

Java Concurrent Programming

An Introduction



Presentation Topics



In this presentation we will cover:

- Introduction to Concurrent Programming
- Java's Threading Model
- Managing Threads

Overview



When we are done, you should be able to:

- ◉ Describe concurrent programming
- ◉ Identify two ways to create a thread
- ◉ Discuss the phases of a thread's lifecycle

Introduction to Concurrent Programming

An Overview of Concepts





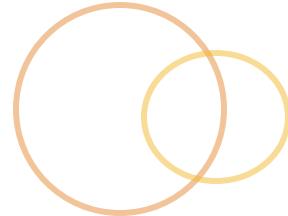
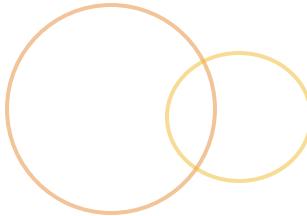
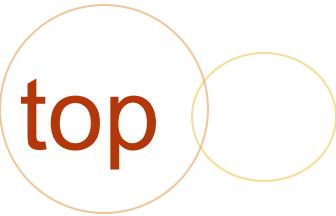
What is concurrent software?

- ◉ Software that can do more than one *thing* at a time
- ◉ Concurrent *things* are usually considered:
 - ◉ Processes
 - ◉ Threads
- ◉ Both are units of execution

What is a Process?



- ◉ Usually described in terms of operating systems
- ◉ Characteristics of a process:
 - ◉ Distinct execution context
 - ◉ Execution information (program code)
 - ◉ Memory
 - ◉ Priority
 - ◉ Hierarchy
 - ◉ Managed by scheduler
 - ◉ State



top

12:35:37

Processes: 120 total, 4 running, 1 stuck, 115 sleeping, 560 threads
Load Avg: 0.79, 0.55, 0.38 CPU usage: 1.43% user, 2.63% sys, 95.92% idle
SharedLibs: 1344K resident, 0B data, 0B linkedit. MemRegions: 16500 total, 895M resident, 63M private, 378M shared.
PhysMem: 907M wired, 1517M active, 348M inactive, 2772M used, 1321M free.
VM: 270G vsize, 1026M framework vsize, 96251(0) pageins, 0(0) pageouts.
Networks: packets: 16208/19M in, 5933/821K out. Disks: 52404/1677M read, 10985/223M written.

PID	PPID	COMMAND	%CPU	STATE	#TH	RSIZE	VSIZE
1533	177	screencapture	0.0	sleeping	2	3468K	2431M
1531	1515	top	4.0	running	1/1	1764K	2376M
1515	1514	bash	0.0	sleeping	1	1256K	2376M
1514	1428	login	0.0	sleeping	2	4572K	2409M
1511	1509	less	0.0	sleeping	1	724K	2376M
1509	1505	sh	0.0	sleeping	1	384K	2376M
1505	1504	sh	0.0	sleeping	1	444K	2376M
1504	1503	sh	0.0	sleeping	1	896K	2376M
1503	1431	man	0.0	sleeping	1	696K	2376M
1499	218	sleep	0.0	sleeping	1	472K	2376M
1443	1441	WebProcess	0.0	sleeping	11	76M	3616M
1441	164	Safari	0.5	sleeping	21	57M	3594M
1431	1430	bash	0.0	sleeping	1	1252K	2376M
1430	1428	login	0.0	sleeping	2	4628K	2409M
1428	164	Terminal	0.5	sleeping	5	24M	2495M
1400-	164	Microsoft Databa	0.0	sleeping	4	9576K	698M
1399-	164	Microsoft AU Dae	0.0	sleeping	3	3568K	656M
1395-	164	Microsoft PowerP	0.5	sleeping	7	95M	868M
1375	164	printtool	0.0	sleeping	2	1916K	2408M
1371-	1367	fsnotifier	0.0	sleeping	3	1120K	615M
1367	164	idea	2.2	running	45/1	277M	3963M
1351	164	AppleSpell	0.0	sleeping	2	3100K	2430M
1342	164	mdworker	0.0	sleeping	4	17M	2439M
1329	1325	cfprefsd	0.0	sleeping	2	1108K	2408M

top -stats pid,ppid,command,cpu,state,threads,rsize,vsize

What is a Thread?



