Learning Objectives in this Part of the Lesson

- Understand the Java thread lifecycle & its various states
- Recognize the steps involved in starting a Java thread
- Know various ways to stop Java threads
Learning Objectives in this Part of the Lesson

• Understand the Java thread lifecycle & its various states
• Recognize the steps involved in starting a Java thread
• Know various ways to stop Java threads
  • Stopping a thread with a volatile flag
Stopping Java Threads
Stopping Java Threads

• It may be necessary to stop a Java thread for various reasons
Stopping Java Threads

• It may be necessary to stop a Java thread for various reasons, e.g.
  • Users may want to cancel a long-running operation
    • e.g., they get bored or tired of waiting for it to complete
Stopping Java Threads

- It may be necessary to stop a Java thread for various reasons, e.g.
  - Users may want to cancel a long-running operation
  - An activity is destroyed, stopped, or paused
    - e.g., due to runtime configuration changes or pressing the “back” button

The GCD Concurrent app contains an (intentional) design flaw where it “leaks” threads when an orientation change occurs

See github.com/douglascraigschmidt/POSA/tree/master/ex/M3/GCD/Concurrent
Stopping Java Threads

- It may be necessary to stop a Java thread for various reasons, e.g.
  - Users may want to cancel a long-running operation
  - An activity is destroyed, stopped, or paused
  - Cancel other “speculative computations” after results are found
    - e.g., The ExecutorService invokeAny() method will cancel other threads after a result is found

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#invokeAny
Stopping Java Threads

• Stopping Java threads is surprisingly hard
Stopping Java Threads

• Stopping Java threads is surprisingly hard
  • i.e., the “Sorcerer’s Apprentice” problem

See [www.youtube.com/watch?v=5rzyuY8-Ao8](www.youtube.com/watch?v=5rzyuY8-Ao8)
Stopping Java Threads

• There’s no safe way to stop a Java thread involuntarily

See docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html
Stopping Java Threads

• There’s no safe way to stop a Java thread involuntarily
  • The stop() method is deprecated since it’s inherently unsafe

See [geekexplains.blogspot.com/2008/07/why-stop-suspend-resume-of-thread-are.html](geekexplains.blogspot.com/2008/07/why-stop-suspend-resume-of-thread-are.html)
Stopping Java Threads

• There’s no safe way to stop a Java thread involuntarily
  • The stop() method is deprecated since it’s inherently unsafe, e.g.
  • All locked monitors are unlocked as the exception propagates up the stack

Call Stack

```
method1()

\[\text{calls}\]

method2() throws IOException {…}

\[\text{calls}\]

method3() throws IOException {…}

\[\text{calls}\]

method4() throws IOException {…}
```
Stopping Java Threads

• There’s no safe way to stop a Java thread involuntarily
  • The stop() method is deprecated since it’s inherently unsafe, e.g.
    • All locked monitors are unlocked as the exception propagates up the stack
  • Any objects protected by these monitors are thus left in an inconsistent state

```
method1()

method2() throws IOException {…}

method3() throws IOException {…}

method4() throws IOException {…}
```

Call Stack
Stopping Java Threads

• Long running operations in a thread must be coded to stop *voluntarily*!

```java
public void run(){
    while (true) {
        // Check to see
        // if the thread
        // should stop
    }
}
```
Stopping Java Threads

• There are two ways to stop a Java thread voluntarily
Stopping Java Threads

- There are two ways to stop a Java thread voluntarily
- Use a volatile flag

```java
public class MyRunnable implements Runnable {
    private volatile boolean mIsStopped = false;

    public void stopMe() {
        mIsStopped = true;
    }

    public void run() {
        while(mIsStopped != true) {
            // a long-running operation
        }
    }
}

See en.wikipedia.org/wiki/Volatile_variable#In_Java
```
Stopping Java Threads

• There are two ways to stop a Java thread voluntarily
  • Use a volatile flag
  • Use Java thread interrupt requests

