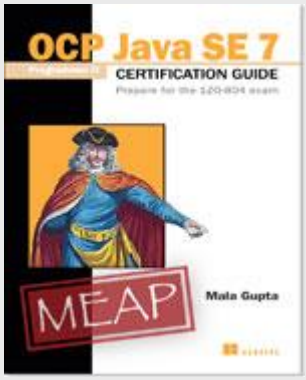


## Ways to pause thread execution in OCP Java SE 7

*By Mala Gupta*

Imagine while debugging or testing your code, you need to reproduce a bug and you need to make a thread pause its execution and give up its current use of the processor. In this article, we discuss some methods you can use to do that.

*This article is excerpted from OCP Java SE 7 Programmer II Certification Guide. Save 39% on* OCP Java SE 7 Programmer II Certification Guide *with code 15dzamia at manning.com.*

Imagine while debugging or testing your code, you need to make a thread pause its execution and give up its current use of the processor. A thread might pause its execution due to the calling of an explicit method or when its time slice with the processor expires.

### THREAD SCHEDULING

A processor's time is usually time-sliced to allow multiple threads to run, with each thread using one or more time slices. A thread scheduler might suspend a running thread and move it to the ready state, executing another thread from the ready queue. Thread scheduling is specific to JVM implementation and beyond the control of an application programmer. The scheduler moves a thread from the `READY` state to `RUNNING` and vice versa, to support concurrent processing of threads.


**EXAM TIP** As a Java programmer, you can't control or determine when the thread scheduler moves a thread from the `RUNNING` state to `READY` and vice versa. It's specific to an OS.

### METHOD THREAD.YIELD()

Let's say that while debugging or testing your code, you need to reproduce a bug due to race conditions (that is, when multiple threads compete). You can insert a call to `Thread.yield()` in one of the threads. A static method `yield()` makes the currently executing thread pause its execution and give up its current use of the processor. But it only acts as a hint to the scheduler. The scheduler might also ignore it. The static method `yield` can be placed literally anywhere in your code—not only in the `run` method:

```
class YieldProcessorTime {
    public static void main(String args[]) {
```

```
        Thread sing = new Sing();
        sing.start();
        Thread.yield();                    #A
    }
}
class Sing extends Thread{
    public void run() {
        yield();                           #B
        System.out.println("Singing");
    }
}
```

**#A Might cause thread main to yield its processor time**
**#B When executed, might cause thread Sing to yield its processor time**

As shown in figure 1, when called from two threads, thread 1 and thread 2, `yield()` might or might not *yield* its execution.

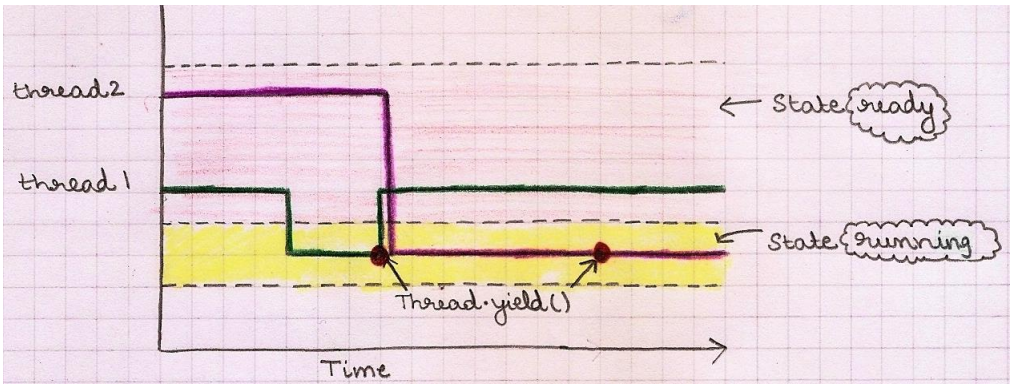


Figure 1 Calling `yield()` might yield the current thread's execution slot as soon as it's called or after a delay, or it might be ignored.

