Some methods of class Thread

We briefly describe some of the methods in class Thread. Scanning this page will give you an idea what you can do with Thread objects. We do not give full details. You don't have to memorize anything here.

1. **Who's running?** Static method Thread.currentThread() gives you a pointer to the Thread object that is currently executing.

2. **Dead or alive?** A thread is alive if it has been started but has not died —has not completed execution of its method run(). You can tell whether thread t is alive by calling t.isAlive() —it returns the obvious boolean.

3. **What's its name?** When demoing or debugging programs with threads, it's useful to have a recognizable name for a thread. Therefore, class Thread has these two methods:

```
t.setName(String)   and   t.getName()
```

4. **I want to sleep!** Method sleep(n) causes the currently executing thread to sleep (cease execution) for n milliseconds. It doesn't lose ownership of any locks that it has (you'll learn what this means later).

Another thread may interrupt this thread —or interrupt its sleep, causing the call on sleep(n) to throw an InterruptedException. Therefore, this call has to be placed in a try-statement:

```
try {sleep(100);}
catch (InterruptedException e) {}
```

Whether something should be done in the catch block depends on the situation. At most, in examples we give, you will see a println statement in the catch block.

5. **What's my priority?** Each thread has a priority, which helps determine when it gets a time-slice and perhaps how long that time-slice will be. We won't be demoing programs that use the following components. The minimum, maximum, and default priorities are static constants of class Thread:

```
Thread.MIN_PRIORITY          Thread.MAX_PRIORITY          Thread.NORM_PRIORITY
```

In addition, there are two instance methods:

```
t.setPriority(int)          t.getPriority()
```

6. **I'll wait for you!** A thread may want to wait for another thread t to die before doing some other task. For this purpose, the method should execute the following call —as shown, it should be in a try-statement because it might throw an InterruptedException.

```
try {t.join();}
catch (InterruptedException e) {}
```

7. **Watch out for those daemons!** For daemon threads and methods that deal with them, see JavaHyperText entry *daemon thread*.

8. **Don't interrupt me!** Calling method t.interrupt() will interrupt thread t. But what that means depends on the state the thread is in —is it sleeping? Waiting? Blocked in an IO operation? This is too complicated to explain here. Just know that threads can be interrupted. In addition, one can ask whether the thread has been interrupted by calling t.isInterrupted).

8. **Don't use me!** The following four methods were introduced in the first version of Java. Since then, it has been determined that they are inherently unsafe or deadlock-prone, so they have been *deprecated*. That means they are still available and can be used, but it's better that you don't use them.

```
t.destroy()    t.stop()    t.resume()    t.suspend()
```