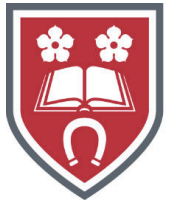
CO3090/CO7090

Distributed Systems and Applications

by

Marco Perez and Emmanuel Tadjouddine



UNIVERSITY OF LEICESTER

Distributed Systems

Overview

Why are distributed systems/applications so widely spread?

- Hardware has been quite expensive in the past
  - minicomputers ~ 10000$
  - UNIVAC I: 1.250.000 – 1.500.000$
  - 46 systems were eventually built and delivered
  - 1, 905 operations per second running on a 2.25 MHz clock

  Price/performance gain of 10^{13}
- High-speed networks
  - LAN: 10^7 – 10^{10} bit/sec WAN:
  - 64Kbps - gigabits/sec
- Organisations geographically spread
  - intra-organisational interaction
  - inter-organisation interaction

Why are distributed systems/applications so widely spread?

- Hardware has been quite expensive in the past
  - minicomputers $\sim$ 10000$
  - UNIVAC I: 1.250.000 $-$ 1.500.000$
  - 46 systems were eventually built and delivered
  - 1, 905 operations per second running on a 2.25 MHz clock

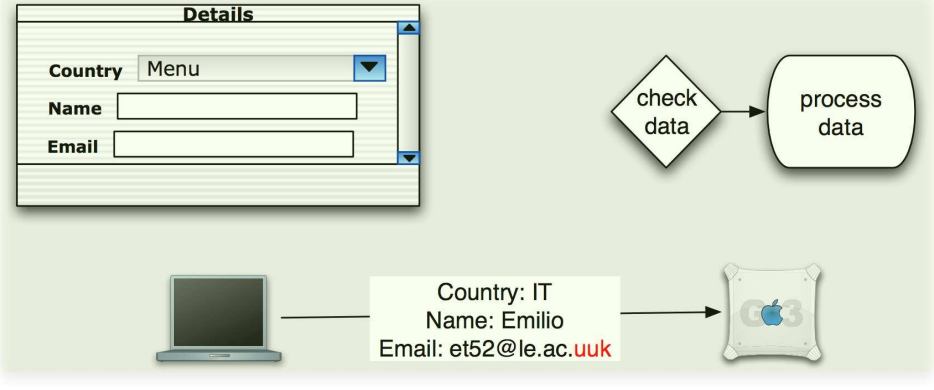
  Price/performance gain of 10^{13}
- High-speed networks
  - LAN: $10^7 - 10^{10}$ bit/sec WAN:
  - 64Kbps - gigabits/sec
- Organisations geographically spread
  - intra-organisational interaction
  - inter-organisation interaction

Distribution issues (1)

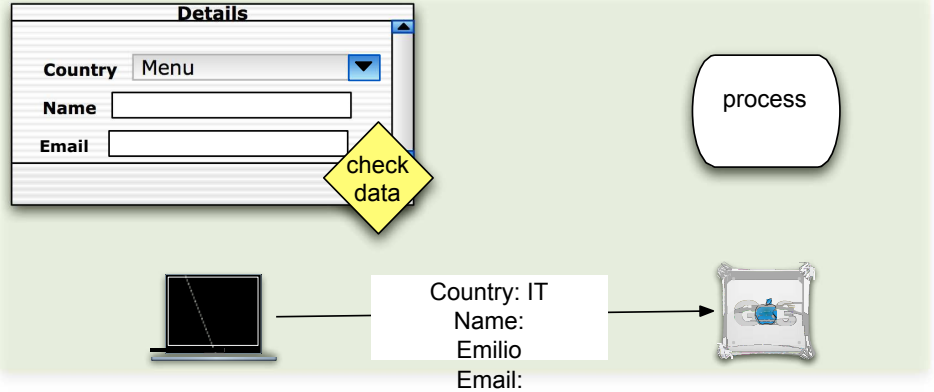
When computation/data must be distributed, several aspects intimately connected to distribution must be considered:

Example 1 (see [TV06], Chapt.1) Consider a client-server application where the client asks the user to fill-in a form and send data to a server to be processed. Suppose that data have to be checked before processing them.

Solution (a)



Solution (b)



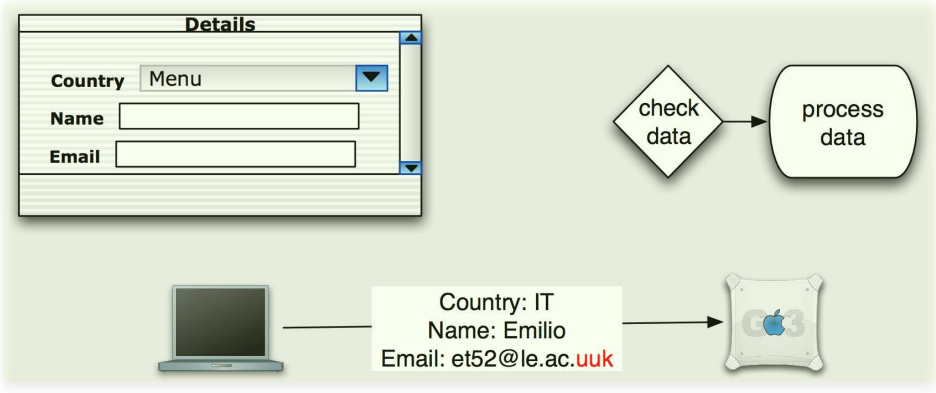
Exercise 1 Reflect on solutions (a) and (b) in Example 1. Which one do you prefer? Motivate your answer.

Exercise 2 Suggest an alternative solution for the application of Exercise 1.

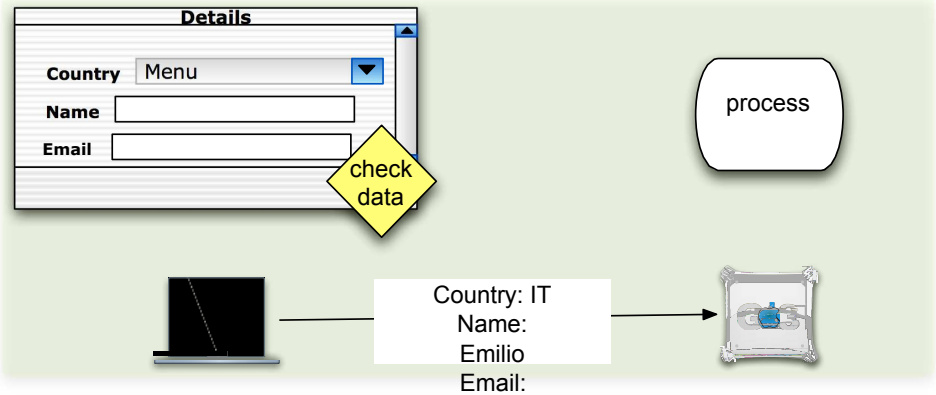
Distribution issues (1)

When computation/data must be distributed, several aspects intimately connected to distribution must be considered:

Solution (a)



Solution (b)



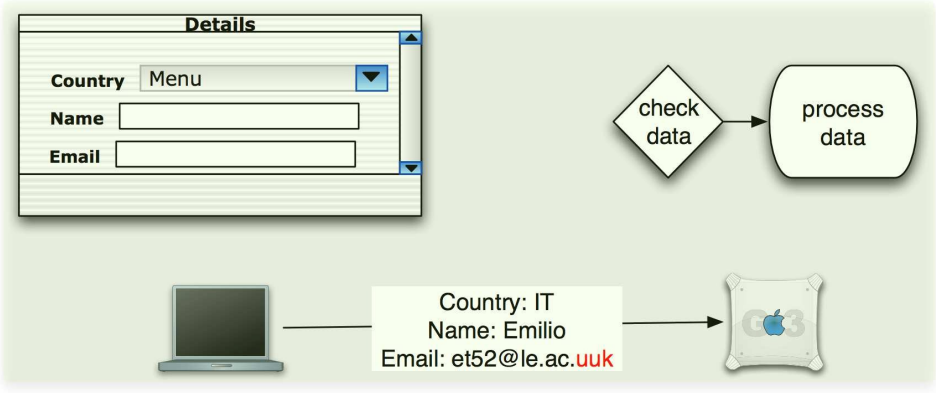
It depends …

(1) What if we want to check whether the email address entered is (syntactically) valid
(2) .. whether "email" field matches "confirm email" field
(3) .. whether email address is already used by another registered user
(4) .. whether the email account exist in the University Exchange Server
(5) … whether the email server is blacklisted (where the list is kept on spam filter server)
(6) What if more than one clients attempted to submit two forms using the same email address?
(7) What if the client has very limit bandwidth and unstable connection?

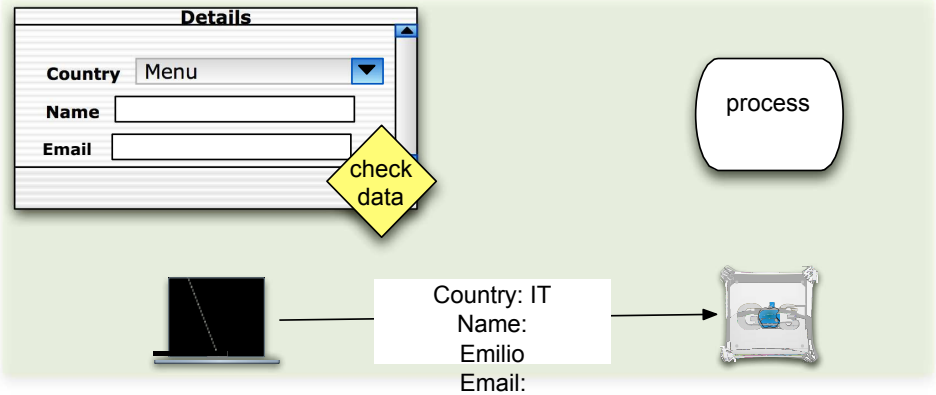
Distribution issues (1)

When computation/data must be distributed, several aspects intimately connected to distribution must be considered:

Solution (a)



Solution (b)



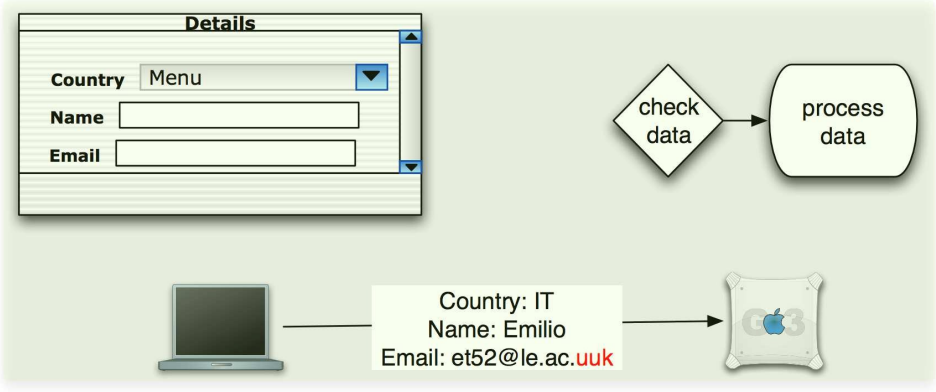
It depends …

(1) What if we want to check whether the email address entered is (syntactically) valid
(2) .. whether "email" field matches "confirm email" field
(3) .. whether email address is already used by another registered user
(4) .. whether the email account exist in the University Exchange Server
(5) … whether the email server is blacklisted (where the list is kept on spam filter server)
(6) What if more than one clients attempted to submit two forms using the same email address?
(7) What if the client has very limit bandwidth and unstable connection?