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](docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html)
Stopping Java Threads with a Volatile Flag
Stopping Java Threads with a Volatile Flag

• One way to stop a Java thread is to use a “stop” flag

```java
public class MyRunnable
    implements Runnable
{
    private volatile boolean mIsStopped = true;

    public void stopMe()
    {
        mIsStopped = false;
    }

    public void run()
    {
        while(mIsStopped != true)
        {
            // a long-running operation
        }
        ...
    }
}
```
Stopping Java Threads with a Volatile Flag

- One way to stop a Java thread is to use a “stop” flag, e.g.
  - Add a volatile boolean flag “mIsStopped” to a class that implements Runnable
    ```java
    public class MyRunnable implements Runnable {
        private volatile boolean mIsStopped = false;

        public void stopMe() {
            mIsStopped = true;
        }

        public void run() {
            while (mIsStopped != true) {
                // a long-running operation
            }
            ...
    }
    ```
One way to stop a Java thread is to use a “stop” flag, e.g.

- Add a volatile boolean flag “mIsStopped” to a class that implements Runnable
- Volatile ensures changes to a variable are consistent & visible to other threads atomically

```java
public class MyRunnable
    implements Runnable
{
    private volatile boolean mIsStopped = false;

    public void stopMe()
    {
        mIsStopped = true;
    }

    public void run()
    {
        while(mIsStopped != true) {
            // a long-running operation
        }
    }
}
```

See en.wikipedia.org/wiki/Volatile_variable#In_Java
Stopping Java Threads with a Volatile Flag

• One way to stop a Java thread is to use a “stop” flag, e.g.
  
  - Add a volatile boolean flag “mIsStopped” to a class that implements Runnable
  
  - Add a stopMe() method that sets “mIsStopped” to true
  
  • volatile essentially defines a “critical section” for the mIsStopped field

```java
public class MyRunnable implements Runnable {
    private volatile boolean mIsStopped = false;

    public void stopMe() {
        mIsStopped = true;
    }

    public void run() {
        while(mIsStopped != true) {
            // a long-running operation
        }
        ...
    }
}
```

However, reads & writes to a volatile variable do not block threads
Stopping Java Threads with a Volatile Flag

- One way to stop a Java thread is to use a “stop” flag, e.g.
  - Add a volatile boolean flag “mIsStopped” to a class that implements Runnable
  - Add a stopMe() method that sets “mIsStopped” to true
  - Check “mIsStopped” periodically to see if thread’s been stopped

```java
public class MyRunnable implements Runnable {
    private volatile boolean mIsStopped = false;

    public void stopMe() {
        mIsStopped = true;
    }

    public void run() {
        while(mIsStopped != true) {
            // a long-running operation
        }
    }
}
```
Stopping Java Threads with a Volatile Flag

• One way to stop a Java thread is to use a “stop” flag, e.g.
  public class MyRunnable
   implements Runnable
   {
      private volatile boolean
       mIsStopped = false;

      public void stopMe() {
         mIsStopped = true;
      }

      public void run() {
         while(mIsStopped != true) {
            // a long-running operation
         }
         return;
   }

• Add a volatile boolean flag “mIsStopped” to a class that implements Runnable
• Add a stopMe() method that sets “mIsStopped” to true
• Check “mIsStopped” periodically to see if thread’s been stopped
• Return from the run() method when the thread’s been stopped
• Although this solution is lightweight, it isn’t transparently integrated into the Java execution environment.

```java
public class MyRunnable
    implements Runnable
{
    private volatile boolean mIsStopped = false;

    public void stopMe() {
        mIsStopped = true;
    }

    public void run() {
        while (mIsStopped != true) {
            // a long-running operation
        }
        return;
    }
}
```
Stopping Java Threads with a Volatile Flag

• Although this solution is lightweight, it isn’t transparently integrated into the Java execution environment