- ◉ Typically referred to as *thread of execution*
- ◉ Characteristics of thread:
 - ◉ Operate within a process
 - ◉ Execution information
 - ◉ Memory (can be shared across threads)
 - ◉ Priority
 - ◉ Hierarchy
 - ◉ Managed by scheduler
 - ◉ State

top [again]



Processes: 120 total, 3 running, 117 sleeping, 607 threads 12:47:57
Load Avg: 0.53, 0.65, 0.53 CPU usage: 2.57% user, 4.68% sys, 92.74% idle SharedLibs: 1312K resident, 0B data, 0B linkedit.
MemRegions: 18011 total, 1046M resident, 67M private, 415M shared.
PhysMem: 909M wired, 1591M active, 537M inactive, 3037M used, 1057M free.
VM: 272G vsize, 1026M framework vsize, 103777(0) pageins, 0(0) pageouts. Networks: packets: 21380/22M in, 10688/1961K out.
Disks: 55475/1705M read, 16259/308M written.

PID	PPID	COMMAND	%CPU	STATE	#TH	RSIZE	VSIZE
1721	177	screencapture	0.1	sleeping	2	3464K	2431M
1720	1515	top	5.8	running	1/1	2084K	2376M
1713	1	ocspd	0.0	sleeping	2	2268K	2396M
1711	164	assistantd	0.0	sleeping	4	9224K	2434M
1710	1431	top	5.7	running	1/1	2344K	2376M
1707	164	DictationIM	0.0	sleeping	3	15M	2455M
1695	164	mdworker	0.0	sleeping	3	9016K	2417M
1657	1	java	0.1	sleeping	26	90M	2809M
1656	1	java	0.0	sleeping	7	90M	2100M
1599	164	quicklookd	0.0	sleeping	4	10M	2949M
1515	1514	bash	0.0	sleeping	1	1256K	2376M
1514	1428	login	0.0	sleeping	2	4572K	2409M
1499	218	sleep	0.0	sleeping	1	472K	2376M
1443	1441	WebProcess	0.0	sleeping	11	163M	3724M
1441	164	Safari	0.5	sleeping	21	62M	3596M
1431	1430	bash	0.0	sleeping	1	1260K	2376M
1430	1428	login	0.0	sleeping	2	4628K	2409M
1428	164	Terminal	1.3	sleeping	7	29M	2514M
1400-	164	Microsoft Databa	0.0	sleeping	3	9572K	697M
1399-	164	Microsoft AU Dae	0.0	sleeping	2	3564K	655M
1395-	164	Microsoft PowerP	0.5	sleeping	9	110M	890M
1375	164	printtool	0.0	sleeping	2	1916K	2408M
1371-	1367	fsnotifier	0.0	sleeping	3	1144K	615M
1367	164	idea	2.0	sleeping	45	278M	3963M

Concurrent Programming with Java



Built into the Java platform

- Considered multi-threaded
- Runs within single JVM process
- Managed by JVM thread scheduler
- Been around since JDK 1.0
- Available to all applications

VisualVM



VisualVM 1.3.4

Applications Start Page Tomcat (pid 1657)

Overview Monitor Threads Sampler Profiler

Tomcat (pid 1657)

Monitor CPU Memory Classes Threads

Uptime: 11 min 55 sec

Perform GC Heap Dump

CPU

CPU usage: 0.2% GC activity: 0.0%

0% 50% 100%

12:54:15 PM 12:54:30 PM 12:54:45 PM 12:55:00 PM 12:55:15 PM

■ CPU usage ■ GC activity

Heap PermGen

Size: 87,162,880 B Used: 11,529,136 B
Max: 132,120,576 B

75 MB 50 MB 25 MB 0 MB

12:54:15 PM 12:54:30 PM 12:54:45 PM 12:55:00 PM 12:55:15 PM

■ Heap size ■ Used heap

Classes

Total loaded: 3,719 Shared loaded: 0
Total unloaded: 0 Shared unloaded: 0

4,000 2,000 0

12:54:15 PM 12:54:30 PM 12:54:45 PM 12:55:00 PM 12:55:15 PM

■ Total loaded classes ■ Shared loaded classes

Threads

Live: 20 Daemon: 19
Live peak: 22 Total started: 26

20 10 0

12:54:15 PM 12:54:30 PM 12:54:45 PM 12:55:00 PM 12:55:15 PM

■ Live threads ■ Daemon threads

Local VisualVM IntelliJ Platform (pid 1367) Tomcat (pid 1657)

