

---

# ***SYSC 3303 Real-Time Concurrent Systems***

## **Introduction to Java Threads**

- Copyright © 2001-2004 D.L. Bailey and and 2004-2007 L.S. Marshall,  
Systems and Computer Engineering, Carleton University
- revised July 11<sup>th</sup>, 2007

---

# *Threads of Execution in Java Programs*

- When the Java Virtual Machine (JVM) starts executing a Java program, there is a single thread of execution that typically calls the static `main()` method of some designated class
  - aside: the JVM also creates one or more *daemon threads* (threads that run in the background in support of user-defined, non-daemon threads)
    - the Java garbage collector is a daemon thread
- In this set of slides, we'll look at how to create additional user-defined threads of execution in Java programs

---

## ***Java Threads - A First Example***

- In this example, the thread executing `main()` creates and starts a second thread that calculates and outputs 0! through 20!
- The application has a simple graphical user interface (GUI) built from Swing components
  - each thread displays its output in its own `JTextArea`
  - the code for building the GUI has been omitted from the slides, but the complete source code for the example is on the Web site

---

## ***Class ThreadExample1***

```
class ThreadExample1 extends JFrame
{
    // JTextArea for the factorial thread.
    private JTextArea ta1;

    // JTextArea for the thread executing main().
    private JTextArea status;

    /**
     * Build the GUI.
     */
    public ThreadExample1(String title) {
        // Code not shown. Inits ta1 and status.
    }
}
```

---

## ***Class ThreadExample1***

```
public static void main(String[] args) {  
    ThreadExample1 frame = new  
        ThreadExample1("Thread Example 1");  
    ...  
  
    // Size the window to fit the preferred  
    // size and layout of its subcomponents,  
    // then show the window.  
    frame.pack();  
    frame.setVisible(true);  
}
```

---

## ***Class ThreadExample1: Creating a Thread***

```
Thread factorialThread =  
    new Factorial("Factorial calculator",  
                  frame.ta1);  
  
frame.status.append("Created: " +  
    factorialThread + '\n');  
  
frame.status.append("Starting thread\n");  
  
factorialThread.start();  
}  
}
```

---

## ***Class Factorial***

```
class Factorial extends Thread
{
    /**
     * The text area where this thread's output
     * will be displayed.
     */
    private JTextArea transcript;

    public Factorial(String name,
                     JTextArea transcript) {
        super(name);
        this.transcript = transcript;
    }
}
```

---

## ***Class Factorial***

```
public void run() {  
    // 0! = 1  
    long factorial = 1;  
    transcript.append("0! = " + factorial +  
                      '\n');  
    for (int n = 1; n <= 20; n++) {  
        // Sleep for between 0 and 2 seconds  
        // before calculating n!  
        try {  
            Thread.sleep((int) (Math.random()  
                             * 2000));  
        } catch (InterruptedException e) {}  
    }  
}
```



---

## ***Class Factorial***

```
// n! = n * (n-1)!
factorial = n * factorial;
transcript.append(n + "! = " +
                  factorial + '\n');
}
transcript.append(Thread.currentThread()
                  + " finished\n");
}
}
```

---

## ***main () : Creating a New Thread***

- The factorial-calculating thread is created by this statement in `ThreadExample1.main()`:

```
Thread factorialThread =  
    new Factorial("Factorial calculator",  
                  frame.ta1);
```

- Syntactically, this statement looks like typical object creation in Java
  - `new` creates a new object from `Factorial`, and assigns the reference to this object to variable `factorialThread`
- Notice that `Factorial` is a subclass of `Thread`

---

## ***main () : Creating a New Thread***

- Java threads of execution are modelled by instances of class `Thread` in package `java.lang`
  - `Thread` objects are abstractions of the threads provided by the platform on which the JVM is running
    - the platform-specific details are hidden by this class
- When an instance of `Thread` (or an instance of a subclass of `Thread`) is created, the JVM arranges for a corresponding thread of execution to be created

---

## ***main () : Creating a New Thread***

- the `Thread` object's code will be executed by this thread - we'll see how this is arranged a few slides from now
- Normally, we don't distinguish between a `Thread` object and the thread of execution that executes the `Thread` object's code, so we can simply say that this statement:

```
Thread factorialThread =  
    new Factorial("Factorial calculator",  
                  frame.ta1) ;  
creates a new Java thread from Factorial
```

