Sistemi Concorrenti e di Rete LS
Il Facoltà di Ingegneria - Cesena
a.a 2007/2008

[module lab 1.1]
CONCURRENT PROGRAMMING IN JAVA: OVERVIEW
CONCURRENT PROGRAMMING IN JAVA

- Java has been the first mainstream programming language to provide a first native support to concurrent programming
  - “conservative approach”: everything is still an object
  - + mechanisms for concurrency

- Recently extended with the `java.util.concurrent` library to provide a more fine-grained control for synchronization
  - First developed as a Java Specification Request: JSR-166
    - Doug Lea et al.

- An extension with a more coarse-grained support is expected in in Java 7
  - Fork-Join Framework
BASIC MECHANISMS: OVERVIEW

- Class **Thread** (along with a few related utility class) used to initiate and control concurrent activities
  - Runnable interface
- Keyword **synchronized** and **volatile**, used to control the execution of code in objects that may participate in multiple threads
  - for mutual exclusion
- Methods **wait**, **notify**, and **notifyAll** as defined in **java.lang.Object** used to coordinate activities across threads
  - for real synchronization
DEFINING THREADS

• Java provides a basic API for defining new types of thread, and for dynamically create and (partially) manage thread execution
  – threads are mapped onto OS threads, with strategies that depend on the specific system
  – typically a one-to-one approach is adopted
• A thread is represented by the abstract class `Thread`, characterised by the abstract method `run`, which defines the behaviour of the thread
  – a concrete thread can be defined by extending `Thread` class, and implementing the `run` method
• To start thread asynchronous execution, the method `start` is provided
  – must be invoked on the instance of a thread object
  – it returns immediately, and a new activity executing what specified in `run` method is launched
• The thread terminates as soon as the execution of the method `run` body is completed
Thread CLASS

- Thread class is provided in the package java.lang

```java
public class MyWorker extends Thread {
    public MyWorker(String name) {
        super(name);
    }

    public void run() {
        ...  // <active behaviour>
    }
}
```
MAIN THREAD API

- Main features provided by the Thread class:
  - `Thread(String name)`
    - to construct a thread with a specified name
  - `String getName();`
    - get the thread name
  - `void sleep(long ms)`
    - to suspend thread execution for ms milliseconds
  - `void join();`
    - wait for the termination of the thread
  - `void interrupt()`
    - causes a sleep, wait or join to abort with an InterruptedException, which can be caught and deal with in an application-specific way
  - `static Thread currentThread()`
    - to get the reference to current thread in execution
SPAWNING THREADS

```java
public class Test {
    public static void main(String[] args) {
        Thread myWorkerA = new MyWorker("worker-A");
        myWorkerA.start();
        Thread myWorkerB = new MyWorker("worker-B");
        myWorkerB.start();
    }
}
```

• NOTE
  – the method executed on the thread object is `start`, not `run`
    • what if we execute the method `run` instead? what is the behaviour of the program `Test` if we invoke `run` instead of `start` for both the workers?
  – a Java application has always at least one thread in execution
    • it’s the main thread on the application, invoking the static method `main`
MONITORING THREADS: JConsole TOOL

- JConsole is the Java Monitoring and Management Console, a graphical tool shipped in J2SE JDK 5.0 (and later versions)
  - it uses the instrumentation of the Java virtual machine to provide information on performance and resource consumption of applications running on the Java platform
  - based on the Java Management Extension (JMX) technology
- Useful (also) to monitor the thread spawned by a running Java programs
  - including VM threads, such as the one used for garbage collecting
- Usage
  - start the Java application that you want to monitor with the option:
    -Dcom.sun.management.jmxremote
  - Example
    - java -Dcom.sun.management.jmxremote -cp bin sisco.test00.Test
  - from another shell, launch the jconsole tool by simply typing jconsole and connect to the running application
  - it is part of the JRE (Java Runtime Environment)
DEPRECATED

• All the public methods to asynchronously act on the control flow of the thread have been deprecated
  - stop
  - suspend
  - resume
  - destroy
  - ...

• The same functionality is achieved through proper patterns
  - next lab modules
Runnable Interface

- An alternative approach is provided to define a thread, based on Runnable interface, useful when the class used to implement the thread belongs to some class hierarchy
  - ..already extending some class, which is not Thread

```
public class MyWorker implements Runnable {
    public void run(){
        ...
    }
    ...
    Thread th = new Thread(new MyWorker());
    th.start();
```

- Note the Runnable object parameter in Thread constructor
IMPLICIT SYNCHRONIZATION:
synchronized

• By applying the keyword synchronized as a qualifier to any code block within any method, only one thread at a time can obtain access to the object where synchronized is defined
  – prevents arbitrary interleaving of the actions in the method bodies
  > prevents unintended interactions among thread accessing the same objects
• Suggestion
  – to be used in passive objects that are shared and concurrently accessed (for updates) by multiple thread
EXPLICIT SYNCHRONIZATION

- Set of mechanisms used for explicit synchronization among threads, through shared objects
  - `wait` method
    - any synchronized method in any object can contain a `wait`, which suspend the current thread
  - `notifyAll` method
    - all threads waiting on the target object are resumed upon the invocation of the method `notifyAll` on the target object
    - also the `notifyAll` method must be contained in a synchronized method or block
  - `notify` method
    - one (arbitrarily chosen) thread waiting on the target object is resumed upon invocation of method `notify`
    - also the `notify` method must be contained in a synchronized method or block
VADEMECUM: ATOMIC ASSIGNMENTS

• Java guarantees that most primitive operations are atomic and will work always safely in multithreaded context without explicit synchronization.
  – These operations include individual access and assignment to all built-in scalar types (including reference type) except long and double.
  – Without explicit synchronization concurrent assignment to long and double variables are allowed to be interleaved in ways that could result in reading of inconsistent values.
• The specification allows compilers to perform optimizations that caches values of variables in ways that may cause assignments in other thread to be ignored.

