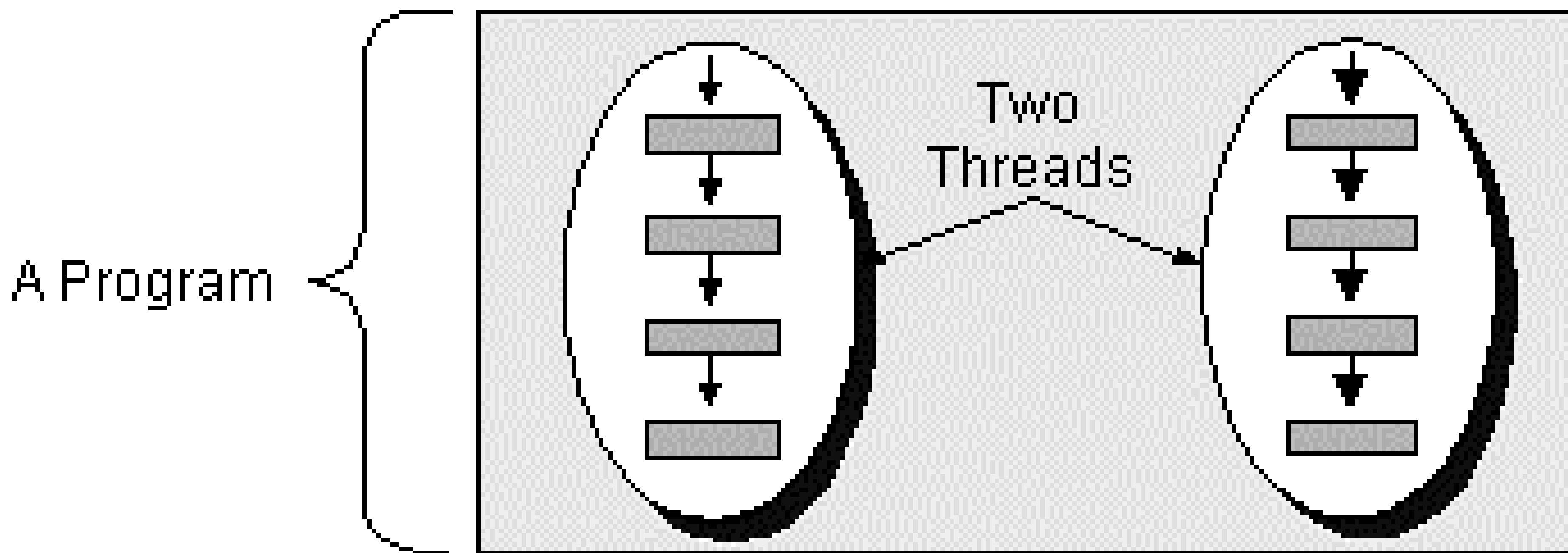
# Java

## Threads

*Presented by Bartosz Sakowicz*

# Threads - basics

**A thread is a single sequential flow of control within a program**. A thread itself is not a program; it cannot run on its own. Rather, it runs within a program.



# Threads – basics(2)

- Some texts use the name **lightweight process** instead of thread.
- A thread is similar to a real process in that a thread and a running program are both a single sequential flow of control.
- A thread is considered lightweight because it runs within the context of a full-blown program and takes advantage of the resources allocated for that program and the program's environment.
- As a sequential flow of control, a thread must carve out some of its own resources within a running program. (It must have its own execution stack and program counter for example) The code running within the thread works only within that context. Thus, some other texts use **execution context** as a synonym for thread.

# Using the Timer and TimerTask

```
import java.util.*;
public class Reminder {
    Timer timer;
    public Reminder(int seconds) {
        timer = new Timer();
        timer.schedule(new RemindTask(), seconds*1000); }
    class RemindTask extends TimerTask {
        public void run() {
                System.out.println("Time's up!");
                timer.cancel(); //Terminate the timer thread } }
    public static void main(String args[]) {
new Reminder(5); System.out.println("Task scheduled."); } }
```

# Scheduling task for execution at particular hour

```
...
//Date corresponding to 10:01:00 pm today.
Calendar calendar = Calendar.getInstance();
calendar.set(Calendar.HOUR_OF_DAY, 22);
calendar.set(Calendar.MINUTE, 1);
calendar.set(Calendar.SECOND, 0);
Date time = calendar.getTime();
timer = new Timer();
timer.schedule(new RemindTask(), time);
...
```

