Concurrent and Event-Based Programming

course slides

(c)Dan Cosma, 2008-2014

v. 1.4.1

Preliminaries

Course structure

- Two main parts
- Laboratory support
- Exam?

Interaction

- Course: presentation -> discussion
- Laboratory: problems -> solutions
- Both: feedback -> improvement

Feedback

- E-mail: danc@cs.upt.ro
- Forum (if needed)
- 'Live' discussions at the course or laboratory

Course support (bibliography)

- [1] Java Concurrency in Practice by Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea; Addison Wesley Professional, 2006 ISBN-10: 0-321-34960-1
- [2] Pattern Oriented Software Architectures - Volume 2 - Patterns for Concurrent and Networked Objects, by Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann, Wiley&Sons, 2000, ISBN-10: 0-471-60695-2

Course support (bibliography)

- [3] Event-Based Programming: Taking Events to the Limit, by Ted Faison, Apress 2006, ISBN-10: 1-59059-643-9
- [4] Concurrent and distributed computing in Java by Vijay K. Garg., Wiley & Sons, 2004, ISBN 0-471-43230-X

Part A. Concurrent Programming

Chapter 1. Introduction

What is concurrency?

- several computing tasks that execute at the same time
- the tasks may interact with each other at various times during execution

Do we need it?

Do we need it?

- Yes.

Do we need it?

- Yes.
- Why?

Why really study it?

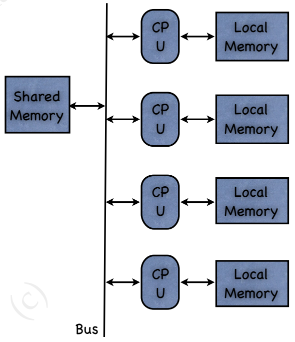
- Understand the differences between concurrent and mono-thread programming
- Know the main problems that may arise
- Understand the solutions
- Grasp the concepts
- Learn the language-specific primitives

Where is it applied?

- Parallel systems
- Multithreaded or multiprocess programs (even on non-parallel computers)
- Distributed systems

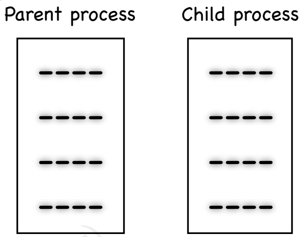
Parallel systems

- Machines with many processors, shared bus, shared memory
- The model also applies to parallel software infrastructures or languages
- Solve problems by breaking them in parallel subtasks



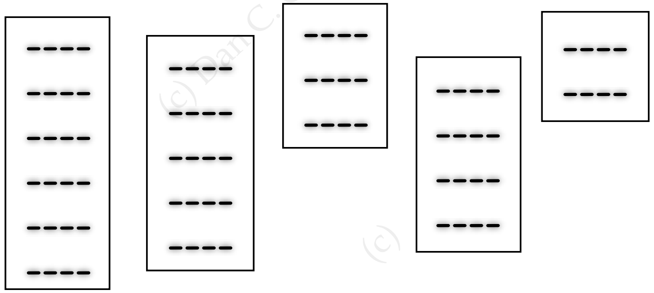
Multithreaded or multiprocess programs

```
…
if( ( pid=fork() ) < 0) {
      perror("Error");
      exit(1);
}
if(pid==0) {
      /* child */
      …
      exit(0);
}
/* parent */
…
wait(&status);
```



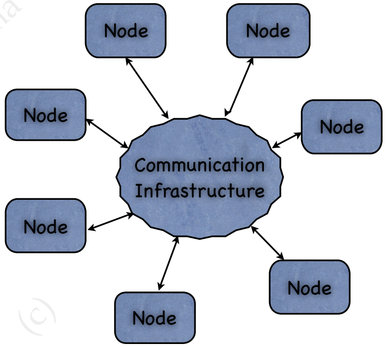
Task scheduling

- Even on mono-processor architectures, the OS provides parallel functionality



Distributed Systems

- multiple processing units running at different locations
- the components can be:
  - relatively independent
  - heterogenous
- the model can be applied to both hardware and software



Thread safety

Definition

- Applies to classes, methods, ...
- The part of the program behaves safely even when executed in multiple threads
- The part of the program runs correctly when executed in multiple threads (no changes in behavior)

A code sequence...

```
global int number;

void function increment()
{
   number =  number + 1;
}
```

```
memory stored number;

assembly routine increment()
{
   load value of number in register;
   increment register;
   store register in number;
}
```

Solution

- ...?

The State

- It's all about the state
  - variables, instance fields, static fields
- State:
  - mutable
  - immutable

Thread-safe class

- A stateless class is always thread safe
- When a state variable can be modified without synchronization the class is NOT thread-safe
- Good OO techniques help (encapsulation, immutability, etc.) but do not guarantee thread safety

Terminology

- Race conditions: the correctness of a computation depends on the relative timing of the runtime threads
- Atomic execution: the sequence is executed without interruption
- Critical region: the part of the code where race conditions may occur, and which has to be executed atomically
- Mutual exclusion: atomic execution of critical regions

Classical Issues. Synchronization Primitives

Binary Semaphores

- A primitive that can be shared between threads
- Two operations: DOWN (P), UP (V)
- DOWN
  - if value==0 blocks the thread
  - if value==1 decrements the value
- UP
  - if value==0 unblock a thread or increments
  - if value==1 does nothing
- An UP when value=0 releases a thread

Application

- Mutual exclusion between code sequences

A code sequence...

```
global int number;

void function increment()
{
   semaphore.down();
   number =  number + 1;
   semaphore.up()
}
```

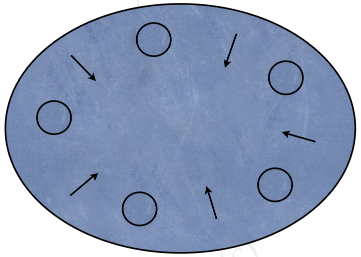
Generalized semaphores

- The value can be a natural number (0..N)
- UP increments the semaphore when value!=0
- DOWN decrements the semaphore when value!=0

Dining Philosophers Problem

- Five philosophers need to eat spaghetti
- They are poor
- Sit at a round table with 5 plates and only 5 forks
- Each philosopher needs 2 forks to eat
- They don't mind sharing
- Three phases for each: think, hungry, eat

A Picture



A philosopher

```
//this is not a good solution
void philosopher()
{
    think();
    hungry();
    semaphore[left_fork].down();
    semaphore[right_fork].down();
    eat();
    semaphore[left_fork].up();
    semaphore[right_fork].up();
}
```

Deadlock

- Two or more threads wait for each other to release a resource
- Several threads wait for resources in a circular chain
- -> The concurrent program enters a state where none of the involved parties progress

Necessary Conditions for Deadlock*

- Mutual exclusion: a resource exists that cannot be used by more than one threads at a time
- Hold and wait: threads holding resources may request new resources
- No preemption: only the resource holder can release it
- Circular wait: two or more threads form a circular chain -- one waits for the next to release the resource

Deadlock can occur only when all four conditions hold true

*E. G. Coffman, 1971

Another philosopher

```
void philosopher()
{
   think();
   hungry();
   if(left_fork is available)
     left_fork.take();
   if(right_fork unavailable for 5 minutes)
     left_fork.release();
   else
     right_fork.take();
   eat();
   left_fork.release();
   right_fork.release();
}
```

What if ALL philosopher threads start EXACTLY at the same time?

Livelock

- A situation where one or more threads do not progress, while constantly changing their relative state
- The threads are not actually blocked, but they don't advance either
- Example: Two polite persons in a narrow doorway

Resource Starvation

- A condition that occurs when one ore more threads never acquire the needed resource
- The system itself is not deadlocked
- The condition may occur due to faulty scheduling algorithms or even unfortunate timings

Starvation Examples

- The dining philosophers: imagine a solution where a philosopher takes BOTH forks at the same time: one of the five may remain hungry

Starvation Examples

- Task scheduling algorithm: three processes: A, B, C
- A: priority HIGH, B: priority LOW, C: priority VERY HIGH
- C depends on B
- The algorithm always selects the higher priority unblocked (ready) process
- A never blocks

Producer-Consumer

- A buffer shared between threads
- Producer: puts items in the buffer
- Consumer: extracts items from the buffer
- Constraints:
  - consumer: should not try and read the empty buffer
  - producer: should not try to write on full buffer

Solution 1. Correct or not?

```
int itemCount

procedure producer() {
    while (true) {
        item = produceItem()

        if (itemCount == BUFFER_SIZE) {
            sleep()
        }

        putItemIntoBuffer(item)
        itemCount = itemCount + 1

        if (itemCount == 1) {
            wakeup(consumer)
        }
    }
}
```

```
procedure consumer() {
    while (true) {
        if (itemCount == 0) {
            sleep()
        }

        item = removeItemFromBuffer()
        itemCount = itemCount - 1

        if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer)
        }
    consumeItem(item)
    }
}
```

source: wikipedia.org

Solution 2. Correct?

```
semaphore fillCount = 0
semaphore emptyCount = BUFFER_SIZE

procedure producer() {
    while (true) {
        item = produceItem()
        down(emptyCount)
        putItemIntoBuffer(item)
        up(fillCount)
    }
 }
```

```
procedure consumer() {
    while (true) {
        down(fillCount)
        item = removeItemFromBuffer()
        up(emptyCount)
        consumeItem(item)
    }
}
```

source: wikipedia.org

Solution 3. Correct

```
semaphore fillCount = 0
semaphore emptyCount = BUFFER_SIZE
semaphore mutex = 1

procedure producer() {
    while (true) {
        item = produceItem()
        down(emptyCount)
        down(mutex)
        putItemIntoBuffer(item)
        up(mutex)
        up(fillCount)
    }
}

procedure consumer() {
    while (true) {
        down(fillCount)
        down(mutex)
        item = removeItemFromBuffer()
        up(mutex)
        up(emptyCount)
        consumeItem(item)
    }
}
```

source: wikipedia.org

Readers-Writers

- Deals with concurrent access to a shared database-like resource
- Constraints:
  - A reader and a writer must not access the resource at the same time
  - Two or more writers cannot access the resource at the same time
  - Multiple readers can access the resource at the same time

A solution

```
class Readerwriter {
int numReaders = 0;
BinarySemaphore mutex = new BinarySemaphore(true);
BinarySemaphore wlock = new BinarySemaphore(true);

public void startRead() {
  mutex.down();
  numReaders++;
  if (numReaders == 1)
     wlock. down();
  mutex.up();
}

public void endRead() {
  mutex.down();
  numReaders--;
  if(numReaders == 0) wlock.up( );
  mutex.up();
}

public void startWrite() {
  wlock.down();
}

public void endWrite() {
  wlock.up() ;
}

}//end class
```

source: Concurrent and Distributed Computing in Java

Monitors

- A synchronization mechanism part of the programming language
- Entry methods: methods that are guaranteed to be synchronized (are “inside” the monitor)
- Only one thread can enter the monitor (call an entry method) at a time (“acquires the monitor lock”)

Monitors

```
//this is NOT Java
class AClass {
  entry_method aMethod()
  {
   ...
  }

  entry_method anotherMethod()
  {
   ...
  }

  method nonentryMethod()
  {
   ...
  }
}
```

Condition Variables in Monitors

- Once in monitor, a thread can apply two operations on the condition variable:
    - wait (threads can wait for the condition variable)
    - notify (other threads are signaled the condition is met)
- Waiting threads are blocked until notification (moved in the variable's waiting queue)
- The wait operation releases the monitor lock

Condition Notification

- On notify, which thread continues: the waiting thread, or the one that called notify?

  A. Hoare monitors: one of the waiting threads

  B. The thread that called notify continues; when it exits the monitor, one of the waiting threads can enter the monitor (Java behavior)

Differences

- Case A: the waiting thread that is unblocked can be certain the condition is true:

  ```
  if(!condition) conditionVariable.wait();
  ```

- Case B: the waiting thread only knows the condition MAY be true (it may have been at a previous moment):

  ```
  while(!condition) conditionVariable.wait();
  ```

Concurrency in Java

Threads

```
public class MyThread extends Thread {

public void run ( )
{
  System.out. println("Hello World");
}

public static void main(String[]args)
{
  MyThread th = new MyThread() ;
  th.start();
}
```

```
public class MyClass implements Runnable {

public void run ( )
{
  System.out. println("Hello World");
}

public static void main(String[]args)
{
  MyCLass cls = new MyClass();
  Thread th = new Thread(cls);
  th.start();
}
```

Monitors in Java

```
class AClass {
  public synchronized void method1()
  {
    ...
  }

  public synchronized void method2()
  {
    ...
  }

  public void otherMethod()
  {
    ...
  }
}
```

Monitors in Java

- Each Java object provides an 'intrinsic lock' ('monitor lock') which is automatically acquired when entering a synchronized method or block
- Java does not have explicit condition variables
- Two wait queues for the monitor
  - one for the lock to enter the monitor
  - one for a condition (threads waiting to be notified)

wait() and notify()

- wait() - blocks the calling thread
- notify() - wakes up a waiting thread
- notifyAll() - wakes up all waiting threads

```
public class SimpleProducerConsumer {
    //Message sent from producer to consumer.
    private String message;

    //True if consumer should wait for producer to send message,
    //false if producer should wait for consumer to retrieve message.
    private boolean empty = true;

        public synchronized void put(String message) {
        //Wait until message has been retrieved.
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        //Toggle status.
        empty = false;
        //Store message.
        this.message = message;
        //Notify consumer that status has changed.
        notifyAll();
    }
```

continued on next slide

source: Java Tutorial at java.sun.com

```
public synchronized String take() {
        //Wait until message is available.
        while (empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        //Toggle status.
        empty = true;
        //Notify producer that status has changed.
        notifyAll();
        return message;
    }
}
```

source: Java Tutorial at java.sun.com

Synchronized statements

```
...
private Object anObject = new Object();
...
public void aMethod()
{
  ...
  synchronized (anObject)
  {
    //this block is protected with the anObject lock
    ...
  }
  ...
}
...
```

Synchronized statements

- A synchronized method is equivalent with:

```
public void aMethod()
{
    synchronized(this)
    {
        //method body
        ...
    }
}
```

Synchronized statements

- Example of usage:

```
public class IndependentLocking {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }
                                                    public void inc2() {
                                                        synchronized(lock2) {
                                                            c2++;
                                                        }
                                                    }
                                                }
```

source: Java Concurrency in Practice

Reentrancy

- Intrinsic locks in Java are reentrant: if a thread tries to acquire a lock that it already holds, the operation succeeds
- Locks are acquired by per-thread rather than per-invocation basis

Code that would deadlock if locks were not reentrant

```
public class AnAncestor {
  public synchronized void method()
  {
    ...
  }
}

public class AClass extends AnAncestor {
   public synchronized void method()
   {
       System.out.println("Hello World!");
       super.method();
    }
}
```

Rules for guarding the state with locking

- There is no inherent link between the lock and the state it protects
- A mutable state variable must be guarded by using the same lock object from all threads
- Every shared state variable must be guarded using only one lock object
- For an invariant that uses several state variables, all the respective variables must be guarded by the same lock

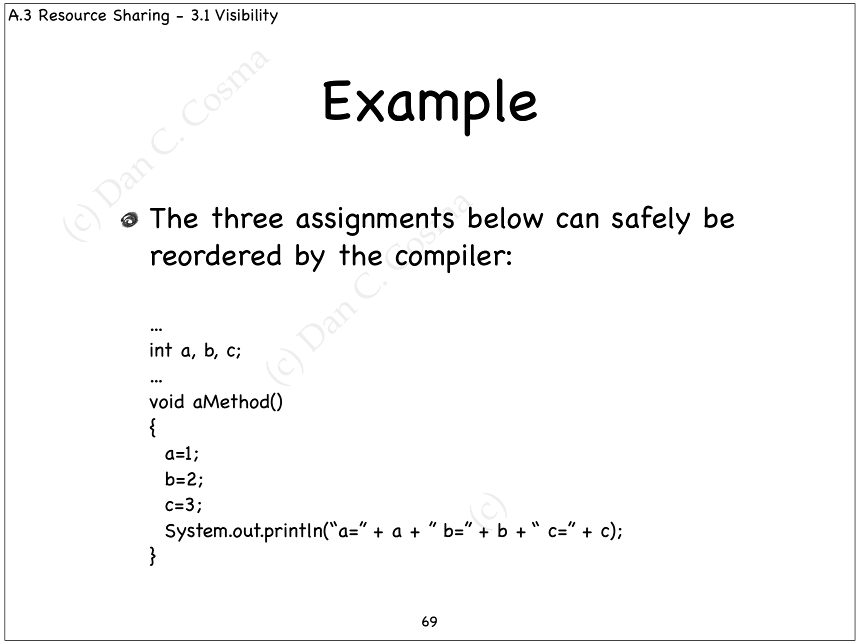
Visibility

Compiler optimizations

- To benefit on the pipelined, multiprocessor, or multicore architectures, compilers do complex optimizations on the code
- Frequent cases:
  - reordering -- operations can be done in a different order than the one specified by the program
  - caching -- variables can be cached in registers or processor caches

Compiler optimizations

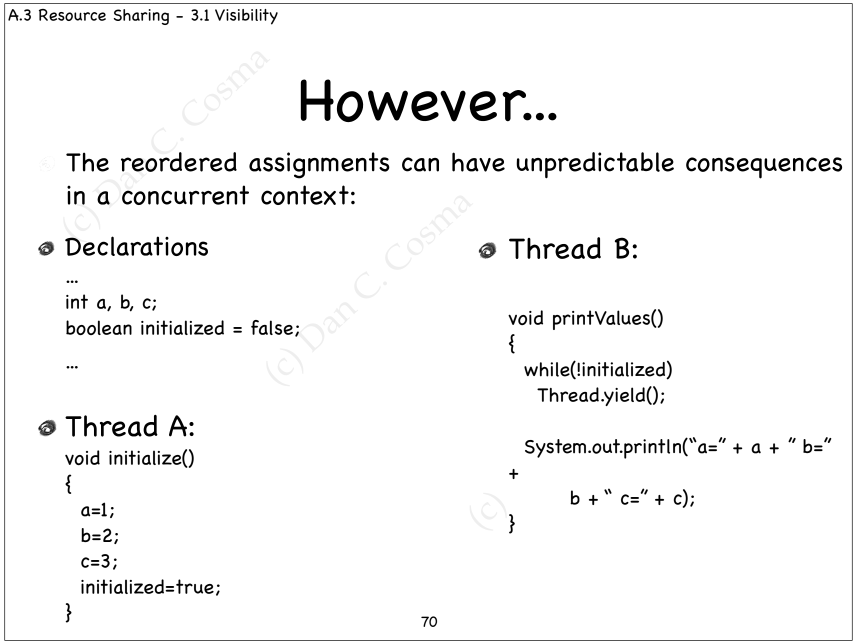
- If explicit synchronization is missing, the compiler optimizes the code so that it runs faster as a single thread
- The programmer cannot make assumptions regarding
  - the order the operations are executed
  - the time or sequence when memory values become visible for other threads



Example

- The three assignments below can safely be reordered by the compiler:

```
…
int a, b, c;
…
void aMethod()
{
  a=1;
  b=2;
  c=3;
  System.out.println(“a=” + a + ” b=” + b + “ c=” + c);
}
```



However…

- The reordered assignments can have unpredictable consequences in a concurrent context:

- Declarations

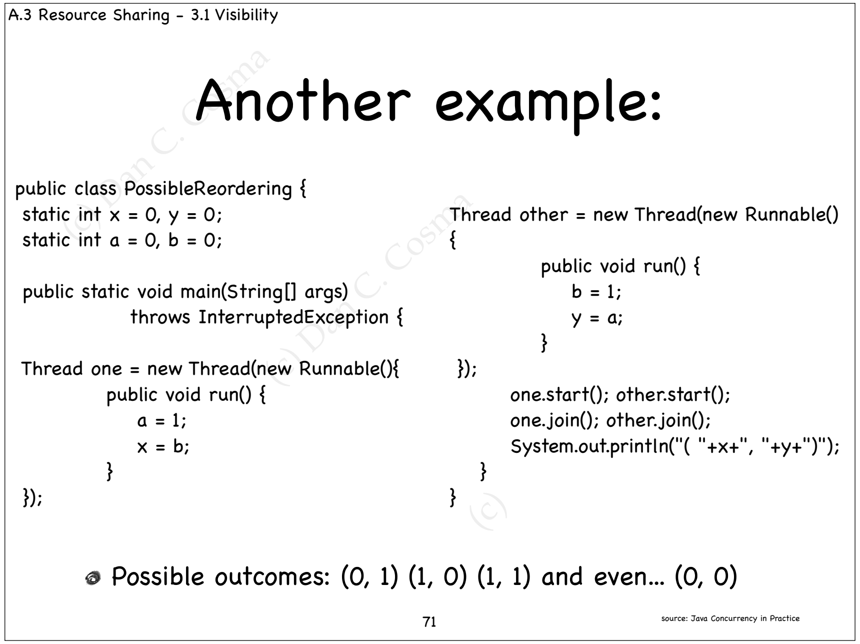
```
…
int a, b, c;
boolean initialized = false;
…
```

- Thread A:

```
void initialize()
{
  a=1;
  b=2;
  c=3;
  initialized=true;
}
```

- Thread B:

```
void printValues()
{
  while(!initialized)
    Thread.yield();

  System.out.println(“a=” + a + ” b=”
+
        b + “ c=” + c);
}
```



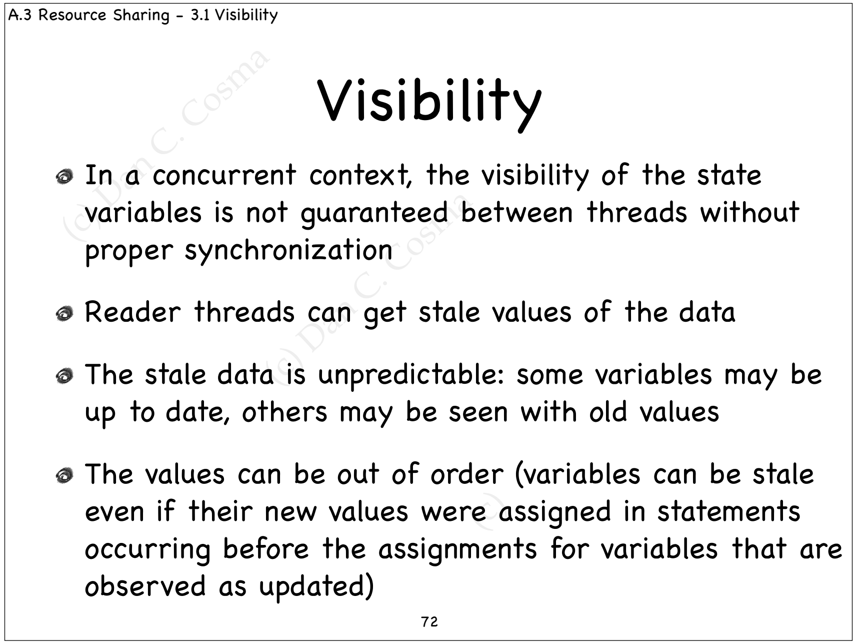
Another example:

```
public class PossibleReordering {
 static int x = 0, y = 0;
 static int a = 0, b = 0;

 public static void main(String[] args)
              throws InterruptedException {

 Thread one = new Thread(new Runnable(){
            public void run() {
                a = 1;
                x = b;
            }
 });
```

```
Thread other = new Thread(new Runnable()
{
            public void run() {
                b = 1;
                y = a;
            }
 });

        one.start(); other.start();
        one.join(); other.join();
        System.out.println("( "+x+", "+y+")");
   }
}
```

- Possible outcomes: (0, 1) (1, 0) (1, 1) and even… (0, 0)

source: Java Concurrency in Practice



Visibility

- In a concurrent context, the visibility of the state variables is not guaranteed between threads without proper synchronization
- Reader threads can get stale values of the data
- The stale data is unpredictable: some variables may be up to date, others may be seen with old values
- The values can be out of order (variables can be stale even if their new values were assigned in statements occurring before the assignments for variables that are observed as updated)

The solution

- In order to ensure correct visibility, always use explicit synchronization when accessing shared state
- Intrinsic locking:
  - Thread A executes a synchronized block
  - Thread B subsequently locks on the same lock

  --> all the variables visible to A before releasing the lock are guaranteed to be visible to B when acquiring the same lock

Volatile variables

Volatile variables

- A Java construct that provides cheap yet weak synchronization on memory accesses

```
…
volatile int variable;
…
```

- Usually, volatile provides better performance than synchronized
- Must be used with great care

What does volatile do?

