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
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/7/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](docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html)
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

---

**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>static boolean</th>
<th>interrupted() – Tests whether the current thread has been interrupted</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>isInterrupted() – 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
• 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();
    ...
```

Stop the 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();
...}
```
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();
    ...
}
```

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) {
        ... }
    ...
}

Certain blocking operations will return automatically & throw the InterruptedException if the thread is interrupted

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

`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.
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
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 example explicitly throws a new InterruptedException.

A Java exception is allocated & treated like a normal object.
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()

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…
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>

Interrupts at the hardware or OS layers have several properties

- **Asynchronous**
  - Can occur essentially anytime & are independent of the instruction currently running

Interrupts at the hardware or OS layers have several properties

- **Asynchronous**
  - Can occur essentially anytime and 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
• 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)
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.

```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;
}
```

See [www.thegeekstuff.com/2012/03/catch-signals-sample-c-code](http://www.thegeekstuff.com/2012/03/catch-signals-sample-c-code)
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
Java Thread Interrupts vs Hardware/OS Interrupts

- 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](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

See [en.wikipedia.org/wiki/Reentrancy_(computing)](en.wikipedia.org/wiki/Reentrancy_(computing))

A non-reentrant function cannot be interrupted in the middle of its execution & then safely called again before its previous invocations complete execution.
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](https://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”

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

See bugs.java.com/bugdatabase/view_bug.do?bug_id=4514257
Patterns of Stopping Java Threads via Interrupts
Patterns of Stopping Java Threads via Interrupts

- There are patterns for dealing with Java InterruptedException

See www.ibm.com/developerworks/java/library/j-jtp05236/index.html?ca=drs-
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](https://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();
    }
}
```

See [docs.oracle.com/javase/tutorial/essential/exceptions/declaring.html](http://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;
    }
    ...
```
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
  • Preserve evidence that the exception occurred for higher levels of the call stack

```java
public void run() {
    try {
        while (true) {
            Runnable r =
                       queue.take(10,
                                  TimeUnit.SECONDS);
            r.run();
        }
    }
    catch (InterruptedException e){
        ...
        Thread.currentThread().
        interrupt();
    }
}
```

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