Core Java

15. Multi-threaded Programming
Processes and Threads

- Distinction is very clear in case of non-interpreted programs
- Threads are lightweight processes
- Processes don't share memory, although they can communicate with each other through separate shared memories/other IPC mechanisms
- Threads share common segments, except execution stack
Why Threads?

• Most programs are straightforward, single path of execution; But due to programming difficulty, some issues can never be solved by single-threaded execution unless using a large amount of state variables

• Single-threading not useful when you want to do other things in the background

• Multi-threading is the next big advancement in computers, and a lot of speed improvements in the future would benefit from parallelization
Threads

• Implementing a thread in Java is done by extending the `java.lang.Thread` class.

• It is also possible to wrap a new thread around a reference to an object of a class implementing the `java.langRunnable` interface.

• In both cases, the `run()` method is overridden.

• Parameters are set through the Constructor or equivalent Setter methods.
public class MyThread extends Thread {
    public void run() {
        /* code here*/
    }
}

public class MyTask implements Runnable {
    public void run() {
        /* code here*/
    }
}
Threads

Thread myth=new MyThread();
Thread mytask=new Thread(new MyTask());
myth.start();
mytask.start();
myth.join();
mytask.join();
Thread Operation

- The `run()` method gets invoked concurrently when you fire the `start()` method.
- The `join()` method is used to wait for a thread to finish running.
- Execution can be prioritized by `setPriority()`.
- `Thread.currentThread()` returns the `Thread` which is handling the current object.
- `main()` is executed by the `Thread` “Main”.
Thread Operation

- The `run()` method gets invoked concurrently when you fire the `start()` method.
- The `join()` method is used to wait for a thread to finish running.
- `Thread.currentThread()` returns the `Thread` which is handling the current object.
- `main()` is executed by the `Thread` “Main”
Thread Operation

• The *stop()*, *resume()* and *suspend()* methods are deprecated because they were deadlock prone

• If the *Thread* has a loop and must be paused/resumed/stopped, solution is to keep variables which can be polled, and perform little amount of work within the loop, and handling the interruption

• The *Thread* could *wait()* and be *notify()*ed by the object which owns the thread.
Thread Interruption

- *Threads* could be interrupted by calling a thread's *interrupt()* method

- A *Thread* could *sleep()* for a while, during which it may be interrupted too

- A *Thread* may additionally wish to *yield()*; but that is not an interruption
Thread Safety

- Threads must be synchronized on objects they act, so the object isn't affected adversely.
- Commonly a problem of shared variables.
- Use `synchronized(object) { }` blocks in `Thread`.
- Alternative is to keep `synchronized` methods in objects (easier), but all updates to it must only be done through its methods.
Thread Safety

• As a good practice, all methods called by threads must be re-entrant
• Avoid using global/shared variables in Threads as much as possible
• If using shared variables, put appropriate synchronizations on them
• When passing shared references, make sure objects are immutable or accesses to them are synchronized
Demonstration

• Compile and Execute a few programs
Questions?