# Stoping Timer Threads

By default, a program keeps running as long as its timer threads are running. You can terminate a timer thread in four ways:

- Invoke cancel on the timer. You can do this from anywhere in the program, such as from a timer task's run method.
- Make the timer's thread a "daemon" by creating the timer like this: new Timer(true). If the only threads left in the program are daemon threads, the program exits.
- After all the timer's scheduled tasks have finished executing, remove all references to the Timer object. Eventually, the timer's thread will terminate.
- Invoke the System.exit method, which makes the entire program (and all its threads) exit.

# Programming threads

There are two techniques for providing a run method for a thread:

- Subclassing Thread and Overriding run
- Implementing the Runnable Interface

java.lang.Thread represents a thread of control. It offers methods that allow you to set the priority of the thread, to assign a thread to a thread group and to control the running state of the thread (e.g., whether it is running or suspended).

The java.lang.Runnable interface represents the body of a thread. Classes that implement the Runnable interface provide their own run() methods that determine what their thread actually does while running. If a Thread is constructed with a Runnable object as its body, the run() method on the Runnable will be called when the thread is started.

# Programming threads(2)

**There are two ways to define a thread:**

One is to subclass Thread, override the run() method, and then instantiate your Thread subclass.

The second is to define a class that implements the Runnable method (i.e., define a run() method) and then pass an instance of this Runnable object to the Thread() constructor.

In either case, the result is a Thread object, where the run() method is the body of the thread.

# Programming threads(3)

When you call the **start()** method of the Thread object, the interpreter creates a new thread to execute the **run()** method.

This new thread continues to run until the run() method exits, at which point it ceases to exist. Meanwhile, the original thread continues running itself, starting with the statement following the start() method.

# Programming threads(4)

```
final List list; // Some unsorted list ; initialized elsewhere
 /** A Thread class for sorting a List in the background */
class BackgroundSorter extends Thread {
     List l;
     public BackgroundSorter(List l) { this.l = l; }
      public void run() { Collections.sort(l); }
 }
```

# Programming threads(5)

```
...
Thread sorter = new BackgroundSorter(list); // Start it running;
/** the new thread runs the run() method above, while the original thread continues with whatever statement comes next.*/
sorter.start();
 // Here's another way to define a similar thread
Thread t = new Thread(new Runnable() {  public void run() { Collections.sort(list); }  });
t.start();
...
```

# Subclassing Thread

```
public class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str); }
    public void run() {
        for (int i = 0; i < 10; i++) {
                System.out.println(i + " " + getName());
                try {
                        sleep((long)(Math.random() * 1000));
                } catch (InterruptedException e) {}
        } System.out.println("DONE! " + getName());
    }
}
```