- Guarantees visibility but does NOT provide atomicity or locking
- Operations on a volatile variable are not reordered: threads will see the most up to date value of a volatile variable
- The effect extends to other variables:

  -> all variable values visible (at the time of writing) to the thread that writes a volatile are guaranteed to be visible to threads that subsequently read the respective volatile value

When is it safe to use volatile?

- Both of the following criteria must be met:
  - writes to the volatile variable must not depend on its current value
  - the volatile variable does not participate in invariants with other variables

Therefore...

- The first criterion shows that
  - a volatile variable can NOT be safely used as a counter or for similar purposes: the incrementing/modification is NOT atomic.
  - Still, if the write on a volatile is done from a SINGLE thread, this criterion can be ignored
- The second criterion warns the programmer there are many cases when using volatile is dangerous (its effect may not be that obvious)

Example of participation in an invariant

```
public class NumberRange {
    private volatile int lower, upper;
    public int getLower() { return lower; }
    public int getUpper() { return upper; }

    public void setLower(int value) {
        if (value > upper)
            throw new IllegalArgumentException(...);
        lower = value;
    }

    public void setUpper(int value) {
        if (value < lower)
            throw new IllegalArgumentException(...);
        upper = value;
    }
}
```

- Class invariant: lower < upper
- Initial state: (0,7); Concurrent access: thread A: setLower(5), thread B: setUpper(3)
- Can result in a wrong state of (5,3) because of timing and lack of locking

source: http://www.ibm.com/developerworks/java/library/j-jtp06197.html

Example of using volatile

- A status flag:

```
volatile boolean exitProgram = false;
...
void setExit(boolean value) {
  exitProgram = true;
}

void run() {
  while(!exitProgram)
  {
    //exitProgram not modified here
    ...
  }
...
```

- If volatile wasn't used, the compiler may have optimized the code:

```
...
if(!exitProgram)
  while(true)
    {
      //exitProgram not modified here
      ...
    }
...
```

Object Publication

The Problem

- When using primitive variables the synchronization is easier (they can be modified only within the language-specific visibility scope)
- Objects can be referred from more than one places (multiple references are possible)
- We must control the way references are created and passed between threads

The simplest mistake

- Publishing an object reference in a static field

```
public static Set<Secret> knownSecrets;
public void initialize() {
    knownSecrets = new HashSet<Secret>();
}
```

- The Secret object can be modified by any class

source: Java Concurrency in Practice

Escaped objects

- An object is considered escaped when it is published when it should not have been
- Escaped objects can make the code thread unsafe.
- Why? -- It's all about control (who, where, how accesses the object)

Therefore...

- The programmer must be careful when allowing the reference to an object to become available for other classes
- The thread safety related characteristics of objects must be well documented by the programmer:
  - is the object thread safe?
  - can the field be published?
  - is the object part of an invariant?
  - ...

Escaped internal state

- The internal mutable state of objects should be carefully protected
- If possible, do not create public methods that return references to state objects

```
    //do NOT do this
    class UnsafeStates {

    private String[] states = new String[] {
        "AK", "AL" ...
    };
    public String[] getStates() { return states; }
}
```

source: Java Concurrency in Practice

Side effects

- Publishing an object will automatically publish
  - all its public fields
  - all objects its public methods return
- Complex chains of published objects can be created by a single inadvertent publication

Alien methods

- Alien method: a method whose behavior is not fully specified by the current class
- Examples:
  - methods in other classes
  - overrideable methods (neither <u>private</u>, nor <u>final</u>)
- Passing an object to an alien method is dangerous for the thread safety

2 (problems) in 1

```
public class ThisEscape {
    public ThisEscape(EventSource source) {
        source.registerListener(
            new EventListener() {
                public void onEvent(Event e) {
                    doSomething(e);
                }
            });
    }
}
```

- Publishing an inner class implicitly publishes the enclosing object
- An object can inadvertently escape during construction without being constructed completely (here: this escapes due to the publishing of the enclosed EventListener)

source: Java Concurrency in Practice

Object construction

- The reference to the current object should not be published:
  - avoid calling alien methods with the object as a parameter (either explicit or implicit)
  - even if the escape is the last statement in the constructor, it is NOT safe
- Escaping the current object during construction can lead to threads being provided incomplete (partially initialized) objects

A solution

```
public class SafeListener {
    private final EventListener listener;

    private SafeListener() {
        listener = new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);
            }
        };
    }

    public static SafeListener newInstance(EventSource source) {
        SafeListener safe = new SafeListener();
        source.registerListener(safe.listener);
        return safe;
    }
}
```

- A factory method is used for creating the object
- The actual construction is done in a private constructor

source: Java Concurrency in Practice

Thread Confinement

The context

- The simplest solution for thread-safety problems related to shared objects: do not share
- It's not always feasible
- However, if applicable, it can solve the concurrency problems in an efficient manner

Thread Confinement

- Ensure the non-thread safe code is executed within a single thread
- Examples of such approaches:
  - Swing: the visual components are not thread safe; there is a single dispatch thread that confines them all
  - Pooling JDBC Connection objects (the Connection object is not thread safe)

Ad-hoc Thread Confinement

- The confinement is entirely managed by the implementation
- There is no support in languages for this scenario
- The confinement must be carefully implemented and thoroughly documented
- Advantage: provides flexibility and complete control over the way the thread confinement behaves

Stack Confinement

- Exploit the way local variables are allocated
  - they are stored on the thread's stack
- Consequently, the local variables are not shared between threads
- Local primitive variables (int, long, etc.) are always safe to use
- For local objects -- their escape from the method must also be prevented

Example of stack confinement

```
public int loadTheArk(Collection<Animal> candidates) {
    SortedSet<Animal> animals;
    int numPairs = 0;
    Animal candidate = null;

    // animals confined to method, don't let them escape!
    animals = new TreeSet<Animal>(new SpeciesGenderComparator());
    animals.addAll(candidates);
    for (Animal a : animals) {
        if (candidate == null || !candidate.isPotentialMate(a))
            candidate = a;
        else {
            ark.load(new AnimalPair(candidate, a));
            ++numPairs;
            candidate = null;
        }
    }
    return numPairs;
}
```

- numPairs is safe
- animals is kept inside the method
- the confinement of animals MUST be DOCUMENTED

source: Java Concurrency in Practice

ThreadLocal

- ThreadLocal is a class provided by the Java API
- Encapsulates an user object and makes them private to each thread
- Each thread will work on its own copy of the object

ThreadLocal

```
public class ThreadLocal<T> {
  public T get(); // Returns the value of the current thread's copy of the variable.

  public void set(T newValue); // Sets the current thread's copy of the variable.

  void remove(); // Removes the current thread's copy of the variable.

  public T initialValue(); // Returns null; can be overriden. Invoked in each thread
                           // at the first get() that was not preceded by a set(),
                           // or the first get() after a remove().
}
```

- Each thread that accesses a ThreadLocal will be provided a separate copy for the enclosed variable
- It's as if ThreadLocal stored a map of values for the threads (although the actual implementation is different)

Example

```
private static ThreadLocal<Connection> connectionHolder
    = new ThreadLocal<Connection>() {
        public Connection initialValue() {
            return DriverManager.getConnection(DB_URL);
        }
    };

public static Connection getConnection() {
    return connectionHolder.get();
}
```

source: Java Concurrency in Practice

Immutability

Immutable objects

- Objects whose state
  - does not change after construction
  - cannot be changed by other objects
- Immutable objects are always thread-safe

Benefits of immutability

- Simplicity: they only have one state during their entire life cycle
- Safety: they can be safely passed to untrusted code, as their state cannot be modified maliciously or due to bugs
- Can be easily cached: their state doesn’t change, therefore the cached values are consistent with the original object

Example: problem…

```
java.util.Date crtDate = new java.util.Date();
AppointmentManager.setAppointment(crtDate, team, “Off-world mission to ” + planet);
crtDate.setTime(crtDate.getTime() + ONEDAY);
AppointmentManager.setAppointment(crtDate, team, “Briefing for mission to ” + planet);
```

- java.util.Date is mutable
- If the implementation of AppoinmentManager.setAppointment does not copy (clone) the date value into its internal state:
  - both appointments may be set to the next day
  - the internal data used in the AppointmentManager may become corrupt in a concurrent context
- This is a subtle and easy to make mistake

…and solution

```
public final class ImmutableDate {
private final Date date;

public ImmutableDate(Date date) {this.date = date.clone();}
public ImmutableDate(long milliseconds) {this.date = new Date(milliseconds);}

public long getTime() {
  return this.date.getTime();
}
}
```

```
…
ImmutableDate crtDate = new ImmutableDate(new java.util.Date());
AppointmentManager.setAppointment(crtDate, team, "Off-world mission to " + planet);
crtDate = new ImmutableDate(crtDate.getTime() + ONEDAY);
AppointmentManager.setAppointment(crtDate, team, "Briefing for mission to " + planet);
```

Immutable class containing mutable objects

```
public final class FlagColors_RO {
  private final Set<String> colors = new HashSet<String>();

  public FlagColors() {
    colors.add("blue"); colors.add("yellow"); colors.add("red");
  }

  public boolean isOnFlag(String color) {
    return colors.contains(color);
  }
}
```

Conditions for immutability

- For a class to be immutable, all of the following conditions must be true:
  - all fields are final
  - the class is declared final
  - the this reference does not escape during construction
  - any fields referring mutable objects must:
    + be private
    + never be returned or exposed in any way to callers
    + be the only references to the respective objects
    + not change the state of the referenced objects after construction

Safe Publication

Improper publication

- Do not publish an object without proper synchronization

Unwanted effects

- Improper publication can lead to threads accessing partially constructed objects
- The state of the improperly published objects can change from “partially constructed” to “initialized” at any time

Example

```
// Unsafe publication
public Holder holder;

public void initialize() {
    holder = new Holder(42);
}

public class Holder {
    private int n;

    public Holder(int n) { this.n = n; }

    public void assertSanity() {
        if (n != n)
            throw new AssertionError("This statement is false.");
    }
}
```

- assertSanity() may throw the exception!

source: Java Concurrency in Practice

When coding...

- There are two separate concerns regarding the objects:
  - how safely are they published
  - how safely are they used
- Both concerns are essential for thread safety

Immutable objects

- Can be safely published without synchronization
- Can be safely used without synchronization
- Note: to be immutable, the object must follow all the immutability requirements specific to the programming language

Effectively immutable objects

- Objects that are not immutable by the definition, but are never modified after publication
- Example: a Date object that is never modified
- The only concern is to publish them properly; afterwards they can be used without synchronization

Safe Publication

- For objects that are not immutable, safe publication of properly constructed objects can be done by:
  - initializing the reference from a static initializer
  - storing the reference in a volatile variable
  - storing a reference in a final field of a properly constructed object
  - storing a reference in a variable correctly guarded by a lock

Sharing objects safely

- When acquiring a reference to an object, the programmer must clearly understand:
  - does a lock need to be acquired?
  - is it allowed to modify the object's state?
  - does it need to be copied rather than used directly?

Strategies for safe use

- Thread confined objects can be safely used by the confining thread
- Shared read-only objects are safe to read without synchronization
- Shared thread-safe -- an object documented as thread safe manages the synchronization internally, therefore is safe to use
- Guarded objects -- protecting objects with locks makes them safe to use

Building Concurrent Applications

Designing a thread-safe class

Designing a thread safe class

- Identify the variables that form the state
- Identify the invariants constraining the state
- Understand the preconditions and postconditions for the operations
- Establish a synchronization policy

Synchronization policy

- How an object coordinates the access to its state
- Specifies:
  - the combination of techniques such as immutability, thread confinement, locking
  - which variables are guarded by which locks

The object's state

- The state of an object is made of:
  - its primitive type fields
  - some of the object's fields that refer other objects
- Encapsulating the state is very important: significantly eases the process of making the class thread safe

The state space

- State space: the range of possible states objects and variables can take on
- The smaller the state space, the easier to reason about it

The state space

- Invariants specify the valid states:
  -> if certain states are invalid, the respective state variables must be encapsulated to prevent clients to create invalid states
- Complex invariants may imply several state variables
  -> atomic operations: the related variables must be all modified in a single atomic operation. Example: the interval bounds (a,b)

Postconditions

- Operations may have postconditions that identify which state transitions are valid
  - example: a counter n, the only possible next state is n+1
- Operations for which invalid transitions are possible must be made atomic

Preconditions

- Some operations can have preconditions regarding the state.
  - example: a queue must not be full before storing an item
- These operations are called state-dependent
- In a non-concurrent context, an operation for which the precondition is false simply fails
- In concurrent programs, threads can wait for the precondition to become true

State ownership

- Not all the objects stored in a class' fields form its state
- Only the objects it owns are part of the state
- Example: a collection owns its internal storage-related data, but not the stored objects
- Usually, encapsulation and ownership are good together: the object encapsulates the owned state, and owns the state it encapsulates

Types of ownership

- Exclusive ownership -- only one class owns the objects
- Long-term shared ownership -- the same state object is owned by more classes
  - example: a state object published to communicate with another class
- Temporary shared ownership -- a class is given an object to use it temporarily
  - example: parameters in constructors
- Split ownership -- although it receives the reference, it does not own or use the object
  - example: the objects stored in collections

Understanding the ownership

- Is important so that the fields that form the state are identified
- Classes should not modify objects they do not own
- While the programming language doesn’t specifically support the ownership delimitation, this is a very important issue in the class design
- The ownership must be documented

Instance confinement

Instance confinement

- A method that makes implementing thread safety simpler
- It implies the encapsulation of a non-thread-safe object within another object which we control
  -> the paths accessing the data are known
  -> to analyze the thread safety we have to look only at a small part of the code
- Combined with proper locking, ensures thread safety

Example

```
@ThreadSafe
public class PersonSet {
    @GuardedBy("this")
    private final Set<Person> mySet = new
HashSet<Person>();

    public synchronized void addPerson(Person p) {
        mySet.add(p);
    }

    public synchronized boolean
containsPerson(Person p) {
        return mySet.contains(p);
    }
}
```

- The HashSet is not thread safe
- By encapsulating it and by locking on the implicit lock this code ensures thread safety

source: Java Concurrency in Practice

Instance Confinement with the Java Monitor Pattern

- Encapsulate all the state and guard it with the intrinsic lock of the object
- Alternatively to this pattern, private locks can also be used:

```
public class PrivateLock {
    private final Object myLock = new Object();
    @GuardedBy("myLock") Widget widget;

    void someMethod() {
        synchronized(myLock) {
            // Access or modify the state of widget
        }
    }
}
```

source: Java Concurrency in Practice

A slightly more complex example

- A class that tracks the locations of vehicles
- Designed to be used concurrently by a view and updater thread (in an MVC pattern)

A slightly more complex example

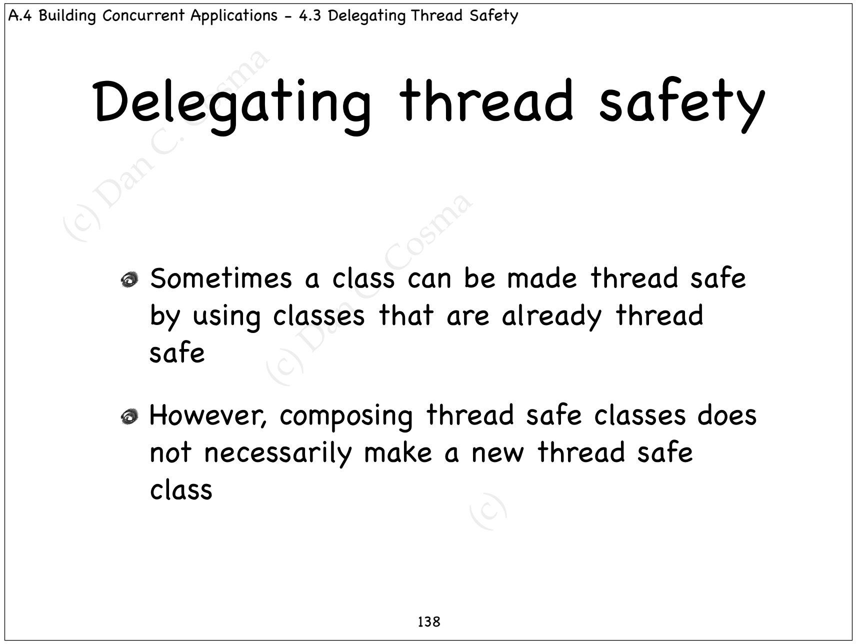
```
@ThreadSafe
public class MonitorVehicleTracker {
    @GuardedBy("this")
    private final Map<String, MutablePoint>
locations;

    public MonitorVehicleTracker(
            Map<String, MutablePoint> locations) {
        this.locations = deepCopy(locations);
    }

    public synchronized Map<String, MutablePoint>
getLocations() {
        return deepCopy(locations);
    }

    public synchronized  MutablePoint
getLocation(String id) {
        MutablePoint loc = locations.get(id);
        return loc == null ? null : new
MutablePoint(loc);
    }

    public synchronized  void
setLocation(String id, int x, int y) {
        MutablePoint loc = locations.get(id);
        if (loc == null)
            throw new
IllegalArgumentException("No such ID: " + id);
        loc.x = x;
        loc.y = y;
    }

    private static Map<String, MutablePoint>
deepCopy(
            Map<String, MutablePoint> m) {
        Map<String, MutablePoint> result =
                new HashMap<String,
                                MutablePoint>();
        for (String id : m.keySet())
            result.put(id, new
MutablePoint(m.get(id)));
        return
Collections.unmodifiableMap(result);
    }
}
```

source: Java Concurrency in Practice

A slightly more complex example. Remarks

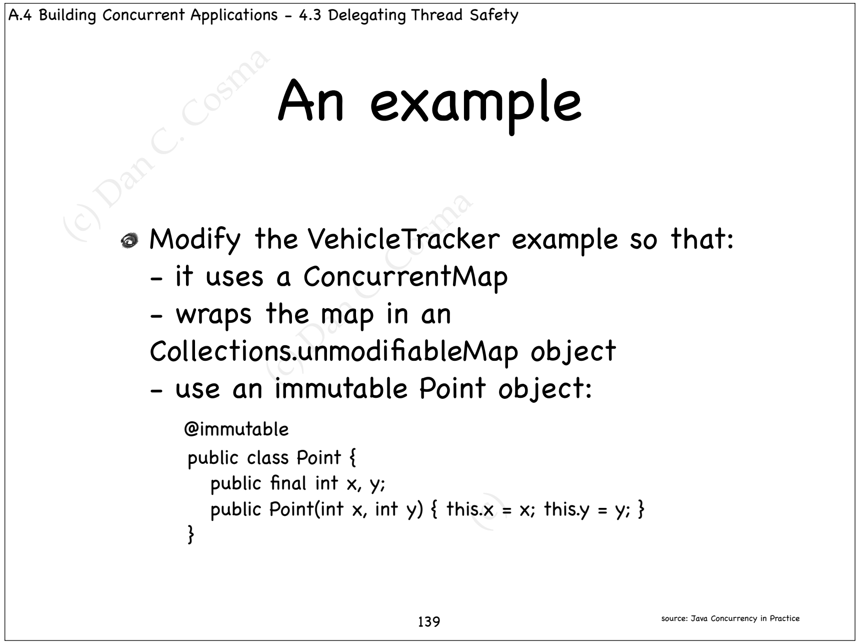
- MutablePoint is not thread safe
- VehicleTracker is thread safe
  - the map and the contained points are never published
  - when initialized and returned, the locations are copied in depth (both the map and the elements)
  - deepCopy() holds the lock for a relatively complex operation (performance problem)

