3. Threads in Java. Runnable. Creating and using threads.

IT Learning &
Outsourcing Center

www.pragmatic.bg

Lector: Stefan Vadev
Skype: stefanvadev77
E-mail: stefan.vadev@gmail.com
Facebook:
https://www.facebook.com/stefan.vadev.7

Table of contents

- Basics – what is multithreading, when to use it , why use it
- Introduction to threads
- Runnable
- Thread synchronization
- Managing multiple threads

Concurrency

- Computer users take it for granted that their systems can do more than one thing at a time. They assume that they can continue to work in a word processor, while other applications download files, manage the print queue, and stream audio.
- Even a single application is often expected to do more than one thing at a time.
- For example, that streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display.
- Even the word processor should always be ready to respond to keyboard and mouse events, no matter how busy it is reformatting text or updating the display.

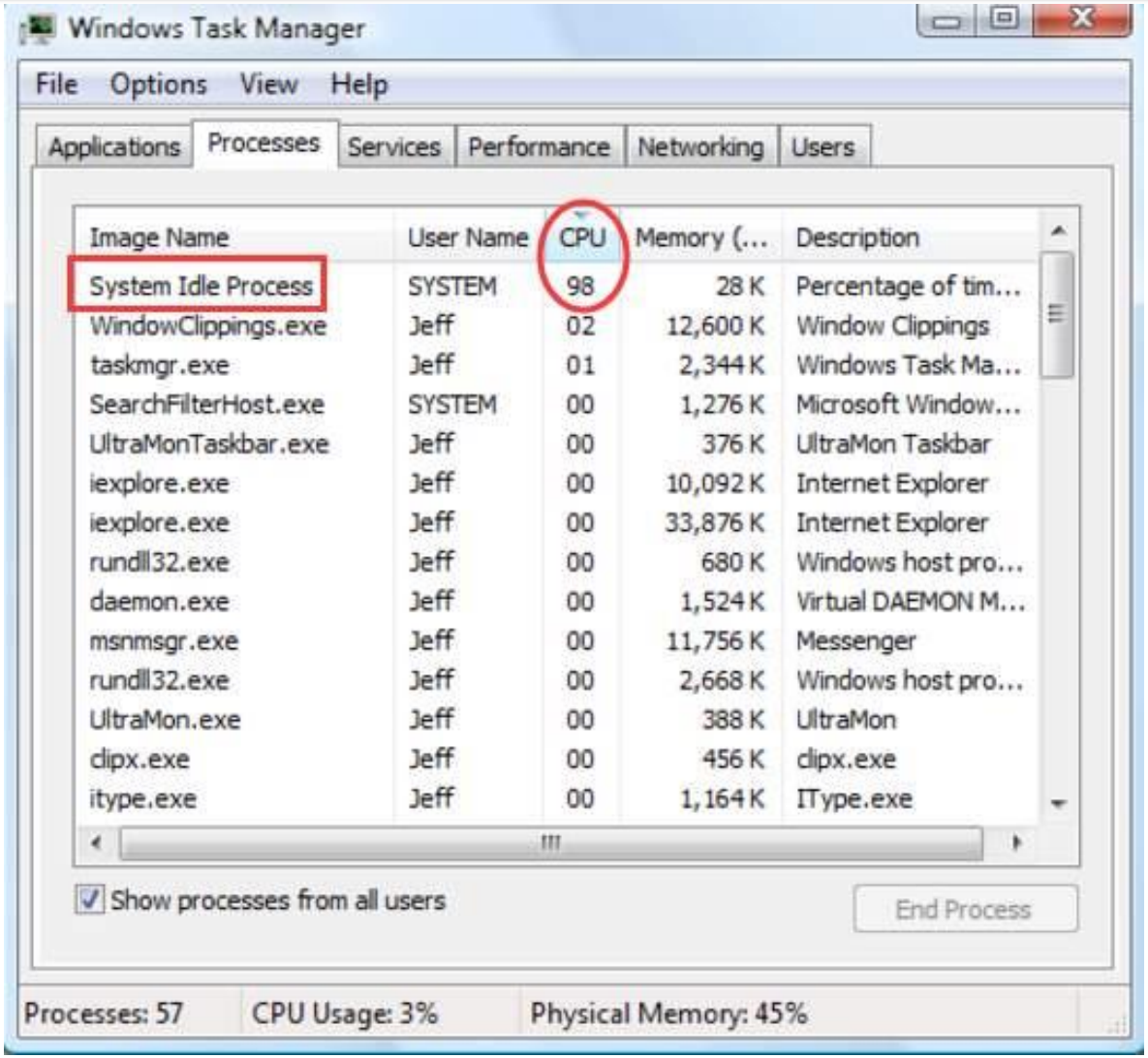
Processes and Threads

In concurrent programming, there are two basic units of execution: *processes* and *threads*. In the Java programming language, concurrent programming is mostly concerned with threads.

