# COP 4610L: Operating Systems Lab

*Distributed Applications in the Enterprise*

Lecture Set 2

Dr. R. Lent

# Concurrency and Multithreading

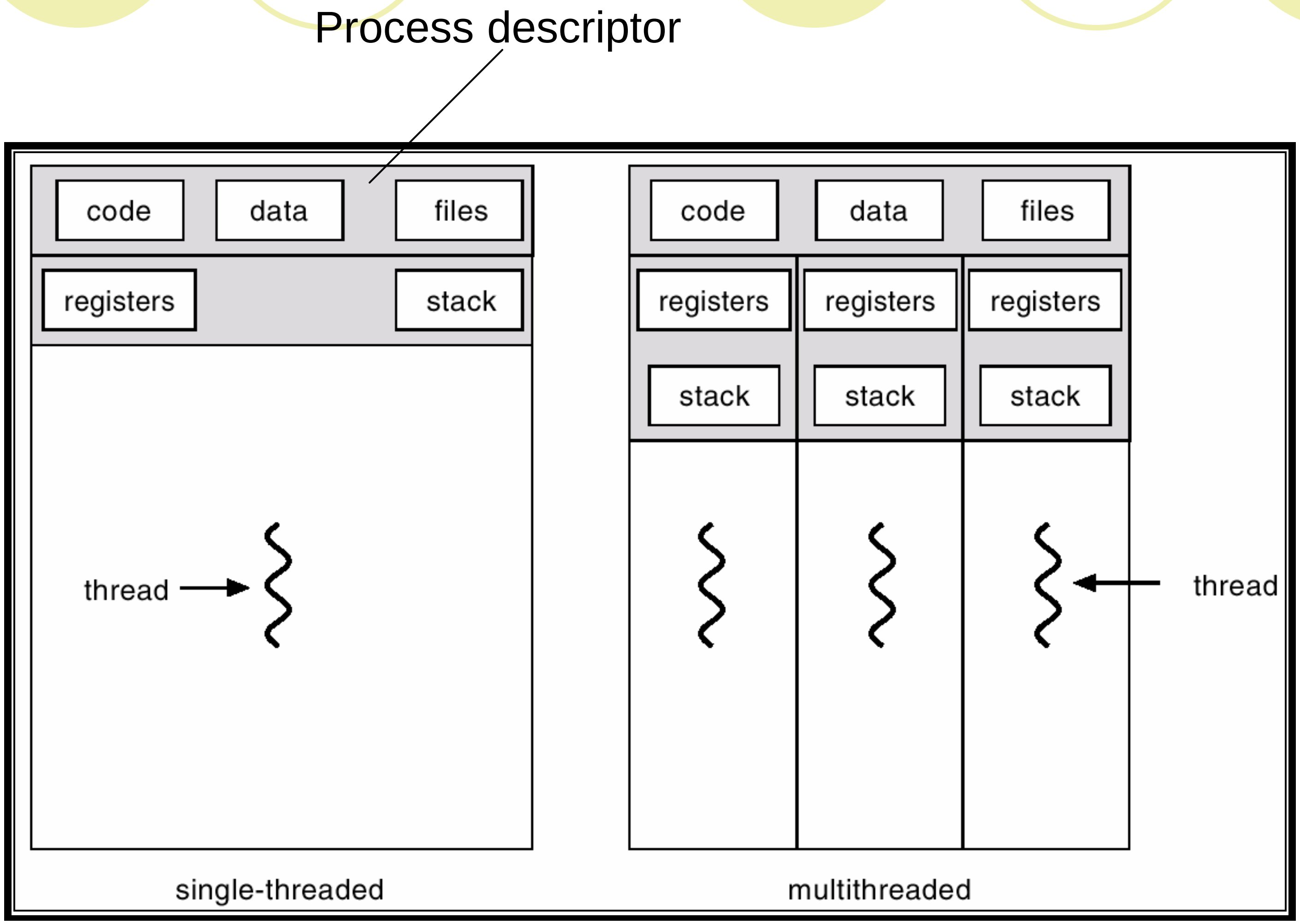
## Concurrency

- Concurrency : The execution of different activities (processes) in parallel
- Concurrency cannot be avoided:
  - Users are concurrent - a person can handle several tasks at once and expects the same from a computer.
  - Multiprocessors are becoming more prevalent.
  - A distributed system (client/server system) is naturally concurrent.
  - A windowing system is naturally concurrent.
  - When doing I/O (slow activity) it is helpful to handle concurrently additional work.

