Overview of Java Threads

(Part 2)

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

- Understand how Java threads support concurrency
- Learn how our case study app works
- Know alternative ways of giving code to a thread
- Learn how to pass parameters to a Java thread
- Know how to run a Java thread
• Understand how Java threads support concurrency
• Learn how our case study app works
• Know alternative ways of giving code to a thread
• Learn how to pass parameters to a Java thread
• Know how to run a Java thread
• Recognize common thread methods
Learning Objectives in this Part of the Lesson

• Understand how Java threads support concurrency
• Learn how our case study app works
• Know alternative ways of giving code to a thread
• Learn how to pass parameters to a Java thread
• Know how to run a Java thread
• Recognize common thread methods
• Appreciate Java thread “happens-before” orderings
Running Java Threads
There are multiple layers involved in creating & starting a thread

Runnable Java Threads

- Operating System Kernel
- System Libraries
- Java Execution Environment (e.g., JVM, ART, etc)
- Threading & Synchronization Packages

See Part 2 of the upcoming lesson on “Managing the Java Thread Lifecycle”
Running Java Threads

• There are multiple layers involved in creating & starting a thread
• Creating a new thread object doesn’t allocate a run-time call stack of activation records

See en.wikipedia.org/wiki/Call_stack
Running Java Threads

- There are multiple layers involved in creating & starting a thread
  - Creating a new thread object doesn’t allocate a run-time call stack of activation records
  - The runtime stack & other thread resources are only allocated after the start() method is called
Running Java Threads

- There are multiple layers involved in creating & starting a thread
  - Creating a new thread object doesn’t allocate a run-time call stack of activation records
  - The runtime stack & other thread resources are only allocated after the start() method is called
- The Java execution environment calls a thread’s run() hook method after start() creates its resources

See wiki.c2.com/?HookMethod
Running Java Threads

• There are multiple layers involved in creating & starting a thread
  • Creating a new thread object doesn’t allocate a run-time call stack of activation records
  • The runtime stack & other thread resources are only allocated after the start() method is called
  • The Java execution environment calls a thread’s run() hook method after start() creates its resources
  • Each thread can run concurrently & block independently
• Any code can generally run in a thread

```java
public void run(){
    // code to run goes here
}
```
Any code can generally run in a thread.
However, windowing toolkits often restrict which thread can access GUI components.

Running Java Threads:

- My Component
- MyThread

Diagram:
- onCreate()
- new()
- start()
- run()
Running Java Threads

- Any code can generally run in a thread
- However, windowing toolkits often restrict which thread can access GUI components
  - e.g., only the Android UI thread can access GUI components

See [developer.android.com/training/multiple-threads/communicate-ui.html](http://developer.android.com/training/multiple-threads/communicate-ui.html)
Running Java Threads

- A thread can live as long as its run() hook method hasn’t returned
Running Java Threads

- A thread can live as long as its `run()` hook method hasn’t returned
- The underlying thread scheduler can suspend & resume a thread many times during its lifecycle

See en.wikipedia.org/wiki/Scheduling_(computing)
Running Java Threads

- A thread can live as long as its run() hook method hasn’t returned.
- The underlying thread scheduler can suspend & resume a thread many times during its lifecycle.

These scheduler operations are largely transparent to user code, as long as synchronization is performed properly.
• For a thread to execute “forever,” its `run()` hook method needs an infinite loop

```java
public void run()
{
    while (true) {
        ... 
    }
}
```
Running Java Threads

- The thread is dead after run() returns

```
new() : MyThread
onCreate() => start()
```

Diagram shows:
-.onCreate()
- new()
- start()
- run()
**Running Java Threads**

- The thread is dead after `run()` returns
- A thread can end normally

```java
public void run(){
    while (true) {
        ...
        return;
    }
}
```
Running Java Threads

- The thread is dead after run() returns
  - A thread can end normally
  - Or an uncaught exception can be thrown

```java
public void run(){
    while (true) {
        // ...
        throw new SomeException();
    }
}
```
Running Java Threads

- The `join()` method allows one thread to wait for another thread to complete.
Running Java Threads

- The `join()` method allows one thread to wait for another thread to complete.

