

# Java Threads

---

Università di Modena e Reggio Emilia  
*Prof. Nicola Bicocchi ([nicola.bicocchi@unimore.it](mailto:nicola.bicocchi@unimore.it))*



# JVM and Operating System

---

- A multitasking operating system assigns CPU time (slices) to threads
- Small time-slices (20ms) provide the **illusion of parallelism** (on multi-core machines it is a partial illusion)
- OS is **preemptive**, if a thread is executed until
  - time slice is over or it ends its execution
  - it blocks (synchronization with threads or resources)
  - another thread acquires more priority



# JVM and Operating System

---

- The JVM gets the CPU as assigned by the OS's scheduling mechanism
- Java is a specification with many different implementations\*
  - Some JVMs operate like a mini-OS and schedule their own threads
  - Most JVMs use the OS scheduler (a Java thread is actually mapped to a system thread)

\*[https://en.wikipedia.org/wiki/List\\_of\\_Java\\_virtual\\_machines](https://en.wikipedia.org/wiki/List_of_Java_virtual_machines)



# Processes

---

- In operating systems (OS), a process is an instance of a running application
- A process has its own private address space, code, data, opened files, etc..
- Processes do not share memory (separate address spaces), thus they must communicate through IPC mechanisms offered by the operative systems (i.e., pipes, signals)
- A process might contain one or more threads running within the context of the process.

# Threads

---

- Threads are sometimes called **lightweight processes**.
- Like processes, **they work independently** and each thread has its own stack, program counter, and local variables.
- However, **threads within the same process** are less **insulated from each other** than processes. They share the same address space and, consequently, have access to shared variables and objects.
- Sharing variables is **very fast way** for communicating but frequently causes bugs unseen in single-thread programs.
- OOP principle of *separation of concerns* is broken!



# Threads

---

- Every Java program must have at least one thread, the main thread. When the program starts running, the JVM creates this thread and calls the main() method within that thread.
- There are other threads created by the JVM that users usually don't notice (e.g., garbage collector).



# Threads

---

- There are many reasons to use threads in your Java programs. If you use Android, Swing, JavaFX, Servlets, RMI, or Enterprise JavaBeans (EJB) technology, you may already be using threads without realizing it.
- Main reasons for using threads:
  - make the UI more responsive
  - take advantage of multiprocessor systems
  - perform asynchronous operations or background processing



# Why threads ?

---

- Imagine a **stock-broker application** with three capabilities:
  - Download current stock prices
  - Store stock prices into a database
  - Short-term (5 days) analysis for buy/sell signals
  - Long-term (1 year) analysis for buy/sell signals



# Single-threaded scenario

---

- In a single-threaded runtime environment, actions execute one after another
  - An action happens only when the previous one is completed.
- If the download takes 5 minutes, should the user enjoy an unresponsive UI?
- If short-term analysis takes additional 10 minutes...the result may have come too late. Prices could already have changed!



# Multi-threaded scenario

---

- In a multi-threaded scenario
  - the **download can execute in background** (i.e. in another thread)
  - the **analysis, too, can execute in background** (i.e. in another thread)
  - ...all while the user is interacting with the application and awaiting for eventual notifications (remember your phone?)