## Terminology

- Program: an executable file
- Process: an instance of a program active in the system. An O.S. process is a unit of resource allocation both for CPU time and memory.
- thread: a light-weight process (LWP). Name comes from expression “thread of control”
- task: used either as process or thread
- a program can have several processes or threads - some generated by the program itself, some by the operating system
- a process can have several threads

## Single and Multithreaded Processes



## Time Slicing

- A thread/process runs for a short time (*quantum*) and then is pre-empted (O.S. re-evaluate which thread should be run)
- This allows even single processor machines to run multiple threads
- On PC's a timeslice tends to be about about fifty-five milliseconds long

## Advantages of Multithreading

- Some programs are required to do more than one thing at a time
  - These programs are easier to design and implement with threads
- Concurrency allows you to maintain a high availability of services
  - each request for service can be handled by a new thread
  - reduces bottleneck of pending requests

## Advantages of Multithreading (cont’d)

- Concurrency can use CPU cycles more efficiently
  - if one thread becomes blocked, other threads can be run
- Concurrency supports asynchronous message passing
  - objects can send messages and continue without having to wait for the message to be processed
- Concurrency supports parallelism
  - on machines with multiple CPUs, concurrent programming can be used to exploit available computing power to improve performance

## Limitations of Multithreading

- Safety
  - Since threads within a program share the same address space, they can interfere with one another
  - Synchronization mechanisms are needed to control access to shared resources
- Liveness
  - Threads can stop running for any number of reasons
  - Deadlock can occur when threads depend upon each other to complete their activities
- Nondeterminism
  - Multithreaded activities can be arbitrarily interleaved
    - no two executions of the program need be identical
    - makes multithreaded programs harder to predict, understand and debug

# Limitations of Multithreading (cont'd)

- Thread Construction Overhead
  - Constructing a thread and setting it in motion is slower and more memory intensive than constructing a normal object and invoking a method on it
- Context Switching Overhead
  - When there are more active threads than CPUs, the Java runtime must switch from one thread to another
- Synchronization Overhead
  - increases the cost of a method call by at least four times

## Threads in Java

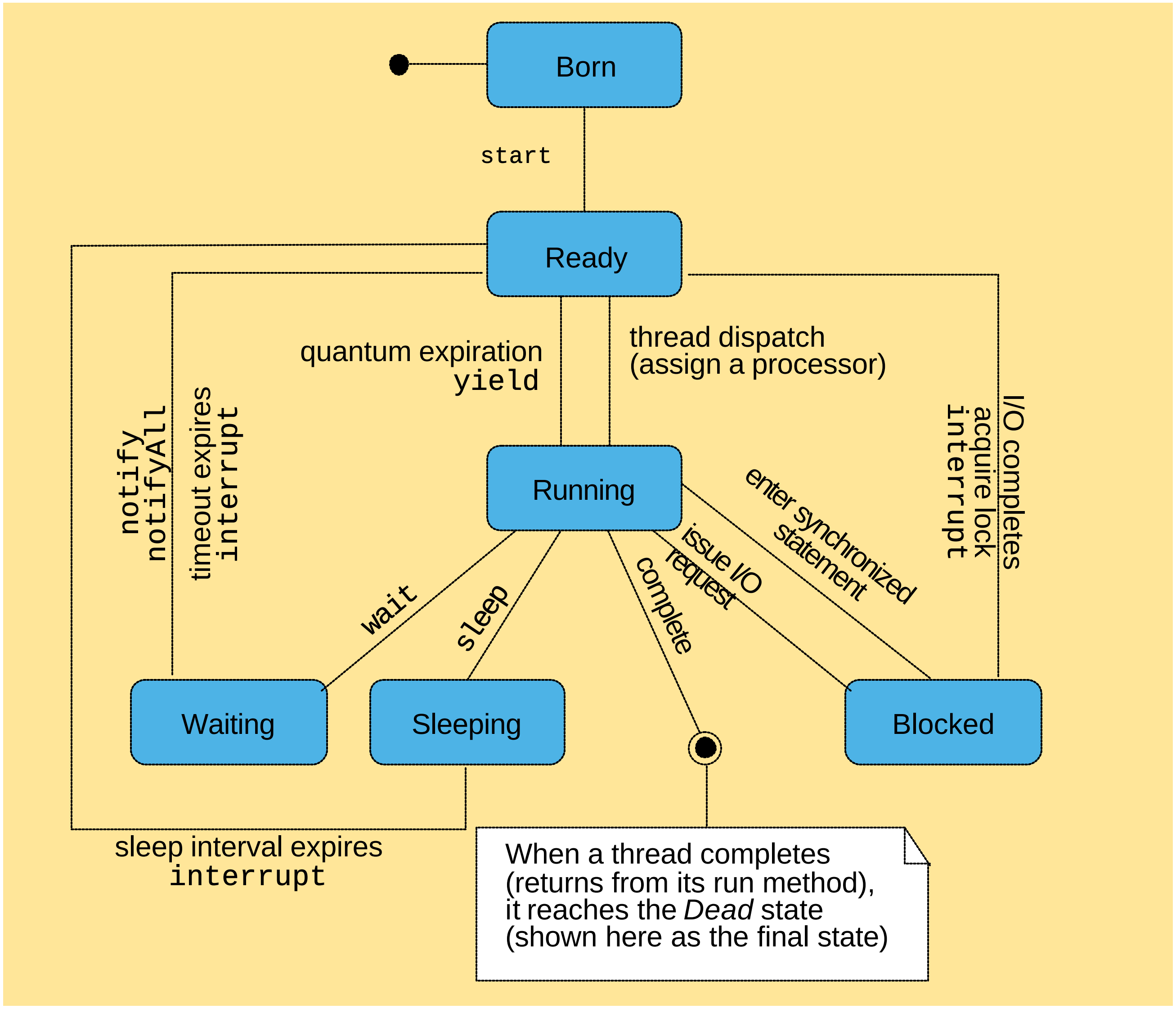
Concurrency is normally available via OS primitives but Java provides built-in multithreading

