

Java Threads

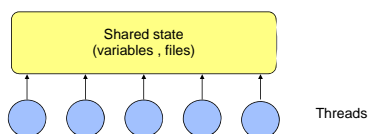


Why Threads ?

- Imagine a stock-broker application with a lot of complex capabilities:
 - download last stock option prices
 - check prices for warnings
 - analyze historical data for company XYZ



What Are Threads?



- General-purpose solution for managing concurrency.
- Multiple independent execution streams
- Shared state



Single-threaded scenario

- In a single-threaded runtime environment, these actions execute one after another
 - The next action can happen *only when the previous one is finished*.
- If a historical analysis takes half an hour, and the user selects to perform a download and check afterward...
 - ...the result may come too late to buy or sell stock as a result.



What Are Threads Used For?

- **Operating systems:** one kernel thread for each user process.
- **Scientific applications:** one thread per CPU (solve problems more quickly).
- **Distributed systems:** process requests concurrently (overlap I/Os).
- **GUIs:**
 - Threads correspond to user actions; they can help display during long-running computations.
 - Multimedia, animations.

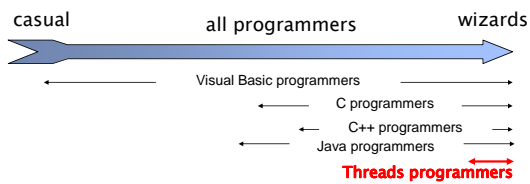


Multi-threaded scenario

- Multithreading can really help
 - The download should happen in the background (i.e. in another thread).
 - Other processes could happen at the same time e.g. a warning could be communicated instantly. All the while, the user is interacting with other parts of the application.
 - The analysis, too, could happen in a separate thread, so the user can work in the rest of the application while the results are being calculated.



What's Wrong With Threads?



- Too hard for most programmers to use
- Even for experts, development is painful
- Threads break abstraction: can't design modules independently.

SoftEng

Multitasking

- For User, multitasking is the ability to have several applications open and working at the same time.
 - A user can edit a file with one application while another application is printing or recalculating a spreadsheet.
- For Developer, multitasking is the ability to create applications which create processes that use more than one thread of execution
 - one thread that handles interactions with the user (keyboard and mouse input), while
 - another threads with lower priority perform the other work of the process.

SoftEng

Process

- From an Operating System viewpoint, a **Process** is an instance of a running application
- A Process has its own private virtual address space, code, data, and other O.S. resources like opened files, etc..
- A process also contains one or more threads that run in the context of the process.

Multitasking

- A multitasking operating system assigns CPU time (*slices*) to threads
- O.S. is *preemptive*, if a thread is executed until
 - time slice is over or it ends its execution;
 - It blocks (synchronization with threads or resources)
 - another thread acquires more priority
- Using small time-slice (e.g. 20 ms) thread execution seems to be parallel
 - which can be actually parallel in multiprocessor systems

SoftEng

Thread

- A thread is the basic entity to which the operating system allocates CPU time.
- A thread can execute any part of the application's code, including a part currently being executed by another thread.
- All threads of a process share the virtual address space, global variables, and operating system resources of the process.

Multitasking Problems

- O.S. consumes memory for the structures required by both processes and threads.
 - Keeping track of a large number of threads also consumes CPU time.
- Multiple threads accessing the same resources must be synchronized to avoid conflicts, or can lead to problems such as **deadlock** and **race conditions** :
 - System resources (communications ports, disk drives),
 - Handles to resources shared by multiple processes (files)
 - Resources of a process (variables used by multiple threads)

SoftEng

JVM and Operating System

- Code running concurrently given that there's only one CPU on most of the machines running Java.
- The JVM gets its turn at the CPU by whatever scheduling mechanism the OS uses
- JVM operates like a mini-OS and schedules its own threads regardless of the underlying operating system.
- In some JVMs, the Java threads are actually mapped to native OS threads.

SoftEng

Create a Thread

- Threads can be created by extending Thread and overriding the `run()` method.
- Thread objects can also be created by calling the Thread constructor that takes a **Runnable** argument (the target of the thread)
- It is legal to create many Thread objects using the same Runnable object as the *target*.

