Basics: All about Java threads

BY CHAITANYA SINGH | FILED UNDER: MULTITHREADING

What are Java Threads?

A thread is a:

- Facility to allow multiple activities within a single process
- Referred as lightweight process
- A thread is a series of executed statements
- Each thread has its own program counter, stack and local variables
- A thread is a nested sequence of method calls
- Its shares memory, files and per-process state

Read: Multithreading in Java

Whats the need of a thread or why we use Threads?

- To perform asynchronous or background processing
- Increases the responsiveness of GUI applications
- Take advantage of multiprocessor systems
- Simplify program logic when there are multiple independent entities

What happens when a thread is invoked?

When a thread is invoked, there will be two paths of execution. One path will execute the thread and the other path will follow the statement after the thread invocation. There will be a separate stack and memory space for each thread.

Risk Factor

- Proper co-ordination is required between threads accessing common variables [use of synchronized and volatile] for consistence view of data
- overuse of java threads can be hazardous to program's performance and its maintainability.

Threads in Java

Java threads facility and API is deceptively simple:
Every java program creates at least one thread [main() thread]. Additional threads are created through the Thread constructor or by instantiating classes that extend the Thread class.

Thread creation in Java

Thread implementation in java can be achieved in two ways:

1. Extending the java.lang.Thread class
2. Implementing the java.lang.Runnable Interface

Note: The Thread and Runnable are available in the java.lang.* package

1) By extending thread class

- The class should extend Java Thread class.
- The class should override the run() method.
- The functionality that is expected by the Thread to be executed is written in the run() method.

void start(): Creates a new thread and makes it runnable.
void run(): The new thread begins its life inside this method.

Example:

```
public class MyThread extends Thread {
   public void run(){
    System.out.println("thread is running...");
  }
   public static void main(String[] args) {
     MyThread obj = new MyThread();
     obj.start();
}
```

2) By Implementing Runnable interface

- The class should implement the Runnable interface
- The class should implement the run() method in the Runnable interface
- The functionality that is expected by the Thread to be executed is put in the run() method

Example:

```
public class MyThread implements Runnable {
   public void run(){
     System.out.println("thread is running..");
   }
   public static void main(String[] args) {
     Thread t = new Thread(new MyThread());
     t.start();
}
```

Extends Thread class vs Implements Runnable Interface?

- Extending the Thread class will make your class unable to extend other classes, because of the single inheritance feature in JAVA. However, this will give you a simpler code structure. If you implement Runnable, you can gain better object-oriented design and consistency and also avoid the single inheritance problems.
- If you just want to achieve basic functionality of a thread you can simply implement Runnable interface and override run() method. But if you want to do something serious with thread object as it has other methods like suspend(), resume(), ..etc which are not available in Runnable interface then you may prefer to extend the Thread class.

Thread life cycle in java

Read full article at: Thread life cycle in java

Ending Thread

A Thread ends due to the following reasons:

- The thread ends when it comes when the run() method finishes its execution.
- When the thread throws an Exception or Error that is not being caught in the program.
- Java program completes or ends.
- Another thread calls stop() methods.

Synchronization of Threads

- In many cases concurrently running threads share data and two threads try to do operations on the same variables at the same time. This often results in corrupt data as two threads try to operate on the same data.
- A popular solution is to provide some kind of lock primitive. Only one thread can acquire a particular lock at any particular time. This can be achieved by using a keyword “synchronized” .
- By using the synchronize only one thread can access the method at a time and a second call will be blocked until the first call returns or wait() is called inside the synchronized method.

Deadlock

Whenever there is multiple processes contending for exclusive access to multiple locks, there is the possibility of deadlock. A set of processes or threads is said to be deadlocked when each is waiting for an action that only

one of the others can perform.
In Order to avoid deadlock, one should ensure that when you acquire multiple locks, you always acquire the locks in the same order in all threads.

Guidelines for synchronization

- Keep blocks short. Synchronized blocks should be short — as short as possible while still protecting the integrity of related data operations.
- Don’t block. Don’t ever call a method that might block, such as InputStream.read(), inside a synchronized block or method.
- Don’t invoke methods on other objects while holding a lock. This may sound extreme, but it eliminates the most common source of deadlock.