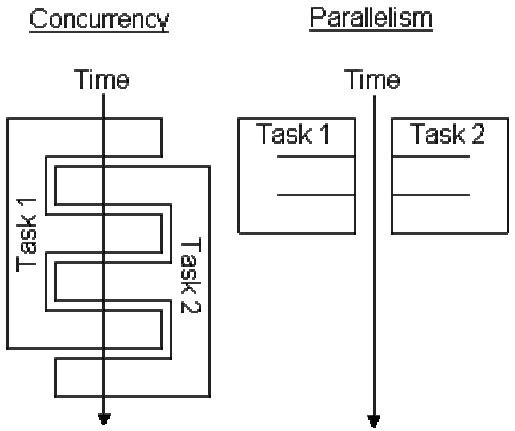
POLITECNICO DI MILANO
Piazza Leonardo da Vinci, 32 - 20133 Milano
Tel. +39.02.2399.1 - http://www.polimi.it

Concurrent Programming in Java

Paolo Costa
Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy
paolo.costa@polimi.it

Concurrency

- Concurrency is an important area of computer science studied in different contexts: machine architectures, operating systems, distributed systems, database, etc
- Objects provide a way to divide a program into independent sections. Often, you also need to turn a program into separate, independently running subtasks
- Concurrency may be *physical* (parallelism) if each unit is executed on a dedicated processor or *logical* if the CPU is able to switch from one to another so that all units appear to progress simultaneously



Process & Thread

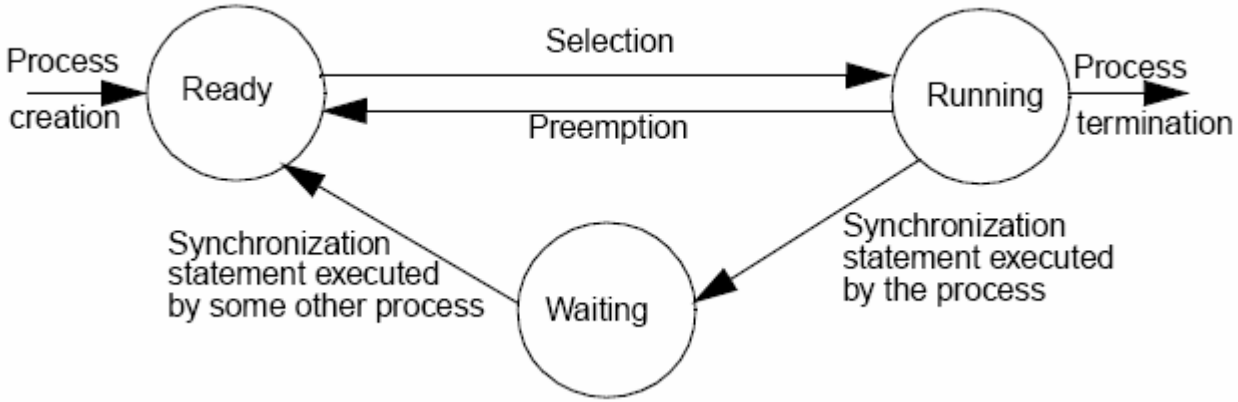
- A *process* is a self-contained running program with its own address space
- A *multitasking* operating system is capable of running more than one process at a time by periodically switching the CPU from one task to another
- The term *thread* (a.k.a. lightweight process) instead is used when the concurrent units share a single address space

Cooperation

- If the abstract machine does not support concurrency, it can be simulated by transferring control explicitly from one unit to another (*coroutines*) according to a cooperative model
- Runtime support is simpler to implement but it is more error prone:
  - the programmer has to handle cooperation: bugs in code may lock the systems

Preemption

- Modern systems adopts a *preemptive* model
- Preemption is an action performed by the underlying implementation: it forces a process to abandon its running state even if it could safely execute
- Usually a time slicing mechanism is employed where a process is suspend when a specified amount of time has expired



Concurrency in Java

- Java supports concurrency at *language level* rather than through run time libraries (like POSIX threads for C/C++)
- It employs a *preemptive* model
- If time-slicing is available (implementation dependent), Java ensures equal priority threads execute in round-robin fashion otherwise they runs to completion
- Priority is handled differently according to the specific implementation

Thread Class

- The simplest way to create a thread is to inherit from **`java.lang.Thread`**, which has all the wiring necessary to create and run threads. The most important method for **`Thread`** is **`run()`**

The thread object is defined by extending **Thread** class

You must override **run()**, to make the thread do your bidding.

Thread object is instantiated as usual through a **new()**

To run the thread you must invoke the **start()** method on it

```
class MyThread extends Thread {
  private String message;
  public MyThread(String m) {message = m;}
  public void run() {
    for(int r=0; r<20; r++)
      System.out.println(message);
  }
}
public class ProvaThread {
  public static void main(String[] args) {
    MyThread t1,t2;
    t1=new MyThread("primo thread");
    t2=new MyThread("secondo thread");
    t1.start();
    t2.start();
  }
}
```

Runnable Interface

- You can use the alternative approach of implementing the **Runnable** interface. **Runnable** specifies only that there be a **run()** method implemented, and **Thread** also implements **Runnable**

Your class must implement **Runnable** interfaces

run() must be overridden

To produce a thread from a **Runnable** object, you must create a separate Thread object

start() method is invoked to execute the thread

```
class MyThread implements Runnable {
  private String message;
  public MyThread(String m) {message = m;}
  public void run() {
    for(int r=0; r<20; r++)
      System.out.println(message);
  }
}

public class ProvaThread {
  public static void main(String[] args) {
    Thread t1, t2;
    MyThread r1, r2;
    r1 = new MyThread("primo thread");
    r2 = new MyThread("secondo thread");
    t1 = new Thread(r1);
    t2 = new Thread(r2);
    t1.start();
    t2.start();
  }
}
```

Exercise - 1

- Implement a multi-threaded program to compute the matrix product
- Each thread is responsible for a different line of the resulting matrix
- Use join() in the main program to wait all threads to finish before printing out the resulting matrix
  - the **`join()`** method used to wait until thread is done.
  - the caller of **`join()`** blocks until thread finishes

Exercise – 1 bis

- Implement a simple counter in Java which increments its value each second
- Use `sleep(1000)` to delay each print
- The counter stops whenever the user insert the value 0 from console
  - to read from console, use the following snippet (from J2SE 5.0):

```
Scanner reader = new Scanner(System.in);
int n = reader.nextInt();
```

  - note that the program cannot make any assumption on when the user will press the key

Non-determinism

- Threads execution proceeds without a predefined order
- The same code, if run on different computers, could produce different output
  - it depends on how internal scheduling is performed, on processor features, ...
- Such behavior is define as *non-deterministic*
- Non-determinism is a key point in concurrency
  - it is what makes it so hard to handle

