

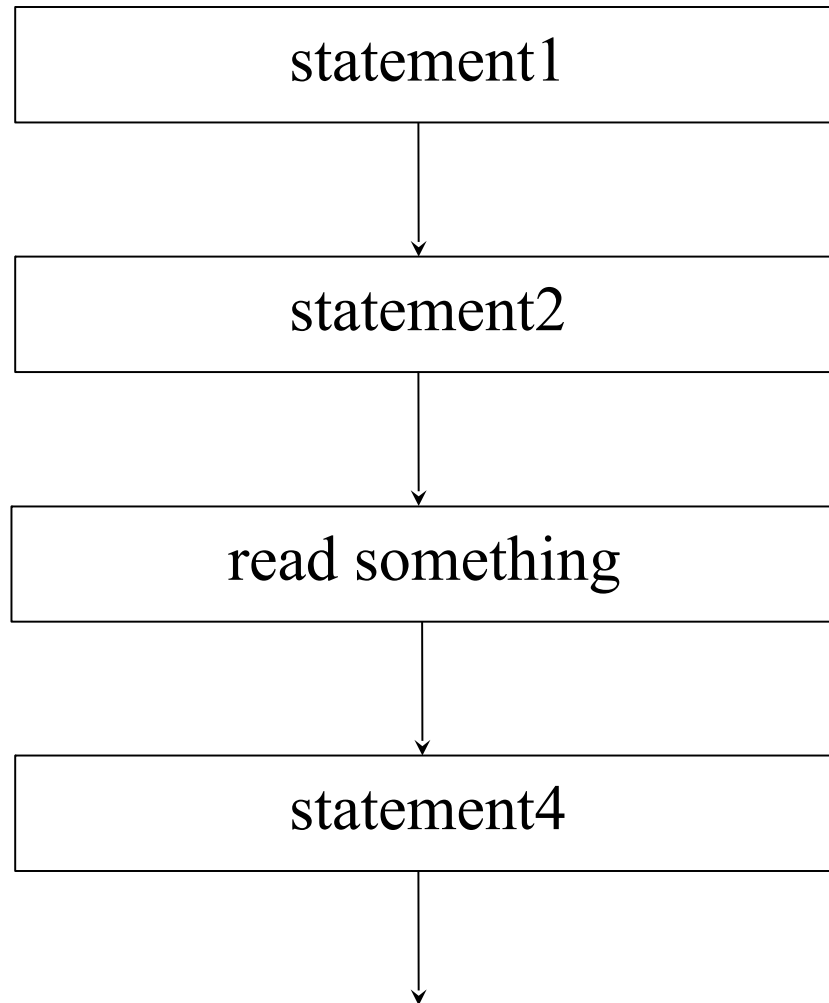


# Threads in Java

---

James Brucker

# A Thread is a single flow of control

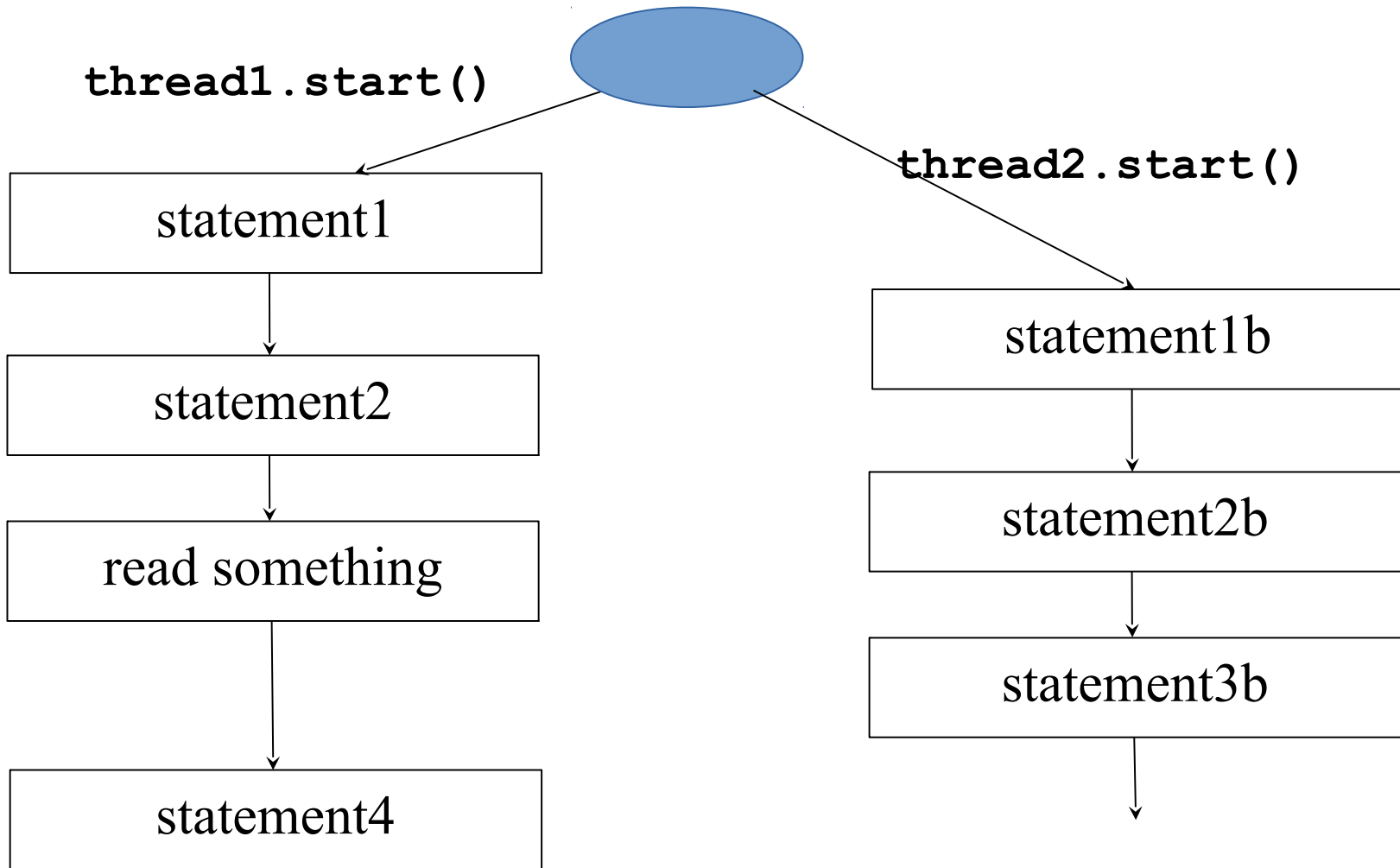


Each statement must finish before next one is executed.

This is the way our console-based programs run.

# A Program can create multiple threads

Threads execute independently. They are not synchronized.



# Why Use Threads?

- Perform tasks in parallel
  - connection to database
  - I/O operations
  - long-running computations (can be in parallel)
- Prevent user interface from "freezing"

UI remains responsive while work is being done.  
Android requires responsive UI... or it will kill you!

`AsyncTask` class for background tasks.

# Basic way to create a Thread

Use the Thread class. There are 2 ways:

1. extend Thread class and override run()

```
public class PrompterThread extends Thread {
    private String message;
    public Prompter(String message) {
        this.message = message;
    }

    /** the work we want to perform */
    public void run() {
        System.out.println( message );
    }
}
```

# Basic way to create a Thread

2. Define class that implements Runnable.  
Put an instance of this Runnable in a Thread.

```
public class Prompter implements Runnable {
    private String message;
    public Prompter(String message) {
        this.message = message;
    }
    /** the work we want to perform */
    public void run() {
        System.out.println( message );
    }
}
Thread promptThread = new Thread( new Prompter() );
promptThread.start();
```

# Running a Thread

Call the `start( )` method of Thread.

It runs in a separate *thread* and returns control immediately

```
// Method 1. use a subclass of Thread
Thread thread1 = new PrompterThread("Hello");
thread1.start( );

// Method 2. use a Runnable and wrap in a Thread
Runnable prompter = new Prompter("Hello");
Thread thread2 = new Thread( prompter );
thread2.start( );
```

# Sequence Diagram

---



# Demo

---

## 1. Hello and Goodbye threads.

Say "Hello", "and", "Goodbye" in separate threads, wait 500ms, repeat.

... the messages are not always synchronized.

Note:

if "main" method quits without waiting for threads or killing them, they will keep running, even if console window is closed.

