Theory: Thread management

19 minutes 0 / 5 problems solved

We've already learned how to start a new thread by simply invoking the `start` method on a corresponding object. However, sometimes it is necessary to manage the lifecycle of a thread while it's working rather than just start it and leave it be.

In this topic we will consider two commonly used methods in multithreading programming: `sleep()` and `join()`. Both methods may throw a checked `InterruptedException` that is omitted here for brevity.

§1. Sleeping

The static method `Thread.sleep()` causes the currently executing thread to suspend execution for the specified number of milliseconds. This is an efficient means of making processor time available for the other threads of an application or other applications that might be running on a computer.

We will often use this method throughout our educational platform to simulate expensive calls and difficult tasks.

```
System.out.println("Started");

Thread.sleep(2000L); // suspend current thread for 2000 millis

System.out.println("Finished");
```

Let's see what this code does. At first, it prints **"Started"**. Then the current thread is suspended for 2000 milliseconds (it may be longer, but not less than indicated). Eventually, the thread wakes up and prints **"Finished"**.

Another way to make the current thread sleep is to use the special class `TimeUnit` from the package `java.util.concurrent:`

- `TimeUnit.MILLISECONDS.sleep(2000)` performs `Thread.sleep` for 2000 milliseconds;
- `TimeUnit.SECONDS.sleep(2)` performs `Thread.sleep` for 2 seconds;

There are more existing periods: `NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS`, `SECONDS`, `MINUTES`, `HOURS`, `DAYS`.

§2. Joining

The `join` method forces the current thread to wait for the completion of another thread on which the method was called. In the following example, the string **"do something else"** will not be printed until the thread terminates.

```
Thread thread = ...
thread.start(); // start thread

System.out.println("Do something useful");

thread.join();  // waiting for thread to die

System.out.println("Do something else");
```

The overloaded version of this method takes a waiting time in milliseconds:

```
thread.join(2000L);
```

This is used to avoid waiting for too long or even infinitely in case if the thread is hung.

Let's consider another example. The `Worker` class is developed to solve "a difficult task" simulated by the sleep:

Current topic:

Thread management

Topic depends on:

Stage 7
✕ Exception handling

✕ Custom threads

Topic is required for:

Exceptions in threads

Table of contents:

↑ Thread management

§1. Sleeping

§2. Joining

Feedback & Comments

```
class Worker extends Thread {

    @Override
    public void run() {
        try {
            System.out.println("Starting a task");
            Thread.sleep(2000L); // it solves a difficult task
            System.out.println("The task is finished");
        } catch (Exception ignored) {

        }

    }

}
```

Here is the `main` method where the **main** thread waits for completion of the `worker`.

```
public class JoiningExample {
    public static void main(String []args) throws InterruptedException {
        Thread worker = new Worker();
        worker.start(); // start the worker

        Thread.sleep(100L);
        System.out.println("Do something useful");

        worker.join(3000L);  // waiting for the worker

        System.out.println("The program stopped");

    }

}
```

The main thread waits for `worker` and cannot print the message `The program stopped` until the worker terminates or the timeout is exceeded. We know exactly only that `Starting a task` precedes `The task is finished` and `Do something useful` precedes `The program stopped`. There are several possible outputs

First possible output (the task is completed before the timeout is exceeded):

```
Starting a task
Do something useful
The task is finished
The program stopped
```

Second possible output (the task is completed before the timeout is exceeded):

```
Do something useful
Starting a task
The task is finished
The program stopped
```

Third possible output:

```
Do something useful
Starting a task
The program stopped
The task is finished
```

Fourth possible output:

```
Starting a task
Do something useful
The program stopped
The task is finished
```

Report a typo

188 users liked this theory. **5** didn't like it. **What about you?**

Start practicing

Comments (3) Hints (0) Useful links (0) Show discussion