SoftEng

JVM and Operating System

- Do not interpret the behavior on one machine as "the way threads work"
- Design a program so that it will work regardless of the underlying JVM.
- Thread motto:

**When it comes to threads,
very little is guaranteed**

SoftEng

Create a Thread

1. Extends Thread class
 - class X extends **Thread** {}
 - t = new X(); t.start(); // Create and start
2. Implementing **Runnable** interface (better)

```
class Y implements Runnable {  
    public void run() { //code here }  
}
```

Thread r = new Thread (new Y);
r.start(); //invoke run() & create a new call-stack

SoftEng

JVM Scheduler

- The Scheduler is the JVM part that decides
 - which thread should run at any given moment,
 - takes threads out of the running state.
 - Some JVMs use O.S. scheduler (**native threads**)
- Assuming a single processor machine:
 - Only one thread can actually run at a time.
 - Only one stack can ever be executing at one time
- The order in which runnable threads are chosen to be THE ONE running is **NOT** guaranteed.

SoftEng

Example: extends Thread

- Write two threads, each counting till X

```
class Counter extends Thread {  
    private int num; String name;  
    public Counter(String nn, int n) {  
        name= nn; num = n;    }  
    public void run(){  
        for( int i=0; i<num; ++i)  
            System.out.print(name+": "+i + " ");  
    }  
}
```

SoftEng

Example: implements Runnable

```
class Counter2 implements Runnable {
    private int num;
    public Counter2(int n) { num = n; }
    public void run(){
        for(int i=0; i<num; ++i)
            System.out.print( i+" ");
    }
}

public static void main(String args[] ) {
    Thread t1,t2;
    t1 = new Thread(new Counter2(10));
    t2 = new Thread(new Counter2(5));
    t1.start();
    t2.start();
}
```

SoftEng

Running Multiple Threads

```
class NameRunnable implements Runnable {
    public void run() {
        for (int x = 1; x <= 3; x++){
            System.out.println("Run by "+ Thread.currentThread().getName()+ ", x is "+ x);
        }
    }
}

public class ManyNames {
    public static void main(String [] args) { // Make one Runnable
        NameRunnable nr = new NameRunnable();
        Thread one = new Thread(nr);   Thread two = new Thread(nr);
        Thread three = new Thread(nr);
        one.setName("Paolo");   two.setName("Marco");
        three.setName("Maurizio");
        one.start(); two.start(); three.start();
    }
}
```

Output is
Non-Deterministic

SoftEng

Start a Thread

- When a Thread object is created, it does not become a thread of execution until its **start()** method is invoked.
- When a Thread object exists but hasn't been started, it is in the new state and is not considered alive.
- Method start() can be called on a Thread object only once.
- If start() is called more than once on same object, it will throw a RuntimeException

SoftEng

Running Multiple Threads

- Note Well: It is not guaranteed that threads will start running in the order they were started
- It is not guaranteed that a thread keeps executing until it's done.
- It is not guaranteed that a loop completes before another thread begins
- Nothing is guaranteed except:
 - Each thread will start, and each thread will run to completion, hopefully.**

SoftEng

Starting a Thread

```
public static void main(String [] args) {
    // running
    // some code
    // in main()
    method1();
    // running
    // more code
}

void method1() {
    Runnable r = new MyRunnable();
    Thread t = new Thread(r);
    t.start();
    // do more stuff
}
```

1) main() begins

2) main() invokes method1()

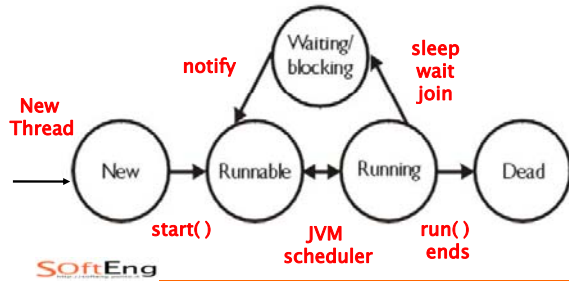
3) method1() starts a new thread

SoftEng

Java Thread States

SoftEng

Java Thread States



Thread state: Blocked



- This is the state a thread is in when it's NOT eligible to run.
 - It might return to a runnable state later if a particular event occurs.
- A thread may be blocked waiting for a resource (I/O or an object's lock)
 - e.g. if data comes in through the input stream the thread code is reading from,
 - the object's lock suddenly becomes available.