• e.g., blocking operations won’t be awakened, which impedes shutdown processing

```java
public class MyRunnable
    implements Runnable
{
    private volatile boolean
        mIsStopped = false;

    public void stopMe() {
        mIsStopped = true;
    }

    public void run() {
        while(mIsStopped != true) {
            // a long-running operation
        }
        return;
    }
}
```
End of Managing the Java Thread Lifecycle (Part 3)
Managing the Java Thread Lifecycle

(Part 4)

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 Module

• Understand the Java thread lifecycle & its various states
• Recognize the steps involved in starting a Java thread
• Know various ways to stop Java threads
  • Stopping a thread with a volatile flag
  • Stopping a thread with an interrupt request
Learning Objectives in this Part of the Module

- Understand the Java thread lifecycle & its various states
- Recognize the steps involved in starting a Java thread
- Know various ways to stop Java threads
  - Stopping a thread with a volatile flag
  - Stopping a thread with an interrupt request
- Understand differences between a Java thread interrupt & a hardware/OS interrupt
Learning Objectives in this Part of the Module

- Understand the Java thread lifecycle & its various states
- Recognize the steps involved in starting a Java thread
- Know various ways to stop Java threads
  - Stopping a thread with a volatile flag
  - Stopping a thread with an interrupt request
- Understand differences between a Java thread interrupt & a hardware/OS interrupt
- Learn the patterns of interrupting Java threads

Java theory and practice: Dealing with InterruptedException

Many Java language methods, such as Thread.sleep() and Object.wait(), throw InterruptedException. You can ignore it because it's a checked exception, but what should you do with it? In this module, Java theory and practice, contributor Brian Goetz explains what InterruptedException means, why it is thrown, and what you should do when you catch it.

Block me out. How about loop (while)...

This story is probably familiar: You're writing a test program and you need to pause for some amount of time, so you call Thread.sleep(). But then the compiler or IDE balks that you haven't dealt with the checked InterruptedException. What is InterruptedException, and why do you have to deal with it?

The most common response to InterruptedException is to swallow it — catch it and do nothing (or perhaps log it, which isn't any better) — as we've seen later in Listing 4. Unfortunately, this approach throws away important information about the fact that an interrupt occurred, which could compromise the application's ability to cancel activities or shut down in a timely manner.

Blocking methods

When a method throws InterruptedException, it's telling you several things in addition to the fact that it can throw a particular checked exception. It's telling you that it's a blocking method and that it will make an attempt to unlock and return early — if you ask nicely.

A blocking method is different from an ordinary method that just takes a long time to run. The completion of an ordinary method is dependent only on how much work you've asked it to do and whether adequate computing resources (CPU cycles and memory) are available. The completion of a blocking method, on the other hand, is also dependent on some external event, such as thread expiration, IO completion, or the action of another thread (releasing a lock, writing a flag, or placing a task on a work queue). Ordinary methods complete as soon as their work can be done, but blocking methods are less predictable because they depend on external events. Blocking methods can compromise responsiveness because it can be hard to predict when they will complete.

Because blocking methods can potentially take forever if the event they are waiting for never occurs, it's often useful for blocking operations to be cancellable. (It is often useful for long-running non-blocking methods to be cancellable as well.) A cancellable operation is one that can be externally moved to completion in advance of when it would ordinarily complete on its own. The interruption mechanism provided by Thread and supported by Thread.sleep() and Object.wait() is a cancellation mechanism; it allows an thread to request that another thread stop what it is doing early. When a method throws InterruptedException, it's telling you that the thread executing the method is interrupted, it will make an attempt to stop what it is doing and return early and indicates its early return by throwing InterruptedException. Well-behaved blocking methods should be responsive to interruption and throw InterruptedException so they can be used within cancellable activities without compromising responsiveness.