A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called time slicing.

It's becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores. This greatly enhances a system's capacity for concurrent execution of processes and threads.

Processes



Process

- Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support *Inter Process Communication* (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.

- Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a **ProcessBuilder** object.

Process Explorer

- Process Explorer Demo

Threads

- Let’s have a quick look at the Thread’s methods:

https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

Threads

- Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.
- Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Threads

- Multithreaded execution is an essential feature of the Java platform.
- Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling.
- But from the application programmer's point of view, you start with just one thread, called the *main thread*.

Threads

- Each thread is associated with an instance of the class Thread. There are two basic strategies for using Thread objects to create a concurrent application.

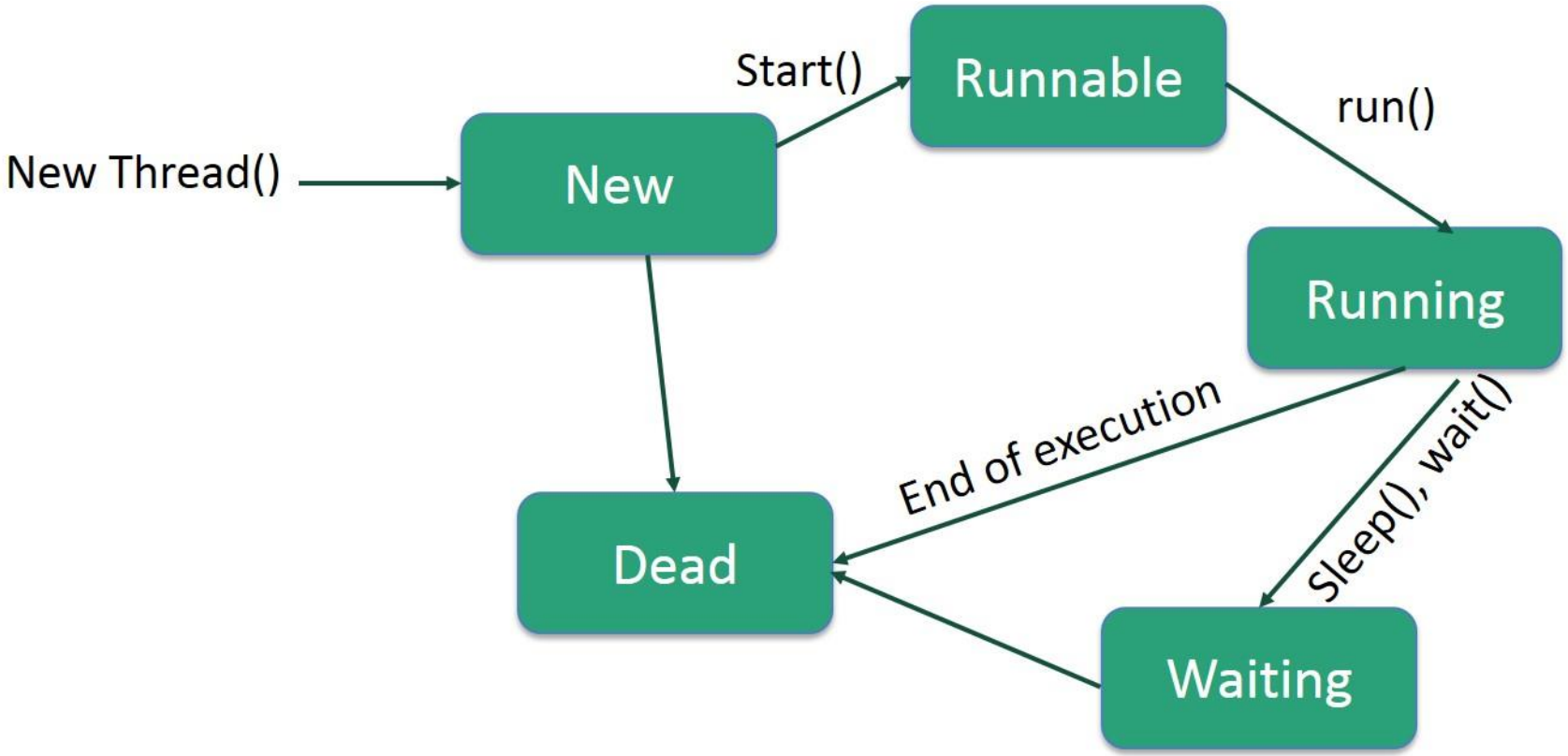
- To directly control thread creation and management, simply instantiate Thread each time the application needs to initiate an asynchronous task.

Threads

- Each thread is associated with an instance of the class Thread. There are two basic strategies for using Thread objects to create a concurrent application.

- To directly control thread creation and management, simply instantiate Thread each time the application needs to initiate an asynchronous task.

Thread Life cycle



And what does that mean?