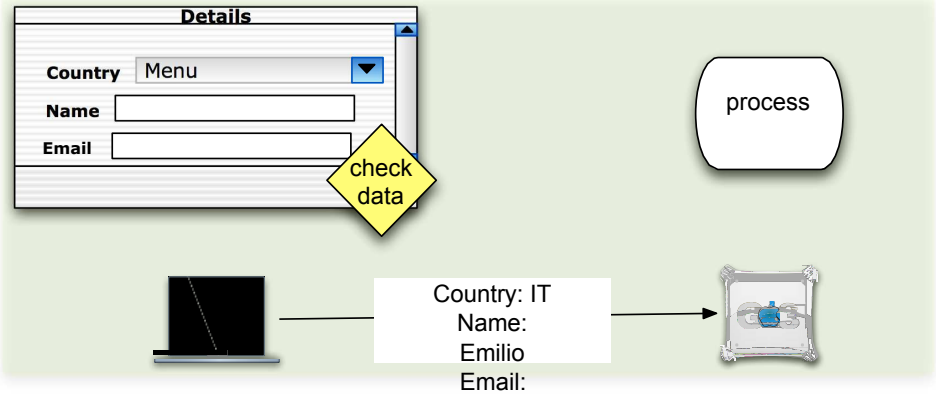
Distribution issues (1)

When computation/data must be distributed, several aspects intimately connected to distribution must be considered:

Solution (a)



Solution (b)



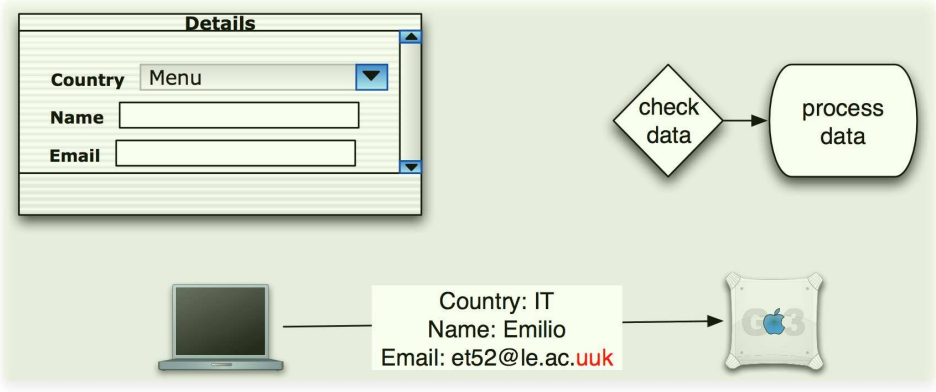
Thanks about …

(1) What if we want to check whether the email address entered is (syntactically) valid
(2) .. whether "email" field matches "confirm email" field
(3) .. whether email address is already used by another registered user
(4) .. whether the email account exist in the University Exchange Server
(5) What if more than one clients attempted to submit two forms using the same email address?
(6) What if the client has very limit bandwidth and unstable connection?
……

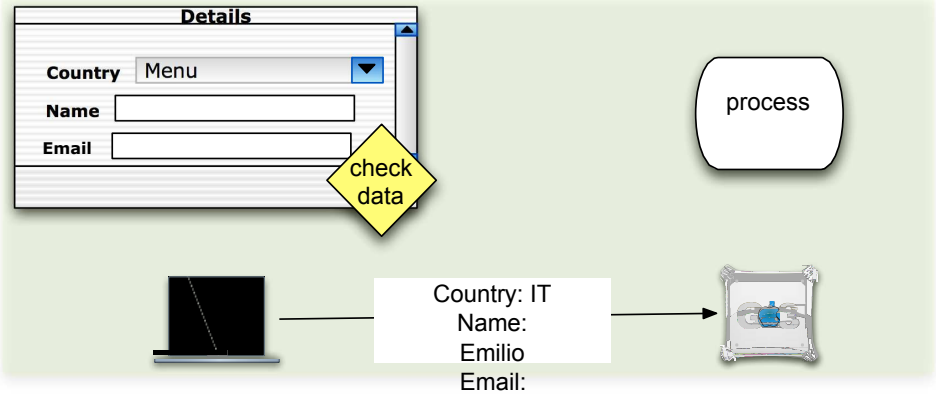
Distribution issues (1)

When computation/data must be distributed, several aspects intimately connected to distribution must be considered:

Solution (a)



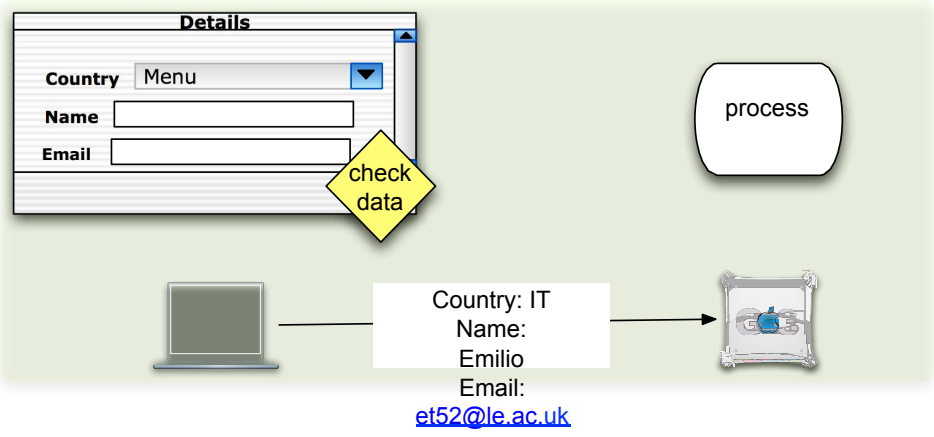
Solution (b)



Thanks about …

(1) What if we want to check whether the email address entered is (syntactically) valid
(2) .. whether "email" field matches "confirm email" field
(3) .. whether email address is already used by another registered user
(4) .. whether the email account exist in the University Exchange Server
(5) What if more than one clients attempted to submit two forms using the same email address?
(6) What if the client has very limit bandwidth and unstable connection?
……

Distribution issues (2)



Other issues to consider are

- concurrent access to shared resources
  synchronous/asynchronous
- failures / (distributed) transactions
- heterogeneity
- caching

Typically, when programming distributed applications you cannot assume

- network reliability
- exclusive access to resources
- homogeneity
- fixed topology
- single administrative authority
- ...

Concurrency is an “intrinsic” property of distributed systems:

- A sequential program has a single thread of control. Intuitively, this means that there is only one instruction of the program at time that is executed
- A concurrent program might have many threads of control. Intuitively, at the same time many instructions of the program can be executed.

Example 2 (R. Milner "Communication and Concurrency”, Prentice Hall 1989)
Consider this two “programs”

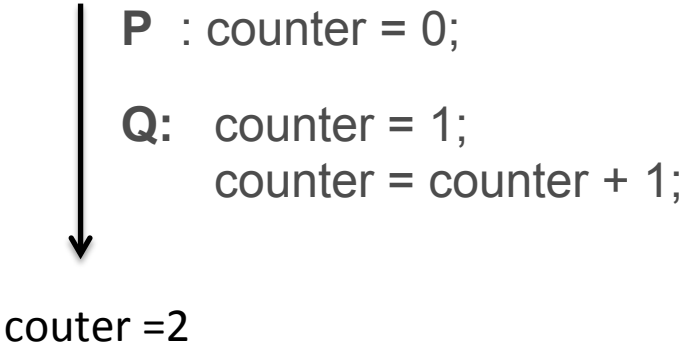
P : counter = 0; Q : counter = 1; counter = counter + 1;

The behaviour of P or Q is different if they are executed sequentially or in parallel.

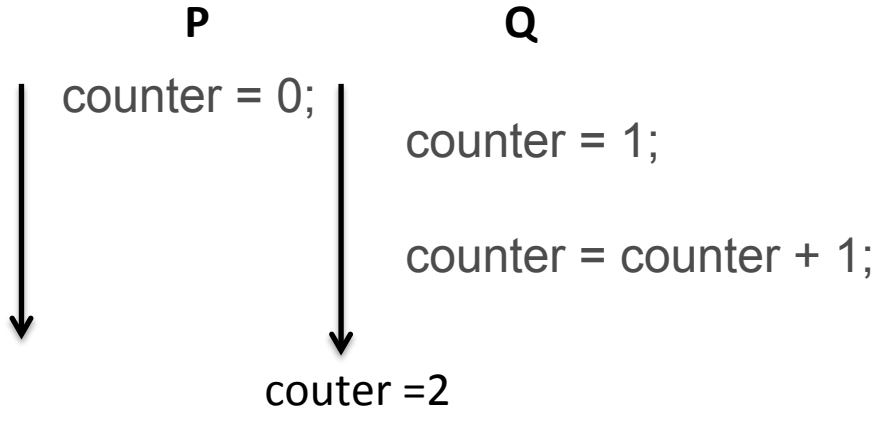
Exercise 3 Consider P and Q above. What value does counter get after executing

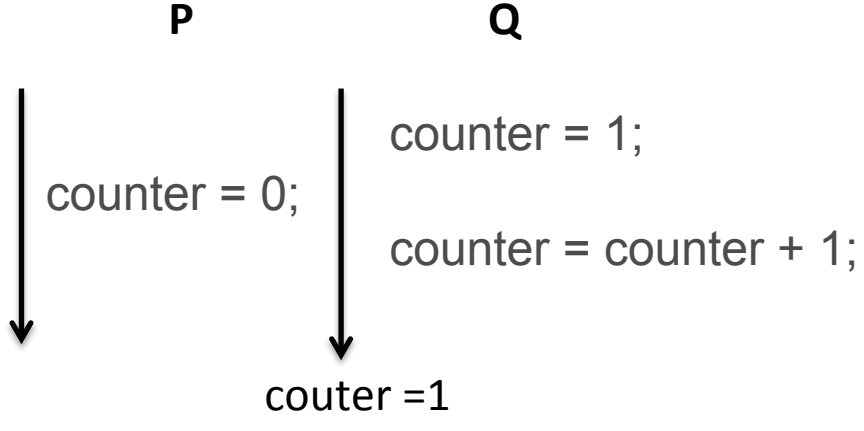
- first P and then Q? P
- in parallel with Q?