# Implementing Runnable

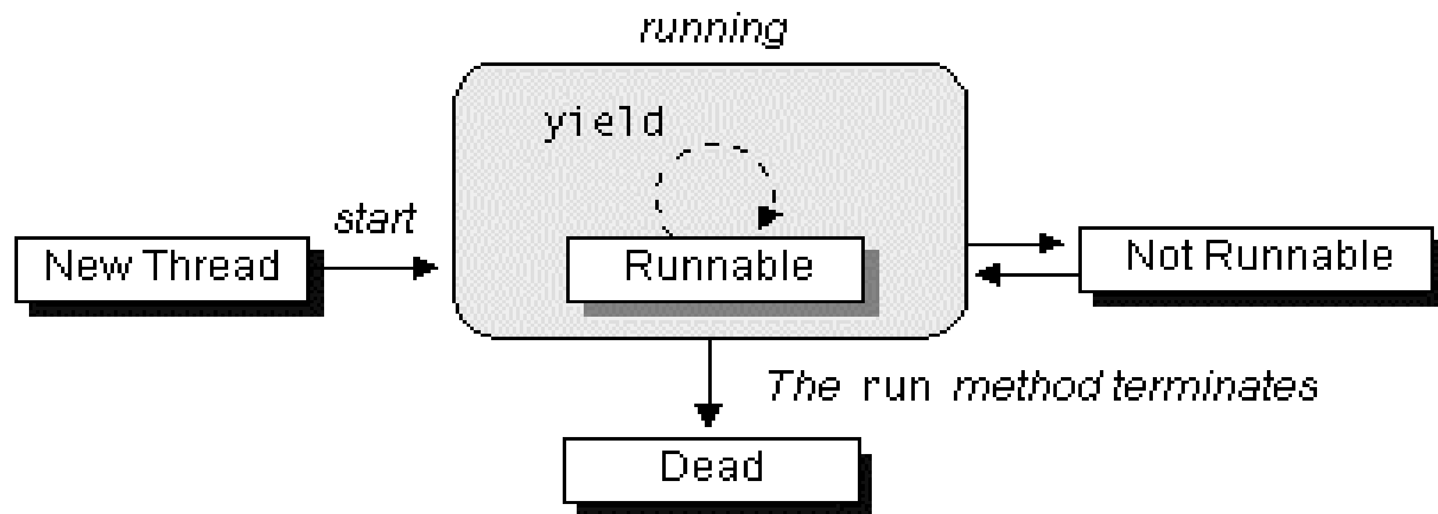
The difference between the two classes is that a Thread is supposed to represent how a thread of control runs (its priority level, the name for the thread), and a Runnable defines what a thread runs. In both cases, defining a subclass usually involves implementing the run() method to do whatever work you want done in the separate thread of control.

**If your class must subclass some other class (the most common example being Applet), you should use Runnable.**

# Creating threads

```
public class TwoThreadsDemo {
    public static void main (String[] args) {
        new SimpleThread("One").start();
        new SimpleThread("Two").start();
    }
}
```

# The Thread Life Cycle



# The Thread Life Cycle(2)

```
public class Clock extends Applet implements Runnable {
    private Thread clockThread = null;
    public void start() {
        if (clockThread == null) {
                // create a Thread
                clockThread = new Thread(this, "Clock");
                // start a Thread
                clockThread.start();
        }
    }
...
```

# Making a Thread not runnable

A thread becomes not runnable when one of these events occurs:

- Its sleep method is invoked.
- The thread calls the wait method to wait for a specific condition to be satisifed.
- The thread is blocking on I/O.

The escape route for every entrance into the not runnable state:

- If a thread has been put to sleep, then the specified number of milliseconds must elapse.
- If a thread is waiting for a condition, then another object must notify the waiting thread of a change in condition by calling notify or notifyAll.
- If a thread is blocked on I/O, then the I/O must complete.

# Stopping a Thread

A thread arranges for its own death by having a run method that terminates naturally. For example, the while loop in this run method is a finite loop - it will iterate 100 times and then exit:

```
public void run() {
        int i = 0;
        while (i < 100) {
                i++;
                System.out.println("i = " + i);
        }
}
```

A thread with this run method dies naturally when the loop completes and the run method exits.

# Thread priority

- When a Java thread is created, it inherits its priority from the thread that created it.
- You can also modify a thread's priority at any time after its creation using the **setPriority** method.
- Thread priorities are integers ranging between MIN_PRIORITY and MAX_PRIORITY (constants defined in the Thread class).
- The higher the integer, the higher the priority.
- At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution.
- At any given time, the highest priority thread is running, but this is not guaranteed. For this reason, use priority only to affect scheduling policy for efficiency purposes. Do not rely on thread priority for algorithm correctness.

# The consumer/producer example (synchronization)

- The Producer generates an integer between 0 and 9 and stores it in a CubbyHole object, and prints the generated number.
- The Consumer consumes all integers from the CubbyHole (the exact same object into which the Producer put the integers) as quickly as they become available.