SoftEng

Thread state: Running



- This is the state a thread is in when the thread scheduler selects it (from the runnable pool) to be the currently executing process.
- A thread can transition out of a running state for several reasons, including because "the thread scheduler decided it"
- Only one way to get to the running state: the scheduler chooses a thread from the runnable pool.

SoftEng

Thread state: Sleeping



- A thread may be sleeping because the thread's run() code tells it to sleep for some period of time,
- Back to Runnable state when it wakes up because its sleep time has expired.

```
try {
    Thread.sleep(5*60*1000); // Sleep for 5 min
} catch (InterruptedException ex) { }
```

SoftEng

Thread state: Runnable



- A thread is queued & eligible to run, but the scheduler has not selected it to be the running thread
- A thread first enters the runnable state when the start() method is invoked
- A thread can also return to the runnable state after either running or coming back from a blocked, waiting, or sleeping state
- When the thread is in the runnable state, it is considered alive

Example sleep

This code outputs Homer, Lisa, Bart alternating nicely

```
class NameRunnable implements Runnable {
    public void run() {
        for (int x = 1; x < 4; x++) {
            System.out.println("Run by"+Thread.currentThread().getName());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) { }
        }
    }
}
```

```
public class ManyNames {
    public static void main (String [] args) {
        NameRunnable nr = new NameRunnable();
        Thread one = new Thread(nr); one.setName("Homer");
        Thread two = new Thread(nr); two.setName("Lisa");
        Thread three = new Thread(nr); three.setName("Bart");
        one.start(); two.start(); three.start();
    }
}
```

SoftEng

Thread state: Waiting



- A thread run code causes it to wait
- It come back to Runnable state when another thread sends a notification
- Used for threads interaction
- *Note Well:* one thread does not tell another thread to block.

SoftEng

JVM Scheduling Policy

JVM Scheduler policy can be:

- **non-preemptive:** current thread is executed until the end, unless thread explicitly releases CPU to let another thread take its turn
 - used in real-time apps (interruption can cause problems)
- **preemptive time-slicing:** thread is executed until its time-slice is over, then the JVM suspends it and starts another runnable thread
 - Simpler development, as all resources handled by JVM
 - Apps do not require to use `yield()` to release resources

High priority threads:

- Are executed more often, or have longer time-slice
- Stop execution of lower-priority threads before their time-slice is over

Join

- The `join()` method lets one thread "join onto the end" of another thread.

```
Thread t = new Thread();
t.start(); t.join();
```
- Current thread move to Waiting state and it will be Runnable when thread `t` is dead
- A timeout can be set to wait for a thread's end

```
t.join( 5000); // wait 't' for 5 seconds: if 't' is not
// finished, then current thread is Runnable again
```

SoftEng

Setting a Thread's Priority

- By default, a thread gets the priority of the thread of execution that creates it.
- Priority values are defined between 1 and 10

```
Thread.MIN_PRIORITY    (1)
Thread.NORM_PRIORITY   (5)
Thread.MAX_PRIORITY    (10)
```
- Priority can be directly set

```
FooRunnable r = new FooRunnable();
Thread t = new Thread(r);
t.setPriority(8); t.start();
```

SoftEng

Thread Priorities

- A thread always runs with a priority number
- The scheduler in most JVMs uses preemptive, priority-based scheduling
- Usually **time-slicing** is used:
 - each thread is allocated a fair amount of time
 - After that a thread is sent back to runnable to give another thread a chance
- JVM specification does not require a VM to implement a time-slicing scheduler !!!
 - some JVM may use a scheduler that lets one thread stay running until the thread completes its `run()` method

SoftEng

yield

- The method `yield()` make the currently running thread back to Runnable state
 - It allows other threads of the same priority to get their turn
- `yield()` will cause a thread to go from running to runnable, but it might have no effect at all
 - There's no guarantee the yielding thread won't just be chosen again over all the others!