## Thread States: Life Cycle of a Thread

Thread states:

- Born state (Created)
  - Thread was just created
- Ready state (Runnable)
  - Thread’s `start` method invoked
  - Thread can now execute
- Running state
  - Thread is assigned a processor and running
- Dead state (Terminated)
  - Thread has completed or exited
  - Eventually disposed of by system
- Non runnable (Waiting, Sleeping, or Blocked)

## Thread Life-Cycle Statechart Diagram

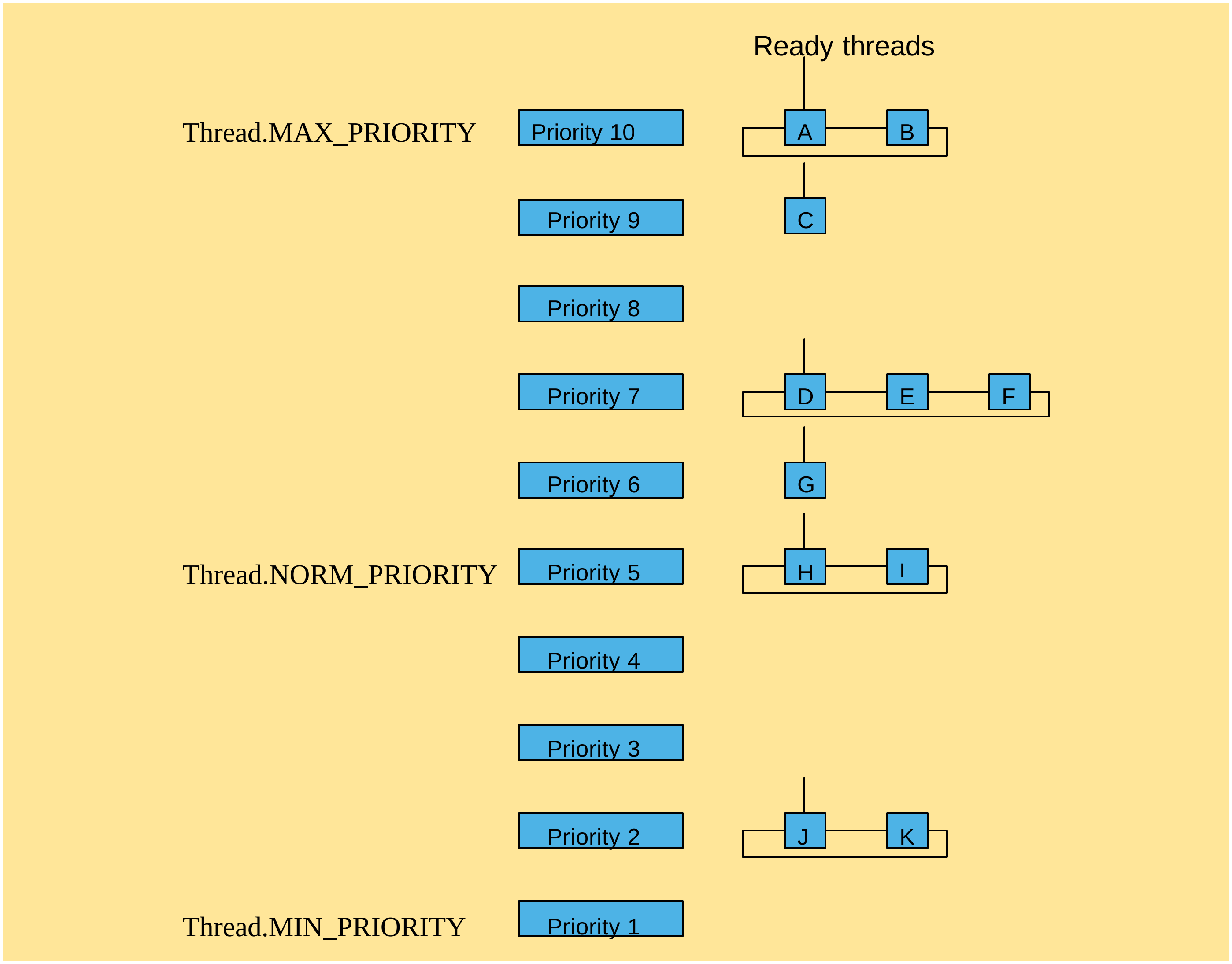


When a thread completes (returns from its run method), it reaches the *Dead* state (shown here as the final state)

## Thread Priorities and Thread Scheduling

- Most computers have only one CPU, so threads must share the CPU with other threads
- The execution of multiple threads on a single CPU, in some order, is called scheduling
- The Java runtime supports a very simple, deterministic scheduling algorithm known as fixed priority scheduling
- Each Java thread is given a numeric priority between `MIN_PRIORITY` (1) and `MAX_PRIORITY (10)`
- Each thread is assigned time on the processor (called a quantum)
- Keeps highest priority threads running

## Thread priority scheduling example



## Scheduling of Threads is Preemptive

- If a thread with a higher priority than the currently executing thread needs to execute, the higher priority thread is immediately scheduled
- Threads of equal priority are not preempted

## Java Multithreading Support

Java contains only a few basic constructs and classes to support concurrent programming

- **`Thread`** class and **`Runnable`** interface
  - used to initiate and control threads
- **`synchronized`** and **`volatile`** keywords
  - used to control code in objects that may participate in multiple threads
- **`wait`**, **`notify`** and **`notifyAll`** methods
  - used to coordinate activities across threads

## Creating Java Threads

Java provides two techniques:

1. Extend the **`Thread`** class and overriding **`run`**
2. implementing the **`Runnable`** interface (useful when a class is already extending another class)

## 1. Extending **Thread**

```
class myThread extends Thread {
   long someValue;
    myThread (long sValue) {
        this.someValue = sValue;
   }
   public void run() {
        // compute something
   }
}
                             *   *   *
myThread p = new myThread (123);
p.start();
```

## 2. Implementing the **`Runnable`** Interface

```
class myThread implements Runnable {
   long someValue;
