Java course - IAG0040

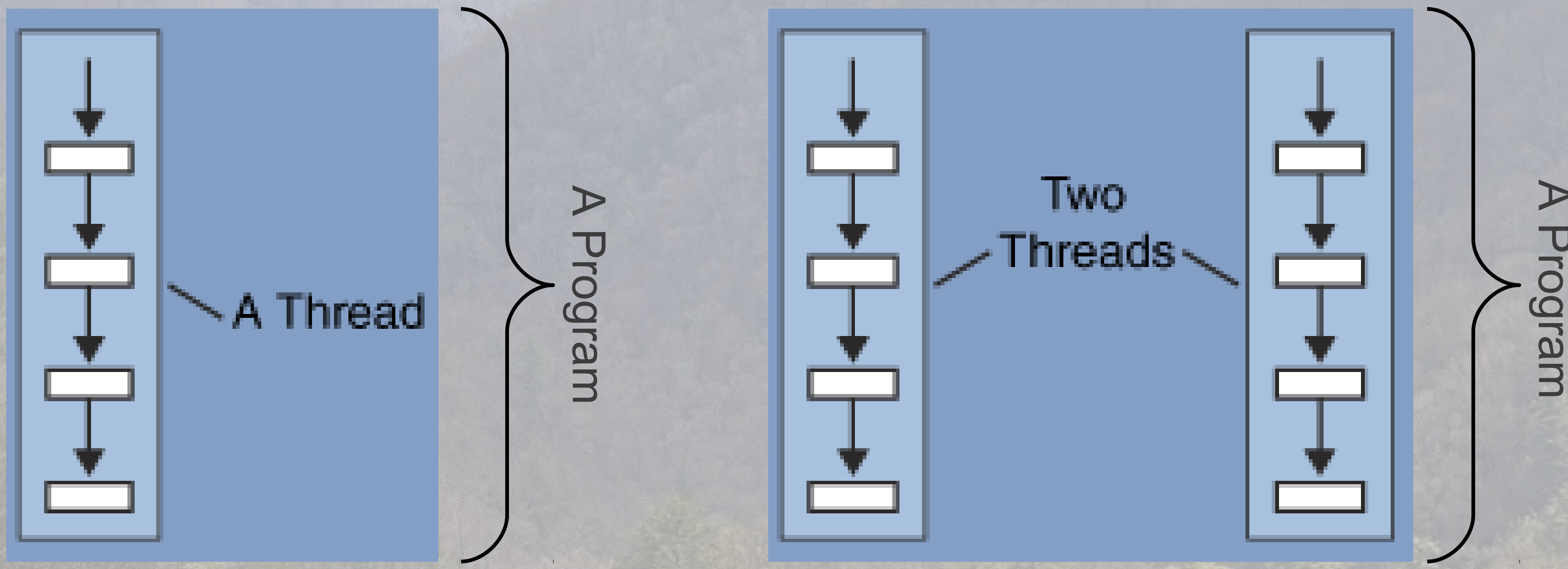
Threads & Concurrency

Anton Keks

2011

Introduction to Threads

- A *Thread* is a lightweight process
- A *Thread* is a single sequential flow of control within a program



More formal definition

- A **thread** is pure activity, i.e. the information about the current position of execution (instruction pointer), the registers and the current stack
- An **address space** is a range of addressable memory that is only accessible by activities that are allowed to read/write from it
- A **process** consists of an address space and one or multiple threads

Threads & Java

- Java supports Threads and synchronization natively
- Threads can be used for doing two or more tasks at once
- main() method is always executed in the “main” thread
  - there are always several more system threads running in your Java program, e.g. garbage collector
- You can always start as many new threads as your program needs using the *Thread* class

Thread class

- Threads run *Runnable* tasks, there are two possible implementations
  - extend the *Thread* class itself, overriding the run method:
    ```
    class MegaThread extends Thread { ... }
    Thread t = new MegaThread();
    ```
  - implement the *Runnable* interface and pass it to a new *Thread*:
    ```
    class MegaRunnable implements Runnable { ... }
    Thread t = new Thread(new MegaRunnable());
    ```
- Every *Thread* has a name: it can be specified on creation or using the `setName()` method
- *Thread* class provides some useful static methods:
  - `currentThread()` returns the current Thread's instance
  - `sleep()` pauses the current Thread for the given time
  - `yield()` allows other Threads to run by pausing the current one

Thread class (cont)

- `start()` starts the new Thread in the background, note: `run()` method doesn't start the Thread!
- `join()` waits for another Thread to die
- `setDaemon(true)` sets the daemon flag. JVM terminates if there are only daemon threads left running.
- `getId()` returns a unique Id (long) of the Thread
- `isAlive()` tests if the Thread is alive (started and not dead)
- `getState()` provides more information on the state
- There are many other interesting `is/get` methods, see Javadoc

Exception handling

- Each thread has a default top-level exception handler
  - that calls `printStackTrace()`
- Just like the **main** thread, where the `main()` method is executed
- If an exception is thrown out of a thread, the thread **is terminated**

