Java Concurrency in Practice

Chapter-7:
Cancellation and Shutdown

Upcode Software
Engineer Team

-2023-

CONTENT

1. **Task cancellation**
2. **Interruption**
3. **Summary**
4. **Reference**

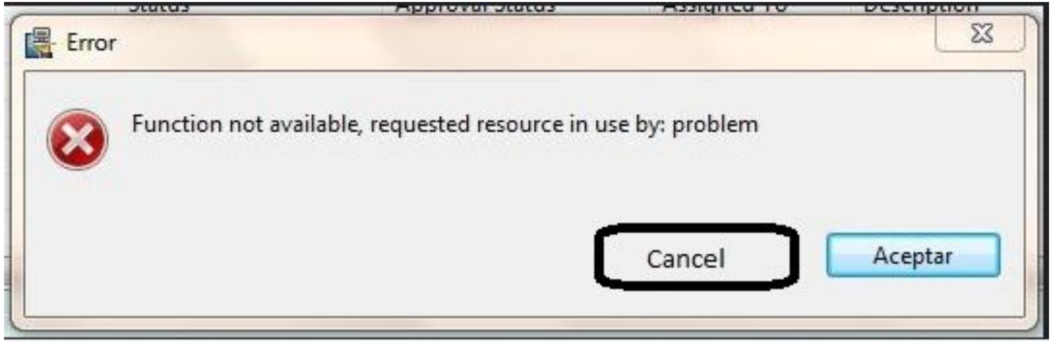
Task cancellation (1/n)

- **An activity is cancellable if external code can move it to completion before its normal completion.**
    - User-requested cancellation
    - Time-limited activities
    - Application events
    - Errors
    - Shutdown

Task cancellation (2/n) - User-requested cancellation

- **User-requested cancellation**

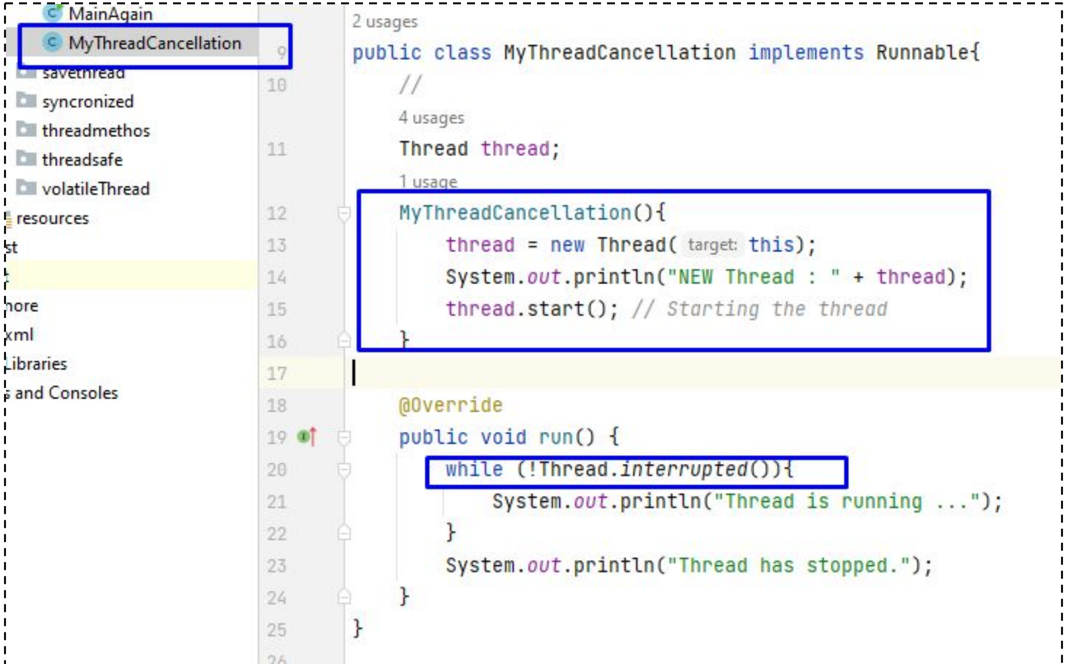
The user clicked on the “cancel” button in a GUI application, or requested cancellation through a management interface such as JMX (Java Management Extensions).



