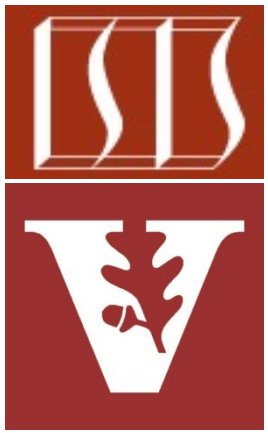
How Java Threads Start & Run

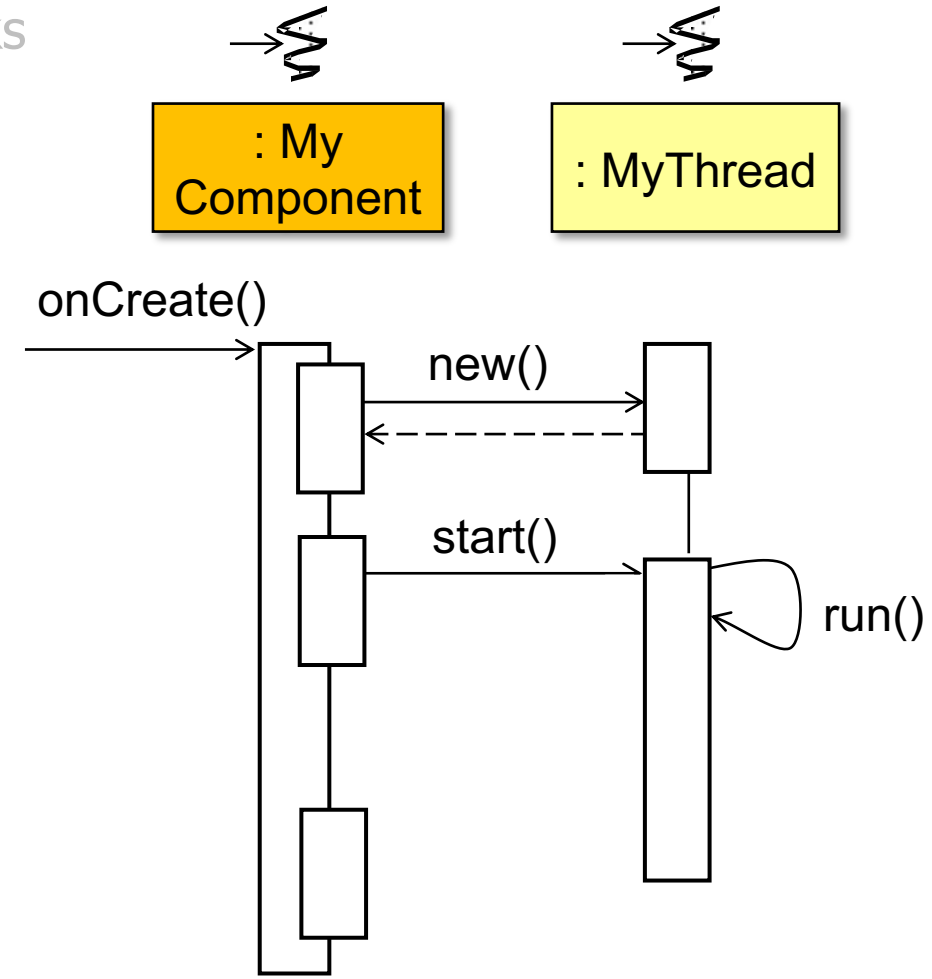


Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

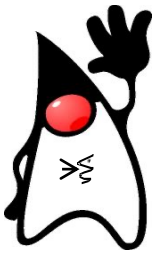
Learning Objectives in this Part of the Lesson

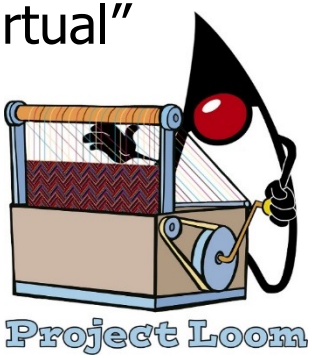
- Understand how Java threads support concurrency
- Learn how our case study app works
- Know alternative ways of giving code to a thread
- Learn how to pass parameters to a Java thread
- Know the differences between Java platform & virtual threads
- Be aware of how a Java thread starts & runs



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 the differences between Java platform & virtual threads
- Be aware of how a Java thread starts & runs
  - Including traditional "platform" threads & modern "virtual" threads





Platform threads

`Thread` supports the creation of *platform threads* that are typically mapped 1:1 to kernel threads scheduled by the operating system. Platform threads will usually have a large stack and other resources that are maintained by the operating system. Platforms threads are suitable for executing all types of tasks but may be a limited resource.

Platform threads are designated *daemon* or *non-daemon* threads. When the Java virtual machine starts up, there is usually one non-daemon thread (the thread that typically calls the application's `main` method). The Java virtual machine terminates when all started non-daemon threads have terminated. Unstarted daemon threads do not prevent the Java virtual machine from terminating. The Java virtual machine can also be terminated by invoking the Runtime.exit(int) method, in which case it will terminate even if there are non-daemon threads still running.

In addition to the daemon status, platform threads have a thread priority and are members of a thread group.

Platform threads get an automatically generated thread name by default.

Virtual threads

`Thread` also supports the creation of *virtual threads*. Virtual threads are typically *user-mode threads* scheduled by the Java virtual machine rather than the operating system. Virtual threads will typically require few resources and a single Java virtual machine may support millions of virtual threads. Virtual threads are suitable for executing tasks that spend most of the time blocked, often waiting for I/O operations to complete. Virtual threads are not intended for long running CPU intensive operations.

Virtual threads typically employ a small set of platform threads used as *carrier threads*. Locking and I/O operations are the *scheduling points* where a carrier thread is re-scheduled from one virtual thread to another. Code executing in a virtual thread will usually not be aware of the underlying carrier thread, and in particular, the currentThread() method, to obtain a reference to the *current thread*, will return the `Thread` object for the virtual thread, not the underlying carrier thread.

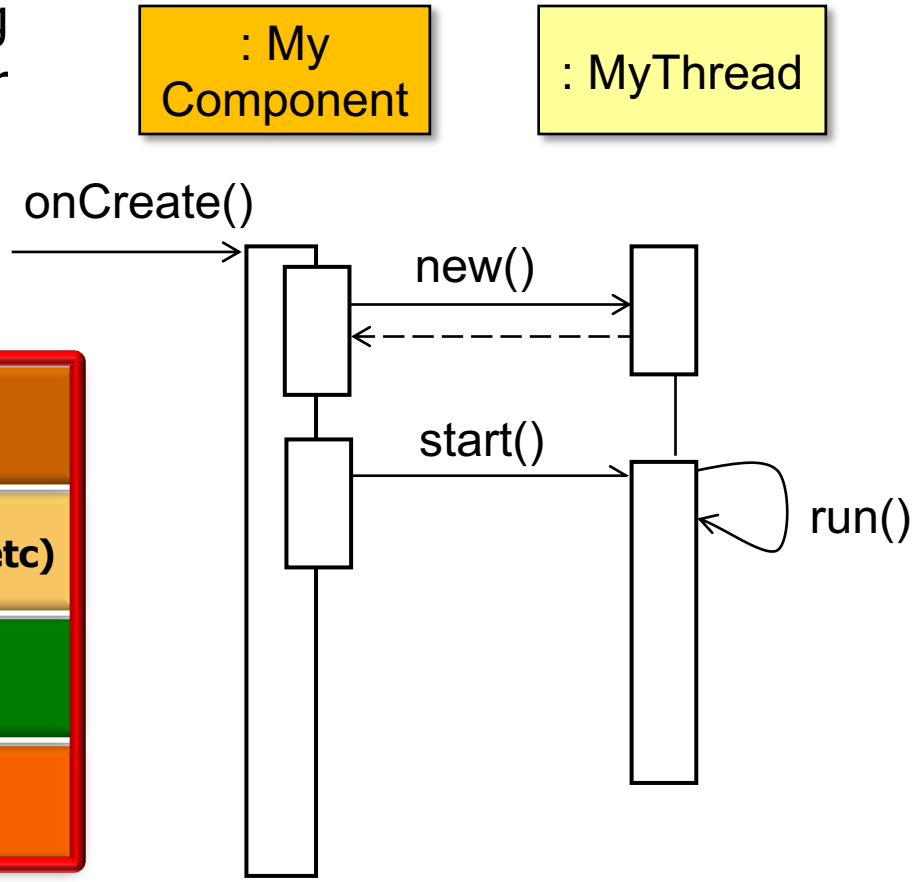
Virtual threads gets a fixed name by default.

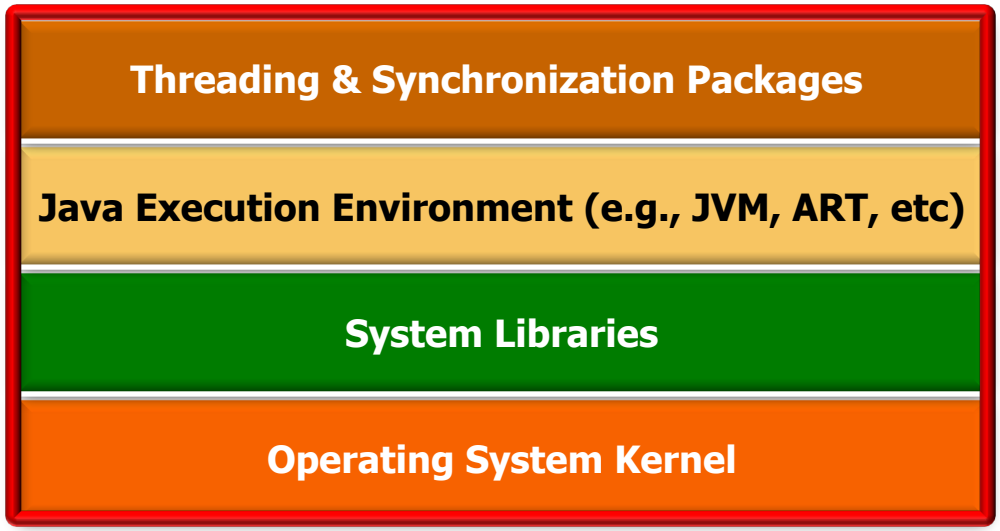
See download.java.net/java/early_access/loom/docs/api/java.base/java/lang/Thread.html

Starting Java Platform Threads

Starting Java Platform Threads

- Multiple layers are involved in creating & starting a traditional Java thread (or new platform thread)

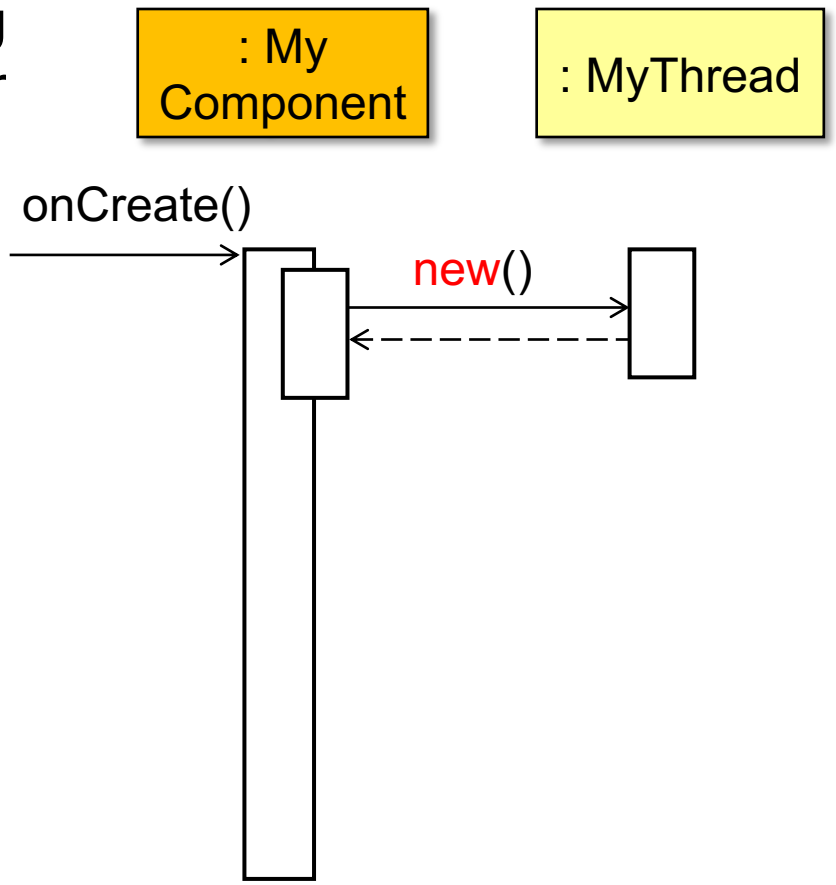




See the upcoming lessons on "*Managing the Java Thread Lifecycle*"