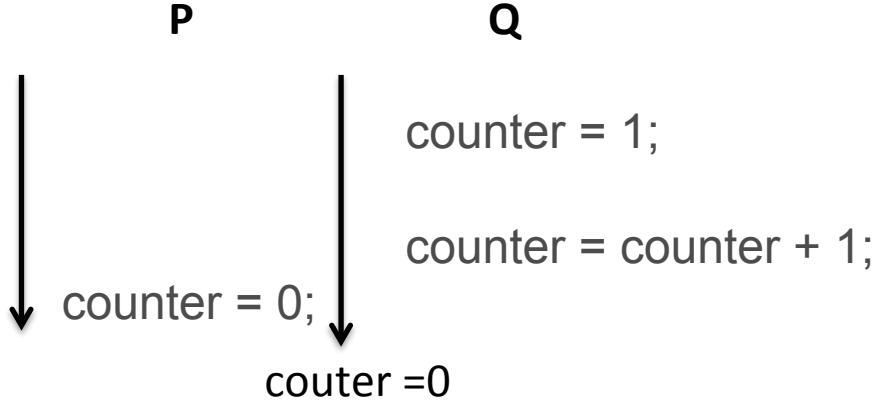
Sequential first P and then Q



P in Parallel with Q







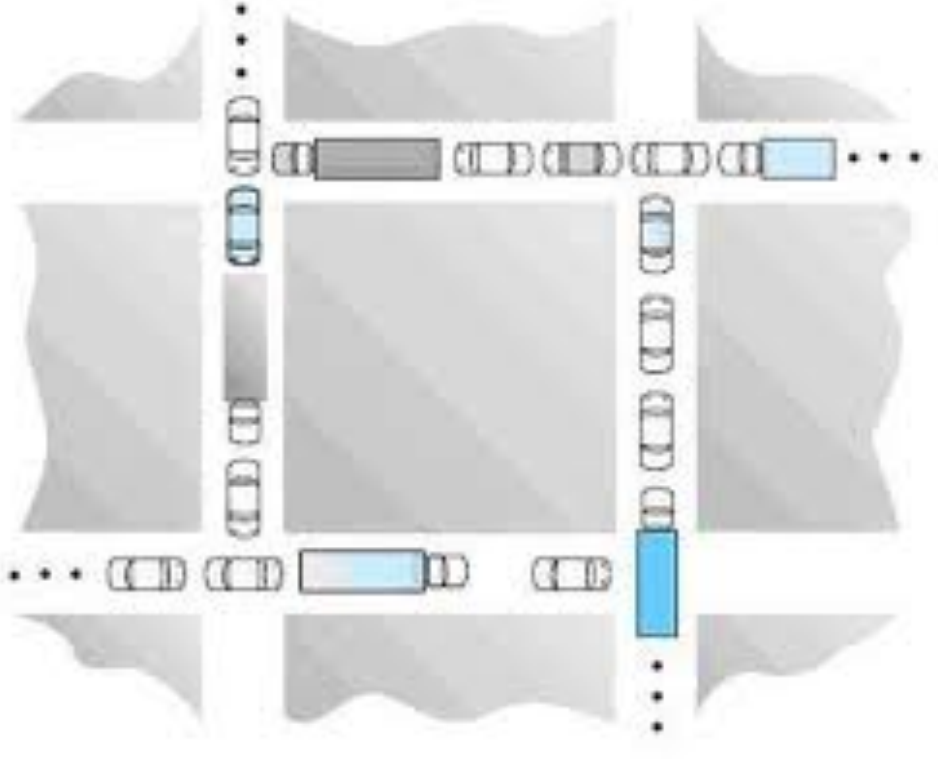
Concurrency: problems

Concurrent programs

- are complex to design
  - shared resources must be considered
  - deadlocks
- are hard to test
  - too many scenarios (concurrent events may occur in any order)
- are hard to debug
  - errors are subtle: they might depend on the interaction of threads,
- require non-trivial reasoning techniques
- yield non-deterministic results that show up when
  - threads share memory or
  - the result may depend on “the order” of execution (see Exercise 3)

Real life deadlock situation

Traffic jam at the junction



Concurrency: problems

Concurrent programs

- are complex to design
  - shared resources must be considered
  - deadlocks
- are hard to test
  - too many scenarios (concurrent events may occur in any order)
- are hard to debug
  - errors are subtle: they might depend on the interaction of threads,
- require non-trivial reasoning techniques
- yield non-deterministic results that show up when
  - threads share memory or
  - the result may depend on “the order” of execution (see Exercise 3)

From [Eck02]

"Concurrent programming is like stepping into an entirely new world and learning a new programming language, or at least a new set of language concepts. With the appearance of thread support in most microcomputer operating systems, extensions for threads have also been appearing in programming languages or libraries. In all cases, thread programming:

1. Seems mysterious and requires a shift in the way you think about programming
2. Looks similar to thread support in other languages, so when you understand threads, you understand a common tongue.

[...] threads are tricky."

"Some examples were developed on a dual-processor Win2K machine which would immediately show collisions. However, the same example run on single-processor machines might run for extended periods without demonstrating a collision– this is the kind of scary behavior that makes multithreading difficult. You can imagine developing on a single-processor machine and thinking that your code is thread safe, then discovering breakages as soon as it's moved to a multiprocessor machine."

From [Eck02]

"Concurrent programming is like stepping into an entirely new world and learning a new programming language, or at least a new set of language concepts. With the appearance of thread support in most microcomputer operating systems, extensions for threads have also been appearing in programming languages or libraries. In all cases, thread programming:

1. Seems mysterious and requires a shift in the way you think about programming
2. Looks similar to thread support in other languages, so when you understand threads, you understand a common tongue.

[...] threads are tricky."

"Some examples were developed on a dual-processor Win2K machine which would immediately show collisions. However, the same example run on single-processor machines might run for extended periods without demonstrating a collision– this is the kind of scary behavior that makes multithreading difficult. You can imagine developing on a single-processor machine and thinking that your code is thread safe, then discovering breakages as soon as it's moved to a multiprocessor machine."

Concurrency: advantages

So, why should we write concurrent programs?

- concurrency naturally arises in many applications sometime concurrency cannot
- be avoided

  Example 3

  - while polling an external device, something else should be done
  - distributed applications are also concurrent applications
- speed up response to users interacting with the system

  Example 4

  - time-consuming tasks (e.g., waiting for users' to react) can go in parallel with other tasks
  - servers usually consists of a thread dealing with invocations and other threads actually serving requests
- ...

Distributed/concurrent domains/applications

(Web) services while serving a request, new requests can be processed. It is worth to do so in order to minimise service latency and improve availability

Real parallelism when multiple CPUs are available, it is worth to structure applications such that parallelism can be exploited

I/O handling even on small modern machines, external devices operate independently from the CPU. Hence, it is a good idea to make I/O a concurrent activity of e.g., computing tasks

Simulation many real situation can be naturally modelled as a number of independent interacting objects

Component-based software concurrency helps when components must be glued together

Mobile code: Concurrent mobile apps

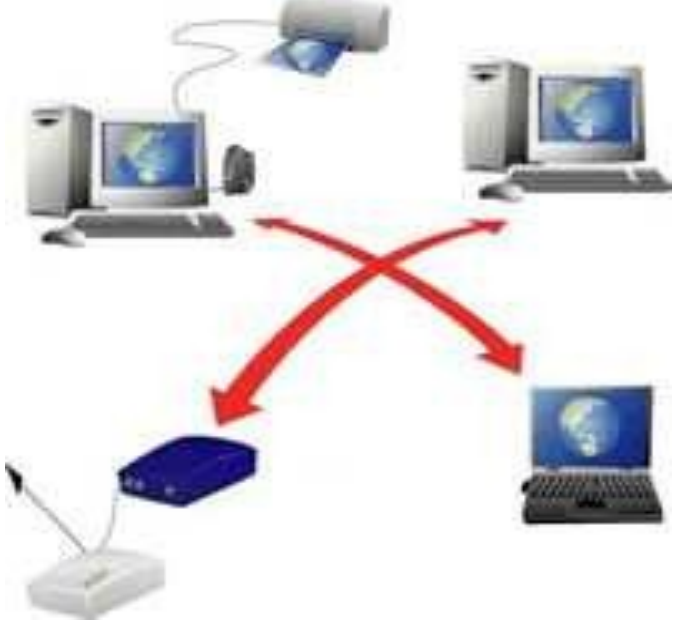
Embedded systems real-time applications could not exist without concurrency

Big Data Analytics storing and processing large sets of data.

What is distribution?

" In the term distributed computing, the word distributed means spread out accross space. Thus, distributed computing is an activity performed on a spatially distributed system. " [LL90]

Remark 1 According to this acception, very few systems are not "distributed"! For instance, your laptop is a distributed system...even if not connected to other computers!



Typically, distributed systems are thought of as a bunch of computers connected through a communication infrastructure.

Their main peculiarities are:

- heterogeneity
- no shared memory / clock topology
-

Distributed view

Indeed, there's a further specification

> " Although one usually speaks of a distributed system, it is more accurate to speak of a distributed view of a system. " [LL90]

Therefore, the level of abstraction (namely, your current view of the system) is always important. Hence, among the possible views of your laptop, you have

Programmer view

Hardware designer view

A taxonomy of (abstract) models of distributed systems

- Network topology
- Synchrony
- Failures
- Message buffering

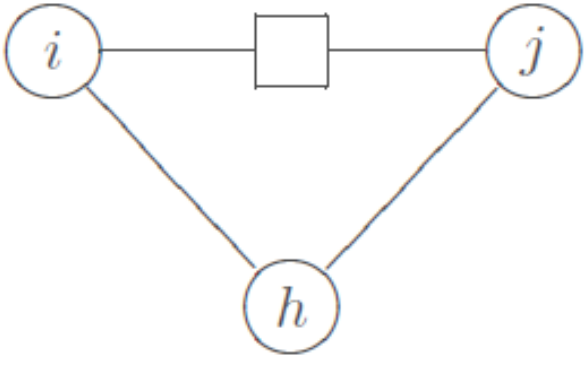
A taxonomy of (abstract) models of distributed systems

- Network topology
- Synchrony
- Failures
- Message buffering

A taxonomy of (abstract) models of distributed systems

- Network topology
- Synchrony
- Failures
- Message buffering

An abstract representation of a distributed system is given as (communication) graph:



- nodes represent “processes” and boxes represent memory
- links connecting 2 processes represent the capacity for them to communicate while links connecting a process and a memory represent the fact that the former has access to the latter (links can be oriented)

For instance, h can interact with both i and j, while i and j share a memory to communicate

A taxonomy of (abstract) models of distributed systems

- Network topology
- Synchrony
- Failures
- Message buffering

Assuming no failure, we can have:

Completely asynchronous models: no assumption on the amount of time elapsed between send and receive (e.g. AJAX)

"Weakly synchronous" models: a message sent at time t is received at most within t + x and the delay x is known

Strongly synchronous models: the entire computation proceeds in a sequence of distinct rounds and, at each round, every process sends messages based upon previously received messages

"Strongly synchronous" models

Example 1:

Step 1

```
boolean verified=
verifyCreditCardPayments(bank, card);

……
//waitng for confirmation from bank
```

Step 2

```
If(verified==true){
  mailsender.sendEmail
    ("Thank you for your payment", email);
}else{
    screen.sendMessage("payment
failed");
}
```

A taxonomy of (abstract) models of distributed systems

- Network topology
- Synchrony
- Failures
- Message buffering

(This course does not go into the details of failures)

- Process failures
- Communication failures
  - Link failure
  - Loss of messages

A taxonomy of (abstract) models of distributed systems

- Network topology
- Synchrony
- Failures
- Message buffering

Delays between sending and reception of messages (latency) implies that there is some form of buffer mediating communications:

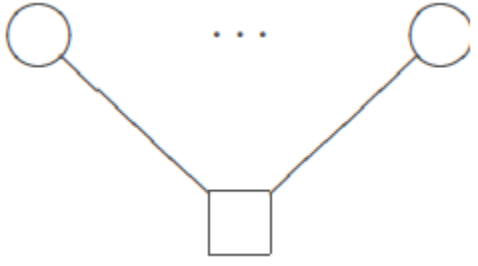
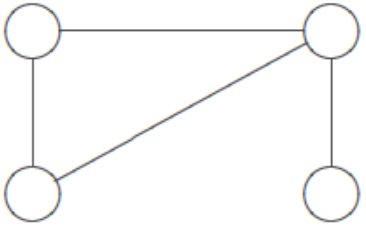
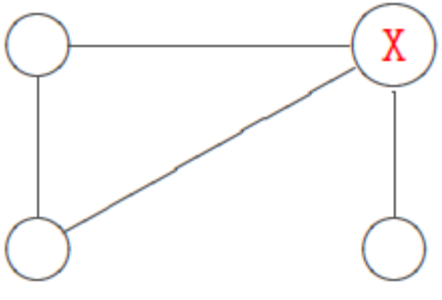
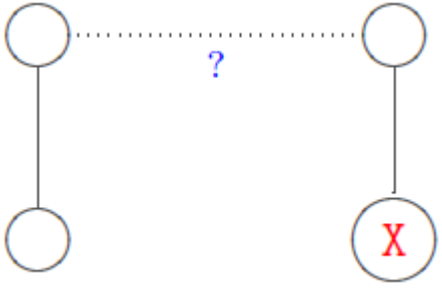
- Limited vs. unlimited buffer
- models FIFO buffering (low level)
- FIFO buffering is not usually assumed at higher level

A taxonomy of (abstract) models of distributed systems

- Network topology
- Synchrony
- Failures
- Message buffering

We'll see how (some of) the features of these models are mirrored in the middlewares.

Examples of distributed models

Shared memory multiprocessors	• Synchrony • Efficiency • Non-partial failures	
Distributed memory multiprocessors	• Off-the-shelf computers • Fast communication infrastructure • Standard LAN technology	
Distributed memory Multi-processors with partial failures	• As LAN, but • Partial failures are possible (with appropriate HW/SW)	
Open distributed systems	• Openness, i.e., WAN • Naming & security • Partial failure • loosely coupled computations	

Middlewares 1

Traditionally, distributed applications

- were made of a terminal (the client) and a mainframe (the server)
- In 1969, the Department of Defense's Advanced Research Projects Agency built a net of 4 nodes (in 1972, it grew to 50 nodes)...this was the embryonic Internet
- Shortly later (mid 70s), the exigency of "middlewares" became urgent (file transfer, remote printing...)