---

## ***Class Factorial - Constructor***

- Factorial is defined as a subclass of Thread
- It has a two-argument constructor:

```
public Factorial(String name,  
                  JTextArea transcript)
```
- The first argument is the thread's name, which is passed to superclass constructor `Thread(String)` by the statement:

```
super(name) ;
```
- The second argument is a reference to the `JTextArea` where the thread will display its output

---

## ***main () : Starting the Thread***

`factorialThread.start();`

- Makes the thread associated with the `Thread` object referred to by `factorialThread` *ready to run*, so that it can be scheduled for execution by the Java Virtual Machine (JVM)

---

## ***Class `Factorial`: `run()` method***

- When a new thread is started it begins execution in its `Thread` object's `run()` method
- Subclasses of `Thread` inherit the `run()` method defined in `Thread`, but this method simply returns, so user-defined subclasses of `Thread` must override `run()` to define the required thread behaviour
- Example: the `run()` method in `Factorial` outputs the values of `0!` through `20!` in its `JTextArea`

---

## ***Concurrent Thread Execution***

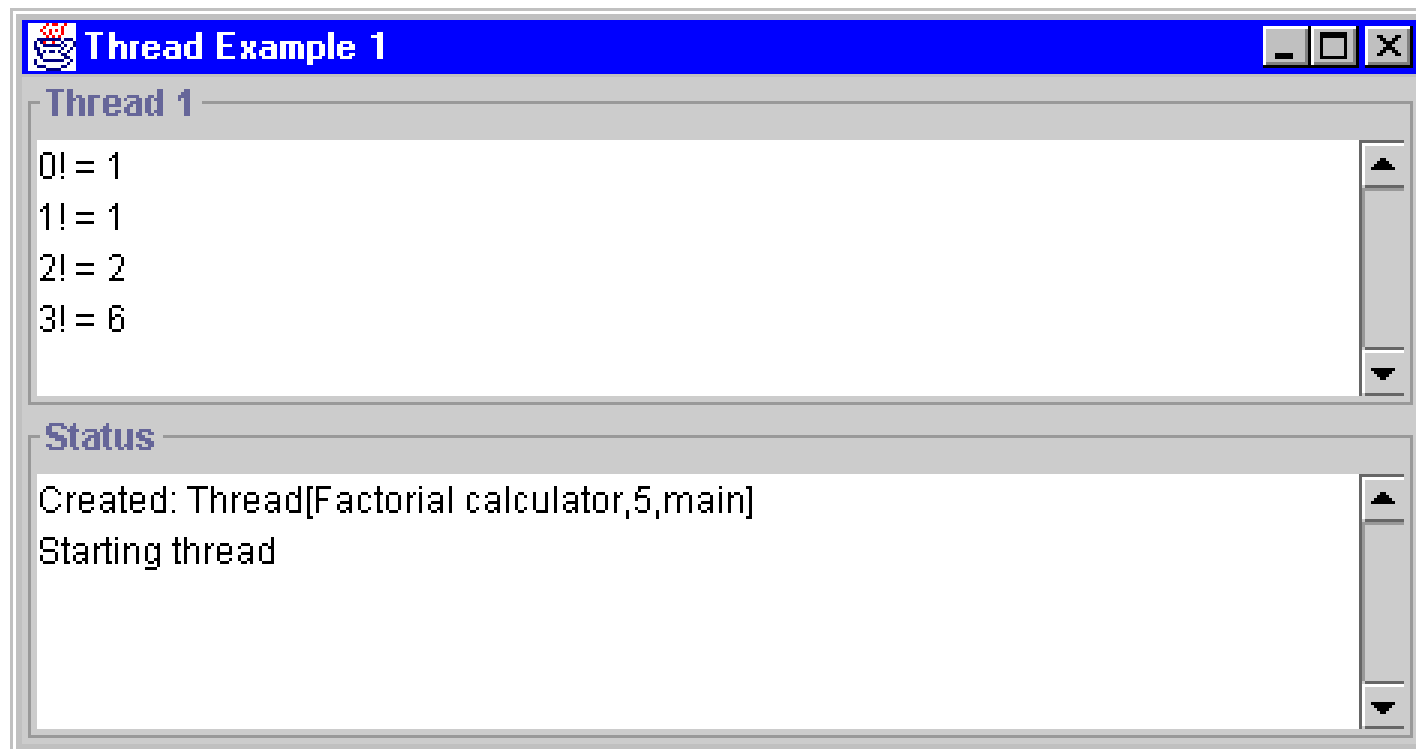
- After `factorialThread.start()` returns, we now have two user-defined threads of execution
  - the thread executing `ThreadExample1.main()`
  - the thread executing `Factorial.run()`
- Each thread sends its output to a different `JTextArea` (see next slide)