Remote VM Core dumps Snapshots

VisualVM [Thread View]



VisualVM 1.3.4

Applications Start Page Tomcat (pid 1657)

Overview Monitor Threads Sampler Profiler

Tomcat (pid 1657)

Threads Threads visualization Thread Dump

Live threads: 20 Daemon threads: 19

Timeline Table Details

Show: All Threads

Thread	Running	Sleeping	Wait	Monitor	Total
*** Profiler Agent Communication Thread	2:23.009 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	2:23.009
RMI TCP Connection(3)-10.1.10.17	2:23.009 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	2:23.009
RMI TCP Accept-0	2:23.009 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	2:23.009
Attach Listener	2:23.009 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	2:23.009
TP-Processor4	2:23.009 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	2:23.009
http-8080-Acceptor-0	2:23.009 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	2:23.009
Signal Dispatcher	2:23.009 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	2:23.009
main	2:23.009 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	2:23.009
*** Profiler Agent Special Execution Thread 6	0.0 (0.0%)	0.0 (0.0%)	2:23.009 (100.0%)	0.0 (0.0%)	2:23.009
*** JFluid Monitor thread ***	0.0 (0.0%)	2:23.009 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	2:23.009
RMI TCP Connection(idle)	0.0 (0.0%)	0.0 (0.0%)	40.917 (100.0%)	0.0 (0.0%)	40.917
JMX server connection timeout 26	0.0 (0.0%)	0.0 (0.0%)	2:23.009 (100.0%)	0.0 (0.0%)	2:23.009
RMI Scheduler(0)	0.0 (0.0%)	0.0 (0.0%)	2:23.009 (100.0%)	0.0 (0.0%)	2:23.009
TP-Monitor	0.0 (0.0%)	0.0 (0.0%)	2:23.009 (100.0%)	0.0 (0.0%)	2:23.009
TP-Processor3	0.0 (0.0%)	0.0 (0.0%)	2:23.009 (100.0%)	0.0 (0.0%)	2:23.009
TP-Processor2	0.0 (0.0%)	0.0 (0.0%)	2:23.009 (100.0%)	0.0 (0.0%)	2:23.009
TP-Processor1	0.0 (0.0%)	0.0 (0.0%)	2:23.009 (100.0%)	0.0 (0.0%)	2:23.009
ContainerBackgroundProcessor[StandardEng...	0.0 (0.0%)	2:23.009 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	2:23.009

Java's Threading Model



What is a Thread?



- ◉ All characteristics previously described hold true
 - ◉ Operate within a process
 - ◉ Execution information
 - ◉ etc.
- ◉ What is a Thread?
 - ◉ First and foremost: a Java Object (memory)
 - ◉ Secondly: a `java.lang.Thread`

Java Thread as an `java.lang.Object`



- Can be treated as regular object
 - Constructors, Methods, etc.
 - `toString`, `hashCode`, `equals`
- Contains core threading “control” methods
 - `wait`
 - `notify/notifyAll`

Java Thread as a `java.lang.Thread`



- Special Type of `java.lang.Object`
- Contains information dealing with:
 - Thread identification (name and group)
 - Priority
 - Executable code
 - Hierarchy
 - Management

Creating a Thread



Two ways to create Thread

1. Full definition - subclass Thread
2. Partial definition - implement Runnable

Choosing a Thread Development Model



- Full-Definition (Subclass)
 - Easy to implement
 - Full access to Thread's inner-workings and phase transitions
 - Not-reusable