SoftEng

Example: Checking JVM Scheduler type

- This code can be used to check if scheduler is preemptive or not:

```
public class Hamlet implements Runnable {
    public void run() {
        while (true)
            System.out.println(Thread.currentThread().getName());
    }
}
public class TryHamlet {
    public static void main(String argv[]) {
        Hamlet aRP = new Hamlet ();
        new Thread(aRP, "To be ").start();
        new Thread(aRP, "Not to be").start();
    }
}
```

- If non-preemptive the thread chosen first run forever and it never releases CPU
- If preemptive threads randomly alternate on output

Move a Thread out from Running state

There are 4 cases when JVM scheduler does it:

- The thread's run() method completes
- Thread calls wait() on an object
- A thread can't acquire the *lock on the object*
- The thread scheduler can decide to move the current thread from running to runnable in order to give another thread a chance to run.

SoftEng

Example with yield()

- Code is less dependent from the scheduler type, because each thread releases CPU after one iteration:

```
public class Hamlet implements Runnable {
    public void run() {
        while (true) {
            System.out.println(Thread.currentThread().getName());
            Thread.yield(); // allow other thread to run
        }
    }
}
```

Output:

```
To be
Not to be
To be
Not to be
To be
Not to be
To be
Not to be
...
```

SoftEng

A word of advice

- Some methods may look like they tell another thread to block, but they don't.
- If *t* is a thread object reference, you can write something like this:
t.sleep() or t.yield()
- They are static methods of the Thread class:
 - they don't affect the instance t !!!
 - instead they affect the thread in execution
 - That's why it's a bad idea to use an instance variable to access a static method

SoftEng

Leaving the Running state

There are 3 ways for a thread to do it:

- **sleep()**: guaranteed to cause the current thread to stop executing for **at least** the specified sleep duration
- **yield()**: the currently running thread moves back to runnable, to give room to other threads with same priority
- **join()**: stop executing until the thread it joins with completes

SoftEng

Synchronization in Java

SoftEng
http://www.softeng.com.ua

Example scenario



- What happens when two different threads are accessing the same data ?
- Imagine that two people each have ATM cards, but both cards are linked to only one account.



```
class Account {
    private int balance = 50;
    public int getBalance() {
        return balance;
    }
    public void withdraw(int amount) {
        balance = balance - amount;
    }
}
```



SoftEng

Example: code

```
public class AccountDanger implements Runnable {
    private Account account = new Account();
    public static void main (String [] args) {
        AccountDanger r = new AccountDanger();
        Thread one = new Thread(r); Thread two = new Thread(r);
        one.setName("Homer"); two.setName("Marge");
        one.start(); two.start();
    }
    public void run() {
        for (int x = 0; x < 5; x++) {
            makeWithdraw(10);
            if (account.getBalance() < 0)
                System.out.println("account is overdrawn!");
        }
    }
}
```

Example scenario (II)



- Before one of them makes a withdrawal, first check the balance to be certain there's enough to cover the withdrawal
- Check the balance.
 - If there's enough in the account (in this example, at least 10), make the withdrawal
- What happens if something separates step 1 from step 2 ?

SoftEng

```
private void makeWithdrawal(int amount) {
    if (account.getBalance() >= amount) {
        System.out.println(Thread.currentThread().getName() +
            " is going to withdraw");
        try {
            Thread.sleep(500);
        } catch (InterruptedException ex) {}
        account.withdraw(amt);
        System.out.println(Thread.currentThread().getName()+
            " completes the withdrawal");
    } else {
        System.out.println("Not enough in account for "+
            Thread.currentThread().getName()+ "to withdraw "+
            account.getBalance());
    }
}
```

Example scenario (III)

- Marge checks the balance and there is enough (10)
- Before she withdraws money, Homer checks the balance and also sees that there's enough for his withdrawal.
- He is seeing the account balance before Marge actually debits the account...
- Both Marge and Homer believe there's enough to make their withdrawals !
- If Marge makes her withdrawal...
- ...there isn't enough in the account for Homer's withdrawal
- ... but he thinks there is since when he checked, there was enough!



