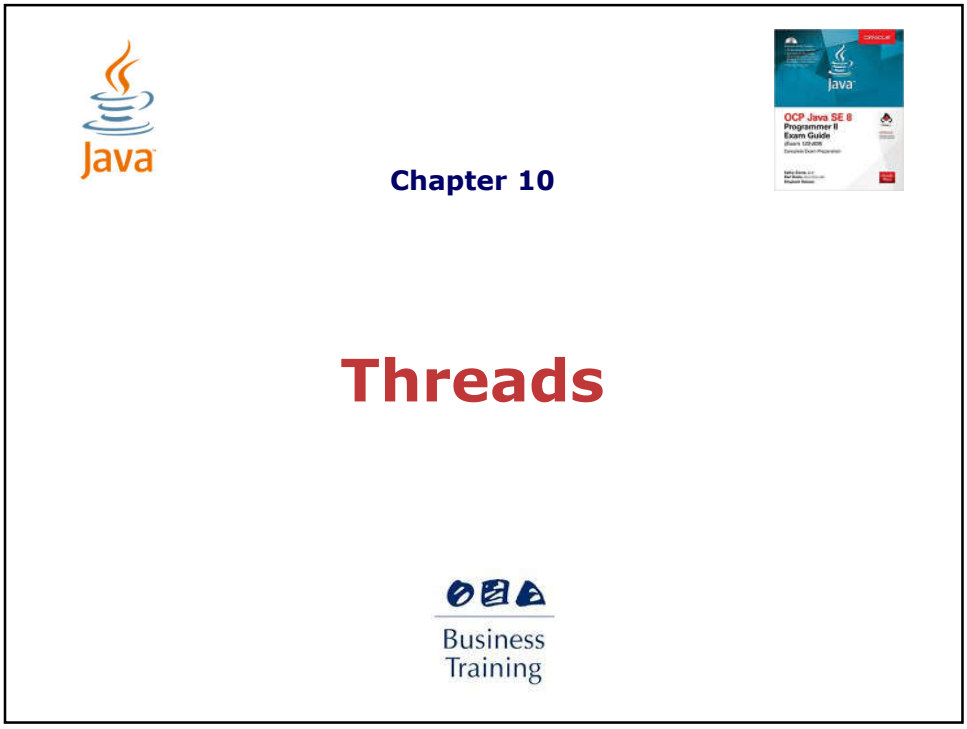


1

Content

- Defining and Instanciating Threads
- Thread States and Transitions
- Synchronizing Code, Thread Problems
- Thread Interaction

2

Defining and Instanciating Threads

3

What is a Thread

- **Like any other object in Java, it has variables and methods, and lives and dies on the heap.**
  - But a thread of execution is an individual process (a "lightweight" process) that has its own call stack. In Java, there is one thread per call stack—or, to think of it in reverse, one call stack per thread
  - The main() method, that starts the whole ball rolling, runs in one thread, called (surprisingly) the *main* thread. If you looked at the main call stack (and you can, any time you get a stack trace from something that happens after main begins, but not within another thread), you'd see that main() is the first method on the stack— the method at the bottom.
  - As you'll learn a little later, different JVMs can run threads in profoundly different ways. For example, one JVM might be sure that all threads get their turn, with a fairly even amount of time allocated for each thread in a nice, happy, round-robin fashion.

4

Defining a Thread

- **Two Ways**
  1. Extending **java.lang.Thread** and overriding the **run()** method

```
class MyThread extends Thread {
  public void run() {
   System.out.println("Important job running in MyThread");
  }
```

The limitation with this approach is that if you extend Thread, you can't extend anything else.

  2. Implementing **java.lang.Runnable**

The Runnable interface gives you a way to extend from any class you like:

```
class MyRunnable implements Runnable {
   public void run() {
     System.out.println("Important job running in MyRunnable");
    }
}
```


5

Instantiating a Thread

1. **If you extended the Thread class, instantiation is dead simple :**
   - **MyThread t = new MyThread( )**
2. **If you implement Runnable, instantiation is only slightly less simple:**
   - First, you instantiate your Runnable class:

     MyRunnable r = new MyRunnable();
   - Next, you get yourself an instance of java.lang.Thread and you give it your job!

     Thread t = new Thread(r); // Pass your Runnable to the Thread

You can pass a single Runnable instance to multiple Thread objects, so that the same Runnable becomes the target of multiple threads, as follows:

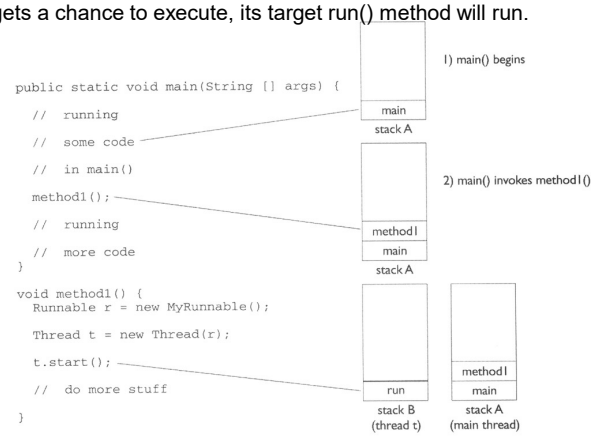
```
public class TestThreads {
  public static void main (String [] args) {
    MyRunnable r = new MyRunnable();
    Thread foo = new Thread(r);
    Thread bar = new Thread(r);
    Thread bat = new Thread(r);
  }
}
```


6

Starting a Thread

- **It's so simple : t.start();**
  - So what happens after you call start()?
    - A new thread of execution starts (with a new call stack).
    - The thread moves from the new state to the runnable state.
    - When the thread gets a chance to execute, its target run() method will run.