    myThread (long sValue) {
        this.someValue = sValue;
   }
   public void run() {
        //compute something
   }
}
                              *   *   *


Thread p = new Thread (new myThread(123));
p.start();
```

## ThreadTester.java 1/3

```java
// Fig. 16.3: ThreadTester.java
// Multiple threads printing at different intervals.

public class ThreadTester {

   public static void main( String [] args )
   {
      // create and name each thread
      PrintThread thread1 = new PrintThread( "thread1" );
      PrintThread thread2 = new PrintThread( "thread2" );
      PrintThread thread3 = new PrintThread( "thread3" );

      System.err.println( "Starting threads" );

      thread1.start(); // start thread1 and place it in ready state
      thread2.start(); // start thread2 and place it in ready state
      thread3.start(); // start thread3 and place it in ready state

      System.err.println( "Threads started, main ends\n" );

   } // end main

} // end class ThreadTester

```

## ThreadTester.java 2/3

```
// class PrintThread controls thread execution
class PrintThread extends Thread {
   private int sleepTime;

   // assign name to thread by calling superclass constructor
   public PrintThread( String name )
   {
      super( name );

      // pick random sleep time between 0 and 5 seconds
      sleepTime = ( int ) ( Math.random() * 5001 );
   }

   // method run is the code to be executed by new thread
   public void run()
   {
      // put thread to sleep for sleepTime amount of time
      try {
         System.err.println(
            getName() + " going to sleep for " + sleepTime );

         Thread.sleep( sleepTime );
      }

```

## ThreadTester.java 3/3

```java
      // if thread interrupted during sleep, print stack trace
      catch ( InterruptedException exception ) {
         exception.printStackTrace();
      }

      // print thread name
      System.err.println( getName() + " done sleeping" );

   } // end method run

} // 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 and Stopping Threads

- There are three main methods to control a thread:

  `public void start()`

  Prepares thread to be run

  `public void run()`

  Performs the work of the thread

  `public final void stop()`

  Halts the thread

- A thread “dies” when run() finishes or stop() is invoked
- You do not invoke run(), instead you invoke start()

## Naming Threads

- Useful for debugging
- To give a name to a thread:

```
public Thread(String name)
```

- This is normally called from a subclass:

```
class ThreadTest extends Thread {
     ThreadTest(String name) {
         super(name);
     }
     public void run() {
        System.out.println(this.getName()); ...
   } }
```

- **`getName()`** returns the name of the thread

## Issues of Concurrency

- Multi-threading increases the complexity of coding; the programmer now has to consider each thread and the interactions between threads

- We need to deal with the following general classes of problems:
  - Interferences (race conditions)
  - Deadlocks

    when all threads are loop blocked while waiting for a resource to be freed, making further progress impossible

## Interference

- Destructive update, caused by arbitrary interleaving of read and write actions (see producer-consumer example)
- A general solution to the problem is *mutual exclusion*
- *Mutual exclusion* ensures that only one thread can access a shared resource at a time

## Synchronized Java Methods

- Java associates a monitor with each object
- This monitor enforces MUTEX access to synchronized methods invoked on the object
- When a threads exists a synchronized method, it releases the monitor

## Example (no sync)

```
class Counter {
private int count = 0;

public void Inc() {
   int n = count;
   count = n+1;
}

}
```

## Possible Scenario

| Thread 1 | Thread 2 | Value of count |
|---|---|---|
| | counter.inc(); | 0 |
| | n=count; // 0 | 0 |
| counter.inc(); | | 0 |
| n=count; // 0 | | 0 |
| count = n+1; //1 | | 1 |
| | count = n+1; //1 | 1 |

## Example (sync)

```
class Counter {

private int count = 0;

public void synchronized Inc() {
   int n = count;
   count = n+1;
}

}
```

## Possible Scenario (sync)

| Thread 1 | Thread 2 | count |
|---|---|---|
| | counter.inc(); | 0 |
| | Acquires monitor | 0 |
| | n=count; // 0 | 0 |
| counter.inc(); | | 0 |
| Can not acquire monitor ! | | 0 |
| blocked | count = n+1; //1 | 1 |
| blocked | Releases monitor | 1 |
| Acquires monitor | | 1 |
| n=count; // 1 | | 1 |
| count = n+1; //2 | | 2 |
| Releases monitor | | 2 |

## Thread Coordination

- **`wait()`**

  this halts execution of the thread and returns any monitor keys held by that thread. The thread is put in the 'waiting' pool

- **`notify()`**

  wakes up one object in the waiting pool

- **`notifyAll()`**

  wakes up all of the objects in the waiting pool

Generally, the wait() method is called at the beginning of a **synchronized** method, and the notifyAll() method at the end of a synchronized method

## Consumer-Producer Example

## Synchronized Block

- To isolate only part of the code inside a method:

```
synchronized(syncObject) {
  // This code can be accessed
  // by only one thread at a time
}
```

- Before the block can be entered, the lock must be acquired on syncObject

## Example of Deadlock

A well-known example of deadlock arises in the so-called *Dining Philosophers Problem*

- Four philosophers are dining at a table. In the middle of the table is a bowl of spaghetti, and between each pair of philosophers is a fork. In order to eat. a philosopher must pick up two forks (to their left and right).
- Each Philosopher can pick up only one fork at a time, and when they pick up a fork they will wait for the other fork to become available.
- Each philosopher will put down their fork only after they have eaten.

Suppose all the philosophers pick up the fork to their left at the same time; the forks to their right are unavailable, their neighbour has picked it up. Now each philosopher will hold onto their single fork, waiting (in vain) for the other fork to become available.

The system is in deadlock.

A possible solution can be found here:

`http://jliusun.bradley.edu/~jiangbo/appletdemo/DiningPhilosophers/demo.html`

## Preventing Deadlocks

- Deadlock generally arises when all threads are waiting for some event to happen before they can make progress, but while they are waiting, some resource is tied up, preventing that event from happening.

- One way of preventing deadlock is to allow processes to free resources e.g. forks in the Dining Philosophers Problem, or the monitor key in the Queue/Consumer example.

## Swing and Multithreading

- Multithreaded programming prevents the program from knowing exactly when a thread will execute
- Swing components are not thread-safe
- Interactions with Swing GUI should be performed one thread at a time
- To prevent incorrect results, use the event-dispatching thread:
  - class **`SwingUtilities`**
  - method **`invokeLater`**

## Example: RandomCharacters