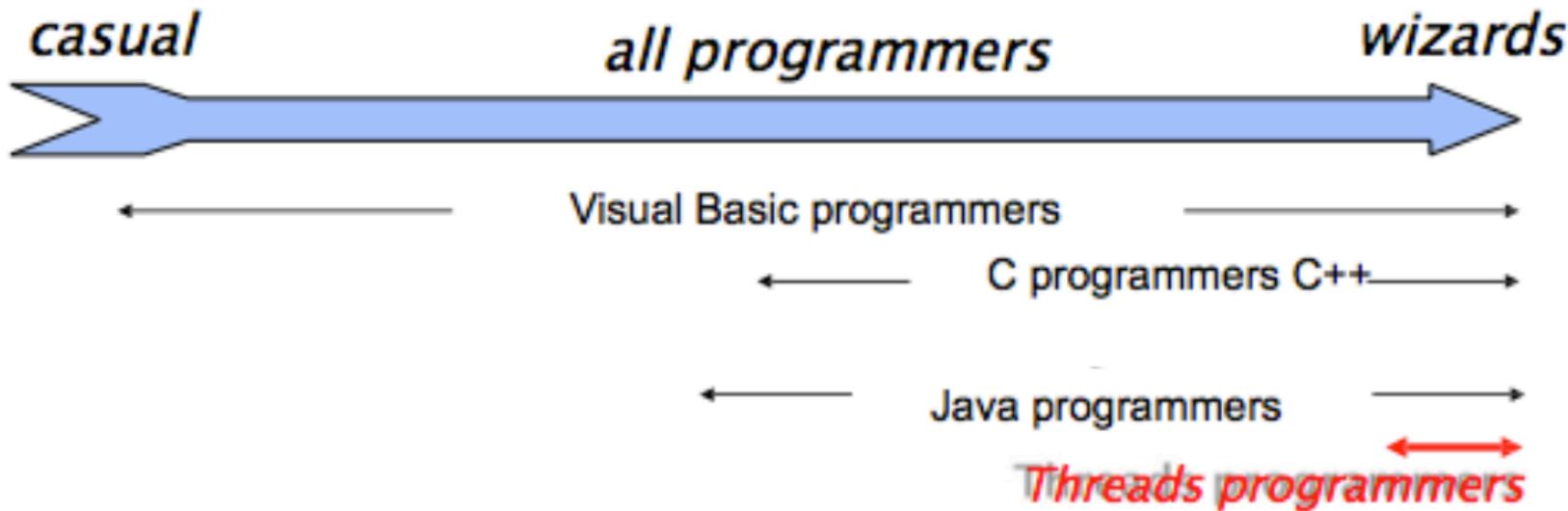
# The good

---

- Enable parallelism
- Lighter than processes for both
  - Creation(i.e., fork())
  - Communication (i.e., r/w pipes ...)

