Thread management

Theory | Practice | 100% completed, 0 problems solved

Theory

8 minutes reading

Unskip this topic | Start practicing

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.

2 required topics

- Exception handling
- Custom threads

1 dependent topic

- Exceptions in threads

§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 the thread for which the method `join` 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
```

```
7
8  System.out.println("Do something else");
```

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

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

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

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

```
1  class Worker extends Thread {
2
3      @Override
4      public void run() {
5          try {
6              System.out.println("Starting a task");
7              Thread.sleep(2000L); // it solves a difficult task
8              System.out.println("The task is finished");
9          } catch (Exception ignored) {
1
0          }
1
1      }
1
2  }
```

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

```
1  public class JoiningExample {
2      public static void main(String[] args) throws
InterruptedException {
3          Thread worker = new Worker();
4          worker.start(); // start the worker
5
6          Thread.sleep(100L);
7          System.out.println("Do something useful");
8
9          worker.join(3000L);  // waiting for the worker
1
0          System.out.println("The program stopped");
1
1      }
1
2  }
```

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):

```
1  Starting a task
2  Do something useful
3  The task is finished
4  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

Table of contents:

361 users liked this piece of theory. **13** didn't like it. **What about you?**

Start practicing

Unskip this topic

Comments (23) Useful links (0) Show discussion