

# Introduction to Java threads

ICS432 - Fall 2008  
Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

## Threads in Programming Languages

- Several programming languages have long provided constructs/abstractions for writing concurrent programs
  - Modula, Ada, etc.
- Java does it like it does everything else, by providing a Thread class
  - You create a thread object
  - Then you can start the thread
- Java provides several higher level abstractions, which we'll see later in the semester

## Extending the Thread class

- To create a thread, you can extend the thread class and override its "run()" method

```
class MyThread extends Thread {
    public void run() {
        ...
    }
    ...
}
```

```
myThread t = new MyThread();
```

## Example

```
public class MyThread extends Thread {
    public void run() {
        for (int i=0; i<10; i++) {
            System.out.println("Hello world #" + i);
        }
    }
    ...
}
```

```
myThread t = new MyThread();
```

## Spawning a thread

- To launch, or **spawn**, a thread, you just call the thread's **start()** method
- **WARNING:** Don't call the run() method directly to launch a thread
  - If you call the run() method directly, then you just call some method of some object, and the method executes
    - Fine, but probably not what you want
  - The start() method, which you should not override, does all the thread launching
    - It launches a thread that starts its execution by calling the run() method

## Example

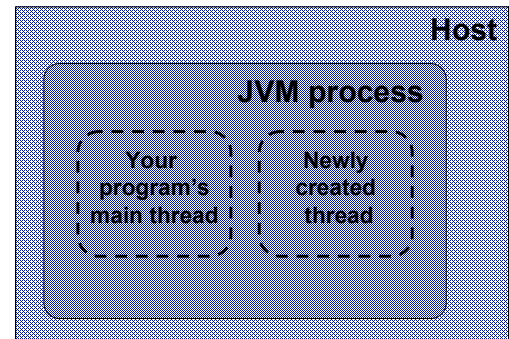
```
public class MyThread extends Thread {
    public void run() {
        for (int i=0; i<5; i++) {
            System.out.println("Hello world #" + i);
        }
    }
}

public class MyProgram {
    public MyProgram() {
        MyThread t = new MyThread();
        t.start();
    }
    public static void main(String args[]) {
        MyProgram p = new MyProgram();
    }
}
```

## What happens

- The previous program runs as a Java process
  - that is, a thread running inside the JVM
- When the start() method is called, the main thread creates a new thread
- We now have two threads
  - The “main”, “original” thread
  - The newly created thread
- Both threads are running
  - The main thread doesn’t do anything
  - The new thread prints messages to screen and exits
- When both threads are finished, then the process terminates

## What happens



## Example

- The previous example wasn’t very interesting because the main thread did nothing
  - Admittedly, this example is not interesting because the program doesn’t do anything useful, but we’ll get there eventually
- In fact, we could have achieved the same result with no thread at all
- So, let’s have the main thread to something

## Example

```
public class myThread extends Thread {
    public void run() {
        for (int i=0; i<5; i++)
            System.out.println("Hello world #" + i);
    }
}

public class MyProgram {
    public MyProgram() {
        MyThread t = new MyThread();
        t.start();
        for (int i=0; i<5; i++)
            System.out.println("foo");
    }
    public static void main(String args[]) {
        MyProgram p = new MyProgram();
    }
}
```

## What happens?

- Now we have the main thread printing to the screen **and** the new thread printing to the screen
- Question: what will the output be?
- Answer: Impossible to tell for sure
  - If you know the implementation of the JVM on your particular machine, then you can probably tell
  - But if you write this code to be run anywhere, then you can’t expect to know what happens
- Let’s look at what happens on my laptop

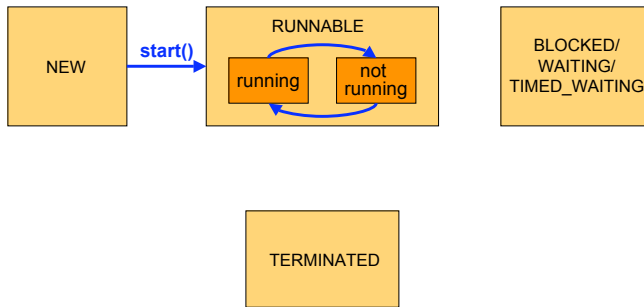
## Example Execution

```
~% java MyProgram
Hello world #0
Beep 0
Beep 1
Beep 2
Beep 3
Beep 4
Hello world #1
Hello world #2
Hello world #3
Hello world #4
~% █
```