- Partial Definition (Runnable)
 - Easy to implement
 - Limited access to Thread's inner-workings
 - Reusable

Thread Development: Full Definition



- Create a class that extends Thread
- Provide fields, constructors, and methods
- Override run to provide “execution information”

Full Definition Example



```
class PrimeThread extends Thread {  
    long minPrime;  
    PrimeThread(long minPrime) {  
        this.minPrime = minPrime;  
    }  
  
    public void run() {  
        // compute primes larger than minPrime  
        . . .  
    }  
}
```



Runnable [Partial] Definition

- ◉ Create a class that implements `java.lang.Runnable`
 - ◉ Class is not actually a Thread
 - ◉ It will be used by a Thread
 - ◉ Provide fields, constructors, and methods
- ◉ Implement `run` to provide “execution information”



Partial Definition Example

```
class PrimeRun implements Runnable {  
    long minPrime;  
  
    PrimeRun(long minPrime) {  
        this.minPrime = minPrime;  
    }  
  
    public void run() {  
        // compute primes larger than minPrime  
        . . .  
    }  
}
```



Instantiating a Thread

- Full Implementation
 - Create instance of class
 - `Thread t = new PrimeThread();`
- Partial Implementation
 - Create instance of Runnable
 - `Runnable pr = new PrimeRun();`
 - Pass Runnable to constructor of Thread class
 - `Thread t = new Thread(pr);`
- **NOTE:** This does not start the thread

Starting a Thread



- Creating a new Thread does not start the thread
- Thread needs to be registered with the “Thread Scheduler”
 - Scheduler is part of JVM
 - Scheduler determines when the thread runs

```
Thread t = new PrimeThread();  
t.start();
```

```
Runnable pr = new PrimeRun();  
Thread t = new Thread(pr);  
t.start();
```

LAB: Basic Thread



- Create a simple application that creates a Thread and causes it to run.
- The Thread should be a sub-class of the Thread class. It should keep track of when it was started and when it runs. In the run method, it should determine how long it “had to wait” before it transitioned from the runnable state to the running state
- Duration: 20 minutes



Basic Thread Lab: Solution

```
1 package labs.solutions.concurrent.basic;  
2  
3 /** * */  
4 public class SimpleThreadExample {  
5     public static void main(String[] args) {  
6         Thread t = new SimpleThread();  
7         t.start();  
8     }  
9 }  
10
```

```
1 package labs.solutions.concurrent.basic;  
2  
3 /** * */  
4 public class SimpleThread extends Thread {  
5  
6     private long startTime;  
7     private long runTime;  
8  
9     @Override  
10    public void run() {  
11        runTime = System.nanoTime();  
12        System.out.println("SimpleThread.run(): run was called");  
13        System.out.printf("Elapsed time: %d nanoseconds\n", (runTime - startTime));  
14    }  
15  
16    @Override  
17    public void start() {  
18        System.out.println("SimpleThread.start(): start was called");  
19        startTime = System.nanoTime();  
20        super.start();  
21    }  
22  
23 }  
24  
25 }  
26
```

Thread Lifecycle





What is lifecycle of a Thread?

- ◉ 5 phases of a thread
 - ◉ New
 - ◉ Runnable
 - ◉ Running
 - ◉ Blocked or waiting
 - ◉ Dead
- ◉ Lifecycle and lifecycle transitions managed by the Thread Scheduler (aka JVM)

Lifecycle Phase: New



- Thread is in the “new” phase after it has been created
- Newly created Thread inherits execution information from “Parent” thread
- **NOTE:** A Thread in the new phase is not registered with the Thread Scheduler
- **NOTE:** A Thread can only ever be in the “new” phase one time



Lifecycle Phase: Runnable

- Phase describing a thread ready to run
 - Thread has all of the resources needed to execute
 - Thread scheduler is aware of Thread
- Runnable is result of state transition:
 - From New to Runnable via call on new thread
`t.start();`
 - From Blocked to Runnable via scheduler
 - From Running to Runnable via pre-emption
- Will be Runnable at least one time