Correctness

- A concurrent system is correct if and only if it owns the following properties:
  - *Safety*: bad things do not happen
  - *Liveness*: good things eventually happen
- Safety failures lead to unintended behavior at run time — *things just start going wrong*
  - Read / write conflicts
  - Write / write conflicts
- Liveness failures lead to no behavior — *things just stop running*
  - locking
  - waiting
  - I/O
  - CPU contention
  - failure
- Sadly enough, some of the easiest things you can do to improve liveness properties can destroy safety properties, and vice versa (e.g. locking)

J2SE 5 (a.k.a Tiger)

- Last SUN JDK release provides a number of enhancements such as support for Metadata, Generics, Enumerated types, Autoboxing of primitive types
- It improves support for concurrency too
- Package **`java.util.concurrent`** includes versions of the utilities described hereafter, plus some others
- **`http://java.sun.com/developer/technicalArticles/J2SE/concurrency/`**

Exclusion

- In a safe system, every object protects itself from integrity violations
- Exclusion techniques preserve object invariants and avoid effects that would result from acting upon even momentarily inconsistent state representations
- Three strategies:
  - Immutability
  - Dynamic Exclusion (locks)
  - Structural Exclusion

Immutability

```
class StatelessObject{
  public static int add(int a, int b) {
        return a + b;
  }
}
```

Stateless Objects

```
class ImmutableAdder {
 private final int offset;

 public ImmutableAdder(int a) {
        offset = a;
}
 public int addOffset(int b) {
        return offset + b; }
}
```

Immutable Objects (e.g., **`Integer`**)

- PROs & CONs
  - ✓ No need for synchronizing
  - ✓ Value containers (a new object is provided when value is changed)
  - ✓ Useful for sharing objects among threads
  - ✘ Limited applicability

Synchronization

```
public class RGBColor {
    private int r;
    private int g;
    private int b;

    public void setColor(int r, int g, int b) {
        checkRGBVals(r, g, b);
        this.r = r;
        this.g = g;
        this.b = b;
    }
```

- Imagine you have two threads, one thread named "red" and another named "blue". Both threads are trying to set their color of the same **`RGBColor`** object
- If the thread scheduler interleaves these two threads in just the right way, the two threads will inadvertently interfere with each other, yielding a write/write conflict. In the process, the two threads will corrupt the object's state

from *http://www.javaworld.com/jw-08-1998/jw-08-techniques-p2.html*

Objects and Locks

- Every instance of class **`Object`** and its subclasses possess a lock
- Scalar fields can be locked only via their enclosing objects
- Locking may be applied only to the use of fields within methods
- Locking an array of **`Object`** does not automatically lock all its elements

Synchronized Blocks and Methods

- Block synchronization takes an argument of which object to lock

```
synchronized (object){
    // Lock is held
    …
}
// Lock is released
```

- Declaring a method as **synchronized** would preclude conflicting traces. Locking serializes the execution of **synchronized** methods

```
synchronized void f() { /* body */ }
```

is equivalent to

```
void f() { synchronized(this) { /* body */ } }
```

Acquiring Locks

- A lock is acquired *automatically* on entry to a **`synchronized`** method or block, and released on exit, even if the exit occurs due to an exception
- Locks operate on a per-thread, not per-invocation basis. A thread hitting synchronized passes if the lock is free or the thread already possess the lock, otherwise it blocks
- A synchronized method or block obeys the acquire-release protocol only with respect to other synchronized methods and blocks on the same target object
  - *methods that are not synchronized may still execute at any time, even if a synchronized method is in progress*
- Synchronized is not equivalent to atomic, but synchronization can be used to achieve atomicity

Exercise - 2

- Consider the following class

```
class Even {
 private int n = 0;
 public int next(){
   // POST?: next is     always even
   ++n;
   Thread.sleep(500);
   ++n;
   return n;
  }
 }
```

- If multiple threads access Even, post-conditions may not be kept
- Write the unsynchronized and synchronized version of a program with two threads calling next concurrently

synchronized & OO

- The **synchronized** keyword is not considered to be part of a method's signature:
  - the **synchronized** modifier is *not* automatically inherited when subclasses override superclass methods
  - methods in interfaces cannot be declared as **synchronized**
- Synchronization in an inner class method is independent of its outer class
  - however, a non-static inner class method can lock its containing class, say **OuterClass**, via code blocks using:

```
synchronized(OuterClass.this) { /* body */ }
```

- Static synchronization employs the lock possessed by the **Class** object associated with the class the static methods are declared in.
  - The static lock for class **C** can also be accessed inside instance methods via:

```
synchronized(C.class) { /* body */ }
```

Key Rules

I. Always lock during updates to object fields

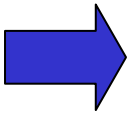
```
synchronized (point) {
   point.x = 5;
   point.y = 7; }
```

II. Always lock during access of possibly updated object fields

```
synchronized(point) {
   if(point.x > 0) {
            ...
}}
```

III. You do not need to synchronize stateless parts of methods

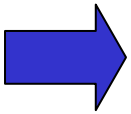
```
public synchronized void service(){
  state = ...; // update state
  operation();
 }
```



```
public void service(){
  synchronized(this) {
    state = ...; // update state
  }
  operation();
}
```

IV. Never lock when invoking methods on other objects

```
public synchronized void service(){
  ...
  h.foo();
 }
```



```
public void service(){
  synchronized(this) {
    state = ...; // update state
  }
  operation();}
```

Fully Synchronized Objects

- The safest (but not always the best) concurrent OO design strategy based on locking is to restrict attention to fully synchronized objects (also known as atomic objects) in which:
  - all methods are synchronized
  - there are no public fields or other encapsulation violations
  - all methods are finite (no infinite loops or unbounded recursion), and so eventually release locks
  - all fields are initialized to a consistent state in constructors
  - the state of the object is consistent (obeys invariants) at both the beginning and end of each method, even in the presence of exceptions

Traversal

```
class ExpandableArray {
      protected Object[] data;  // the elements
      protected int size = 0;   // the number of array slots used
      // INV: 0 <= size <= data.length
      public ExpandableArray(int cap) {
       data = new Object[cap];
      }
      public synchronized int size() { ...  }
      public synchronized Object get(int i) { ... }
      public synchronized void add(Object x) { ... }
      public synchronized void removeLast() { ... }
}
```

- In fully synchronized classes, you can add another atomic operation just by encasing it in a synchronized method
- However, this strategy does not work for another common usage of collections, *traversal*. A traversal iterates through all elements of a collection and performs some operation
- Three strategies:
  - aggregate operations
  - snapshot
  - versioned iterators

Aggregate Operations

```
interface Procedure {
 void apply(Object obj);
}

class ExpandableArrayWithApply extends
ExpandableArray {

 public ExpandableArrayWithApply(int cap) {
super(cap); }

 synchronized void applyToAll(Procedure p) {
  for (int i = 0; i < size; ++i)
   p.apply(data[i]);
 }
}
```