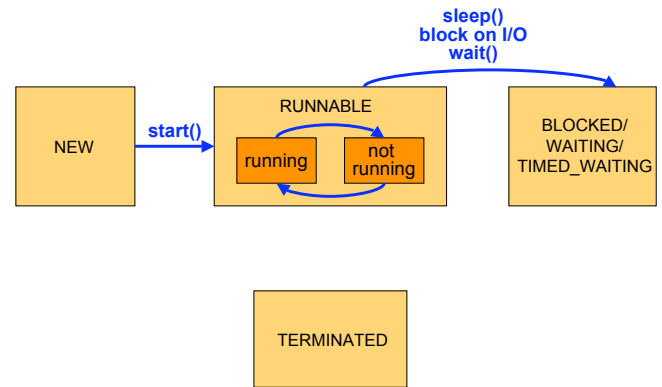




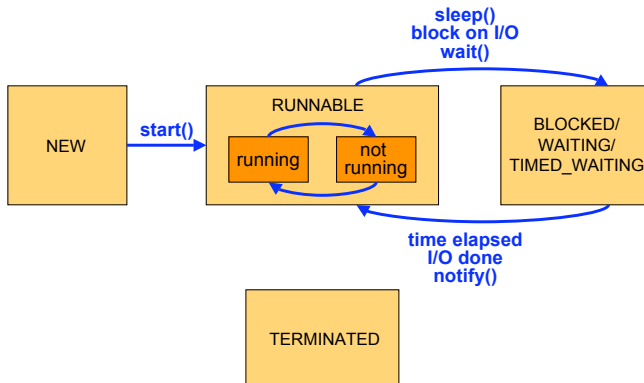
## Thread Lifecycle: 4 states



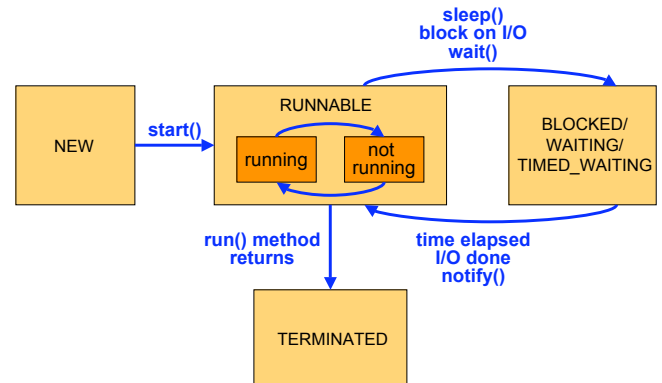
## Thread Lifecycle: 4 states



## Thread Lifecycle: 4 states

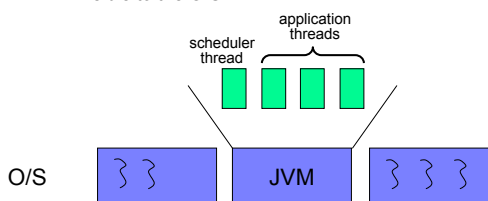


## Thread Lifecycle: 4 states



## Thread Scheduling

- The JVM keeps track of threads, enacts the thread state transition diagram
- Question: who decides which runnable thread to run?
- Old versions of the JVM used **Green Threads**
  - User-level threads implemented by the JVM
  - Invisible to the O/S

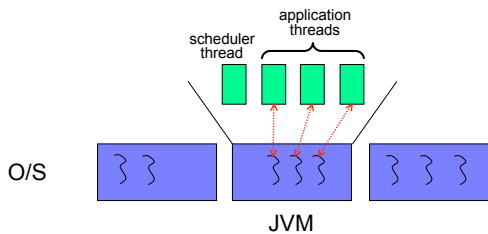


## Beyond Green Threads

- Green threads have all the disadvantage of user-level threads (see previous set of lecture notes)
  - Most importantly: Cannot exploit multi-core, multi-processor architectures
- Later, the JVM provided **native threads**
  - Green threads are typically not available anymore
  - you can try to use "java -green" and see what your system says