So what's guaranteed from this method call? To be precise, nothing. It might not make the currently executing thread give up its processor time. If it does, it doesn't guarantee when it will happen and when the thread will resume its execution.


**EXAM TIP** The `yield` method is static. It can be called from any method, and it doesn't throw any exceptions.

### METHOD THREAD.SLEEP()

The static `Thread.sleep` method is *guaranteed* to cause the currently executing thread to temporarily give up its execution for *at least* the specified number of milliseconds (and nanoseconds) and move to the `READY` state. You use `sleep()` to *slow down* the execution of

your thread. Imagine you're using a thread to animate a ball across a visible frame. Execute the following class and you'll see that a black ball zooms across the screen:

```
import javax.swing.*;
import java.awt.*;
class MyFrame {
    public static void main(String args[]) {
        JFrame frame = new JFrame();                    #A
        frame.setSize(400, 300);                        #A
        frame.setVisible(true);                         #A
        MovingBall ball = new MovingBall(60, frame);       #B
        ball.start();                                      #B
    }
}
class MovingBall extends Thread{
    int radius;
    Graphics g;
    int xPos, yPos;                                      #C
    JFrame frame;
    MovingBall(int radius, JFrame frame) {
        this.radius = radius;
        this.g = frame.getGraphics();
        this.frame = frame;
    }
    public void run() {
        while (true) {                                  #D
            g.setColor(Color.WHITE);                                    #E
            g.fillRect(0, 0, frame.getWidth(), frame.getHeight());  #E
            ++xPos; ++yPos;                               #F
            g.setColor(Color.BLACK);                       #G
            g.fillOval(xPos, yPos, radius, radius);        #G
        }
    }
}
```

**#A Create a frame**
**#B Create and start MovingBall thread**
**#C x and y positions to draw an oval**
**#D Execute it forever**
**#E Paint whole screen white**
**#F Modify x and y position of moving ball**
**#G Draw ball at new x and y position**

In the preceding code, the `run` method tries to animate a ball by first painting the complete visible frame area with white and then drawing a black oval at the specified `x` and `y` positions. Each loop repeats these steps with modified `x` and `y` coordinates that make the ball move across the frame. You can slow down the moving ball by making `MovingBall` sleep for the specified milliseconds (modifications are in bold):

```
class MovingBall extends Thread{
    int radius;
    Graphics g;
    int xPos, yPos;
```

```
    JFrame frame;
    MovingBall(int radius, JFrame frame) {
        this.radius = radius;
        this.g = frame.getGraphics();
        this.frame = frame;
    }
    public void run() {
        while (true) {
            try {
                Thread.sleep(10);                    #A
            }
            catch (InterruptedException e) {         #B
                System.out.println(e);
            }
            xPos = xPos + 2; yPos = yPos + 2;
            g.setColor(Color.WHITE);
            g.fillRect(0, 0, frame.getWidth(), frame.getHeight());
            g.setColor(Color.BLACK);
            g.fillOval(xPos, yPos, radius, radius);
        }
    }
}
```

**#A Guarantees to sleep for at least 10 milliseconds (if not interrupted)**
**#B If interrupted, sleeping thread might throw InterruptedException**

Class `Thread` defines the overloaded versions of `sleep()` as follows:

```
public static native void sleep(long milli) throws InterruptedException;
public static void sleep(long milli, int nanos) throws InterruptedException
```

Whether a thread will sleep for the precise duration specified in nanoseconds will depend on an underlying system. Unless interrupted, the currently executing thread will sleep at least for the specified duration. On the exam, watch out for questions that state a thread will become runnable *exactly* after the expiration of the sleep duration. Unless interrupted, a thread is guaranteed to sleep for *at least* the specified duration. The exact time to resume execution depends on the thread scheduler.