- **The Figure shows the process of starting a thread.**



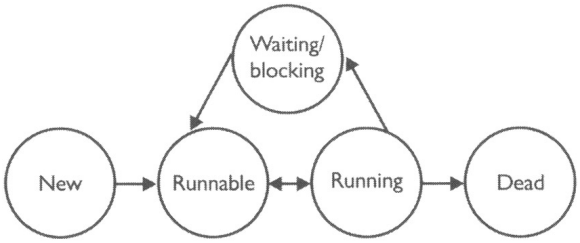
7

Thread States and Transitions

8

Thread States

- **A thread can be only in one of five states :**
  1. **New**
  2. **Runnable** :This is the state a thread is in when it's eligible to run, but the scheduler has not selected it to be the running thread.
  3. **Running** : The "big time." Where the action is.
  4. **Waiting/blocked/sleeping** :This is the state a thread is in when it's eligible to run
  5. **Dead** :A thread is considered dead when its run() method completes



Sleeping

- **The sleep() method is a static method of class Thread. Y**
  - You use it in your code to "slow a thread down" by forcing it to go into a sleep mode before coming back to runnable

```
try {
  Thread.sleep(5*60*1000);  // Sleep for 5 minutes
} catch (InterruptedException ex) { }
```

    - Notice that the sleep() method can throw a **checked InterruptedException** (handle or declare). Typically, you wrap calls to sleep() in a try/catch, as in the preceding code.

Exam Watch

- Just keep in mind that the behavior in the preceding output is still not guaranteed.
  - Just because a thread's sleep() expires, and it wakes up, does not mean it will return to running!
  - Remember, when a thread wakes up, it simply goes back to the runnable state. So the time specified in sleep() is the minimum duration in which the thread won't run, but it is not the exact duration in which the thread won't run. So you can't, for example, rely on the sleep() method to give you a perfectly accurate timer.

Thread Priorities and yield( )

- **To understand yield(), you must understand the concept of thread priorities.**
  - Threads always run with some priority, usually represented as a number between 1 and 10
  - The scheduler in most JVMs uses preemptive, priority-based scheduling
  - This does not mean that all JVMs use time slicing.
- **What yield() is supposed to do is make the currently running thread head back to namable to allow other threads of the same priority to get their turn.**
  - **So the intention is to use yield() to promote graceful turn-taking among equal-priority threads**.
    - There's no guarantee the yielding thread won't just be chosen again over all the others!

11

The join() Method

- **The non-static join() method of class Thread lets one thread "join onto the end" of another thread.**
  - If you have a thread B that can't do its work until another thread A has completed its work, then you want thread B to "join" thread A. This means that thread B will not become runnable until A has finished (and entered the dead state).

```
Thread t = new Thread();
t.start();
t.join();
```

  - A call to join() Guaranteed to cause the current thread to stop executing until the thread it joins with (in other words, the thread it calls join() on) completes, or if the thread it's trying to join with is not alive, however, the current thread won't need to back out.
- **Besides the three(sleep() , yield() and join()), we also have the following scenarios in which a thread might leave the running state:**
  - The thread's **run() method completes**. Duh.
  - A call to **wait()** on an object (we don't call wait() on a thread, as we'll see in a moment).
  - A thread can't acquire the lock on the object whose method code it's attempting to run.
  - The thread scheduler can decide to move the current thread from running to runnable in order to give another thread a chance to run. No reason is needed—the thread scheduler can trade threads in and out whenever it likes.

12

Synchronizing Code, Thread Problems

13

Race Conditions

- **When two or more threads are trying to access a variable and one of them wants to modify it, you get a problem known as a race condition (also known as data race or race hazard).**
- **So, what is the problem?**
  - The expression **Counter.count++** is a write operation, and the next System.out.print statement has a read operation for Counter.count.
    - When the three threads execute, each of them has a local copy of the value **Counter.count** and when they update the counter with **Counter.count++**, they need not immediately reflect that value in the main memory.
- **Synchronized Blocks**
  - 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.

```
// This class exposes a publicly accessible counter
// to help demonstrate race condition problem
class Counter {
    public static long count = 0;
}

// This class implements Runnable interface
// Its run method increments the counter three times
class UseCounter implements Runnable {
    public void increment() {
        // increments the counter and prints the value
        // of the counter shared between threads
        Counter.count++;
        System.out.print(Counter.count + "  ");
    }
    public void run() {
        increment();
        increment();
        increment();
    }
}

// This class creates three threads
public class RaceCondition {
    public static void main(String args[]) {
        UseCounter c = new UseCounter();
        Thread t1 = new Thread(c);
        Thread t2 = new Thread(c);
        Thread t3 = new Thread(c);
        t1.start();
        t2.start();
        t3.start();
    }
}
```