- This can be exploited as follow:

```
v.applyToAll(new Procedure() {
  public void apply(Object obj) {
    System.out.println(obj) }
});
```

Snapshot

- Aggregate operation eliminates potential interference holding the lock on the collection for prolonged periods
- Alternatively, clients can make a snapshot of the objects'current state and operate on it

```
Object[] snapshot;
synchronized(v) {
 snapshot = new Object[v.size()];
 for (int i = 0; i < snapshot.length,
++i)
  snapshot[i] = v.get(i);
}
for (int i = 0; snapshot.length; ++i) {
 System.out.println(snapshot[i]);
}
```

Versioned Iterators

- The third approach to traversal is for a collection class to support *iterators* that throw an exception if the collection is modified in the midst of a traversal
- The simplest way to arrange this is to maintain a version number that is incremented upon each update to the collection
- The iterator can then check this value whenever asked for the next element and throw an exception if it has changed

Deadlock

- Deadlock is possible when two or more objects are mutually accessible from two or more threads, and each thread holds one lock while trying to obtain another lock already held by another thread
- Although fully synchronized atomic objects are always safe, they may lead to deadlock

Deadlock: Example

```
class Cell {  // Do not use
 private long value;
 synchronized long getValue() { return
value; }
 synchronized void setValue(long v) {
value = v; }

 synchronized void swapValue(Cell other)
{
  long t = getValue();
  long v = other.getValue();
  setValue(v);
  other.setValue(t);
 }
}
```

- If two generic instances of **`Cell`**, say **`a`** and **`b`**, are concurrently invoking **`swapValue`**, the program may block indefinitely because a needs **`b`**'s lock and vice versa

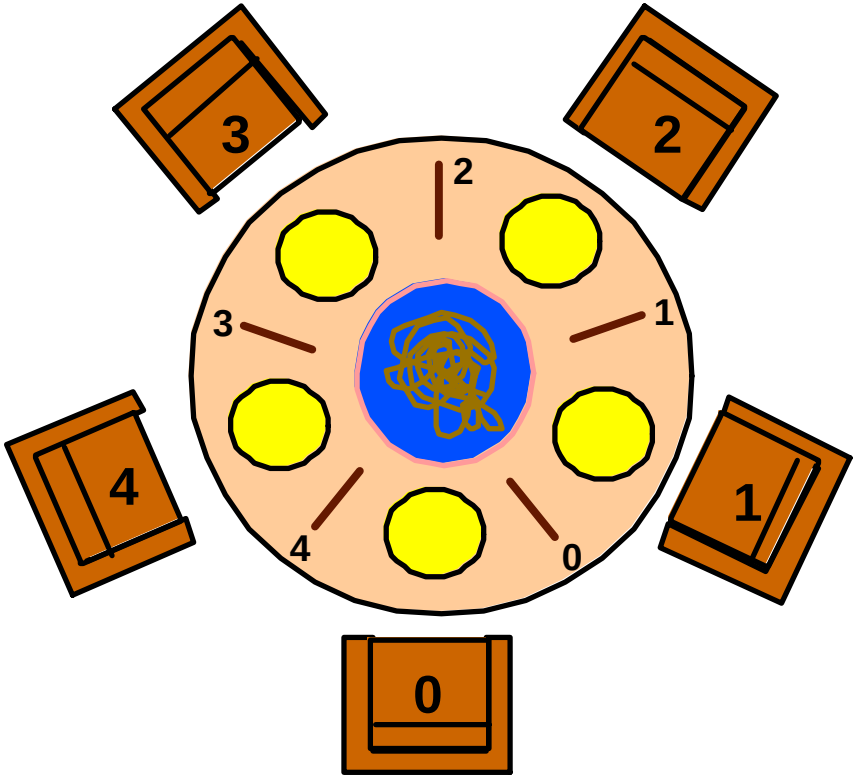
Resource Ordering

- Resource ordering can be applied to classes such as Cell without otherwise altering their structure
  - each object is associated with a tag
  - synchronization is always performed in least-first order with respect to object tags
  - then situations can never arise in which one thread has the synchronization lock for x while waiting for y and another has the lock for y while waiting for x.

```
public void swapValue(Cell other) {
 if (other == this) // alias check
  return;
 else if (System.identityHashCode(this)
      <  System.identityHashCode(other))
  this.doSwapValue(other);
 else
  other.doSwapValue(this);
}

protected synchronized void
doSwapValue(Cell other) {
 // same as original public version:
 long t = getValue();
 long v = other.getValue();
 setValue(v);
 other.setValue(t);
}
```

Dining Philosophers

- Five philosophers sit around a circular table
- Each philosopher spends his life alternately thinking and eating
- In the centre of the table is a large bowl of spaghetti
  - A philosopher needs two forks to eat a helping of spaghetti
  - One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left



Dining Philosophers…

- Philosophers grab whatever forks are available when they are hungry
  - the fork is not released unless they have got a chance to eat
- It is obvious that:
  - no two neighbors can eat at the same time
  - only two philosophers can eat at the same time
- Deadlock can occur when all philosophers decide to take their right fork , then trying to lift up the left fork but is blocked since the left fork is already taken
  - if nobody puts down a fork, a circular dependency of the waiting condition on a resource exists, i.e. a deadlock

DEMO

from *http://www-dse.doc.ic.ac.uk/concurrency/book_applets/Diners.html*

Dining Philosophers: Resource Order

- Deadlock happens because each philosopher is trying to pick up their chopsticks in a particular sequence: first left, then right
  - if each of them is holding their left chopstick and waiting to get the right one, we end with a circular wait condition
- However, if the last philosopher is initialized to try to get the right chopstick first and then the left, then that philosopher will never prevent the philosopher on the immediate left from picking up his or her right chopstick, so the circular wait is prevented

Using External Lock

- Alternatively, an additional object can be used to guarantee atomicity

```
class ExternalLock {
        public synchronized void swapCells(Cell cell1, Cell cell2) {
                cell1.swapValue(other);
        }
}
```

- In such a way, only one thread at a time is enabled to swap the cells and the latter is blocked, thus preventing deadlock

Exercise - 3

- Consider the following java classes:

```
class A {

   ...
   synchronized public void ma1(B b)
{
      ...
      b.mb2();
   }

   synchronized public void ma2() {
      ...
   }
 }
```

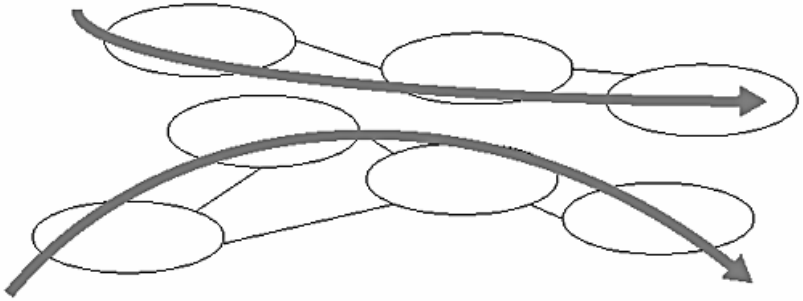
```
class B {

   ...
   synchronized public void mb1(A a)
{
      ...
      a.ma2();
   }

   synchronized public void mb2() {
      ...
   }
 }
```

- How deadlock can arise if two thread, say X and Y, concurrently access to A and B respectively ?
- How can you effectively solve the issue ?