## Java Threads / Kernel Threads

- In modern JVMs, application threads are *mapped* to kernel threads



## Java Threads / Kernel Threads

- This gets a bit complicated
  - The JVM has a thread scheduler for application threads, which are mapped to kernel threads
  - The O/S also has a thread scheduler for kernel threads
  - Several application threads could be mapped to the same kernel thread!
- The JVM is itself multithreaded!
- We have threads everywhere
  - Application threads in the JVM
  - Kernel threads that run application threads
  - Threads in the JVM that do some work for the JVM
- Let's look at a running JVM

## A Running JVM

- On my laptop, a Java program that does nothing

```

Terminal - vim - 111x15
% ps auxwwt
casanova 21435  0.0  0.7  279656  7888  p3  S   10:02PM  0:00.26  31  0:00.01  0:00.01  java DoNothing
21435  0.0  0.7  279656  7888  S   10:02PM  0:00.26  31  0:00.06  0:00.09
21435  0.0  0.7  279656  7888  S   10:02PM  0:00.26  47  0:00.00  0:00.00
21435  0.0  0.7  279656  7888  S   10:02PM  0:00.26  44  0:00.00  0:00.00
21435  0.0  0.7  279656  7888  S   10:02PM  0:00.26  47  0:00.00  0:00.00
21435  0.0  0.7  279656  7888  S   10:02PM  0:00.26  41  0:00.00  0:00.00
21435  0.0  0.7  279656  7888  S   10:02PM  0:00.26  44  0:00.00  0:00.00
21435  0.0  0.7  279656  7888  S   10:02PM  0:00.26  44  0:00.01  0:00.08
21435  0.0  0.7  279656  7888  S   10:02PM  0:00.26  31  0:00.00  0:00.00
21435  0.0  0.7  279656  7888  S   10:02PM  0:00.26  31  0:00.00  0:00.00
21435  0.0  0.7  279656  7888  S   10:02PM  0:00.26  44  0:00.00  0:00.00
    
```

- 10 threads!

## A Running JVM

- On my laptop, a Java program that creates 4 threads

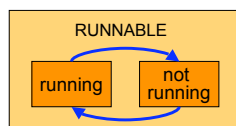
```

Terminal - vim - 114x19
% ps auxwwt
casanova 21480  0.0  0.8  281920  7968  p1  S   10:10PM  0:00.30  31  0:00.01  0:00.01  java ThreadExample
21480  0.0  0.8  281920  7968  S   10:10PM  0:00.30  31  0:00.06  0:00.09
21480  0.0  0.8  281920  7968  S   10:10PM  0:00.30  47  0:00.00  0:00.00
21480  0.0  0.8  281920  7968  S   10:10PM  0:00.30  44  0:00.00  0:00.00
21480  0.0  0.8  281920  7968  S   10:10PM  0:00.30  47  0:00.00  0:00.00
21480  0.0  0.8  281920  7968  S   10:10PM  0:00.30  41  0:00.00  0:00.00
21480  0.0  0.8  281920  7968  S   10:10PM  0:00.30  44  0:00.00  0:00.00
21480  0.0  0.8  281920  7968  S   10:10PM  0:00.30  44  0:00.01  0:00.08
21480  0.0  0.8  281920  7968  S   10:10PM  0:00.30  31  0:00.00  0:00.00
21480  0.0  0.8  281920  7968  S   10:10PM  0:00.30  44  0:00.01  0:00.01
21480  0.0  0.8  281920  7968  S   10:10PM  0:00.30  31  0:00.00  0:00.00
21480  0.0  0.8  281920  7968  S   10:10PM  0:00.30  31  0:00.00  0:00.00
21480  0.0  0.8  281920  7968  S   10:10PM  0:00.30  31  0:00.00  0:00.00
    
```

- 14 threads!
  - 10 from before, one for each application thread

## So what?

- At this point, it seems that we throw a bunch of threads in, and we don't really know what happens
- To some extent it's true, but we have ways to have some control
- In particular, what happens in the RUNNABLE state?