Delegating thread safety



Delegating thread safety

- Sometimes a class can be made thread safe by using classes that are already thread safe
- However, composing thread safe classes does not necessarily make a new thread safe class

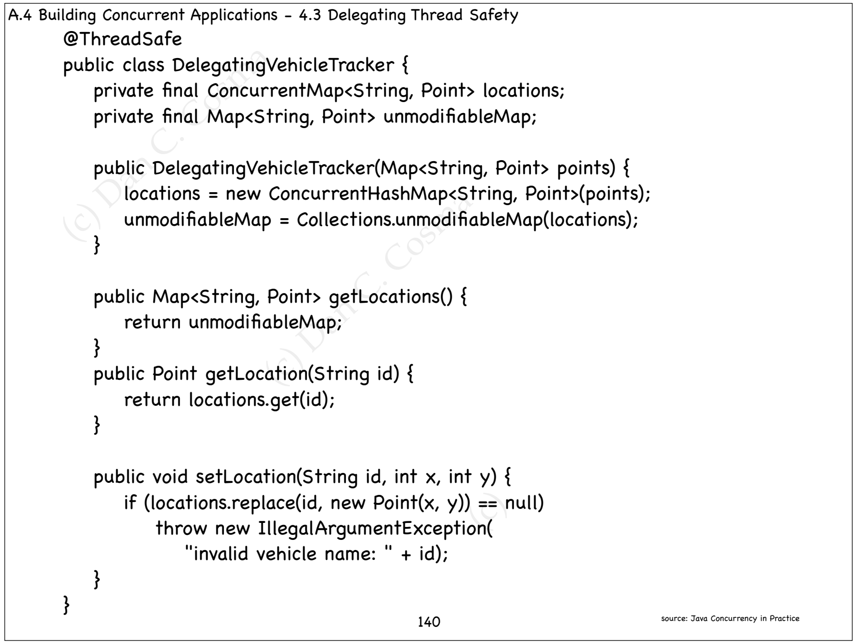


An example

- Modify the VehicleTracker example so that:
  - it uses a ConcurrentMap
  - wraps the map in an Collections.unmodifiableMap object
  - use an immutable Point object:

```
@immutable
public class Point {
    public final int x, y;
    public Point(int x, int y) { this.x = x; this.y = y; }
}
```

source: Java Concurrency in Practice



```
@ThreadSafe
public class DelegatingVehicleTracker {
    private final ConcurrentMap<String, Point> locations;
    private final Map<String, Point> unmodifiableMap;

    public DelegatingVehicleTracker(Map<String, Point> points) {
        locations = new ConcurrentHashMap<String, Point>(points);
        unmodifiableMap = Collections.unmodifiableMap(locations);
    }

    public Map<String, Point> getLocations() {
        return unmodifiableMap;
    }
    public Point getLocation(String id) {
        return locations.get(id);
    }

    public void setLocation(String id, int x, int y) {
        if (locations.replace(id, new Point(x, y)) == null)
            throw new IllegalArgumentException(
                "invalid vehicle name: " + id);
    }
}
```

source: Java Concurrency in Practice

Remarks

- The Point was made immutable because it is published by the getLocations() method
- As getLocations() does not copy the map upon returning it, the modifications in the location elements with setLocations() are visible "live" in threads. If a static copy is needed, the method must be changed as follows:

```
return Collections.unmodifiableMap(
          new HashMap<String, Point>(locations));
```

source: Java Concurrency in Practice

Independent state variables

- The thread safety is properly delegated when:
  - the delegation is done to one or several independent thread safe state variables (which do NOT participate in an invariant for the state)
  - there are no operations that have invalid state transitions

Incorrect delegation

```
public class NumberRange {
    // INVARIANT: lower <= upper
    private final AtomicInteger lower =
new AtomicInteger(0);
    private final AtomicInteger upper =
new AtomicInteger(0);

    public void setLower(int i) {
        //Warning:unsafe check-then-act
        if (i > upper.get())
            throw new
                IllegalArgumentException(
                    "can't set lower to " +
                     i + " > upper");
        lower.set(i);
    }

    public void setUpper(int i) {
        //Warning:unsafe check-then-act
        if (i < lower.get())
            throw new
                IllegalArgumentException(
                    "can't set upper to " + i + "
                     < lower");
        upper.set(i);
    }

    public boolean isInRange(int i) {
        return (i >= lower.get() && i <=
                upper.get());
    }
}
```

- Delegation to variables participating in an invariant

source: Java Concurrency in Practice

Publishing underlying state variables

- Can we publish a (thread safe) state variable to which we have delegated the thread safety?
  -> YES, but only if:
  - it does not participate in invariants constraining its value
  - has no invalid state transitions for its operations

Extending existing thread-safe functionality

The problem

- We are provided thread safe classes that almost meet our requirements
- Can we modify/extend/use them by adding the needed (thread safe) functionality?

Solution 1. Modify the source code of the class

- If the source code is available, modify it to add the new functionality
- Make sure the thread safe requirements of the existing class are followed
- This is the safest solution, because all the thread-safety-related issues remain addressed within the class itself

Solution 2. Extend the class

- When the source code is not available, we can extend the class if permitted
- We must make sure the class was designed to be extended
- It is more fragile than Solution 1, because:
  - the thread safety is addressed within multiple source files
  - if the base class changes its thread safe policy (e.g. changes the locks it uses), the extensions may cease to work properly

Example for Solution 2.

```
@ThreadSafe
public class BetterVector<E> extends Vector<E> {
    public synchronized boolean putIfAbsent(E x)
    {
        boolean absent = !contains(x);
        if (absent)
            add(x);
        return absent;
    }
}
```

- Vector is thread safe
- The synchronization policy for Vector is fixed and specified in its documentation, therefore is safe to extend it this way

source: Java Concurrency in Practice

Solution 3. Client-side locking

- Extend the functionality without extending the class itself
- It is more fragile than Solutions 1 and 2, as it adds thread safety handling to classes unrelated to the extended one

Incorrect example for Solution 3

- Attempt to add a put-if-absent method to a synchronized list

```
@NotThreadSafe
public class ListHelper<E> {
  public List<E> list =
     Collections.synchronizedList(new ArrayList<E>());
  ...
  public synchronized boolean putIfAbsent(E x) {
      boolean absent = !list.contains(x);
      if (absent) list.add(x);
      return absent;
 }}
```

- Extends the synchronizedList behavior
- Why is it incorrect?

source: Java Concurrency in Practice

Why is it incorrect?

- putIfAbsent synchronizes on the wrong lock! (the List implementation certainly doesn’t lock on OUR object)

Correct example for Solution 3

- Add a put-if-absent functionality in a synchronized list

```
@ThreadSafe
public class ListHelper<E> {
public List<E> list =
Collections.synchronizedList(new ArrayList<E>());
...
public boolean putIfAbsent(E x) {
synchronized (list)  {
    boolean absent = !list.contains(x);
    if (absent)
        list.add(x);
    return absent;
}}}
```

- The documentation states that the synchronized wrapper classes support client-side locking on the intrinsic lock of the wrapper collection (not the wrapped one!)

source: Java Concurrency in Practice

Solution 4. Composition

- Use composition to add the needed functionality
- The solution is better than client-side locking

Example for Solution 4

- Add a put-if-absent functionality to a list

```
@ThreadSafe
public class ImprovedList<T> implements List<T> {
private final List<T> list;
    public ImprovedList(List<T> list) { this.list = list; }
    public synchronized boolean putIfAbsent(T x) {
        boolean contains = list.contains(x);
        if (contains)
            list.add(x);
        return !contains;
    }
    public synchronized void clear() { list.clear(); }
    // ... similarly delegate other List methods
}
```

- It works even if List is not thread safe
- Assumes the client will not use the underlying (initial) list directly, only through ImprovedList

source: Java Concurrency in Practice

Rules of Engagement

for writing concurrent programs

Rules of Engagement

- Beware the mutable state
- Make all fields final, unless they need to be mutable
- Remember: immutable objects are always thread safe
- Encapsulate the state: it eases the thread safe design

Rules of Engagement

- Guard each mutable variable with a lock
- Guard all variables in an invariant with the same lock
- Hold the lock during critical compound actions
- Do not access a mutable variable without locking

Rules of Engagement

- Don’t rely on “clever” reasonings about why you shouldn’t use synchronization
- Think about thread safety from the beginning (at design time)
- If your class is not thread safe, say so (in the documentation)
- Document the details of the synchronization policy

Java API Support

Synchronized Collections

Synchronized collections

- Two types of synchronized classes:
  - Collections: Vector, Hashtable
  - Wrapper classes for other collections
- Ensure thread safety by synchronizing all their public methods
- The state is encapsulated by the classes

Wrappers

- The wrapper classes
  - are usable with several collections
  - are returned by specific factory methods in the class Collections:
    synchronizedCollection(), synchronizedList(), synchronizedMap(), synchronizedSet(), synchronizedSortedMap(), synchronizedSortedSet()
  - encapsulate the collections and synchronize them

Possible problems

- Compound actions on synchronized collections may have undesired results:

```
Vector v;
...
//in thread A:
System.out.println(v.get(v.size()-1));

...
//in thread B:
v.remove(v.size()-1);
```

- This code may throw ArrayIndexOutOfBoundsException if the last item is removed before thread A reads the data

Possible problems

- Iterations can throw ArrayIndexOutOfBoundsException:

```
for(int i=0; i<v.size(); i++) System.out.println(v.get(i));
```

- Solution: use client-side locking on v:

```
synchronized(v) {
  for(int i=0; i<v.size(); i++;
     System.out.println(v.get(i));
}
--> inefficient due to long-time locking
```

Possible problems

- Using iterators:
  - iterators are built so that they capture concurrent modifications and throw an exception (ConcurrentModificationException)
  - beware of hidden iterators! :

```
Vector v;
...
System.out.println(v); //<-- println actually iterates the
                       //vector elements
```

Concurrent Collections

Concurrent Collections

- More efficient (scalable) than synchronized collections
- Built specifically for multiple threads

Concurrent Collections

- The operations on the entire collection have weaker semantics: size(), isEmpty()...
- Client-side locking can NOT be used
- Iterators are weakly consistent:
  - can tolerate concurrent modifications
  - do not guarantee to reflect the changes in the collection after the iterator was constructed
- The collections cannot be locked for exclusive access

CopyOnWriteArrayList

- Replaces ArrayList for some concurrent contexts
- They are effectively immutable objects:
  - on each modification, a new copy of the list is created and re-published

Blocking Queues

- Several implementations of queues providing blocking put() and take() methods
- Suitable for producer-consumer problems
- Implement FIFO and priority-based policies

Blocking Queues

- They are properly synchronized so that the objects are safely published from the producers to the consumer
- The blocking methods throw InterruptedException when the calling thread was interrupted

Interruptible Methods

- The InterruptedException must be handled with care
- The code that catches it must either
  - propagate it after necessary cleanup is done (throw it again)
  - restore the interrupt status by calling Thread.currentThread().interrupt() (e.g. when throwing InterruptedException is not possible)
- Catching the exception and doing nothing is NOT recommended (unless you know what you are doing)

Synchronizers

Synchronizers

- Several primitives provided by the Java API supporting various thread synchronization needs
- A synchronizer coordinates the control flow of the threads based on its state
- The encapsulated state determines whether the calling threads block or are allowed to continue execution

Synchronizers. Latches

- Latches delay the progress of threads until a terminal state is reached
- All calling threads are blocked until the terminal state is reached
- Once reached, the latch does not change its state (remains open, threads never block again)

Usage of latches

- Waiting for initialization of resources before proceeding with the computations
- Implementing dependencies between activities -- an activity does not start until all tasks it depends on finish
- Wait until all the parties involved in a collaborative task (such as a game) are ready to go on

CountDownLatch

- The state is a counter initialized upon construction
- countdown() decrements the counter
- await() blocks the calling thread until the counter reaches 0
- Once counter is 0, it never changes value
- Binary Latch: a latch initialized with counter=1

CountDownLatch Example

```
public class TestHarness {
        public long timeTasks(int nThreads, final Runnable task)
            throws InterruptedException {
        final CountDownLatch startGate = new CountDownLatch(1);
        final CountDownLatch endGate = new CountDownLatch(nThreads);

        for (int i = 0; i < nThreads; i++) {
            Thread t = new Thread() {
                public void run() {
                    try {
                        startGate.await();
                        try {
                            task.run();
                        } finally {
                            endGate.countDown();
                        }
                    } catch (InterruptedException ignored) { }
                }
            };
            t.start();
        }

        long start = System.nanoTime();
        startGate.countDown();
        endGate.await();
        long end = System.nanoTime();
        return end-start;
    }
}
```

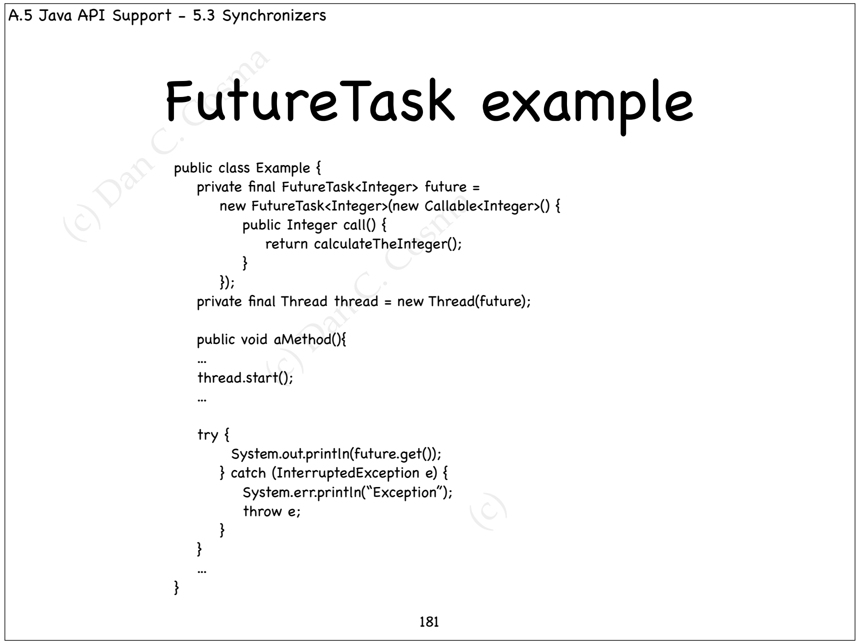
- none of the worker threads start until all of them are ready to start (startGate)

- the main thread awaits the termination of all threads efficiently (it does not have to sequentially wait for them, it blocks until the last thread ends) (endGate)

source: Java Concurrency in Practice

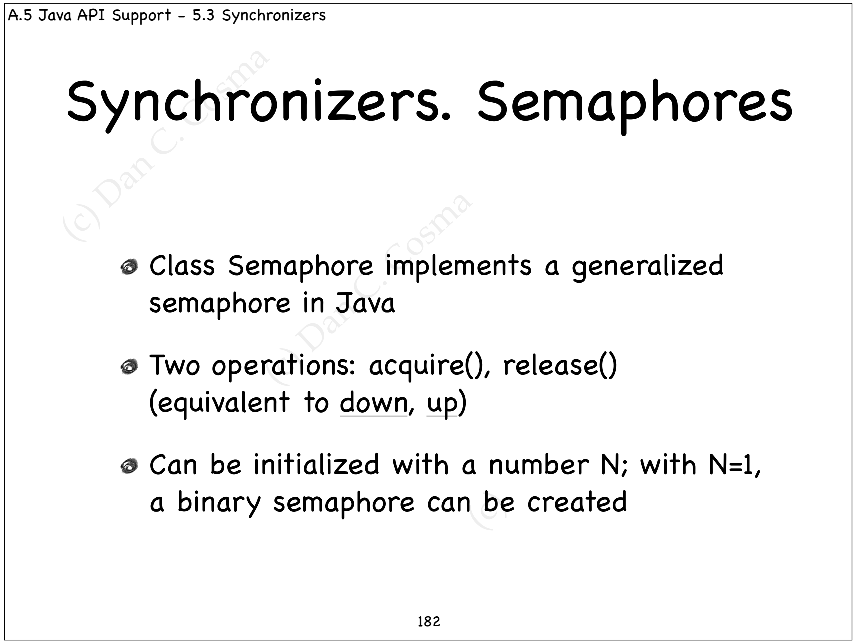
Synchronizers. FutureTask

- A class that allows for starting tasks in advance
- It acts as a latch, the open condition is the termination of the task
- The task returns a result upon termination
- Threads calling get() receive the result; if the task isn't finished, they block until the task ends
- Once the task was ended, get() never blocks



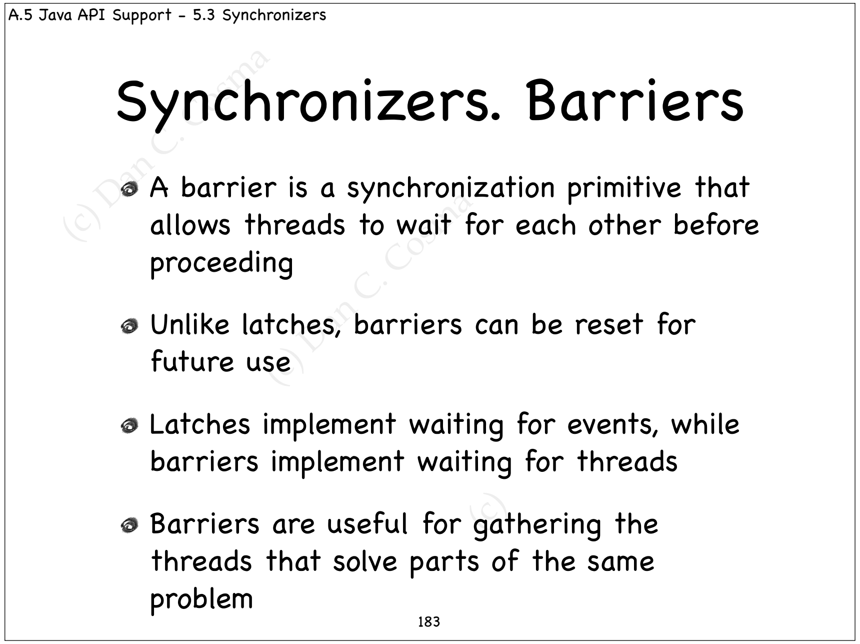
FutureTask example

```
public class Example {
    private final FutureTask<Integer> future =
        new FutureTask<Integer>(new Callable<Integer>() {
            public Integer call() {
                return calculateTheInteger();
            }
        });
    private final Thread thread = new Thread(future);

    public void aMethod(){
    ...
    thread.start();
    ...

    try {
          System.out.println(future.get());
        } catch (InterruptedException e) {
            System.err.println(“Exception”);
            throw e;
        }
    }
    ...
}
```



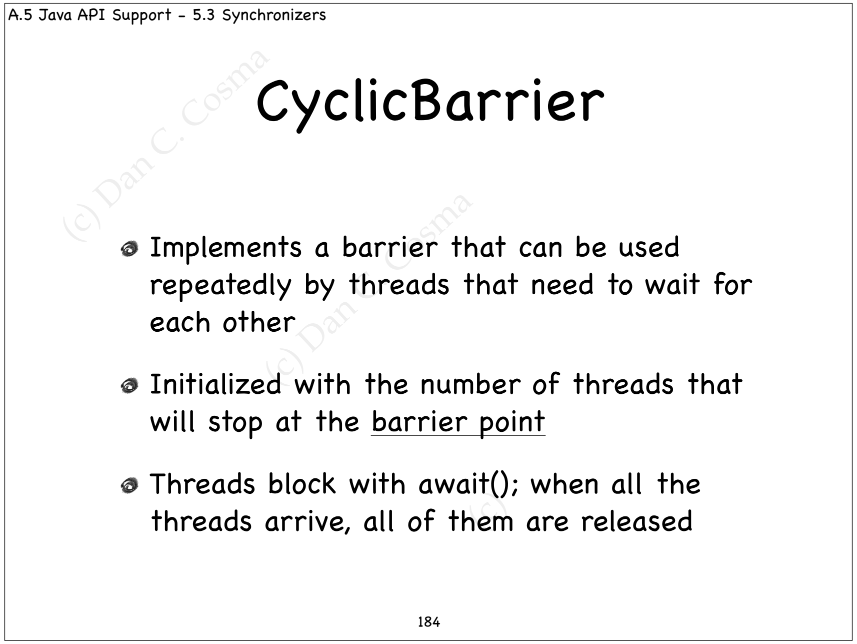
Synchronizers. Semaphores

- Class Semaphore implements a generalized semaphore in Java
- Two operations: acquire(), release() (equivalent to down, up)
- Can be initialized with a number N; with N=1, a binary semaphore can be created



Synchronizers. Barriers

- A barrier is a synchronization primitive that allows threads to wait for each other before proceeding
- Unlike latches, barriers can be reset for future use
- Latches implement waiting for events, while barriers implement waiting for threads
- Barriers are useful for gathering the threads that solve parts of the same problem



CyclicBarrier

- Implements a barrier that can be used repeatedly by threads that need to wait for each other
- Initialized with the number of threads that will stop at the barrier point
- Threads block with await(); when all the threads arrive, all of them are released

CyclicBarrier

- await() returns an unique index of arrival for each thread, which can be used in programs
- If a blocked thread is interrupted or a timeout occurs, the barrier becomes <u>broken</u>
- When barriers break, all the waiting threads receive a specific exception

CyclicBarrier

- When constructed, the barrier can be configured with an action to be done when the barrier is passed
- The action is given as a Runnable class
- The class is executed when all threads have arrived, but before they are released
- The action executes in one of the threads (usually the last one)