Assignment & Lock

- The act of setting the value of a variable (except for a long or a double) is atomic
- That means there is generally no need to synchronize access simply to set or read the value of a variable
- However, threads are allowed to hold the values of variables in local memory (e.g. in a machine register). In that case, when one thread changes the value of a variable, another thread may not see the changed value (especially true for loop)
- You may synchronize access to that variable or mark it as **`volatile`**, which means that every time the variable is used, it must be read from main memory

Structural exclusion



- Confinement employs encapsulation techniques to structurally guarantee that at most one activity at a time can possibly access a given object
- This statically ensures that the accessibility of a given object is unique to a single thread without needing to rely on dynamic locking on each access
- Only one thread, or one thread at a time, can ever access a confined object

Escaping

- The key point is to avoid references escaping from their owner **Thread**
- There are four categories to check to see if a reference `r` to an object `x` can escape from a method `m` executing within some activity:
  - `m` passes `r` as an argument in a method invocation or object constructor
  - `m` passes `r` as the return value from a method invocation
  - `m` records `r` in some field that is accessible from another activity (in the most flagrant case, static fields that are accessible anywhere)
  - `m` releases (in any of the above ways) another reference that can in turn be traversed to access `r`

Confinement across Methods

- If a given method invocation creates an object and does not let it escape, then it can be sure that no other threads will interfere with (or even know about) its use of that object

```
class Plotter {    // Fragments
 // ...

 public void showNextPoint() {
  Point p = new Point();
  p.x = computeX();
  p.y = computeY();
  display(p);
 }

 protected void display(Point p) {
  // somehow arrange to show p.
 }
}
```

Confinement Across Methods - 2

- Tail-call hand-offs do not apply if a method must access an object after a call or must make multiple calls

```
public void showNextPointV2() {
   Point p = new Point();
    p.x = computeX();
    p.y = computeY();
   display(p);
   recordDistance(p); // added }
```

- Three solutions:
  - Caller copies: `display(new Point(p.x, p.y))`
  - Receiver copies: `Point localPoint = new Point(p.x, p.y)`
  - Scalar Arguments: `display(p.x, p.y)`

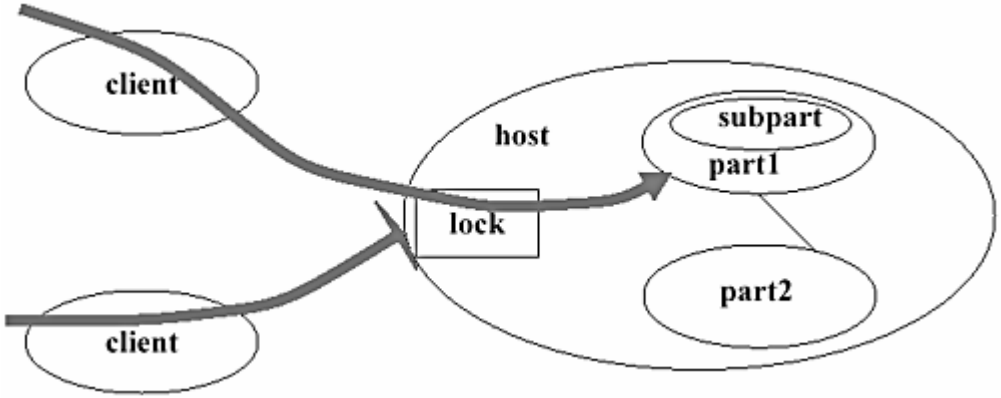
Confinement Within Threads

- The simplest and often best technique is to use a thread-per-session design:

```
class ThreadPerSessionBasedService{
public void service() {
  Runnable r = new Runnable() {
   public void run() {
    OutputStream output = null;
    try {
      output = new FileOutputStream("...");
      doService(output);
    }
    catch (IOException e) {
     handleIOFailure();
    }
    finally {
     try { if (output != null) output.close(); }
     catch (IOException ignore) {}
    } } };
  new Thread(r).start();
 }
```

Confinement Within Object

- Sometimes it is needed to share objects among threads
- You can confine all accesses internal to that object so that no additional locking is necessary once a thread enters one of its methods



- In this way, the exclusion control for the outer Host container object automatically propagates to its internal Parts

Host and Parts

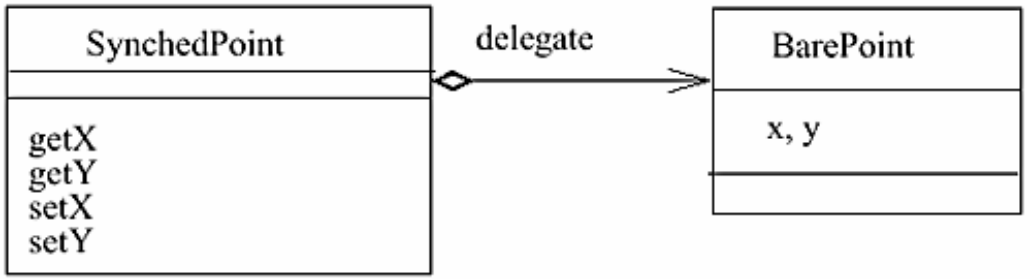
- Host object may be thought of as owning the inner Parts
- Host object constructs new instances of each Part guaranteeing that references to the Part objects are not shared by any other object
- The Host object must never leak references to any Part object:
  - it must never pass the references as arguments or return values of any method, and must ensure that the fields holding the references are inaccessible
- All appropriate methods of the host object are synchronized (while Part's are not)

Host and Parts: Example

```
class Pixel {
    private final Point pt_;
    Pixel(int x, int y) { pt_ = new Point(x, y); }
    synchronized Point location() {
       return new Point(pt_.x, pt_.y);
    }
    synchronized void moveBy(int dx, int dy){
       pt_.x += dx; pt_.y += dy;
    }
}
```

- **`Pixel`** provides synchronized access to **`Point`** methods
  - the reference to **`Point`** object is immutable, but its fields are in turn mutable (and public!) so is unsafe without protection
- Must make copies of inner objects when revealing state (see **`location()`**)