Interrupting Threads

- There are many deprecated methods in the Thread class, which have been proved to be dangerous (may cause deadlocks, etc): `suspend()`, `resume()`, `stop()`, `destroy()`, etc
- Instead, `interrupt()` can be used
  - it interrupts blocking/waiting method calls, e.g. `sleep()`, `wait()`, `join()`, etc, with an `InterruptedException`
  - it signals the Thread that it should terminate
    - interrupted flag can be checked using either the static `Thread.interrupted()` or `isInterrupted()` method
  - it is a safe and graceful way to interrupt a Thread, because it involves interaction from the Thread itself

Scheduling

- Scheduling is execution of multiple threads on a single CPU, dividing available CPU time into **time slices**
- Multiple threads are the only possibility for a program to use multiple CPUs / cores (if they are available)
- Each Thread has a priority between MIN_PRIORITY and MAX_PRIORITY (currently from 1 to 10)
- Java implements **fixed-priority preemptive scheduling**
  - at any given time, a Thread with the highest priority is running, but this is not guaranteed: lower-priority threads may be executed to avoid starvation
  - if any higher-priority Thread appears, it is executed pausing others

Synchronization

- Independent and asynchronous Threads are fine without synchronization
- However, access to shared data must be synchronized
  - thread scheduling is unpredictable: threads may access (read-modify-write) variables in any order
- Synchronization prevents interruption of critical regions in the code
- Dijkstra's *Dining Philosophers Problem* is often used for illustration of various synchronization/concurrency problems

Synchronization (cont)

- Java provides the **synchronized** keyword
  - it is used to obtain a lock on an arbitrary object instance
  - you can declare **synchronized** blocks of code:
    `synchronized (lockObject) { ... }`
  - code within the **synchronized** block is accessed atomically
  - you can also declare methods as **synchronized** - the lock is obtained on the instance of the object or the Class object if method is **static**

Synchronization (cont)

- Object class provides `wait()` and `notify()` methods
  - both require acquiring a lock on the object first (**synchronized**)
  - they can be used for signaling to different parts in the code
  - `wait()` can have a timeout and may be interrupted
- Java 1.5 provides many exciting new features for concurrent programming in the **java.util.concurrent** package
  - *Lock and ReadWriteLock* implementations (e.g. *ReentrantLock), Semaphore,* etc provide additional features and sometimes better scalability
  - ```
    ReentrantLock rlock = new ReentrantLock();
    rlock.lock();
    try { ... } finally { rlock.unlock(); }
    ```
    ```

Thread safety

- Rule #1: Document thread safety in Javadoc!!!
- Many classes or methods can have various levels of thread safety
  - **Immutable** – always thread-safe, because its state never changes
  - **Thread-safe** – can be used in multiple threads concurrently (either synchronization is not needed or it is synchronized internally), this is a rather strict requirement
  - **Conditionally thread-safe** – each individual operation may be thread safe, but certain sequences of operations may require external synchronization, e.g. *Iterator* returned by *Vector* and *Hashtable*
  - **Thread compatible** – not thread safe, but thread safety can be achieved by external synchronization of method calls, e.g. *StringBuilder*, all Collections, etc
  - **Thread hostile** – external synchronization of individual method calls will not make an object thread safe, this is rare and can be a result of bad design

Timers and Tasks

- *Timer* and *TimerTask* (both in **java.util**) can be used for conveniently scheduling task execution at intervals or with delays
- Extend the abstract *TimerTask* to create your task (implement the run() method)
- Use a *Timer* for scheduling this task arbitrarily. Every instance of a *Timer* represents a single Thread.
  - `schedule()` - schedules a task for either single or periodical execution. Counting to the next execution begins after the previous one is finished.
  - `scheduleAtFixedRate()` - schedules with the exact period. Counting to the next execution is not affected by the previous one.
  - any Timer can have many TimerTasks scheduled

ThreadLocal

- *java.lang.ThreadLocal* can be used for storing independent data for each Thread (thread-local variables)
  - All Threads share the same *ThreadLocal* instance, but every *Thread* only 'sees' data belonging to it
  - Can be used for storing such things, as User ID, Transaction ID, etc which are handled by current *Thread*
- *ThreadLocal* methods
  - `set(...)` - sets the thread-local variable
  - `get()` - retrieves the previously set thread-local variable

More on java.util.concurrent

- Introduced in Java 1.5, besides higher-level synchronization classes it provides many useful functionality for reduced programming effort, more performance, reliability, maintainability, and productivity
- Task scheduling framework: the *Executor* framework standardizes scheduling, invocation, execution, and control of asynchronous *Runnable* tasks (implementation includes thread pools)
- Concurrent Collections: some new implementations of Map, List, and Queue
- Atomic variables: **atomic** subpackage provides classes for atomically manipulating of variables, e.g. *AtomicInteger* (for higher performance)
- Synchronizers and Locks: more general purpose implementations with timeouts, reader/writer differentiation, non-nested scoping, etc
- Nanosecond-granularity timing, e.g. `System.nanotime();`