Lifecycle Phase: Running

- Phase describing a running thread
 - Execution of `run` method
 - Result of state transition from `Runnable`
 - Governed by thread scheduler
- Dependent upon pre-emptive nature
- May be time-sliced

Lifecycle Phase: Blocked / Waiting



- Phase describing a thread “on hold”
 - Execution of “run” method pauses
 - Pause could be temporary and indeterministic
 - Or pause could be temporary and deterministic
- Result of interacting with a blocked resource
- Transition out of blocked back to Runnable



Lifecycle Phase: Dead

- ◉ Phase representing a thread that has ceased execution
 - ◉ Can't return from dead
 - ◉ Dead doesn't mean object no longer exists
 - ◉ Means thread can't return to Runnable phase
- ◉ Result of state transition from Running
 - ◉ Caused by unhandled exception in run
 - ◉ Caused by completion of method body

Functions of the Thread Scheduler



- ◉ Built-into JVM
 - ◉ Responsible for managing lifecycle of threads
 - ◉ Entity that moves threads between transitions
 - ◉ Completely hidden from “code”
- ◉ Fully pre-emptive
 - ◉ May adopt time-slicing
 - ◉ May map threads to light-weight-processes in operating system
 - ◉ May be effected by underlying operating system

Thread Identity



- Threads are objects
 - ==, equals, and hashCode
 - toString
- Threads are java.lang.Thread
 - name – human readable name of Thread
 - Defaults to “Thread”
 - Can over-ride with call to Constructor
 - id – long id representing thread
 - Priority – pre-emptive scheduler
 - MAX_PRIORITY
 - DEFAULT_PRIORITY
 - MIN_PRIORITY

SimpleThread Identity



```
1 package labs.solutions.concurrent.basic;
2
3 +/** ...
4
5 public class SimpleThread extends Thread {
6
7     private long startTime;
8     private long runTime;
9
10
11    @Override
12    public void run() {
13        runTime = System.nanoTime();
14
15        System.out.printf("SimpleThread[%s:%d:%d].run(): run was called\n",
16                          getName(),getId(),getPriority());
17        System.out.printf("Elapsed time: %d nanoseconds\n", (runTime - startTime));
18    }
19
20
21    @Override
22    public void start() {
23        System.out.println("SimpleThread.start(): start was called");
24        startTime = System.nanoTime();
25        super.start();
26    }
27 }
```



Thread Identity [cont.]

- Threads belong to a ThreadGroup
 - ThreadGroup is a “group of threads”
 - Formed in hierarchies
 - Supports manipulation of all threads in ThreadGroup
 - Thread is created as part of its “Parent” ThreadGroup
- ThreadGroup has identity
 - ==, equals, and hashCode
 - toString
 - Name – getName
 - Hierarchy - getParent

ThreadGroup Example



```
1 package labs.solutions.concurrent.basic;
2
3 + /** ...
4
5 public class SimpleThread extends Thread {
6
7     private long startTime;
8     private long runTime;
9
10
11    @Override
12    public void run() {
13        runTime = System.nanoTime();
14
15        System.out.printf("SimpleThread[%s:%d:%d].run(): run was called\n",
16                          getName(),getId(),getPriority(), getThreadGroup().getName());
17        System.out.printf("Elapsed time: %d nanoseconds\n", (runTime - startTime));
18    }
19
20
21    @Override
22    public void start() {
23        System.out.println("SimpleThread.start(): start was called");
24        startTime = System.nanoTime();
25        super.start();
26    }
27}
```

LAB: Basic Thread Part 2



- In this lab, you will create three instances of the SimpleThread class, each with a with customized names.
- After creating three instances, start all three sequentially. Determine which Thread completes first.
- Run the lab multiple times, do you get the same result?
- Duration: 20 minutes