Adapters




```
class SynchedPoint {

 protected final BarePoint delegate =
new BarePoint();

 public synchronized double getX() {
         return delegate.x;}
 public synchronized double getY() {
         return delegate.y; }
 public synchronized void setX(double v)
{        delegate.x = v; }
 public synchronized void setY(double v)
{        delegate.y = v; }
}
```

- Adapters are a specialized form of Host and Parts pattern
- Adapters can be used to wrap bare unsynchronized ground objects within fully synchronized host objects

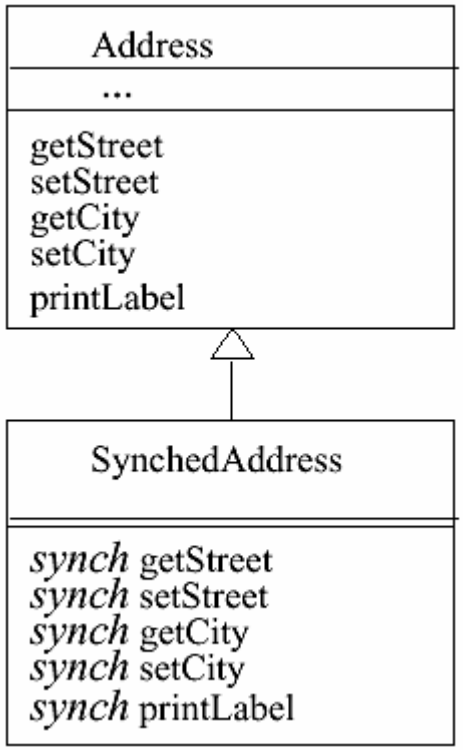
Synchronizing Collections

- The **`java.util.Collection`** framework uses an Adapter-based scheme to allow layered synchronization of collection classes
- Except for **`Vector`** and **`Hashtable`**, the basic collection classes (such as **`java.util.ArrayList`**) are unsynchronized
- However, anonymous synchronized Adapter classes can be constructed around the basic classes using for example:

```
List l = Collections.synchronizedList(new ArrayList());
```

Subclassing

- When instances of a given class are always intended to be confined within others, there is no reason to synchronize their methods
- But when some instances are confined and some are not, the safest practice is to synchronize them appropriately, even though locking is not required in all usage contexts



Why Synchronization is not so fine ?

- Invoking a **`synchronized`** method takes up to 4 times as long as unsynchronized one
- It reduces concurrency and affects performance
  - When many threads all contend for the same entry-point lock, most threads will spend most of their time waiting for the lock, increasing latencies and limiting opportunities for parallelism
- It may lead to deadlock and prevent liveness
- Next slides will present some alternatives to remove unneeded synchronization

Double Check

- The idea is to conditionally relax synchronization surrounding initialization checks
- When required value is encountered, the accessing method acquires a lock, rechecks to see if initialization is really necessary and if so performs the initialization while still under the synchronization lock, to prevent multiple instantiations

```
class Foo {
int value;
void fun() {
   if (value == 0) { // the unsynchronized check
    synchronized(this) {
     if (value == 0) { // the double-check
         // do something …
     }
    }}}
}
```

Splitting Locks

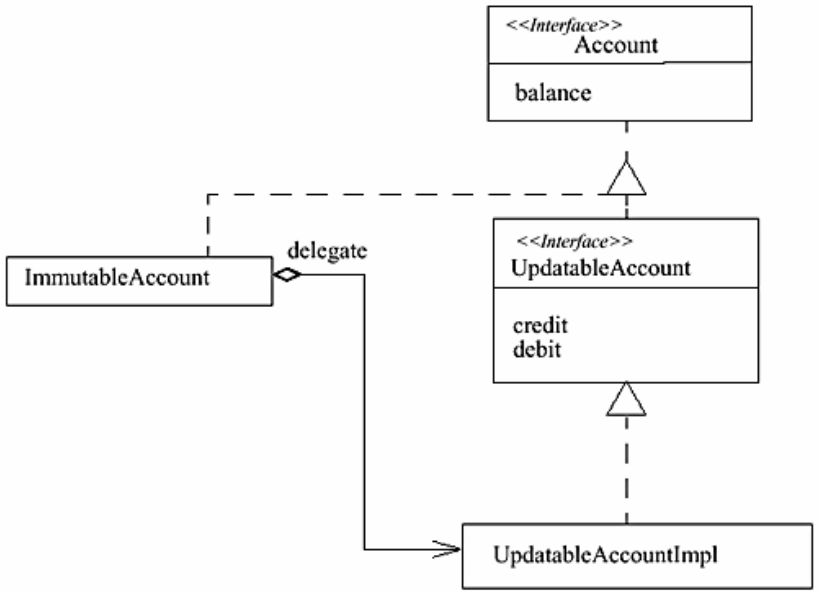
- Even if you do not want to or cannot split a class, you can still split the synchronization locks associated with each subset of functionality
- For each independent subset of functionality, declare a final object, say **`lock`**, initialized in the constructor for the Host class and never reassigned:
  - the lock object can be of any subclass of class **`Object`**.
    - *if it will not be used for any other purpose, it might as well be of class **`Object`** itself*
    - *if a subset is uniquely associated with some existing object uniquely referenced from a field, you may use that object as the lock*
- Declare all methods corresponding to each subset as unsynchronized, but surround all code with **`synchronized(lock) { ... }`**.

Splitting Locks

```
class LockSplitShape {
 protected double x = 0.0;
 protected double y = 0.0;
 protected double width = 0.0;
 protected double height = 0.0;

 protected final Object locationLock = new Object();
 protected final Object dimensionLock = new Object();

 public double x() {
  synchronized(dimensionLock) { return x; }
 }
 public double y() {
  synchronized(dimensionLock) { return y; }
 }
 public void adjustLocation() {
  synchronized(locationLock) {
   x = longCalculation1();
   y = longCalculation2();
  }
 }
 }
```

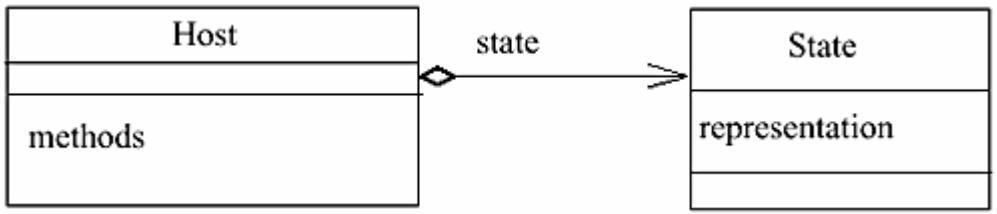
Read-Only Adapters

- Copying could be too expensive and does not make sense when dealing with objects that maintain references
- You can instead selectively permit some leakage by constructing and returning an **`Adapter`** object surrounding the part that exposes only read-only operations