- Can we control how multiple RUNNABLE threads become running or not running?

## The yield() method: example

- With the `yield()` method, a thread will pause and give other RUNNABLE threads the opportunity to execute for a while

```

public class MyThread extends Thread {
    public void run() {
        for (int i=0; i<5; i++) {
            System.out.println("Hello world #" + i);
            Thread.yield();
        }
    }
}

public class MyProgram {
    public MyProgram() {
        MyThread t = new MyThread();
        t.start();
        for (int i=0; i<5; i++) {
            System.out.println("foo");
            Thread.yield();
        }
    }

    public static void main(String args[]) {
        MyProgram p = new MyProgram();
    }
}
    
```

## Example Execution

```
% java MyProgram
foo
Hello world #0
foo
Hello world #1
foo
Hello world #2
foo
Hello world #3
foo
Hello world #4
% java MyProgram
foo
Hello world #0
foo
Hello world #1
foo
Hello world #2
Hello world #3
Hello world #4
%
```

- The use of `yield` made the threads' executions more interleaved
  - Switching between threads is more frequent
- But it's still not deterministic!
- Programs should NEVER rely on `yield()` for correctness
  - `yield()` is really a "hint" to the JVM

## Thread Priorities

- The `Thread` class has a `setPriority()` and a `getPriority()` method
  - A new `Thread` inherits the priority of the thread that created it
- Thread priorities are integers ranging between `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY`
  - The higher the integer, the higher the priority

## Thread Priorities and Scheduling

- Whenever there is a choice between multiple runnable threads, the JVM picks the higher priority one
  - High priority threads may yield to prevent starvation of low-priority threads
- The JVM is preemptive
  - If a higher priority thread is started, it gets to run
- Modern JVMs (post green threads) use time slicing
  - Threads of the highest priorities get chosen in a round-robin fashion
  - The use of `yield()` isn't required but, as we saw, it can increase the frequency of switching between threads
- In spite of all this:
  - The JVM can only influence the way in which threads are scheduled
  - Ultimately, the decision is left to the O/S

## So what?

- It is important to know the basics of thread scheduling to understand the behavior of concurrent programs
- One should NEVER rely on scheduling aspects to ensure correctness of the program
  - Since scheduling depends on the JVM and on the O/S, correctness due to scheduling is not portable

## The join() method

- The `join()` method causes a thread to wait for another thread's termination
- This is useful for "dispatching" work to a worker thread and waiting for it to be done
- Let's see it used on an example

```
public class JoinExample {
    public JoinExample() {
        Thread t1 = new Thread1();
        t1.start();
        System.out.println("Parent thread: waiting for child to finish");
        try { t1.join(); } catch (InterruptedException e) {}
        System.out.println("Parent thread: child has finished");
    }

    public static void main(String args[]) {
        JoinExample j = new JoinExample();
    }

    private class Thread1 extends Thread {
        public void run() {
            for (int i=0; i<3; i++) {
                System.out.println("Child thread: iteration"+i);
                try { Thread.sleep(1000); } catch (InterruptedException e) {}
            }
        }
    }
}
```

## The Runnable Interface

- What if you want to create a thread that extends some other class?
  - e.g., a multi-threaded applet is at the same time a Thread and an Applet
- Java does not allow for double inheritance
- Which is why it has the concept of interfaces
- So another way to create a thread is to have runnable objects
- It's actually the most common approach
  - Allows to add inheritance in a slightly easier way after the fact
- Let's see this on an example

## Runnable Example

```
public class RunnableExample {  
  
    class MyTask implements Runnable {  
        public void run() {  
            for (int i=0; i<50; i++)  
                System.out.print("#");  
        }  
    }  
  
    public RunnableExample() {  
        Thread t = new Thread(new MyTask());  
        t.start();  
        for (int i=0; i<50; i++)  
            System.out.println(".");  
    }  
  
    public static void main(String args[]) {  
        RunnableExample p = new RunnableExample();  
    }  
}
```

## Conclusion

- Two ways to create threads
  - extends Thread
  - implements Runnable
- Thread Scheduling is complex, not fully deterministic, and should not be counted on to guarantee program correctness