Theory: Custom threads

28 minutes 0 / 5 problems solved

The **main** thread is a starting place from which you may spawn new threads to perform your tasks. To do that you have to write code to be executed in a separated thread and then start it.

§1. Create custom threads

Java primarily has two ways to create a new thread that performs a task you need.

- by extending the `Thread` class and overriding its `run` method

```
class HelloThread extends Thread {

    @Override
    public void run() {
        String helloMsg = String.format("Hello, i'm %s", getName());
        System.out.println(helloMsg);
    }
}
```

- by implementing the `Runnable` interface and passing the implementation to the constructor of the `Thread` class

```
class HelloRunnable implements Runnable {

    @Override
    public void run() {
        String threadName = Thread.currentThread().getName();
        String helloMsg = String.format("Hello, i'm %s", threadName);
        System.out.println(helloMsg);
    }
}
```

In both cases, you should override the `run` method that is a regular Java method and contains code to perform a task. What approach to choose depends on the task and on your preferences. If you extend the `Thread` class, you can accept fields and methods of the base class, but you cannot extend other classes since Java doesn't have multiple-inheritance of classes.

Here are two objects obtained by the approaches described above accordingly:

```
Thread t1 = new HelloThread(); // a subclass of Thread

Thread t2 = new Thread(new HelloRunnable()); // passing runnable
```

And here's another way to specify a name of your thread by passing it to the constructor:

```
Thread myThread = new Thread(new HelloRunnable(), "my-thread");
```

If you are already familiar with lambda expressions, you may do the whole thing like this:

```
Thread t3 = new Thread(() -> {

 System.out.println(String.format("Hello, i'm %s", Thread.currentThread().getName()));
});
```

Now you've created objects for threads, but you're not done yet. To perform the tasks you need, you'll have to start them.

Current topic:

Custom threads

Topic depends on:

✕ Interface

✕ Threads as objects

Topic is required for:

Thread management

Multithreading in Swing

Sockets

Table of contents:

§2. Starting threads

The class `Thread` has a method called `start()` that is used to start a thread. At some point after you invoke this method, the method `run` will be invoked automatically, but it'll not happen immediately.

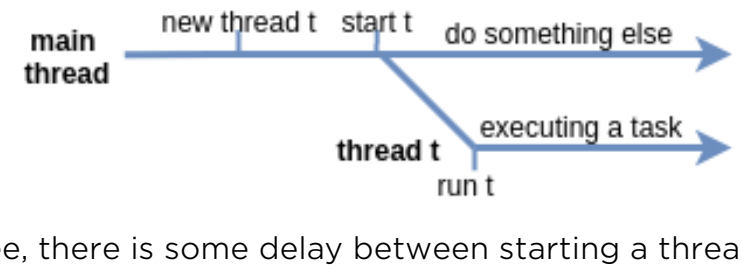
Let's suppose that inside the **main** method you create an object of `HelloThread` named `t` and start it.

```
Thread t = new HelloThread(); // an object representing a thread
t.start();
```

Eventually, it prints something like:

```
Hello, i'm Thread-0
```

Here's a picture that explains how a thread actually starts and why it is not happening immediately.



As you may see, there is some delay between starting a thread and the moment when it really starts working (running).

By default, a new thread is running in **non-daemon** mode. Reminder: the difference between the **daemon** and the **non-daemon** mode is that JVM will not terminate the running program while there're still **non-daemon** threads left, while the **daemon** threads won't prevent JVM from terminating.

Do not confuse methods `run` and `start`. You must invoke `start` if you'd like to execute your code inside `run` in another thread. If you invoke `run` directly, the code will be executed in the same thread.

If you try to start a thread more than once, the `start` throws `IllegalThreadStateException`.

Despite the fact that within a single thread all statements are executed sequentially, it is impossible to determine the relative order of statements between multiple threads without additional measures that we will not consider in this lesson.

Consider the following code:

```
public class StartingMultipleThreads {

    public static void main(String[] args) {
        Thread t1 = new HelloThread();
        Thread t2 = new HelloThread();

        t1.start();
        t2.start();


        System.out.println("Finished");

    }

}
```

The order of displaying messages may be different. Here is one of them:

```
Hello, i'm Thread-1
Finished
Hello, i'm Thread-0
```

It is even possible that all threads may print their text after the **main** thread prints "Finished":

```
Finished
Hello, i'm Thread-0
Hello, i'm Thread-1
```

This means that even though we call the `start` method sequentially for each thread, we do not know when the `run` method will be actually called.

> Do not rely on the order of execution of statements between different threads, unless you've taken special measures.

§3. A simple multithreaded program

Let's consider a simple multithreaded program with two threads. The first thread reads numbers from the standard input and prints out their squares. At the same time, the **main** thread occasionally prints messages to the standard output. Both threads work simultaneously.

Here is a thread that reads numbers in a loop and squares them. It has a break statement to stop the loop if the given number is 0.

```
class SquareWorkerThread extends Thread {
    private final Scanner scanner = new Scanner(System.in);

    public SquareWorkerThread(String name) {
        super(name);
    }

    @Override
    public void run() {

        while (true) {

            int number = scanner.nextInt();

            if (number == 0) {

                break;

            }

            System.out.println(number * number);

        }

        System.out.println(String.format("%s finished", getName()));

    }

}
```

Inside the `main` method, the program starts an object of `SquareWorkerThread` class that writes messages to the standard output from the **main** thread.

```
public class SimpleMultithreadedProgram {

    public static void main(String[] args) {
        Thread worker = new SquareWorkerThread("square-worker");
        worker.start(); // start a worker (not run!)

        for (long i = 0; i < 5_555_555_543L; i++) {
            if (i % 1_000_000_000 == 0) {
                System.out.println("Hello from the main!");

            }

        }

    }

}
```

Here is an example of inputs and outputs with comments:

```
Hello from the main!   // the program outputs it
2                      // the program reads it
4                      // the program outputs it
Hello from the main!   // outputs it
3                      // reads it
9                      // outputs it
5                      // reads it
Hello from the main!   // outputs it
25                     // outputs it

0                      // reads it

square-worker finished // outputs it

Hello from the main!   // outputs it

Hello from the main!   // outputs it



Process finished with exit code 0
```

As you can see, this program performs two tasks "at the same time": one in the main thread and another one in the worker thread. It may not be "the same time" in the physical sense, however, both tasks are given some time to be executed.

Report a typo

218 users liked this theory. 1 didn't like it. What about you?

Start practicing

Comments (14) Hints (0) Useful links (0) Show discussion