Function not available, requested resource in use by: problem

Task cancellation (3/n) - Time-limited activities

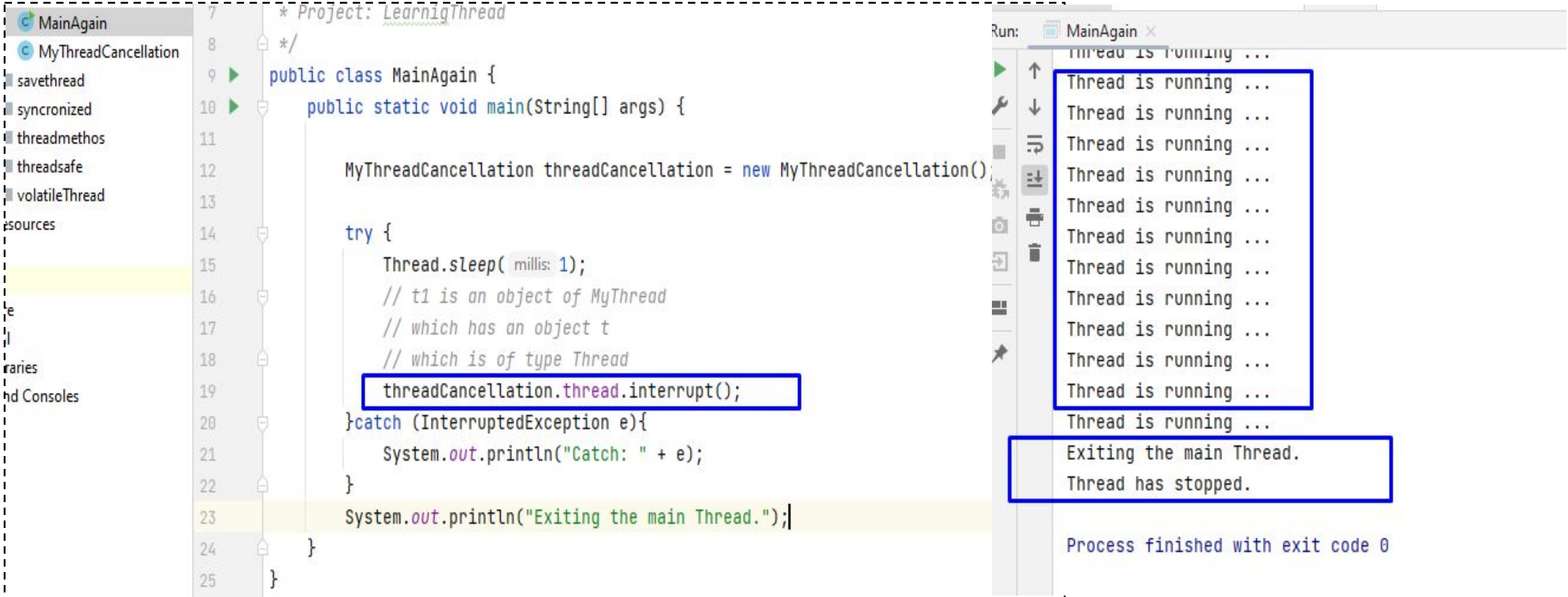
- An application searches a problem space for a finite amount of time and chooses the best solution found within that time.
- When the timer expires, any tasks still searching are cancelled.



```java
2 usages
public class MyThreadCancellation implements Runnable{
    //
    4 usages
    Thread thread;
    1 usage
    MyThreadCancellation(){
        thread = new Thread( target: this);
        System.out.println("NEW Thread : " + thread);
        thread.start(); // Starting the thread
    }

    @Override
    public void run() {
        while (!Thread.interrupted()){
            System.out.println("Thread is running ...");
        }
        System.out.println("Thread has stopped.");
    }
}
```

Task cancellation (4/n) - Application events

- An application searches a problem space by decomposing it so that different tasks search different regions of the problem space.
- When one task finds a solution, all other tasks still searching are cancelled.



```
 * Project: LearnigThread
 */
public class MainAgain {
    public static void main(String[] args) {

        MyThreadCancellation threadCancellation = new MyThreadCancellation();

        try {
            Thread.sleep( millis: 1);
            // t1 is an object of MyThread
            // which has an object t
            // which is of type Thread
            threadCancellation.thread.interrupt();
        }catch (InterruptedException e){
            System.out.println("Catch: " + e);
        }
        System.out.println("Exiting the main Thread.");
    }
}
```

```
Thread is running ...
Thread is running ...
Thread is running ...
Thread is running ...
Thread is running ...
Thread is running ...
Thread is running ...
Thread is running ...
Thread is running ...
Thread is running ...
Thread is running ...
Thread is running ...
Thread is running ...
Exiting the main Thread.
Thread has stopped.

Process finished with exit code 0
```