---

# *Concurrent Thread Execution*

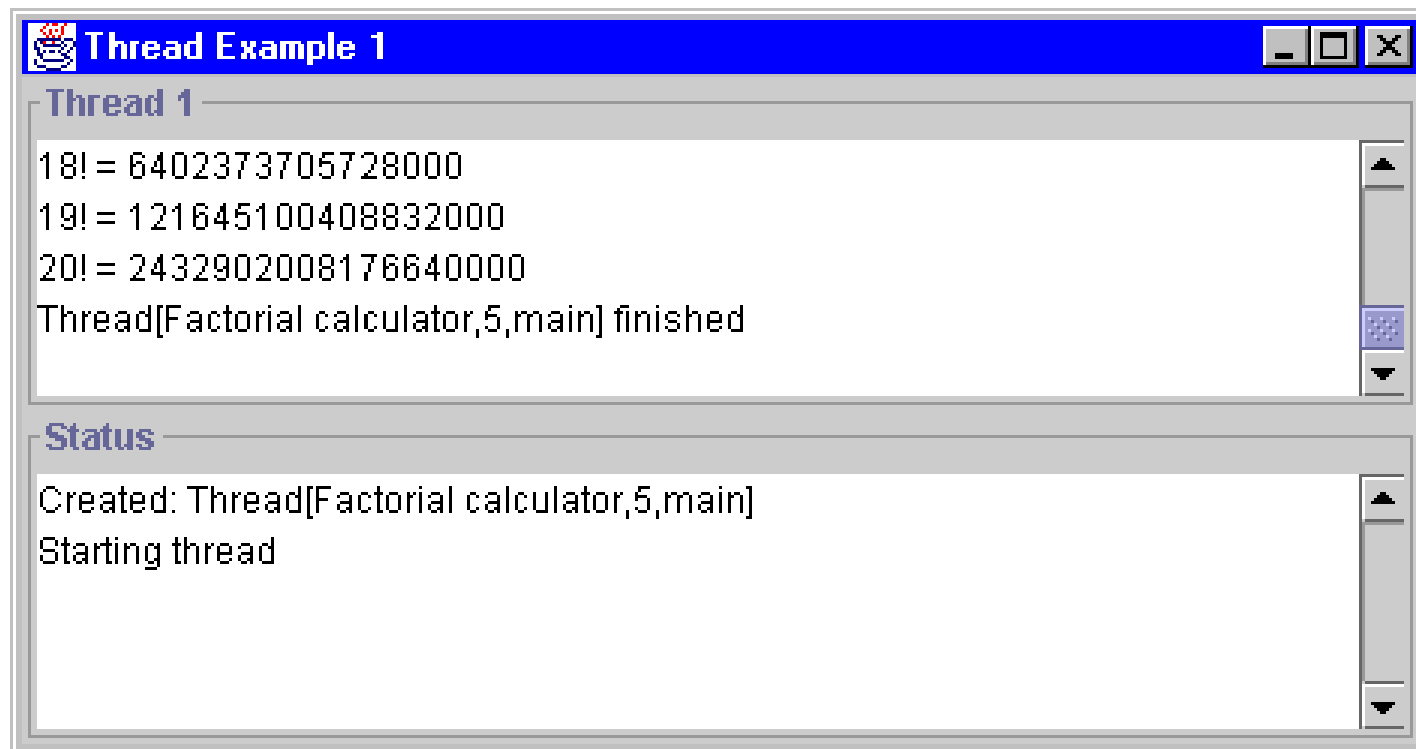
- After calculating 3!