Basic Thread Lab 2 : Solution



```
1 package labs.solutions.concurrent.basic;
2
3 /**
4  * SimpleThread Example
5  */
6 public class SimpleThread extends Thread {
7     private long startTime;
8     private long runTime;
9
10    public SimpleThread() {
11        super();
12    }
13
14    public SimpleThread(String s) {
15        super(s);
16    }
17
18    @Override
19    public void run() {
20        runTime = System.nanoTime();
21
22        System.out.printf("SimpleThread[%s:%d:%d][%s].run(): run was called\n",
23                          getName(),getId(),getPriority(), getThreadGroup().getName());
24        System.out.printf("Elapsed time: %d nanoseconds\n", (runTime - startTime));
25    }
26
27    @Override
28    public void start() {
29        System.out.println("SimpleThread.start(): start was called");
30        startTime = System.nanoTime();
31        super.start();
32    }
33
34 }
```

```
1 package labs.solutions.concurrent.basic;
2
3 /**
4  * SimpleThread Example
5  */
6 public class SimpleThreadExample {
7     public static void main(String[] args) {
8         Thread t = new SimpleThread("one");
9         Thread t2 = new SimpleThread("two");
10        Thread t3 = new SimpleThread("three");
11        t.start();
12        t2.start();
13        t3.start();
14    }
15 }
```

Managing Threads

Basic lifecycle management





Manipulating the Lifecycle

- Basic APIs provide “control” over scheduler
 - Not precise - more suggestive
 - Moves thread into different phases of lifecycle
- Lifecycle nudging on Thread
 - sleep – pauses currently executing thread
 - yield – to another same priority thread
 - interrupt – interrupt a thread
 - setPriority – change priority
 - setDaemon – change type of thread
 - join – wait until thread dies
- Lifecycle nudging on ThreadGroup
 - interrupt
 - setMaxPriority
 - setDaemon



Determining Lifecycle Phase

- ◉ Prior to 1.5
 - ◉ No way to precisely determine phase of Thread
 - ◉ Could only determine:
 - ◉ “aliveness” - `isAlive`
 - ◉ “interruptedness” - `isInterrupted`
 - ◉ “activeness” - `currentThread`
 - ◉ Made it hard to monitor thread activities
- ◉ 1.5 defines 7 states for threads
 - ◉ Phases pretty much map to states
 - ◉ More specific than lifecycle phases
 - ◉ Represented in new enum `Thread.State`
 - ◉ Get state of thread by calling `getState`

1.5 States



Pre 1.5	1.5 +	Notes
“new”	ThreadState.NEW	
“runnable”	ThreadState.RUNNABLE	
“blocked”	ThreadState.BLOCKED	io
“blocked”	ThreadState.WAITING	waiting specifically for object lock
“blocked”	ThreadState.TIMED_WAITING	time-specific WAITING
“dead”	ThreadState.TERMINATED	
“running”	N/A	ironically, there is not way to determine if a thread is running



Managing Thread Transitions

- Thread API provides methods to “force” lifecycle transitions
 - Not as deliberate as might like
 - No setState call
 - Scheduler manages transitions
- Primarily give instructions on what to do
- Many transitions rely on Object locks



RUNNABLE Transitions

- Transition into RUNNABLE, from:
 - NEW
 - BLOCKED
 - WAITING
- Transition out of RUNNABLE into “running”
 - Based on pre-emptive / time-sliced rules
 - Can “force” this by changing a thread’s priority



BLOCKED Transitions

- ◉ Caused by mutexes
 - ◉ Transition from “running” to WAITING
 - ◉ Typically when object lock not available
 - ◉ Transition to RUNNABLE when lock is released
- ◉ Determined and managed by scheduler
- ◉ Potential for deadlock

WAITING Transitions



- ◉ Similar to BLOCKED transition
 - ◉ Transition into WAITING caused by `wait()` call
 - ◉ Transition out of WAITING caused by `notify()` or `notifyAll()` call
- ◉ More control over thread transitions
 - ◉ Manually tell scheduler to transition thread
 - ◉ Still need synchronized mechanisms
- ◉ TIMED_WAITING transitions are similar to WAITING
 - ◉ But are deterministic
 - ◉ Have “wait” duration - `wait(long timeout)`

LAB: Thread Control



- **Modify the SimpleThread and SimpleThreadExample.**

Implement the run method in such a way that the threads execute the run method 100 times. After each iteration of the run method, allow another thread to move into the “running” state.