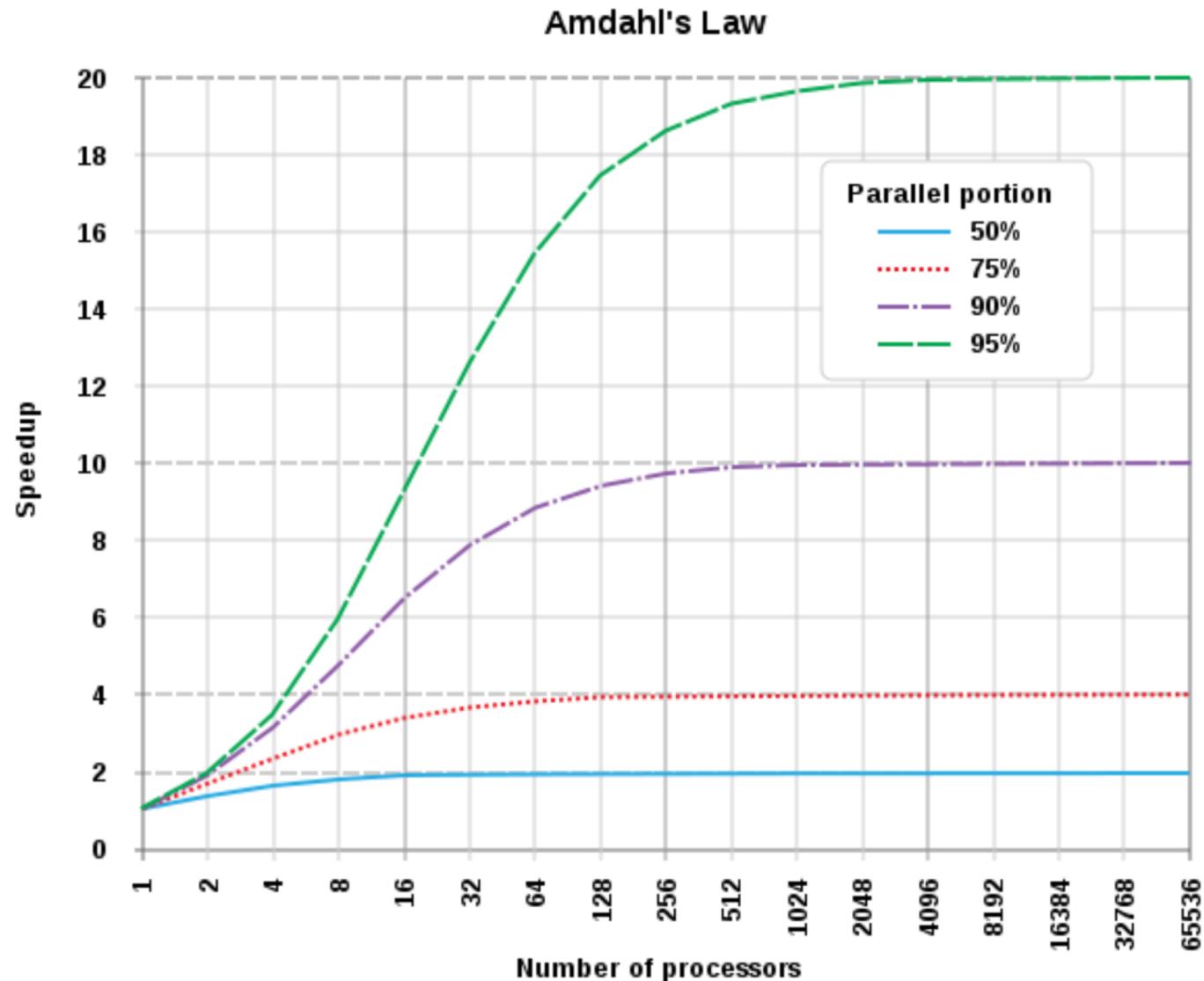


# The bad



- Hard for most programmers
- Even for experts, development is **painful**
- **Threads break abstraction**: can't design modules independently.

# The important!



# Creating a Thread

---

- In Java it is not possible to explicitly call the syscall fork() as in C. Syscalls fork() and exec()  
can be jointly used as:

```
Process p = Runtime.getRuntime().exec("/bin/ls -a -l");
```

Or, alternatively

```
Process p = (new ProcessBuilder("/bin/ls", "-a", "-l")).start()
```



# Creating a Thread

---

- Threads can be created by
  - extending the **Thread class** and overriding its `run()` method.
  - passing to the **Thread class constructor** an object implementing the **Runnable Interface**.
- It is legal to create many threads using the same `Runnable` object as the target.



# Creating a Thread

---

## Extending Thread

```
Class T extends Thread {  
    public void run() {  
        //code here  
    }  
}  
T t = new T();  
t.start();
```

## Implementing Runnable interface (usually better)

```
Class R implements Runnable {  
    public void run() {  
        //code here  
    }  
}  
Thread t = new Thread(new R());  
t.start();
```



# Creating a Thread

---

```
class Counter implements Runnable {  
    public void run() {  
        for(int i=0; i<10; i++)  
            System.out.println(Thread.currentThread().getName() + i);  
    }  
}  
  
public class Runner{  
    public static void main(String[] args) {  
        Thread t1 = new Thread(new Counter(), "T_A");  
        Thread t2 = new Thread(new Counter(), "T_B");  
        Thread t3 = new Thread(new Counter(), "T_C");  
        t1.start(); t2.start(); t3.start();  
    }  
}
```



# Starting a Thread

---

- When a Thread object is created, it does not start executing until its `start()` method is invoked.
- When a Thread object exists but hasn't been started, it is in the *new state* and is not considered alive.
- Method `start()` can be called on a Thread object only once. If `start()` is called more than once on same object, it will throw a `RuntimeException`