CyclicBarrier example

```
class Solver {
   final int N;
   final float[][] data;
   final CyclicBarrier barrier;

   class Worker implements Runnable {
     int myRow;
     Worker(int row) { myRow = row; }
     public void run() {
        while (!done()) {
          processRow(myRow);

          try {
             barrier.await();
          } catch (InterruptedException ex) {
             return;
          } catch (BrokenBarrierException ex) {
             return;
          }
        }
     }
   }
```

```
  public Solver(float[][] matrix) {
    data = matrix;
    N = matrix.length;
    barrier = new CyclicBarrier(N,
                              new Runnable() {
                                public void run() {
                                  mergeRows(...);
                                }
                              });
    for (int i = 0; i < N; ++i)
      new Thread(new Worker(i)).start();

    waitUntilDone();
  }
}
```

- Workers wait until all other workers have finished their tasks (computing a row in a matrix), at which point mergeRows() is called

source: Java API reference at java.sun.com

Task Execution

Executing tasks in threads

Task identification

- Concurrency is useful in many real-world applications
- At design time, the tasks that can or need to be executed concurrently should be clearly identified
- The designer must define the task boundary which should delimit activities that are:
  - relatively independent
  - focused on clear goals
  - contributors to a balanced execution

Examples of tasks

- Subproblems of a larger problem
  - e.g. matrix processing approached at the row or column level
- Stateless responses to client requests
  - e.g. a ‘current time’ service
- Stateful services available to clients
  - e.g. electronic e-mail access for users

Task execution

- The design must specify how the tasks will be executed at runtime so that the application
  - is responsive
  - has good throughput
  - exhibits graceful degradation at overload

Sequential execution

- The simplest method of executing the tasks, by serializing them in a single thread

```
...
while(acceptServiceRequest()) {
   processRequest();
}
...
```

Unbounded thread creation

- For each request, a new thread is created

```
...
while(acceptServiceRequest()) {
   Runnable task = new Runnable() {
       public void run() { processRequest(); }
   };
   Thread t = new Thread(task);
   t.start();
}
...
```

Consequences

- Consequences of unbounded thread creation:
  - The requests are decoupled from the main thread, allowing it to respond immediately to new clients
  - Multiple clients are served in parallel which can improve the throughput
  - The code that implements the task must be thread-safe

Disadvantages

- Drawbacks of unbounded thread creation
  - creating and managing threads takes time and processing power (OS and JVM)
  - threads consume resources (especially memory)
  - may lead to stability / scalability /security problems: the number of threads that can be run at the same time is limited

Thread Pools and the Executor Framework

Thread pools

- A thread pool is a set of threads that are used for executing a set of activities
- The pool is associated a task queue that stores the activities to be executed
- When free, a thread reads a task from the queue, and executes it
- Upon terminating the task, the thread becomes available for a new activity

Thread pools

- The size of the pool (number of threads) and the policy of task scheduling vary by thread pool design
- The behavior when threads end abruptly or are interrupted is specific to the various types of pools
- The pool must be fit (in terms of size and behavior) for the necessities of the particular application

Advantages of thread pools

- The threads are created a limited number of times and reused, independent on the number of requests
  => good performance regarding the thread management
  => adequate system resources used for the threads
  => The system is not overloaded, as the number of threads can be easily controlled

Execution Policies

- Another advantage of using thread pools: easy implementation of execution policies
- Facilitated by the submission/execution decoupling
- Execution policies specify:
  - in what threads will the tasks be executed
  - the order of task execution (FIFO, LIFO, etc.)
  - the number of concurrent/postponed tasks
  - task control (e.g. which tasks are rejected)
  - task environment control

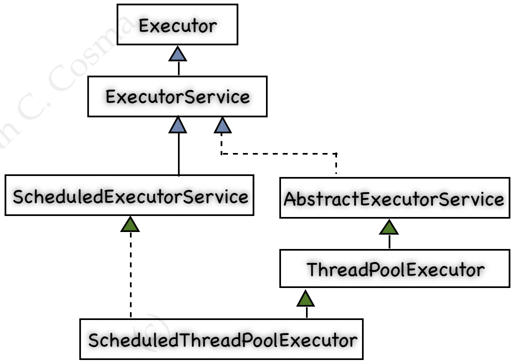
The Executor Framework

- A framework provided by the Java API for asynchronous task execution
- Decouples the task submission from the actual task execution
- Provides support for various task execution policies

The framework consists of...

- Three interfaces:
  Executor,
  ExecutorService,
  ScheduledExecutorService

- A set of Executor implementations



```
public interface Executor {
    void execute(Runnable command);
}

public interface ExecutorService extends Executor { //adds lifecycle management
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException;
    <T> Future<T> submit(Callable<T> task);
    Future<?> submit(Runnable task);
    ...
}

public interface ScheduledExecutorService extends ExecutorService {
    <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit);
    ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit);
    ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay,
                                           long period, TimeUnit unit);
    ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay,
                                              long delay, TimeUnit unit)
}
```

The Executors class

- The preferred way of creating Executors
- Provides a set of factory methods for different variants of executors:
  - newSingleThreadExecutor() -- an executor that executes a single task at a time
  - newFixedThreadPool() -- an executor that uses a fixed thread pool (the number of threads is specified)
  - newCachedThreadPool() -- an executor that uses an expandable thread pool
  - newScheduledThreadPool() -- an executor using a thread pool capable of scheduling tasks to run after a given delay or periodically
  - ...
- Directly instantiating the executor classes in the hierarchy is practical only when additional options are needed

Example

- A thread pool that executes client requests:

```
private static final Executor executor = Executors.newFixedThreadPool(50);
...
while(acceptServiceRequest()) {
   Runnable task = new Runnable() {
       public void run() { processRequest(); }
   };
   executor.execute(task);
}
...
```

Example

- A custom executor that creates one thread per task:

```
public class ThreadPerTaskExecutor implements Executor {
   public void execute(Runnable r) {
      new Thread(r).start();
   };
}
```

source: Java Concurrency in Practice

Example

- A custom executor that executes tasks sequentially, in the current thread:

```
public class WithinThreadExecutor implements Executor {
   public void execute(Runnable r) {
        r.run();
   };
}
```

Tasks that return results

The Callable Interface

- Runnable represents a task to be run, but it does not provide means for the task to return a result
- Callable solves this problem:

```
public interface Callable<V> {
   V call() throws Exception;
}
```

The Future Interface

- Represents an asynchronous task
- Provides methods allowing to
  - test whether the task is completed
  - cancel the task
  - retrieve the result

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException,
                CancellationException;
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException,
                CancellationException, TimeoutException;
}
```

Executing tasks that return results

- Use FutureTask and run it in a thread. As FutureTask implements Runnable, it can also be passed to an Executor
- Use the submit methods in an ExecutorService, e.g.: <T> Future<T> submit(Callable<T> task);
- As of Java 6, an override-able method exists in AbstractExecutorService:

```
protected <T> RunnableFuture<T> newTaskFor(Callable<T> task) {
   return new FutureTask<T>(task);
}
```

Example

- A web page renderer that consists of two tasks:
  - one that renders the text
  - one that downloads images
- The image download is done asynchronously, as a task submitted to an Executor

```
public class FutureRenderer {
    private final ExecutorService executor = ...;

    void renderPage(CharSequence source) {
        final List<ImageInfo> imageInfos = scanForImageInfo(source);
        Callable<List<ImageData>> task =
                new Callable<List<ImageData>>() {
                    public List<ImageData> call() {
                        List<ImageData> result
                                = new ArrayList<ImageData>();
                        for (ImageInfo imageInfo : imageInfos)
                            result.add(imageInfo.downloadImage());
                        return result;
                    }
                };

        Future<List<ImageData>> future =  executor.submit(task);
        renderText(source);

        try {
            List<ImageData> imageData =  future.get();
            for (ImageData data : imageData)
                renderImage(data);
        } catch (InterruptedException e) {
            // Re-assert the thread's interrupted status
            Thread.currentThread().interrupt();
            // We don't need the result, so cancel the task too
            future.cancel(true);
        } catch (ExecutionException e) {
            ...
        }
}}
```

source: Java Concurrency in Practice

Cancellation and Shutdown

The problem

- The tasks that run in parallel usually end by themselves and provide results
- The need to stop them before their normal termination may arise because of various reasons:
  - timeouts
  - errors
  - the user cancelled the operation
  - the application must shutdown

The problem

- The termination policy must deal with the following issues:
  - it must manage possibly numerous concurrent entities
  - the threads can be in various stages of accomplishing the tasks

The problem

- The cancellation or shutdown must be:
  - coordinated
    -> all threads must finish when requested
  - quick
    -> the reason for cancellation is usually important
  - reliable
    -> the cancellation policy must deal with all the necessary cleanup so that the application/task ends safely

Issues

- The concurrent entities must be designed so that they properly consider interruption
- The cancellation activities in threads must be quick
- The design must ensure that no thread remains uninformed on the cancellation event

The policy

- The cancellation/shutdown policy
  - is a property of the application's design
  - can rely on specific mechanisms provided by the software platform (operating system, virtual machine)

Considering cancellation

- The programs can be written with the cancellation as one of their features

```
public class AClassThatCanBeCancelled {
  private volatile boolean isCancelled = false;

  public void cancel() { this.isCancelled = true; }

  public void doTheWork()
  {
     while(!isCancelled)
     {
       ...
     }
  }
```

Possible problems

- The threads only check for the cancellation status at certain times (cancellation points)
- The response may be too slow
- The solution does NOT cover the case of blocked threads (waiting on blocking queues, etc.)

An example: this program may block forever

```
public class Producer implements Runnable {

volatile boolean isCancelled = false;
BlockingQueue<String> queue;

public void Producer(BlockingQueue<String> q)
{
  this.queue = q;
}

public void cancel() { isCancelled = true;}

public void run()
{
  while(!isCancelled)
  {
    String item = generateItem();
    queue.put(item);
  }
}
}
```

```
public class Consumer implements Runnable {

BlockingQueue<String> queue;

public void Consumer(BlockingQueue<String> q)
{
  this.queue = q;
}

public void run()
{
  while(itemsAreNeeded())
  {
    String item = queue.take();
    useItem(item);
  }
}
}
```

- If the producer generates items fast and blocks as the consumer asks for cancellation -> the producer will never exit the blocking method, thus never noticing the cancellation status

Java

- Java provides a mechanism fit for cancellation, based on a cooperative policy:
  - The main concept: threads can be requested to interrupt
  - The implementation: threads can choose how to respond
- A properly designed application will always respond to interruption
- If interruption is used for cancellation or shutdown the thread should do the necessary cleanup and end as soon as possible

Interruption

- Each Thread has an 'interrupted' status which is initially set on false
- The status can be set to true by specific methods in class Thread
- A thread can set the interrupt status of another thread at any time; however, the interrupted thread's behavior is dependent on the system's design

Thread

- Interruption-related methods in Thread:

```
public class Thread extends Object implements Runnable {
  ...
  public void interrupt(); // interrupts the current thread
  public static boolean interrupted(); // tests whether the current
                                       // thread has been interrupted
                                       // and clears the interrupted status
  public boolean isInterrupted(); // tests whether this thread has
                                  // been interrupted; the interrupted
                                  // status remains unchanged
  ...
}
```

What about those deprecated methods?

- The Thread class defines a set of methods that can force changes in the execution status of threads: stop(), suspend(), resume(), destroy()
- These methods were marked as deprecated relatively early in the API development
- Their usage can lead to serious concurrency-related issues

Why was stop() deprecated?

- stop() forcibly stops the execution of a thread
- If the thread is inside a monitor, the lock will eventually be released, which can lead to other threads being able to access inconsistent state (“damaged” objects)

Why is suspend() deprecated?

- suspend() suspends a thread until another thread calls resume()
- suspend() can generate deadlock:
  -> if the suspended thread owns a lock it will be not be released before resume()

Why is destroy() deprecated?

- destroy() is meant to end a thread abruptly
- Actually, destroy() was never implemented!
- The reason: it is deadlock-prone
  -> unlike suspend(), there isn't even the possibility of resuming the thread to release the lock

Are there replacements?

- No.
- However, if suspend/resume or stop functionalities are necessary they can be implemented in programs by combining
  - volatile status variables for the respective state (e.g. isSuspended, isStopped)
  - wait() and notify() for suspending/resuming, if needed (you can use them, with care!)

Back to the topic...

- The best mechanism to implement proper cancellation of threads is making use of the interruption status
- While the interruption feature was not explicitly built for cancellation, using it for other purposes is not practical

Interruption behavior

- If the target (interrupted) thread is working (not blocked) the status change will not affect its current activities
  -> The thread must explicitly test the interrupt status periodically
- If the target thread is blocked, the blocking method returns immediately and throws InterruptedException

InterruptedException

- The InterruptedException must be handled with care
- The code that catches it must either
  - propagate it after necessary cleanup is done (throw it again)
  - restore the interrupt status by calling Thread.currentThread().interrupt() (e.g. when throwing InterruptedException is not possible)
- Catching the exception and doing nothing is NOT recommended (unless you know what you are doing)

Yes, you did see this slide before ☺

An example: cancellation using interruption

```
public class Producer implements Runnable {

BlockingQueue<String> queue;

public void Producer(BlockingQueue<String> q)
{
  this.queue = q;
}

public void cancel() { interrupt(); }

public void run()
{
  while(!Thread.currentThread.isInterrupted())
  {
    String item = generateItem();
    try{
      queue.put(item);
    } catch(InterruptedException e) {
      ... //cleanup and exit
    }
  }
}}
```

```
public class Consumer implements Runnable {

BlockingQueue<String> queue;

public void Consumer(BlockingQueue<String> q)
{
  this.queue = q;
}

public void run()
{
  while(itemsAreNeeded())
  {
    String item = queue.take();
    useItem(item);
  }
}
}
```

- On interruption, the blocked method will return and throw the exception -> the producer can check the interruption status and end properly

Notes about interruption

- Interrupting a thread does not necessarily stop its current activities -- it is only a request
- Each thread has its interruption policy. Therefore, DO NOT interrupt a thread unless you know how the interruption is handled in the thread
- Only code implementing the thread's interruption policy may swallow the InterruptedException. General-purpose or library code should never do it

Implementing timeouts

The problem

- There are many cases when the execution of asynchronous tasks does not end in a timely manner, and they must be cancelled after a pre-defined time
  - tasks that perform complex calculations
  - activities that continuously monitor states (e.g. monitoring a thermal sensor)
  - tasks that log system activities
  - ...

Example of implementing a timeout

```
@ThreadSafe
public class PrimeGenerator implements Runnable {
    @GuardedBy("this")
    private final List<BigInteger> primes
            = new ArrayList<BigInteger>();
    private  volatile boolean cancelled;

    public void run() {
        BigInteger p = BigInteger.ONE;
        while (!cancelled ) {
            p = p.nextProbablePrime();
            synchronized (this) {
                primes.add(p);
            }
        }
    }

    public void cancel() { cancelled = true;  }

    public synchronized List<BigInteger> get() {
        return new ArrayList<BigInteger>(primes);
    }
}
```

```
List<BigInteger> aSecondOfPrimes() throws
InterruptedException {
    PrimeGenerator generator = new
PrimeGenerator();
    new Thread(generator).start();
    try {
        TimeUnit.SECONDS.sleep(1);
    } finally {
        generator.cancel();
    }
    return generator.get();
}
```

source: Java Concurrency in Practice

Discussion

- In the previous example the main thread does not catch the exceptions the task may throw
- This may be important, as the exceptions may show the calculations did not perform correctly

An attempt of implementing timeout

```
//This code has some issues
private static final ScheduledExecutorService cancelExec = ...;

public static void timedRun(Runnable r,
                            long timeout, TimeUnit unit) {
    final Thread taskThread = Thread.currentThread();
    cancelExec.schedule(new Runnable() {
        public void run() { taskThread.interrupt(); }
    }, timeout, unit);
    r.run();
}
```

- Explanation: the timedRun() method schedules an interruption of the current thread, then runs the run() method of the Runnable it receives
- A Scheduled Executor is used for executing the interruption

source: Java Concurrency in Practice

Discussion

- The previous example is able to catch the exceptions thrown by the task
- However, it does not follow the rule that foreign threads should not be interrupted (their interruption policy being unknown)
- If the task does not handle the interruption, it may end long after the timeout has expired (or it may even run forever)
- If the task ends before the timeout, the interrupt could go off after timedRun() returns, interrupting an unknown code

A correct solution

- Use a dedicated thread for the task
- The exceptions from the task are caught, stored and re-thrown to the client thread

```
public static void timedRun(final Runnable r,
                            long timeout, TimeUnit unit)
                            throws InterruptedException {
    class RethrowableTask implements Runnable {
        private volatile Throwable t;
        public void run() {
            try { r.run(); }
            catch (Throwable t) { this.t = t; }
        }
        void rethrow() {
            if (t != null)
                throw t;
        }
    }

    RethrowableTask task = new RethrowableTask();
    final Thread taskThread = new Thread(task);
    taskThread.start();
    cancelExec.schedule(new Runnable() {
        public void run() { taskThread.interrupt();  }
    }, timeout, unit);
    taskThread.join(unit.toMillis(timeout));
    task.rethrow();
}
```

source: Java Concurrency in Practice

Another solution

- Use Future for canceling
- The exceptions from the task are caught, and immediately re-thrown to the client thread

```
public static void timedRun(Runnable r,
                            long timeout, TimeUnit unit)
                            throws InterruptedException {
    Future<?> task = taskExec.submit(r);
    try {
        task.get(timeout, unit);
    } catch (TimeoutException e) {
        // task will be cancelled below
    } catch (ExecutionException e) {
        // exception thrown in task; rethrow
        throw e;
    } finally {
        // Harmless if task already completed
        task.cancel(true);  // mayInterruptIfRunning=true
    }
}
```

source: Java Concurrency in Practice

Non-interruptible blocking

- Some blocking methods are not interrupted by interrupt(): synchronous socket I/O, waiting to acquire an intrinsic lock, etc.
- These cases must be dealt with by using specific mechanisms
  - socket I/O: close the socket from another thread
  - instead of intrinsic locks, use the explicit Lock classes (which provide interruptible locking methods)

Example

- Non-standard canceling of socket I/O operations by overriding the interrupt() method

```
public class ReaderThread extends Thread {
    private final Socket socket;
    private final InputStream in;

    public ReaderThread(Socket socket) throws IOException {
        this.socket = socket;
        this.in = socket.getInputStream();
    }

    public void  interrupt()  {
        try {
            socket.close();
        }
        catch (IOException ignored) { }
        finally {
            super.interrupt();
        }
    }

    public void run() {
        try {
            byte[] buf = new byte[BUFSZ];
            while (true) {
                int count = in.read(buf);
                if (count < 0)
                    break;
                else if (count > 0)
                    processBuffer(buf, count);
            }
        } catch (IOException e) { /*  Allow thread to exit  */  }
    }
}
```

source: Java Concurrency in Practice

Suggestions for a good design

- To correctly design the cancellation policies, several aspects must be addressed:
  1. Use the concept of thread ownership
  -> each thread is own by a single entity (application, class, etc.)
  -> example: a thread pool owns its threads
  2. Only the owner can manipulate the thread
  -> never interrupt a thread you do not own
  3. Provide lifecycle methods (stop, suspend, cancel, etc.) in thread-owning services that can run for a longer time than their clients

Abnormal thread termination

The problem

- A thread can terminate abruptly, by throwing an unchecked exception (e.g. NullPointerException)
- The default behavior in Java is to print the stack trace on the console, and end the thread

The problem

- Not handling the unchecked exceptions (e.g. RuntimeException) can create problems in the application:
  - the console may be invisible for the user, and the exception is not noticed
  - a thread that ends abruptly may damage the state of the application -- other threads may depend on it

Running foreign code

- There are many cases when an application (or a class) runs foreign code (e.g. plugins, event handlers, etc.):
  - the code is provided through abstractions such as Runnable, Callable
  - the code may throw unchecked exceptions
  - the unchecked exceptions in the foreign code must NOT make the application fail
- The application must handle the unchecked exceptions

Example

- An implementation of a worker thread in a thread pool
- The worker catches the Throwable, and informs the thread pool that the task ended in error
- The pool may decide to end the thread or reuse it

```
...
public void run() {
    Throwable thrown = null;
    try {
        while (!isInterrupted())
            runTask(getTaskFromWorkQueue());
    } catch (Throwable e) {
        thrown = e;
    } finally {
        threadExited(this, thrown);
    }
}
...
```

source: Java Concurrency in Practice

Uncaught exception handlers

- A mechanism complementary to the explicit handling of Throwable
- Applications can implement the interface UncaughtExceptionHandler and register the implementation to the JVM
- The registration can be done at the thread level (since JDK 1.5), up to the System level

The interface

```
public interface Thread.UncaughtExceptionHandler {
    void uncaughtException(Thread t, Throwable e);
}
```

- The uncaughtException() method is invoked when the thread t terminates because of the uncaught exception e

Registering the handler

- There are three ways of registering an uncaught exception handler:

- Per-thread handler:
Thread.setUncaughtExceptionHandler(java.lang.Thread.UncaughtExceptionHandler),

- Per-thread-group handler:
ThreadGroup.uncaughtException(java.lang.Thread, java.lang.Throwable)

- The default handler:
Thread.setDefaultUncaughtExceptionHandler(java.lang.Thread.UncaughtExceptionHandler),

-> An uncaught exception is delegated to the per-thread handler; if it does not exist it is delegated upwards; if not even a default handler exists, the stack trace is printed to the console

Java Virtual Machine shutdown

JVM Shutdown

- The JVM can be shut down in two ways:
  - orderly (all threads end, System.exit(), SIGINT)
  - abruptly (Runtime.halt(), SIGKILL)

Shutdown hooks

- For an orderly shutdown, application can register shutdown hooks
- A shutdown hook is an unstarted thread that is registered through Runtime.addShutdownHook().
- The registered threads will be started by the JVM when the orderly shutdown is performed
- Shutdown hooks must be thread-safe, as they can be started concurrently

Daemon threads

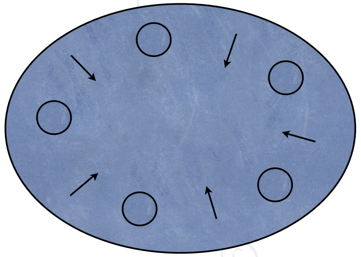
- Threads that run in background, but their execution does not influence the decision of doing an orderly shutdown
- If all other threads end, the JVM will initiate a shutdown and forcibly stop the daemon threads
- Any thread can be: normal/daemon. A thread inherits the status of the thread that created it
- All JVM internal threads are daemon threads, application threads are normal by default
- Related methods in Thread: setDaemon(boolean), isDaemon().

Finalizers

- When disposing objects, the garbage collector offers the option of calling a special method of the object: finalize()
- An object that implements this method will be able to do additional cleanup
- However, this technique should be avoided -- the preferred way of doing cleanup is through try...finally blocks

Avoiding Deadlock

A case of deadlock...



Some solutions to the Dining Philosophers problem can deadlock, as previously seen

Lock ordering deadlock

- A case that can be identified within the code
- Threads try to acquire the same locks in a different order
- Can be avoided by ordering the locking

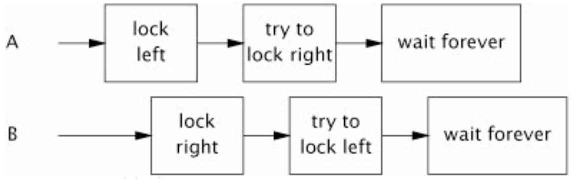
Example