- **Experiment with :**
 - Being mean – consuming the CPU
 - Yielding – either with yield or changing your priority
 - Sleeping
- **Duration:** 30 minutes

Mutual Exclusion





What is a Lock?

- ◉ Sometimes referred to as a *monitor lock*
 - ◉ Associated with / available for every object
 - ◉ Only one “lock” per object
 - ◉ Held by at most **one** thread at any given time
 - ◉ When lock is occupied, other threads are “on hold”
- ◉ Locking mechanisms inherited from
`java.lang.Object`
 - ◉ `wait`
 - ◉ `notify`
 - ◉ `notifyAll`



Creating Mutual Exclusion

- ◉ Mutual exclusion
 - ◉ Prevents simultaneous use of a common resource
 - ◉ Commonly referred to as a mutex
- ◉ Mutexes are structured around:
 - ◉ An object lock
 - ◉ A critical section
- ◉ Two mechanisms for defining mutexes:
 - ◉ synchronized methods
 - ◉ synchronized blocks



Mutex: Synchronized Methods

- ◉ Need mutual exclusion at method level
 - ◉ Entire method is considered critical section
 - ◉ Lock associated with object “containing” method
 - ◉ Lock obtained when method is invoked
 - ◉ Released when method “returns”
- ◉ Defined using synchronized
 - ◉ `public synchronized void put(Object o)`
 - ◉ `public synchronized Object get()`



Mutex: Synchronized Blocks

- ◉ Need mutual exclusion within a method
 - ◉ Block represents critical section containing critical code
 - ◉ Lock can be associated with:
 - ◉ Object containing method with synchronized block
 - ◉ Or monitor object
 - ◉ Lock behavior:
 - ◉ Lock obtained upon entering synchronized block
 - ◉ Released upon exit
- ◉ Defined using synchronized modifier on block:
 - ◉ `synchronized(this) { . . . }`
 - ◉ `synchronized(list) { . . . }`
 - ◉ `synchronized(MyClass.class) { . . . }`

Thread Communication





Purpose of Communication

- Communication – imparting or exchanging information between people, places, things (Objects)
- Communication in software – messaging passing to facilitate cooperation in completing task(s)



What hinders communication?

- ◉ As humans:

- ◉ As objects:

- ◉ As threads:

Types of Thread Communication



- ◉ Object Semantics
 - ◉ Threads talk directly to other threads as objects
 - ◉ When one thread completes its task, it updates state of another thread
 - ◉ What are the Design Implications?
- ◉ Thread Semantics
 - ◉ Threads belong to a ThreadGroup
 - ◉ Threads can talk to one another through the ThreadGroup
 - ◉ What are the Design Implications?



Thread Communication

- ◉ Can mutual exclusion help us with communication?
 - ◉ Does not provide cross-thread communication
 - ◉ Can cause
 - ◉ “blocked” starvation
 - ◉ Deadlock
- ◉ Commonly, need communication between threads
 - ◉ Need some form of traffic signal mechanism
 - ◉ Should offer alternative to starvation
 - ◉ May not prevent all dead-lock

Semaphores



- ◉ Semaphores – a system of sending messages
- ◉ Simplest semaphore is mutex
 - ◉ Create mutual exclusion
 - ◉ Ensure threads execute in specific order
- ◉ Utilizes communication to:
 - ◉ Notify threads when resource is free
 - ◉ Threads end up WAITING when resource is not free
- ◉ Semaphore implementations rely on:
 - ◉ Object locks
 - ◉ Synchronization
 - ◉ `wait`, `notify`, and `notifyAll`

Basic Concurrency Example



Producer – Consumer is classic threading problem

- Producer produces something
- Consumer consumes something
- Consumer can only consume when things are available
- Producer and Consumer share the supply chain

Example: Shared Resource [the order board]



```
1 package examples.concurrent.basic;
2
3 +import ...
5
6 +/** ...
14 public class OrderBoard {
15
16     private List<Order> orders;
17
18     +/** ...
22     public OrderBoard() {
23         orders = new ArrayList<Order>();
24     }
25 }
```