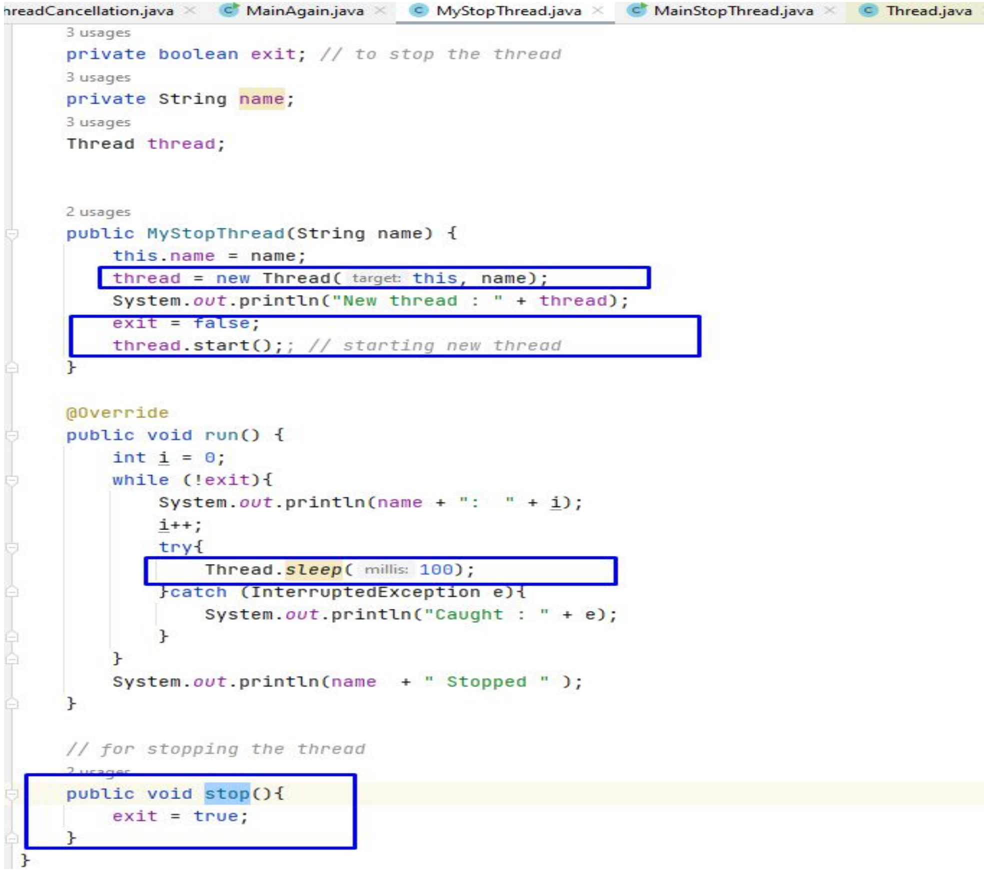
Task cancellation (5/n) - Errors

- A web crawler searches for relevant pages, storing pages or summary data to disk.
- When a crawler task encounters an error (for example, the disk is full), another crawling tasks are cancelled, possibly recording their current state so that they can be restarted later