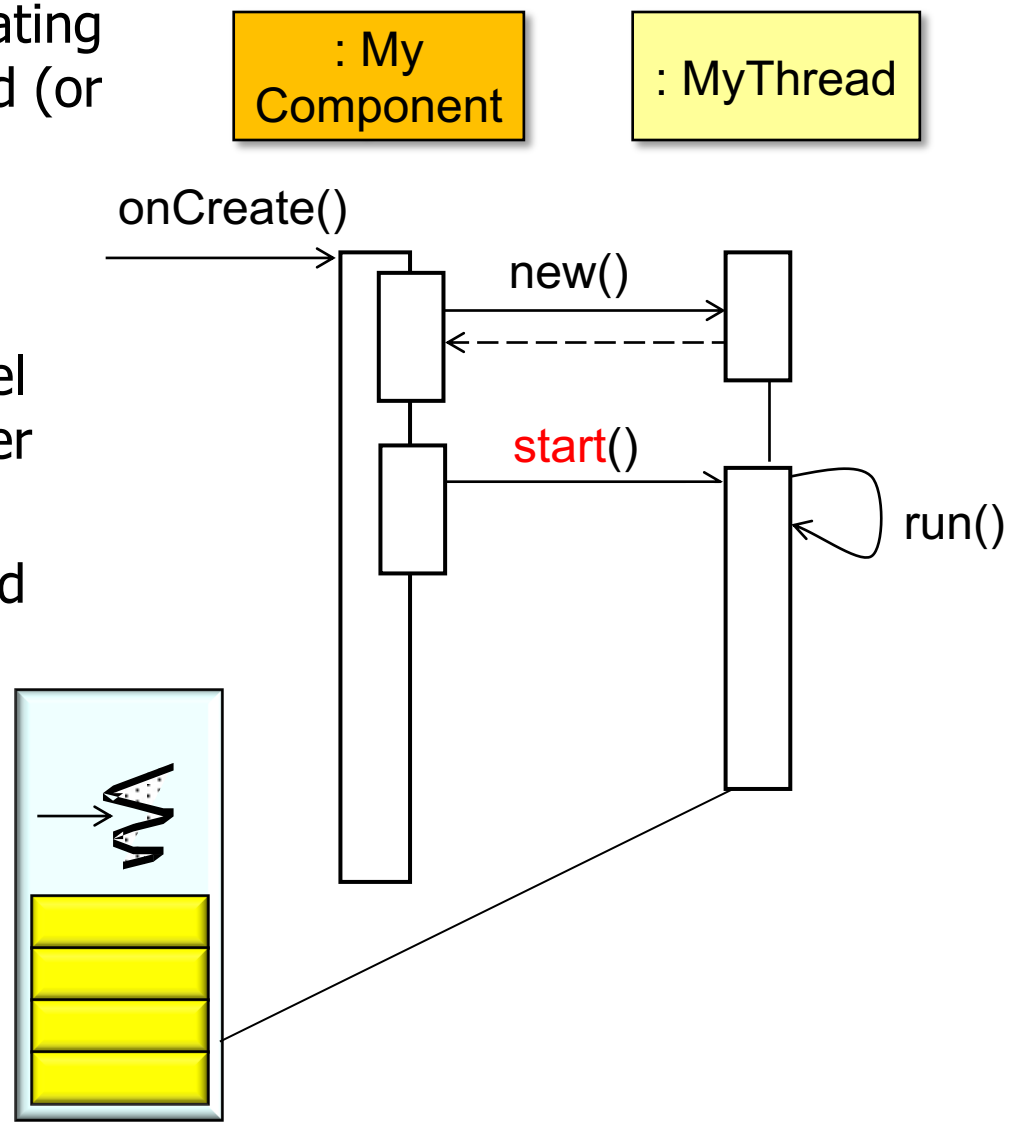
Starting Java Platform Threads

- Multiple layers are involved in creating & starting a traditional Java thread (or new platform thread)
  - Creating a new Thread object allocates little system state
    - e.g., no kernel resources are allocated



Starting Java Platform Threads

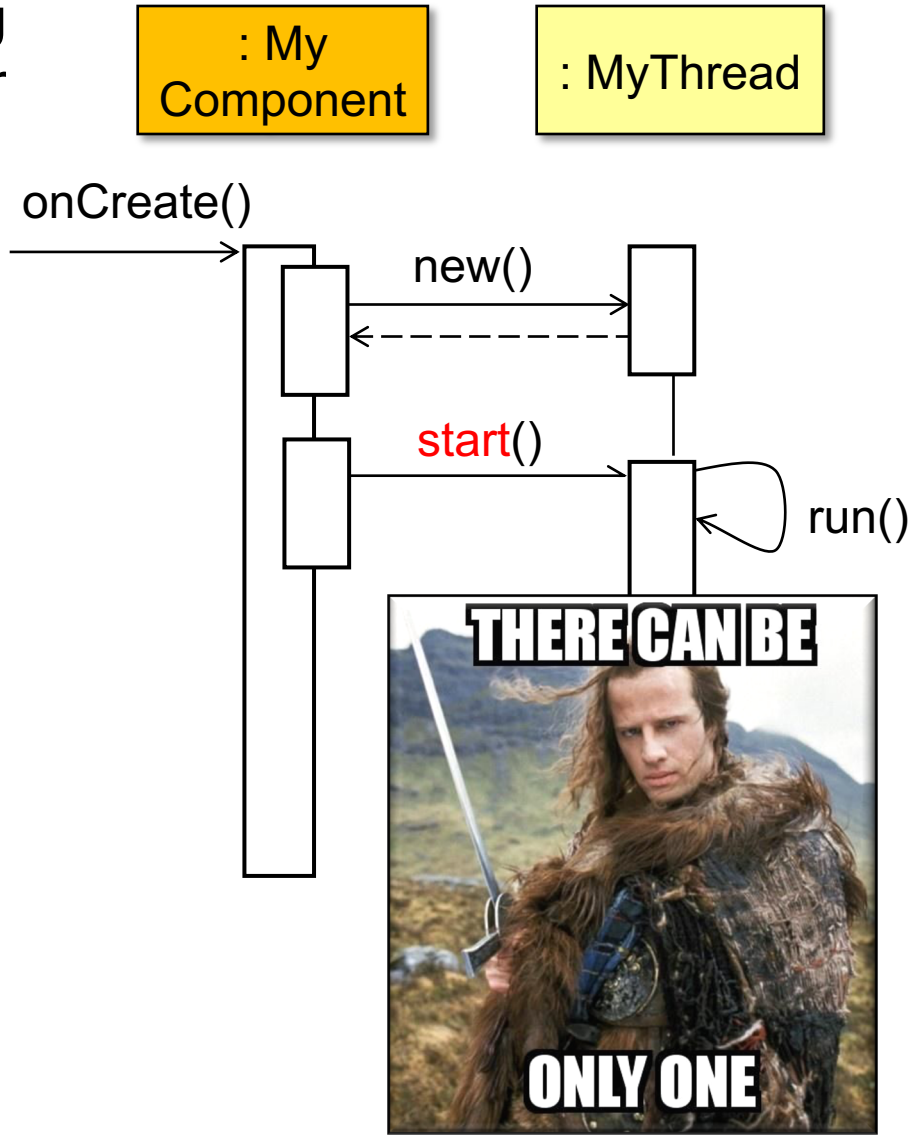
- Multiple layers are involved in creating & starting a traditional Java thread (or new platform thread)
  - Creating a new Thread object allocates little system state
  - The runtime stack & other kernel resources are only allocated after the start() method is called
    - Either Thread.start() or Thread .Builder.OfPlatform.start()



See en.wikipedia.org/wiki/Call_stack

Starting Java Platform Threads

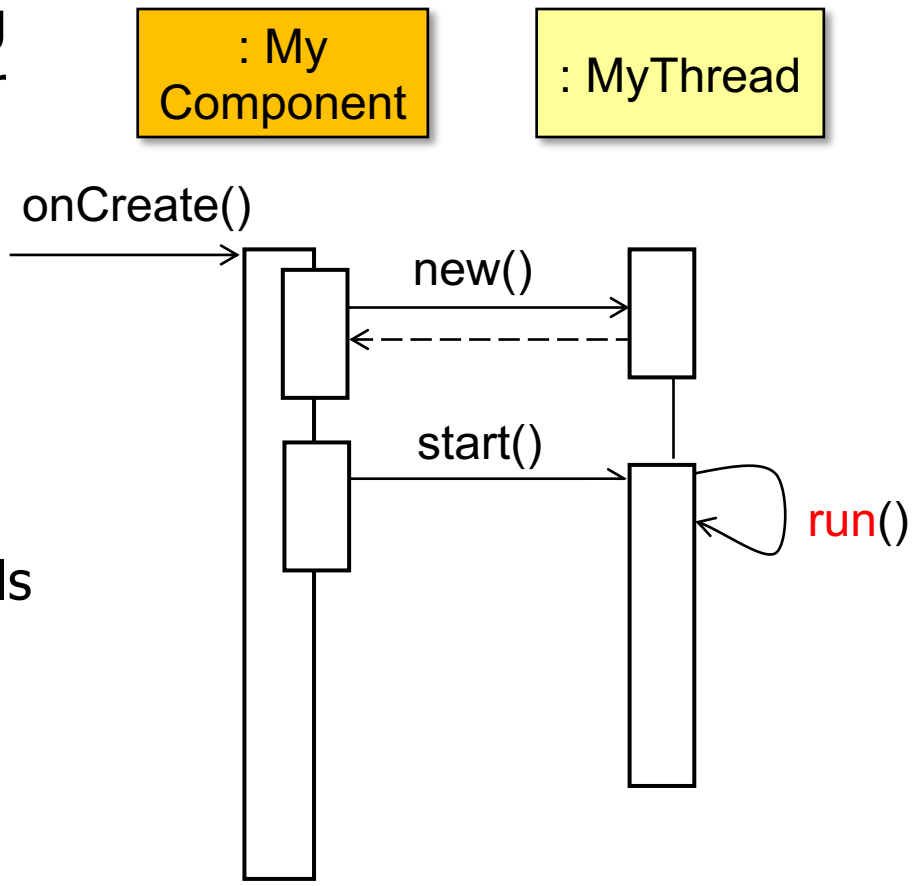
- Multiple layers are involved in creating & starting a traditional Java thread (or new platform thread)
  - Creating a new Thread object allocates little system state
  - The runtime stack & other kernel resources are only allocated after the start() method is called
    - Either Thread.start() or Thread .Builder.OfPlatform.start()



