

# Java Threads

---

Università di Modena e Reggio Emilia

*Prof. Nicola Bicocchi (nicola.bicocchi@unimore.it)*



# Process

---

- In operating systems (OS), a process is an instance of a running application
- A process has its own private virtual address space, code, data, opened files, etc..
- A process also might contain one or more threads that run in the context of the process.



# Thread

---

- A thread is the basic entity to which the operating system allocates CPU time. A single process can execute multiple threads.
- All threads of a process share address space, global variables, and resources.
- A thread can execute any part of application's code, including parts currently being executed by other threads.



# Communication

---

- **Processes do not share memory** (separate address spaces), thus they must communicate through IPC mechanisms offered by the operative systems (i.e., pipes, signals)
- Multiple **threads belonging to the same process share the same address space**. Thus, they do need to explicitly communicate. They can use shared variables. **Fast but dangerous!**



# Multitasking

---

- A multitasking operating system assigns CPU time (slices) to threads
- O.S. 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 execution seems to be parallel
- Small time-slices (20ms) provide the **illusion of parallelism** (on multi-core machines it is a partial illusion)



# JVM and Operating System

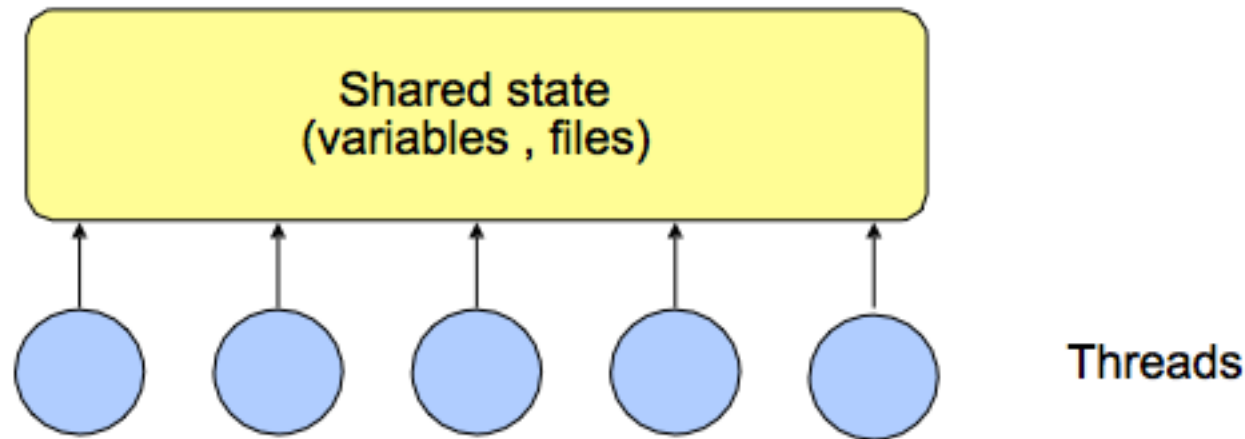
---

- The JVM gets its turn at the CPU by whatever scheduling mechanism the OS uses
- JVM operates like a mini-OS and schedules its own threads
  - Most JVMs use O.S. scheduler (native threads, a Java thread is actually mapped to a system thread)
- Assuming a single processor machine:
  - Only one thread can actually run at a time.
  - The order in which runnable threads are chosen to be THE ONE running is NOT guaranteed.



# What are threads ?

---



- General-purpose solution for managing concurrency.
- Multiple independent execution streams
- Shared state

# What are threads used for ?

---

- **Operating systems**: one kernel thread for each user process.
- **Scientific applications**: one thread per CPU (solve problems more quickly).
- **Distributed systems**: process requests concurrently (overlap I/Os).
- **GUIs**:
  - Threads correspond to user actions; they can help display during long-running computations.  
Multimedia, animations.





# Why threads ?

---

- Imagine a stock-broker application with a lot of complex capabilities:
  - download last stock option prices
  - check prices for warnings
  - analyze historical data for company



# Single-threaded scenario

---

- In a single-threaded runtime environment, these actions execute one after another
  - The next action can happen only when the previous one is finished.
- If a historical analysis takes half an hour, and the user selects to perform a download and check afterward...
  - ...the result may come too late to buy or sell the stock



# Multi-threaded scenario

---

- In a multi-threaded scenario
  - the download can execute in background (i.e. in another thread).
  - other processes could execute at the same time e.g. a notification could be fired instantly, while the user is interacting with other parts of the application (remember your phone?)
  - The analysis, too, can execute in a separate thread, so the user can work in the rest of the application while the results are being calculated.