Thread interruption

Every thread has a boolean property associated with it that represents its interrupted status. The interrupted status is initially false; when a thread is interrupted by some other thread through a call to Thread.interrupt(), one of two things happens: if the thread is executing a low-level interruptible blocking method like Thread.sleep(), Thread.join(), or Object.wait(), it unlocks and throws InterruptedException. Otherwise, interrupt() merely sets the thread's interruption status. Code running in the interrupted thread can then poll the interrupted status to see if it has been requested to stop what it is doing; if the interrupted status can be read with Thread.isInterrupted() and can be read and cleared in a single operation with the poorly named Thread.interrupt().

InterruptedException is a cooperative mechanism. When a thread interrupts another, the interrupted thread does not necessarily stop what it is doing immediately. Instead, interruption is a way of politely asking another thread to stop what it is doing if it wants to, at its convenience. Some methods, like Thread.sleep(), take this request seriously, but methods are not required to pay attention to interruption. Methods that do not block but that still may take a long time to execute can request respectful interruption by polling the interrupted status and return early if interrupted. You are free to ignore an interruption request, but doing so may compromise responsiveness.

One of the benefits of the cooperative nature of interruption is that it provides more flexibility for safely constraining cancellable activities. We rarely want an activity to stop immediately; program data structures could be left in an inconsistent state if the activity were canceled mid-update. Interruption allows a cancellable activity to clean up any work in progress, restore invariants, notify other activities of the cancellation...
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.

**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.

See [docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html](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.
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

**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
- Programs can check this flag via two Thread accessor methods

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

Each method has different side-effects on the interrupted status, as shown in upcoming examples
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();
    ...
```
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();
    ...
}
```
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();
    ...
}
```
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();
    ... // Run concurrently for a while
    t1.interrupt();
    ...
}
```

**After the main thread performs some computations it interrupts thread t₁**

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 \( t_1 \) check periodically to see if the thread's been stopped.
void processBlocking(String args) {
    ...
    while (true) {
        try {
            Thread.currentThread().
            sleep(interval);
            synchronized(this) {
                while (someConditionFalse)
                    wait();
            }
        }
        catch (InterruptedException e) {
            ...
        }
    }
    catch (InterruptedException e) {
        ...
    }
}

e.g., wait(), join(), sleep() & blocking I/O calls on “interruptable channels”

Certain blocking operations will return automatically & throw the Interrupted Exception if the thread is interrupted
Here’s a simple Java program that starts, runs, & interrupts a background thread

```java
void processNonBlocking(String args) {
    ... // 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.

Non-blocking operations must periodically check if `Thread.interrupt()` has been called.
Stopping Java Threads with an Interrupt Request

- Here's a simple Java program that starts, runs, & interrupts a background thread

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

This method clears the interrupt status of the current thread the first time it's called
Here’s a simple Java program that starts, runs, & interrupts a background thread.

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

This example explicitly throws a new `InterruptedException`.

A Java exception is allocated & treated like a normal object.
**Stopping Java Threads with an Interrupt Request**

- Here's a simple Java program that starts, runs, & interrupts a background thread

```java
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 on a thread without 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();
    }
}
```

See stackoverflow.com/questions/23369891/overriding-interrupt-isinterrupted-method-in-thread-class
Programs can override thread interrupt methods since they are virtual
• e.g., interrupt(), interrupted(), & isInterrupted()
Patterns of Stopping Java Threads via Interrupts
There are patterns for dealing with Java InterruptedException

Patterns of Stopping Java Threads via Interrupts

- There are patterns for dealing with Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it
  - Caller(s) must then handle the exception properly

```java
public class StringBlockingQueue {
    private BlockingQueue<String> queue = new LinkedBlockingQueue<String>();

    public void put(String s) throws InterruptedException {
        queue.put(s);
    }

    public String take() throws InterruptedException {
        return queue.take();
    }
}
```

See docs.oracle.com/javase/tutorial/essential/exceptions/declaring.html
Patterns of Stopping Java Threads via Interrupts

- There are patterns for dealing with Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it
  - Caller(s) must then handle the exception properly