For an historical account of middlewares, see chapter 2 of [BB04].

Modern distributed applications

- are moving from 2-tier to 3-tier (or n-tier) architectures
- are built by composing (possibly distributed) functionalities (sometimes belonging to different domains)
- functionalities (services) have "public" interfaces
- invokers (simply) must be compliant with service interfaces
- require complex interactions (event-notification based interactions, multicast,...)

Modern applications have to tackle

distribution usually, it is meant as distribution over a wide area network (WAN) (but also over a local area network (LAN) or a metropolitan area network (MAN))

heterogeneity applications are developed using different designs, languages or executed in different execution environments providing different execution support (eg., OS and hardware)

openness applications have to be easily extensible; this is not peculiar (also non-modern applications needed to be extensible), however nowadays this aspect is more and more important and it is often linked to “dynamic binding”

scalability the possibility of expanding the number of possible clients requires applications to be able to serve a large number of requests per time unit in order to minimise latency of service invocations

inter-organisation this is an important factor with many consequences, the most important of which (for this course) is the need of middlewares (notice that this is related to heterogeneity).

What middlewares do for us?

Middlewares provide a programming environment for simplifying the task of implementing (and designing) such applications. It is hard to give a precise and general definition of middlewares...

"Middleware is the enabling technology of Enterprise application integration. It describes a piece of software that connects two or more software applications so that they can exchange data." [Mid]

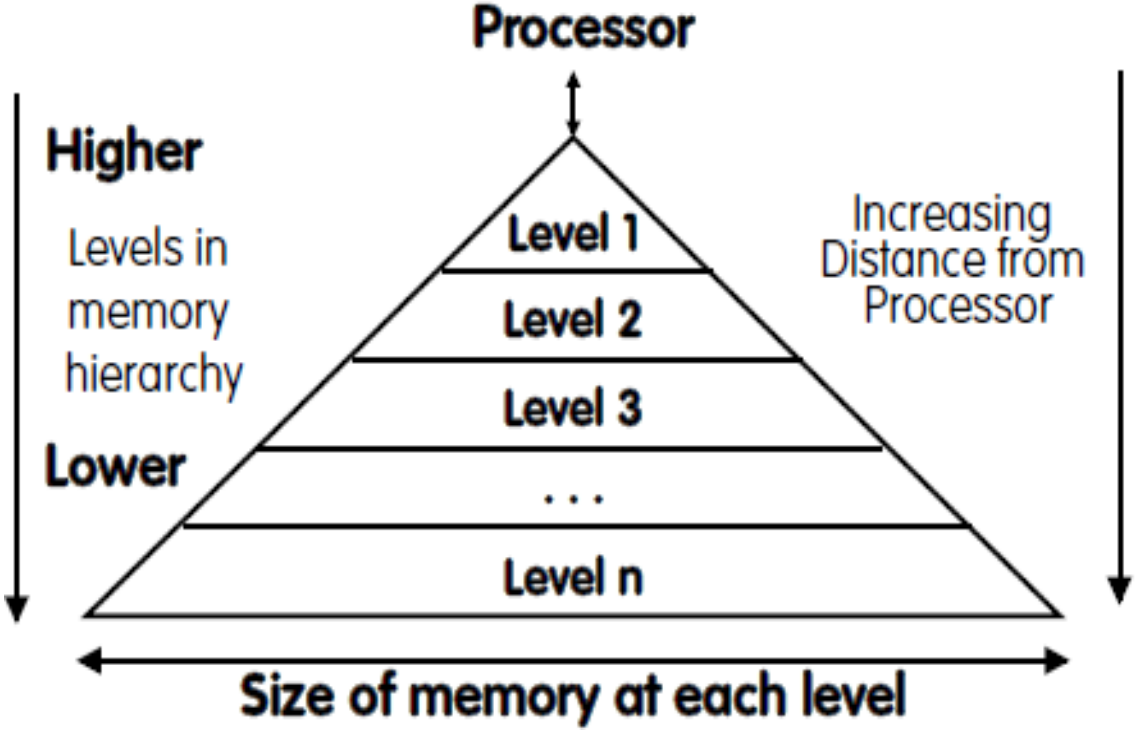
Example of middlewares

- RPC/RMI
- Publish/subscribe
- Transaction processing (TP) monitors
- Message Oriented Middlewares
- Hadoop MapReduce

  Apache Spark

Recall – Memory Hirearchy?

Memory hierarchy gives the illusion of a very large and fast memory. The reality is that memory traffic is key to performance.



If level is closer to processor, it is

- Smaller
- More expensive
- Faster

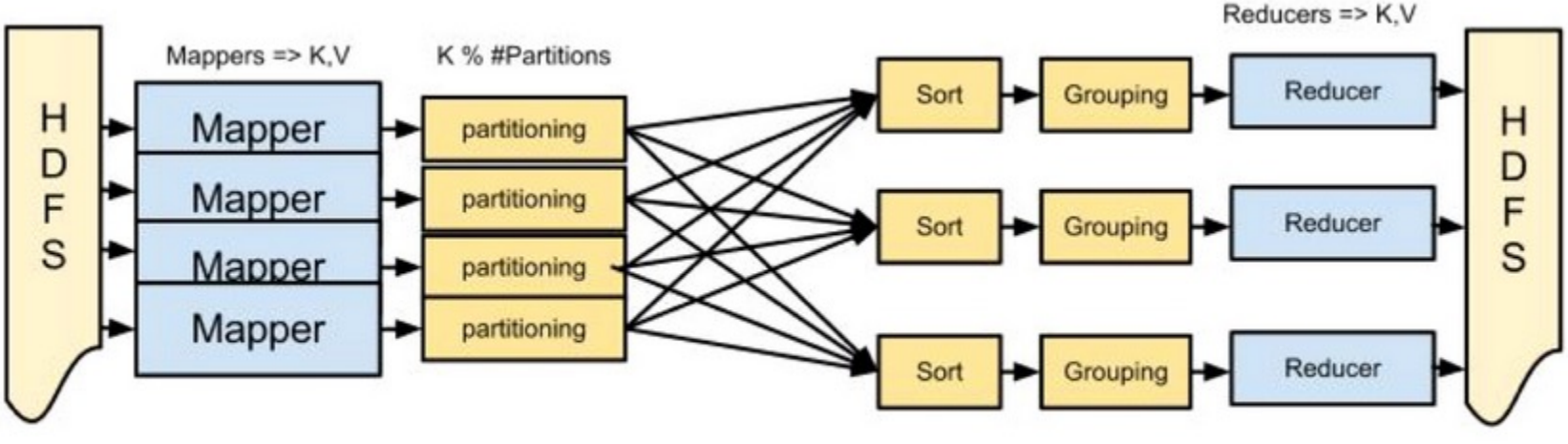
Challenges in distribution

We can summarise the challenges in distributed computing as follows

- Communication is difficult (data distribution, shared resources,
- Communicating results, handling failures, machines have different memories, network issues, etc.)
- Need to make algorithms and data structures parallel

Solution - Google's MapReduce:

- Rely on functional programming for intrinsic parallelism and data locality for faster processing



The MapReduce Pipeline

Summary & a few questions

We have informally discussed

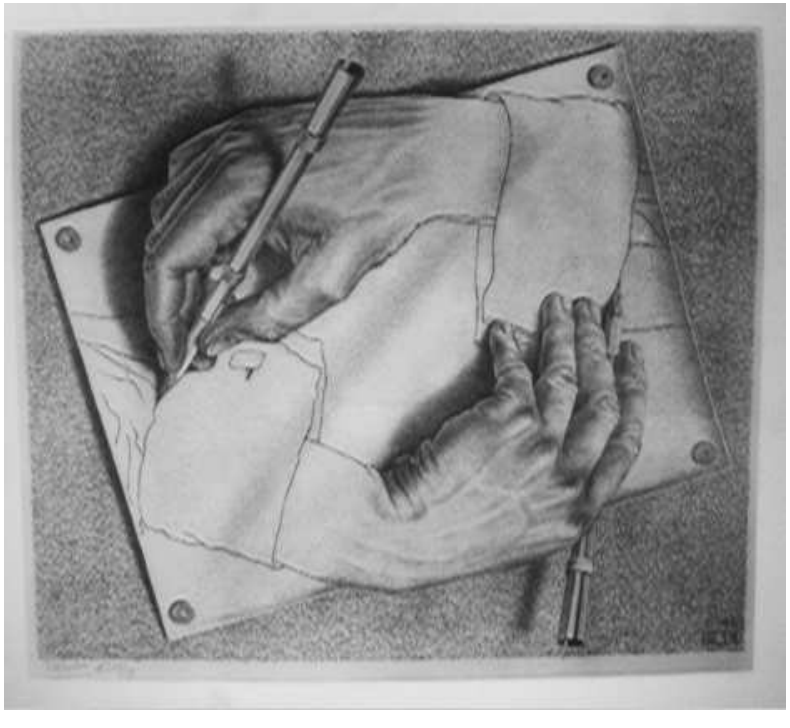
- Concurrency
- Distribution
- Middlewares

And we started to analyse their interrelationships

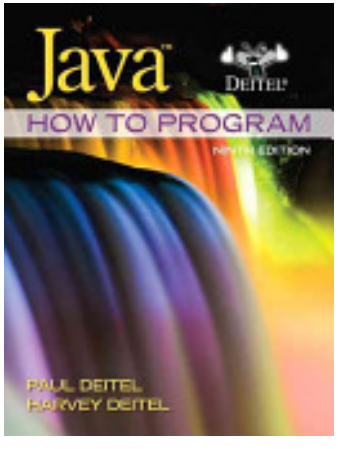
Exercise Answer the following questions:

- Is it possible to have concurrency without distribution?
- Is it possible to have distribution without concurrency?

Concurrency (in JAVA)

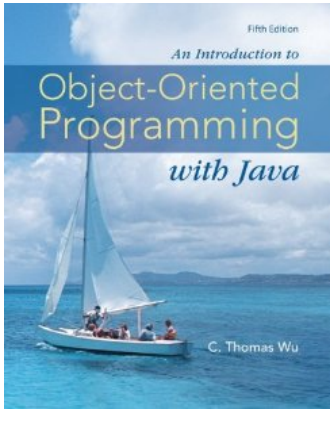


Java Books & Tutorial



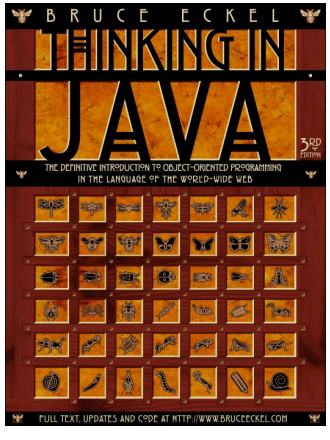
Java How to Program

Paul J. Deitel, Harvey M. Deitel



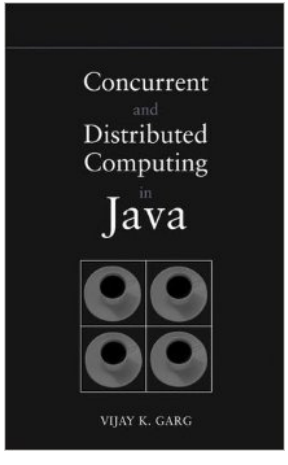
An introduction to object-oriented programming with Java

B C. Thomas Wu



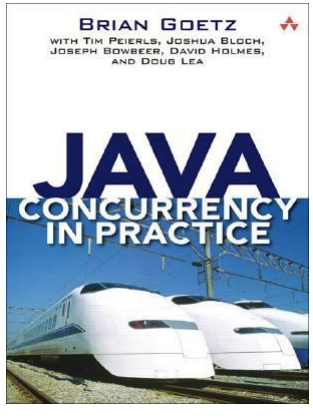
Thinking in Java

Bruce Eckel



Concurrent And Distributed Computing In Java

Garg, Vijay K.