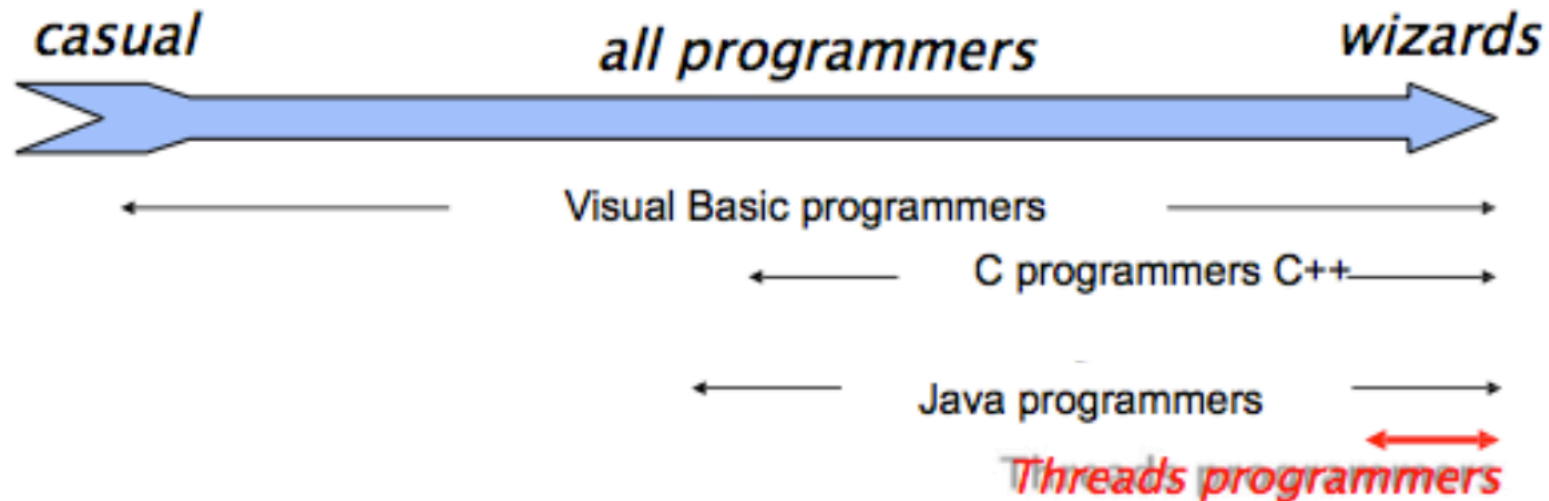
# The good

---

- Enable parallelism
- Lighter than processes (i.e., `fork()`, `pipe()`, ...)



# The bad



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

# How to create a Thread

---

- Threads can be created by **extending Thread** and overriding the `run()` method.
- Thread objects can also be created by passing to the Thread class constructor a **Runnable** object (the target of the thread)
- It is legal to create many Thread objects using the same Runnable object as the target.



# How to create a Thread

---

- Extending Thread class

```
Class T extends Thread {}  
\\ write code into the run() method (override)  
T t = new T();  
t.start();
```

- Implementing Runnable interface (better)

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



# Example: extends Thread

---

```
class Counter extends Thread {  
    private int n;  
    String name;  
    public Counter(String name, int n) {  
        this.name = name;  
        this.n = n;  
    }  
    public void run() {  
        for(int i=0; i<n; ++i)  
            System.out.println("name :" + i);  
    }  
}
```





# Example: implements Runnable

---

```
class Counter2 implements Runnable {  
    private String name;  
    private int n;  
    public Counter(String name, int n) {  
        this.name = name; this.n = n;  
    }  
    public void run() {  
        for(int i=0; i<n; ++i)  
            System.out.println("name :" + i);  
    }  
}
```



