Thread management

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.

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.

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

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

System.out.println("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.**

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
```

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

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

thread.join(2000L);

```
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) {
        }
    }
}

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 (the task is completed after the timeout is exceeded):

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

Fourth possible output (the task is completed after the timeout is exceeded):

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

Timed *join()* is dependent on the OS for timing. So, we cannot assume that *join()* will wait exactly as long as specified.