---

# ***Concurrent Thread Execution***

- The factorial thread has terminated



---

## *Thread.sleep()*

- `run()` contains this statement:

```
try {  
    Thread.sleep((int) (Math.random() * 2000));  
} catch (InterruptedException e) {}
```

- `sleep()` is a static method that causes the invoking thread to sleep (temporarily stop executing) for the specified number of milliseconds
- The method throws an `InterruptedException` if the sleeping thread is interrupted by another thread (more about interrupts later in the course)

---

## ***Thread.sleep()***

- `sleep()` is invoked simply to slow down the thread so we can view its progress
- This method invocation is not required to ensure correct thread behaviour

---

## ***Thread.toString()***

- `main()` contains this statement:

```
frame.status.append("Created: " +  
                    factorialThread + '\n');
```

which (implicitly) invokes `Thread.toString()`

- `Thread.toString()` returns a string representation of the thread, containing the thread's *name*, *priority*, and *thread group*:

"Thread[*name*, *priority*, *group*]"

- more about thread priorities when we look at thread scheduling

---

## ***Thread.currentThread()***

- `run()` contains this statement:

```
transcript.append(Thread.currentThread()  
                + " finished\n");
```

which invokes `Thread.currentThread()`

- This class (static) method returns a reference to the `Thread` object for the currently executing thread; i.e., the thread that invoked the method

---

## ***Thread and Program Termination***

- The thread executing `main()` terminates when it reaches the end of that method and returns
- The thread executing `Factorial.run()` terminates when it reaches the end of that method and returns
- The program doesn't terminate after both threads terminate, because the GUI has a background thread that is eligible to run
  - when the frame's Close button is clicked, this thread invokes a window listener that terminates the program by invoking `System.exit()` (see code on Web site)