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

Starting Java Platform Threads

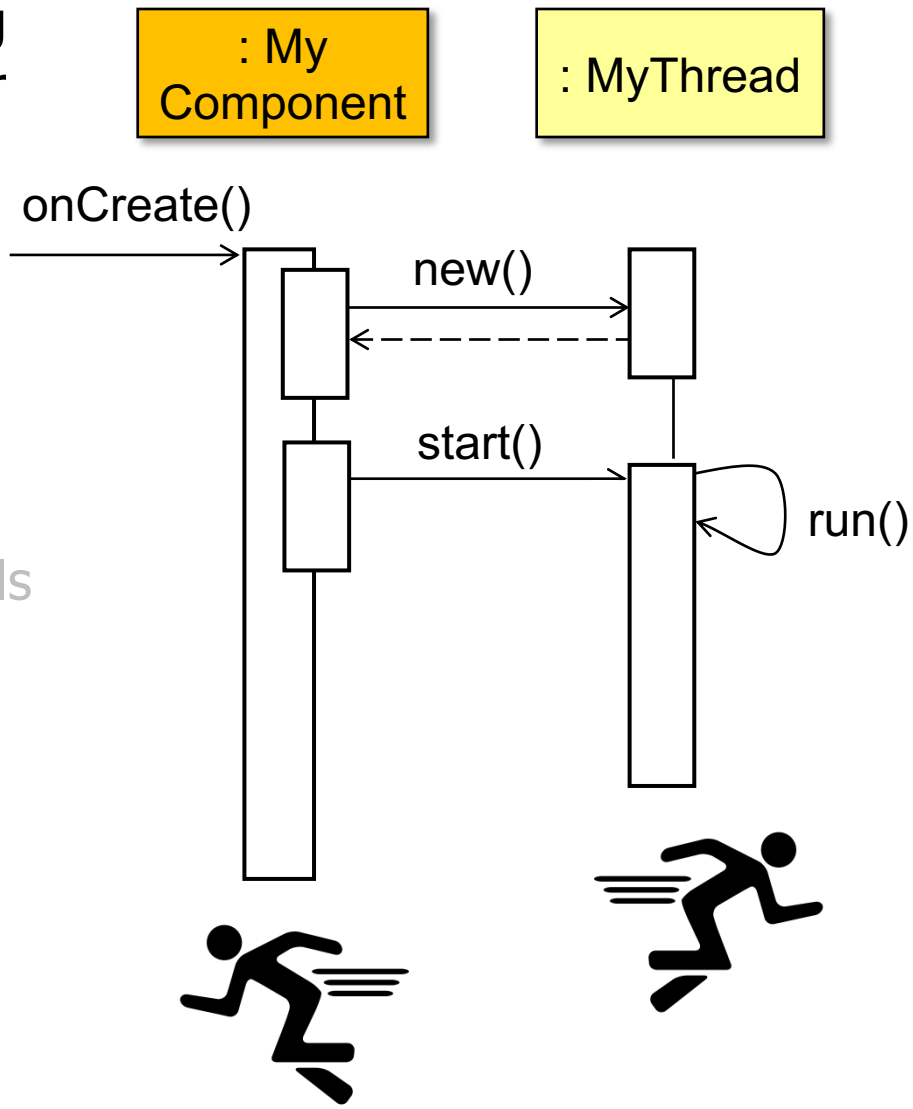
- Multiple layers are involved in creating & starting a traditional Java thread (or new platform thread)
  - Creating a new Thread object allocates little system state
  - The runtime stack & other kernel resources are only allocated after the start() method is called
  - The Java execution environment calls a thread's run() hook method after start() creates its resources



See wiki.c2.com/?HookMethod

Starting Java Platform Threads

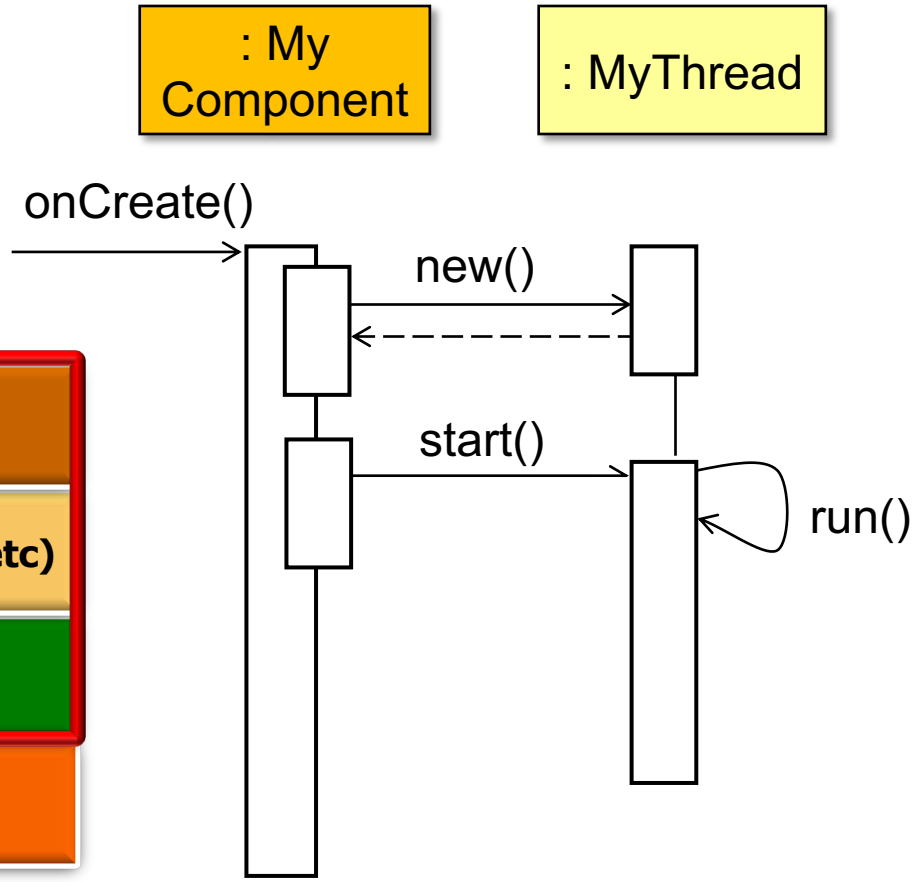
- Multiple layers are involved in creating & starting a traditional Java thread (or new platform thread)
  - Creating a new Thread object allocates little system state
  - The runtime stack & other kernel resources are only allocated after the start() method is called
  - The Java execution environment calls a thread's run() hook method after start() creates its resources
  - Each thread can run concurrently & block independently

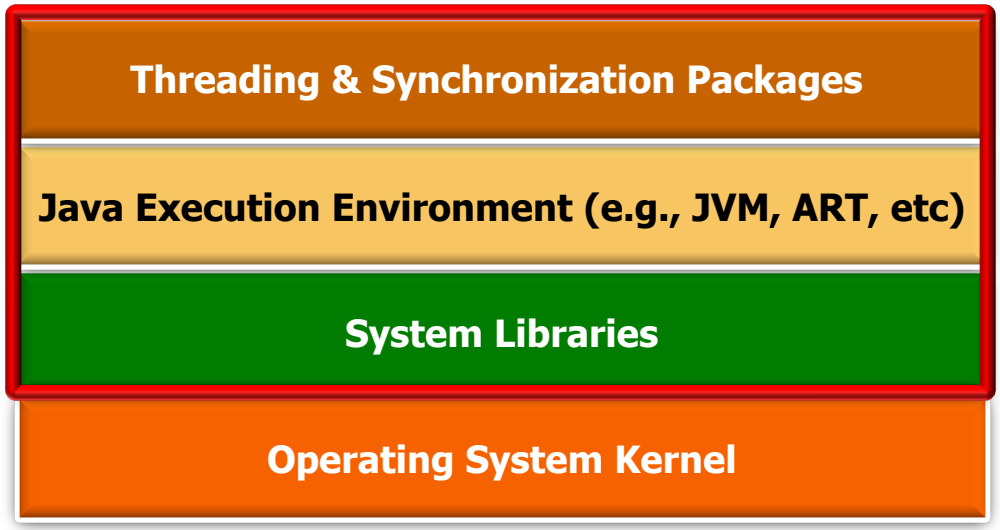


Starting Java Virtual Threads

Starting Java Virtual Threads

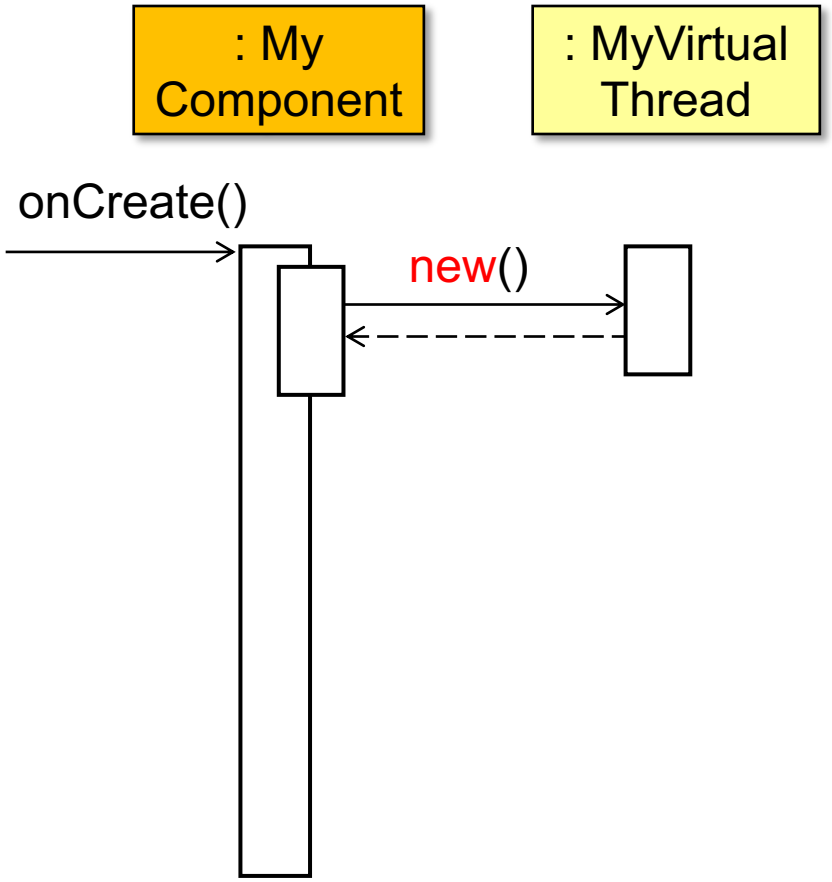
- Fewer layers are involved in creating & starting a Java virtual thread (in contrast to a Java platform thread)





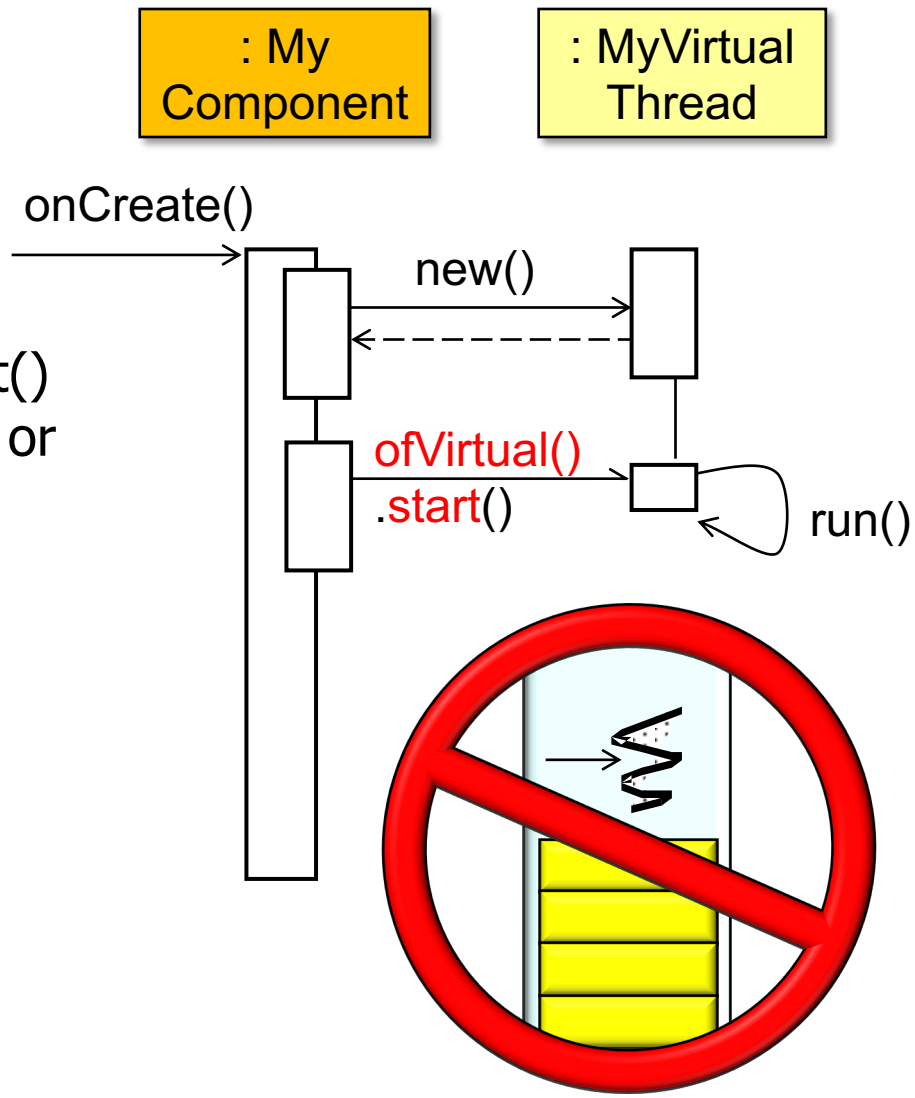
Starting Java Virtual Threads

- Fewer layers are involved in creating & starting a Java virtual thread (in contrast to a Java platform thread)
  - Again, creating a new Thread object allocates little system state
    - e.g., no kernel resources are allocated