# Starting a Thread

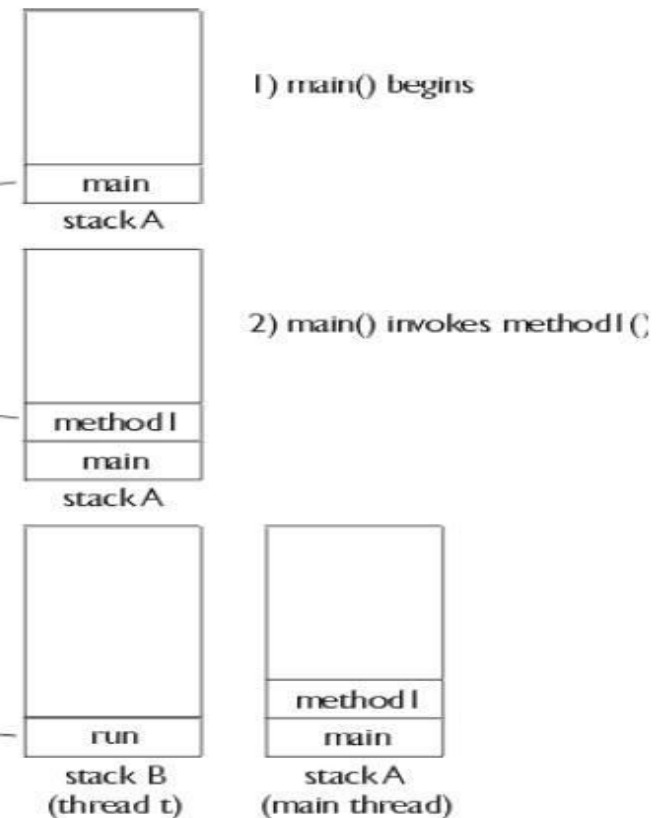
---

- 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

```
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  
}
```



# Running multiple threads

---

```
class Counter implements Runnable {
    public void run() {
        for(int i=0; i<10; i++)
            System.out.println(Thread.currentThread().getName());
    }
}

public class Runner{
    public static void main(String[] args) {
        Thread t1 = new Thread(new Counter());
        Thread t2 = new Thread(new Counter());
        Thread t2 = new Thread(new Counter());
        t1.setName("Abramo"); t2.setName("Luisa"); t3.setName("Elvira");
        t1.start(); t2.start(); t3.start();
    }
}
```