---

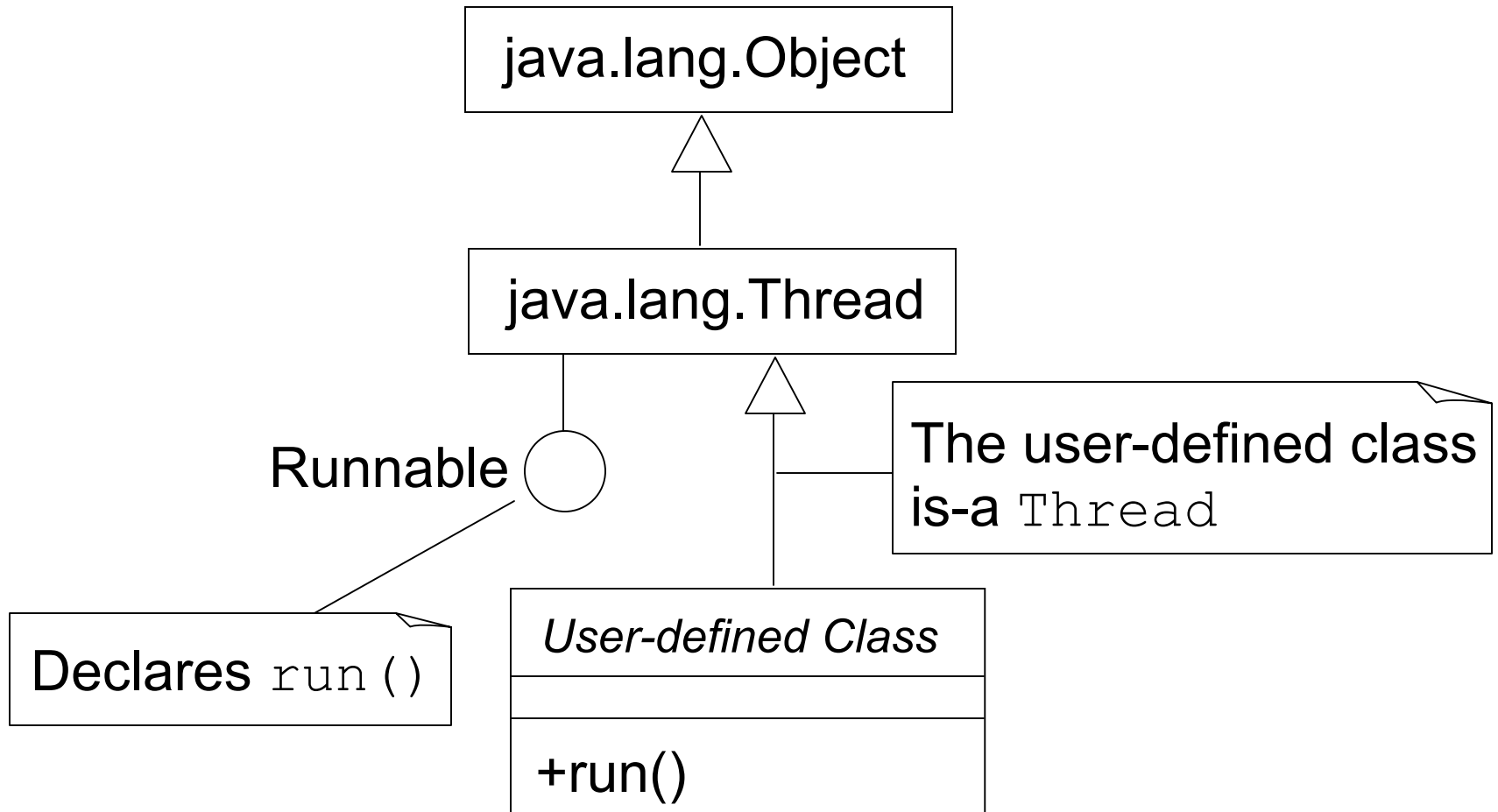
## ***Thread and Program Termination***

- A multithreaded Java program executes until
  - all threads that are not daemon threads have terminated, either by returning from `run()` or by throwing an exception that propagates beyond `run()`, or
  - the `exit()` method of class `Runtime` has been invoked and the security manager has permitted the exit operation to take place



---

# Class Relationships



---

## ***Another Way to Create Threads***

```
class ThreadExample2 extends JFrame
{
    ...
    public static void main(String[] args) {
        ...
        Thread factorialThread = new Thread(
            new Factorial(frame.ta1),
            "Factorial calculator");
        ...
        factorialThread.start();
    }
}
```

---

## ***Another Way to Create Threads***

```
class Factorial implements Runnable
{
    /**
     * The text area where this thread's output
     * will be displayed.
     */
    private JTextArea transcript;

    public Factorial(JTextArea transcript) {
        this.transcript = transcript;
    }

    public void run() {
        // No changes to this method.
    }
}
```

---

## *Another Way to Create Threads*

- As before, this program creates a new Java thread that will begin execution in the `run()` method of the `Factorial` object
- Notice that `Factorial` no longer extends `Thread`; instead, the class declaration states that it implements the `Runnable` interface
  - the compiler would issue an error if `Factorial` did not define a `run()` method

---

## *Another Way to Create Threads*

- The statement in `main()` for creating the new thread is different:

```
Thread factorialThread = new Thread(  
    new Factorial(frame.tal),  
    "Factorial calculator");
```

- This statement creates a new thread by invoking a two-argument `Thread` constructor
  - the first argument is a reference to a `Runnable` object
  - the second argument is the thread name (a reference to a `String` object)

