

Introduction to Multithreading in Java

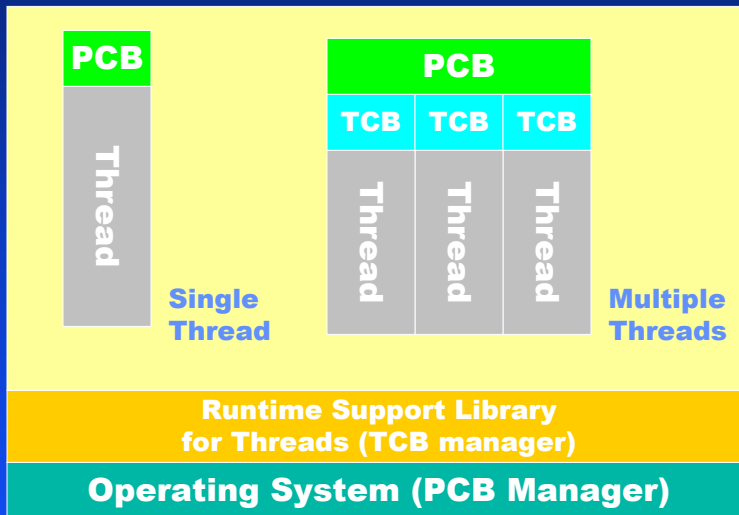
- Alessio Bechini -




Special thanks to A. Vecchio for the previous version of these slides



Multitasking and Multithreading






Introducing Java

Why Are Threads Useful in Java?

- To easily implement algorithms that are intrinsically parallel
- In GUIs, to handle the graphical display on multiple windows
- To enhance the program performances in multiprocessor platforms
- To handle asynchronous actions/tasks required by the program

3



Introducing Java

Use of Asynchronous Actions

- Non-blocking I/O
- Management of timed alarms, timers, etc.
- Tasks to be carried out in an actual concurrent fashion
- Management of multiple service requests with unpredictable arrival time

4

Thread Usage: Example (I)

Sequential case

```
public class MyProg {  
    public static void main() {  
        Hello h = new Hello();  
        h.run();  
        <show MPEG>  
    }  
}  
  
class Hello {  
    public void run() {  
        for(int i=0; i<1000; i++)  
            System.out.println("Hello");  
    }  
}
```

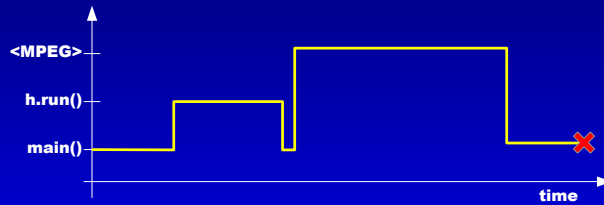
Concurrent case

```
public class MyProg {  
    public static void main() {  
        Hello h = new Hello();  
        h.start();  
        <show MPEG>  
    }  
}  
  
class Hello extends Thread {  
    public void run() {  
        for(int i=0; i<1000; i++)  
            System.out.println("Hello");  
    }  
}
```

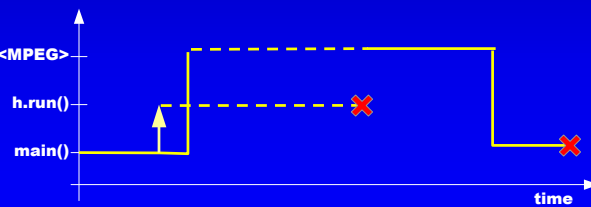
5

Thread Usage: Example (II)

Sequential Case



Concurrent Case



6

Threads by Inheritance/Interfaces

Thread by means of:

- Inheritance
 - The most straightforward way to create a thread is to build it in a sub-class of *java.lang.Thread*
 - Single inheritance prevents this class from reusing code from other classes through a derivation relationship: the thread code cannot fully exploit the advantages of inheritance.
- Interface
 - A possible solution is the creation of threads by classes that implement a particular interface: *java.lang.Runnable*