Task cancellation (6/n) - Shutdown

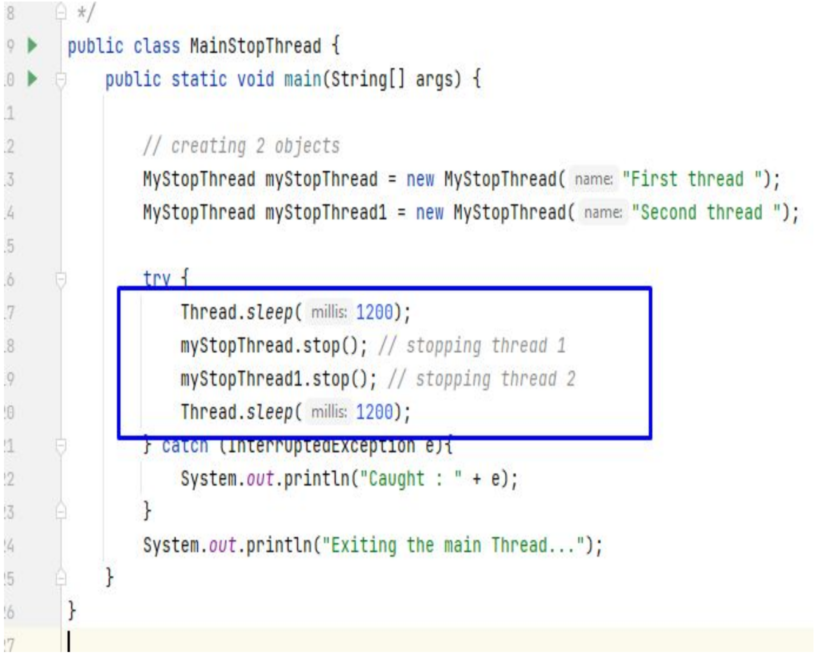
- When an application or service is shut down, something must be done about work that is currently being processed or queued for processing.
- In a graceful shutdown, tasks currently in progress might be allowed to complete;
    - in a more immediate shutdown, currently executing tasks might be cancelled.

Task cancellation (7/n) - Stop thread

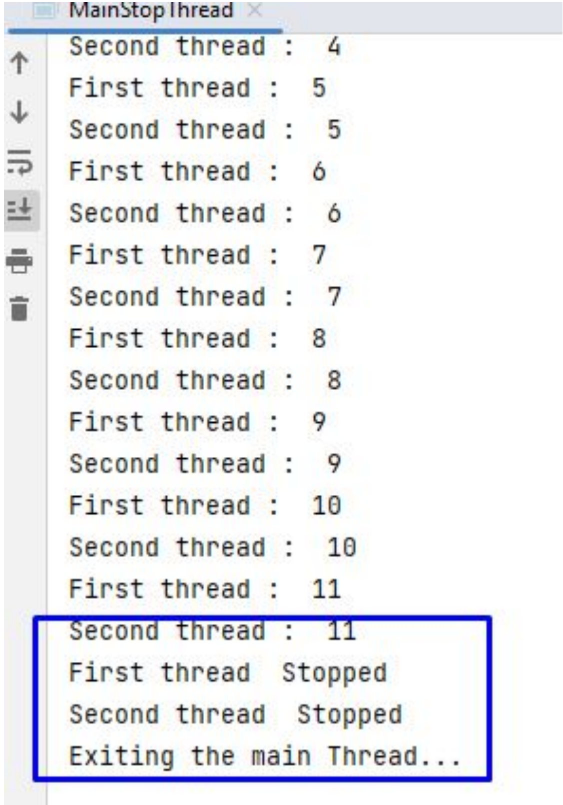


```
    3 usages
    private boolean exit; // to stop the thread
    3 usages
    private String name;
    3 usages
    Thread thread;


    2 usages
    public MyStopThread(String name) {
        this.name = name;
        thread = new Thread( target: this, name);
        System.out.println("New thread : " + thread);
        exit = false;
        thread.start();; // starting new thread
    }

    @Override
    public void run() {
        int i = 0;
        while (!exit){
            System.out.println(name + ":  " + i);
            i++;
            try{
                Thread.sleep( millis: 100);
            }catch (InterruptedException e){
                System.out.println("Caught : " + e);
            }
        }
        System.out.println(name  + " Stopped " );
    }

    // for stopping the thread
    2 usages
    public void stop(){
        exit = true;
    }
}
```

Task cancellation (7/n) - Stop thread2



```java
*/
public class MainStopThread {
    public static void main(String[] args) {

        // creating 2 objects
        MyStopThread myStopThread = new MyStopThread( name: "First thread ");
        MyStopThread myStopThread1 = new MyStopThread( name: "Second thread ");

        try {
            Thread.sleep( millis: 1200);
            myStopThread.stop(); // stopping thread 1
            myStopThread1.stop(); // stopping thread 2
            Thread.sleep( millis: 1200);
        } catch (InterruptedException e){
            System.out.println("Caught : " + e);
        }
        System.out.println("Exiting the main Thread...");
    }
}
```