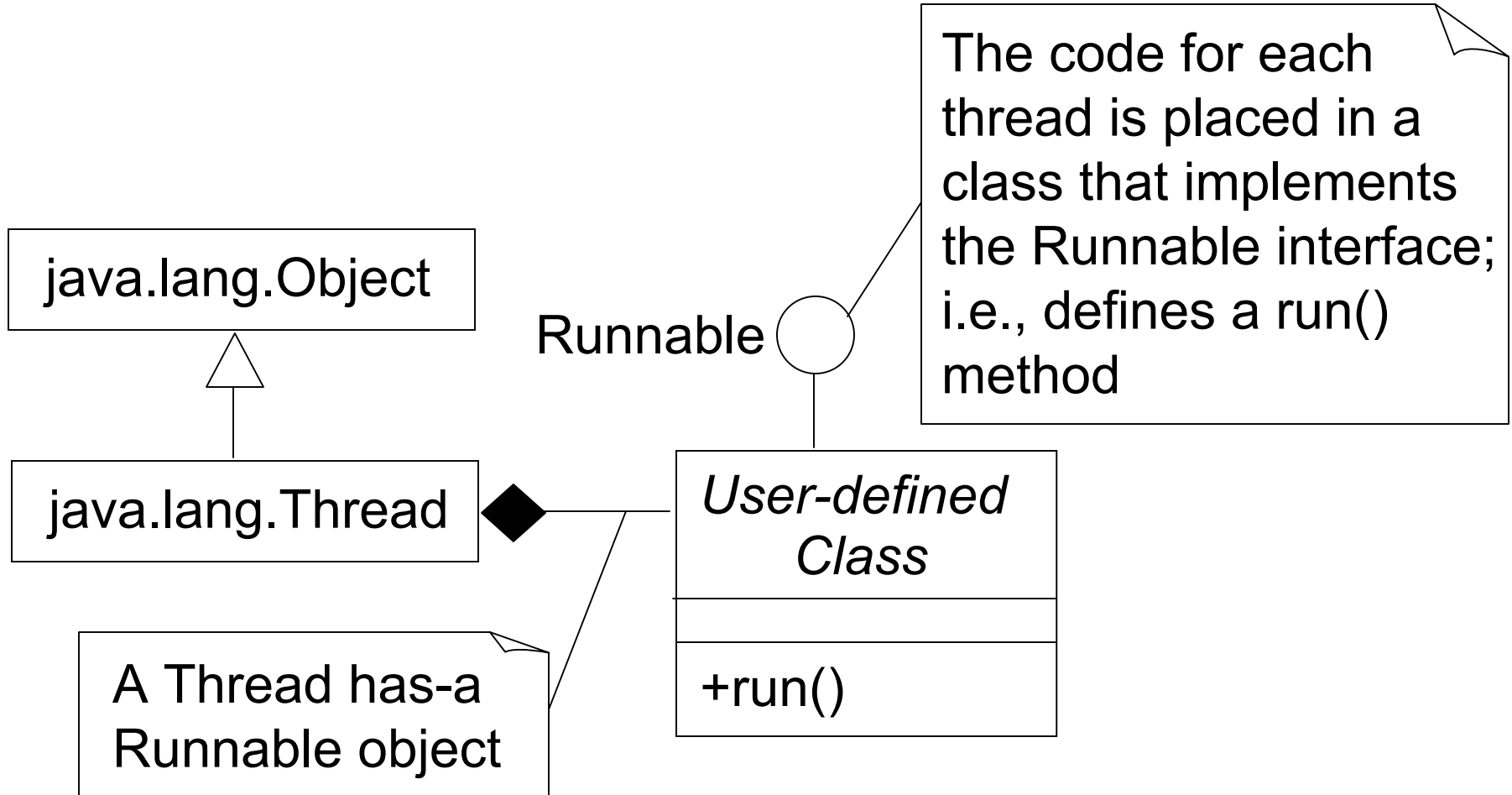
---

## ***Another Way to Create Threads***

- As before, `factorialThread.start()` makes the newly created thread ready to run
  - `factorialThread.start()` invokes the `run()` method defined in `Thread`
  - because the thread was constructed using a separate `Runnable` object, `Thread.run()` invokes the `Runnable` object's `run()` method

---

# ***Class Relationships***



---

## ***Implementing the Runnable Interface***

- `run()` has no parameters and does not return a value
- `run()` must never be explicitly invoked from within the program
  - it is always invoked indirectly, as a result of sending the `start()` message to a `Thread` object



---

## ***Implementing the Runnable Interface***

- The thread that is executing `run()` can send messages to objects, but doing so does not cause those objects to become threads
- When a thread sends a message to an object, the method is executed in the context of the invoking thread

---

## ***Creating Multiple Threads***

- Any thread can create multiple threads that execute the same code
- We create a separate `Thread` object for each thread, and pass each `Thread` object a unique instance of the `Runnable` object

---

## *Creating Multiple Threads*

- Example: creating two completely independent threads from `Factorial`

```
Thread factorialThread = new Thread(  
    new Factorial(frame.ta1),  
    "Factorial calculator");  
Thread secondFactorialThread = new Thread(  
    new Factorial(frame.ta2),  
    "Second factorial calculator");  
...  
factorialThread.start();  
secondFactorialThread.start();
```

---

## ***Creating Multiple Threads***

- Any thread can create one or more threads from *different* `Runnable` classes
- The program in `ThreadExample3.java` creates two threads from `Factorial` and a third thread that calculates Fibonacci numbers (its `Thread` object is passed an instance of `Fibonacci`)
- Code fragment: see next slide