Concurrent programming in Java: design principles and patterns

Lea, Douglas.

http://docs.oracle.com/javase/tutorial/

So far...

- overview of the module
  - concurrent programming
  - distributed applications
  - middlewares
- their relations
- Issues of distributed applications

We focus on

- concurrency
  - design
  - Java threads
- concurrency at O.S. level is different
- please review operating system concepts as presented in CO2017

So far...

- overview of the module
  - concurrent programming
  - distributed applications
  - middlewares
- their relations
- Issues of distributed applications

We focus on

- concurrency
  - design
  - Java threads
- concurrency at O.S. level is different
- please review operating system concepts as presented in CO2017

Multi-threading using

`java.lang.Thread`

http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

Question:

Is multi-threading useful on a single core machine?

Yes.

Multithreading is distinguished from multiprocessing systems (such as dual-core, quad-core systems) in that the threads have to share the resources of a single core.

- **Multiprocessing** systems include multiple complete processing units.

- **Multithreading** aims to increase utilization of a single core by using thread-level as well as instruction-level parallelism.

Concurrency

Intuitively, concurrency can be though of as a property of systems in which several "actors" (computational entities) run mostly independently and, from time to time, interact with each other.

Example 5 Let us consider again Example 2 :

P : counter = 0; Q : counter = 1; counter = counter + 1;

where we assume that P and Q are executed concurrently.

- Noteworthy, P and Q "share" a variable
- Neither P nor Q know the "execution status" of the other program
- Actually, they can even ignore that other programs are running

Terminology

- A process is a running program with its own execution environment containing basic run-time resources (e.g. the process address space, program counter, stack, heap...).

  Processes can be roughly though of as single applications, though some facilities (e.g., sockets) permit inter-process communications (which implies that you can write applications consisting of several processes),

- A multitasking OS allows many processes to be executed at the same time without them "to be aware" of each other (e.g., using time slicing, managing memory, ...)

- A thread is a single sequential flow of control within a process (a process can consist of many concurrent threads)

  Threads are also known as lightweight processes because:

  - creating a new thread requires fewer resources than creating a new process
  - Threads "live" within a process and can share the process's resources (e.g., memory, files)

  In general multi-threaded applications have a "main" thread which can create new threads

Terminology

- A process is a running program with its own execution environment containing basic run-time resources (e.g. the process address space, program counter, stack, heap...).

  Processes can be roughly though of as single applications, though some facilities (e.g., sockets) permit inter-process communications (which implies that you can write applications consisting of several processes),

- A multitasking OS allows many processes to be executed at the same time without them "to be aware" of each other (e.g., using time slicing, managing memory, ...)

- A thread is a single sequential flow of control within a process (a process can consist of many concurrent threads)

  Threads are also known as lightweight processes because:

  - creating a new thread requires fewer resources than creating a new process
  - Threads "live" within a process and can share the process's resources (e.g., memory, files)

  In general multi-threaded applications have a "main" thread which can create new threads

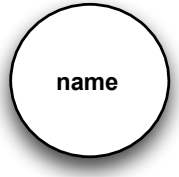
Concurrency and OO: some rule of thumb

There is no general approach to concurrent programming. However, when programming concurrent OO applications, it is good practice to

- Individuate active and passive objects
  - an active object is basically a thread object (i.e., executes the control part of a thread)
  - a passive object is a resource amenable to be accessed by active objects
- Reason about how objects interacts
  - how active objects' steps interleave?
  - how active objects access shared resources?
- Access policy for acquiring/releasing
  - in which order active objects acquire shared resources? under
  - which conditions shared resources can be invoked?
  - do active objects release all the acquired resources when they are not needed any longer?

Notation for Active/passive interaction

We shall use the following notation to design concurrent applications. The elements of the informal notation we adopt are:



Active object

Passive object

Write operation

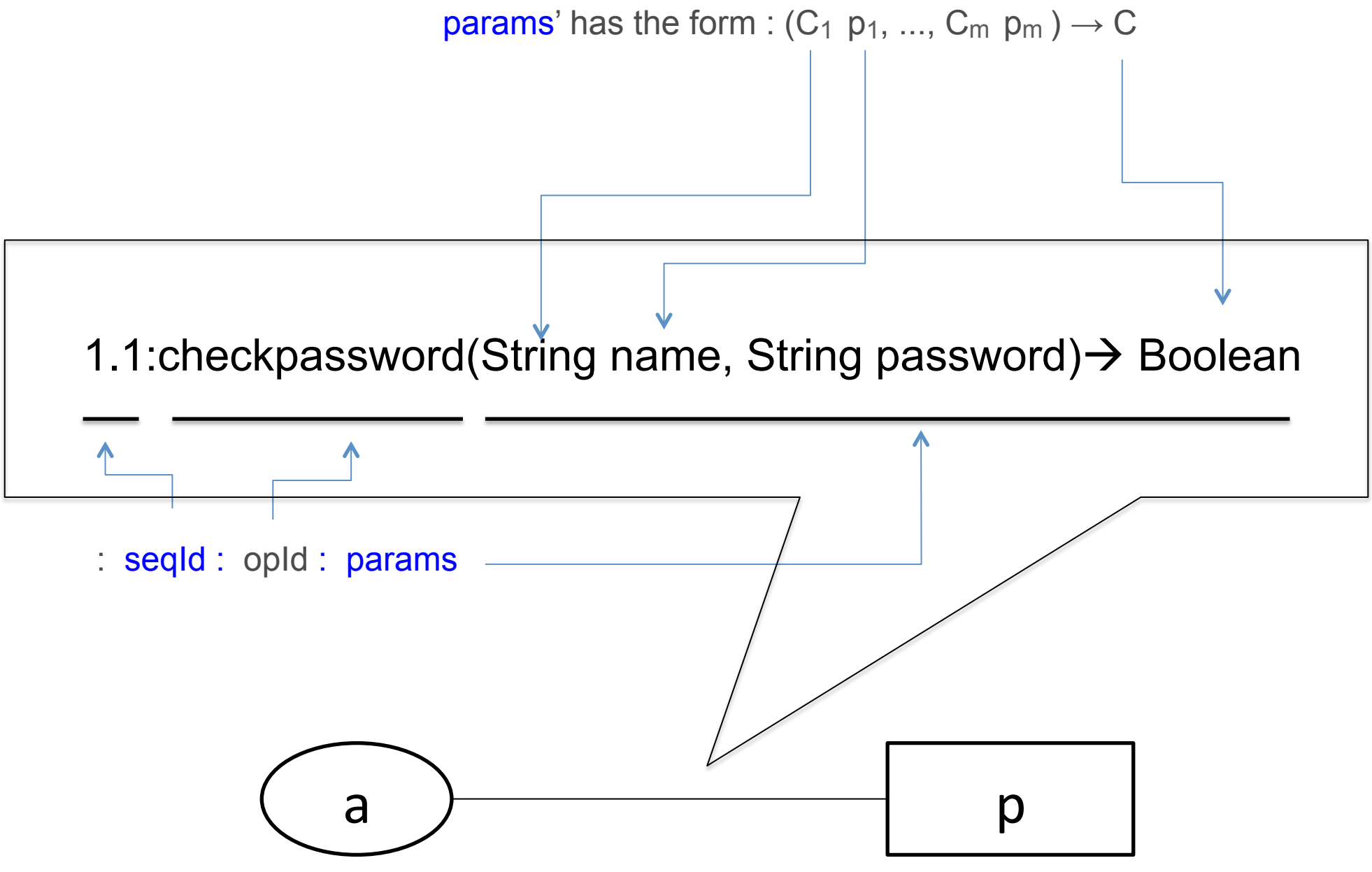
Read operation

- Active/Passive objects may have local “decorations” (objects/values)
- A label takes the form: seqId : opId : params where ’seqId:’ and ’:params’ are
- optional ’seqId:’ has the form $\alpha_1.\alpha_2. \cdot \cdot \cdot .\alpha_n$:, where αs are positive integers or strings
- ’:params’ has the form : $(C_1\ p_1, ..., C_m\ p_m) \rightarrow C$
  - it may be m = 0, in which case ’:params’ has the form : $() \rightarrow C$
  - if no value/object is returned, ’:params’ has the form : ()

Remember:

- active objects represent threads with their own independent control flow
- passive objects represent shared resources

Notation for Active/passive interaction

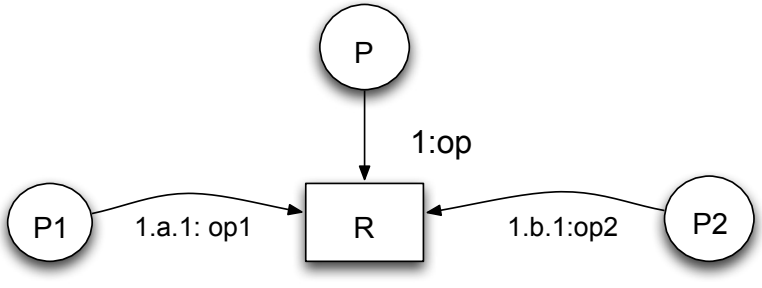


Labels (1)

Labels allow us to specify dependencies among flows of active objects.

Example 6 In $\alpha_1.\alpha_2. \cdot \cdot \cdot .\alpha_n$, α's represent the order in which operations must be executed. For instance, 1.1 : op means that op is labelled by 1.1 and it must be executed after the operation labelled 1.

Strings in labels are used to distinguish operations from different active objects when they cannot be linearly ordered.

Example 7 Consider 

then op1 and op2 are concurrently exectuted after op.

Labels must respect the following constraints to enforce consistency of diagrams:

1. two different operations cannot have the same seqId
2. seqIds from an object have the same length and differ only on the last index

Labels (2)

Labels are a bit tricky: they impose an order on the execution of some operations, but the order is not total.

1. if $\alpha_1.\alpha_2.\cdots.\alpha_n.u$: op and $\alpha_1.\alpha_2.\cdots.\alpha_n.v$: op1 are operations from the same (active) object and u < v then op is executed before op1
2. if op is labelled by $\alpha_1.\alpha_2.\cdots.\alpha_n.\alpha_{n+1}$, then op must be executed after the operation labelled $\alpha_1.\alpha_2.\cdots.\alpha_n$ (on turn executed after $\alpha_1.\alpha_2.\cdots.\alpha_{n-1}$ and so on and so forth)

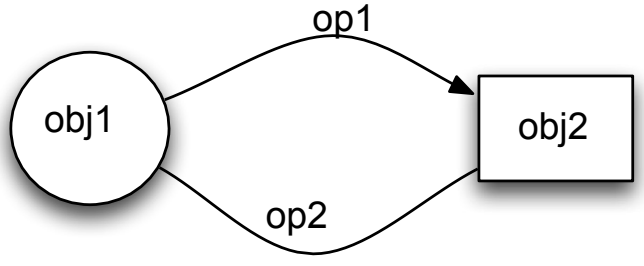
1. Operation labelled by 1.1 should be executed before operation labelled by 1.2

2. Operation labelled by 1.2 should be executed before operation labelled by 1.2.1

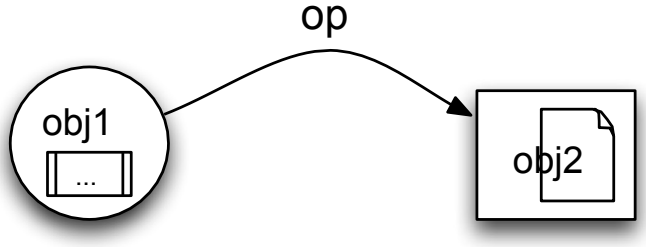
Confused?
Let's look at some examples

A few examples

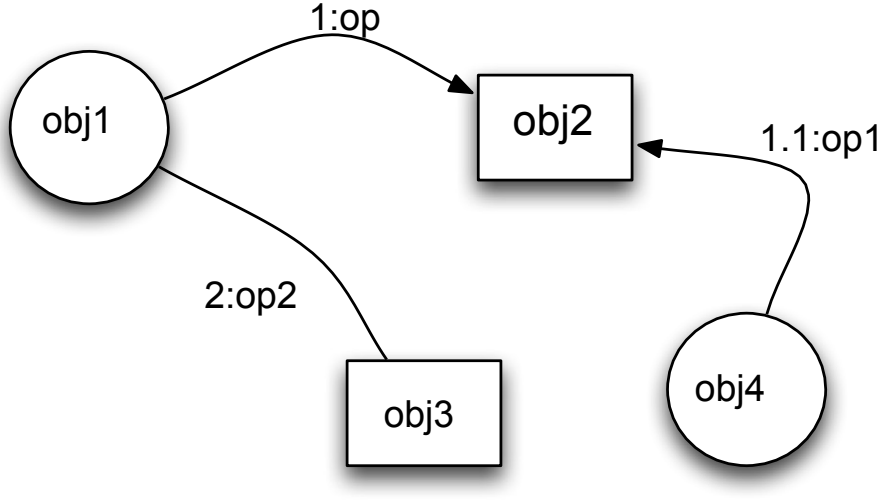
In the examples below obj1 and obj4 are active while obj2 and obj3 are passive.