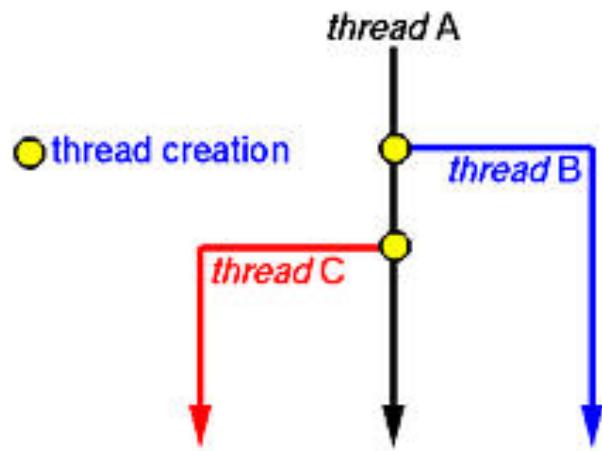
# Starting a Thread

---

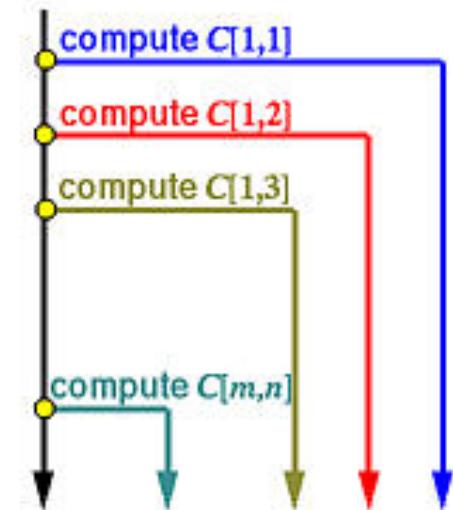
- It is not guaranteed that threads will start running in the same order in which their start() methods have been called.
- It is not guaranteed that a thread keeps executing until it's done (it is not guaranteed that its loop completes before another thread begins)
- Nothing is guaranteed except:
  - Each thread will start and will run to completion after acquiring the CPU a finite number of times (hopefully)



# Starting a Thread, examples

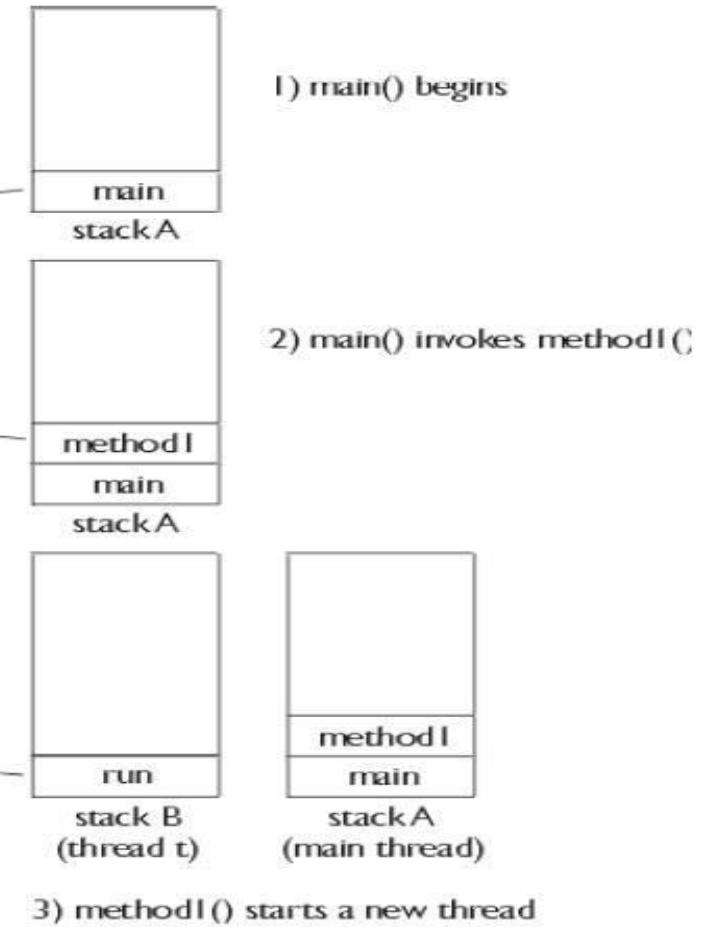


```
for (int i = 1; i <= m; i++)  
    for (int j = 1; j <= n; j++)  
        create a thread to compute C[i,j];
```