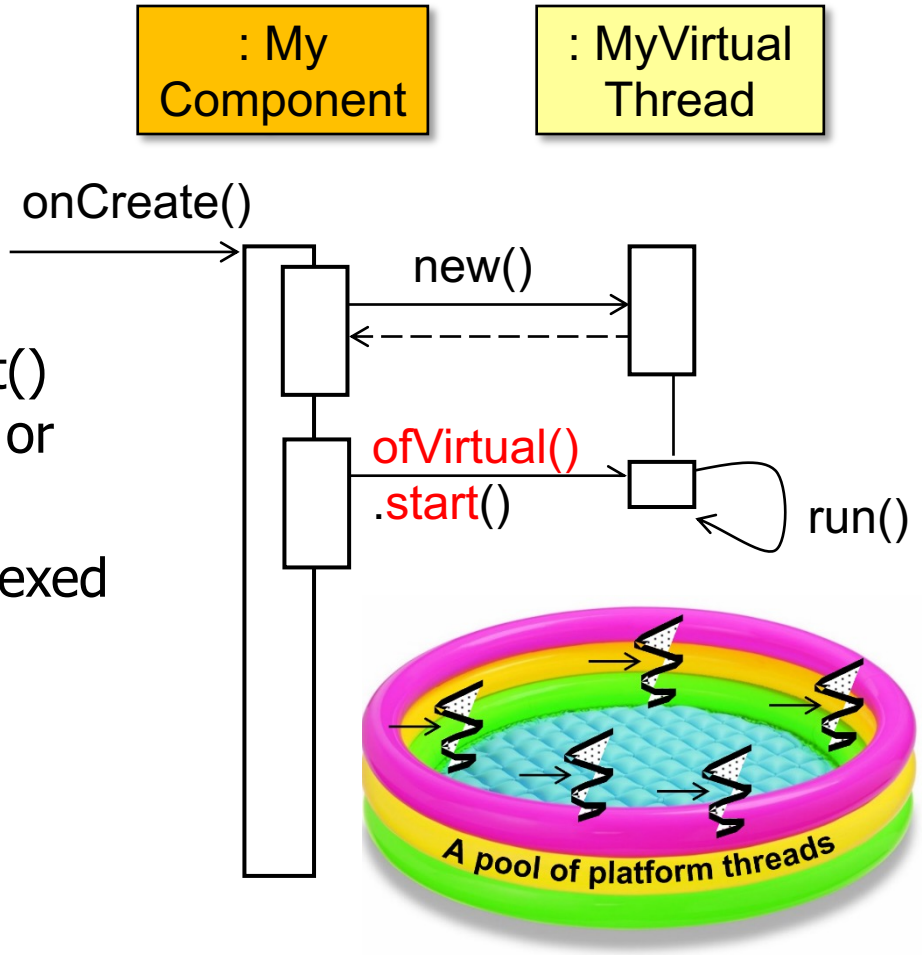
Starting Java Virtual Threads

- Fewer layers are involved in creating & starting a Java virtual thread (in contrast to a Java platform thread)
  - Again, creating a new Thread object allocates little system state
  - Calling Thread.Builder.OfVirtual.start() does *not* allocate any runtime stack or other kernel resources



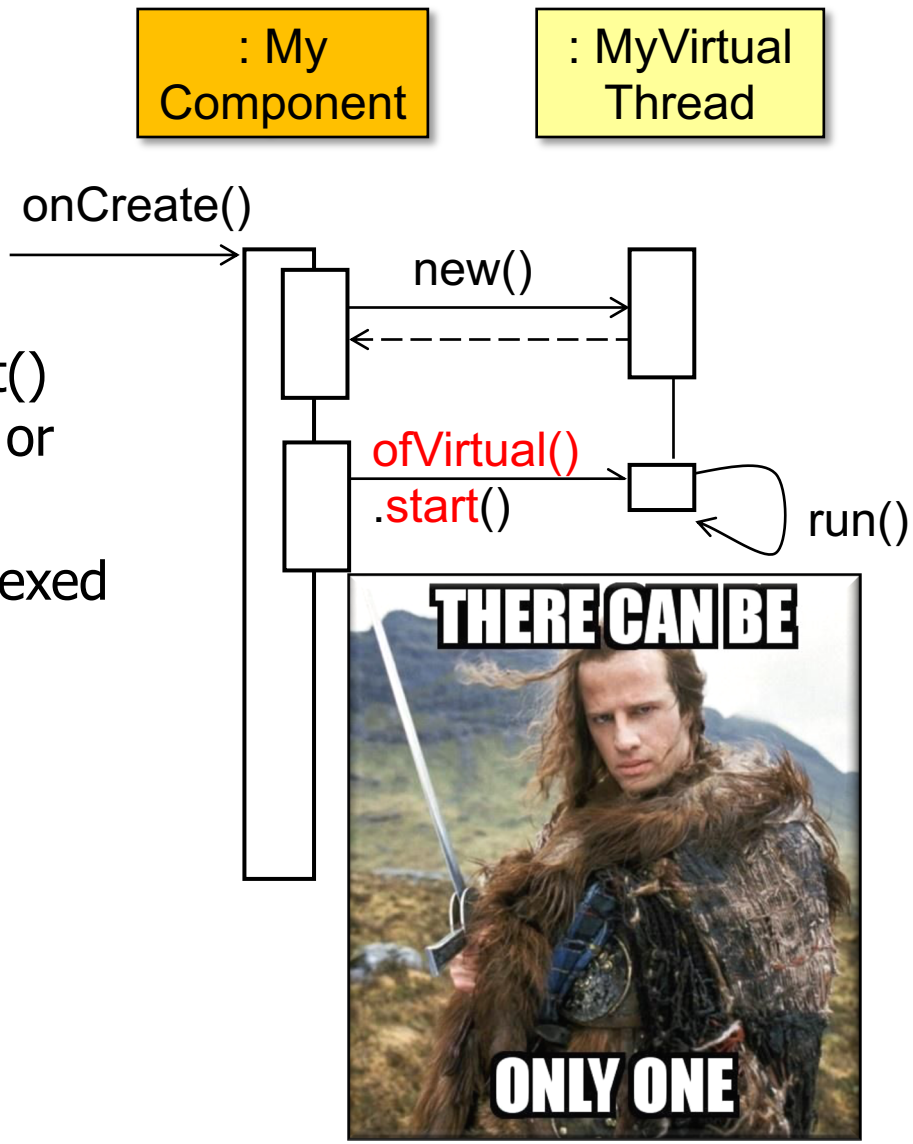
Starting Java Virtual Threads

- Fewer layers are involved in creating & starting a Java virtual thread (in contrast to a Java platform thread)
  - Again, creating a new Thread object allocates little system state
  - Calling Thread.Builder.OfVirtual.start() does *not* allocate any runtime stack or other kernel resources
    - Instead, a virtual thread is multiplexed over a pool of platform threads



Starting Java Virtual Threads

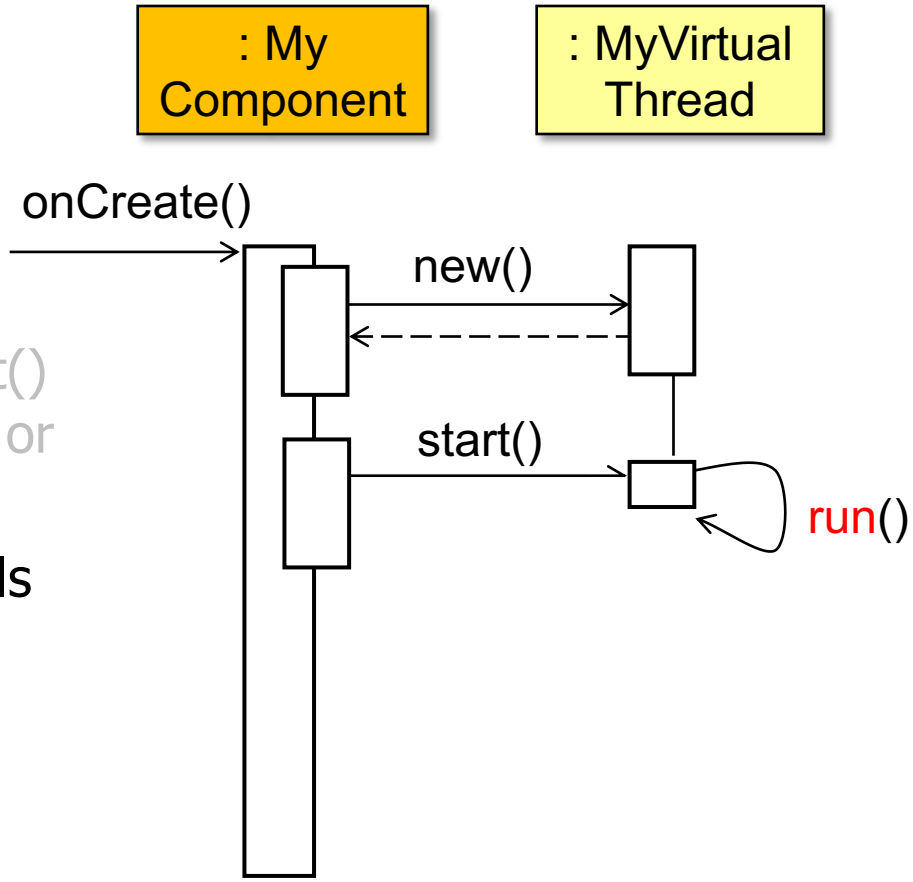
- Fewer layers are involved in creating & starting a Java virtual thread (in contrast to a Java platform thread)
  - Again, creating a new Thread object allocates little system state
  - Calling Thread.Builder.OfVirtual.start() does *not* allocate any runtime stack or other kernel resources
    - Instead, a virtual thread is multiplexed over a pool of platform threads



The start() method can only be called once per virtual thread object

Starting Java Virtual Threads

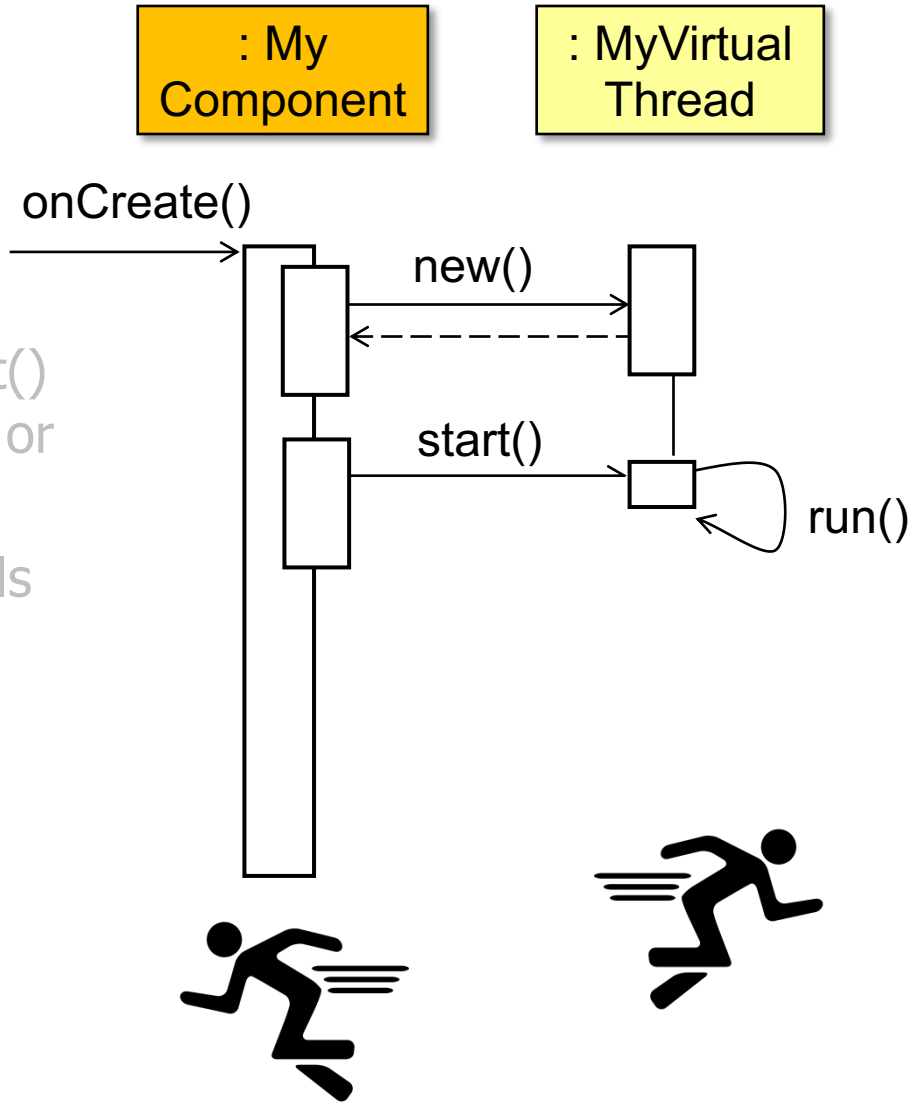
- Fewer layers are involved in creating & starting a Java virtual thread (in contrast to a Java platform thread)
  - Again, creating a new Thread object allocates little system state
  - Calling Thread.Builder.OfVirtual.start() does *not* allocate any runtime stack or other kernel resources
  - The Java execution environment calls a thread's run() hook method after start() creates its resources