SoftEng

Example: a possible output

- Homer is going to withdraw
- Marge is going to withdraw
- Homer completes the withdrawal
- Homer is going to withdraw
- Marge completes the withdrawal
- Marge is going to withdraw
- Homer completes the withdrawal
- Homer is going to withdraw
- Marge completes the withdrawal
- Marge is going to withdraw
- Homer completes the withdrawal

5: Marge completes her withdrawal and then before Homer completes his, Marge does another check on the account on line 6. And so it continues until line 8, where Homer checks the balance and sees that it's 20.

9: Marge completes a withdrawal, and now balance is 10.

10: Marge checks again, sees that the balance is 10, so she knows she can do a withdrawal. *But she didn't know that Homer, too, has already checked the balance on line 8 so he thinks it's safe to do the withdrawal!*

11: Homer completes the withdrawal he approved on line 8. This takes the balance to zero. But Marge still has a pending withdrawal that she got approval for on line 10!

Example: a possible output (II)

12. Not enough in account for Homer to withdraw 0
 13. Not enough in account for Homer to withdraw 0
 14. Marge completes the withdrawal
 15. account is overdrawn!
 16. Not enough in account for Marge to withdraw -10
 17. account is overdrawn!
 18. Not enough in account for Marge to withdraw -10
 19. account is overdrawn!

12-13: Homer checks the balance and finds that there's not enough in the account

14: Marge completes her withdrawal and D'Oh!
 The account is now overdrawn by 10 !!!!!

This is an example of Race Condition

SoftEng

Preventing Race Conditions

- You can't guarantee that a single thread will stay running during the atomic operation.
- But even if the thread running the atomic operation moves in and out of the running state, no other running thread will be able to act on the same data.
- How to protect the data:
 - Mark the variables private.
 - Synchronize the code that modifies the variables.

SoftEng

Race Condition

A problem happening whenever:

- Many threads can access the same resource (typically an object's instance variables)
- This can produce corrupted data if one thread "races in" too quickly before an operation has completed.

SoftEng

Synchronization in Java

- The modifier **synchronized**
 - can be applied to a method or a code block
 - locks a code block: ONLY ONE thread can access

private **synchronized** void makeWithdrawal(int amount)

- Adding a word and example problem is solved!

Homer is going to withdraw
 Homer completes the withdrawal
 Marge is going to withdraw
 Marge completes the withdrawal
 Homer is going to withdraw
 Homer completes the withdrawal
 Marge is going to withdraw
 Marge completes the withdrawal

Homer is going to withdraw
 Homer completes the withdrawal
 Not enough in account for Marge to withdraw 0
 Not enough in account for Homer to withdraw 0
 Not enough in account for Marge to withdraw 0
 Not enough in account for Homer to withdraw 0
 Not enough in account for Marge to withdraw 0

Example: Preventing Race Conditions

- We must guarantee that the two steps of the withdrawal are NEVER split apart.
- It must be an **atomic operation**:
 - It is completed before any other thread code that acts on the same data
 - ...regardless of the number of actual instructions

SoftEng

Synchronization and Locks



- Every object in Java has one built-in lock
- Enter a synchronized non-static method => get the lock of the current object code we're executing.
- If one thread got the lock, other threads have to **wait** to enter the synchronized code until the lock has been released (thread exits the synch. method)
- Not all methods in a class need to be synchronized.
- Once a thread gets the lock on an object, no other thread can enter **ANY** of the synchronized methods in that class (for that object).

SoftEng

Synchronization and Locks

- Multiple threads can still access the class's non-synchronized methods
 - Methods that don't access the data to be protected, don't need to be synchronized
- Thread going to sleep, doesn't release locks
- A thread can acquire more than one lock, e.g.
 - a thread can enter a synchronized method
 - then immediately invoke a synchronized method on another object

SoftEng

When Do I Need To Synchronize?