# The consumer/producer ...(2)

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;
    public Producer(CubbyHole c, int number) {
        cubbyhole = c; this.number = number; }
    public void run() {
        for (int i = 0; i < 10; i++) {
        cubbyhole.put(i);
        System.out.println("Producer #" + this.number + " put: " + i);
        try {
                sleep((int)(Math.random() * 100));
         } catch (InterruptedException e) { } } } }
```

# The consumer/producer ...(3)

```
public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;
    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number; }
    public void run() {
    int value = 0;
    for (int i = 0; i < 10; i++) {
        value = cubbyhole.get();
        System.out.println("Consumer #" + this.number + " got: " +
        value); } } }
```

# The consumer/producer ...(4)

One problem arises when the Producer is quicker than the Consumer (e.g. higher priority) and generates two numbers before the Consumer has a chance to consume the first one. Part of the output might look like this:

*Consumer #1 got: 3*

*Producer #1 put: 4*

*Producer #1 put: 5*

*Consumer #1 got: 5*

Or Consumer is quicker then Producer:

*Producer #1 put: 4*

*Consumer #1 got: 4*

*Consumer #1 got: 4*

*Producer #1 put: 5*

# The consumer/producer ...(5)

**The main program:**

```
public class ProducerConsumerTest {
    public static void main(String[] args) {
            CubbyHole c = new CubbyHole();
            Producer p1 = new Producer(c, 1);
            Consumer c1 = new Consumer(c, 1);
            p1.start();
            c1.start();
    }
}
```

# The consumer/producer ...(6)

Problems such as described are called **race conditions**.

The activities of the Producer and Consumer must be synchronized in two ways:

- The two threads must not simultaneously access the CubbyHole. A Java thread can prevent this from happening by locking an object. When an object is locked by one thread and another thread tries to call a synchronized method on the same object, the second thread will block until the object is unlocked.
- The two threads must do some simple coordination. That is, the Producer must have some way to indicate to the Consumer that the value is ready and the Consumer must have some way to indicate that the value has been retrieved.

# The consumer/producer ...(7)

**Will it work?**

```
public class CubbyHole {
    private int contents;
    private boolean available = false;
    public synchronized int get() {
        if (available == true) { available = false; return contents; }
    }
    public synchronized void put(int value) {
        if (available == false) { available = true; contents = value; }
    }
}
```

**Answer: No, it will not.**

# The consumer/producer ...(8)

**The correct implementation of get method:**

```
public synchronized int get() {
    while (available == false) {
        try {
                wait(); // wait for Producer to put value
        } catch (InterruptedException e) { }
    }
    available = false;
     notifyAll(); // notify Producer that value has been retrieved
    return contents;
}
```

# The consumer/producer ...(9)

**The correct implementation of put method:**

