Java: Using Threads and Synchronizers

Alessio Bechini Dept. of Information Engineering, Univ. of Pisa

a.bechini@ing.unipi.it

Outline

Understanding how to use threads and, in the Java scenario, what high-level mechanisms help support their synchronization and coordination

- Threads in a language
- JVM Runtime Data Areas (RDA)
- Basic thread handling in Java
- Threads vs Tasks
- Thread pooling and the Executor framework
- Synchronizers: Latches, Semaphores, Barriers, etc.
- Thread-safe Data Structures

Language-level Threading

Defining Threads in a Program

Defining a thread means:

- constructing all the supporting data structures, possibly specifying through them the thread characteristics.
- providing the relative management actions.

All this, in Java, is done through the `Thread` class.

Thread body: sequence of instructions.

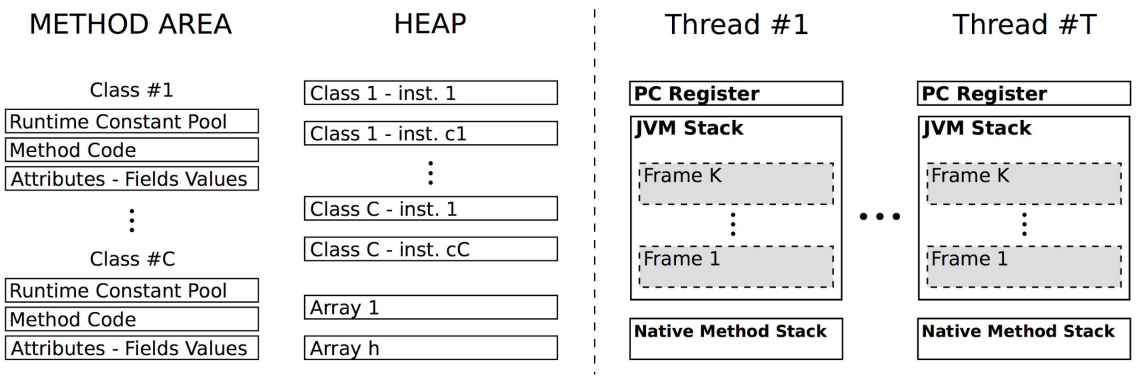
How to define it? Natural choice: via a function/method body.

The body of a Java thread corresponds to a *dedicated method*.

JVM Runtime Data Area

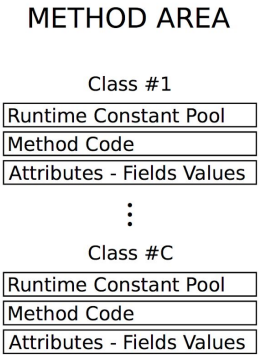
JVM RDA (Runtime Data Areas)

3 distinct parts to host: Bytecode, Objects, and Stacks for Threads



Method Area

Shared among all threads. Holds class-level data of each .class file.

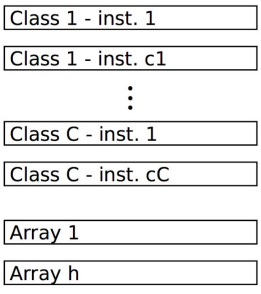


For each class, it contains:

- the Runtime Constant Pool, acting similarly to an ordinary symbol table
- code for methods and constructors
- field and method data

Heap

Shared among all threads. Aimed to host objects of the program.



It contains the representations of all the **instances** of all classes handled in the program, as well as representations of arrays.

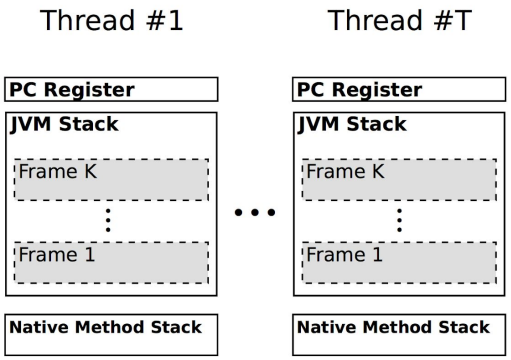
It is subject to garbage collection.

Stack Area

Each Thread has its own stack. Any function call is managed using a Frame in the pertinent stack.

Each frame contains:
- an array of local variables
- its operand stack
  (the JVM is a stack machine!)
- a ref to its runtime const pool

Native methods are supported by “C stacks.”