# Starting a Thread, the stack

```
public static void main(String [] args) {  
    // running  
    // some code  
    // in main()  
    method1();  
    // running  
    // more code  
}  
  
void method1() {  
    Runnable r = new MyRunnable();  
  
    Thread t = new Thread(r);  
  
    t.start();  
    // do more stuff  
}
```



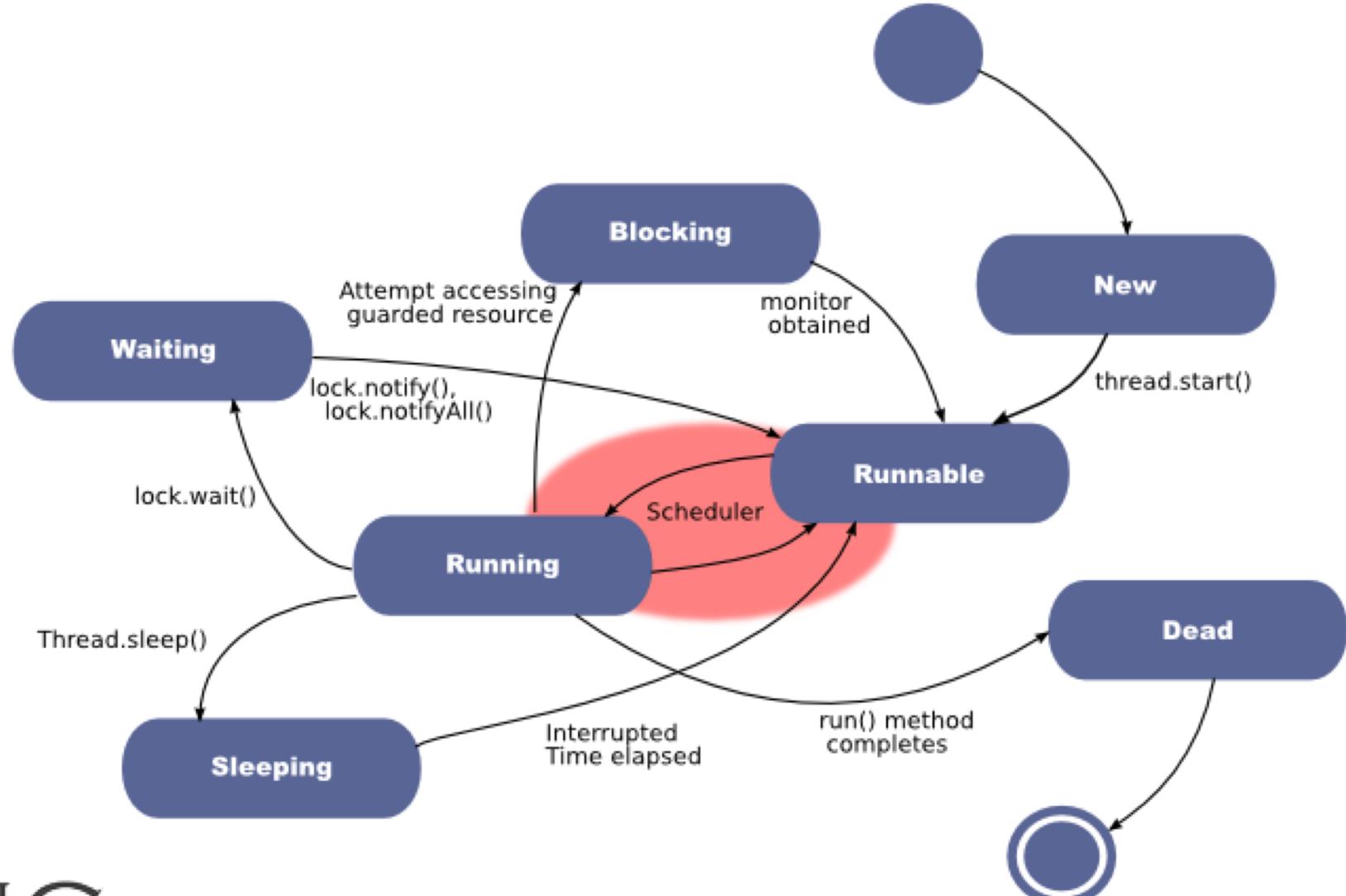
# Terminating a Thread

---

- If the parent thread terminates, all of its child threads terminate as well.
- Child threads share resources with the parent thread, including variables. When the parent thread terminates, the child threads will not be able to access to those resources that the parent thread owns.
- Thus, if the parent thread terminates earlier than its own child threads, synchronization mechanisms are required.



