Contents

[Thread Name, Priority, and Group 3](#_Toc510840303)

[The States of a Thread 3](#_Toc510840304)

[Two States in “Runnable” State 4](#_Toc510840305)

[Concurrent Access Problems 4](#_Toc510840306)

[Data Race 5](#_Toc510840307)

[Deadlocks 7](#_Toc510840308)

[Livelocks 7](#_Toc510840309)

[Lock Starvation 7](#_Toc510840310)

[The Wait/Notify Mechanism 8](#_Toc510840311)

**REFERENCES**

1. Oracle Certified Professional Java SE 7 Programmer Exams 1Z0-804 and 1Z0-805

# Thread Name, Priority, and Group

Every thread has a name, which we can used to identify the thread. If we do not give a name explicitly, a thread will get a default name. The priority can vary from 1, the lowest, to 10, the highest.

The priority of the normal thread is by default 5, and we can change this default priority value by explicitly providing a priority value.

Every thread is part of a thread group. It’s a rarely used feature.

The toString() method of Thread prints these three details

We can change the name of the threads as we wish and it does not change the behavior of the program. However, we need to be careful in changing thread priority since it can affect scheduling of threads.

The sleep() method throws InterruptedException. Since InterruptedException is a checked exception (it extends from the Exception class), we need to provide a try-catch block around sleep() or declare the run() method that throws the exception InterruptedException. However, if we declare void run() throws InterruptedException, we won’t be overriding the run() method since the exception specification is different (the run() method does not throw any checked exceptions).

What InterruptedException means and when it gets thrown. A thread can “interrupt” another thread, say, to request it to stop working. In that case, the thread receiving the interrupt—if it is

in sleep() or wait() (which we’ll revisit later)—results in throwing an InterruptedException. The thread receiving the interrupt can ignore the interrupt and continue execution (which is not a good idea, but it is possible to do so), or it can stop the execution.

# The States of a Thread



## Two States in “Runnable” State

Once a thread makes the state transition from the new state to the runnable state, we can think of the thread having two states at the OS level: the ready state and running state. A thread is in the ready state when it is waiting for the OS to run it in the processor.

When the OS actually runs it in the processor, it is in the running state. There might be many

threads waiting for processor time. The current thread may end up taking lots of time and finally may give up the CPU voluntarily. In that case, the thread again goes back to the ready state. These two states are shown in Figure





# Concurrent Access Problems

Concurrent Access problems are mainly of two types

* Data Races
* Deadlocks

## Data Race

Threads share memory, and they can concurrently modify data. Since the modification can be done at the same time without safeguards, this can lead to unintuitive results.

When two or more threads are trying to access a variable and one of them wants to modify it, we get a problem known as a data race (also called as race condition or race hazard).

*(Refer to com.dikshit.threads.basics.synchronised and com.dikshit.threads.basics.synchronised1 packages for examples)*

To avoid this problem, we need to ensure that a single thread does the write and read operations together (atomically).

The section of code that is commonly accessed and modified by more than one thread is known as **critical section.** To avoid the data race problem, we need to ensure that the critical section is executed by only one thread at a time.

This is done by acquiring a lock on the object. Only a single thread can acquire a lock on an object at a time, and only that thread can execute the block of code (i.e., the critical section) protected by the lock. Until then, the other threads have to wait. Internally, this is implemented with monitors and the process is called locking and unlocking (i.e., thread synchronization).

In Java it is done by **synchronized** keyword. Java has a keyword, synchronized, that helps in thread synchronization.

We can use it in two forms—

* Synchronized blocks
* synchronized methods.

In **synchronized blocks**, we use the synchronized keyword for a reference variable and follow it by a block of code.

A thread has to acquire a lock on the synchronized variable to enter the block; when the execution of the block completes, the thread releases the lock. For example, you can acquire a lock on this reference if the block of code is within a non-static method:

**synchronized(this) {**

**// code segment guarded by the mutex lock**

**}**

What if an exception gets thrown inside the synchronized block? Will the lock get released? Yes, no matter whether the block is executed fully or an exception is thrown, the lock will be automatically released by the JVM.

With synchronized blocks, you can acquire a lock on a reference variable only. If you use a primitive type, you will get a compiler error.

A **synchronized method** is equivalent to a synchronized block if you enclose the whole method body in a synchronized(this) block.

We can declare ***static methods*** synchronized. However, what is the reference variable on which the lock is obtained? Remember that static methods do not have the implicit this reference.

Static synchronized methods acquire locks on the class object. Every class is associated with an object of Class type, and we can access it using ClassName.class syntax.

For example,

**class SomeClass {**

**private static int val;**

**public static synchronized void assign(int i)**

**{**

**val = i;**

**}**

**// more members . . .**

**}**

In this case, the assign method acquires a lock on the SomeClass.class object when it is called. Now the equivalent assign() method using synchronized blocks can be written as

**class SomeClass {**

**private static int val;**

**public static void assign(int i) {**

**synchronized(SomeClass.class)**

**{**

**val = i;**

**}**

**}**

**// more members . . .**

**}**

*Beginners commonly misunderstand that a synchronized block obtains a lock for a block of code.*

*Actually, the lock is obtained for an object and not for a piece of code. The obtained lock is held until all the statements in that block complete execution.*

Constructors can’t be synchronized because JVM ensures that only one thread can invoke a

constructor call (for a specific constructor) at a given point in time. So, there is no need to declare a constructor synchronized. However, if you want, you can use synchronized blocks inside constructors.