- Two threads executing the same method at the same time may:
 - use different copies of local vars => no problem
 - access fields that contain shared data
- To make a thread-safe class:
 - methods that access changeable fields need to be synchronized.
 - Access to static fields should be done from static synchronized methods.
 - Access to non-static fields should be done from non-static synchronized methods

Synchronize a code block

```
public synchronized void doStuff() {  
    System.out.println("synchronized");  
}
```

Is equivalent to this:

```
public void doStuff() {  
    synchronized(this) {  
        System.out.println("synchronized");  
    }  
}
```

SoftEng

Example

```
public class NameList {  
    private List names =  
        Collections.synchronizedList(new LinkedList());  
    public void add(String name) {  
        names.add(name);  
    }  
    public String removeFirst() {  
        if (names.size() > 0)  
            return (String) names.remove(0);  
        else return null;  
    }  
}
```

Returns a List whose methods are all synchronized and "thread-safe"

Can the NameList class be used safely from multiple threads?

SoftEng

Synchronize a static method

```
public static synchronized int getCount() {  
    return count;  
}
```

Is equivalent to this:

```
public static int getCount() {  
    synchronized(MyClass.class) {  
        return count;  
    }  
}
```

MyClass.class represents the single lock on the class which is different from the objects' locks

SoftEng

Example (II)

```
class NameDropper extends Thread {  
    public void run() {  
        String name = nl.removeFirst(); System.out.println(name);  
    }  
    public static void main(String[] args) {  
        final NameList nl = new NameList(); nl.add("Jacob");  
        Thread t1 = new NameDropper(); t1.start();  
        Thread t2 = new NameDropper(); t2.start();  
    }  
}
```

Thread t1 executes names.size(), which returns 1.
Thread t2 executes names.size(), which returns 1.
Thread t1 executes names.remove(0), which returns Jacob.
Thread t2 executes names.remove(0), which throws an exception because the list is now empty.

SoftEng

Example (III)

- In a "thread-safe" class each individual method is synchronized.
- But nothing prevents another thread from doing something else to the list *in between those two calls*
- Solution: **synchronize the code yourself !**

```
public class NameList {
    private List names = new LinkedList();
    public synchronized void add(String name) {
        names.add(name);
    }
    public synchronized String removeFirst() {
        if (names.size() > 0)
            return (String) names.remove(0);
        else return null;
    }
}
```

SoftEng

Thread Interactions



Deadlock

- Deadlock occurs when two threads are blocked, with each waiting for the other's lock.
- ⇒ Neither can run until the other gives up its lock, so they wait forever
- Poor design can lead to deadlock
 - It is hard to debug code to avoid deadlock

SoftEng

Synchronization in Object class

- **void wait()**
 - Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
- **void notify()**
 - Wakes up a single thread that is waiting on this object's lock.
- **void notifyAll()**
 - Wakes up all threads that are waiting on this object's lock.

SoftEng

Thread Deadlock

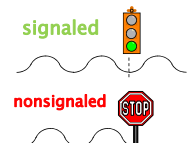
```
public class DeadlockRisk {
    private static class Resource { public int value; }
    private Resource resourceA = new Resource();
    private Resource resourceB = new Resource();
    public int read() {
        synchronized(resourceA) { // May deadlock here !
            synchronized(resourceB) {
                return resourceB.value + resourceA.value;
            }
        }
    }
    public void write(int a, int b) {
        synchronized(resourceB) { // May deadlock here !
            synchronized(resourceA) {
                resourceA.value = a;
                resourceB.value = b;
            }
        }
    }
}
```

- The reader thread will have resourceA,
- the writer thread will have resourceB,
- ... and both will get stuck waiting for the other !!!

SoftEng

Wait

- The wait() method lets a thread say: "There's nothing for me to do now, so put me in your waiting pool and notify me when something happens that I care about."
- An object lock can be **signaled** or **nonsignaled**
- When calling wait() :
 - On a **signaled** object lock thread keeps executing
 - On a **nonsignaled** object lock thread is suspended



SoftEng

Notify & NotifyAll

- The **notify()** method send a signal to one of the threads that are waiting in the same object's waiting pool.
- The notify() method CANNOT specify which waiting thread to notify.
- The method **notifyAll()** is similar but only it sends the signal to all of the threads waiting on the object.