- obj1 modifies obj2 with op1
- obj1 reads from obj2 with op2



- obj1 and obj2 have local data structures (an array and a file respectively)



- obj1 modifies obj2 which is then modified by obj3
- the 2nd opearation executed by obj1 is op2

Exercise 8 In the last example which is the execution order between op1 and op2?

Threads in JAVA

Two basic ways of defining JAVA threads:

- Runnable
  - interface
  - requires to implement run()
- Threads
  - class
  - implements Runnable
  - its run() method is just empty

"When something has a Runnable interface, it simply means that it has a run() method, but there's nothing special about that –it doesn't produce any innate threading abilities, like those of a class inherited from Thread." [Eck02]

Javadoc Thread

http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

Define and start a thread:

Method (1) implementing **Runnable** interface:

```
package CO3090.basic;

public class MyFirstThread1 implements Runnable{

    public static void main(String[] args){
        MyFirstThread1 runnableObj=new MyFirstThread1();
        Thread t=new Thread(runnableObj);
        t.start();
        //You can also write everything in one line
        //new Thread(new MyFirstThread1()).start();
    }

    @Override
    public void run() {
        // TODO Auto-generated method stub
         System.out.println("Hello World! I'm a thread");
    }

}
```

Steps

(1) Create a class, implement ***Runnable*** interface run() method.
(2) Create a ***runnable*** object
(3) Create a Thread object, pass the object to the constructor
(4) Call ***start()*** method*
* not ***run()*** method!

Thread()
Thread(Runnable target)
Thread(Runnable target, String name)
Thread(String name)
….

Javadoc http://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html

Define and start a thread:

Method (2) extending **Thread** class:

```
package CO3090.basic;

public class MyFirstThread2 extends Thread {

    public static void main(String[] args){

        MyFirstThread2 t=new MyFirstThread2();
        t.start();
    }

    @Override
    public void run() {
        // TODO Auto-generated method stub
         System.out.println("Hello World! I'm a thread");
    }

}
```

Steps

(1) Extends Thread class
(2) Provide your own implementation of run() method.
(3) Create an object of your thread class
(4) call obj.***start()*** *

* not ***run()*** method!

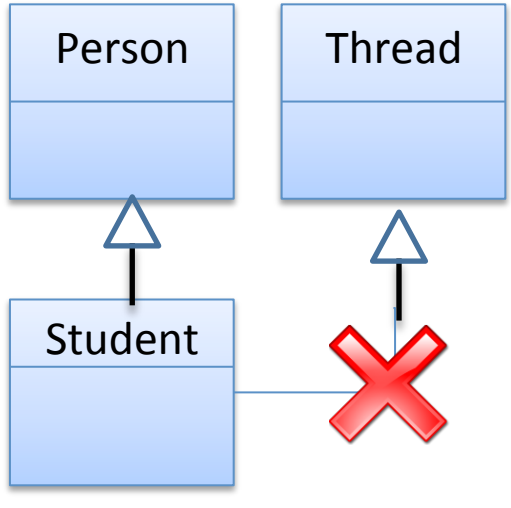
Javadoc http://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html

Question: what are the pros and cons for each method?
Implementing Runnable vs ***Extending Thread class***

preferable

Method (1) **implementing Runnable interface**

More general and flexible
Can subclass any classes other than Thread
Applicable to the high-level thread management APIs



Method (2) **Extending Thread class**

Easier to use in simple application
But class must be a descendant of Thread
(Java does NOT support multiple inheritances)

Example: Launching 2 threads

```
package CO3090.basic;

public class DemoTwoThread {

    public static void main(String[] args){


        MyFirstThread1 runnableObj=new MyFirstThread1();
        Thread t1=new Thread(runnableObj);

        MyFirstThread2 t2=new MyFirstThread2();


        t1.start();
        t2.start();

        System.out.println("This is main thread");


    }

}
```

Question 1:

How many thread in total? 3

Question 2

Which of the following output are possible***?***

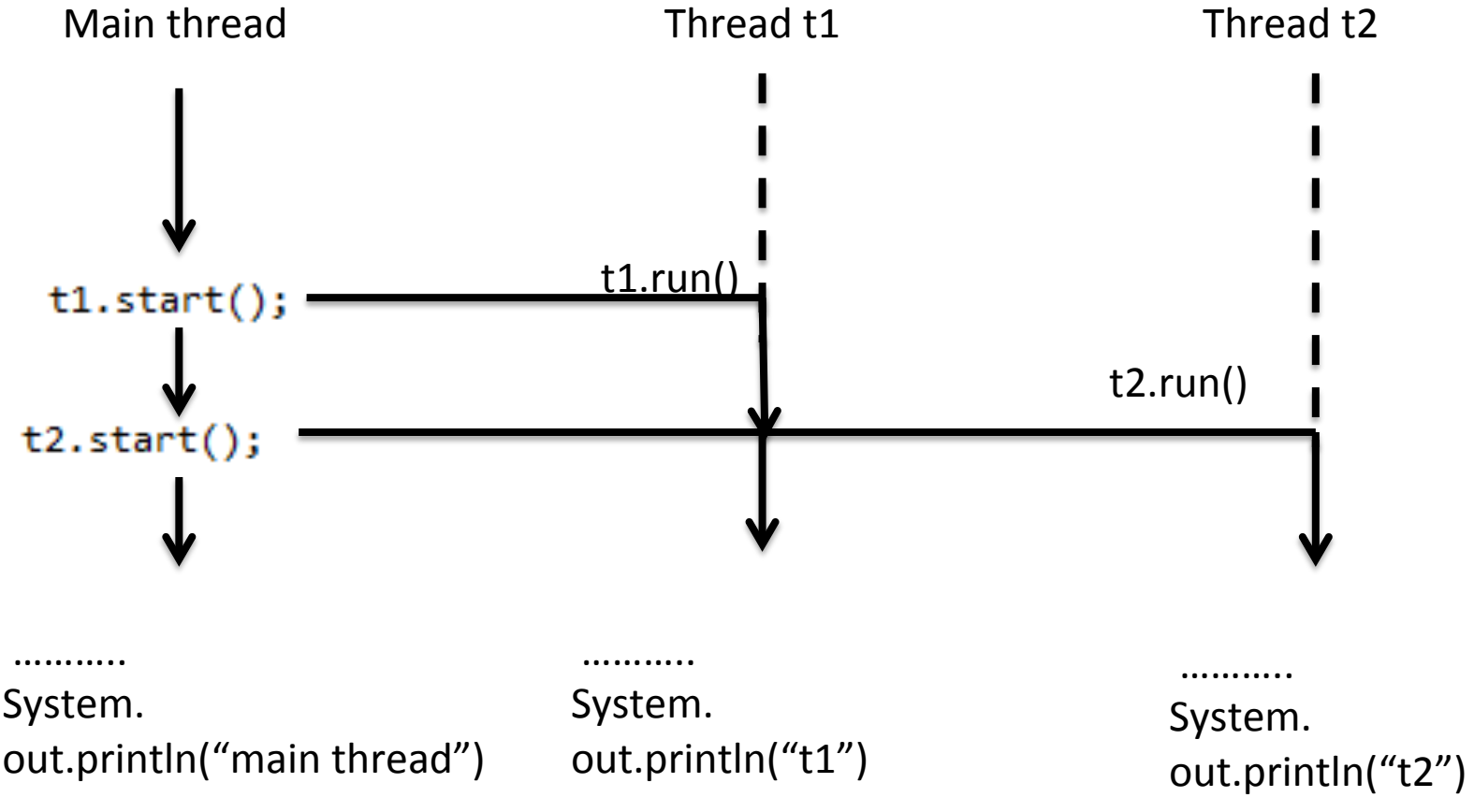
(1) (2) (3)
(1) (3) (2)
(2) (1) (3)
(2) (3) (1)
(3) (2) (1)
(3)(1) (2)

All of them

(1) Hello World! I'm a thread (by extending Runnable interface)
(2) Hello World! I'm a thread (by extending Thread)
(3) This is main thread

Execution steps

```
t1.start();
t2.start();
System.out.println("This is main thread");
```



execution order is non-deterministic

Creating Threads

To create and run JAVA thread from Threads:

- Constructors
  - Thread()
  - Thread(Runnable target)
  - Thread(Runnable target, String name)
  - Thread(String name)
  - ... and many others (see the JAVA thread API)
- invoke the start method of the thread object (which performs some initialisations and then calls the run() method)

To create and run JAVA thread from a "Runnable object":

- create the "Runnable object"
- use the special Thread constructors with runnable objects
- run the thread by invoking its start() method

Example: creating multiple threads using loops:

```
package CO3090.basic.example2;

public class ConcurrentPrint implements Runnable{

    public void run(){
        System.out.println
         ("Printed from "+ Thread.currentThread().getName());
    }

}
```

Use Thread.currentThread() to Get the current thread object, call getName() to get the current thread name. (e.g. Thread_1)

```
package CO3090.basic.example2;

public class Main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        for(int i=1;i<=10;i++){
            // create a runnable object in each iteration
            ConcurrentPrint p=new ConcurrentPrint();
            //pass the runnable object to the thread constructor:
            // Thread(Runnable target, String name)
            //named the thread "Thead_1", "Thead_2" ..."Thead_10"
            Thread t=new Thread(p, "Thread_"+i);
            t.start();
        }
    }
}
```

Possible output:

Printed from Thread_1
Printed from Thread_10
Printed from Thread_8
Printed from Thread_6
Printed from Thread_4
Printed from Thread_2
Printed from Thread_3
Printed from Thread_5
Printed from Thread_7
Printed from Thread_9

* result may vary

Controlling threads

- **interrupt()**: Interrupts the thread on which it is invoked.
- **yield()**: Occasionally, a thread can decide to "give a hint to the thread scheduling mechanism" ([Eck02]) that it is keen to pass the control to another thread.
- In JAVA this is done by invoking yield() in the run method of the thread.

join(): When invoked on a thread object, the invoking thread waits for the first thread to complete before proceeding (there is also a version with timeout).

- **join()** must be withing a try-catch statement because an interrupt() signal can abort the calling thread.
- **isAlive()**: Returns 'true' if the thread is not died yet.

sleep(long milliseconds [, int nanoseconds]): the currently executing thread temporarily ceases its execution for the **milliseconds** (more or less). If **nanoseconds** is specified, the sleeping time will be augmented with the specified number of nanoseconds.