**Synchronized Blocks vs. Synchronized Methods (Which to use ?)**

If you want to acquire a lock on an object for only a small block of code and not the whole method, then synchronized blocks are sufficient; using synchronized methods is overkill in that case. In general, it is better to acquire locks for small segments of code instead of locking methods unnecessarily, so synchronized blocks are useful there. In synchronized blocks, you can explicitly provide the reference object on which you want to acquire a lock.

However, in the case of a synchronized method, you do not provide any explicit reference to acquire a lock on. A synchronized method acquires an implicit lock on the this reference (for instance methods) and class object (for static methods).

On the other hand, if you want to acquire a lock on the entire body of a small method, then using synchronized as a method attribute is more elegant and convenient than synchronized blocks. In synchronized methods, while reading the declaration of the method itself, it becomes clear that a method is synchronized; with synchronized blocks, you need to read the documentation or look inside the code to understand that some synchronization is performed.

## Deadlocks

A deadlock arises when locking threads result in a situation where they cannot proceed and thus wait indefinitely for others to terminate. Say, one thread acquires a lock on resource r1 and waits to acquire another on resource r2.

At the same time, say there is another thread that has already acquired r2 and is waiting to obtain a lock on r1.

Neither of the threads can proceed until the other one releases the lock, which never happens—so they are stuck in a deadlock.

(Refer com.dikshit.threads.basics.deadlock)

* Deadlocks can arise in the context of multiple locks.
* If multiple locks are acquired in the same order, then a deadlock will not occur; however, if you acquire them in a different order, then deadlocks may occur.
* Deadlocks (just like other multi-threading problems) are non-deterministic; you cannot consistently reproduce deadlocks.

## Livelocks

To help understand livelocks, let’s consider an analogy. Assume that there are two robotic cars that are programmed to automatically drive in the road. There is a situation where two robotic cars reach the two opposite ends of a narrow bridge. The bridge is so narrow that only one car can pass through at a time. The robotic cars are programmed such that they wait for the other car to pass through first. When both the cars attempt to enter the bridge at the same time, the following situation could happen: each car starts to enter the bridge, notices that the other car is attempting to do the same, and reverses! Note that the cars keep moving forward and backward and thus appear as if they’re doing lots of work, but there is no progress made by either of the cars. This situation is called a livelock.

Consider two threads t1 and t2. Assume that thread t1 makes a change and thread t2 undoes that change. When both the threads t1 and t2 work, it will appear as though lots of work is getting done, but no progress is made. This situation is called a livelock in threads.

The similarity between livelocks and deadlocks is that the process “hangs” and the program never terminates.

However, in a deadlock, the threads are stuck in the same state waiting for other thread(s) to release a shared resource; in a livelock, the threads keep executing a task, and there is continuous change in the process states, but the application as a whole does not make progress.

## Lock Starvation

Consider the situation in which numerous threads have different priorities assigned to them (in the range of lowest priority, 1, to highest priority, 10, which is the range allowed for priority of threads in Java).

When a mutex lock is available, the thread scheduler will give priority to the threads with high priority over low priority. If there are many high-priority threads that want to obtain the lock and also hold the lock for long time periods, when will the low-priority threads get a chance to obtain the lock? In other words, in a situation where low-priority threads “starve” for a long time trying to obtain the lock is known as lock starvation.

# The Wait/Notify Mechanism

Refer com.dikshit.threads.basics.waitNotify

**Should you use notify() or notifyAll()?**

The notify() method wakes up one thread waiting for the lock (the first thread that called wait() on that lock).

The notifyAll() method wakes up all the threads waiting for the lock; the JVM selects one of the threads from the list of threads waiting for the lock and wakes that thread up.

In the case of a single thread waiting for a lock, there is no significant difference between notify() and notifyAll(). however, when there is more than one thread waiting for the lock, in both notify() and notifyAll(), the exact thread woken up is under the control of the JVM and you cannot programmatically control waking up a specific thread.

At first glance, it appears that it is a good idea to just call notify() to wake up one thread; it might seem unnecessary to wake up all the threads. however, the problem with notify() is that the thread woken up might not be the suitable one to be woken up (the thread might be waiting for some other condition, or the condition is still not satisfied for that thread etc). In that case, the notify() might be lost and no other thread will wake up potentially leading to a type of deadlock (the notification is lost and all other threads are waiting for notification—forever!).

To avoid this problem, it is always better to call notifyAll() when there is more than one thread waiting for a lock (or more than one condition on which waiting is done). the notifyAll() method wakes up all threads, so it is not very efficient. however, this performance loss is negligible in real world applications.

* Never call the start() method twice on the same thread.
* Never write a catch block for handling IllegalThreadStateException. If you get this exception,there is certainly a bug in the code. Fix that bug.
* Call wait and notify/notifyAll only after acquiring the relevant lock. This can be by declaring the run method synchronized