Immutable State

- When a set of fields comprising the state of an object must maintain a set of interrelated invariants, you can isolate these fields in another object that preserves the intended semantic guarantees
- A good way to go about this is to rely on immutable representation objects that at all times maintain consistent snapshots of legal object states
- Relying on immutability eliminates the need to otherwise coordinate separate readings of related attributes. It also normally eliminates the need to hide these representations from clients



Java Synchronization Drawbacks

- There is no way to back off from an attempt to acquire a lock if it is already held, to give up after waiting for a specified time, or to cancel a lock attempt after an interrupt
  - difficult to recover from *liveness* problems
- There is no way to alter the semantics of a lock, for example with respect to reentrancy, read versus write protection, or fairness
- There is no access control for synchronization. Any method can perform `synchronized(obj)` for any accessible object, thus leading to potential denial-of-service problems caused by the holding of needed locks
- Synchronization within methods and blocks limits use to strict block-structured locking. For example, you cannot acquire a lock in one method and release it in another

Mutex

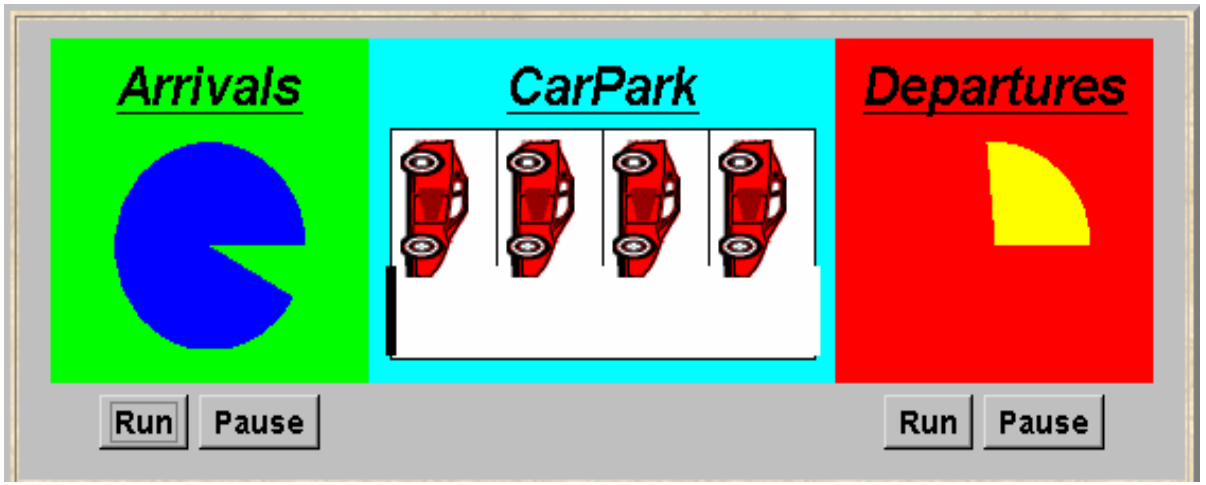
- These problems can be overcome by using utility classes to control locking such as **`Mutex`**
- However, **`Mutex`** is a rather low-level mechanism:
  - programs are difficult to write
  - no automatic check can be performed

```
public class Mutex {
 // implementation will be given later
 public void acquire() throws
                    InterruptedException;
 public void release();
 public boolean attempt(long msec) throws
                   InterruptedException;
}
```

```
try {
  mutex.acquire();
  try {
   /* body */
  }
  finally {
    mutex.release();
  }
}
catch (InterruptedException ie) {
 /* response to thread cancellation during
acquire */
}
```

Condition Synchronization

from *http://www-dse.doc.ic.ac.uk/concurrency/book_applets/CarPark.html*



- A controller is required for a carpark, which only permits cars to enter when the carpark is not full and does not permit cars to leave when there are no cars in the carpark
- Car arrival and departure are simulated by separate threads

CarParkControl Monitor

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() {
    …  --spaces; …
    }

  synchronized void depart() {
    … ++spaces; …
    }
}
```

mutual exclusion by synch methods

condition synchronization ?

block if full? (spaces==0)

block if empty? (spaces==N)

Condition Synchronization in Java

- Java provides a thread wait set per monitor (actually per object) with the following methods:

```
public final void notify()
```

Wakes up a single thread that is waiting on this object's set

```
public final void notifyAll()
```

Wakes up all threads that are waiting on this object's set

```
public final void wait()
    throws InterruptedException
```

Waits to be notified by another thread. The waiting threa releases the synchronization lock associated with the monitor. When notified, the thread must wait to reacquire the monitor before resuming execution

condition synchronization in Java

```
when cond act -> NEWSTAT
```

```
Java:  public synchronized void act()
           throws InterruptedException {
         while (!cond) wait();
           // modify monitor data
           notifyAll()
         }
```

- The **while** loop is necessary to retest the condition **cond** to ensure that **cond** is indeed satisfied when it re-enters the monitor
- **notifyAll** is necessary to awaken other thread(s) that may be waiting to enter the monitor now that the monitor data has been changed

Condition Synchronization

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() throws InterruptedException {
    while (spaces==0) wait();
    --spaces;
    notify();
  }

  synchronized void depart() throws InterruptedException {
    while (spaces==capacity) wait();
    ++spaces;
    notify();
  }
}
```

Why is it safe to use **`notify()`** *here rather than* **`notifyAll()`***?*

Single Notifications

- You can reduce the context-switch overhead associated with notifications by using a single **`notify`** rather than **`notifyAll`**
- Single notifications can be used to improve performance when you are sure that at most one thread needs to be woken. This applies when:
  - all possible waiting threads are necessarily waiting for conditions relying on the same notifications, usually the exact same condition
  - each notification intrinsically enables at most a single thread to continue. Thus it would be useless to wake up others

Summary

- Each guarded action in the model of a monitor is implemented as a synchronized method which uses a while loop and wait() to implement the guard
- Changes in the state of the monitor are signaled to waiting threads using **`notify()`** or **`notifyAll()`**
- The monitor is referred to the object instance which is used to communicate among Threads
- The lock must always be acquired before wait is invoked:

```
syncronized(lock) {
        ...
        lock.wait();
}
```

- Always re-check condition, after a wait:

```
syncronized(lock) {
  while(!cond) {
    lock.wait();
  }
}
```

YES

```
syncronized(lock) {
  if(!cond) {
    lock.wait();
  }
}
```

NO

Timed Waits

- Rather than waiting forever for a condition to become true in a guarded method, time-out designs place a bound on how long any given wait should remain suspended
- Time-outs are typically more useful than other techniques that detect unanticipated liveness problems (such as deadlock) because they make fewer assumptions about contexts

```
synchronized(object) {
  while(!cond) {
    object.wait(time);
  }
}
```

Busy Waits

- Implementing guards via waiting and notification methods is nearly always superior to using an optimistic-retry-style busy-wait "spinloop" of the form:

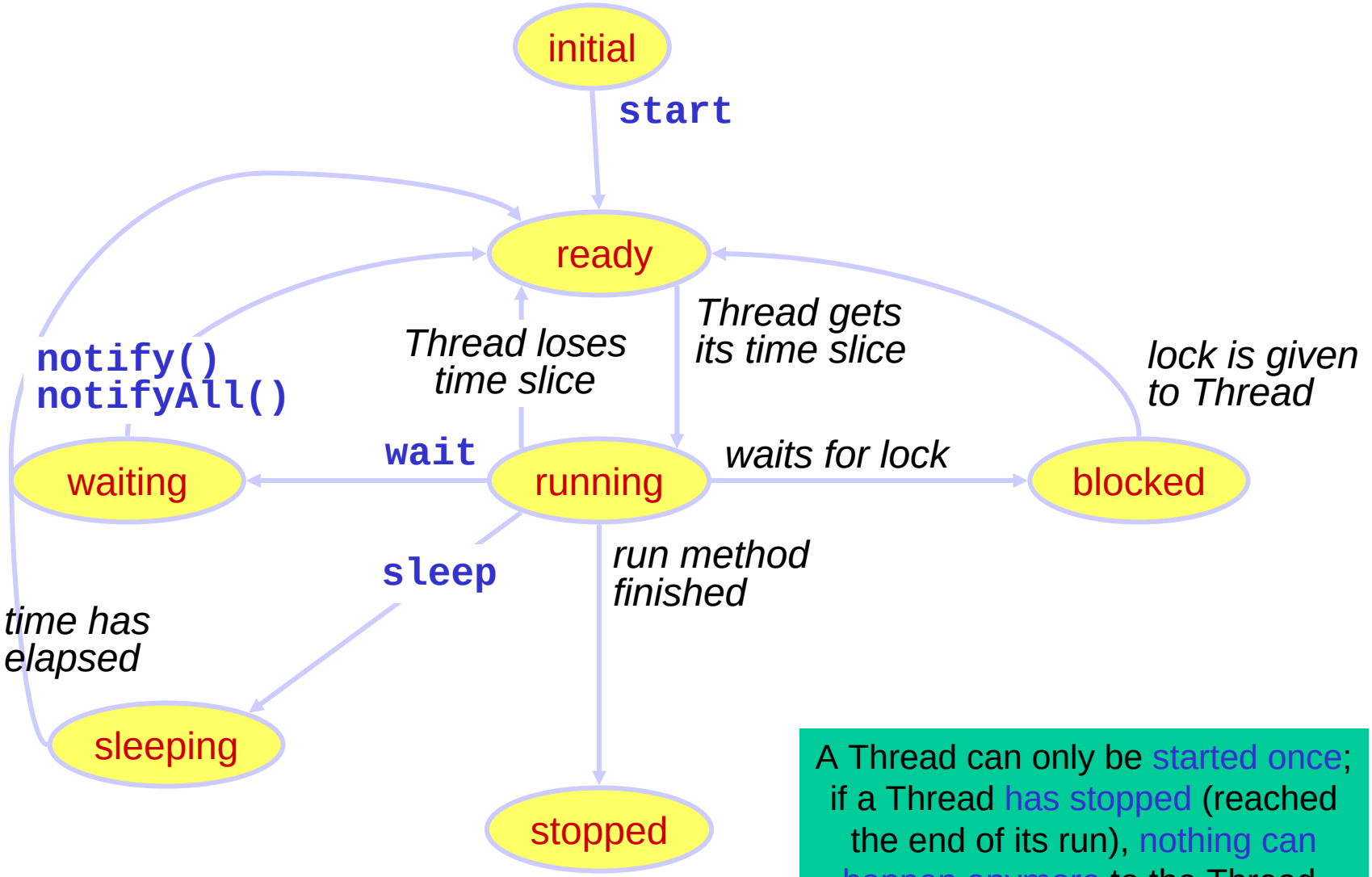
```
protected void busyWaitUntilCond() {
   while (!cond)
       Thread.yield(); }
```

- Reasons are:
  - Efficiency
  - Scheduling
  - Triggering
  - Synchronizing actions
  - Implementations

sleep()

- Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds
- The thread *does not* lose ownership of any monitors
- Useful for:
  - periodic actions
    - *need to wait for some period of time before doing the action again.*
  - allow other threads to run

Transition States



A Thread can only be started once; if a Thread has stopped (reached the end of its run), nothing can happen anymore to the Thread.

Exercise - 4

- Semaphores are classic concurrency control constructs. They conform to an acquire-release protocol
- Conceptually, a semaphore maintains a set of permits initialized in a constructor. Each **`acquire`** blocks if necessary until a permit is available, and then takes it
  - a binary Semaphore can be used to implement a **`Mutex`**
- Method **`tryAcquire`** is the same except that it returns immediately if lock is not available

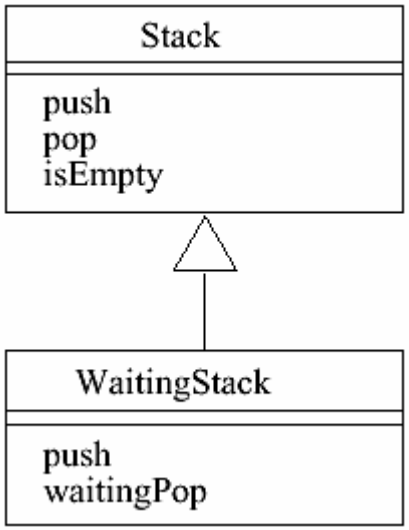
```
interface Semaphore {
        // the number of permits is defined
        // through the class constructor
        public void acquire();
        public void tryAcquire();
        public void release();
}
```

Exercise – 4 (contd.)

- Write a semaphore accepting an arbitrary number of permits
- Write an enhanced version of the class Semaphore, supporting FIFO variance

Layering Guards

- Guards may be added to basic data structure classes that were originally written in balking form




```
class WaitingStack extends Stack {
 public synchronized void push(Object x) {
  super.push(x);
  notifyAll();
 }
 public synchronized Object waitingPop()
  throws InterruptedException {
   while (isEmpty()) {
    wait();
   }
   try {
     return super.pop();
   }
   catch (StackEmptyException cannothappen) {
    // only possible if pop contains a bug
    throw new Error("Implementation bug");
   }
  }
}
```