Basic Thread Handling in Java

Thread Creation and Running

This topic has been already presented in other courses.
Two basic ways:

- **Extension of class `Thread`** - a.k.a. "Threading by subclassing"
  problem: no possibility to extend any other class
- **Use of a `Runnable` instance in the constructor of `Thread`** -
  a more flexible mechanism

This is a "functional" interface,
i.e. with only one abstract method

Explicit thread spawning:
"unblocking" instance method `myhtread.start()`

Thread Definition by `Runnable`

The Runnable object can be defined either via a class that implements Runnable, or via an anonymous class, or via a 0-parameter lambda.

For example:

```
Runnable mytask = () -> {
    String myName = Thread.currentThread().getName();
    System.out.println("This is " + myName);
};

mytask.run();  //the run method can be either invoked...
// ... or used as the body of a thread
Thread thread = new Thread(mytask);
mythread.start();
```

Ordinary method call

Thread spawning

Thread Termination

Some early methods deprecated due to related problems

`stop()` - A stopped thread releases all the locked monitors, so it may lead to state inconsistencies

`suspend()/resume()` - Deadlock-prone: a thread may suspend in a critical section, and another thread able to resume it cannot access the code to do it.

Solution: cooperative mechanism based on *inter-thread interruption*, where one thread asks another to stop.

Inter-Thread Interruption

Each Thread holds an "interrupted" flag to be possibly set by another thread, by invoking the `interrupt()` method on it.

A Thread can check its own interrupted flag via the static method `Thread.interrupted()`, which *clears the flag as well*.

The flag of any thread can be checked via the instance method `isInterrupted()`, which does not modify the flag value.

Typically, a thread that becomes aware of being interrupted is programmed to perform cleanup actions and terminate.

Interruption and Locking

What if an inter-thread interruption is performed to a blocked thread? It depends...

- If it is blocked on an explicit waiting operation, an `InterruptedException` awakes the target thread (and clears the flag).
- If it is blocked on accessing a lock/monitor, it is not awaken; with explicit locks, `lockInterruptibly()` can also be used.

Dealing with InterruptedException

It is particularly important to provide the proper handling of interrupt exceptions.

Guidelines: apart possible cleanup action, we should either:

- re-throw the exception, after a possible re-wrapping
- or restore the interrupted flag by invoking `Thread.currentThread().interrupt()`, so to keep memory of the occurrence of the interruption event

Thread Pooling and the Executor Framework

Task vs Thread, i.e. What vs How

“Thread”: a sequence of instructions that can be executed independently of others in the same program, typically under the control of a scheduler.

Prime concern in concurrent programs: identification of actions/jobs to be carried out, *regardless of the way they will be executed*.
In this cases, the term **“task”** is used for the relative instructions

Practical example: in a server, requests have to trigger tasks; such tasks could be executed using multithreading.

Separating Definition and Execution

A ***task*** can be defined via a `Runnable` class, in the `run()` method.

The task execution could be done in many different ways;
A general solution relies asks for *developing classes aimed at this*.
Such classes have to implement, at least, the following interface:

```
public interface Executor {
    void execute(Runnable command);
}
```


Use of an Executor Class

In a program, specific executor classes can be first defined and then instantiated.

A task can be executed by means of them.

Interface

Concrete class

```
Executor myExecutor = new MyCustomExecutor();
myExecutor.execute(new RunnableTask1());
myExecutor.execute(new RunnableTask2());
```

Two Trivial Examples

A task can be executed as part of the same thread that asks its execution...

```
class DirectExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    }
}
```

Or we can spawn a brand new thread to execute one task!

```
class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
}
```

Of course, none of these looks to be a brilliant solution.

Performance Penalties: Common Causes

Penalties come from wasting time in *management* actions, which can be seen as *overhead* in our program. Possible causes:

- **Context switches** - not only they take time, but also determine cache flushes that slow down the subsequent execution
- **Lock contention** - determines task serialization (no parallelism), hampering scalability: thread waiting should be reduced.
- **Memory Synch** - many threads may need to see last updates. Some unnecessary synch can be removed by the JVM (*lock elision*) or upon static *escape analysis* to spot out thread-local objects.

```
synchronized (new Object()){
    foo();
}
```

Towards Thread Pooling

A thread creation/disposal determines computational overhead.