```
// Warning: deadlock-prone!
public class ADeadlock {
    private final Object left = new Object();
    private final Object right = new Object();

    public void method1() {
        synchronized (left) {
            synchronized (right) {
                doSomething();
            }
        }
    }

    public void method2() {
        synchronized (right) {
            synchronized (left) {
                doSomethingElse();
            }
        }
    }
}
```

- The methods are called in distinct threads
- The locking is done on the same objects, but the order of locking in different

source: Java Concurrency in Practice

When it goes wrong...



- The above picture shows a case when the previous example deadlocks

source: Java Concurrency in Practice

...we need a solution

- Acquire the locks in the same order, in the entire program
- Issues:
  - The locks used in similar sequences must be identified throughout the program
  - The order must be maintained even when the program evolves

source: Java Concurrency in Practice

Dynamic lock ordering

- There are cases when the identification of the lock ordering problems is not easy

```
// Warning: deadlock-prone!
public void transferMoney(Account fromAccount,
                          Account toAccount,
                          DollarAmount amount)
        throws InsufficientFundsException {
    synchronized (fromAccount) {
        synchronized (toAccount) {
            if (fromAccount.getBalance().compareTo(amount) < 0)
                throw new InsufficientFundsException();
            else {
                fromAccount.debit(amount);
                toAccount.credit(amount);
            }
        }
    }
}
```

```
A: transferMoney(myAccount, yourAccount, 10);
B: transferMoney(yourAccount, myAccount, 20);
```

source: Java Concurrency in Practice

A solution

```
private static final Object tieLock = new Object();

public void transferMoney(final Account fromAcct,
                          final Account toAcct,
                          final DollarAmount amount)
        throws InsufficientFundsException {
    class Helper {
        public void transfer() throws InsufficientFundsException {
            if (fromAcct.getBalance().compareTo(amount) < 0)
                throw new InsufficientFundsException();
            else {
                fromAcct.debit(amount);
                toAcct.credit(amount);
            }
        }
    }
```

```
int fromHash = System.identityHashCode(fromAcct);
    int toHash = System.identityHashCode(toAcct);

    if (fromHash < toHash) {
        synchronized (fromAcct) {
            synchronized (toAcct) {
                new Helper().transfer();
            }
        }
    } else if (fromHash > toHash) {
        synchronized (toAcct) {
            synchronized (fromAcct) {
                new Helper().transfer();
            }
        }
    } else {
        synchronized (tieLock) {
            synchronized (fromAcct) {
                synchronized (toAcct) {
                   new Helper().transfer();
                }
            }
        }
    }}
```

- The system-generated (Object) hash code is used as an ad-hoc ordering for the locks
- Plan B: if the hash code doesn't help, the locking sequence is protected by a third lock (the last else statement, makes the compound acquiring atomic)

source: Java Concurrency in Practice

Deadlock Between Cooperating Objects

```
/ Warning: deadlock-prone!
class Taxi {
    @GuardedBy("this") private Point location, destination;
    private final Dispatcher dispatcher;

    public Taxi(Dispatcher dispatcher) {
        this.dispatcher = dispatcher;
    }

    public synchronized Point getLocation() {
        return location;
    }

    public synchronized void setLocation(Point location) {
        this.location = location;
        if (location.equals(destination))
            dispatcher.notifyAvailable(this);
    }
}
```

```
class Dispatcher {
    @GuardedBy("this") private final Set<Taxi> taxis;
    @GuardedBy("this") private final Set<Taxi> availableTaxis;

    public Dispatcher() {
        taxis = new HashSet<Taxi>();
        availableTaxis = new HashSet<Taxi>();
    }

    public synchronized void notifyAvailable(Taxi taxi) {
        availableTaxis.add(taxi);
    }

    public synchronized Image getImage() {
        Image image = new Image();
        for (Taxi t : taxis)
            image.drawMarker(t.getLocation());
        return image;
    }
}
```

source: Java Concurrency in Practice

Analysis

- A thread calling setLocation() locks Taxi then Dispatcher
- A thread calling getImage() locks Dispatcher then Taxi

  => lock ordering deadlock

A Solution

```
@ThreadSafe
class Taxi {
    @GuardedBy("this") private Point location, destination;
    private final Dispatcher dispatcher;
    ...
    public synchronized Point getLocation() {
        return location;
    }

    public void setLocation(Point location) {
        boolean reachedDestination;
        synchronized  (this) {
            this.location = location;
            reachedDestination =
location.equals(destination);
        }
        if (reachedDestination)
            dispatcher.notifyAvailable(this);
    }
}
```

```
@ThreadSafe
class Dispatcher {
    @GuardedBy("this") private final Set<Taxi> taxis;
    @GuardedBy("this") private final Set<Taxi> availableTaxis;
    ...
    public synchronized void notifyAvailable(Taxi taxi) {
        availableTaxis.add(taxi);
    }

    public Image getImage() {
        Set<Taxi> copy;
        synchronized  (this) {
            copy = new HashSet<Taxi>(taxis);
        }
        Image image = new Image();
        for (Taxi t : copy)
            image.drawMarker(t.getLocation());
        return image;
    }
}
```

- A Call to a method without locks held = "Open Call"
- Use the locking to protect only the shared resources

source: Java Concurrency in Practice

Synchronizers. Explicit locks*

*This is actually a part of Chapter 5, Section 3

When synchronized is not enough

- Implicit locking has some limitations
  - There is no way to back off from attempting to acquire an already held lock
  - Timeouts for acquiring cannot be specified
  - The blocking for acquiring a lock can not be interrupted
  - The acquiring and release is limited to structured blocks (e.g., you cannot acquire a lock in a method and release it in another)

Explicit locks

- Alternatives to implicit locking
- Provided by the Java API since JDK 1.5
- Enable advanced features regarding locking

The Lock interface

```
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long timeout, TimeUnit unit)
        throws InterruptedException;
    void unlock();
    Condition newCondition();
}
```

ReentrantLock

- Implements Lock
- Provides the same mutual exclusion and visibility traits as the implicit locking
- Adds a few features: timeouts, polled locking, interruptible locking, etc.

Using ReentrantLock

- The most important issue: the lock MUST be released in a finally block! (reason: exceptions may leave the lock held)

```
Lock lock = new ReentrantLock();
...
lock.lock();
try {
    // update object state
    // catch exceptions and restore invariants if necessary
} finally {
    lock.unlock();
}
```

Example: Timeout

```
public boolean trySendOnSharedLine(String message,
                                   long timeout, TimeUnit unit)
                                   throws InterruptedException {
    long nanosToLock = unit.toNanos(timeout)
                     - estimatedNanosToSend(message);
    if (!lock.tryLock(nanosToLock, NANOSECONDS))
        return false;
    try {
        return sendOnSharedLine(message);
    } finally {
        lock.unlock();
    }
}
```

source: Java Concurrency in Practice

Example: using polling to avoid lock-ordering deadlock

- The sleep time has a random component to minimize the probability of livelock

```
public boolean transferMoney(Account fromAcct, Account toAcct, DollarAmount amount,
                             long timeout, TimeUnit unit)
        throws InsufficientFundsException, InterruptedException {
    long fixedDelay = getFixedDelayComponentNanos(timeout, unit);
    long randMod = getRandomDelayModulusNanos(timeout, unit);
    long stopTime = System.nanoTime() + unit.toNanos(timeout);

    while (true) {
        if (fromAcct.lock.tryLock()) {
            try {
                if (toAcct.lock.tryLock()) {
                    try {
                        if (fromAcct.getBalance().compareTo(amount)
                                < 0)
                            throw new InsufficientFundsException();
                        else {
                            fromAcct.debit(amount);
                            toAcct.credit(amount);
                            return true;
                        }
                    } finally {
                        toAcct.lock.unlock();
                    }
                }
            } finally {
                fromAcct.lock.unlock();
            }
        }
        if (System.nanoTime() > stopTime)
            return false;
        NANOSECONDS.sleep(fixedDelay + rnd.nextLong() % randMod);
    }
}
```

source: Java Concurrency in Practice

Example: interruptible lock acquisition

```
public boolean sendOnSharedLine(String message)
        throws InterruptedException {
    lock.lockInterruptibly();
    try {
        return cancellableSendOnSharedLine(message);
    } finally {
        lock.unlock();
    }
}

private boolean cancellableSendOnSharedLine(String message)
    throws InterruptedException { ... }
```

source: Java Concurrency in Practice

Performance and Scalability

Two coordinates

- Performance: the amount of work done with a given set of resources
  - Resource: CPU, memory, bandwidth, etc.
  - an activity can be bound to a resource (as the limiting factor)
- Scalability: the ability to improve throughput or capacity when new resources are added

Performance vs. Scalability

- Performance=how fast; Scalability=how much
- The two aspects are separate, even at odds
  - To accomplish scalability through parallelism, the individual tasks may have to do more work than their single-threaded versions
  - Optimizations for performance may actually be bad for scalability

How fast is my program?

- Optimizations for performance must be made with care -- ALWAYS consider their possible side effects
  - optimizations can lead to concurrency bugs
- Suggestions:
  1. Design the system properly, considering the goals and long-term challenges
  2. Make the optimizations only afterwards
  3. Be smart when choosing the parts to be optimized: don't guess, measure!
  4. Don't trade safety for performance

Amdahl's Law

- The main source of skepticism regarding the viability of parallelism
- Introduced by Gene Amdahl in 1967
- Essentially states that the speedup gained by adding parallel processing power is limited

Amdahl's Law

$$\text{speedup} \le \frac{1}{F + \frac{1 - F}{N}}$$

- N = number of processors
- F = fraction of the program that must be executed sequentially
- 1 - F = the fraction that can be parallelized
- N -> ∞, speedup -> 1/F

Interpretation

- Examples:
  - F=0.5, Max. speedup = 2
  - F=0.1, N=10, Max. speedup = 5.3
  - F=0.1, N=100, Max. speedup = 9.2

Limitations

- The Amdahl's law does not consider all the realities in parallel/concurrent computing
  - The cumulated cache size grows with the number of processors => higher performance
  - When the problem scales up, the relative sequential fraction usually decreases
  - Processors are not only used for scaling a single problem: they can execute many independent tasks (e.g. multiple programs)

Thread costs

- Sources that add to the cost of multithreading:
  - Context switching
    - CPU time for the JVM and OS
    - Flushing cached data
  - Synchronization
    - uncontended: low cost, optimized by the JVM
    - contended: higher, depending on the type of synchronization: blocking/non-blocking, granularity of locking, memory bus traffic

Reducing lock contention

- The main negative impact on scalability: exclusively locking resources
- Reducing lock contention
  - Hold locks for a short time
  - Minimize the frequency of locking
  - When possible, use other mechanisms than exclusive locking

Narrowing Lock Scope

```
@ThreadSafe
public class AttributeStore {
    @GuardedBy("this") private final Map<String, String>
            attributes = new HashMap<String, String>();

    public synchronized  boolean
userLocationMatches(String name, String regexp) {
        String key = "users." + name + ".location";
        String location = attributes.get(key);
        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```

- This is better ---->

```
@ThreadSafe
public class BetterAttributeStore {
    @GuardedBy("this") private final Map<String, String>
            attributes = new HashMap<String, String>();

    public boolean userLocationMatches(String name, String regexp) {
        String key = "users." + name + ".location";
        String location;
        synchronized (this) {
            location = attributes.get(key);
        }
        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```

source: Java Concurrency in Practice

Lock Splitting

```
@ThreadSafe
public class ServerStatus {
    @GuardedBy("this") public final Set<String> users;
    @GuardedBy("this") public final Set<String> queries;
    ...
    public synchronized void addUser(String u) { users.add(u); }
    public synchronized void addQuery(String q) { queries.add(q); }
    public synchronized void removeUser(String u) {
        users.remove(u);
    }
    public synchronized void removeQuery(String q) {
        queries.remove(q);
    }
}
```

- This is better ---->

```
@ThreadSafe
public class ServerStatus {
    @GuardedBy("users") public final Set<String> users;
    @GuardedBy("queries") public final Set<String> queries;
    ...
    public void addUser(String u) {
        synchronized  (users) {
            users.add(u);
        }
    }

    public void addQuery(String q) {
        synchronized  (queries) {
            queries.add(q);
        }
    }
    // remove methods similarly refactored to use split locks
}
```

source: Java Concurrency in Practice

Lock Striping

- Find partitions within the code that can be locked with (a variable number of) different locks (i.e., find independent partitions in the set of guarded resources, if possible)
- Example: ConcurrentHashMap: an array of 16 locks
- Not all operations can be partition locked -> example: some need to lock the entire collection: need to acquire all 16 locks

source: Java Concurrency in Practice

Distributed Software Systems

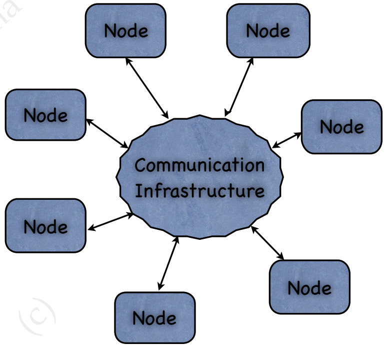
Definitions

- "a collection of independent computers that appears to its users as a single, coherent system" [TS01]
- an arbitrary number of processing elements running at different locations, interconnected by a communication system [Wu99]

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. "Distributed Systems: Principles and Paradigms." Prentice Hall, 2001.
[Wu99] Jie Wu. "Distributed Systems Design." CRC Press LLC, 1999.

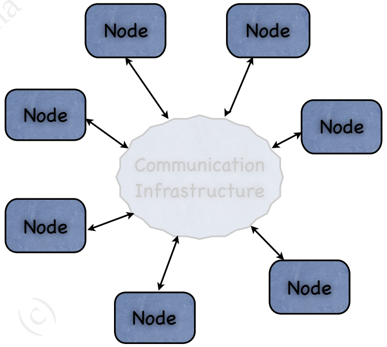
Distributed Systems

- Multiple processing units running in nodes situated at different locations
- Communication is done via an infrastructure
- The architecture is heterogenous

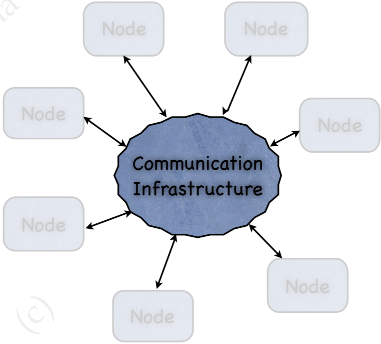


Nodes

- The system components running in nodes:
  - are independent programs that have dual functionality:
    - local
    - network-aware
  - can be written in various languages / can run on different platforms

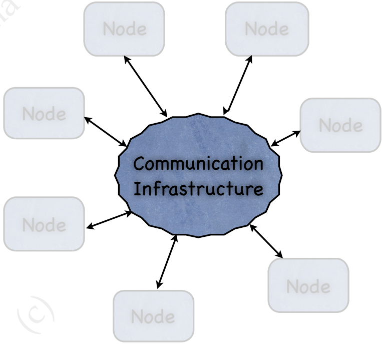


Communication Infrastructure



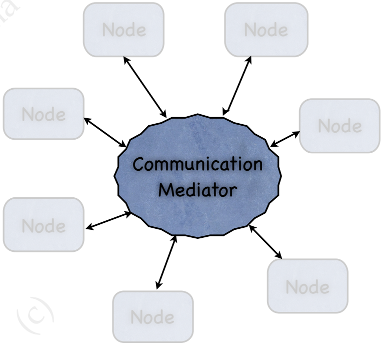
- Handles the data transmission and event notification over the network
- Is available through libraries, language constructs or platform-specific services

Communication Infrastructure



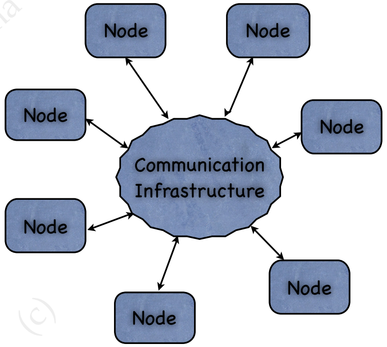
- Is built to hide the communication details, at different levels of abstraction
- Is directly related to technological concerns

Communication Infrastructure



- We call it Communication Mediator

It's a SYSTEM



- The components, however loosely coupled, work together for a common goal, and represent parts of the same system

Technological concerns

- The communication technology is central to distributed software systems

Technology

- For distributed systems, the technology influences the design-time decisions
  -> it imposes a set of constraints regarding
  - system architecture
  - system implementation (coding rules, patterns, conventions)

Technology constraints

- Specific to the type of Communication Mediator used
- Described as a set of rules with various degrees of importance
- May imply important limitations in the design- or implementation-related choices
  -> e.g. some technologies discourage the usage of threads

Communication Technologies

Protocol Stacks

- Describe facilities included in the modern operating systems
- Support network communication at the application level
- Dependent on a layered model describing the various types of concerns addressed
- Each layer defines a communication protocol

The TCP/IP protocol stack

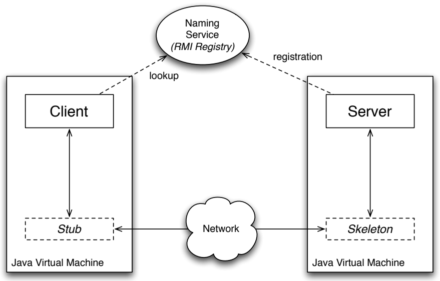
- Network access layer: transmission of the data as datagrams to a remote host
- Internet layer: defines the network as interconnected subnetworks, deals with routing. Defines the IP address
- Transport layer: communication channels, error control, sequence of data arrival, etc.
- Application layer: protocols used by the application -- e.g. FTP, HTTP, SSH, etc.
- (usually) The main primitive for programs: the socket

Application
Transport
Internet
Network Access

Constraints: specific sequence of creating/using the sockets

Remote Method Invocation (Java)

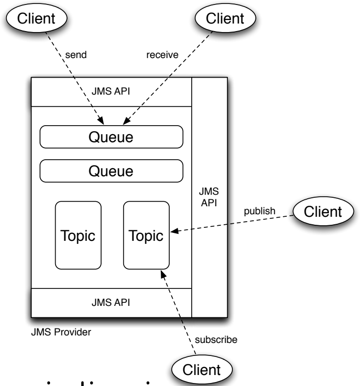
- Uses specific language constructs
- Hides the communication by providing natural ways of remote communication



Constraints:
- specific interfaces extending java.rmi.Remote describing the services; servers must implement them
- specific connection / registration API calls

Messaging systems

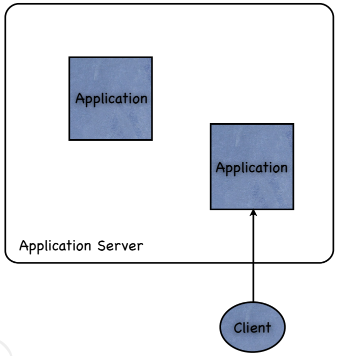
- Example: Java Message Service
- An infrastructure that mediates communication via messages
- Provides point-to-point and publish-subscribe models



Constraints:
- the design must model the communication via message channels
- specific calls for accessing the message infrastructure

Application Servers

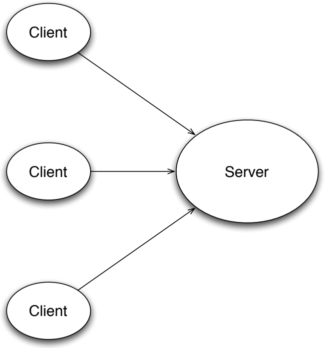
- Provide an environment for running the application
- Applications run inside the application server (hence it is sometimes called container)
- Applications are provided complex features (transactions, persistency, distribution, etc.)
- Constraints: applications are strictly limited to specific rules



Distributed Architectures

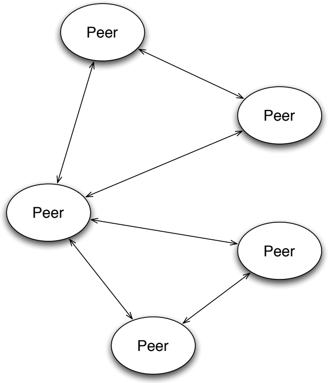
Client-server

- The most common architecture
- Server: provides a set of services
- Client: uses the services
- The client initiates the communication
- Usually clients are lighter than servers



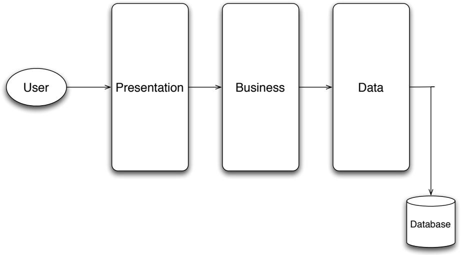
Peer-to-peer

- The components are balanced -- all play both client and server roles



Three Tier

- Largely used in modern enterprise systems
- Presentation: interacts with the user
- Logic (Business): the main system functionality (e.g. algorithms)
- Data: models the data used by Business



Concurrency

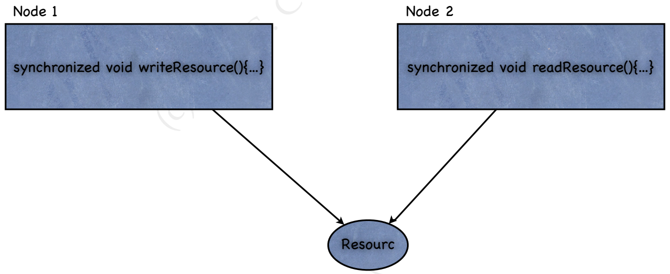
- Distributed software systems are inherently concurrent
- This trait comes from two sources:
  - the components: multiple interacting entities
  - service-related concurrency

Components acting together

- The components dispersed over the network communicate to each other, share resources, etc.
- They must coordinate their actions, as much as any multithread system would have to
- The coordination and synchronization is more difficult than in local systems
  - the reason: they don't work in the same environment

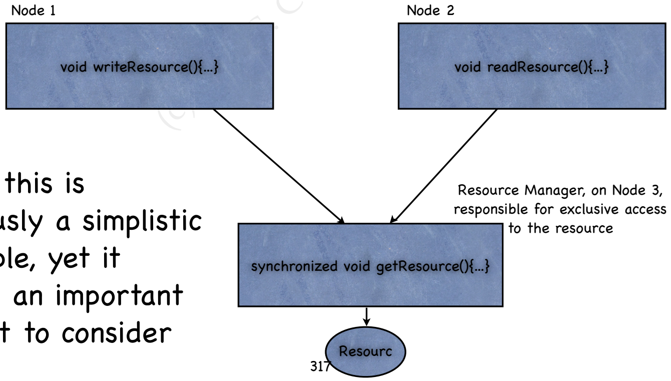
Does synchronized help?

- NO...


Node 1
synchronized void writeResource(){...}
Node 2
synchronized void readResource(){...}
Resourc

Does synchronized help?

- … and yes (sometimes)


Node 1
Node 2
void writeResource(){...}
void readResource(){...}
Resource Manager, on Node 3, responsible for exclusive access to the resource
synchronized void getResource(){...}
Resourc

Note: this is obviously a simplistic example, yet it shows an important aspect to consider

Service-related concurrency

- How many threads are in this program?