# Waiting for threads to finish

- After `Thread.start()` control returns immediately.
- To wait for another thread to finish, use `join`.

```
public static void main(String[] args) {
    Thread thread1 =
        new Thread( new Greeter("Hello") );

    thread1.start();
    try {
        thread1.join();
    } catch (InterruptedException ie) {
        // something interrupted the thread
        // while we were waiting
    }
}
```

# InterruptedException

A Thread can be interrupted by calling `thread.interrupt()`.

If the thread is busy, waiting, or sleeping then an `InterruptedException` is thrown in the thread.

```
try {
    Thread.sleep( 2000 ); // time in millisec
} catch (InterruptedException ie) {
    // something invoked interrupt( ) while
    // we were sleeping!
}
```

# Checking for Interruptions

If your thread *might* be interrupted, you can check for it using `Thread.interrupted( )`

```
class Downloader implements Runnable {
    public void run( ) {
        // read a big file into buffer
        while((size=in.read(buffer)) > 0) {
            // check for interruption
            if ( Thread.interrupted() ) {
                // close streams and return
                in.close();
                return;
            }
            // otherwise, process the buffer
        }
    }
}
```

# Thread class

## Thread

currentThread( ): Thread

interrupt( ): void

interrupted( ): boolean

getName( ): String

sleep(millisec: long): void

*many more methods*

# Interrupt Demo

---

In the "Hello and Goodbye" demo modify main method:  
when user presses ENTER, interrupt the greeter threads.

Use:

```
thread.interrupt( ) - interrupt thread
```

# Impatient Demo

Write an impatient greeter:

Prompt user his name and print "Hello, username."  
but don't wait more than 5 seconds for a reply.

Use a separate thread to get the user's name.

`thread.join(timeout)` - wait for thread at most  
`timeout` millisecs

`thread.interrupt( )` - interrupt thread

`thread.destroy( )` - doesn't

# Demo

---

1. "main" waits for threads to finish.
2. In Hello / Goodbye, give each thread a name.

```
new Thread( String name )
```

```
new Thread( Runnable task, String name )
```

2. In each thread, get its name and print it.

Use `Thread.currentThread()` and `thread.getName()`.



# Thread Managers and Thread Pools

---

For most uses, you should use one of these:

**Timer** - run a task at specified time or frequency

## **ExecutorService**

- manage one or more threads (called a *thread pool*).
- reuse threads (efficient memory use).
- can also run *Callable* tasks (return a value in future)
- shutdown or cancel tasks

# Demo

---

1. Rewrite ImpatientGreeter to use ExecutorService:

```
ExecutorService executor =
```

```
    Executors.newSingleThreadExecutor( );
```

```
executor.submit( runnable );
```

wait for thread:

```
    executor.awaitTermination(5, TimeUnit.SECONDS)
```

kill threads: `executor.shutdownNow( )`

# Demo

---

2. Rewrite Hello / Goodbye to use Timer.

```
timer.schedule( TimerTask, long delay, long period )
```

```
timer.scheduleAtFixedRate( TimerTask, delay, period )
```

# ExecutorService

Defines several kinds of thread pools.

Example:

You want to run some tasks using threads, but use at *most 4 threads at one time* (to avoid task switching).

```
final int MAX_THREADS = 4;
ExecutorService executor =
    Executors.newFixedThreadPool(MAX_THREADS);

// add tasks to queue for execution
executor.execute(runnable1);
executor.execute(runnable2);
. . .
executor.execute(runnable10);
```

# Protected Blocks of Code

---

When multiple threads access the same object, or same memory, what can go wrong?

What if two threads try to change some memory at the same time?

# Thread-safe classes

---

`StringBuffer` - thread safe (OK for 2 threads to share)

`StringBuilder` - not thread safe

`AtomicInteger`, `AtomicLong`, etc. - thread safe Integer, Long, etc.

# "synchronized" methods

If a method is "synchronized" then one one thread can execute it at a time. Other threads must wait.

```
public synchronized List withdraw(double amount) {  
    // only one thread can execute this method  
    // at any time. Other threads must wait.
```

Problem: synchronized methods are sloooooow.

# References

---

The Java Tutorial:

<https://docs.oracle.com/javase/tutorial>

Threads & Executors Tutorial (Winterbe blog)

<http://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/>

*This is a really good blog for Java! Have a look.*

*Concurrency (general)*

<https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>