Managing the Java Thread Lifecycle: Stopping a Thread via an Interrupt

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Know various ways to stop Java threads
  - Stopping a thread with a volatile flag
  - Stopping a thread with an interrupt request
Stopping Java Threads with an Interrupt Request
Stopping Java Threads with an Interrupt Request

• A thread can be stopped voluntarily by calling its interrupt() method

See docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#interrupt
A thread can be stopped voluntarily by calling its interrupt() method.

Posts an *interrupt request* to a thread.

---

**Interrupts**

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson.

A thread sends an interrupt by invoking `interrupt` on the `Thread` object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

See [docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html](http://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html)
Stopping Java Threads with an Interrupt Request

- A thread can be stopped voluntarily by calling its interrupt() method
  - Posts an *interrupt request* to a thread
- Interrupts are implemented via an internal *interrupt status* flag

---

**Interrupts**

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson.

A thread sends an interrupt by invoking *interrupt* on the *Thread* object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.
Stopping Java Threads with an Interrupt Request

- A thread can be stopped voluntarily by calling its interrupt() method
  - Posts an *interrupt request* to a thread
- Interrupts are implemented via an internal *interrupt status* flag
  - Invoking Thread.interrupt() sets this flag

Interruption

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson.

A thread sends an interrupt by invoking *interrupt on the* Thread object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.
### Stopping Java Threads with an Interrupt Request

- A thread can be stopped voluntarily by calling its `interrupt()` method
  - Posts an *interrupt request* to a thread
- Interrupts are implemented via an internal *interrupt status* flag
  - Invoking `Thread.interrupt()` sets this flag
  - Programs can check this flag via two thread accessor methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>interrupted()</code></td>
<td>Tests whether the current thread has been interrupted</td>
</tr>
<tr>
<td><code>isInterrupted()</code></td>
<td>Tests whether this thread has been interrupted</td>
</tr>
</tbody>
</table>

Each method has different side-effects on interrupt status, as discussed shortly.
Here’s a simple Java program that starts, runs, & interrupts a background thread:

```java
static int main(String args[]) {
    Thread t1 =
        new Thread(() -> {
            for (int i = 0;
                 i < args.length; i++) {
                processBlocking(args[i]);
                processNonBlocking(args[i]);
            }
        });
    t1.start();
    ... // Run concurrently for a while
    t1.interrupt();
    ...
}
```

Stopping Java Threads with an Interrupt Request
Here’s a simple Java program that starts, runs, & interrupts a background thread:

```java
static int main(String args[]) {
    Thread t1 =
        new Thread(() -> {
            for (int i = 0;
                 i < args.length; i++) {
                processBlocking(args[i]);
                processNonBlocking(args[i]);
            }
        });

    t1.start();
    ... // Run concurrently for a while
    t1.interrupt();
    ...
```

Create a new thread.
Stopping Java Threads with an Interrupt Request

Here’s a simple Java program that starts, runs, & interrupts a background thread:

```java
static int main(String args[]) {
    Thread t1 =
        new Thread(() -> {
            for (int i = 0;
                i < args.length; i++) {
                processBlocking(args[i]);
                processNonBlocking(args[i]);
            }
        });
    t1.start();
    ... // Run concurrently for a while
    t1.interrupt();
    ...
}
```

Start the new thread
Here's a simple Java program that starts, runs, & interrupts a background thread

```java
static int main(String args[]) {
    Thread t1 =
        new Thread(() -> {
            for (int i = 0;
                 i < args.length; i++) {
                processBlocking(args[i]);
                processNonBlocking(args[i]);
            }
        });

    t1.start();
    ... // Run concurrently for a while
    t1.interrupt();
    ...
}
```

The main thread continues running
Here’s a simple Java program that starts, runs, & interrupts a background thread:

```java
static int main(String args[]) {
    Thread t1 =
        new Thread(() -> {
            for (int i = 0;
                 i < args.length; i++) {
                processBlocking(args[i]);
                processNonBlocking(args[i]);
            }
        });
    t1.start();
    ... // Run concurrently for a while
    t1.interrupt();
    ...
}
```