# Thread states



# Thread state: Running

---

- This is the state a thread is in when the thread scheduler selects it (from the runnable pool) to be **the currently executing thread**.
- A thread can transition from the running state for several reasons:
  - Calling System.exit() (DEAD)
  - Calling Thread.sleep() (SLEEPING)
  - The thread can acquire a resource but there is no work to do. The thread calls object.wait() and waits for another thread calling notify() (WAITING)
  - The thread can't acquire a resource (BLOCKING)



# Thread state: Runnable

---

- A thread which is eligible to run, but the scheduler has not selected it to be the running thread
- A thread first enters the runnable state when the start() method is invoked
- A thread can also return to the runnable state after either the running, blocked, waiting, or sleeping state
- When a thread is in the runnable state, it is considered alive



# Thread state: Waiting

---

- A thread that can acquire a resource but there is no work to do. The thread calls `object.wait()` and waits for another thread calling `object.notify()` or `object.notifyAll()`



# Thread state: Blocking

---

- A thread which is NOT eligible to run
- A thread blocked waiting for a resource ( I/O or an object's lock) e.g.:
  - if data become available in an inputstream the thread is reading from
  - If an object's lock becomes available



# Thread state: Sleeping

---

- A thread which is sleeping after an explicit call to the sleep() method
- Back to Runnable state when the thread wakes up because its sleep time has expired.

```
try {  
    Thread.sleep(1000); // one second  
} catch (InterruptedException ex) {}
```



# Thread priority

---

- By default, a thread gets the priority of the thread creating it.
- Priority values are defined between 1 and 10

Thread.MIN\_PRIORITY    (== 1)

Thread.NORM\_PRIORITY    (== 5)

Thread.MAX\_PRIORITY    (== 10)

- Priority can be directly set

```
Thread t = new Thread(new Runnable());  
t.setPriority(Thread.MAX_PRIORITY);  
t.start();
```



# JVM scheduling policy

---

- A thread always runs with a priority number
- The scheduler in most JVMs uses **time-sliced, preemptive, priority-based scheduling**
  - each thread is allocated a fair amount of time, after that it is sent back to runnable to give another thread a chance
- JVM specification does not require a VM to implement a time-slicing scheduler !
  - some JVM may use a scheduler that lets one thread stay running until it completes its run() method

# Checking JVM scheduler

---

```
public class Hamlet implements Runnable {  
    public void run(){  
        while(true)  
            System.out.println(Thread.currentThread().getName());  
    }  
}  
public class TryHamlet {  
    public static void main(String argv[]){  
        Hamlet r = new Hamlet();  
        new Thread(r, "To be").start();  
        new Thread(r, "Not to be").start();  
    }  
}
```

- If the scheduler is non-preemptive the first thread chosen runs forever
- If the scheduler is preemptive both threads randomly alternate



# Leaving the running state (explicitly)

---

- There are 3 ways for a thread to do it:
- `sleep()`: the currently running thread stops executing for at least the specified sleep duration
- `yield()`: the currently running thread moves back to runnable to give room to other threads
- `join()`: the currently running thread stop executing until the thread it joins completes