```
public MyClass {

    ...

    public int doTheWork(Collection parameters) {
      ...
    }

    ...
}
```

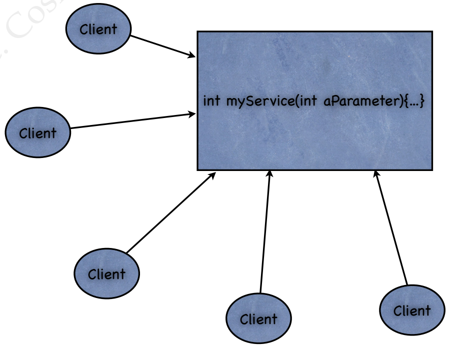
Service-related concurrency

- How many threads are in this program?

```
public MyServer implements MyRemoteInterface extends UnicastRemoteObject {

    MyClass cls;

    public MyServer(...) {
      MyClass cls = new MyClass();
    }

    public int myService(int aParameter) throws RemoteException {
      ...
      cls.doTheWork(c);
      ...
    }

    ...
}
```

How stuff runs

- Multiple clients can call the service at the same time



The answer

- It depends on the technology, BUT:
  - in most cases, a client call translates into a new thread in the server
  - this is true in virtually all server environments: RMI, CORBA, EJB, ...
- Therefore,
  - the services must be designed to be thread-safe

Message Infrastructures (Message-Oriented Middleware)

Java Message Service

Java Message Service (JMS)

- A specification that enables the implementation of message services in the Java environment
- JMS in not a service in itself, it is only adhered to by particular implementations
- The implementations (the actual services) are called JMS providers

Messaging system

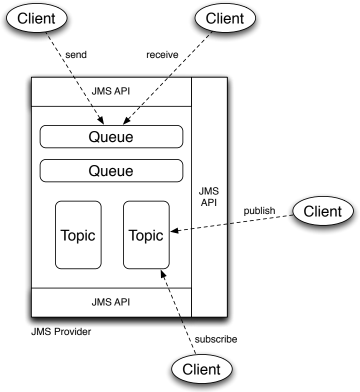
- A peer-to-peer facility enabling clients to send and receive messages to each other
- The messages are sent to an agent that intermediates the communication
- A messaging system enables loosely coupled communication between the components (senders and receivers)
- Note: messaging systems are NOT e-mail or chat applications! They deal with the communication between software components

Messages

- The applications communicate by passing messages to each other
- A message is a structured data entity that basically consists of
  - a header
  - properties (optional, JMS)
  - body

One architecture...

- A JMS provider provides two types of resources (“destinations”)
  - message queues
  - topics



... two messaging domains

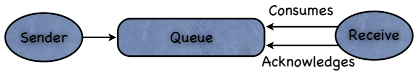
- JMS supports the two main messaging models:
  - Point-to-point
  - Publish-subscribe

Point-to-point

- Gravitates around the concept of message queues
- Senders send messages to a specific queue, thus specifying the intended receiver
- Receivers monitor their respective queues and consume the messages

Point-to-point

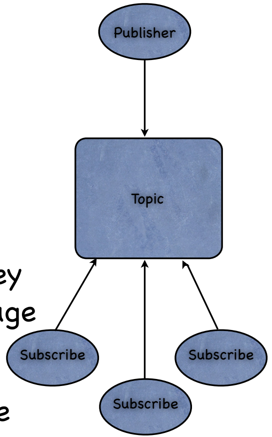
- A message is consumed only by one receiver
- The sender does not wait for the receiver
- The receiver acknowledges the successful processing of the message



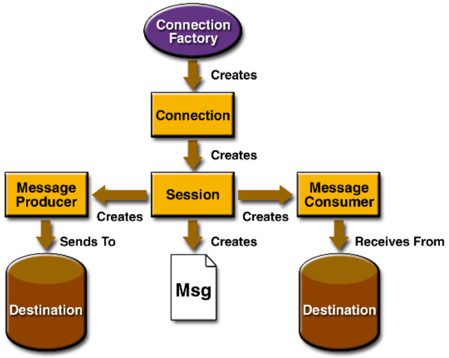
Messages can be consumed both synchronously and asynchronously

Publish-subscribe

- The message is sent to a topic by a publisher client (the equivalent of a sender)
- Multiple receivers, called subscribers may consume the message
- Subscribers specify the messages they are interested in, by describing message selectors
- The message consumption can be done both synchronously and asynchronously



JMS API Programming Model



source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

Administered objects

- Connection Factories and Destinations
- Are managed administratively, rather than programatically
- The administrative details vary from vendor to vendor (providers)
- The access to the resources is done through portable interfaces
-> clients are easily adapted to different providers

Connection Factories

- Create a connection with a JMS provider
- Two types defined in J2EE:
  - QueueConnectionFactory
  - TopicConnectionFactory

Creating and connecting to the factories

```
$ j2eeadmin -addJmsFactory jndi_name queue
$ j2eeadmin -addJmsFactory jndi_name topic
```

```
Context ctx = new InitialContext(); //get the JNDI context; searches
                //the classpath for a vendor-specific jndi.properties file

QueueConnectionFactory queueConnectionFactory =
  (QueueConnectionFactory) ctx.lookup("QueueConnectionFactory");

TopicConnectionFactory topicConnectionFactory =
  (TopicConnectionFactory) ctx.lookup("TopicConnectionFactory");
```

source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

JMS Destinations

- A Destination specifies the target/source of the messages: queues or topics
- Destinations are created through administration:
  ```
  j2eeadmin -addJmsDestination queue_name queue
  j2eeadmin -addJmsDestination topic_name topic
  ```
- Clients can connect using the standard API:
  ```
  Queue myQueue = (Queue) ctx.lookup("MyQueue");
  Topic myTopic = (Topic) ctx.lookup("MyTopic");
  ```

source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

Connections

- Represent the connection with the JMS provider
- Two types: QueueConnection, TopicConnection

```
QueueConnection queueConnection =
  queueConnectionFactory.createQueueConnection();
...
queueConnection.close();


TopicConnection topicConnection =
  topicConnectionFactory.createTopicConnection();
...
topicConnection.close();
```

source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

Sessions

- A session represents a single-threaded context that produces or consumes messages
- Provides support for transactions
- Serializes the execution of message listeners
- Two types: QueueSession, TopicSession

```
TopicSession  topicSession = topicConnection.createTopicSession(false,
Session.AUTO_ACKNOWLEDGE);//non-transacted, automatic
                                    //message acknowledgement
```

source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

Message Producers

- Produce messages that are sent to a Destination
- Two types: QueueSender, TopicPublisher

```
QueueSender queueSender = queueSession.createSender(myQueue);
TopicPublisher topicPublisher = topicSession.createPublisher(myTopic);

...
queueSender.send(message);
...
topicPublisher.publish(message);
...
```

source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

Message Consumers

- An object capable of receiving messages
- Two types: QueueReceiver, TopicSubscriber
- The message consumption can be done:
  - synchronously
  - asynchronously
- Topic subscribers can be made durable (can receive messages that occurred when they were inactive)

Synchronous message consumption

```
QueueReceiver queueReceiver = queueSession.createReceiver(myQueue);

TopicSubscriber topicSubscriber = topicSession.createSubscriber(myTopic);

queueConnection.start();
Message m = queueReceiver.receive();

topicConnection.start();
Message m = topicSubscriber.receive(1000); // time out after a second
```

- Messages are not delivered until the connection is started

source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

Asynchronous message consumption

- To receive messages asynchronously, the application can define message listeners
- A listener implements the MessageListener interface:

```
public interface MessageListener {
   public void onMessage(Message message);
}
```

- The listener is associated with a consumer

```
TopicListener topicListener = new TopicListener();
topicSubscriber.setMessageListener(topicListener);
```

source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

Message Selectors

- Can be used for filtering the messages that arrive to a consumer
- The filtering is done by the JMS provider, not by the application
- The selectors are specified as statements in a subset of SQL92 conditional expression syntax
- Selectors can be passed as arguments to the createReceiver, createSubscriber, and createDurableSubscriber methods

Messages

- A message consists of: header, properties, body
- There are 5 types of messages defined by the API:
  - TextMessage: the body is a text (e.g. XML)
  - MapMessage: a set of name/value pairs
  - BytesMessage: a stream of bytes
  - StreamMessage: a stream of primitive Java values, filled and read sequentially
  - ObjectMessage: a Serializable object

source: Sun JMS Tutorial, http://java.sun.com/products/jms/tutorial

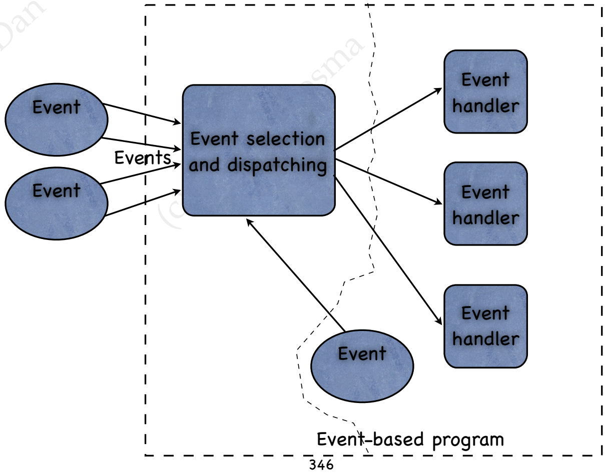
Part B. Event-Based Programming

Introduction

Definition

- A programming paradigm where the flow of the program is determined by the occurrence of events
- The programs are concerned with two main tasks:
  - Event detection
  - Event handling
- There is (usually) no main() program section, no single entry point in the program

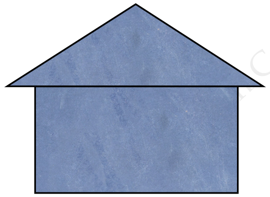
Architectural overview



Event sources

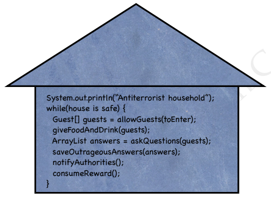
- The events can be generated by various types of sources:
  - User interfaces
  - Hardware sensors
  - External devices
  - Processes and threads
  - Operating system components

Example: Monitoring your home



- A modern home...

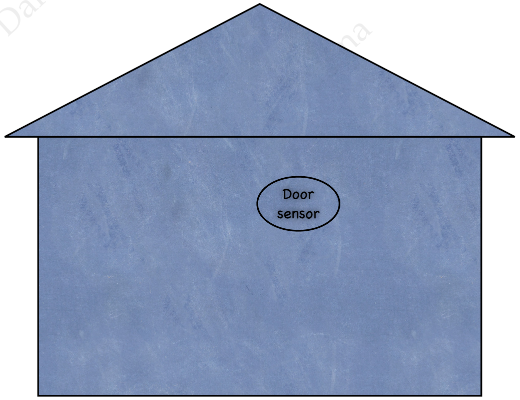
Example: Monitoring your home



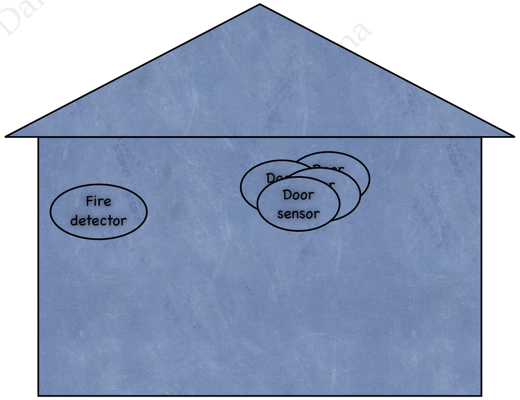
```
System.out.println("Antiterrorist household");
while(house is safe) {
  Guest[] guests = allowGuests(toEnter);
  giveFoodAndDrink(guests);
  ArrayList answers = askQuestions(guests);
  saveOutrageousAnswers(answers);
  notifyAuthorities();
  consumeReward();
}
```

- A modern home...
- ...that runs software

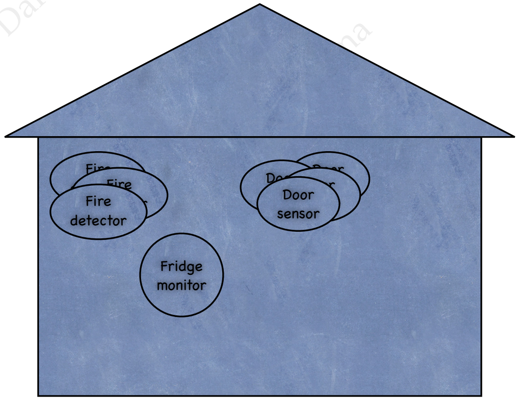
Event Sources



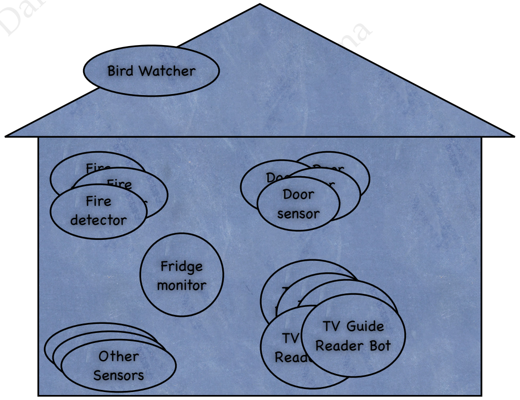
Event Sources



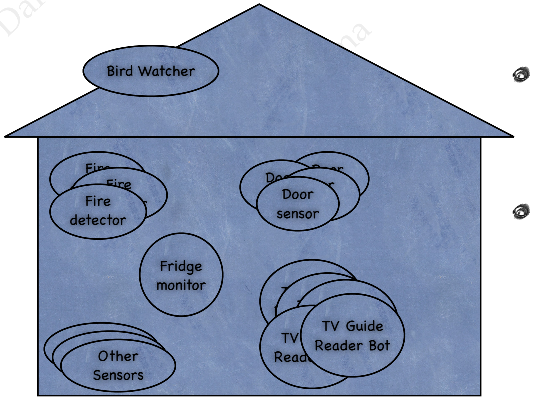
Event Sources



Event Sources



Event Sources



- The event sources must be identified
- They usually generate useful, asynchronous events

Note:

- We are talking about software, therefore:
  - The event sources are, for example, the programs controlling/monitoring the sensors
  - The events can be modeled in various ways: messages, signals, constants used as parameters, etc.

Events

- Fire Detector:
  FIRE_DETECTED
  FIRE_EXTINGUISHED
- Fridge Monitor:
  DOOR_OPEN
  DOOR_CLOSED
  BEER_ALERT
  NO_MORE_BEER
- Door Sensor:
  DOOR_OPEN
  DOOR_CLOSED
  DOOR_SMASHED
- TV Guide Reader Bot:
  SHOW_STARTS
  SHOW_WILL_START_SOON
  SHOW_ENDS
  TV_GUIDE_SHREDDED
- Bird Watcher:
  BIRD_ON_ROOF
  BIRD_SCARED
- Other sensors
  NIGHT_DETECTED
  DAY_DETECTED
  PERSON_ASKS_QUESTION
  ...

Event selection and dispatching

- An example of direct, hardcoded dispatcher

```
while((event=readEvent()) != null) {
  switch(event) {
    case FIRE_DETECTED, FIRE_EXTINGUISHED:
       callFireHandler(event); break;
    case BIRD_ON_ROOF:
       popUpScarecrow(scarecrowObject, event);
    case DOOR_OPEN, DOOR_CLOSED:
       if(event.source() == Sources.FRIDGE)
         callFridgeAlerter(event);
       else
         callDoorHandler(event);
       break;
    ...
  }
}
```

Event handler registration

- An example where handlers are dynamically registered to the dispatcher, allowing for flexibility

```
public interface Registration {
   void registerHandler(String eventType, Handler handler);
}
```

```
public interface Handler {
   void handleEvent(Event event);
}
```

```
public class Dispatcher implements Registration {
   ...
   private void mainLoop() {
      while((event=readEvent()) != null) {
         handlers.get(event.type()).handleEvent(event);
      }
   }
}
```

Event Handlers

- The event handlers must be small, and return quickly

```
public class FridgeHandler implements Handler {
   void handleEvent(Event event) {
     switch(event.getEventId()) {
        case BEER_ALERT:
          startBeerBuyerThread(); break;
        case DOOR_OPEN:
          startChimeAlarmThread(); break;
        ...
        case NO_MORE_BEER:
          startSeriousAlarmThread(); break;
        ...
     }
   }
}
```

