Custom Threads

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.

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

1. 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);
    }
}
```

2. 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);
    }
}
```

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 the 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()));
});
```

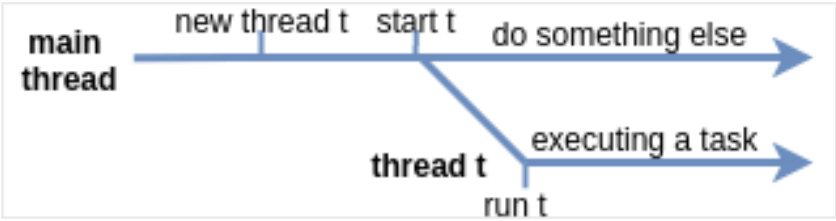
Starting threads

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.

```
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 **daemon** and **non-daemon** mode is that JVM will not terminate the running program while there are **non-daemon** threads left, whereas the **daemon** threads won't prevent the JVM from terminating.

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

If you try to start a thread more than once, the start method 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.

```
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 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 actually be called.

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

```
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()));
    }
}


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 thread!");
            }
        }
    }
}


Hello from the main thread!    // the program outputs it
2                              // the program reads it
4                              // the program outputs it
Hello from the main thread!    // outputs it
3                              // reads it
9                              // outputs it
5                              // reads it
Hello from the main thread!    // outputs it
25                              // outputs it
0                              // reads it
square-worker finished         // outputs it
Hello from the main thread!    // outputs it
Hello from the main thread!    // 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.

An instance of the class Thread has methods start and run. If we invoke the first one, it calls the second one in a new thread

Doesn't take into account the fact that sleep() is only a request and might be ignored by the OS.