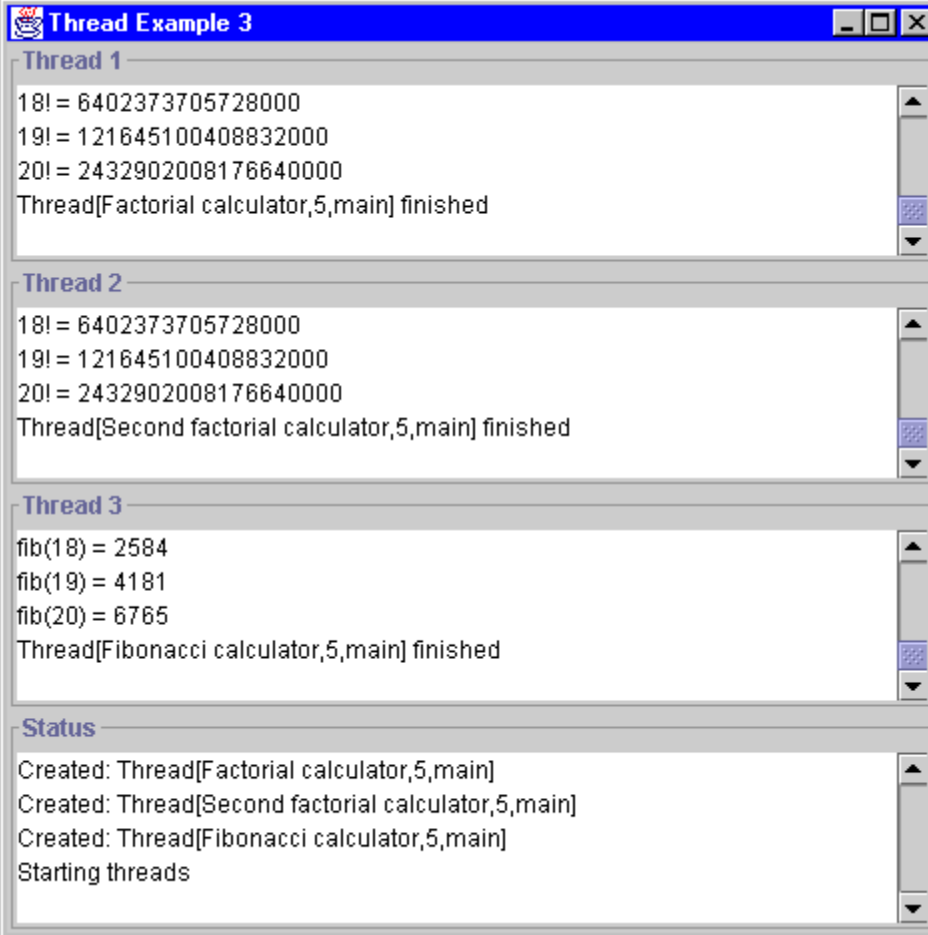
---

## *Creating Multiple Threads*

```
Thread factorialThread = new Thread(  
    new Factorial(frame.ta1),  
    "Factorial calculator");  
Thread secondFactorialThread = new Thread(  
    new Factorial(frame.ta2),  
    "Second factorial calculator");  
Thread fibonacciThread = new Thread(  
    new Fibonacci(frame.ta3),  
    "Fibonacci calculator");  
...  
factorialThread.start();  
secondFactorialThread.start();  
fibonacciThread.start();
```

---

# *Program Output*



```
Thread Example 3

Thread 1
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
Thread[Factorial calculator,5,main] finished

Thread 2
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
Thread[Second factorial calculator,5,main] finished

Thread 3
fib(18) = 2584
fib(19) = 4181
fib(20) = 6765
Thread[Fibonacci calculator,5,main] finished

Status
Created: Thread[Factorial calculator,5,main]
Created: Thread[Second factorial calculator,5,main]
Created: Thread[Fibonacci calculator,5,main]
Starting threads
```

---

## ***Concurrent Thread Execution***

- At any time, a single CPU is executing the code from a single thread
- As such, true concurrency between threads on a uniprocessor is not possible
- It is the job of the *scheduler* and the *context switcher* to create the illusion of concurrency by sharing the CPU between the threads that are eligible to execute
- Will look at scheduling later in the course, but for now we will pretend that all threads that are not temporarily blocked from executing (e.g., by sleeping) execute concurrently