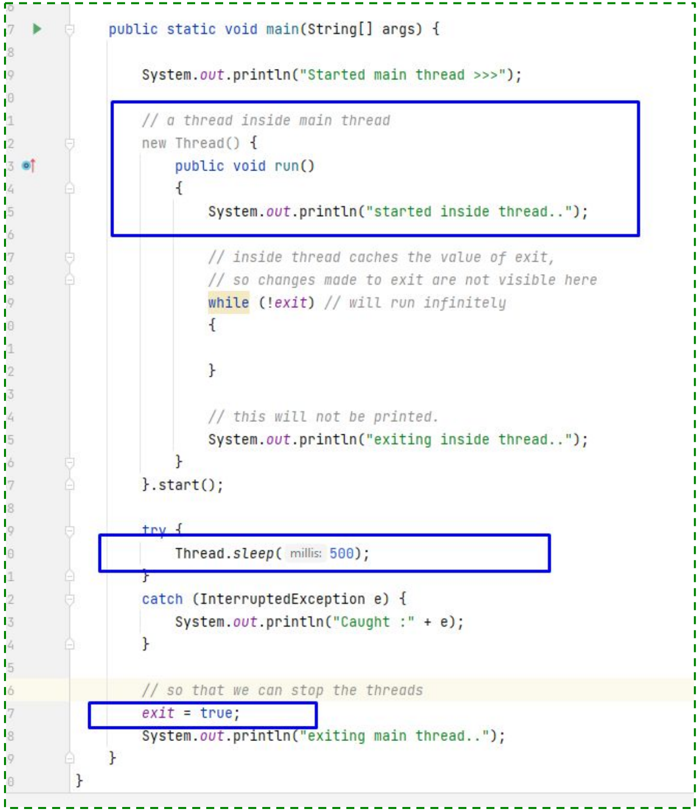
```
Second thread :  4
First thread :  5
Second thread :  5
First thread :  6
Second thread :  6
First thread :  7
Second thread :  7
First thread :  8
Second thread :  8
First thread :  9
Second thread :  9
First thread :  10
Second thread :  10
First thread :  11
Second thread :  11
First thread  Stopped
Second thread  Stopped
Exiting the main Thread...
```

Task Interruption(1/n)

- **Using a volatile boolean flag:** We can also use a volatile boolean flag to make our code thread safe.
- A volatile variable is directly stored in the main memory so that threads cannot **have locally cached values** of it.
- A situation may arise when more than one threads are accessing the same variable and the changes made by one might not be visible to other threads.
  - In such a situation, we can use a volatile boolean flag.

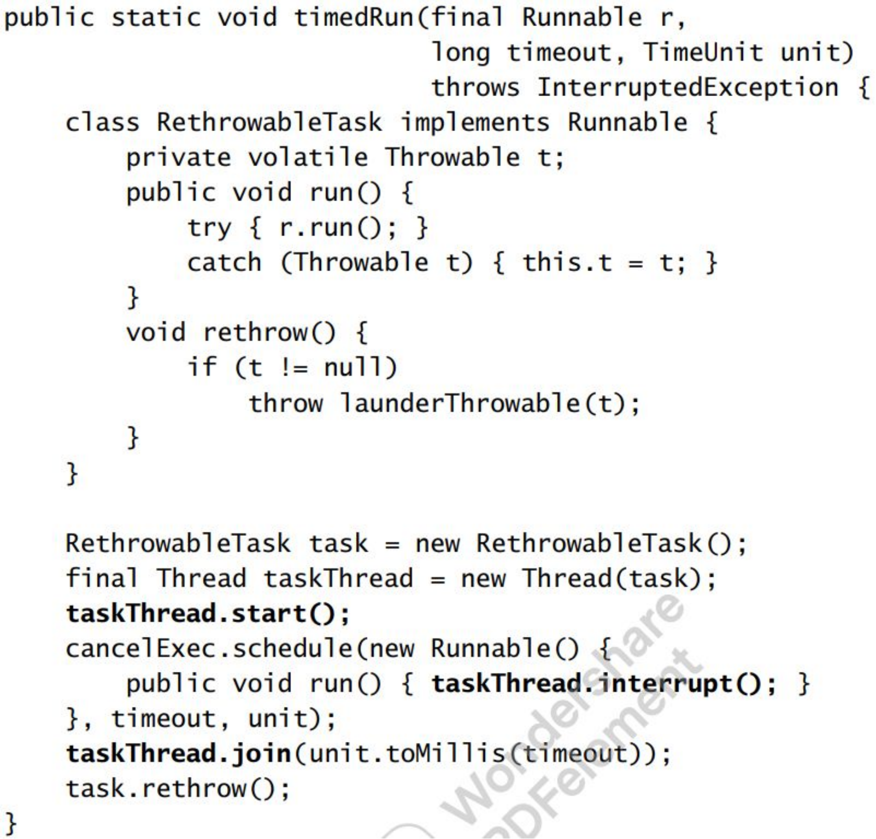
```
// Java program to illustrate non-volatile boolean flag
```