```
public class BirdWatcherHandler implements Handler {
   void handleEvent(Event event) {
     switch(event.getEventId()) {
        case BIRD_ON_ROOF:
           startBirdAlarm();
           startScarecrowThread();
           break;
        case BIRD_SCARED:
           turnOffBirdAlarm();
           break;
        ...
     }
   }
}
```

End of example

X

Events and notifications

- Definitions (T. Faison):
  - Event: a detectable condition that can trigger a notification
  - Notification: event-triggered signal sent to a runtime-defined recipient
- Event: the cause; Notification: the effect

source: Event-Based Programming. Taking Events to the Limit

The event

- "A detectable condition..."
  - Not all conditions detectable in a program qualify as events
  - The condition's nature must be so that it necessarily causes a notification to a runtime recipient
  - The events occur mainly asynchronously, and drive the flow of the program

The notification

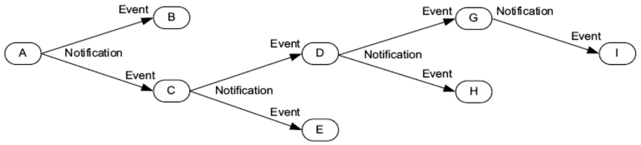
- The notifications are the carriers of events to the intended recipient
- There are two types of notifications:
  - through data transfer
  - through transfer of execution control

The notification

- Data transfer:
  - A message created by the sender and sent
    - directly to the receiver, or
    - to a resource that is shared with the receiver (pipe, shared memory, network connection, OS service, etc.)
- Transfer of execution control:
  - Local or remote procedure/method call

Chains of notifications

- The notifications can trigger new events
- Chains of events and notifications can form during runtime



source: Event-Based Programming. Taking Events to the Limit

Programming support

- Object-oriented languages and APIs support notification through specific mechanisms
  - Java: typed event listeners
- Notification is also supported by separate services:
  - Message-oriented infrastructures (e.g. JMS)
  - Operating system services (e.g. signals)

Terminology

- The entity that detects events:
  -> event publisher, event source, sender
- The entity that receives notifications:
  -> event subscriber, event handler, notification target, notification receiver, receiver
- The act of sending notifications
  -> sending the notification, firing the event

Terminology

- There are many cases where the notification (the act of informing the recipient) is assimilated (as a term) with the event (the condition that occurred)
- Usually, the event is part of the notification, as its payload
  -> e.g., the event is described in the message content, or in the method parameters

Event subscription

Event Subscription

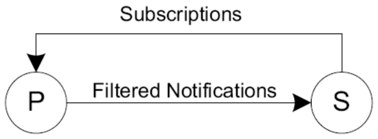
- The process of linking the sender of events with the receiver
- The subscriber declares
  - it needs the future notifications from the sender
  - the types of events it is interested in
- The subscription process is done at runtime

Event Subscription

- A subscriber can subscribe to multiple types of events from the same publisher
- An event can have multiple subscribers
- There can be events with no subscribers

Subscription types

- Direct Delivery
  -> The subscriber and publisher are linked directly, both for the subscription process, and for firing of events



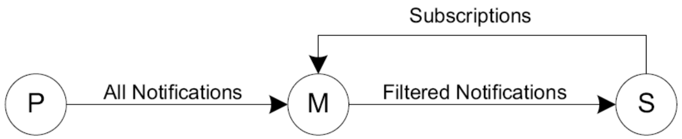
source: Event-Based Programming. Taking Events to the Limit

Subscription types

- Indirect delivery
  - Useful in systems where the number of subscriptions is large, or the process of notification is costly
  - The notification task is delegated to a middleware system
  - The middleware system is responsible for filtering and routing the notifications

Subscription types

- Indirect delivery
  - The subscriber interacts only with the middleware, both for subscription, and with the notification



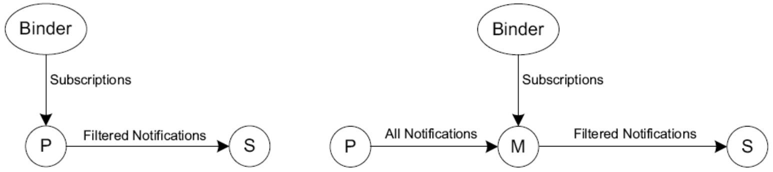
source: Event-Based Programming. Taking Events to the Limit

Binder agents

- There are cases when the subscriber must be kept uncoupled with a sender or a middleware system
- Separate binder agents can be used, with the purpose of making the subscriptions on behalf of the receiver
- The binder is coupled with both the publisher and subscriber
- The subscriber can be decoupled from both the publisher and the binder

Binder agents

- Binder agents can be used for both subscription types



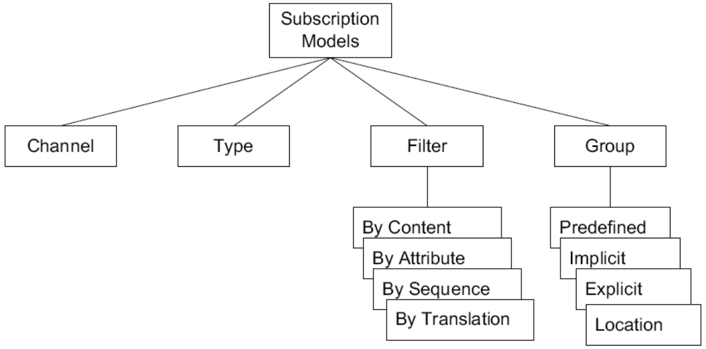
source: Event-Based Programming. Taking Events to the Limit

Subscription Models

Subscription Models

- Describe the way the subscriber identifies the events of interest
- There are four basic models:
  - Channel
  - Type
  - Filter
  - Group

The Subscription Models Hierarchy



Subscription Models

Channel

Type

Filter

- By Content
- By Attribute
- By Sequence
- By Translation

Group

- Predefined
- Implicit
- Explicit
- Location

source: Event-Based Programming. Taking Events to the Limit

Channels

Channels

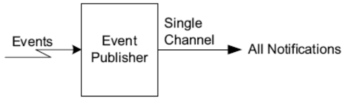
- The channel = the physical or abstract construct through which notifications are routed from the publisher
- Direct Delivery: The publisher is responsible for mapping the events to the various channels
- Indirect Delivery: The middleware is responsible for mapping the events to the various channels

Channels

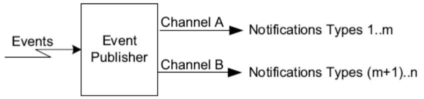
- The publishers can provide one or more channels
- The channels can carry different types of notifications, corresponding to different types of events

Channel types

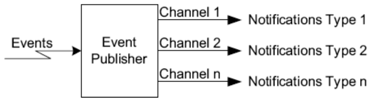
- Single channel, carrying all notifications



- Multiple channels, multiple notifications



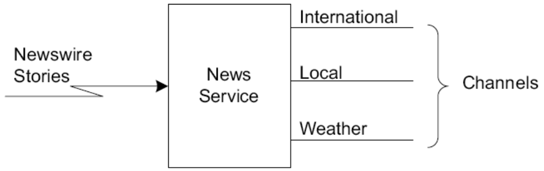
- One channel per notification type



source: Event-Based Programming. Taking Events to the Limit

Example

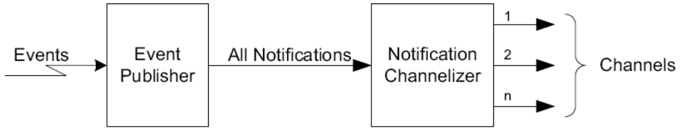
- A news service



source: Event-Based Programming. Taking Events to the Limit

Notification channelizers

- When the process of mapping channels to events is complex, the task can be delegated to a specialized component



source: Event-Based Programming. Taking Events to the Limit

Types

Types

- The method of distinguishing events by assigning them types
- Types can be denoted by
  - Fields in a message header
  - The types of the event generators (e.g. buttons, windows, mouse)
  - The type of the actions that lead to events (e.g., resized, clicked, moved, etc.)
  - ...

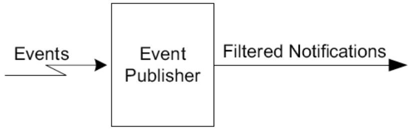
Filters

Filters

- A method to specify the subset of the events of interest by specifying complex conditions
- The subscriber specifies an expression that describes the filtering rules
- The events that match the filtering rules are accepted by the receiver

Filters

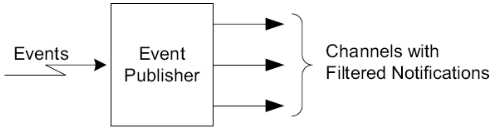
- The filtering can be done by the publisher, so that only the relevant notifications are sent to the subscriber



source: Event-Based Programming. Taking Events to the Limit

Filters and channels

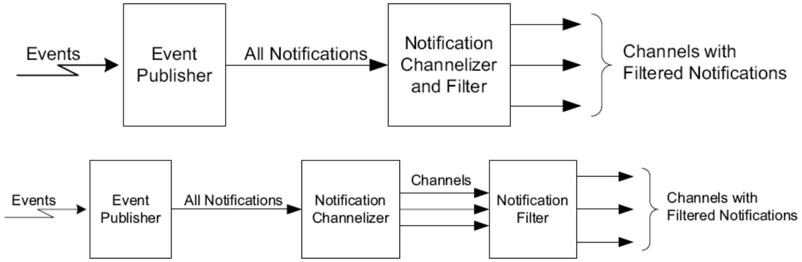
- The filters can be combined with channels, as two parts of the subscription



source: Event-Based Programming. Taking Events to the Limit

Filtering responsibility

- Filtering can be computationally intensive, and can be delegated to separate components, along with the channeling task



source: Event-Based Programming. Taking Events to the Limit

Types of filtering

- There are several possible types of specifying the filters
  - Content filtering
  - Attribute filtering
  - Sequence filtering
  - Translation Filtering

Content filtering

- Applies in the cases where the notifications carry content related to the events
- The filters are specified as conditions regarding the content of the notification payload (e.g., text occurrences, regular expression matches, etc.)
- Content filtering is usually computationally-intensive

Attribute filtering

- Applies when the events can be identified as having a set of attributes
- Also known as topic-based or subject-based filtering
- The attributes are
  - either specified at the source and inserted in an event header, or
  - computed by the notification service, based on various criteria (content, history, patterns, etc.)

Examples

- Source-side filtering
  - filtering the news by location, news provider, and date
- Computed filtering
  - requesting a notification when the value of a stock exchange item raised with a certain percentage

Sequence filtering

- Subscriptions that specify temporal constraints between the events of interest
- Can be used in conjunction with content and attribute filtering or channels

Example

- A sports information system, and a subscriber interested in
  - football scores for a certain team
  - but only if the previous 3 matches were won by the team's competitors

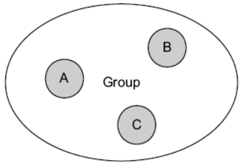
Translation Filtering

- Filtering rules that imply the transformation of the event-related information:
  - altering the content (e.g. language translations)
  - altering the notification type (e.g. notifications of type a, b, c are considered similar, and converted to a type d)
  - altering the sequence: send a notification when a certain sequence of events has occurred in a row

Groups

Groups

- When subscribers are interested in the same events, they can be grouped
- Grouping simplifies the notification delivery
- A group can be seen as a virtual subscriber that replaces the several entities it contains



source: Event-Based Programming. Taking Events to the Limit

Grouping

- Grouping can be done at different levels:
  - The publisher determines the group by information it has about the subscriber (type, role, etc.)
  - The notification infrastructure determines the group based on its known properties (connection speed, location, etc.)
  - The subscriber explicitly joins a group

Grouping

- Groups can overlap, or contain other groups
- Groups can be of several types:
  - Predefined: defined by the publisher or the notification service
  - Implicit: set up automatically when two or more subscribers request the same events
  - Explicit: created at runtime, explicitly
  - Location groups: determined by the subscriber's location in a distributed system

Subscription policies

- Determine the
  - rights for subscribing
  - the (maximum) duration of subscription
  - the way subscriptions can be modified
  - the number of subscriptions for each subscriber
  - ...

Notification Delivery

Notification Delivery

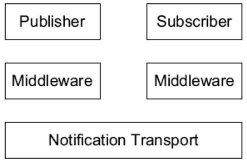
- The process of getting a notification from the publisher to the subscriber
- Its complexity depends on the nature of the connection established between the parties
  - the distance between them
  - the middleware systems that may be involved
  - the delivery protocol
- Different traffic patterns -> different notification architectures

Questions to answer for choosing an architecture

- Should delivery be centralized or not?
- Do we need to use shared resources?
- Is a messaging service needed?
- Should we use complex distributed architectures (e.g. distributed shared memory)?
- Do we need scalability? Is architecture X scalable?

Layered models

- In all non-trivial systems the notifications are delivered indirectly, and usually can be described by a layered model



source: Event-Based Programming. Taking Events to the Limit

Delivery Protocol

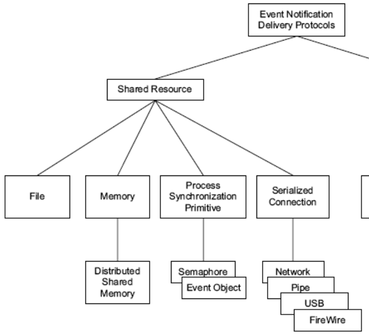
- The rules controlling the interaction between the top layer and the middleware
- Define how the notifications and their payload are transmitted, from the perspective of the clients (senders, receivers)

Two techniques

- There are basically two techniques for sending notifications:
  - through data transfer
    -> implies shared resources
  - through transfer of execution control
    -> implies local or remote procedure calls

Protocols

- Each of the two techniques can be used to describe specialized protocols



Event Notification Delivery Protocols

Shared Resource

Procedure Call

File

Memory

Process Synchronization Primitive

Serialized Connection

Local Procedure Call

Remote Procedure Call

Distributed Shared Memory

Semaphore

Event Object

Network

Pipe

USB

FireWire

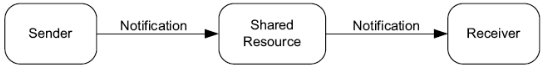
source: Event-Based Programming. Taking Events to the Limit

Notification by data transfer

Notification by data transfer

- A shared resource is implied
- Publishers send data to the resource
- Subscribers receive data from the resource

Notification by data transfer



- The sender uses a push action to write the data to the resource
- The receiver uses a pull action to receive the data from the resource

source: Event-Based Programming. Taking Events to the Limit

Notification by data transfer

- The sender and receiver run in different processes
-> the delivery is usually asynchronous
- The parties usually communicate through messages
- A very common scenario:
-> The sender writes, then continues its work without blocking
-> Later, the receiver detects the data and processes it
-> In some cases, a confirmation is sent to the sender
  -> the sender receives it asynchronously, becoming a receiver for the respective message

Types of shared resources

- Depend on the hardware and software configurations in use
- They can be:
- shared memory
- shared files, pipes
- IPC primitives (e.g. message queues, semaphores)
- Network connections
- ...

Using shared resources

- Shared resources are commonly used, especially in distributed systems
- Advantages:
  - no transfer of execution control is involved
    -> the sender does not depend on the behavior of the receiver, even in extreme cases (e.g. the receiver crashes)
  - have a built-in level of security: the parties do not have access to each other's data
- - can work when the receiver is not present (it can get the data later)

Shared files

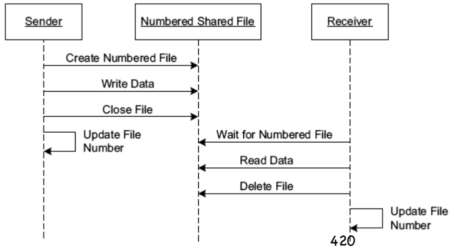
- One of the simplest methods of sending notifications
- Easy to implement when the parties share a common filesystem
- May imply synchronization issues

Scenario 1

- A single file, for one notification
- Sender: creates the file, writes the notification, closes the file
- Receiver: checks for the existence of the file, reads, deletes the file
- Drawback: only one notification at a time

Scenario 1

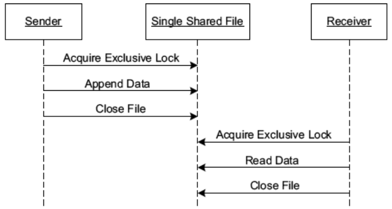
- Modification: multiple files, their name following a pattern
- Drawbacks: inefficient, can generate too many files, awkward implementation



source: Event-Based Programming. Taking Events to the Limit

Scenario 2

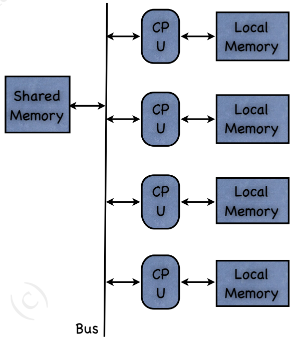
- A single shared file, notifications are appended to it
- Drawback: needs synchronization



source: Event-Based Programming. Taking Events to the Limit

Shared memory

- Effective in parallel/ multiprocessor systems, which implement shared memory in hardware



Shared memory

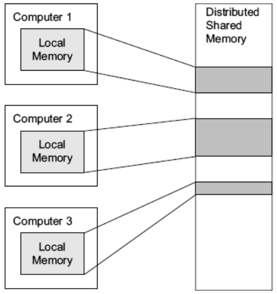
- Regardless the hardware architecture, shared memory is provided by all modern operating systems (e.g. System V IPC)
- Processes can create and access shared memory resources, and use them to send notifications
- Synchronization is necessary, at least through mutual exclusion

Distributed shared memory

- The shared memory can become a bottleneck for the system, when the number of processors is large
- The distributed shared memory model provides a way of decentralizing the memory access
- It applies both to parallel machines, and to distributed systems

Distributed shared memory

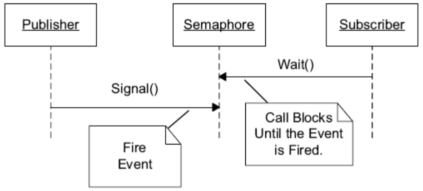
- Each participant system can map a portion of its memory into a distributed shared memory
- The distributed shared memory works like a virtual memory area: the processes do not need to know where the data actually resides



source: Event-Based Programming. Taking Events to the Limit

Semaphores

- Primitives provided by the operating systems, that can be shared among processes
- The semaphores can be used as notification tools



source: Event-Based Programming. Taking Events to the Limit

Notification stealing

- Using semaphores to model events can lead to a serious delivery problem: notification stealing
- Scenario: Processes A, B as subscribers
  - Sender calls signal() twice to notify both
  - A is not yet blocked (is busy, didn't call wait()
  - B unblocks, and possibly calls wait() again, before A
  - A will never receive the notification

SOLUTION: use separate semaphores

Another drawback

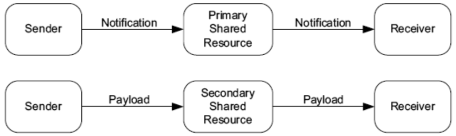
- A specific drawback of using semaphores in event-driven contexts:
  -> the notification cannot carry a payload
- Consequences:
  - cannot be used for complex event architectures
  - cannot be used for implementing subscription filters, selectors, etc.

Out-of-band channels

- Term used for various types of communication systems
- Out-of-band channel = a separate delivery channel used to complement an existing one
- Example: TV broadcasting uses separate channels for subtitles or program guides

Out-of-band channels

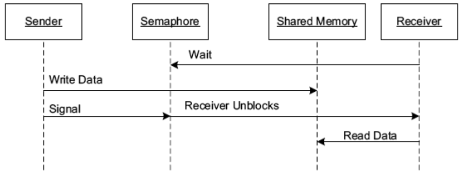
- In the event-based context, it implies
  - a channel that delivers the notification
  - a channel that delivers the notification payload



source: Event-Based Programming. Taking Events to the Limit

Usage of out-of-band channels with semaphores

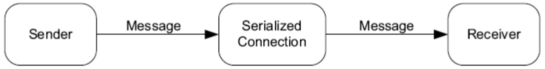
- The payload must be written to the secondary resource before the notification is sent through the semaphore



source: Event-Based Programming. Taking Events to the Limit

Serialized connections

- Serialization: breaking down a data structure into a sequence of bytes
- Serialized connection: a connection that transfers serialized data
- Examples: pipes, sockets



source: Event-Based Programming. Taking Events to the Limit

Serialized connections

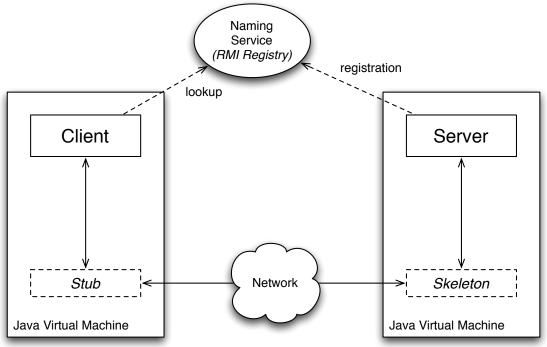
- In the event-based context, using serialized connections implies the following steps:
  - Sender:
    - create the notification as a data structure - marshall the data
    - send it through the connection
  - Receiver:
    - receive the data
    - unmarshall the data
    - read the notification

Notification by procedure calls

Procedure calls

- Can be used for delivering notifications by transferring the execution control from the sender to the receiver
- The notification payload is represented by the procedure parameters
- Can be
  - local, as direct calls
  - remote, through specialized infrastructures (RPC, RMI, CORBA, etc.)

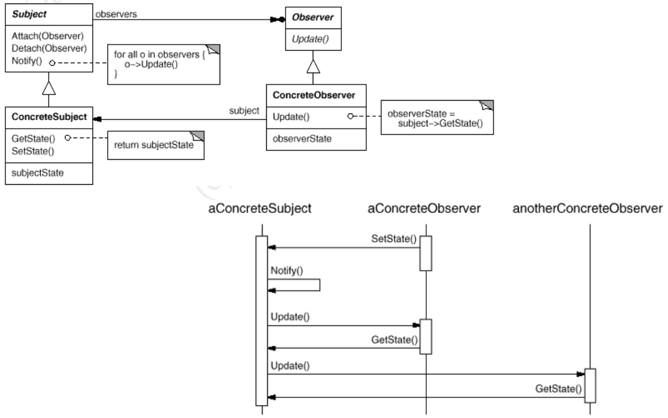
Example: RMI



Sender and receiver

- In event-based systems, the parties involved in communication are established at runtime
- For this purpose, the architecture may use various patterns for creating the publishers and registering the subscribers

Example: the Observer pattern



source: Design Patterns

Synchronicity

- Procedure calls are inherently synchronous
- Through various techniques, asynchronous delivery can also be implemented:
  -> example:
    - the notification call returns immediately so that the sender can continue its work
    - the sender may register a handler for receiving the delivery confirmation

Using direct procedure calls