After the thread starts, it runs this lambda expression, whose methods perform blocking & non-blocking computations.
Here's a simple Java program that starts, runs, & interrupts a background thread:

```java
static int main(String args[]) {
    Thread t1 =
        new Thread(() -> {
            for (int i = 0; i < args.length; i++) {
                processBlocking(args[i]);
                processNonBlocking(args[i]);
            }
        });

t1.start();
...
```
Here’s a simple Java program that starts, runs, & interrupts a background thread:

```java
static int main(String args[]) {
    Thread t1 =
        new Thread(() -> {
            for (int i = 0;
                 i < args.length; i++) {
                processBlocking(args[i]);
                processNonBlocking(args[i]);
            }
        });

    t1.start();
    ... // Run concurrently for a while
    t1.interrupt();
    ...
}
```

Methods running in thread t1 check periodically to see if the thread’s been stopped yet.
void processBlocking(String args) {
    ...
    while (true) {
        try {
            Thread.currentThread().
sleep(interval);
            synchronized(this) {
                while (someConditionFalse)
                    wait();
            }
        }
        catch (InterruptedException e)
        { ... }
    }
    ...
}

See praveer09.github.io/technology/2015/12/06/understanding-thread-interruption-in-java

Stopping Java Threads with an Interrupt Request

- Certain blocking operations in the Java language & class libraries return automatically & throw InterruptedException if the thread is interrupted

  e.g., wait(), join(), sleep() & blocking I/O calls on "interruptable channels"
Methods whose operations do not block must periodically check if Thread.interrupt() has been called.

```java
void processNonBlocking(String args) {
    ...
    while (true) {
        ... // Long-running computation
        if (Thread.interrupted())
            throw new InterruptedException();
    ...
}
```

`interrupted()` is a static method that returns true if the calling thread has its interrupt status flag set.

`interrupted()` clears the current thread’s interrupt status the first time it’s called.
Methods whose operations do not block must periodically check if Thread.interrupt() has been called.

```java
void processNonBlocking(String args) {
    ...
    while (true) {
        ...
        // Long-running computation
        if (Thread.interrupted())
            throw new InterruptedException();
    ...
```

This example explicitly throws an InterruptedException, which is created/treated like a normal object.

See docs.oracle.com/javase/8/docs/api/java/lang/InterruptedException.html
void processNonBlocking(String args) {
    ...
    final myThread = Thread.currentThread();

    while (true) {
        ... // Long-running computation
        if (myThread.isInterrupted())
            throw new InterruptedException();
        ...
    }

    **isInterrupted() is a non-static method that returns true if the designated thread has its interrupt status flag set**

/**

**isInterrupted() can be called multiple times w/out affecting its interrupt status**

}
Stopping Java Threads with an Interrupt Request

- Programs can override thread interrupt methods since they are virtual
  - e.g., `interrupt()`, `interrupted()`, & `isInterrupted()`

```java
public class BeingThread extends Thread {
    volatile boolean mInterrupted;

    BeingThread(Runnable runnable) {
        super(runnable);
        mInterrupted = false;
    }

    public void interrupt() {
        mInterrupted = true;
        super.interrupt();
    }

    public boolean isInterrupted() {
        return mInterrupted || super.isInterrupted();
    }
}
```

Stops Java Threads with an Interrupt Request

- Programs can override thread interrupt methods since they are virtual
  - e.g., interrupt(), interrupted(), & isInterrupted()

```
public class BeingThread extends Thread {
    volatile boolean mInterrupted;

    BeingThread(Runnable runnable) {
        super(runnable);
        mInterrupted = false;
    }

    public void interrupt() {
        mInterrupted = true;
        super.interrupt();
    }

    public boolean isInterrupted() {
        return mInterrupted || super.isInterrupted();
    }
}
```

But make sure you know what you’re doing...
End of Managing the Java Thread Lifecycle: Stopping a Thread via an Interrupt