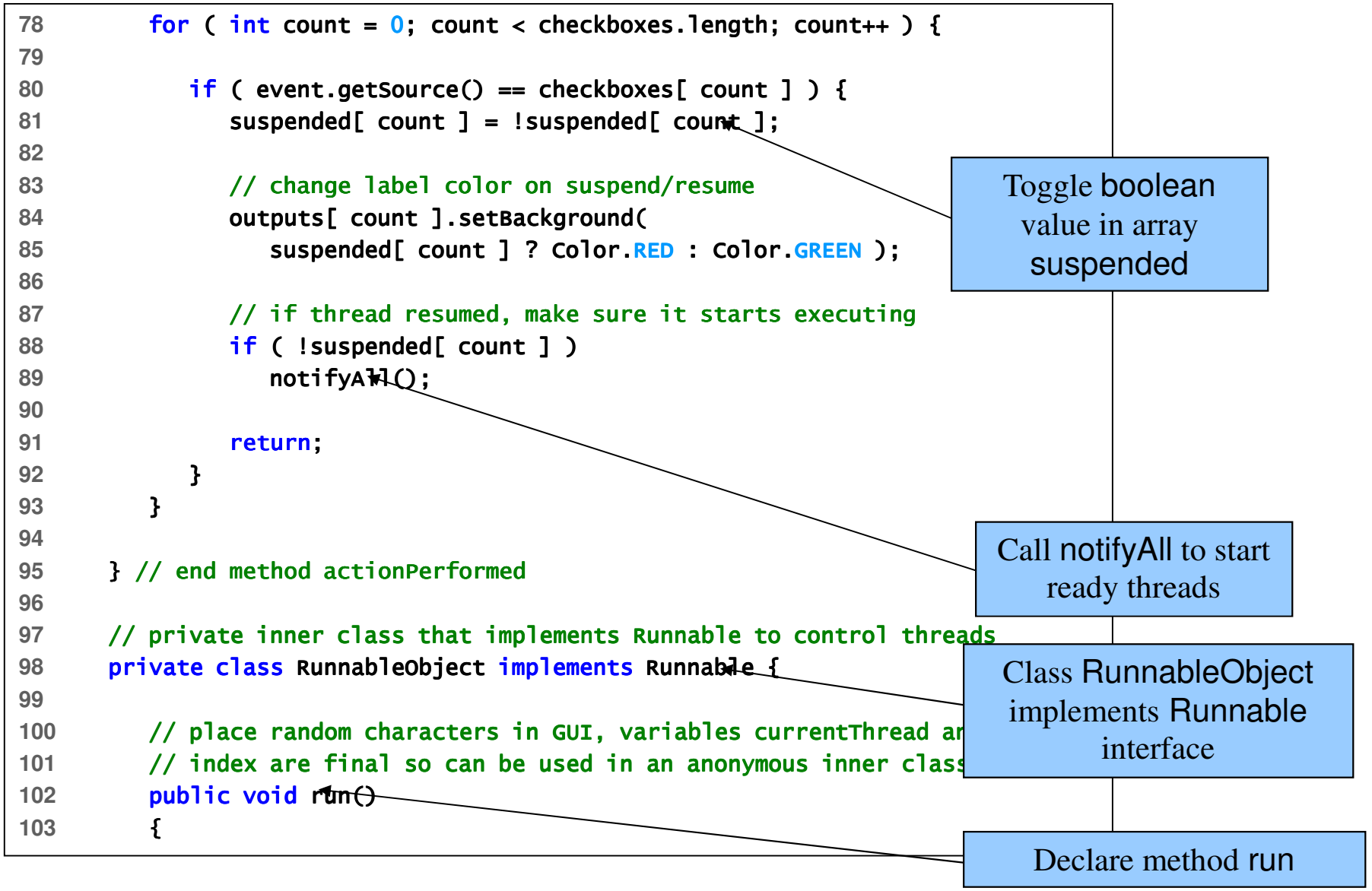


```
53     }
54
55     // determine thread location in threads array
56     private int getIndex( Thread current )
57     {
58        for ( int count = 0; count < threads.length; count++ )
59           if ( current == threads[ count ] )
60              return count;
61
62        return -1;
63     }
64
65     // called when user switches Web pages; stops all threads
66     public synchronized void stop()
67     {
68        // set references to null to terminate each thread's run method
69        for ( int count = 0; count < threads.length; count++ )
70           threads[ count ] = null;
71
72        notifyAll(); // notify all waiting threads, so they can termi
73     }
74
75     // handle button events
76     public synchronized void actionPerformed( ActionEvent event )
77     {
```