14

Threading Problems

- **Deadlocks**
  - Obtaining and using locks is tricky, and it can lead to lots of problems. One of the difficult (and common) problems is known as a deadlock. **A deadlock arises when locking threads result in a situation where they cannot proceed and thus wait indefinitely for others to terminate**
- **Livelocks**
  - 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.
- **Thread Starvation**
  - Starvation is related to livelock. Starvation is when a thread is unable to make progress because it cannot get access to a shared resource that other threads are hogging.
    - An example: the situation where low-priority threads "starve" for a long time trying to obtain the lock is known as lock starvation.

15

Synchronization and Locks

- **Remember the following key points about locking and synchronization:**
  - Only methods (or blocks) can be synchronized, not variables or classes.
  - Each object has just one lock.
  - Not all methods in a class need to be synchronized. A class can have both synchronized and non-synchronized methods.
  - If two threads are about to execute a synchronized method in a class, and both threads are using the same instance of the class to invoke the method, only one thread at a time will be able to execute the method. The other thread will need to wait until the first one finishes its method call. In other words, once a thread acquires the lock on an object, no other thread can enter any of the synchronized methods in that class (for that object).
  - If a class has both synchronized and non-synchronized methods, multiple threads can still access the class's non-synchronized methods! If you have methods that don't access the data you're trying to protect, then you don't need to synchronize them. Synchronization can cause a hit in some cases (or even deadlock if used incorrectly), so you should be careful not to overuse it.
  - If a thread goes to sleep, it holds any locks it has—it doesn't release them.
  - A thread can acquire more than one lock.
  - You can synchronize a block of code rather than a method.

16

So What About Static Methods? Can They Be Synchronized?

- **static methods can be synchronized.**
  - There is only one copy of the static data you're trying to protect, so you only need one lock per class to synchronize static methods—a lock for the whole class. There is such a lock; every class loaded in Java has a corresponding instance of java.lang
  - There's nothing special you have to do to synchronize a static method:

```
public static synchronized int getCount() {
   return count;
}
```

    - Again, this could be replaced with code that uses a synchronized block. If the method is defined in a class called MyClass, the equivalent code is as follows:

```
public static int getCount() {
    synchronized(MyClass.class) {
      return count;
    }
}
```


17

What Happens If a Thread can't Get the Lock?

- **If a thread tries to enter a synchronized method and the lock is already taken, the thread is said to be blocked on the object's lock.**
  - The thread goes into a kind of pool for that particular object and has to sit there until the lock is released and the thread can again become runnable/running.
  - Just because a lock is released doesn't mean any particular thread will get it. There might be other threads waiting for a single lock.
- **When thinking about blocking, it's important to pay attention to which objects are being used for locking.**
  - Threads calling non-static synchronized methods in the same class will only block each other if they're invoked using the same instance.
  - Threads calling static synchronized methods in the same class will always block each other—they all lock on the same Class instance.
  - A static synchronized method and a non-static synchronized method will not block each other, ever. The static method locks on a Class instance while the non-static method locks on the this instance—these actions do not interfere with each other at all.
  - For synchronized blocks, you have to look at exactly what object has been used for locking. (What's inside the parentheses after the word synchronized?) Threads that synchronize on the same object will block each other. Threads that synchronize on different objects will not.

18

Thread Interaction

19

Thread Interaction

- The Object class has three methods, wait(), notify(), and notifyAll() that help threads communicate about the status of an event that the threads care about.
- One key point to remember (and keep in mind for the exam) about wait/notify is this:
  - *wait(), notify(), and notifyAll()* must be called from within a synchronized context! A thread can't invoke a wait or notify method on an object unless it owns that object's lock.
- *Exam Watch :* When the wait() method is invoked on an object, the thread executing that code gives up its lock on the object immediately. However, when notify() is called, that doesn't mean the thread gives up its lock at that moment. If the thread is still completing synchronized code, the lock is not released until the thread moves out of synchronized code. So just because notify() is called doesn't mean the lock becomes available at that moment.

20

Thread Safe Classes

- **When a class has been carefully synchronized to protect its data (using the rules just given, or using more complicated alternatives), we say the class is "thread-safe."**
  - Many classes in the Java APIs already use synchronization internally in order to make the class "**thread-safe**." For example, StringBuffer and StringBuilder are nearly identical classes, except that all the methods in StringBuffer are synchronized when necessary, while those in StringBuilder are not.
- **Thread Deadlock**
  - Deadlock occurs when two threads are blocked, with each waiting for the other's lock. Neither can run until the other gives up its lock, so they'll sit there forever.