See wiki.c2.com/?HookMethod

Starting Java Virtual Threads

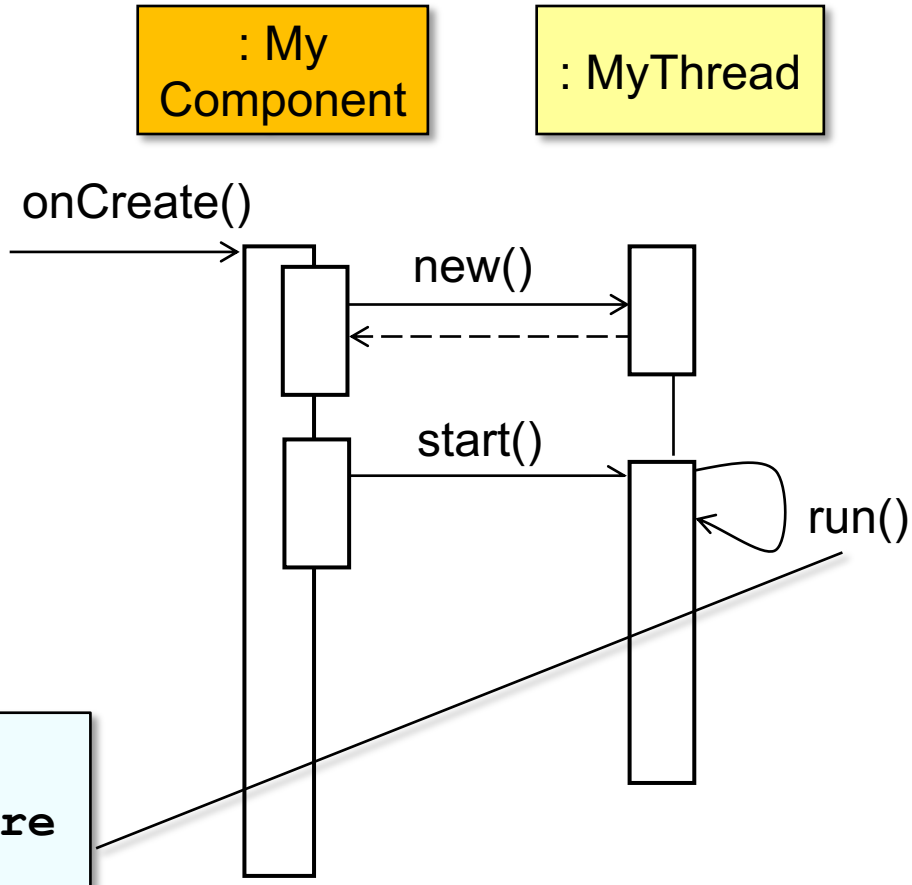
- Fewer layers are involved in creating & starting a Java virtual thread (in contrast to a Java platform thread)
  - Again, creating a new Thread object allocates little system state
  - Calling Thread.Builder.OfVirtual.start() does *not* allocate any runtime stack or other kernel resources
  - The Java execution environment calls a thread's run() hook method after start() creates its resources
  - Each thread can run concurrently & block independently



Running Java Threads

Running Java Threads

- A thread (traditional, platform, or virtual) can generally run any code

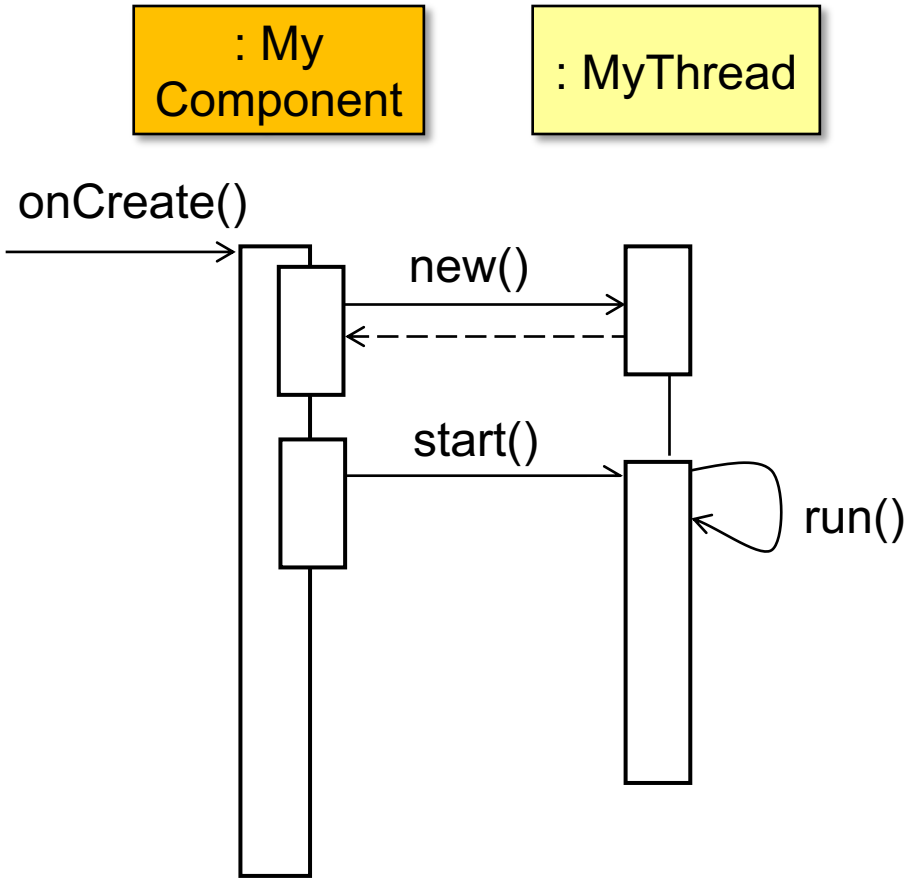



```
public void run(){
   // code to run goes here
}
```

See wiki.c2.com/?HookMethod

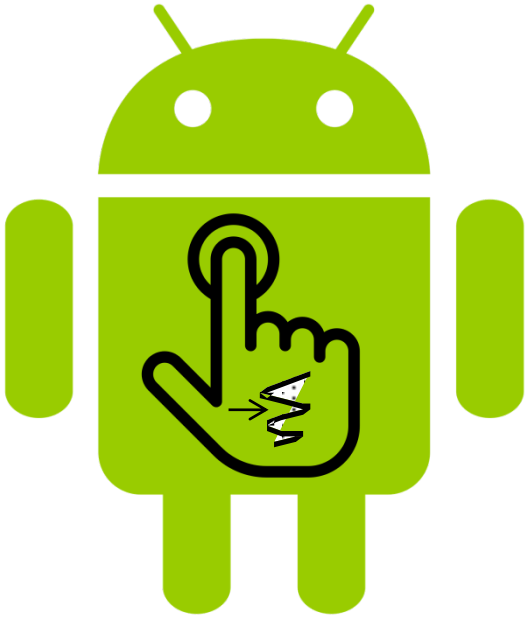
Running Java Threads

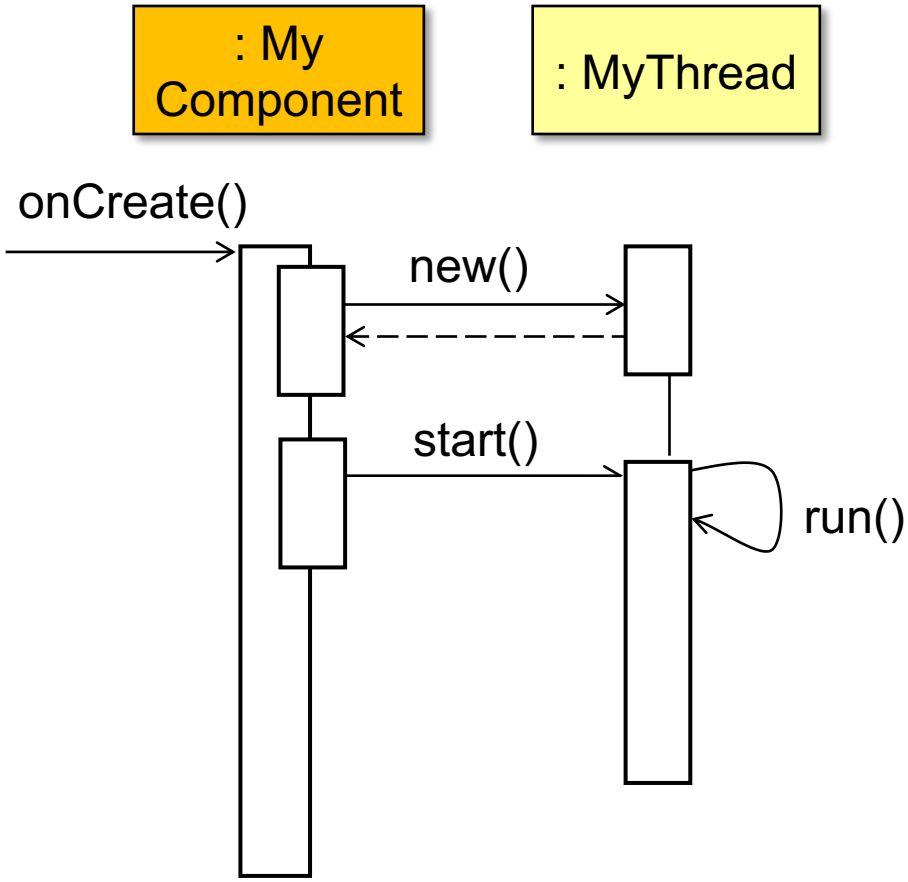
- A thread (traditional, platform, or virtual) can generally run any code
  - However, windowing toolkits often restrict which thread can access GUI components



Running Java Threads

- A thread (traditional, platform, or virtual) can generally run any code
  - However, windowing toolkits often restrict which thread can access GUI components
    - e.g., only the Android UI thread can access GUI components