SoftEng

Example: Java FIFO

```
import java.util.ArrayList;

public class FIFO{
    private ArrayList v;

    FIFO() {
        v = new ArrayList(3);
    }
    public synchronized void
    insert(Object o) {
        v.addElement(o);
        notify();
    }
}

public synchronized
Object extract()
throws Exception {
    Object temp;
    if (v.size()==0)
        wait();
    temp=v.get(0);
    v.remove(0);
    return(temp);
}
```

SoftEng

Mutual Exclusion

- A thread invokes wait() or notify() on a particular object, and the thread **must** currently **hold the lock** on that object
 - Called from within a synchronized context
- A thread owns in mutual exclusion a critical region when he has called wait() and it has not released the object yet (calling notify)
- A critical region is **nonsignaled** when is owned by a thread, **signaled** otherwise.

SoftEng

Example with multiple readers

```
class Reader extends Thread {
    Calculator c;
    public Reader(Calculator calc) {
        c = calc;
    }
    public void run() {
        synchronized(c) {
            try {
                System.out.println("Waiting for
                calculation...");
                c.wait();
            } catch (InterruptedException e) {}
            System.out.println("Total is: " +
                c.total);
        }
    }
}

public static void main(String [] args) {
    Calculator calculator = new Calculator();
    new Reader(calculator).start();
    new Reader(calculator).start();
    new Reader(calculator).start();
    calculator.start();
}

class Calculator extends Thread {
    int total;
    public void run() {
        synchronized(this) {
            for(int i=0;i<100;i++) {
                total += i;
            }
            notifyAll();
        }
    }
}
```

Example

```
class ThreadA {
    public static void main(String [] args) {
        ThreadB b = new ThreadB(); b.start();
        synchronized(b) {
            try {
                System.out.println("Waiting for b to complete");
                b.wait();
            } catch (InterruptedException e) {}
            System.out.println("Total is: "
                + b.total);
        }
    }
}

class ThreadB extends Thread {
    int total;
    public void run() {
        synchronized(this) {
            for(int i=0;i<100;i++)
                total += i;
            notify();
        }
    }
}
```

SoftEng

Example

```
class Machine extends Thread {
    Operator operator; // initialized
    public void run(){
        while(true){
            synchronized(operator){
                try {
                    operator.wait();
                } catch(InterruptedException ie) {}
            }
            // Send machine steps to hardware
        }
    }
}

class Operator extends Thread {
    public void run(){
        while(true){
            // Get shape from user
            synchronized(this){
                // Calculate new steps from shape
                notify();
            }
        }
    }
}
```

ONLY when the operator thread exits from the synchronized block => the hardware thread can start processing the machine steps.

SoftEng

Comments to Examples

- Ok if the waiting threads have called wait() before the other thread executes the notify()
- But what happens if, e.g., the Calculator runs first and calls notify() before the Readers have started waiting?
 - ...the waiting Readers will keep waiting forever !!
- Better use a loop checking a conditional expression
 - and only waits if the thing you're waiting for has not yet happened

SoftEng

Livelock

- A livelock happens when threads are actually running, but no work gets done
 - what is done by a thread is undone by another
- Ex: each thread already holds one object and needs another that is held by the other thread.
- What if each thread unlocks the object it owns and picks up the object unlocked by the other thread ?
 - These two threads can run forever in lock-step!

SoftEng

Better Solution

```
class Machine extends Thread {
    List<Instructions> jobs =new ArrayList<Instructions>();
    public void addJob(Instructions job) {
        synchronized (jobs) {
            jobs.add(job); jobs.notify();
        }
    }
    public void run(){
        while(true){
            synchronized (jobs){
                // wait until at least one job is available
                while (jobs.isEmpty()) {
                    try {
                        jobs.wait();
                    } catch (InterruptedException ie) {}
                }
            }
            // If we get here, we know that jobs is not empty
            Instructions instructions = jobs.remove(0);
            // Send machine steps to hardware
        }
    }
}

class Operator extends Thread {
    Machine machine; //initialized
    public void run(){
        while(true){
            // Get shape from user
            synchronized(this){
                // Calculate new steps from shape
                machine.addJob(job);
            }
        }
    }
}
```

Thread Starvation

- Wait/notify primitives of the Java language do not guarantee **liveness** (= > starvation)
- When wait() method is called
 - thread releases the object lock prior to commencing to wait
 - and it must be reacquired before returning from the method, post notification