Sleep(long milliseconds [, int nanoseconds]):

The currently executing thread temporarily ceases its execution for the **milliseconds**

```
package CO3090.basic.controls;

public class MyThread implements Runnable{

    public static void main(String[] args){

        MyThread runnableObj=new MyThread();
        Thread t=new Thread(runnableObj);
        t.start();
    }

    @Override
    public void run() {
        while(true){
        try {
            //sleep() must be wraped in a try/catch block,
            //Or declare to throw InterruptedException
            Thread.sleep(2000); //Sleep for 2 seconds
            System.out.println("This message get printed every 2 seconds");
            } catch (InterruptedException e) {
                //If the thread is interrupted, do something here
                System.out.println("Current thread has been interrupted");
                e.printStackTrace();
            }
        }

    }
}
```

Sleep() is a **static** method

Thread.sleep() always suspends the current thread

The sleep() method can throw an **InterruptedException** a so it must be placed within a **try-catch** block (or declare that your method can throw this exception)

Sleep(long milliseconds [, int nanoseconds]):
The currently executing thread temporarily ceases its execution for the **milliseconds**

Pitfall: invoking sleep() method from thread object rather than Thread class

Question: which thread is suspended after calling t1.sleep()?

```
public void main(String args[]){
 ….//t1 is a thread
t1.sleep(1000);
…..
}
```

Sleep() is a static method!

Should use Thread.sleep(100) to avoid confusion

Answer : looks like you are telling t1 to sleep but in fact, it's the main thread get suspended rather than t1, because

(1) sleep() always cause the current thread to sleep, no matter what thread object sleep ()is called on

(2) sleep() is a static method and we should use Thread.sleep() rather than threadObject.sleep() to avoid confusion.

yields(): Occasionally, a thread can decide to “give a hint to the thread scheduling mechanism” that it is keen to pass the control to another thread.

```
package CO3090.basic.controls;

public class MyThread4 implements Runnable{

    public static void main(String args[]){

        MyThread4 runnableObj=new MyThread4();

        for(int i=0;i<5;i++){
            Thread t=new Thread(runnableObj, "thread_"+i);
            t.start();
        }
    }

    @Override
    public void run() {

        System.out.println(
                Thread.currentThread().getName()+
                " is running");
        System.out.println(
                Thread.currentThread().getName()+
                " has finished execution");
    }
}
```

A possible result:

```
thread_0 is running
thread_0 has finished execution
thread_2 is running
thread_2 has finished execution
thread_4 is running
thread_4 has finished execution
thread_1 is running
thread_1 has finished execution
thread_3 is running
thread_3 has finished execution
```

X is running is usually followed by X has finished execution because sometime a thread will consume the entire CPU usage of process

yields(): Occasionally, a thread can decide to “give a hint to the thread scheduling mechanism” that it is keen to pass the control to another thread.

```
package CO3090.basic.controls;

public class MyThread4 implements Runnable{

    public static void main(String args[]){

        MyThread4 runnableObj=new MyThread4();

        for(int i=0;i<5;i++){
            Thread t=new Thread(runnableObj, "thread_"+i);
            t.start();
        }
    }

    @Override
    public void run() {

        System.out.println(
                Thread.currentThread().getName()+
                " is running");

        Thread.yield();

        System.out.println(
                Thread.currentThread().getName()+
                " has finished execution");
    }
}
```

* yield()is a static method

A possible result:

```
thread_0 is running
thread_2 is running
thread_4 is running
thread_0 has finished execution
thread_2 has finished execution
thread_4 has finished execution
thread_1 is running
thread_3 is running
thread_1 has finished execution
thread_3 has finished execution
```

Make a **suggestion** to the system to **temporarily pause the current thread to allows a context switch to other threads**, so this thread will not consume the entire CPU usage of the process. It’s likely the scheduler will select a different thread to run instead of the current one.

* There’s **NO guarantee** this will happen!

interrupt(): Interrupts the thread on which it is invoked.

```
package CO3090.basic.controls;

public class MyThread implements Runnable{

    public static void main(String[] args){

        MyThread runnableObj=new MyThread();
        Thread t=new Thread(runnableObj);
        t.start();
        t.interrupt();
    }

    @Override
    public void run() {
        while(true){
        try {
            //sleep() must be wraped in a try/catch block,
            //Or declare to throw InterruptedException
            Thread.sleep(2000); //Sleep for 2 seconds
            System.out.println("This message get printed every 2 seconds");
            } catch (InterruptedException e) {
                //If the thread is interrupted, do something here
                System.out.println("Current thread has been interrupted");
                e.printStackTrace();
            }
        }

    }
}
```

Execution result

```
Current thread has been interrupted
java.lang.InterruptedException: sleep interrupted
        at java.lang.Thread.sleep(Native Method)
        at CO3090.basic.controls.MyThread.run(MyThread.java:19)
        at java.lang.Thread.run(Unknown Source)
This message get printed every 2 seconds
This message get printed every 2 seconds
This message get printed every 2 seconds
```

t.interrupt() : interrupt thread t

isAlive(): Returns 'true' if the thread is not died yet.

```
package CO3090.basic.controls;

public class MyThread2 implements Runnable{

    public static void main(String[] args){

        MyThread2 runnableObj=new MyThread2();
        Thread t=new Thread(runnableObj);
        t.start();

        //check if the thread is still running
        System.out.println("Is t still running? "+ t.isAlive());

        try {  //sleep for 2 seconds
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //check is the thread is still alive after the sleep
        System.out.println("Is t still running? "+ t.isAlive());
    }

    @Override
    public void run() {
        //this thread prints a message and die
        System.out.println("The execution is about to complete.");
    }
}
```

t have been started while main thread will continues

Check if t is alive

Sleep for 2 seconds

Check if t is still alive

Print a message would normally takes a few milliseconds

Output:

```
Is t still running? true
The execution is about to complete.
Is t still running? false
```

join(): When invoked on a thread object, the invoking thread waits for the first thread to complete before proceeding.

```
package CO3090.basic.controls;

public class MyThread3 implements Runnable {

    public static void main(String args[]){

        MyThread3 runnableObj1=new MyThread3();
        MyThread3 runnableObj2=new MyThread3();
        Thread t1=new Thread(runnableObj1, "t1");
        Thread t2=new Thread(runnableObj2, "t2");
        t1.start();
        t2.start();
        System.out.println("main thread.");
    }

    @Override
    public void run() {
        try {
            //sleep for 1 second
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    System.out.println(
     Thread.currentThread().getName()+" thread");
    }
}
```

Usage:

t.join() : **current** thread must wait for the **t** thread to finish

Question 1

What is the output of this program?

The results could be either (1) or (2)

(1)
```
main thread.
t1 thread
t2 thread
```

(2)
```
main thread.
t2 thread
t1 thread
```

Question 2

Can you modify the code using join() to force the main thread to wait for the completion of t1 and t2.

join(): When invoked on a thread object, the invoking thread waits for the first thread to complete before proceeding.

```
package CO3090.basic.controls;

public class MyThread3 implements Runnable {

    public static void main(String args[]){

        MyThread3 runnableObj1=new MyThread3();
        MyThread3 runnableObj2=new MyThread3();
        Thread t1=new Thread(runnableObj1, "t1");
        Thread t2=new Thread(runnableObj2, "t2");
        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("main thread.");
    }

    @Override
    public void run() {
        try {
            //sleep for 1 second
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

     System.out.println(
      Thread.currentThread().getName()+" thread");
     }
}
```

Usage:

t.join() : **current** thread must wait for the **t** thread to finish

Similar to sleep() metod,
Join() method always needs a try-catch block around it because it can throw an InterruptedException

Possible execution results:

(1)
```
t1 thread
t2 thread
main thread.
```

(2)
```
t2 thread
t1 thread
main thread.
```

Controlling threads

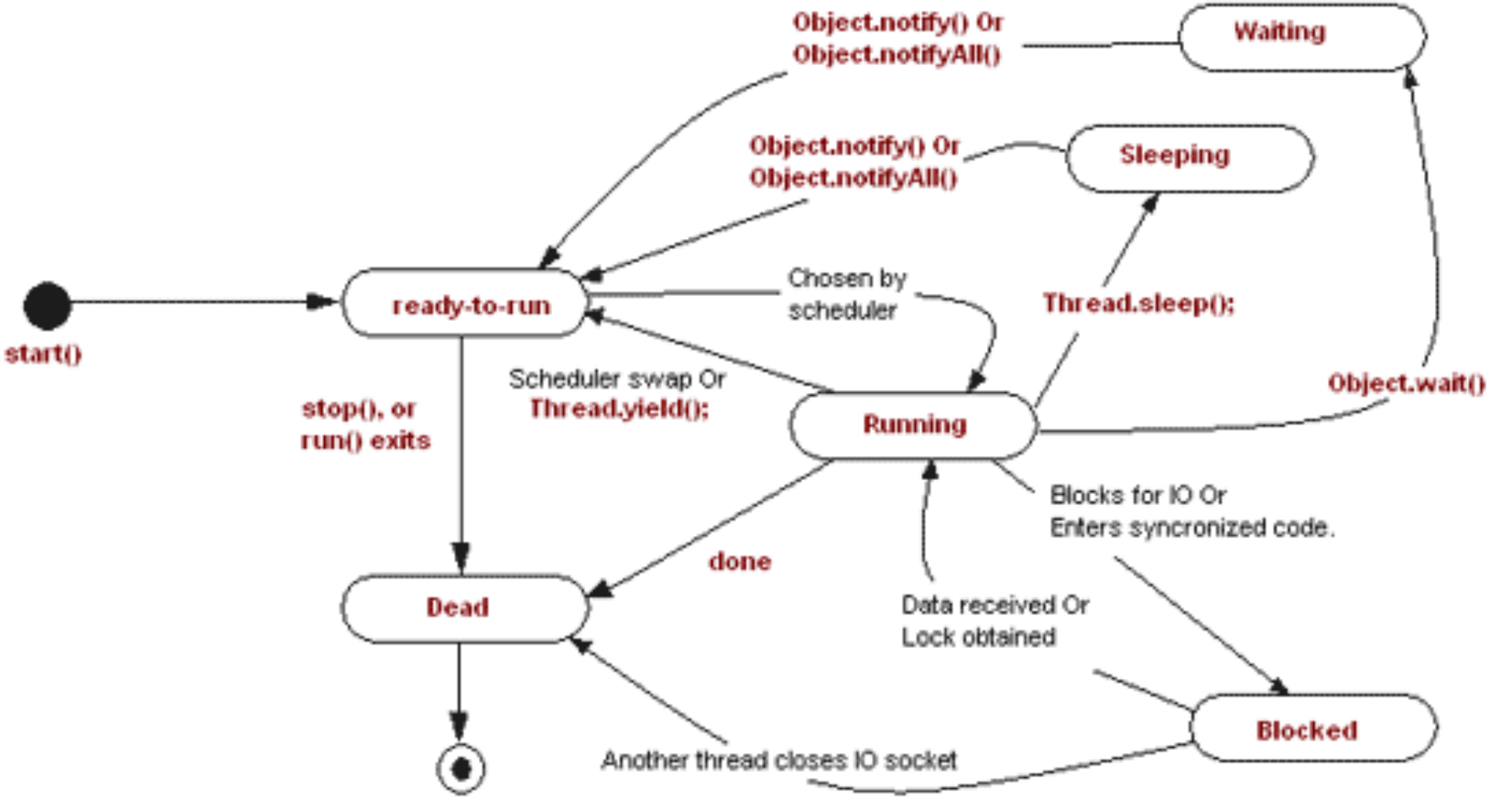
- **interrupt()**: Interrupts the thread on which it is invoked.
- **yield()**: Occasionally, a thread can decide to "give a hint to the thread scheduling mechanism" ([Eck02]) that it is keen to pass the control to another thread.
- In JAVA this is done by invoking yield() in the run method of the thread.

join(): When invoked on a thread object, the invoking thread waits for the first thread to complete before proceeding (there is also a version with timeout).

- **join()** must be withing a try-catch statement because an interrupt() signal can abort the calling thread.
- **isAlive()**: Returns 'true' if the thread is not died yet.

sleep(long milliseconds [, int nanoseconds]): the currently executing thread temporarily ceases its execution for the **milliseconds** (more or less). If **nanoseconds** is specified, the sleeping time will be augmented with the specified number of nanoseconds.