```
public class StringBlockingQueue {
    private BlockingQueue<String>
        queue = new
        LinkedBlockingQueue<String>();

    public void put(String s) throws InterruptedException {
        queue.put(s);
    }

    public String take() throws InterruptedException {
        return queue.take();
    }
}
```

Note how the exception is explicitly listed in the method’s “throw clause”

See docs.oracle.com/javase/tutorial/essential/exceptions/declaring.html
Patterns of Stopping Java Threads via Interrupts

• There are patterns for dealing with Java InterruptedException, e.g.
  • Propagate InterruptedException to callers by not catching it
  • Perform task-specific cleanup before rethrowing
  • Avoid leaking resources or leaving resources in an inconsistent state

```java
if (mustWait) {
    try {
        lock.wait();
    }
    catch (InterruptedException e) {
        synchronized (this) {
            boolean removed = mWaitQueue.remove(lock);
            if (!removed)
                release();
        }
        throw e;
    }
}
...
```
There are patterns for dealing with Java InterruptedException, e.g.

- Propagate InterruptedException to callers by not catching it
- Perform task-specific cleanup before rethrowing
- Restore interrupted status after catching InterruptedException
- Preserve evidence that the exception occurred for higher levels of the call stack

```java
class RunnableExample {
    private Queue<Runnable> queue = new ArrayDeque<>();

    public void run() {
        try {
            while (true) {
                Runnable r = queue.take(10, TimeUnit.SECONDS);
                r.run();
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        catch (InterruptedException e) {
            ... // Perform task-specific cleanup before rethrowing
            Thread.currentThread().interrupt();
        }
    }
}
```

See [daniel.mitterdorfer.name/articles/2015/handling-interruptedexception](daniel.mitterdorfer.name/articles/2015/handling-interruptedexception)
Patterns of Stopping Java Threads via Interrupts

- There are patterns for dealing with Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it
  - Perform task-specific cleanup before rethrowing
  - Restore interrupted status after catching InterruptedException
  - Handle the interrupt & “swallow” it

```java
public boolean gaze() {
    int sleepTime = 0;
    try {
        sleepTime = 1000 +
                    mRandom.nextInt(4000);
        Thread.sleep(sleepTime);
        return true;
    }
    catch (InterruptedException e) {
        return false;
    }
}
```
Patterns of Stopping Java Threads via Interrupts

- There are patterns for dealing with Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it
  - Perform task-specific cleanup before rethrowing
  - Restore interrupted status after catching InterruptedException
  - Handle the interrupt & “swallow” it
    - e.g., often done when the Thread sleep() or join() methods are called