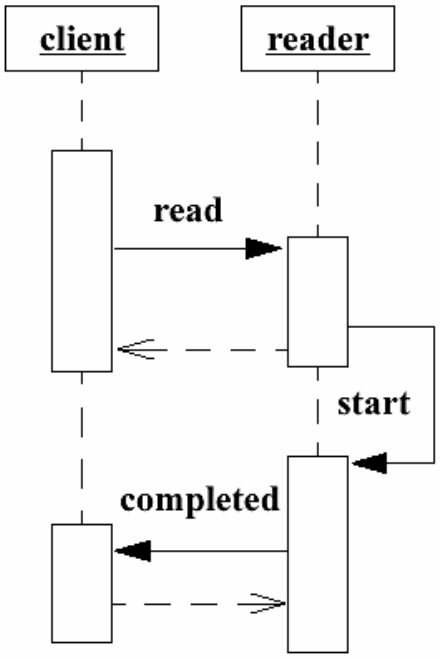
Inheritance

- Care must be taken when subclassing:
  - If a subclass includes guarded waits on conditions about which superclass methods do not provide notifications, then these methods must be recoded as seen in class **`WaitingStack`**
  - Similarly, if a superclass uses **`notify`** instead of **`notifyAll`**, and a subclass adds features that cause the conditions for using notify no longer to hold, then all methods performing notifications must be recoded

Thread per Message


```
class ThreadPerMessageHost {
 protected long localState;
 protected final Helper helper = new Helper();

 protected synchronized void updateState() {
  localState = ...;
 }

 public void req(...) {
  updateState(...);
  new Thread(new Runnable() {
   public void run() {
    helper.handle(...);
   }
  }).start();
 }
}
```

Completion Callbacks




```
interface FileReader {
  void read(String filename, FileReaderClient client);
}
interface FileReaderClient {
  void readCompleted(String filename, byte[] data);
  void readFailed(String filename, IOException ex);
}
```

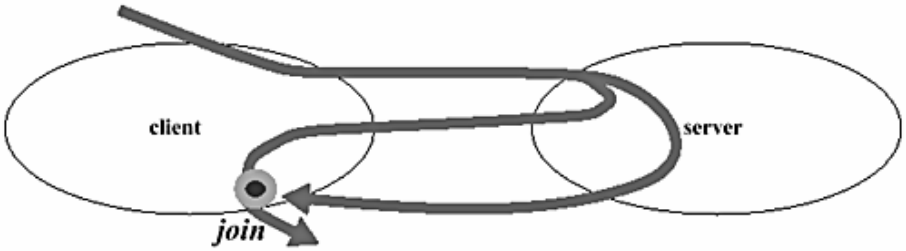
- The most natural way to deal with completion is for a client to activate a task via a oneway message to a server, and for the server later to indicate completion by sending a oneway callback message to the caller

Completion Callbacks

```
class FileReaderApp implements FileReaderClient {protected
FileReader reader = new AFileReader();

 public void readCompleted(String filename, byte[] data) {…}
 public void readFailed(String filename, IOException ex){…}
 public void actionRequiringFile() {
  reader.read("AppFile", this);
 } }
```

```
class AFileReader implements FileReader {
 public void read(final String fn, final FileReaderClient c){
  new Thread(new Runnable() {
   public void run() { doRead(fn, c); }
  }).start();
 }
 protected void doRead(String fn, FileReaderClient client) {
  byte[] buffer = new byte[1024]; // just for illustration
  try {
    FileInputStream s = new FileInputStream(fn);
    s.read(buffer);
    if (client != null) client.readCompleted(fn, buffer);
  }
  catch (IOException ex) {
   if (client != null) client.readFailed(fn, ex);
  }}}
```

Joining Threads



- While completion callbacks are very flexible, they are awkward to use when a caller just needs to wait out a particular task that it started
- This functionality is provided by **`Thread.join`**:
  - the **`join`** method blocks the caller while the target **`isAlive`**
  - terminating threads automatically perform notifications
- Either **`join`** or explicitly coded variants can be used in designs where a client needs a service to be performed but does not immediately rely on its results or effects (a.k.a. *deferred-synchronous invocation*)
  - similar to RPC-promise

Future

```
interface ArchiveSearcher { String search(String target); }
 class App {
   ExecutorService executor = ...
   ArchiveSearcher searcher = ...
   void showSearch(final String target) throws InterruptedException
{
     FutureTask<String> future =
       new FutureTask<String>(new Callable<String>() {
         public String call() {
           return searcher.search(target);
       }});
     executor.execute(future);
     displayOtherThings(); // do other things while searching
     try {
       displayText(future.get()); // use future
     } catch (ExecutionException ex) { cleanup(); return; }
   }}
```

A task that returns a result (J2SE 5)

Exercise 5

- Classes that track the execution state of underlying operations can use this information to decide what to do about new incoming requests.
- Classes based on conflict sets can employ before/after designs in which ground actions are surrounded by code that maintains the intended exclusion relations.
- This is achieved as follows:
  - For each method, declare a counter field representing whether or not the method is in progress.
  - Isolate each ground action in a non-public method.
  - Write public versions of the methods that surround the ground action with before/after control:
    - *Each synchronized before-action first waits until all non-conflicting methods have terminated, as indicated by counters. It then increments the counter associated with the method.*
    - *Each synchronized after-action decrements the method counter and performs notifications to wake up other waiting methods*

Exercise - 6

- Consider a cluster of PCs used for scientific simulations:
  - each simulation is represented by a class **`Task`** extending **`Runnable`**
  - at most N tasks can be accepted simultaneously
    - *exceeding tasks will be blocked until some of the previous tasks complete*
  - since the cluster is equipped with M < N processors, only M threads may run concurrently
  - the client needs to be notified when its computation is over
- These functionalities are to be implemented from scratch:
  - Counting Semaphore (FIFO fairness required)
  - Thread Pool
  - Completion Callback

Thread Pool

- Sometimes it is useful to control the number of worker threads to minimize chances of resource exhaustion and reduce context-switching overhead
- The thread pool contains an array of **Thread** objects. These threads will start and stop as work arrives for them
- If there is more work than threads, the work will wait until a thread free up

```
abstract class ThreadPool {
        List<Task> tasks;
        List<Worker> workers;

        abstract public void assign(Runnable r);

        abstract public Task getAssignment();
}
```

Documentation

- Comments inside code must be used to document program logic
- Here are most common JavaDoc tags:
  - **`(@precondition | @pre) (Expression)`** Describes a precondition which must be true before the method can be safely invoked
  - **`(@postcondition | @post) (Expression)`** Describes a postcondition which is true after a method has completed, successfully or not
  - **`@invariant (Expression)`** An invariant is an expression which must evaluate to true whenever the entity being described is in a *stable* state
  - **`@concurrency (SEQUENTIAL | GUARDED | CONCURRENT | TIMEOUT <value> <Throwable> | FAILURE <Throwable> | SPECIAL)`** (see next slide)

Bibliography

- Throughout the lesson we saw some patterns specific for concurrency as exposed in:
  - Doug Lea, *Concurrent Programming in Java: Design Principles and Patterns* (II edition, 1999)
- SUN Tutorial
  http://java.sun.com/docs/books/tutorial
- http://www-dse.doc.ic.ac.uk/concurrency
- http://www.mcs.drexel.edu/~shartley/ConcProgJava