7

Thread by Interface: Example

```
public class MyProg {  
    public static void main() {  
        Runnable runTarget = new Hello();  
        Thread helloThread = new Thread(runTarget);  
        helloThread.start();  
        <mostra MPEG>  
    }  
}  
  
class Hello implements Runnable {  
    public void run() {  
        for(int i=0; i<1000; i++)  
            System.out.println("Hello");  
    }  
}
```

Target Object of class Hello

Notice that it is used a particular constructor which takes a "Runnable" target as parameter

8



Some Methods of Class Thread

- *currentThread()* returns the currently running thread (static method)
- *join()* waits for the termination of the thread it is invoked on (e.g., *t.join()* waits for the termination of *t*)
- *join(timeout)* a timed variant of the previous one
- *isAlive()* returns true on a thread, in the time interval between its *start()* and its termination
- *suspend()* and *resume()* are used to temporary halt and wake up a thread

9



Control over Shared Resources

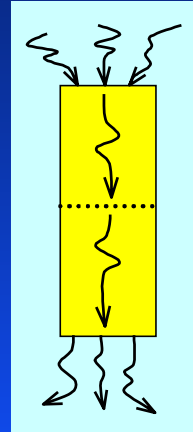
- Two or more concurrent threads may try to use the same resource at a time. This contemporary use may be harmful. Thus, thread collision over a shared resource must be usually prevented.
- A neat solution to this “collision problem” is the following:
- The first thread that accesses a resource locks it, and then the other threads cannot access that resource until it is unlocked, at which time another thread locks and uses it, etc.

10



Synchronization among Threads

- The basic mechanism for thread synchronization is provided by the construct *synchronized*
- The employment of synchronized on a class method, an instance method or a code block assures its execution in mutual exclusion
- The synchronization relies on the use of particular data structures known as "locks"



11



Object Lock and Lock Scope

- A lock is (implicitly) associated to every Java object
- A lock is (implicitly) associated to every Java class
- The synchronized keyword is aimed at grabbing the lock of the object passed as argument (either explicitly or implicitly).
In case the lock isn't currently available (i.e. it has been already acquired), the executing thread blocks until it is allowed to grab the lock
- The "lock scope" is the time interval (or code portion) a thread holds a lock

12



Critical Sections

- A code portion which must be run by one thread at a time is called a *critical section*.
- Java supports critical sections that do not correspond to methods by the *synchronized block*; the keyword **synchronized** is used to specify the object whose lock is being used to synchronize the enclosed code:

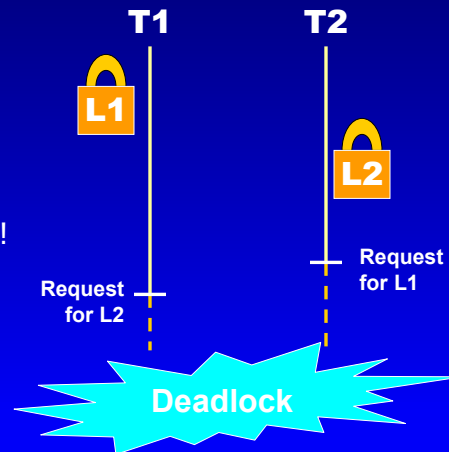
```
synchronized(syncObject) {  
    // This code can be accessed  
    // by only one thread at a time  
}
```

13



Deadlock Situations

BE CAREFUL!
The incautious use
of locks
may lead
to deadlock situations!



14

Wait and Restart in Synchronized Blocks

- Inside a synchronized portion of code, a thread can block itself on a lock by invoking the primitive `wait()`
- A thread which is blocked on a `wait()` can be restarted by another thread by executing the primitives `notify()` / `notifyAll()` on the same lock
- The use of these primitives is quite dangerous: pay particular attention in implementing interaction schemes among threads!