 **EXAM TIP** A thread that's suspended due to a call to `sleep` doesn't lose ownership of any monitors.

You can also expect questions on where to place a call to the `sleep` method. The answer depends on who is executing a call to `sleep()`. Like `yield()`, `sleep()` is `Thread`'s static method. It makes the currently executing thread give up its execution and sleep. It can be called from any piece of code—all code is executed by some thread. If it's placed in `Runnable`'s `run()`, it will cause the thread to sleep. Placed otherwise, it will make the calling thread sleep. What happens if you place `sleep()` in `MovingBall`'s constructor (showing only relevant code)?

```
class MovingBall extends Thread{
```

```
    //..code not shown deliberately
    MovingBall(int radius, JFrame frame) {
        try {
            Thread.sleep(1000);
        }
        catch (InterruptedException e) {
            System.out.println(e);
        }
        //..code not shown deliberately
    }
    public void run() {
        //..code not shown deliberately
    }
}
```

A thread that instantiates `MovingBall` will execute `sleep()`, and then sleep for the specified duration before it can complete `MovingBall`'s instantiation.

### METHOD JOIN()

A thread might need to pause its *own* execution when it's waiting for another thread to complete its task. If thread A calls `join()` on a `Thread` instance B, A will wait for B to complete its execution before A can proceed to its own completion. Imagine multiple teams—design, development, and testing—are working on a software project. The project can't be sent to a customer until all of these teams complete their tasks. Ideally, the delivery process will wait for design development and testing teams to complete their tasks before the delivery process can move ahead with its own task of handing over the project to a customer. Let's code a subset of this example. The design team must complete the design of a screen before a developer can start coding it:

```
class ScreenDesign extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) System.out.println(i);       #A
    }
}
class Developer {
    ScreenDesign design;
    Developer(ScreenDesign design) {
        this.design = design;               #B
    }
    public void code() {
        try {
            System.out.println("Waiting for design to complete");
            design.join();                                                #C
            System.out.println("Coding phase start");
        }
        catch(InterruptedException e) {                    #D
            System.out.println(e);                         #D
        }                                                  #D
    }
}
class Project {
    public static void main(String[] args) {
```

```
        ScreenDesign design = new ScreenDesign(); design.start();     #E
        Developer dev = new Developer(design);
        dev.code();                                                   #F
    }
}
```

**#A run simply outputs a couple of numbers**
**#B Developer stores a reference to Design**
**#C code calls design.join()**
**#D join() throws checked InterruptedException**
**#E Start thread Design**
**#F dev.code(), which runs in thread main, calls design.join**

Figure 10.6 shows probable output of the preceding code. Because class `Developer` isn't a `Runnable` instance, it doesn't execute it in its own thread of execution; rather, it executes in the thread `main`. When `Developer` calls `design.join()`, the thread `main` waits for `design` to complete its execution before executing the rest of its code.

| | | |
|---|---|---|
| 0 | 0 | Waiting for design to complete |
| Waiting for design to complete | 1 | 0 |
| 1 | 2 | 1 |
| 2 | 3 | 2 |
| 3 | Waiting for design to complete | 3 |
| 4 | 4 | 4 |
| Coding phase start | Coding phase start | Coding phase start |

Figure 2 Probable outputs of calling `join()` on a thread

**EXAM TIP** The `join()` method guarantees that the calling thread won't execute its remaining code until the thread on which it calls `join()` completes.

Class `Thread` defines overloaded `join` methods as follows:

```
public final synchronized void join(long milli) throws InterruptedException
public final synchronized void join(long millis, int nanos)
                                              throws InterruptedException
public final void join() throws InterruptedException
```

The variations of `join` that accept milliseconds and nanoseconds wait for at least the specified duration (if they're not interrupted!). Behind the scenes, `join` is implemented using the `wait`, `isAlive`, and `notifyAll` methods.

### METHODS WAIT(), NOTIFY(), AND NOTIFYALL()

Imagine a server accepts and queues multiple requests from users that are processed by a thread. This thread might need to wait and pause its own execution if there are no new requests in the queue. A thread can pause its execution and wait on an object, a queue in this case, by calling `wait()`, until another thread calls `notify()` or `notifyAll()` on the same object.