Task Interruption(2/n)



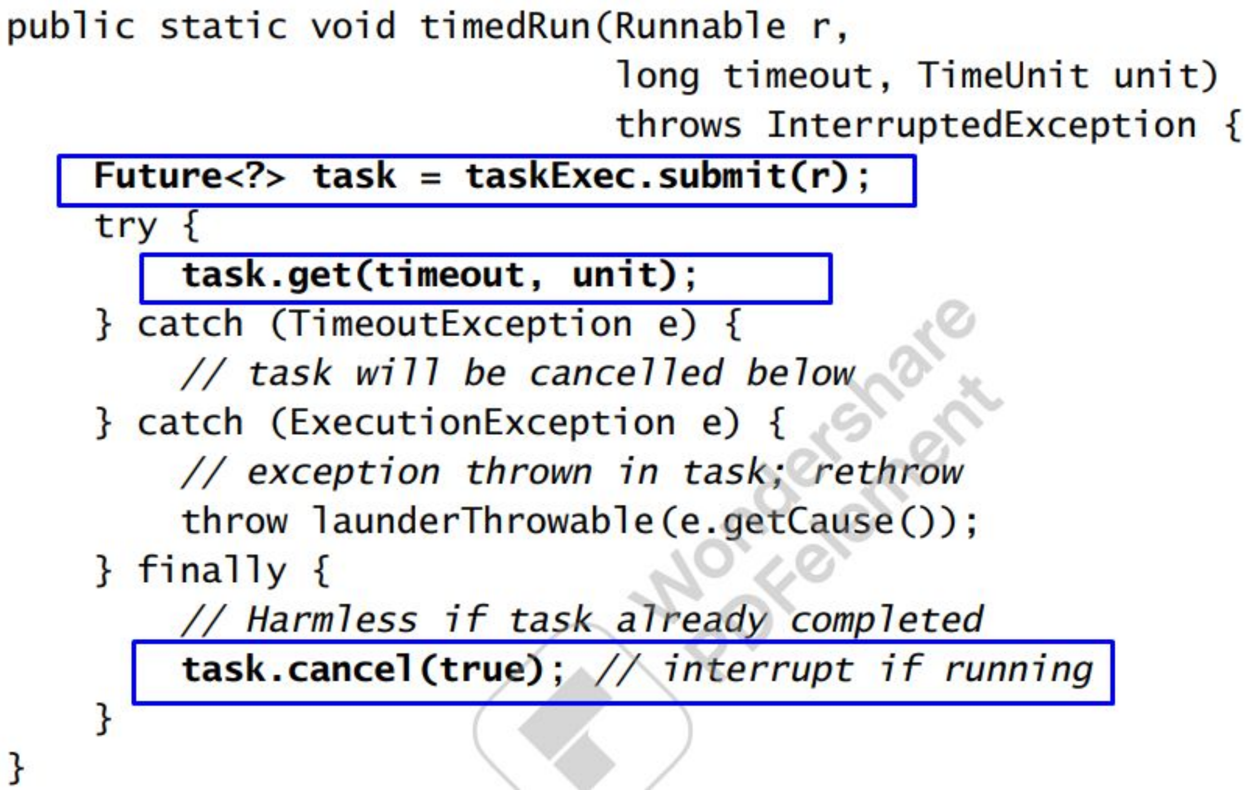
```
    public static void main(String[] args) {

        System.out.println("Started main thread >>>");

        // a thread inside main thread
        new Thread() {
            public void run()
            {
                System.out.println("started inside thread..");

                // inside thread caches the value of exit,
                // so changes made to exit are not visible here
                while (!exit) // will run infinitely
                {

                }

                // this will not be printed.
                System.out.println("exiting inside thread..");
            }
        }.start();

        try {
            Thread.sleep( millis: 500);
        }
        catch (InterruptedException e) {
            System.out.println("Caught :" + e);
        }

        // so that we can stop the threads
        exit = true;
        System.out.println("exiting main thread..");
    }
}
```