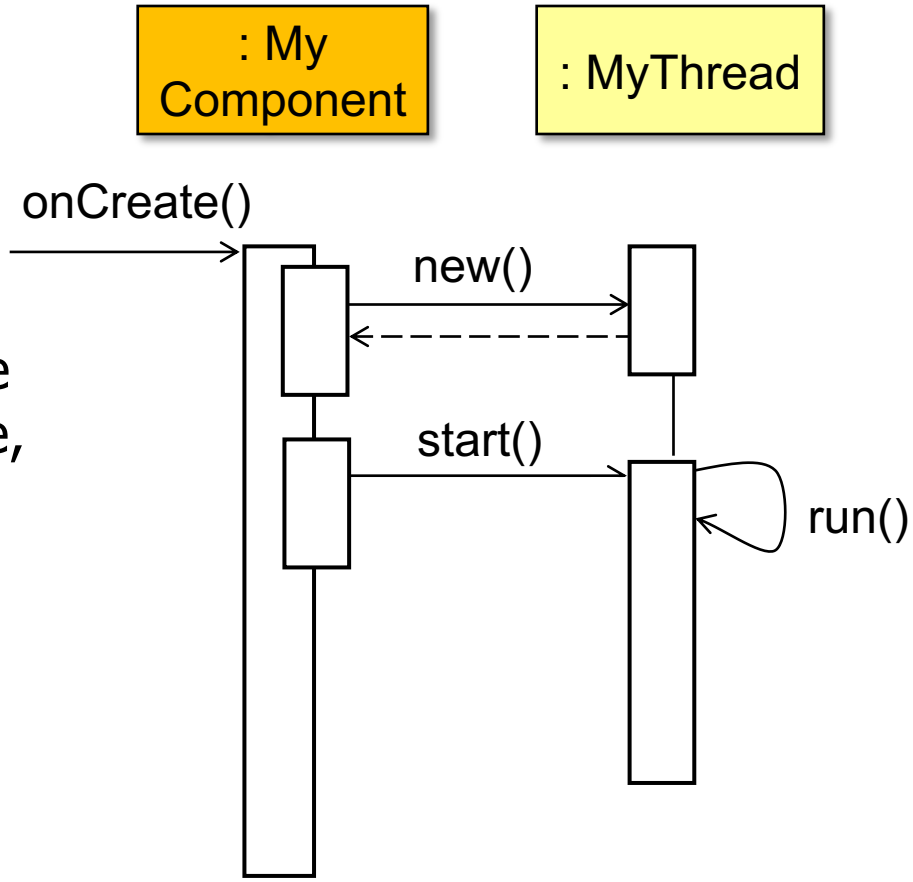




See developer.android.com/training/multiple-threads/communicate-ui.html

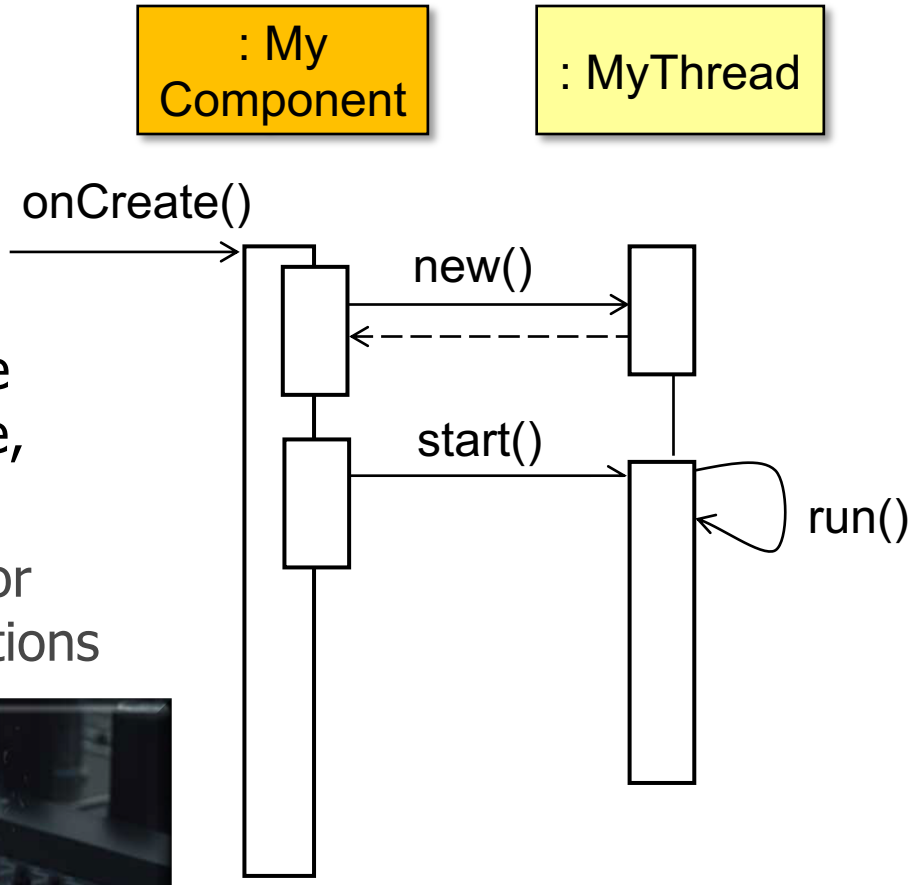
Running Java Threads

- A thread (traditional, platform, or virtual) can generally run any code
  - However, windowing toolkits often restrict which thread can access GUI components
  - Likewise, virtual threads are suitable for tasks that block most of the time, often waiting for I/O to complete



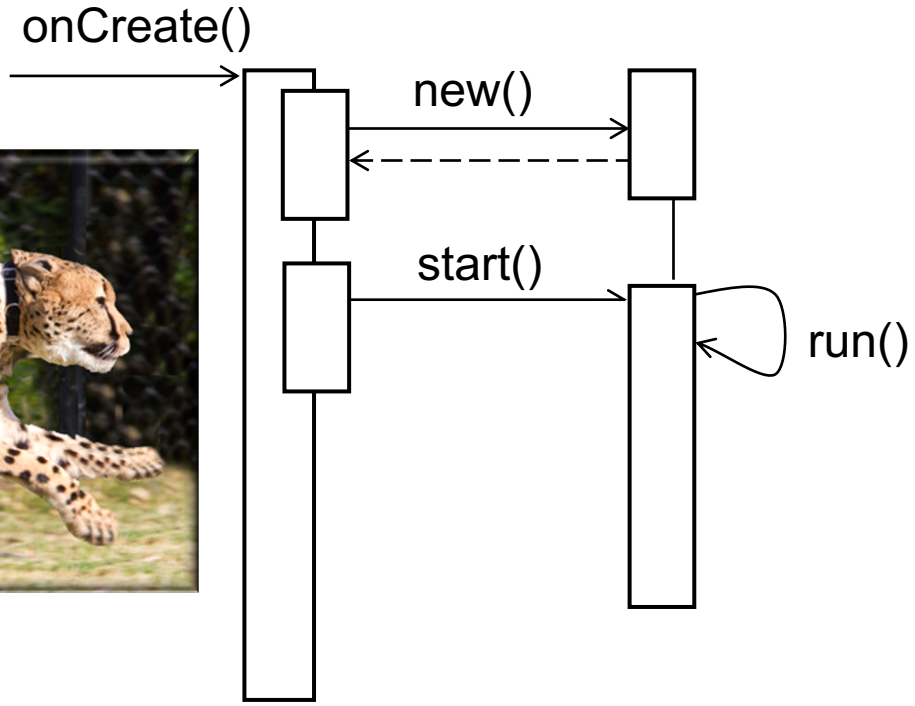
Running Java Threads

- A thread (traditional, platform, or virtual) can generally run any code
  - However, windowing toolkits often restrict which thread can access GUI components
  - Likewise, virtual threads are suitable for tasks that block most of the time, often waiting for I/O to complete
    - Virtual threads are not intended for long running CPU-intensive operations



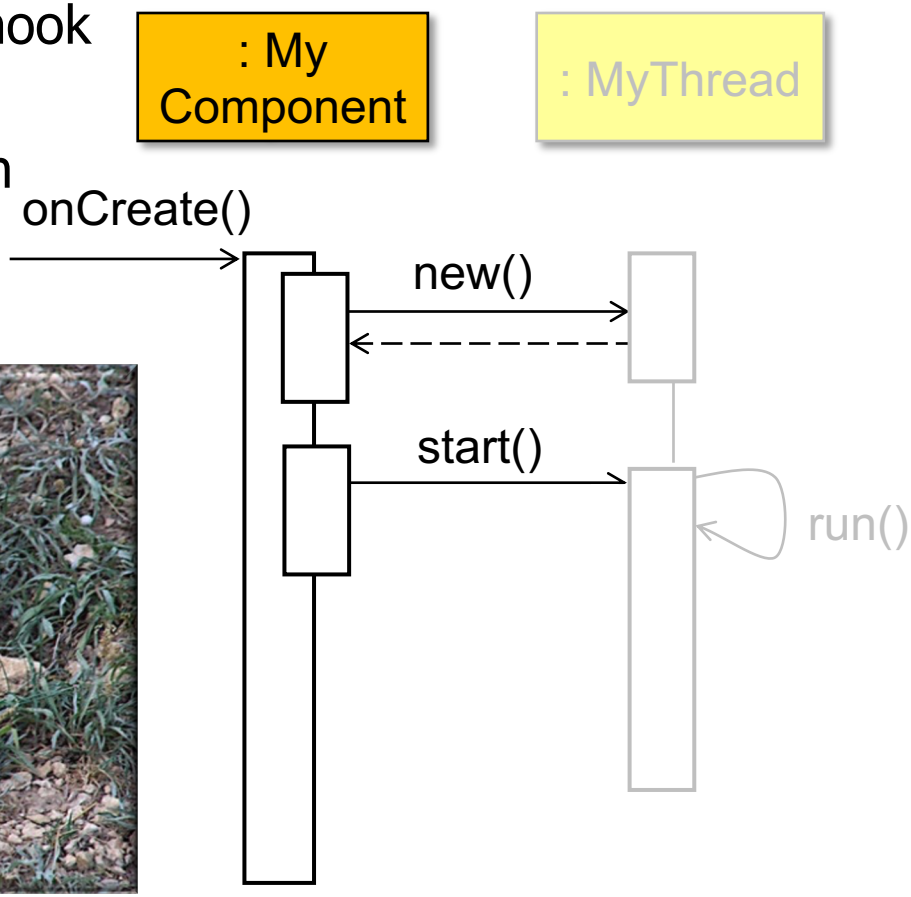
Running Java Threads

- A thread can live as long as its run() hook method hasn't returned



Running Java Threads

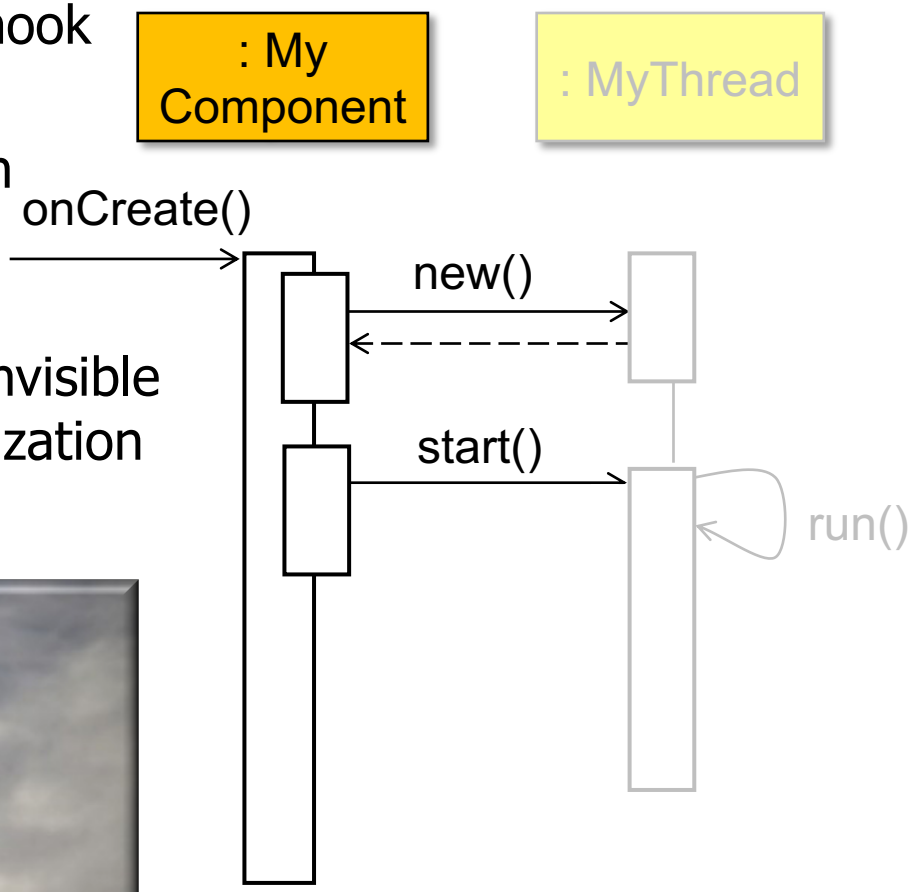
- A thread can live as long as its run() hook method hasn't returned
  - The underlying thread scheduler can suspend & resume a thread many times during its lifecycle



See en.wikipedia.org/wiki/Scheduling_(computing)