>> this can be avoided by marking a variable as volatile
VADEMECUM: SYNCHRONIZED BLOCKS

• In addition to methods, individual code block within any Java method can be synchronized via

```java
synchronized (anyObject) {
    anyCode();
}
```

• Block synchronization is considered to be a more basic construct than method synchronization.
  – A synchronized method is equivalent to one that is not marked as synchronized but has all of its code contained within a synchronized(this) block

• Class level static methods and blocks within static methods may be declared as synchronized.
  – A non-static method can also lock static data via code block enclosed within synchronized(getClass())
VADEMECUM: IMPLICIT SYNCH. AND OBJECT LOCKING

• (Implicit) synchronization is implemented by exclusively accessing the underlying and otherwise inaccessible internal lock (called also *mutex*) that is associated with each Java object (including Class objects for statics).
  – If the count value (of the mutex) is not zero on entry to a synchronized method or block because another thread holds the lock, the current thread is blocked until the count is zero.
  – On entry, the count value is incremented.
  – The count is decremented on exit from each synchronized method or block, even if it is terminated via an exception

• Any method or code block marked as synchronized is executed in its entirely (unless explicitly suspended via wait) before the object is allowed to perform any other synchronized method called from any other thread
VADEMECUM:
SELF-CALL SYNCH METHODS

• Code in one synchronized method may make a self-call to another method in the same object **without blocking**.
  – Similarly for calls on other objects for which current thread has obtained and not yet released the lock.
  – Only those calls stemming from other threads are blocked.
  – Synchronization is retained when calling an unsynchronized method from a synchronized one
VADEMECUM: SYNCHRONIZED & INHERITANCE

• If a method is not marked as synchronized, then it may execute immediately whenever invoked, even while another synchronized method is executing.
  – Thus, declaring a method synchronized is not sufficient to ensure exclusive access: any other unsynchronized methods may run concurrently with it.
• The synchronized qualifier for methods can be overridden in subclasses.
  – A subclass overriding a superclass method must explicitly declare it as synchronized if so desired.
  – Otherwise, it is treated as unsynchronized.
    • In other words, the synchronized qualifier is not automatically inherited.
• Methods declared in Java interfaces cannot be qualified as synchronized.
VADEMECUM: EXPLICIT SYNCHRONIZATION

- Because of the way in which wait sets interact with locks, the methods `wait`, `notify`, and `notifyAll` may be invoked only when the synchronization lock is held on their targets
  - This is normally ensured by using them only within methods or code blocks synchronized on their targets
- Compliance generally cannot be verified at compile time
- Failure to comply causes these operations to throw an `IllegalMonitorStateException` at run time
VADEMECUM: WAIT INVOCATION

- A wait invocation results in the following actions:
  - If the current thread has been interrupted, then the method exits immediately, throwing an `InterruptedException`
    - Otherwise, the current thread is blocked
  - The JVM places the thread in the internal and otherwise inaccessible wait set associated with the target object
  - The synchronization lock for the target object is released, but all other locks held by the thread are retained
    - A full release is obtained even if the lock is re-entrantly held due to nested synchronized calls on the target object
    - Upon later resumption, the lock status is fully restored
VADEMECUM:
NOTIFY INVOCATION

- A `notify` invocation results in the following actions:
  - If one exists, an arbitrarily chosen thread, say T, is removed by
    the JVM from the internal wait set associated with the target
    object.
    - There is no guarantee about which waiting thread will be selected
      when the wait set contains more than one thread
  - T must re-obtain the synchronization lock for the target object,
    which will always cause it to block at least until the thread calling
    `notify` releases the lock.
    - It will continue to block if some other thread obtains the lock first.
  - T is then resumed from the point of its wait
- A `notifyAll` works in the same way as `notify` except that the steps
  occur (in effect, simultaneously) for all threads in the wait set for the
  object.
  - However, because they must acquire the lock, threads continue
    one at a time
VADEMECUM: WAIT INTERRUPTION

- If `Thread.interrupt` is invoked for a thread suspended in a wait, the same notify mechanics apply, except that after re-acquiring the lock, the method throws an `InterruptedException` and the thread's interruption status is set to false.
- If an interrupt and a notify occur at about the same time, there is no guarantee about which action has precedence, so either result is possible
  - Future revisions of JLS may introduce deterministic guarantees about these outcomes.
VADEMECUM:
WAIT-TIMED VERSION

• The timed versions of the wait method, `wait(long msecs)` and `wait(long msecs, int nanosecs)`, take arguments specifying the desired maximum time to remain in the wait set.
• They operate in the same way as the untimed version except that if a wait has not been notified before its time bound, it is released automatically.
• There is no status indication differentiating waits that return via notifications versus time-outs.
  – Counter-intuitively, `wait(0)` and `wait(0,0)` both have the special meaning of being equivalent to an ordinary untimed `wait()`.
• A timed wait may resume an arbitrary amount of time after the requested bound due to thread contention, scheduling policies, and timer granularities.
  – There is no guarantee about granularity. Most JVM implementations have observed response times in the 1-20ms range for arguments less than 1ms.