```java
public boolean gaze() {
    int sleepTime = 0;
    try {
        sleepTime = 1000 +
                   mRandom.nextInt(4000);
        Thread.sleep(sleepTime);
        return true;
    }
    catch (InterruptedException e) {
        return false;
    }
}
```

General-purpose reusable library code should never swallow interrupt requests entirely (i.e., this is an “anti-pattern”)
Java Thread Interrupts vs. Hardware/OS Interrupts
Java Thread Interrupts vs Hardware/OS Interrupts

- Interrupts at the hardware or OS layers have several properties.

<table>
<thead>
<tr>
<th>Interrupt Process (from three potential sources)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware</strong></td>
</tr>
<tr>
<td>Interrupt Request (IRQ) sent from device to processor</td>
</tr>
<tr>
<td><strong>Processor</strong></td>
</tr>
<tr>
<td>Exception / Trap sent from processor to processor</td>
</tr>
<tr>
<td>Processor halts thread execution</td>
</tr>
<tr>
<td>Processor saves thread state</td>
</tr>
<tr>
<td>Processor executes interrupt handler</td>
</tr>
<tr>
<td>Processor resumes thread execution</td>
</tr>
<tr>
<td><strong>Software</strong></td>
</tr>
<tr>
<td>Software Interrupt instruction loaded by processor</td>
</tr>
</tbody>
</table>

Java Thread Interrupts vs Hardware/OS Interrupts

• Interrupts at the hardware or OS layers have several properties
  • **Asynchronous**
    • Can occur essentially anytime & are independent of the instruction currently running

Java Thread Interrupts vs Hardware/OS Interrupts

- Interrupts at the hardware or OS layers have several properties
  - **Asynchronous**
    - Can occur essentially anytime & are independent of the instruction currently running
    - A program needn’t test for them explicitly since they occur “out-of-band”

See vujungle.blogspot.com/2010/12/differentiate-synchronous-and.html
Java Thread Interrupts vs Hardware/OS Interrupts

- Interrupts at the hardware or OS layers have several properties
  - Asynchronous
  - Preemptive
    - Pause (& then later resume) the execution of currently running code without its cooperation

See [en.wikipedia.org/wiki/Preemption_(computing)](en.wikipedia.org/wiki/Preemption_(computing))
This example shows how to catch the UNIX SIGINT signal.

- It occurs asynchronously.
- It preempts the current instruction.
- It needn’t be tested for explicitly.

The SIGINT interrupt is typically generated by typing `^C` in a UNIX shell.

See [www.thegeekstuff.com/2012/03/catch-signals-sample-c-code](www.thegeekstuff.com/2012/03/catch-signals-sample-c-code)

```c
void sig_handler(int signo) {
    if (signo == SIGINT)
        printf("received SIGINT\n");
}

int main(void) {
    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("can't catch SIGINT\n");
    for (;;) {
        sleep(10);
    }
    return 0;
}
```

Java Thread Interrupts vs Hardware/OS Interrupts

The SIGINT interrupt is typically generated by typing `^C` in a UNIX shell.
Java Thread Interrupts vs Hardware/OS Interrupts

• Asynchronous & preemptive interrupt handling make it hard to reason about programs

See en.wikipedia.org/wiki/Unix_signal#Risks
Asynchronous & preemptive interrupt handling make it hard to reason about programs, e.g.:

- Race conditions

Race conditions occur when a program depends on the sequence or timing of threads for it to operate properly.

See en.wikipedia.org/wiki/Race_condition#Software
Java Thread Interrupts vs Hardware/OS Interrupts

- Asynchronous & preemptive interrupt handling make it hard to reason about programs, e.g.
  - Race conditions
  - Re-entrancy problems

A non-reentrant function cannot be interrupted in the middle of its execution & then safely called again before its previous invocations complete execution

See [en.wikipedia.org/wiki/Reentrancy_(computing)](en.wikipedia.org/wiki/Reentrancy_(computing))
Java Thread Interrupts vs Hardware/OS Interrupts

- Asynchronous & preemptive interrupt handling make it hard to reason about programs, e.g.
  - Race conditions
  - Re-entrancy problems
  - Non-transparent restarts

"e.g., an I/O operation returns the # of bytes transferred & it is up to the application to check this & manage its own resumption of the operation until all the bytes have been transferred"

See [en.wikipedia.org/wiki/PCLSRing](en.wikipedia.org/wiki/PCLSRing) #Unix-solution:_restart_on_request
Java Thread Interrupts vs Hardware/OS Interrupts

- Java thread interrupts differ from hardware or operating system interrupts.

See [docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html](docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html)
### Java Thread Interrupts vs Hardware/OS Interrupts

- Java thread interrupts differ from hardware or operating system interrupts, e.g.
  - Delivery is *synchronous* & *non-preemptive* rather than *asynchronous* & *preemptive*
  - i.e., they don’t occur at an arbitrary point & don’t pause (& later resume) running code
Java Thread Interrupts vs Hardware/OS Interrupts

- Java thread interrupts differ from hardware or operating system interrupts, e.g.
  - Delivery is *synchronous* & *non-preemptive* rather than *asynchronous* & *preemptive*
- A program must test for them explicitly