Following are the stages of the life cycle:

New – A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

Runnable – After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting – Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Terminated (Dead) – A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread priorities

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

Runnable


```
public class HelloRunnable implements
Runnable {

    public void run() {
        System.out.println("Hello from a
thread!");
    }

    public static void main(String args[]) {
        (new Thread(new
HelloRunnable())).start();
    }
}
```

Extend Thread


```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a
thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

Which one to use?

- The first idiom, which employs a Runnable object, is more general, because the Runnable object can subclass a class other than Thread.
- The second idiom is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of Thread.
- The Thread class defines a number of methods useful for thread management.

Thread methods demo

Interrupt

- **Interrupt**

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else.

Join

- The join method allows one thread to wait for the completion of another. If t is a Thread object whose thread is currently executing,
  t.join();
- causes the current thread to pause execution until t's thread terminates. Overloads of join allow the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.
- Like sleep, join responds to an interrupt by exiting with an InterruptedException.

Thread synchronization

volatile keyword

- private volatile int counter= 0;

the volatile modifier guarantees that any thread that reads a field will see the most recently written value.

synchronized keyword

To make a method synchronized, simply add the synchronized keyword to its declaration:

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

synchronized keyword

- *If count is an instance of SynchronizedCounter, then making these methods synchronized has two effects:*

- **1. It is not possible** for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

- 2. When a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

synchronized keyword

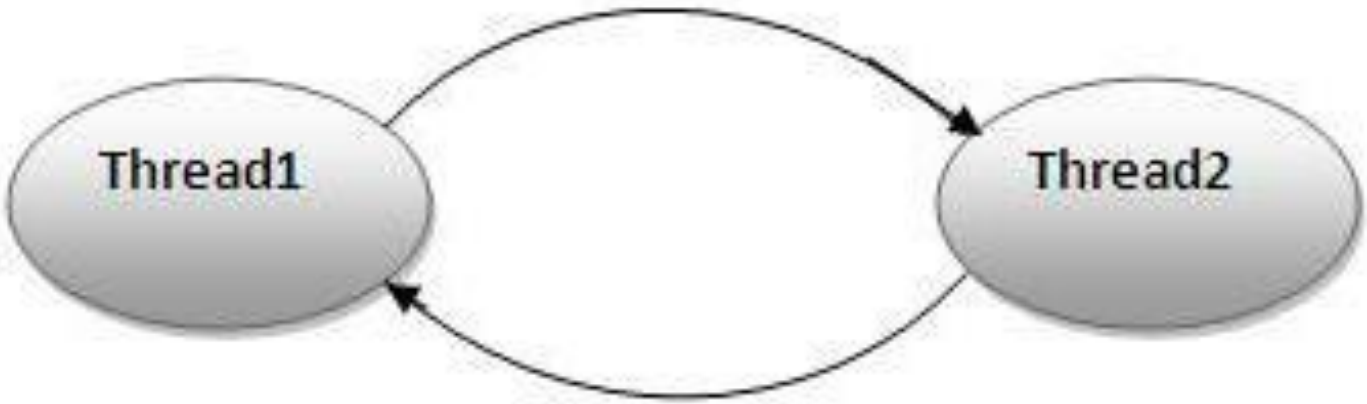
- Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods.
- **An important exception**: final fields, which cannot be modified after the object is constructed, can be safely read through non-synchronized methods, once the object is constructed.
- This is valid as well for all the **immutable objects**!
- They are all **thread safe**.

Locks

- A lock is a thread synchronization mechanism like synchronized blocks except locks can be more sophisticated than Java's synchronized blocks.
- Locks (and other more advanced synchronization mechanisms) are created using synchronized blocks, so it is not like we can get totally rid of the synchronized keyword.
- Next time we will look at the java.util.concurrent.locks and the lock object.

Dead locks in Java

- Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Dead lock example

wait, notify and notifyAll

- The Object class in java contains three final methods that allows threads to communicate about the lock status of a resource. These methods are wait(), notify() and notifyAll().

wait

Object wait methods has three variance, one which waits indefinitely for any other thread to call notify or notifyAll method on the object to wake up the current thread. Other two variances puts the current thread in wait for specific amount of time before they wake up.

wait, notify and notifyAll

- **notify**

  notify method wakes up only one thread waiting on the object and that thread starts execution. So if there are multiple threads waiting for an object, this method will wake up only one of them. The choice of the thread to wake depends on the OS implementation of thread management.

  notifyAll

  notifyAll method wakes up all the threads waiting on the object, although which one will process first depends on the OS implementation.

wait, notify and notifyAll

- These methods can be used to implement producer consumer problem where consumer threads are waiting for the objects in Queue and producer threads put object in queue and notify the waiting threads.

Let’s see an example where multiple threads work on the same object and we use wait, notify and notifyAll methods.

Summary

- Basics – what is multithreading, when to use it , why use it
- Introduction to threads
- Runnable
- Thread synchronization
- Managing multiple threads