v1.0
20140420

Programmazione Avanzata e Paradigmi

Ingegneria e Scienze Informatiche - UNIBO
a.a 2014/2015
Lecturer: Alessandro Ricci

[module lab 2.1]

CONCURRENT PROGRAMMING IN JAVA: INTRODUCTION

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
- Extended with the **`java.util.concurrent`** library to provide a higher level support to concurrent programming
  - semaphores, locks, synchronizers, etc
  - *task* frameworks

BASIC MECHANISMS: OVERVIEW

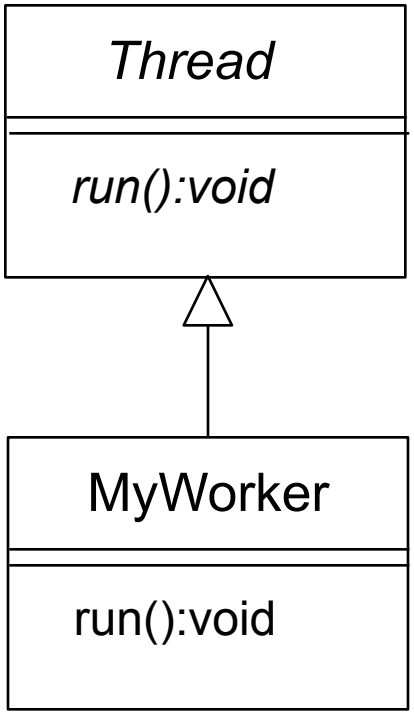
- Class **`Thread`** (along with a few related utility class) used to initiate and control concurrent activities
  - `Runnable` interface
- Keyword **`synchronized`** and **`volatile`**, used to control the execution of code in objects that may participate in multiple threads
  - for mutual exclusion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

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

`Thread` CLASS

- `Thread` class is provided in the package `java.lang`




```
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

```
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

JOINING THREADS

- The join method allows for a thread to synchronize its execution with the termination of an another thread
  - in particular: t.join() suspends the current thread until the thread t has completed its execution
- Example:

```
MyThread t = new MyThread();
t.start();
…
t.join();
System.out.println("spawned thread terminated.");
```

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
  - http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html
- Useful (also) to monitor the thread spawned by a running Java programs
  - including VM threads, such as the one used for garbage collecting

A OPEN-SOURCE PROFILER: VISUALVM

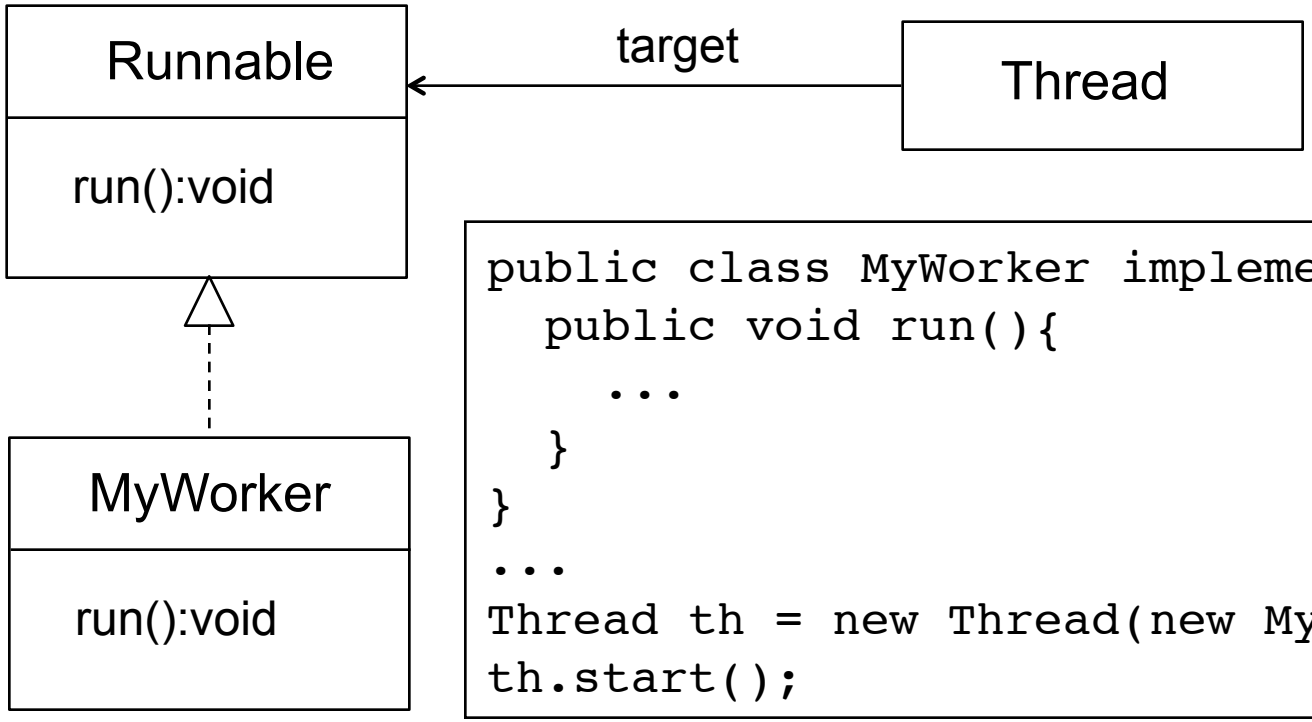
- Similar to JConsole, VisualVm is a full-fledged **profiler** that allow for measuring and visualizing performances of Java programs
  - like JConsole, it uses the instrumentation of the Java virtual machine to provide information on performance and resource consumption of applications running on the Java platform
    - based on the Java Management Extension (JMX) technology
  - shipped with J2SE JDK 5.0 (and later versions)
  - http://visualvm.java.net
- More fine-grained monitoring than JConsole
  - monitoring % CPU used by methods, threads
  - monitoring how long a thread is blocked or running
  - ...

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

A PROGRAMMING DISCIPLINE

- Viewing threads as active objects **encapsulating** state, behaviour and **the control of the behaviour**
  - the object's methods should be called only by the thread represented by the object
    - the use of public methods should be minimized
- Promoting interaction by means of shared (passive) objects
  - not by calling public methods of their interface
    - this would violate encapsulation of control
- Strong conceptual separation between *active* and *passive* entities
  - active entities as agents that are responsible of accomplishing some tasks
  - passive entities as the objects shared and manipulated by such agents in order to accomplish such tasks