```
public synchronized void put(int value) {
    while (available == true) {
        try {
                wait();  // wait for Consumer to get value
        } catch (InterruptedException e) { }
    }
    contents = value;
    available = true;
    notifyAll(); // notify Consumer that value has been set
}
```

# notifyAll, notify and wait

The **notifyAll** method wakes up all threads waiting on the object in question (in this case, the CubbyHole). The awakened threads compete for the lock. One thread gets it, and the others go back to waiting. The Object class also defines the **notify** method, which arbitrarily wakes up one of the threads waiting on this object.

There are two other versions of the wait method:

**wait(long *timeout*)**

Waits for notification or until the timeout period has elapsed. timeout is measured in milliseconds.

**wait(long *timeout*, int *nanos*)**

Waits for notification or until timeout milliseconds plus nanos nanoseconds have elapsed.

# Starvation and deadlock

**The philosophers problem:**

Five philosophers are sitting at a round table. In front of each philosopher is a bowl of rice. Between each pair of philosophers is one chopstick. Before an individual philosopher can take a bite of rice he must have two chopsticks--one taken from the left, and one taken from the right. The philosophers must find some way to share chopsticks such that they all get to eat.

**What is a solution, when all starts immediately?**

# Starvation and deadlock (2)

We can change the rules by numbering the chopsticks 1 through 5 and insisting that the philosophers pick up the chopstick with the lower number first. The philosopher who is sitting between chopsticks 1 and 2 and the philosopher who is sitting between chopsticks 1 and 5 must now reach for the same chopstick first (chopstick 1) rather than picking up the one on the right. Whoever gets chopstick 1 first is now free to take another one. Whoever doesn't get chopstick 1 must now wait for the first philosopher to release it. Deadlock is not possible.

**The best choice is to prevent deadlock rather than to try and detect it. Deadlock detection is very complicated . The same with starvation.**

# Grouping Threads

Every Java thread is a member of a **thread group**. Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually. For example, you can start or suspend all the threads within a group with a single method call. Java thread groups are implemented by the ThreadGroup class in the java.lang package.

The runtime system puts a thread into a thread group during thread construction. When you create a thread, you can either allow the runtime system to put the new thread in some reasonable default group or you can explicitly set the new thread's group. The thread is a permanent member of whatever thread group it joins upon its creation - you cannot move a thread to a new group after the thread has been created.

# The default Thread Group

- If you create a new Thread without specifying its group in the constructor, the runtime system automatically places the new thread in the same group as the thread that created it.
- When a Java application first starts up, the Java runtime system creates a **ThreadGroup** named **main**. **Unless specified otherwise, all new threads that you create become members of the main thread group.**
- If you create a thread within an applet, the new thread's group may be something other than main, depending on the browser or viewer that the applet is running in

# Creating Thread in a Group

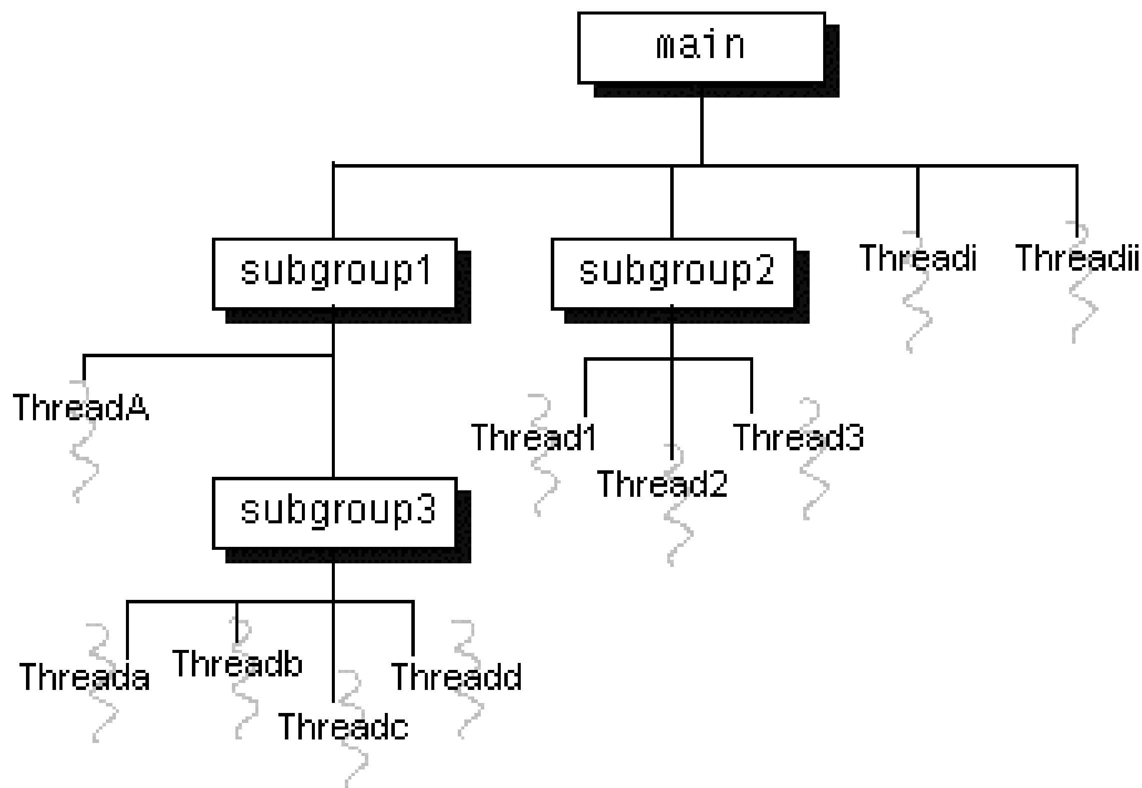
```
ThreadGroup myTG =
    new ThreadGroup( "My Group of Threads");
Thread myThread =
    new Thread(myTG, "a thread for my group");
```

To find out what group a thread is in, you can call its getThreadGroup method:

```
theGroup = myThread.getThreadGroup();
```

# The ThreadGroup class

ThreadGroups can contain not only threads but also other ThreadGroups. The top-most thread group in a Java application is the thread group named **main**. You can create threads and thread groups in the main group. You can also create threads and thread groups in subgroups of **main**.




**For methods you can perform on ThreadGroup please refer to Java API.**