It is not reasonable to allow an unbounded amount of threads to execute at the same time a large number of tasks, because:

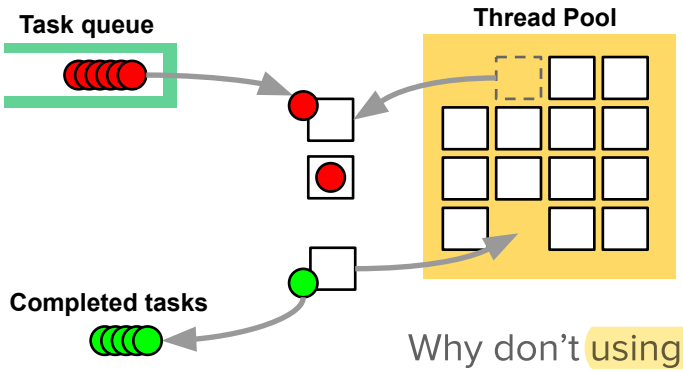
- the number of CPUs/cores is limited
- memory is a limited resource!

This kind of problems are typical of server software.

Idea: Keep some threads ready to execute upcoming tasks and, upon completion of each execution, make the thread available again for another task.

Thread Pooling

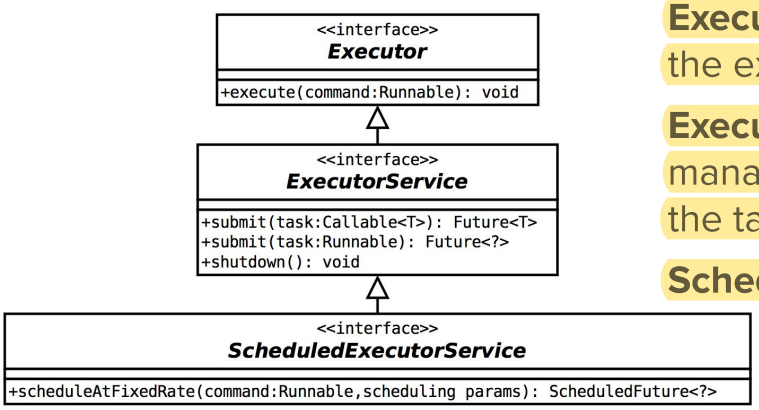
One of the most popular design pattern for multithreaded software.



The pool size can be either static or dynamic, and different policies can be devised for the size update.

Why don't using an Executor for this?

Executor Interfaces



Executor: just support for the execution of a task

ExecutorService: support for managing the lifecycle of both the task and the executor

ScheduledExecutorService: support to timed/periodic execution of tasks.

Separating Submission and Execution

Executors can be designed to *implement an execution policy*, so it is important to tell apart task submission and execution. This is the rationale behind methods in `ExecutorService`.

Executors are a means to separate task submission and execution.

The management of executions can be more complex than expected, as some further aspects must be considered:

- A task result may need to be explicitly managed
- Synchronization for getting the result may be required

Tasks: Beyond Runnable

Whenever a result for a task has to be managed, the task has to be implemented according to the `Callable` interface:

```
public interface Callable<V> {
   V call() throws Exception;
}
```

How to retrieve the returned value? This is not a plain invocation!

The solution is in the way the task is submitted: `submit()` returns back a **`Future<V>`**, which in turn will include the result as it will be computed by the task (and accessible via `get()`).

Futures and Promises

A **future** is a general thread-safe construct adopted in concurrent languages, referring *a proxy for a result computed asynchronously*. The value for a future is allowed to be written only once.

Sometimes the function in charge of setting the value for a future is named **promise**.

Different possible promises may set the value of a given future, though this can be done only once for a given future.

Java Futures

The behavior of a “future” is described by the relative interface:

```
public interface Future<V> {
    boolean isDone();
    V get(); //also a timed version exists
    boolean cancel(boolean m);
    boolean isCancelled();
}
```

It is possible to check if a result is ready (isDone()), retrieve it (get()), try to cancel a task execution, check if a task has been cancelled.

Off-the-shelf ExecutorServices

`java.util.concurrent.Executors` has factory methods to create instances of executors with pooled threads, e.g.:

`newFixedThreadPool(int n)` creates a pool of n threads, with fixed size

`newCachedThreadPool()` provides a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available

`newWorkStealingPool(int parallelism)` creates a pool with dynamically managed threads, able to handle the specified level of parallelism; the task execution order does not necessarily reflect the order of submission.