- Advantages:
  - simple computing model, using familiar, language-specific constructs
  - easy way of including the payload
  - easy error handling (through exceptions)
- Disadvantages:
  - usually the receiver must run at the same time as the sender
  - the sender depends on the receiver's behavior
  - data safety issues: passing references to local sender objects, etc.
  - parameter passing may imply costly data marshaling

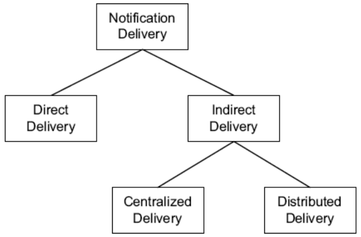
Indirect procedure calls

- To avoid some of the disadvantages, the sender and receiver can be decoupled by using an intermediary service (middleware)
- The indirect delivery systems:
  - can deliver notifications even when the sender and receiver do not run at the same time
  - makes the sender independent on the receiver
  - can avoid data safety by using copies of the original data

Notification Architectures

Notification Architectures

- Describe the infrastructure used for delivering the notifications from the sender to the receiver



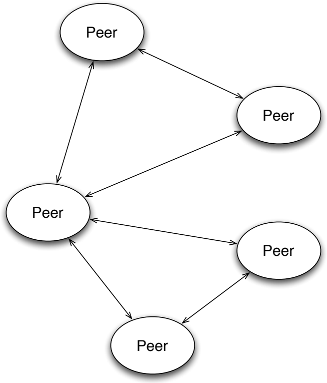
source: Event-Based Programming. Taking Events to the Limit

Direct Delivery

- Also known as point-to-point
- The architecture is simple, the publisher directly sends the notification to the interested receivers
- Disadvantage: not scalable, the sender's performance is affected when many receivers are registered

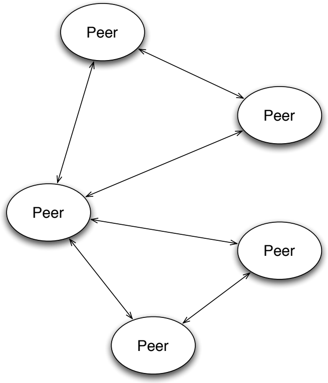
Peer-to-Peer Delivery

- A special case of direct delivery
- Solves the scalability issues by introducing decentralization
- Each node can be both publisher and subscriber of events



Peer-to-Peer Delivery

- How does it work:
  - each node maintains a routing table identifying its neighbors
  - when a notification must be sent to several peers, the sender only sends it to the neighbors
  - new peers can be added through a specific discovery process that finds neighbors for the new node



Indirect Delivery

- A middleware system is involved
- Senders send messages to the middleware system, receivers use the middleware by registering for event notifications
- The communication is usually asynchronous: the sender does not wait for the receiver

Centralized Indirect Delivery

- A single delivery service is used
- Clients connect to the service and send or receive messages
- Two types of delivery:
  - Notification services
  - Messaging services

Notification services

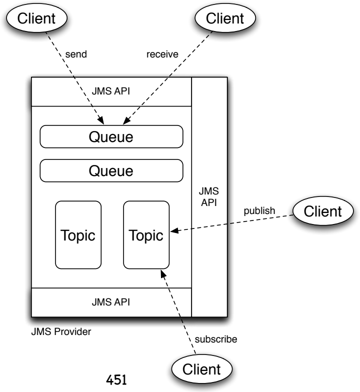
- Use the publish/subscribe model
- Receivers subscribe for events, senders send notifications
- The subscription and filtering are handled by the service

Messaging services

- Use the point-to-point model, usually imply message queues
- In most cases, a queue represents a single receiver
- Hybrid publish/subscribe - point-to-point scenarios can be used: the receivers can subscribe to the service, the senders send notifications without specifying the receivers

Example: JMS

- Both notification and messaging services are provided

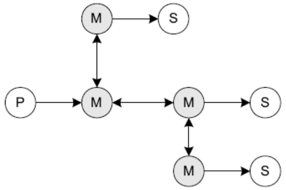


Distributed Indirect Delivery

- The middleware system can become a bottleneck
- Distributed delivery systems can be developed and used to avoid this
- The system is made of several components, each capable of handling multiple senders and receivers

Example

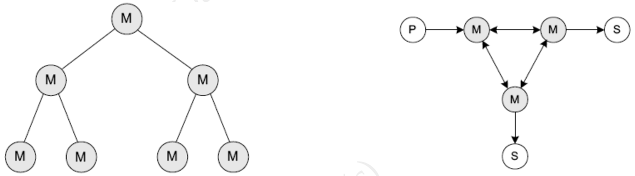
- A string of delivery components



source: Event-Based Programming. Taking Events to the Limit

Example

- The delivery system as a tree of components, or as a graph



- The graph provides connection redundancy

source: Event-Based Programming. Taking Events to the Limit

Sending Notifications

Delivery Synchrony

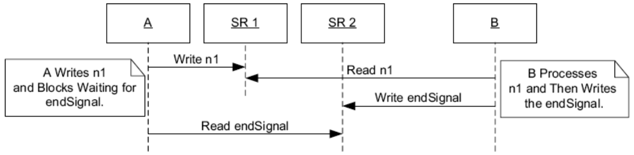
- The event-based systems can use both types of delivery:
  - synchronous
  - asynchronous
- Each method has its advantages and disadvantages

Synchronous delivery

- Advantages
  - simplicity
  - easy notification of event receipt
  - no concurrency
- Disadvantages
  - high sender-receiver coupling
  - no concurrency

Synchronous delivery

- Easy to implement with procedure calls
- With shared resources, a secondary resource must be used to provide synchronicity:


A
SR 1
SR 2
B
A Writes n1 and Blocks Waiting for endSignal.
Write n1
Read n1
B Processes n1 and Then Writes the endSignal.
Write endSignal
Read endSignal

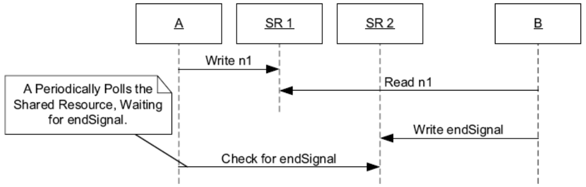
source: Event-Based Programming. Taking Events to the Limit

Asynchronous delivery

- Two scenarios are available
  - the sender is not interested in the receiver's processing of the event
  - the sender needs feedback from the receiver

Asynchronous delivery

- The response from the receiver can be
  - synchronous (e.g. receiver calls a method provided by the sender)
  - asynchronous (e.g. receiver notifies the sender via a secondary shared resource – see Figure:)


A
SR 1
SR 2
B
Write n1
Read n1
A Periodically Polls the Shared Resource, Waiting for endSignal.
Write endSignal
Check for endSignal

source: Event-Based Programming. Taking Events to the Limit

Asynchronous delivery

- Advantages:
  - sender and receiver are loosely coupled
  - events are processed as soon as possible
- Disadvantages
  - complex architectures
  - concurrency issues

Delivery Fanout

Delivery Fanout

- Term borrowed from the digital logic field: the maximum number of inputs an output signal can be connected to
- Event-based context: the number of receivers a sender can notify
- Two types:
  - unicast (only one receiver)
  - multicast (several receivers)

Quality of Service

Quality of Service

- Defines a set of options a delivery system provide in respect to issues as:
  - reliability
  - priority
  - timing
  - throughput
  - order

Reliability

- How hard does the system try to deliver a notification?
  - at most once - sometimes (e.g. if the receiver is not online) notifications may be lost
  - exactly once - always attempts sending, but only one time
  - at most n times
  - best effort - keep trying until a condition (e.g. timeout, number of retries) occurs
  - certified delivery - specifies whether the delivery failed or not

Priority

- Notifications are prioritized based on
  - the notification type
  - the senders
  - the receivers

Timing

- Option to specify time constraints for delivering the notifications
- If the constraints couldn't be met, the notification is considered undeliverable, and discarded

Throughput

- How much information a receiver gets per time
- Useful as an option when the bandwidth is limited, and the delivery service implies different costs

Order

- Specifies the order in which queued notifications are sent
  - Any - the order is not important
  - FIFO
  - Priority-based
  - Deadline - notifications that expire soon are sent first

Transactions

- In some cases, a sequence of notifications must be delivered atomically
- Two types of transactions:
  - a single receiver must get all the notifications in a transaction
  - all receivers in a set must receive the notifications before the transaction is committed

Notification Payload

Notification Payload

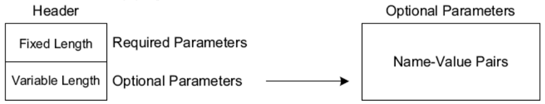
- The information carried by the notification that
  - describes the event
  - enables the receiver to continue without consulting with the sender

Concerns

- The type of delivery (shared resource, procedure call) may impose constraints on the payload in its
  - size
  - transfer throughput
  - transfer latency
  - transfer reliability
- The system must balance the
  - payload size, vs.
  - notification frequency

Notifications sent using shared resources

- The notifications are in fact messages
- Messages: headers+body



source: Event-Based Programming. Taking Events to the Limit

Notifications sent using procedure calls

- The payload: the arguments of the procedure
- The arguments usually specify:
  - event source
  - event type
  - event data

String-Based Payloads

- The simplest type of notification payload
- It is flexible enough for many applications
- Can also be used for structured data (e.g. XML)

	Property Name	Property Value
Message	"PageTitle"	"Baking a Potato"
	"PageNumber"	"14"
	"DocumentAuthor"	"Peter Hughes"

source: Event-Based Programming. Taking Events to the Limit

Record-Based Payload

- The data is stored in structured records
- It is easy and efficient to use
- The recipient must know the offset, type, size and meaning of each field

	Field Name	Field Length (in Bytes)
Message	MessageType	1
	TransactionNumber	8
	Flags	1
	Priority	1
	DateAndTime	4

source: Event-Based Programming. Taking Events to the Limit

Object Payloads

- Objects (including the sender object) can be passed as arguments to procedure calls. This imposes some constraints:
  - the type of the objects must be shared between the sender and receiver
  - for remote calls the objects must be serializable when marshaling is needed

Object arguments

- Passing an object reference (e.g. the sender):
  - may lead to problems or complicated approaches regarding object publication
  - objects sent as references may need to be immutable (they MUST be for multicasting)
  - object ownership must be clearly defined and handled
  - the usage of the object by the receiver may imply inefficient interactions between the object and the receiver

Coupling problems

- When sending object references, an important problem regards the coupling between the entities related to the sender and receiver
- Because of the notification payload, the system may imply unwanted, newly added couplings

Coupling problems

- Example: a set of scenarios of component coupling
- Because of the type coupling, components become coupled artificially

Coupling problems

- No problem here: the scenario does not add new inter-component couplings

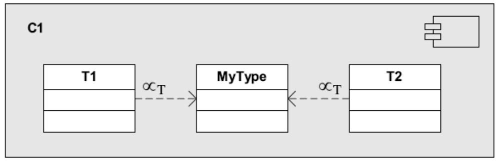


Figure 4-5. *Coupling to MyType, when T1 and T2 are in the same component*

source: Event-Based Programming. Taking Events to the Limit

Coupling problems

- New coupling added: receiver component (C2) depends on sender (C1)

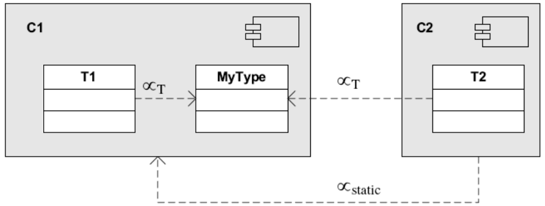


Figure 4-6. *Coupling to MyType, when T1 is packaged with MyType, but not with T2*

source: Event-Based Programming. Taking Events to the Limit

Coupling problems

- New coupling added: sender component (C1) must know the type MyType defined in C2

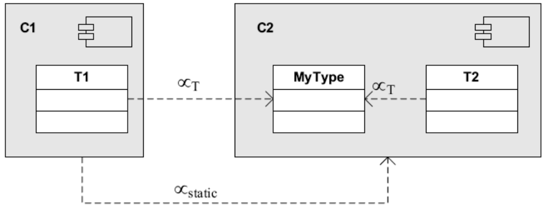


Figure 4-7. *Coupling to MyType, when T2 is packaged with MyType, but not with T1*

source: Event-Based Programming. Taking Events to the Limit

Coupling problems

- New couplings added: sender and receiver components must know the type MyType defined in another component (C3)

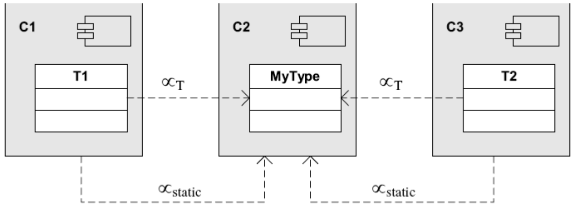
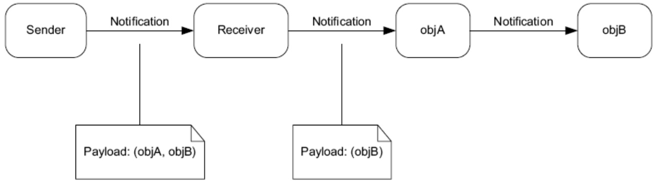


Figure 4-8. *Coupling to MyType when T1, T2, and MyType are all in different components*

source: Event-Based Programming. Taking Events to the Limit

Notification forwarding

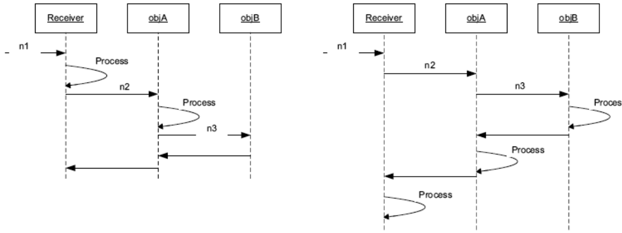
- A particular usage of objects as notification payloads
- Defines a pattern of implementing notification multicasting using delegation



source: Event-Based Programming. Taking Events to the Limit

Notification forwarding

- The processing in the forwarding chain can be done in two ways



A) Breadth-First Processing

B) Depth-First Processing

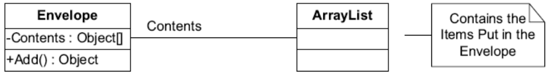
source: Event-Based Programming. Taking Events to the Limit

Envelopes

- Also known as collector objects
- A pattern allowing the gathering of results from receivers, by including in the payload a reference to an envelope object

Envelopes

- The pattern



- The Add() method is called by the receivers to add their specific result
- The envelope object is passed to all subscribers of the respective event
- The receivers cannot modify the data except by adding their own result

Composite payloads

- A method of increasing the delivery performance by carrying the data for multiple events in a single payload
- The sender stores a buffer of notifications and sends them only when possible/needed
- Two types of composite payloads
  - uniform (sequences of payloads for a single type of notification)
  - heterogenous (sequences of payloads for any type of notification)

Composite payloads

- Uniform composite payloads
  - the sender analyzes the buffered notifications and groups the payloads for the same event type
  - drawback: notifications may be sent in noncausal order
    example: T1 T2 T1 T2 --> T1 T1 T2 T2

Composite payloads

- Heterogenous composite payloads
  - the sender groups the buffered notifications regardless of event type
  - advantage: sends the notifications in the correct, causal order
  - drawback: expensive, each payload must be tagged with the notification type

Payload coalescing

- A variation of composing payloads
- When the sequence of events E1 and E2 has the same effect as the occurrence of an event E3, only the notification for E3 is sent
- Example: in a GUI, merging multiple paint events in a single (more complex) one to improve drawing performance and avoid screen flickering due to frequent repainting

Event-Based Interaction Patterns

Interaction Patterns

- The interaction dynamics between the processes in an event-based system:
  - roles of the parties: sender/receiver
  - control: who initiates/controls/terminates the interaction?
  - timing: does the sender wait for the response?
  - flow: is the information sent in a single step or in a sequence of steps?
- Several recurring patterns can be observed

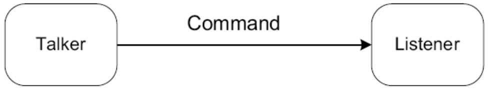
The Push-Pull Model

- Classifies the interaction by focusing on which way the information is sent between the processes
- Push: one party gives the information to another
- Pull: one party requests the information from the other

Push Interactions

- Context:
  - Process P1 needs to send commands to P2
  - Frequency of commands: variable, unknown
  - P2 must never miss a command
- Analysis:
  - infrequent commands => P2 should not poll for P1 for commands
  - P1 knows when commands are available => it should control the interaction

Push Interactions



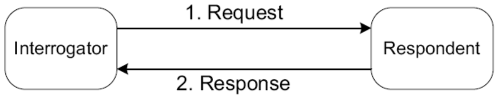
- Useful when
  - The talker controls the listener
  - The talker decides when to send commands

source: Event-Based Programming. Taking Events to the Limit

Pull Interactions

- Context:
  - P1 needs to monitor the status of P2
  - P1 only needs the status at specific times
  - P2 provides information only when requested
- Analysis:
  - P1 doesn't always need the status => it is impractical for P2 to send it at each change
  - P1 should control the interaction

Pull Interactions



- Useful when
  - The interrogator needs to control the time when the information is retrieved
  - The interrogator only needs information at specific times

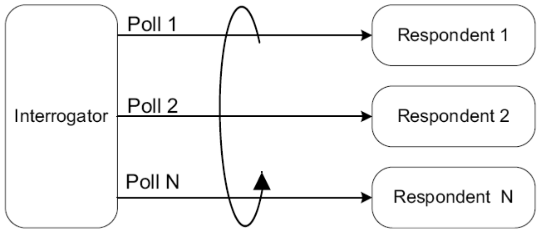
source: Event-Based Programming. Taking Events to the Limit

Pull Interactions -- the Round-Robin Polling Pattern

- Context:
  - P_1 needs to monitor several P_k processes
  - P_k can return the status very fast
  - P_1 can handle status changes only at specific times
- Analysis:
  - if all P_k send notifications, P_1 may not be able to handle them at the same time
  - P_1 should control the interaction

Pull Interactions -- the Round-Robin Polling Pattern

- The interrogator polls the respondents which in turn provide the status quickly



source: Event-Based Programming. Taking Events to the Limit

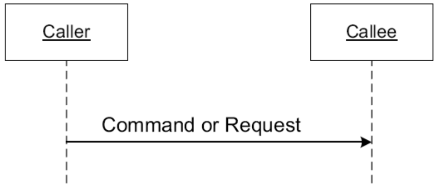
Blind Interactions

- Context:
  - P1 needs to send a message to P2
  - P1 does not need feedback regarding the progress of P2's execution
- The context applies both to push and pull interactions

Synchronous Blind Interactions

- Context:
  - P1 needs to send a command to P2
  - P1 must wait for P2 to finish the command
  - P2 is trusted to respond in a timely manner
- Implications:
  - P1 waits for P2, and progress data is not needed

Synchronous Blind Interactions

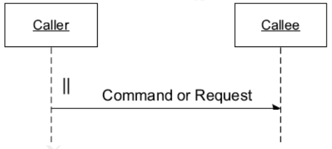


source: Event-Based Programming. Taking Events to the Limit

Asynchronous Blind Interactions

- Context:
  - P1 needs to send a command to P2
  - P1 must do other work while P2 executes the command
- Implications:
  - P1 must be able to run while P2 executes

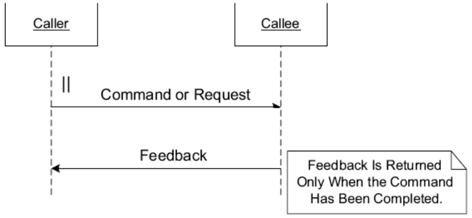
Asynchronous Blind Interactions



- Feedback may be
  - not needed by the caller (“fire and forget”)
  - needed by the caller after or at the end of the command execution

Feedback

- Pushed feedback



Caller

Callee

||

Command or Request

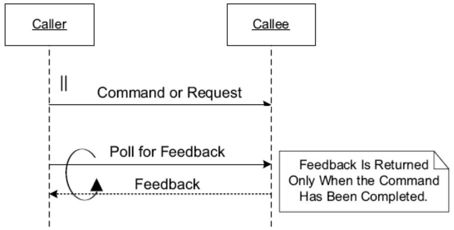
Feedback

Feedback Is Returned Only When the Command Has Been Completed.

source: Event-Based Programming. Taking Events to the Limit

Feedback

- Polled feedback
  ~ the actual command is sent from a secondary thread in the caller



Caller

Callee

||

Command or Request

Poll for Feedback

Feedback

Feedback Is Returned Only When the Command Has Been Completed.

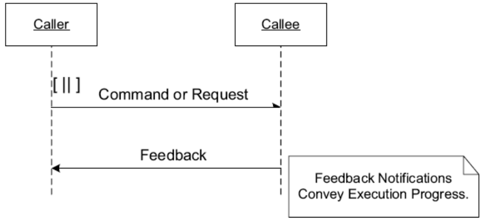
source: Event-Based Programming. Taking Events to the Limit

Transparent Interactions

- Context:
  - P1 needs to send a command to P2
  - P1 needs to know the progress of command execution

Transparent Interactions

- Pushed feedback
  - P1 needs to send a command to P2
  - P1 needs feedback as soon as it is available



Caller

Callee

[||]

Command or Request

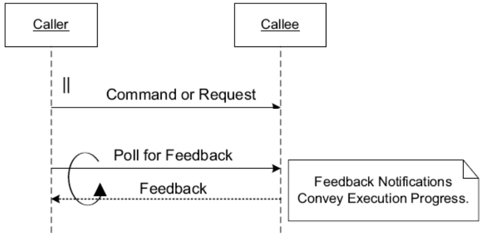
Feedback

Feedback Notifications Convey Execution Progress.

source: Event-Based Programming. Taking Events to the Limit

Transparent Interactions

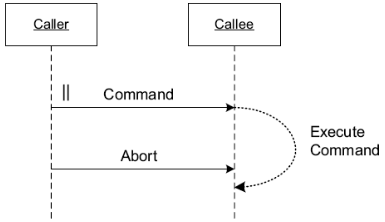
- Polled feedback
  - P1 needs to send a command to P2
  - P1 needs feedback at specific times



source: Event-Based Programming. Taking Events to the Limit

Interruptible Interactions

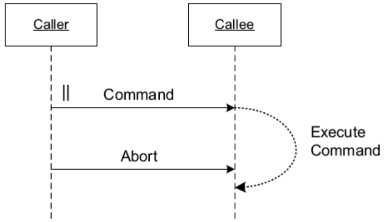
- Context:
  - P1 needs to send a command to P2
  - while the command is executed, it needs to be canceled due to various reasons



source: Event-Based Programming. Taking Events to the Limit

Interruptible Blind Interactions

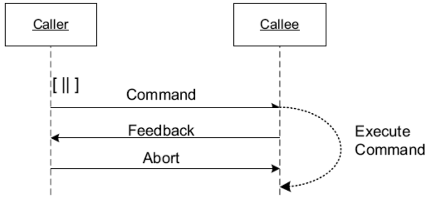
- Context:
  - P1 needs to send a command to P2
  - P1 does not need progress information
  - the command may need to be canceled



source: Event-Based Programming. Taking Events to the Limit

Interruptible Transparent Interactions

- Context:
  - P1 needs to send a command to P2
  - P1 needs progress information
  - the command may need to be canceled

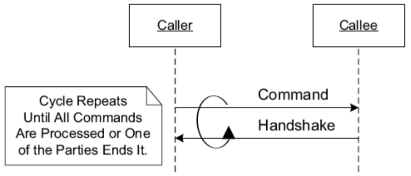


source: Event-Based Programming. Taking Events to the Limit

Handshaking

- Context:
  - P1 needs to transfer a large amount of data to P2
  - The information is broken down in a sequence of messages
  - The sending may need to be stopped before all the data was sent
  - Both P1 and P2 should be able to decide when the message flow must be stopped

Handshaking


Caller
Callee
Cycle Repeats Until All Commands Are Processed or One of the Parties Ends It.
Command
Handshake

- The handshake response informs the sender to send more data
- Any party may decide that the transmission should be ended

source: Event-Based Programming. Taking Events to the Limit