Exploring How Java Threads Run

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
Running
Java Threads
Running Java Threads

- There are multiple layers involved in creating & starting a thread

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

: My Component
: MyThread

onCreate() → new() → start() → run()

See the upcoming lessons 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 much/any system state
    - e.g., it doesn’t allocate any kernel resources
Running Java Threads

- There are multiple layers involved in creating & starting a thread
  - Creating a new thread object doesn’t allocate much/any system state
  - The runtime stack & other kernel resources are only allocated after the start() method is called

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 much/any system state
  • The runtime stack & other kernel resources are only allocated after the start() method is called

The start() method can only be called once per thread object
Running Java Threads

- There are multiple layers involved in creating & starting a thread
  - Creating a new thread object doesn’t allocate much/any system state
  - The runtime stack & other kernel 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 much/any system state
  - The runtime stack & other kernel 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
Running Java Threads

- Any code can generally run in a thread

```java
public void run(){
    // code to run goes here
}
```
Running Java Threads

• Any code can generally run in a thread
• However, windowing toolkits often restrict which thread can access GUI components
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
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)](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
• Scheduler operations are largely invisible 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

: My Component

: MyThread

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) {
        ...
        if (someCondition())
            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) {
        ...
        if (someError())
            throw new SomeException();
    }
}
```

See [www.javamex.com/tutorials/exceptions/exceptions_uncaught_handler.shtml](http://www.javamex.com/tutorials/exceptions/exceptions_uncaught_handler.shtml)
Running Java Threads

• The join() method allows one thread to wait for another thread to complete

```
MyComponent

start()

new()

run()

join()

onCreate()

start()

new()

join()

MyThread
```
Running Java Threads

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

```java
new MyThread()
start()
run()
join()
```

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.
End of Exploring How Java Threads Run