# sleep()

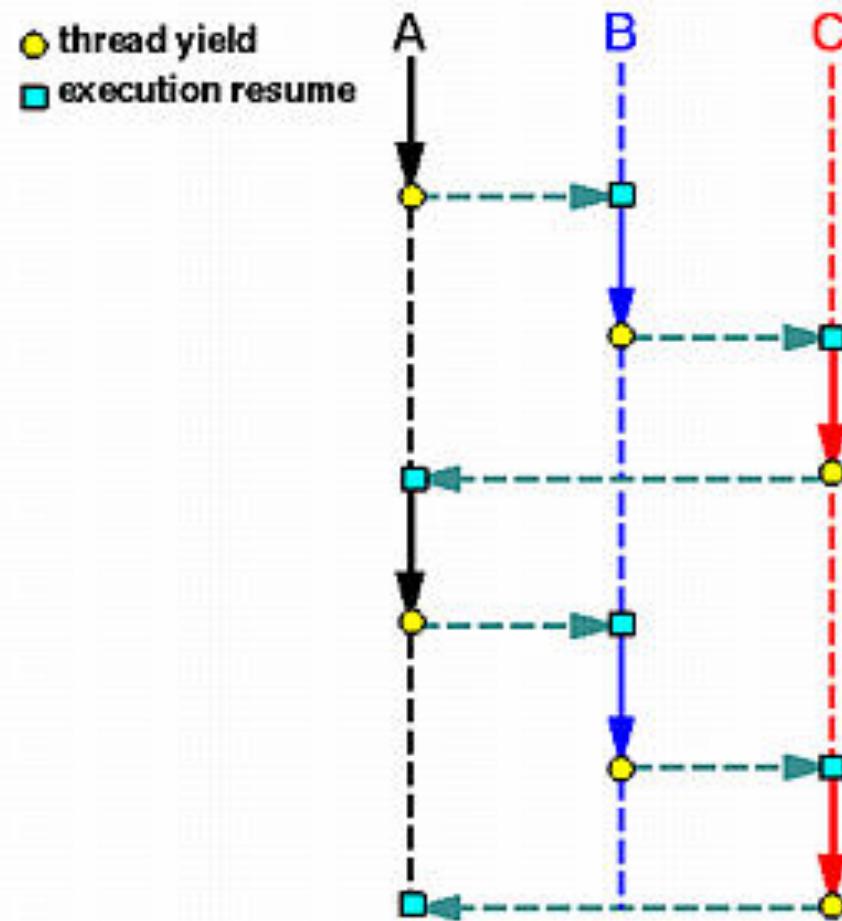
---

```
try {  
    // Sleep for 1 min  
    Thread.sleep(60 * 1000);  
} catch (InterruptedException ex) {  
    //  
}
```



# yield()

- The method `yield()` make the currently running thread back to Runnable state
  - It allows other threads to get their turn
  - However, it might have no effect at all. In fact, there's no guarantee the yielding thread won't be scheduled again for execution.



# yield()

---

- Code is less dependent from the scheduler type, because threads release CPU when needed.
- Frequently used when computation is not possible (no work to do) in a specific time slice.

```
public class Hamlet implements Runnable {  
    public void run() {  
        while (true){  
            System.out.println(Thread.currentThread().getName());  
            Thread.yield(); // allow other thread to run  
        }  
    }  
}
```



# join()

---

- A thread can execute a thread join to wait until the other thread terminates. In general, thread join is for a parent to join with one of its child threads. Thread join has the following activities, assuming that a parent thread P wants to join with one of its child threads C.
  - When P executes a thread join in order to join with C, which is still running, P is suspended until C terminates. Once C terminates, P resumes.
  - When P executes a thread join and C has already terminated, P continues as if no such thread join has ever executed (i.e., join has no effect).

# join()

---

The join() method lets one thread "join onto the end" of another thread.

```
Thread t = new Thread();
t.start();
t.join();
```

Current thread (caller) move to Waiting state and it will be Runnable when thread t is dead. A timeout can be set to wait for a thread's end

```
t.join(5000);
// wait t for 5 seconds: if t is not finished
// then current thread is Runnable again
```



# A word of advice

---

- Some methods may look like they tell another thread to block, but they don't.
- If `t` is a thread object reference, you can write something like: `t.sleep()` or `t.yield()`
- However, **they are static methods of the Thread class:**
  - they don't affect the instance `t` !!!
  - instead they affect the thread which is currently in execution
- **Using an instance variable to access a static method is error-prone!**