SheduledExecutorService

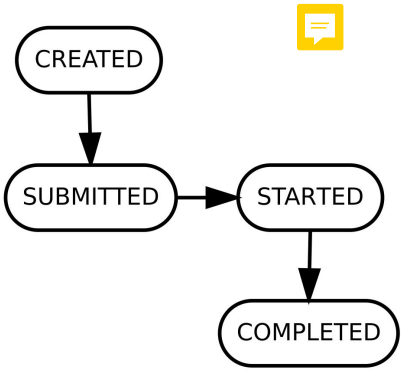
Interface for an ExecutorService that can schedule commands to run after a given delay, or to execute periodically.

Scheduling options: see API documentation

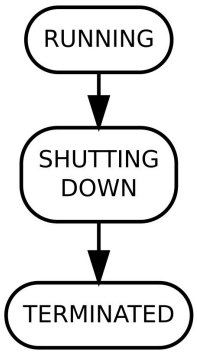
Standard implementation returned by factory method in “Executors”:

`newScheduledThreadPool(int n)` creates a pool of n threads, to be scheduled (often used with n=1)

Lifecycle of Tasks and ExecutorService



Lifecycle of a Task



Lifecycle of an ExecutorService

Stopping Executors

If we want to shut down an executor, we need to take care:
the relative threads *might possibly be still running at that time*.

Distinct operations, as methods of `ExecutorService`:

- `shutdown()` starts the shutdown procedure, previously submitted tasks are completed, and no new task will be accepted.
- `shutdownNow()` - only *currently executing* tasks are completed.
- Check state by using `isShutdown()`;
  `isTerminated()` checks if all tasks have been terminated.
- Methods above are not-blocking;
  to wait for completion of all tasks, call `awaitTermination()`

Java Synchronizers

Synchronizers in Java

Cooperation across threads often requires specific synchronization actions, to be performed according to a given pattern.

A construct designed to address a specific synchronization pattern is called “Synchronizer,” and it coordinates control flow of threads based on its internal state.

Java provides several classes for common special-purpose synchronization. Among them:

Latches, Barriers, Semaphores, Exchangers

Latches

A **latch** is aimed at making a group of threads wait

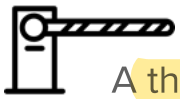
until it reaches its terminal state: ➔ all threads pass through.

An instance of a latch can be used only once.

In Java, the `CountDownLatch` class is present: the constructor takes a positive integer to initialize the state (“count” var).
State: updated by `countDown()` (count--); read by `getCount()`

A thread waits until the release point in time by invoking `await()`
- a timed version is available as well.

Barriers



A **barrier** makes a group of threads meet at a barrier point.
A thread that gets to the barrier waits for all the others to come.
As the last thread arrives, all threads are released to proceed.

The reference Java implementation is `CyclicBarrier`.
A `CyclicBarrier` object can be reused
upon invoking `reset()` on it.

A thread waits until the release point in time by invoking `await()`
- a timed version is available as well.

An optional “barrier action” can be specified as well.

Phasers

A **phaser** is a more sophisticated version of a barrier/latch.

It is a reusable synchronizer.

Differently from CyclicBarrier, the number of parties can change over time: it is designed to operate in successive phases.

For details, refer to the API documentation.

Semaphores

A **semaphore** is aimed to control the number of accesses to a resource. It thus supports a generalization of mutual exclusion.

In Java, the `Semaphore` class implements this synchronizer.

The number of accesses is controlled by using **permits**; the maximum number of available permits is set at initialization time.

`acquire()` is used to get a permit; if it is not available, it blocks.

`release()` is used to give a permit back to the semaphore.

The corresponding classical names are `P()`/`V()` or `down()`/`up()`

Exchangers

An exchanger is aimed at making two threads wait for each other at a synchronization point, and swap objects.

In Java, it is implemented by the `Exchanger<V>` class.

V is the type of the object to be exchanged.