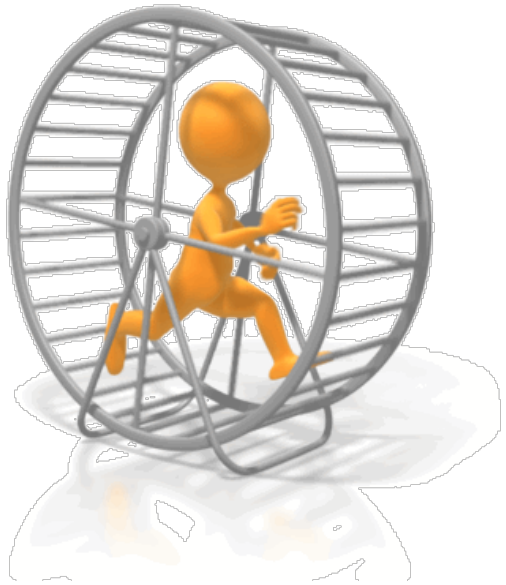
Running Java Threads

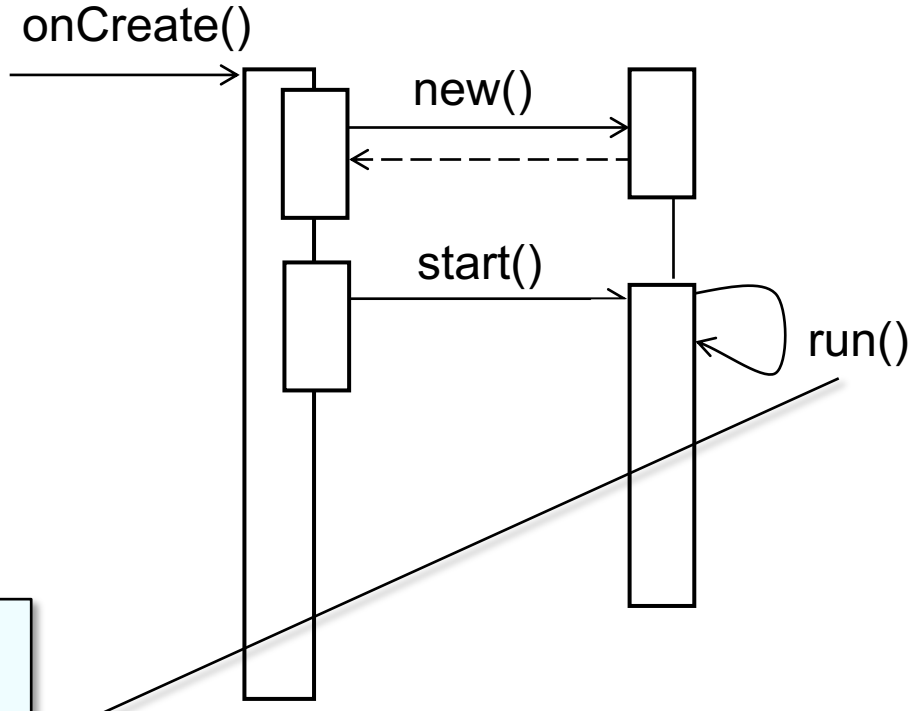
- A thread can live as long as its run() hook method hasn't returned
  - The underlying thread scheduler can suspend & resume a thread many times during its lifecycle
    - Scheduler operations are largely invisible to user code, as long as synchronization is performed properly..



Running Java Threads

- For a thread to execute "forever," its run() hook method needs an infinite loop

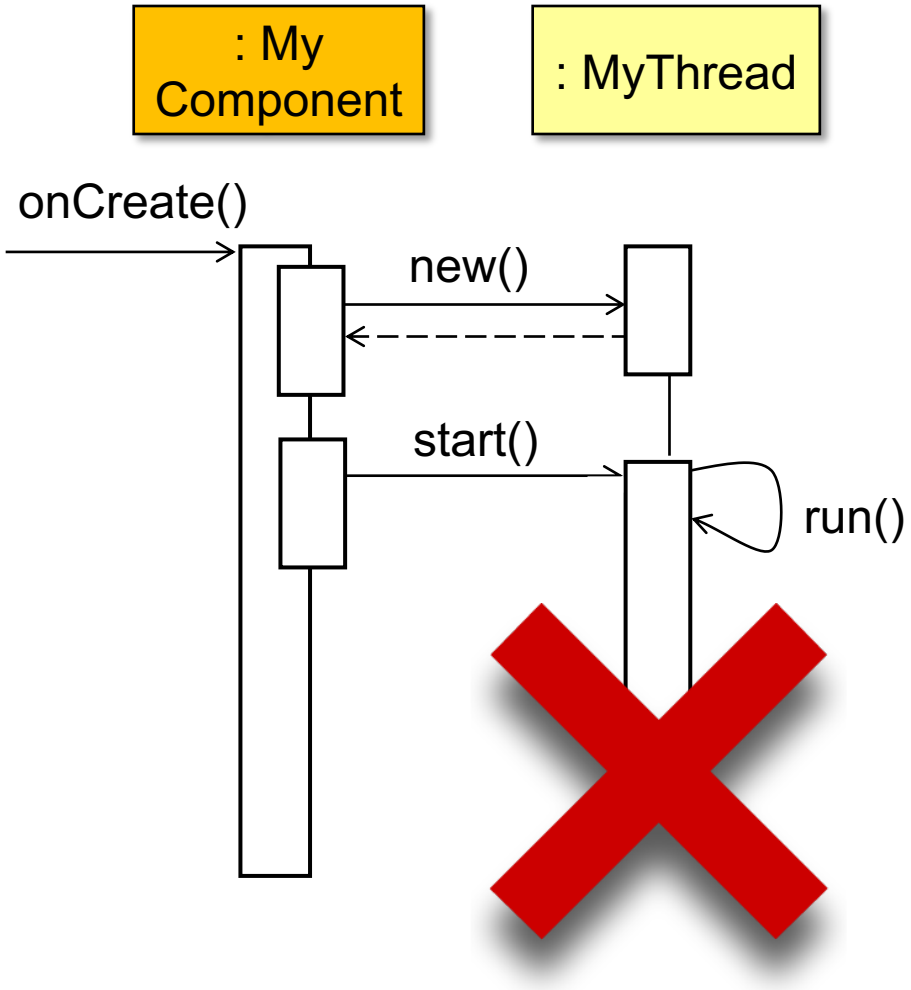





```
public void run(){
   while (true) { ... }
}
```

Running Java Threads

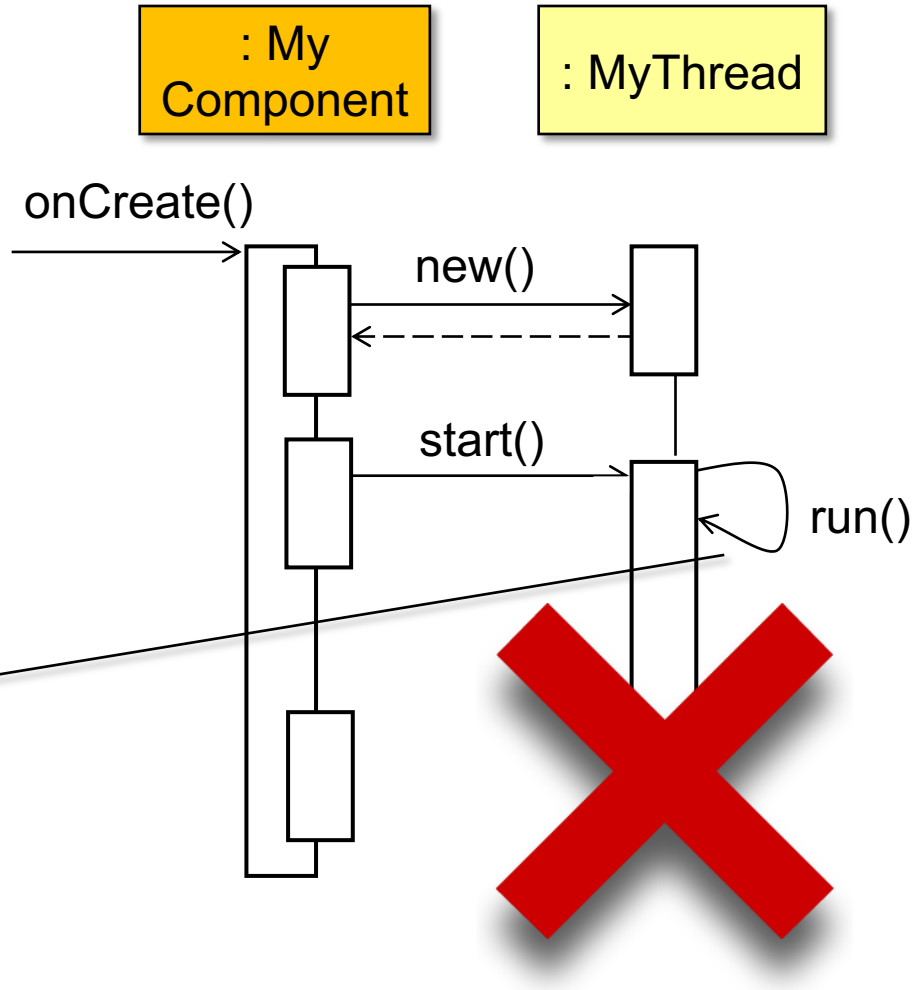
- The thread is dead after run() returns



Running Java Threads

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

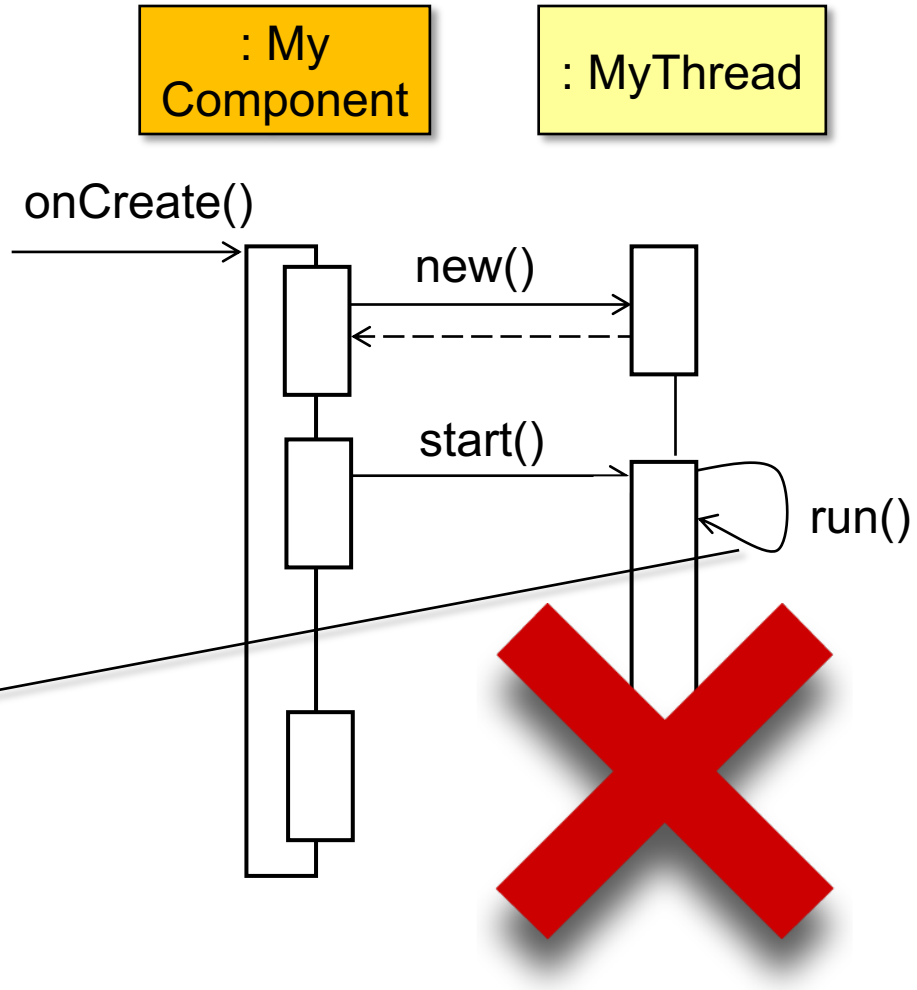
```
public void run(){
  while (true) {
    ...
    if (someCondition())
      return;
  }
}
```