Example: Shared Resource [cont.]



```
26  public void postOrder(Order toBeProcessed) {  
27      synchronized(orders) {  
28          while(orders.size() == 5) {  
29              try {  
30                  orders.wait();  
31              } catch (InterruptedException e) {  
32                  e.printStackTrace();  
33              }  
34          }  
35          orders.notifyAll();  
36          orders.add(toBeProcessed);  
37      }  
38  }  
39  
40  }  
41 }  
42 }  
43 }
```

Example: Shared Resource [cont.]



```
45  public Order cookOrder() {  
46      Order tmpOrder = null;  
47  
48      synchronized(orders) {  
49          while(orders.isEmpty()) {  
50              try {  
51                  orders.wait();  
52              } catch (InterruptedException e) {  
53                  e.printStackTrace();  
54              }  
55          }  
56  
57          tmpOrder = orders.remove(0);  
58          if(orders.size() < 3) {  
59              orders.notify();  
60          }  
61      }  
62  
63      return tmpOrder;  
64  }  
65 }
```

Example: Shared Resource [the order]



```
1 package examples.concurrent.basic;
2
3 /**
4  * public class Order {
5
6     private static int TAB_NUMBER = 0;
7     private int orderNumber;
8     private String menuItem;
9
10    /**
11     * public Order() {
12         orderNumber = ++TAB_NUMBER;
13     }
14
15    /**
16     * public String getMenuItem() {...}
17
18    /**
19     * public int getOrderNumber() {...}
20
21    /**
22     * public void setMenuItem(String menuItem) {...}
23
24 }
```

Example: Producer [the waiter]



```
1 package examples.concurrent.basic;
2
3 /* ... */
4 public class Waiter implements Runnable {
5
6     private OrderBoard ordersToServe;
7
8     public Waiter(OrderBoard orders) {
9         ordersToServe = orders;
10    }
11
12    public void run() {
13        while(true) {
14            Order newOrder = new Order();
15            if(newOrder.getOrderNumber() % 2 == 0) {
16                newOrder.setMenuItem("Hamburger");
17            } else {
18                newOrder.setMenuItem("Cheeseburger");
19            }
20
21            ordersToServe.postOrder(newOrder);
22            System.out.printf("Order IN [%d]: %s\n",
23                             newOrder.getOrderNumber(), newOrder.getMenuItem());
24        }
25    }
26
27 }
28 }
```

Example: Consumer [the cook]



```
1 package examples.concurrent.basic;
2
3 /**
4  * ...
5  */
6
7 public class Cook implements Runnable {
8
9     private OrderBoard ordersToCook;
10
11    public Cook(OrderBoard orders) {
12        ordersToCook = orders;
13    }
14
15    public void run() {
16        while(true) {
17            Order tmpOrder = ordersToCook.cookOrder();
18            try {
19                Thread.sleep(500);
20            } catch (InterruptedException e) {
21                e.printStackTrace();
22            } finally {
23                System.out.printf("Order up [%d]: %s\n",
24                                  tmpOrder.getOrderNumber(), tmpOrder.getMenuItem());
25            }
26        }
27    }
28 }
```



Example: Application [the joint]

```
1 package examples.concurrent.basic;
2
3 /**
4  * ...
5  */
6 public class TheBurgerJoint {
7
8     public static void main(String[] args) {
9         OrderBoard orders = new OrderBoard();
10
11         Runnable cook = new Cook(orders);
12         Runnable waiter1 = new Waiter(orders);
13         Thread producer = new Thread(waiter1);
14         Thread consumer = new Thread(cook);
15         producer.start();
16         consumer.start();
17     }
18 }
19 }
```

Concurrency Lab



- Description: The Burger Joint has been unionized. New union rules mandate that a cook can only cook 300 burgers in one shift. The Burger Joint needs to serve a continuous stream of burgers. Modify the example to support the union rules without sacrificing any burger cooking downtime.
- Duration: 30 minutes

Summary



- Java is multi-threaded
- Thread represents a unit of execution
- A thread can transition through 7 lifecycle phases
- Waiting and blocking phases are governed by object locks
- Threads can't be risen from the dead