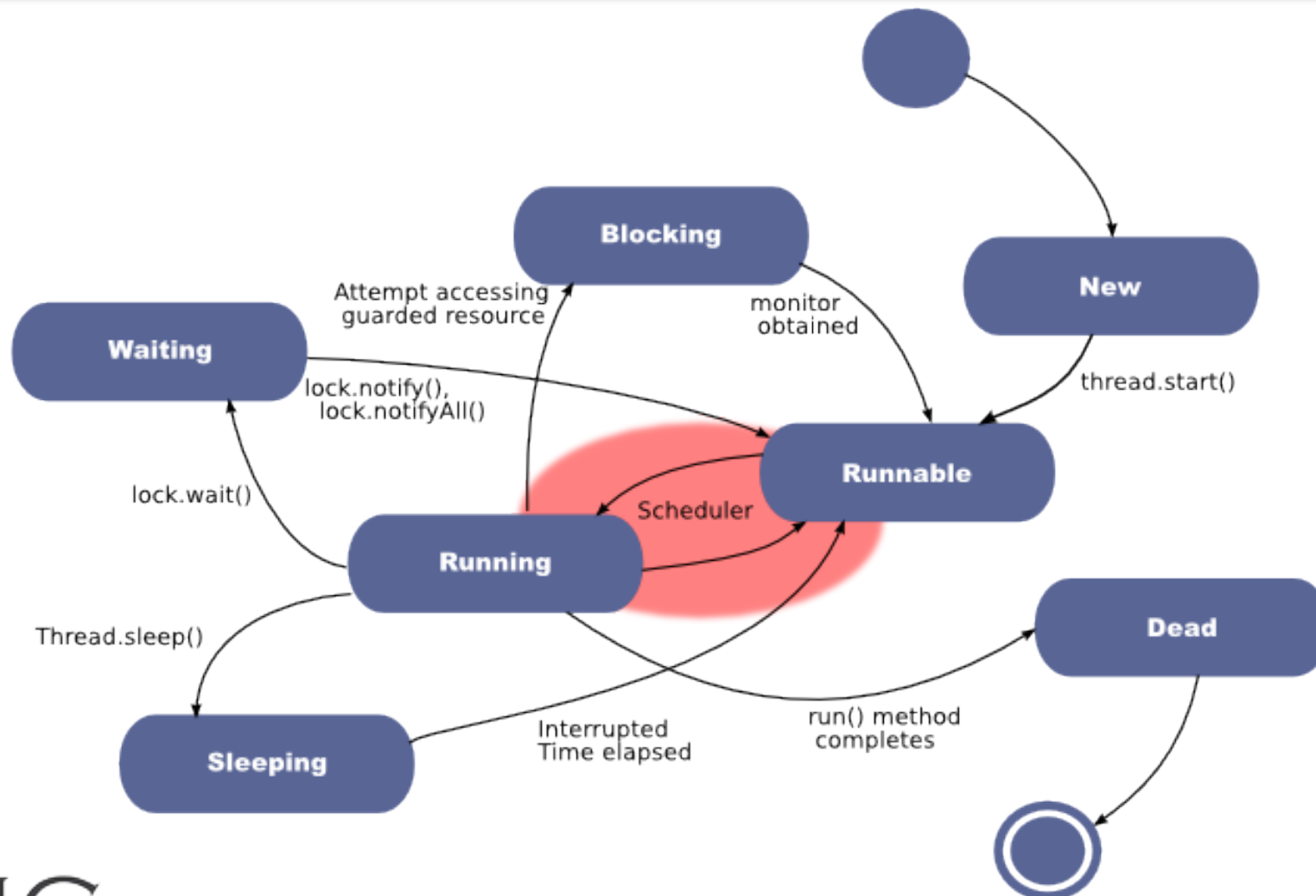
# Running multiple threads

---

- It is not guaranteed that threads will start running in the order they were started
- 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 each thread will run to completion, hopefully.



# 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 process.
- A thread can transition out of a running state for several reasons, including because “the thread scheduler decided it”
- Only one way to get to the running state: the scheduler chooses a thread from the runnable pool.



# Thread state: Runnable

---

- A thread is queued & 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 running or coming back from a blocked, waiting, or sleeping state
- When the thread is in the runnable state, it
- is considered alive





# Thread state: Blocking

---

- This is the state a thread is in when it is NOT eligible to run.
  - It might return to a runnable state later if a particular event occurs.
- A thread may be blocked waiting for a resource ( I/O or an object's lock) e.g.:
  - if data comes in through the input stream the thread code is reading from
  - the object's lock suddenly becomes available.



# Thread state: Sleeping

---

- A thread may be sleeping because the thread's `run()` code tells it to sleep for some period of time,
- Back to Runnable state when it wakes up because its sleep time has expired.

```
try {
```

```
    Thread.sleep(5*60*1000); // Sleep for 5 min
```

```
} catch (InterruptedException ex) { }
```



# Thread state: Waiting

---

- A thread run code causes it to wait
- It comes back to Runnable state when another thread sends a notification
- Used for synchronization
- Note Well: one thread does not tell another thread to block. The OS does!



# 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

```
FooRunnable r = new FooRunnable();
```

```
Thread t = new Thread(r);
```

```
t.setPriority(8);
```

```
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 non-preemptive the thread chosen first run forever and it never releases CPU
- If preemptive threads randomly alternate on output



# Leaving the running state (explicitly)

---

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



# 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
```





# yield()

---

- The method `yield()` make the currently running thread back to Runnable state
  - It allows other threads of the same priority to get their turn (e.g., because computation is terminated)
- `yield()` will cause a thread to go from running to runnable, but it might have no effect at all
  - There's no guarantee the yielding thread won't just be chosen again over all the others!



# yield()

---

- Code is less dependent from the scheduler type, because threads release CPU when needed.
- Used when computation is not possible 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  
        }  
    }  
}
```



# sleep()

---

```
try {  
    // Sleep for 5 min  
    Thread.sleep(5 * 60 * 1000);  
} catch (InterruptedException ex) {  
    //  
}
```



# 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 in execution
- That's why it's a bad idea to use an instance variable to access a static method ;)



# Leaving the running state (implicitly)

---

There are 4 cases when JVM scheduler does it:

- The thread's run() method completes
- The thread scheduler can decide to move the current thread from running to runnable in order to give another thread a chance to run
- Thread calls wait() on an object
- A thread can't acquire the lock on the object