SoftEng

Spontaneous Wakeup

- A thread may wake up even though no code has called notify() or notifyAll()
 - Sometimes the JVM may call notify() for reasons of its own,
 - Other class calls it for reasons you just don't know.
- When your thread wakes up from a wait(), you don't know for sure why it was awakened !
- **Solution:** putting the wait() method in a while loop and re-checking the condition:
 - We ensure that *whatever the reason we woke up, we will re-enter the wait() only if the thing we were waiting for has not happened yet.*

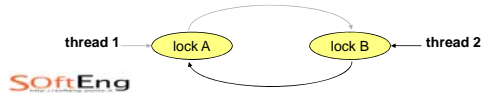
SoftEng

Thread Starvation

- Once a thread releases the lock on an object (following the call to wait), it is placed in a object's **wait-set**
 - Implemented as a queue by most JVMs
 - When a notification happens, a new thread will be placed at the back of the queue
- By the time the notified thread actually gets the monitor, the condition for which it was notified may no longer be true ...
 - It will have to wait again
 - This can continue indefinitely => **Starvation**

Recap: Threads Are Hard

- Synchronization:
 - Must coordinate access to shared data with locks.
 - Forget a lock? Corrupted data.
- Deadlock:
 - Circular dependencies among locks.
 - Each process waits for some other process: system hangs.



Info on threads: applet ThreadListener

```
for(int i = 0; i < num_threads; i++)
    print_thread_info(out, threads[ i ], indent + " ");
for(int i = 0; i < num_groups; i++)
    list_group(out, groups[i], indent + " ");
}
// Find the root thread group and list it recursively
public static void listAllThreads(PrintStream out) {
    ThreadGroup current_thread_group;
    ThreadGroup root_thread_group;
    ThreadGroup parent;
    // Get the current thread group
    current_thread_group = Thread.currentThread().getThreadGroup();
    // Now go find the root thread group
    root_thread_group = current_thread_group;
    parent = root_thread_group.getParent();
    while(parent != null) {
        root_thread_group = parent;
        parent = parent.getParent();
    }
    // And list it, recursively
    list_group(out, root_thread_group, "");
}
```

Recap: Threads Are Hard

- Achieving good performance is hard:
 - Simple locking yields low concurrency.
 - Fine-grained locking increases complexity
 - O.S. limits performance (context switches)
- Threads not well supported:
 - Hard to port threaded code (PCs? Macs?).
 - Standard libraries not thread-safe.
- Hard to debug :
 - data dependencies
 - timing dependencies
 - Few debugging tools

Info on Thread: applet ThreadListener

```
import java.applet.*;
import java.awt.*; import java.io.*;
public class AppletThreadListener extends Applet {
    TextArea textarea;
    // Create a text area to put our listing in
    public void init() {
        textarea = new TextArea(20, 60);
        this.add(textarea);
        Dimension prefsiz = textarea.preferredSize();
        this.resize(prefsiz.width, prefsiz.height);
    }
    // Do the listing. Note the cool use of ByteArrayOutputStream.
    public void start() {
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        PrintStream ps = new PrintStream(os);
        ThreadListener.listAllThreads(ps);
        textarea.setText(os.toString());
    }
}
```

Info on threads: applet ThreadListener

```
import java.io.*;
public class ThreadListener {
    // Display info about a thread.
    private static void print_thread_info(PrintStream out, Thread t, String indent) {
        if (t == null) return;
        out.println(indent + "Thread: " + t.getName() + " Priority: " + t.getPriority() +
            (t.isDaemon()? " Daemon "; " ") +
            (t.isAlive()? " " : " Not Alive"));
    }
    // Display info about a thread group and its threads and groups
    private static void list_group(PrintStream out, ThreadGroup g, String indent) {
        if (g == null) return;
        int num_threads = g.activeCount();
        int num_groups = g.activeGroupCount();
        Thread[] threads = new Thread[num_threads];
        ThreadGroup[] groups = new ThreadGroup[num_groups];
        g.enumerate(threads, false);
        g.enumerate(groups, false);
        out.println(indent + "Thread Group: " + g.getName() +
            " Max Priority: " + g.getMaxPriority() +
            (g.isDaemon()? " Daemon "; " "));
    }
}
```