Method stop stops all threads

Set thread references in array threads to

Invoke method notifyAll to ready waiting threads


78 for (int count = 0; count < checkboxes.length; count++) {
79
80 if (event.getSource() == checkboxes[count]) {
81 suspended[count] = !suspended[count];
82
83 // change label color on suspend/resume
84 outputs[count].setBackground(
85 suspended[count] ? Color.RED : Color.GREEN);
86
87 // if thread resumed, make sure it starts executing
88 if (!suspended[count])
89 notifyAll();
90
91 return;
92 }
93 }
94
95 } // end method actionPerformed
96
97 // private inner class that implements Runnable to control threads
98 private class RunnableObject implements Runnable {
99
100 // place random characters in GUI, variables currentThread ar
101 // index are final so can be used in an anonymous inner class
102 public void run()
103 {
Toggle boolean value in array suspended
Call notifyAll to start ready threads
Class RunnableObject implements Runnable interface
Declare method run

```
104         // get reference to executing thread
105         final Thread currentThread = Thread.currentThread();
106
107         // determine thread's position in array
108         final int index = getIndex( currentThread );
109
110         // loop condition determines when thread should stop; loo
111         // terminates when reference threads[ index ] becomes nul
112         while ( threads[ index ] == currentThread ) {
113
114            // sleep from 0 to 1 second
115            try {
116               Thread.sleep( ( int ) ( Math.random() * 1000 ) );
117
118               // determine whether thread should suspend execution;
119               // synchronize on RandomCharacters applet object
120               synchronized( RandomCharacters.this ) {
121
122                  while ( suspended[ index ] &&
123                     threads[ index ] == currentThread ) {
124
125                     // temporarily suspend thread execution
126                     RandomCharacters.this.wait();
127                  }
128               } // end synchronized statement
```

The while loop executes as long as the index of array threads equals currentThread

The synchronized block helps suspend currently executing thread

Invoke method wait on applet to place thread in waiting state

```
129
130            } // end try
131
132            // if thread interrupted during wait/sleep, print stack trace
133            catch ( InterruptedException exception ) {
134               exception.printStackTrace();
135            }
136
137            // display character on corresponding JLabel
138            SwingUtilities.invokeLater(
139               new Runnable() {
140
141                  // pick random character and display it
142                  public void run()
143                  {
144                     char displayChar =
145                        alphabet.charAt( ( int ) ( Math.random() * 26 ) );
146
147                     outputs[ index ].setText(
148                        currentThread.getName()  + ": " + displayChar );
149                  }
150
151               } // end inner class
152
153            ); // end call to SwingUtilities.invokeLater
```

Anonymous inner class implements Runnable interface

```
154
155          } // end while
156
157          System.err.println( currentThread.getName() + " terminating" );
158
159       } // end method run
160
161    } // end private inner class RunnableObject
162
163 } // end class RandomCharacters
```

When you need something to run in parallel, you can create a class that implements Runnable.

The problem is that it doesn't return anything to you. And if you need some result from the execution?

returns

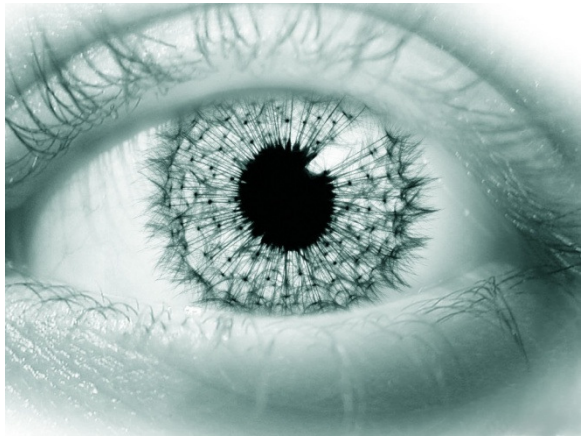
Callable **Future**

Executing a Callable class, you receive an implementation of Future, from which you can retrieve the result when you need.

An implementation of Callable is similar to an implementation of Runnable, except that it returns a value in the end of its execution.

```
public class CallableClass
      implements Callable<Integer> {

      public Integer call() throws Exception
{
            //execute logic
     //return the result
      }
}
```

The Future represents a value that you will have. When you try to get it, it waits until the value is calculated.



```
ExecutorService e =
    Executors.newCachedThreadPool();
CallableClass cc = new CallableClass();
Future<Integer> future = e.submit(cc)
//do other stuff

Integer result = future.get();
//It has the non-blocking method isDone
```

Exercício

- Execute CallableTest em CallableFuture package
- Altere a classe CallableClass de modo a que a tarefa seja concluída antes do final de “doing other stuff”

- Pense em um exemplo de situação onde seria interessante utilizar uma Callable e uma inteface Future

HomeWork

- *Create main method that sum the numbers of an array with 10000 elements.*
- *Create a Callable that sums 100 of these numbers.*
- *Execute many sums in parallel and use Future to get the results.*

Concurrency Tools

IEC - ITA

To create a new Thread is expensive in terms of operating system resources.

To achieve a good performance it is a good practice to maintain a thread pool where threads can be reused.

The ExecutorService can be used to execute runnables, but it can reuse existing threads.

```
ExecutorService es =
    Executors.newCachedThreadPool();

for(int i=0;i<10;i++){
    es.execute(new RunnableImpl());
}
es.shutdown();//can finish the threads
```

- **Executors** é uma classe que contém métodos de fábrica e utilitários

- The method **newCachedThreadPool** creates a **thread pool** that creates new threads as needed, but will reuse previously constructed threads when they are available.

- It will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool.

- Threads that have not been used for sixty seconds are terminated and removed from the cache