**Component: My Component**

**Thread: MyThread**

```
: My Component
- new()
  -> run()
  -> start()
  -> new()

: MyThread
- join()
```

*Simple form of “barrier synchronization”*

See upcoming lessons on “Java Barrier Synchronizers”
Running Java Threads

- The `join()` method allows one thread to wait for another thread to complete.
- Or a thread can simply evaporate!
Running Java Threads

• The `join()` method allows one thread to wait for another thread to complete
  • Or a thread can simply evaporate!
• The Java execution environment recycles thread resources
Running Java Threads

- The `join()` method allows one thread to wait for another thread to complete.
- Or a thread can simply evaporate!
- The Java execution environment recycles thread resources.
  - e.g., runtime stack of activation records, thread-specific storage, etc.
Some Common Java Thread Methods
Some Common Java Thread Methods

- There are a number of commonly used methods in the Java Thread class.

See [docs.oracle.com/javase/8/docs/api/java/lang/Thread.html](https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html)
Some Common Java Thread Methods

- There are a number of commonly used methods in the Java Thread class, e.g.,
  - `void setDaemon()`
    - Marks thread as a “daemon”

See javarevisited.blogspot.com/2012/03/what-is-daemon-thread-in-java-and.html
Some Common Java Thread Methods

- There are a number of commonly used methods in the Java Thread class, e.g.,
  - `void setDaemon()`
  - `void start()`
    - Allocates thread resources & initiates thread execution by calling the run() hook method
Some Common Java Thread Methods

- There are a number of commonly used methods in the Java Thread class, e.g.,
  - `void setDaemon()`  
  - `void start()`  
  - `void run()`  
    - Hook method where user code is supplied
Some Common Java Thread Methods

• There are a number of commonly used methods in the Java Thread class, e.g.,
  • `void setDaemon()`  
  • `void start()`  
  • `void run()`  
  • `void join()`  
  • Waits for a thread to finish

A simple form of "barrier synchronization"
Some Common Java Thread Methods

- There are a number of commonly used methods in the Java Thread class, e.g.,
  - void setDaemon()
  - void start()
  - void run()
  - void join()
  - void sleep(long time)
    - Sleeps for given time in ms
Some Common Java Thread Methods

- There are a number of commonly used methods in the Java Thread class, e.g.,
  - `void setDaemon()`
  - `void start()`
  - `void run()`
  - `void join()`
  - `void sleep(long time)`
- `Thread currentThread()`
  - Object for current Thread
Some Common Java Thread Methods

- There are a number of commonly used methods in the Java Thread class, e.g.,
  - `void setDaemon()`  
  - `void start()`  
  - `void run()`  
  - `void join()`  
  - `void sleep(long time)`  
  - `Thread currentThread()`  
  - `void interrupt()`  

   • Post an interrupt request to a Thread  

See part 3 of upcoming lesson on “Managing the Java Thread Lifecycle”
Some Common Java Thread Methods

- There are a number of commonly used methods in the Java Thread class, e.g.,
  - `void setDaemon()`
  - `void start()`
  - `void run()`
  - `void join()`
  - `void sleep(long time)`
  - `Thread currentThread()`
  - `void interrupt()`
  - `boolean isInterrupted()`
    - Tests whether a thread has been interrupted

`isInterrupted()` can be called multiple times w/out affecting the `interrupted status`
Some Common Java Thread Methods

- There are a number of commonly used methods in the Java Thread class, e.g.,
  - `void setDaemon()`
  - `void start()`
  - `void run()`
  - `void join()`
  - `void sleep(long time)`
  - `Thread currentThread()`
  - `void interrupt()`
  - `boolean isInterrupted()`
  - `boolean interrupted()`
    - Tests whether current thread has been interrupted

interrupted() clears the *interrupted status* the first time it’s called
Some Common Java Thread Methods

- There are a number of commonly used methods in the Java Thread class, e.g.,
  - `void setDaemon()`
  - `void start()`
  - `void run()`
  - `void join()`
  - `void sleep(long time)`
  - `Thread currentThread()`
  - `void interrupt()`
  - `boolean isInterrupted()`
  - `boolean interrupted()`
  - `void setPriority(int newPriority)`
  - `int getPriority()`
- Set & get the priority of a Thread
Java Thread “Happens-Before” Orderings
Java Thread “Happens-Before” Orderings

