Interruptions

A thread terminates when its run method stops.

Sometimes you may have to terminate a task that is being executed, for example when shutting down an application with multiple threads.

Java provides a mechanism called **interruptions** for requesting a thread to stop or to do something else. Interruption does not force a thread to react immediately but notifies it about such demand.

interrupt() and isInterrupted()

Invoking the interrupt() method on an instance of the Thread class sets its interrupted flag as true.
Thread thread = ...
thread.interrupt(); // interrupt this thread

The reaction to this event is determined by the interrupted thread itself. One common case for that is to stop the execution, although a thread may simply ignore it.

Depending on the current state of a thread, interruptions are handled differently. Invoking thread.interrupt() will cause an InterruptedException if the thread is sleeping or joining another thread. Otherwise, the only thing that will happen is that the interrupted flag will be set to true.

Here is an example of how a thread may handle an interruption:

```
public class CustomThread extends Thread {

    @Override
    public void run() {
        while (!isInterrupted()) {
            try {
                doAction();
                Thread.sleep(1000); // it may throw InterruptedException
            } catch (InterruptedException e) {
                System.out.println("sleeping was interrupted");
                break; // stop the loop
            }
        }
        System.out.printf("%s finished", getName());
    }

    private void doAction() {
```

```
        // something useful
    }
}
```

When this thread is running, an interruption may occur on any statement inside the run method, including checking the loop's condition,
performing doAction and during sleep.

If the thread is sleeping, Thread.sleep(1000) throws
an InterruptedException that is handled. In other cases, the loop is stopped according to the condition on the next iteration.

If you prefer implementing Runnable rather than extending Thread directly, you may use the static method Thread.interrupted() inside the run method. The main difference between this and the previous method is that the static method resets the interruption status to false.

An example: counting with interruption
As an example, we will consider a task that counts numbers while the thread is not interrupted.

```
class CountingTask implements Runnable {

    @Override
    public void run() {
        System.out.println("Start counting");
        int i = 1; // the first number to print

        try {
            while (!Thread.interrupted()) {
                System.out.println(i);
                i++;
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Sleeping was interrupted");
        }
        System.out.println("Finishing");
    }
}
```

In this implementation, the sleep takes almost all the time and interruption will often occur during sleeping. If the program does not print the string **"Sleeping was interrupted"** it means that the thread was interrupted during work, not sleep.
In the main method, we create a thread to perform the task and then interrupt the thread.

```
public class InterruptionDemo {

    public static void main(String[] args) throws InterruptedException {
        Thread counter = new Thread(new CountingTask());
        counter.start();
        Thread.sleep(5000L);
        counter.interrupt();
        counter.join();
    }
}
```

Note that in the main method, both the sleep and join methods may throw anInterruptedExceptionupon being interrupted. Handling this was omitted here only for brevity.

The program output is:

```
Start counting
1
2
3
4
Sleeping was interrupted
Finishing
```