Running Java Threads

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

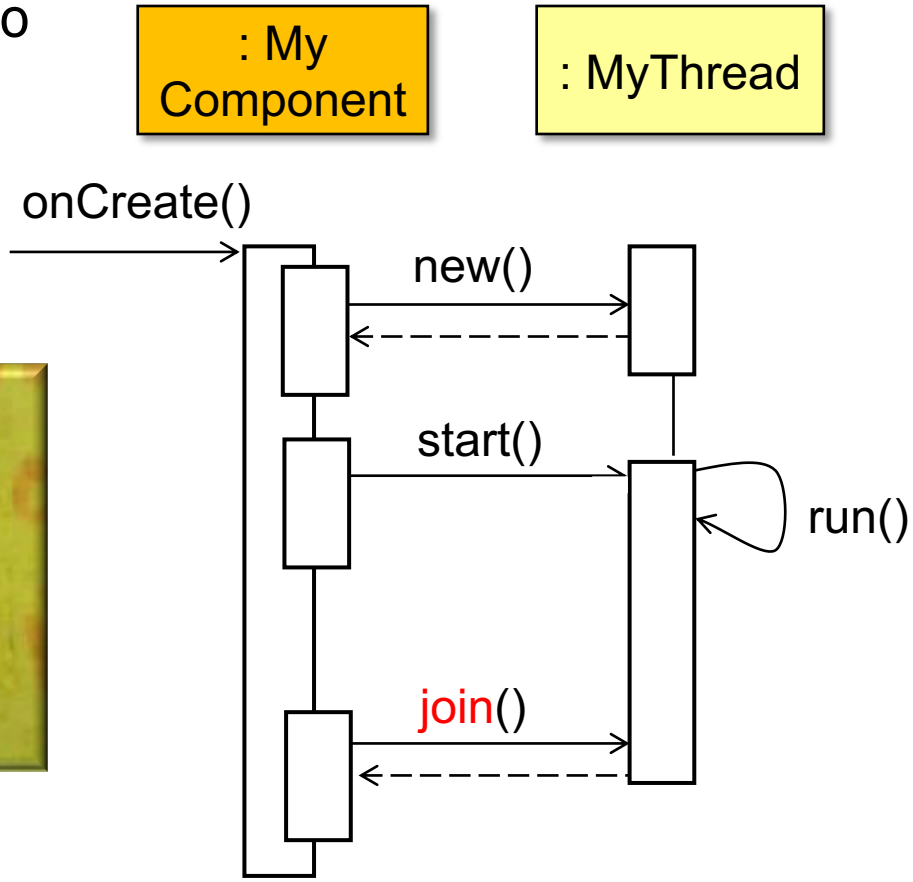
```
public void run(){
  while (true) {
    ...
    if (someError())
      throw new
        SomeException();
  }
}
```



See www.javamex.com/tutorials/exceptions/exceptions_uncaught_handler.shtml

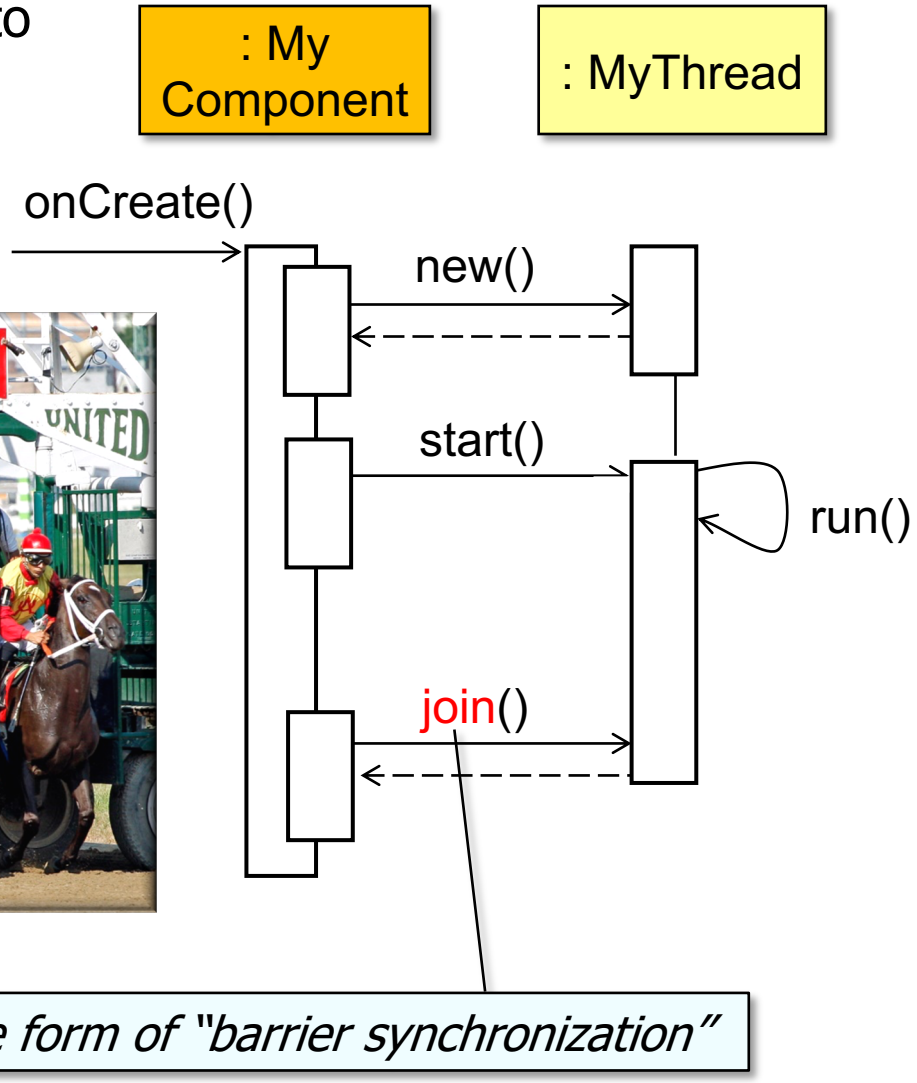
Running Java Threads

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



Running Java Threads

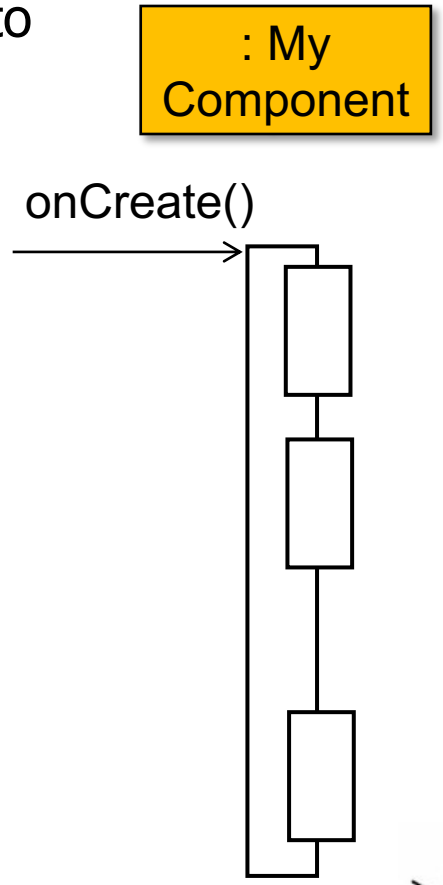
- The join() method allows one thread to wait for another thread to complete



See upcoming lessons on "*Java Barrier Synchronizers*"

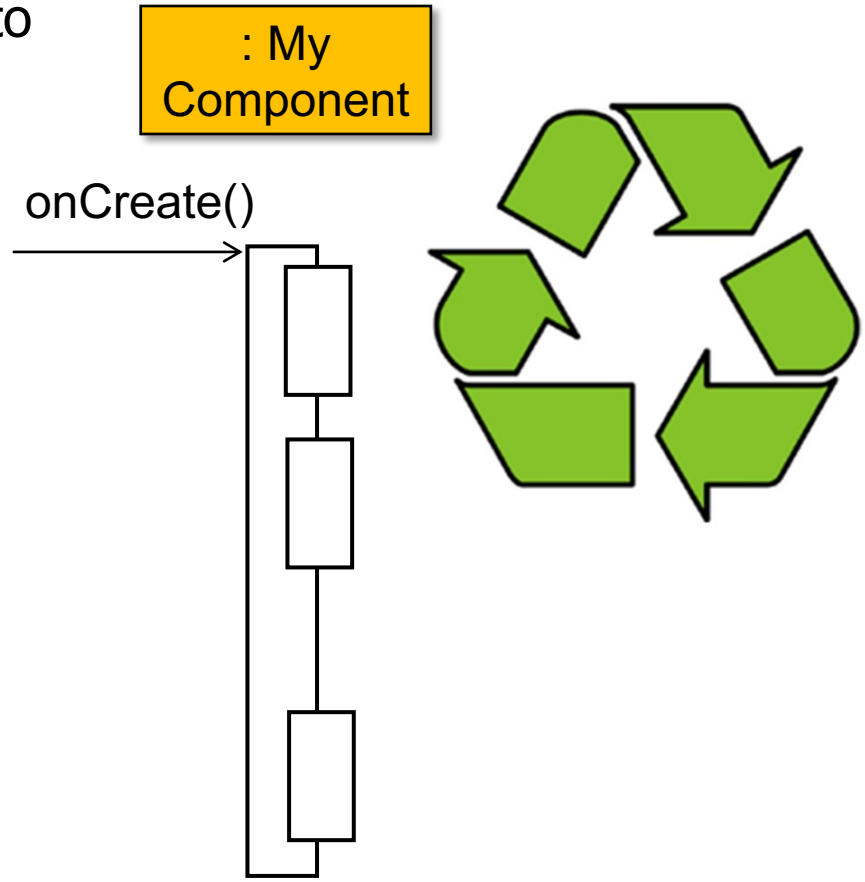
Running Java Threads

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



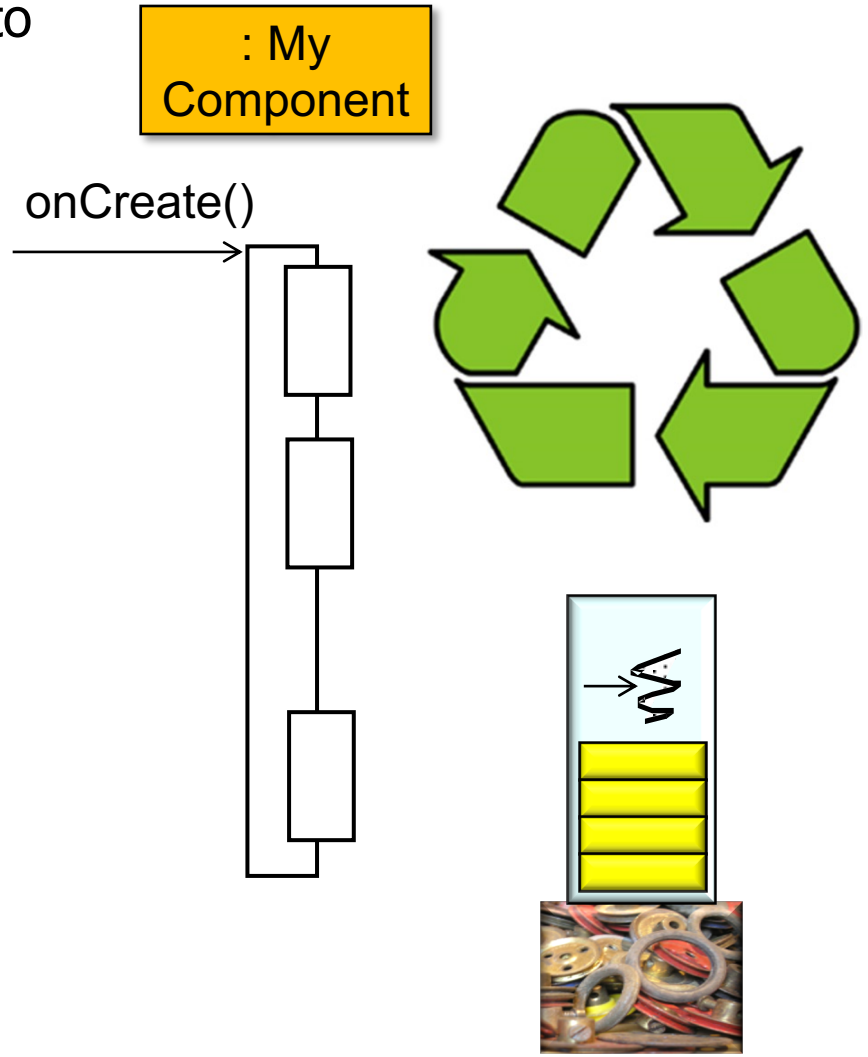
Running Java Threads

- 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-local storage, etc.



End of How Java Threads Start & Run