Cancellation via Future (1/n)



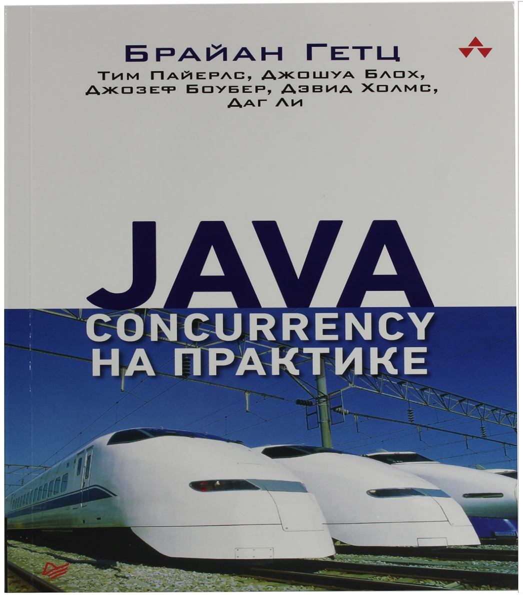
```
public static void timedRun(final Runnable r,
                            long timeout, TimeUnit unit)
                            throws InterruptedException {
    class RethrowableTask implements Runnable {
        private volatile Throwable t;
        public void run() {
            try { r.run(); }
            catch (Throwable t) { this.t = t; }
        }
        void rethrow() {
            if (t != null)
                throw launderThrowable(t);
        }
    }

    RethrowableTask task = new RethrowableTask();
    final Thread taskThread = new Thread(task);
    taskThread.start();
    cancelExec.schedule(new Runnable() {
        public void run() { taskThread.interrupt(); }
    }, timeout, unit);
    taskThread.join(unit.toMillis(timeout));
    task.rethrow();
}
```

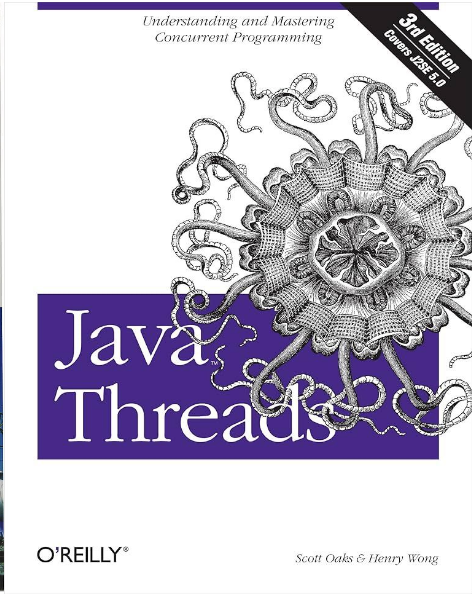
Cancellation via Future (2/n)



```
public static void timedRun(Runnable r,
                            long timeout, TimeUnit unit)
                            throws InterruptedException {
    Future<?> task = taskExec.submit(r);
    try {
        task.get(timeout, unit);
    } catch (TimeoutException e) {
        // task will be cancelled below
    } catch (ExecutionException e) {
        // exception thrown in task; rethrow
        throw launderThrowable(e.getCause());
    } finally {
        // Harmless if task already completed
        task.cancel(true); // interrupt if running
    }
}
```

Resources


Брайан Гетц
Тим Пайерлс, Джошуа Блох,
Джозеф Боубер, Дэвид Холмс,
Даг Ли
JAVA
CONCURRENCY
НА ПРАКТИКЕ


Understanding and Mastering
Concurrent Programming
3rd Edition
Covers J2SE 5.0
Java™
Threads
O'REILLY®
Scott Oaks & Henry Wong

The
Pragmatic
Programmers
Programming
Concurrency
on the JVM
Mastering
Synchronization,
STM, and Actors
Venkat Subramaniam
edited by Brian P. Hogan

Reference

1.Java Concurrency and Multithreading

2. https://flylib.com/books/en/2.558.1/stopping_a_thread_based_service.html

3. https://www.jasonfilippou.com/blog/proper-shutdown-of-a-scheduled-executor-service

4. https://codeahoy.com/java/Cancel-Tasks-In-Executors-Threads/

Thank you!

Presented by

Hamdamboy Urunov

(hamdamboy.urunov@gmail.com)