• Java Threads methods establish “happens-before” orderings

See en.wikipedia.org/wiki/Happened-before
Java Thread “Happens-Before” Orderings

- Java Threads methods establish “happens-before” orderings
- Ensure that if one event “happens before” another event, the result must reflect that, even if those events are actually executed out of order

See en.wikipedia.org/wiki/Happened-before
Java Thread “Happens-Before” Orderings

- Java Threads methods establish “happens-before” orderings
  - Ensure that if one event “happens before” another event, the result must reflect that, even if those events are actually executed out of order
  - e.g., to optimize program flow & concurrency
Java Thread “Happens-Before” Orderings

- Java Threads methods establish “happens-before” orderings
  - Ensure that if one event “happens before” another event, the result must reflect that, even if those events are actually executed out of order
  - In general, a happens-before relationship guarantees that memory written to by statement $A$ is visible to statement $B$

- $i.e.$, statement $A$ completes its write to “ready” before statement $B$ starts its read
Java Thread “Happens-Before” Orderings

- Examples of “happens-before” orderings in Java

Examples of “happens-before” orderings in Java

Starting a thread “happens-before” the run() hook method of the thread is called

```java
Thread t1 =
    new Thread(() ->
        System.out.println
            ("hello world"))
    .start();
```
Examples of “happens-before” orderings in Java:

- Starting a thread “happens-before” the run() hook method of the thread is called, e.g.

```java
t1 = new Thread(() ->
    System.out.println("hello world"))
    .start();
```

This lambda plays the role of the run() hook method!
Java Thread “Happens-Before” Orderings

- Examples of “happens-before” orderings in Java
- Starting a thread “happens-before” the run() hook method of the thread is called, e.g.

```java
Thread t1 =
    new Thread(() ->
        System.out.println("hello world")
    ).start();
```

A thread’s state is consistent & visible before run() starts.
Examples of “happens-before” orderings in Java

- Starting a thread “happens-before” the run() hook method of the thread is called
- Methods in java.util.concurrent package classes also establish “happen-before” orderings

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html
Examples of “happens-before” orderings in Java

- Starting a thread “happens-before” the run() hook method of the thread is called
- Methods in java.util.concurrent package classes also establish “happen-before” orderings, e.g.

```java
// Thread t1
ConcurrentMap concurrentMap =
    new ConcurrentHashMap();
concurrentMap.put("key", "value");

// Thread t2
Object value = concurrentMap.get("key");
```

Placing an object into a concurrent collection happens-before the access or removal of the element from the collection
Java Thread “Happens-Before” Orderings

- Examples of “happens-before” orderings in Java
  - Starting a thread “happens-before” the run() hook method of the thread is called
  - Methods in java.util.concurrent package classes also establish “happen-before” orderings
  - The termination of a thread “happens-before” a join() with the terminated thread

```java
Thread t1 =
    new Thread(() ->
            System.out.println("hello world"))
    .start();

t1.join();
```
### Java Thread “Happens-Before” Orderings

- **Examples of “happens-before” orderings in Java**
  - Starting a thread “happens-before” the `run()` hook method of the thread is called
  - Methods in `java.util.concurrent` package classes also establish “happens-before” orderings
  - The termination of a thread “happens-before” a `join()` with the terminated thread, e.g.

```java
Thread t1 =
    new Thread(() ->
        System.out.println("hello world"))
    .start();

t1.join();
```

This thread terminates after its lambda expression runnable completes
• Examples of “happens-before” orderings in Java
  • Starting a thread “happens-before” the run() hook method of the thread is called
  • Methods in java.util.concurrent package classes also establish “happen-before” orderings
  • The termination of a thread “happens-before” a join() with the terminated thread, e.g.

```java
Thread t1 =
    new Thread(() ->
        System.out.println
            ("hello world"))
    .start();

t1.join();
```

A thread waiting on a (non-timed) join() only resumes after the target thread terminates
Java Thread “Happens-Before” Orderings

• The implementations of these Java thread & library classes are responsible for ensuring that these “happens-before” orderings are preserved.

You don’t need to understand all the nitty-gritty details of Java’s memory model - you just need to understand how to use synchronizers properly!
End of Overview of Java Threads (Part 2)