```java
void processNonBlocking(String input) {
  ...
  while (true) {
    ... // Do some long-running
    // computation
    if (Thread.interrupted())
      throw new
        InterruptedException();
  ...
}
Java Thread Interrupts vs Hardware/OS Interrupts

- Java thread interrupts differ from hardware or operating system interrupts, e.g.
  - Delivery is **synchronous** & **non-preemptive** rather than **asynchronous** & **preemptive**
- A program must test for them explicitly
  - i.e., `InterruptedException` is (usually) thrown synchronously & is handled synchronously

```java
void processNonBlocking(String input) {
    ...
    while (true) {
        ... // Do some long-running computation
        if (Thread.interrupted())
            throw new InterruptedException();
    ...
```

Java Thread Interrupts vs Hardware/OS Interrupts

- Java thread interrupts differ from hardware or operating system interrupts, e.g.
  - Delivery is *synchronous* & *non-preemptive* rather than *asynchronous* & *preemptive*
- A program must test for them explicitly
- Certain operations cannot be interrupted
  - e.g., blocking I/O calls that aren’t “interruptable channels”

```java
static class SleeperThread extends Thread {
    public void run() {
        int c;
        try {
            c = System.in.read();
        }
        ...
    }
}
```

Java Thread Interrupt
Parting Thoughts
Java Thread Interrupt Parting Thoughts

- Portable solutions for stopping Java threads require cooperation
Java Thread Interrupt Parting Thoughts

• Portable solutions for stopping Java threads require cooperation
  • Threads must check periodically to see if they’ve been told to stop
Java Thread Interrupt Parting Thoughts

- Portable solutions for stopping Java threads require cooperation
  - Threads must check periodically to see if they've been told to stop
- Thread interrupts are fragile since they require all parts of a program follow consistent usage patterns

See weblogs.java.net/blog/2009/03/02/cancelling-tasks-threadinterrupt-fragility
Portable solutions for stopping Java threads require cooperation

- Threads must check periodically to see if they’ve been told to stop
- Thread interrupts are fragile since they require all parts of a program follow consistent usage patterns
- Voluntary checking is tedious & error-prone, but it’s the only way to halt Java threads reliably

See stackoverflow.com/questions/8505707/android-best-and-safe-way-to-stop-thread
End of Managing the Java Thread Lifecycle (Part 4)
Managing the Java Thread Lifecycle
(Part 5)

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 Module

- Understand the Java thread lifecycle & its various states
- Recognize the steps involved in starting a Java thread
- Know various ways to stop Java threads
- Learn how the GCD Interrupted program works

Runtime Behavior of the GCD Interrupted App
Runtime Behavior of the GCD Interrupted App

- Use a thread to compute the greatest common divisor (GCD) of two numbers, which is the largest positive integer that divides two integers without a remainder.

The user can interrupt the GCD computation at any point.
Implementation of the GCD Interrupted App
Implementation of the GCD Interrupted App

- This app showcases the Java Thread start() & interrupt() methods

See https://github.com/douglas-craig-schmidt/POSA/tree/master/ex/M3/GCD/Interrupted
Implementation of the GCD Interrupted App

• This app showcases the Java Thread start() & interrupt() methods

Super class that automatically logs lifecycle hook method invocations to aid debugging
Implementation of the GCD Interrupted App

- This app showcases the Java Thread start() & interrupt() methods

Start & interrupts a Java thread that repeatedly computes the GCD of two random numbers
Implementation of the GCD Interrupted App

• This app showcases the Java Thread start() & interrupt() methods

Runs in a thread repeatedly computing the GCD of two numbers in a manner that can be interrupted
End of Managing the Java Thread Lifecycle (Part 5)