Example of code for an exchange operation:

```
myBuffer = exchanger.exchange(myBuffer);
```

Thread-safe Data Structures

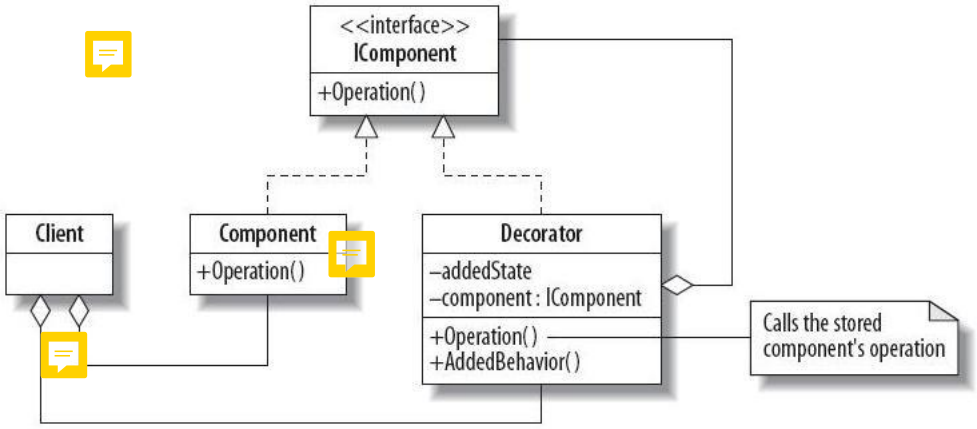
“Synchronized” Collections

A first, trivial attempt to define a thread-safe collection is to make its methods “synchronized.” This can be done in a systematic way by using the Decorator pattern.

A standard way to decorate an ordinary collection can let us obtain a corresponding thread-safe version.

ATTENTION! methods like `isEmpty()` or `size()` are not really meaningful in a concurrent setting, as the returned value might not be necessarily accurate just at the subsequent operation.

The Decorator Pattern



Getting a Synchronized Collection

Specific static factory methods in class `java.util.Collections` of the form `.synchronizedXXX(XXX<T> ds)` return synchronized versions of the plain data structure `ds`.

`XXX` stands for the interface `Collection`, or for `List`, `Map`, etc..

Iteration on synchronized collections needs explicit synchronization *on the whole data structure*, e.g.

```
Collection<T> c = Collections.synchronizedCollection(myCol);
synchronized(c) {
    for (T item : c) foo(item);
}
```

Addressing Performance

A “synchronized” collection shows poor performance *under high contention*.

Main reason: coarse-grained synchronization level.

Solution:
reduction of synchronization overhead, applying *any possible trick*.

“Concurrent” versions of “Synchronized” collections have been developed: e.g.,
`ConcurrentHashMap` is an improvement of `SynchronizedMap`

ConcurrentHashMap

It implements the ConcurrentMap<K,V> interface, which includes also atomic read+write operations, e.g.:

`boolean replace(K key, V oldValue, V newValue)`
- equivalent to the atomic execution of

```
if (map.containsKey(key) && Objects.equals(map.get(key), oldValue)){
   map.put(key, newValue);
   return true;
} else return false;
```

Null-safe equals, since Java 1.7

Iteration over Concurrent Collections

Iterating over a concurrent collection can show problems, because other threads may access the data structure at the same time.

The same problems arise with “for-each” loops, which are compiled to the corresponding iterator-based form.

It is necessary to specify new semantics for a concurrent iterator, so to fully describe its behavior in case of possible interferences.

Fail-fast Iterators

Provided by most of non-concurrent collections.

Used to traverse the *actual* collection: no modification to the collection is allowed.

As soon as a modification is detected (on a best effort basis), `ConcurrentModificationException` is thrown.

Modifications are detected by inspecting specific counters.

Note: it is safe to remove an element by the `remove()` method of the iterator - not the `remove()` of the collection!

Weakly-consistent Iterators

Called also “fail safe.” Provided by most of concurrent collections.

According to the official docs, such iterators:

- may proceed concurrently with other operations
- will never throw `ConcurrentModificationException`
- are guaranteed to traverse elements as they existed upon construction exactly once, and may (but are not guaranteed to) reflect any modifications subsequent to construction.

Snapshot Iterators

Used with copy-on-write collections like `CopyOnWriteArrayList`.

The iterator provides a snapshot of the state of the collection when the iterator was constructed.

No synchronization is needed while traversing the iterator.

The iterator does NOT support the `remove()` method.

In such collections, all “mutative operations” are implemented by making a fresh copy of the underlying data structure, i.e. *Copy-On-Write*.

Blocking Queues

High-performance, thread-safe version of *bounded buffer*, designed according to the `BlockingQueue<E>` interface.

Methods are present to support insertion, extraction, and inspection in different fashions (blocking, throwing exceptions, polled, timed): see API documentation.

Blocking queues support the producers/consumers pattern.

Blocking queues make resource management more reliable.

Impl. of the classic bounded buffer: `ArrayBlockingQueue<E>`