Thread life cycle in Java

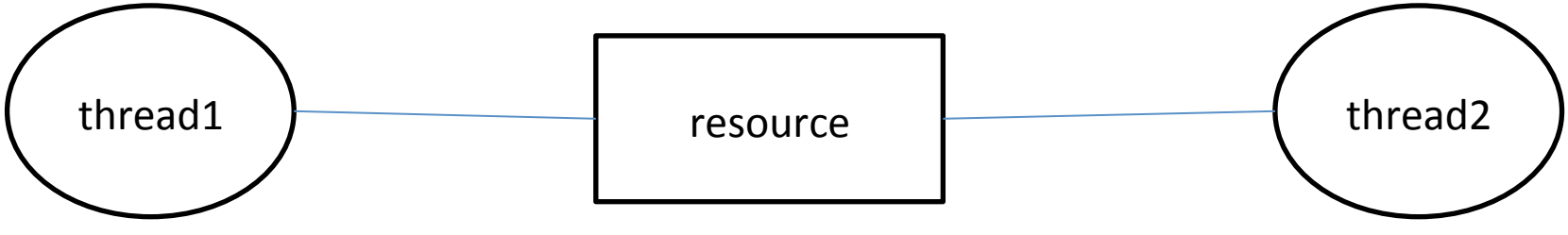


*borrowed from [REF]

So far we have learnt how to …

- Create a thread
- Start a thread
- Thread execution controls
  - sleep(), interrupt(), yield(), join()
- Thread state transitions

Now: accessing shared objects

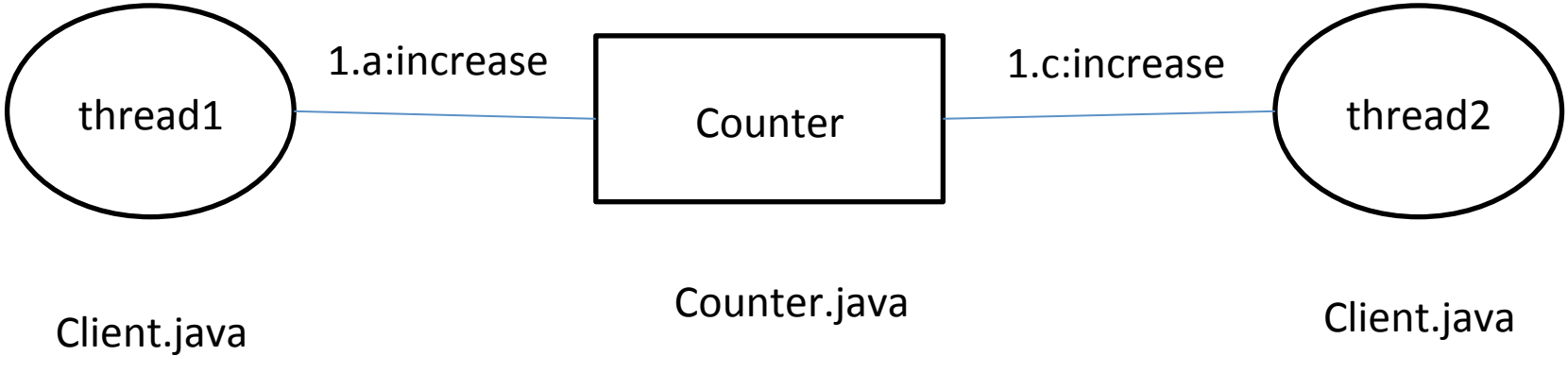


Sharing resources among threads

Example 1 – Website counter

Suppose 2 Clients - thread 1 and thread 2 are running in parallel, counter is an object shared by both threads.

Counter should increase every time it was accessed.



Problem with shared resource

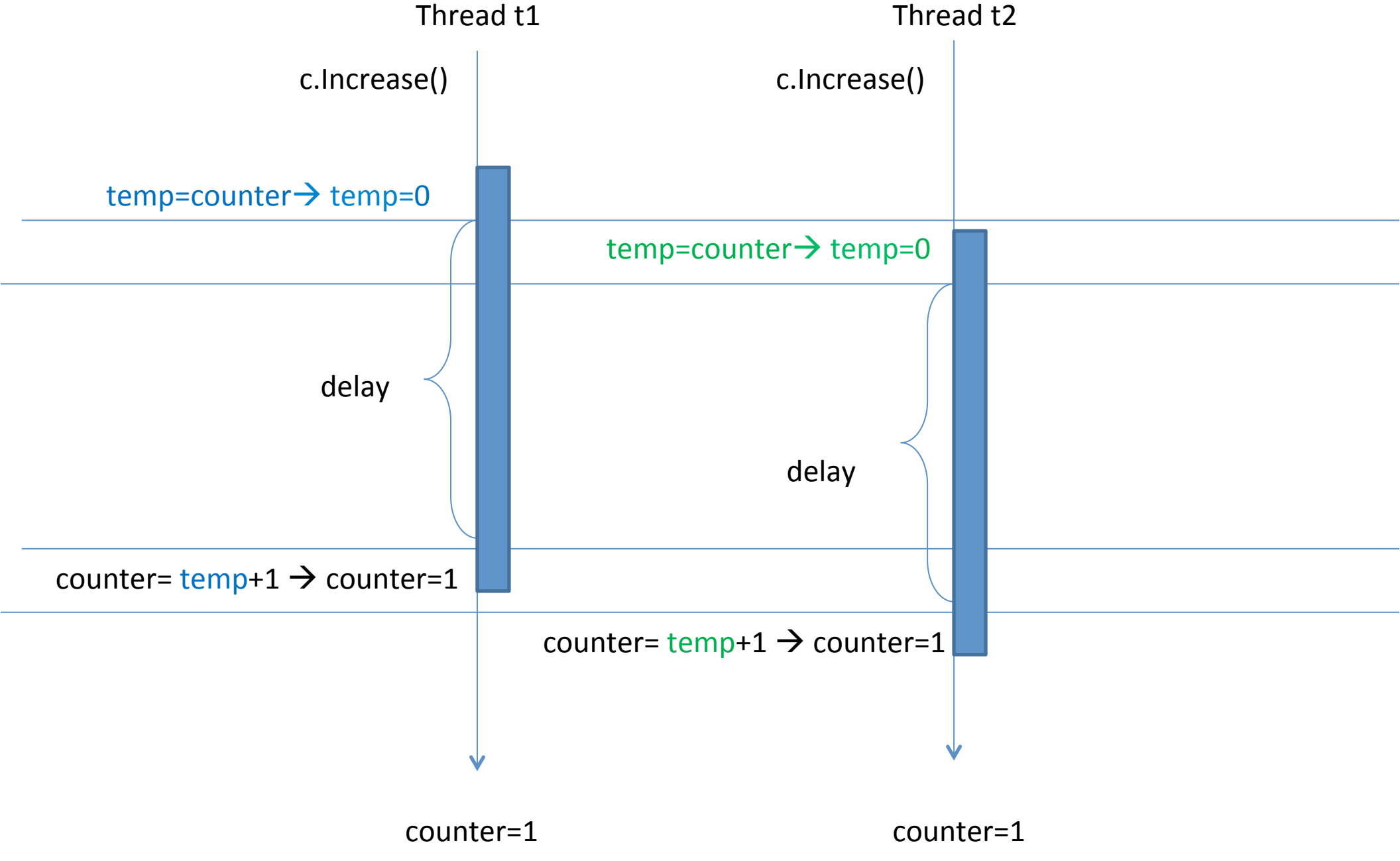
```java
package CO3090.basic.example.counter;

public class Counter {

    int visits=0;

    public void increase(){

    int temp=visits;

    //We use Thread.sleep(100) to simulate
    //a time delay in update
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        visits=temp+1;
    }

    public int getCounterValue(){
        return visits;
    }
}
```

What's the result?
Counter =1? Counter =2?

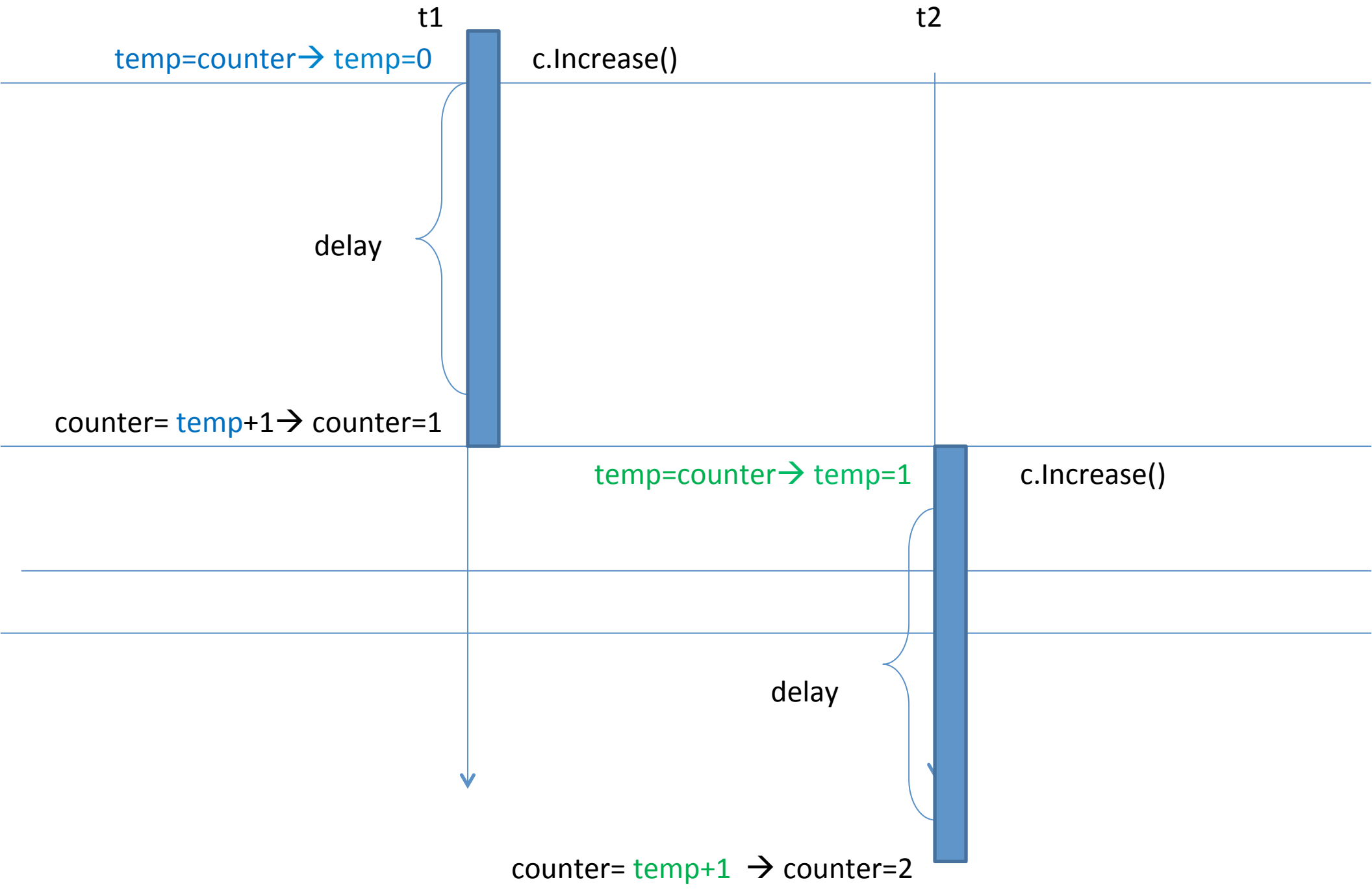
```java
package CO3090.basic.example.counter;

public class Client implements Runnable{
    Counter c;

    public Client(Counter counter){
        c=counter; //assign counter
    }

    @Override
    public void run() {
        //increase the value of counter by 1
        c.increase();
    }

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

        Counter shared_counter=new Counter();

        Client runnableObj1=new Client(shared_counter);
            Thread t1=new Thread(runnableObj1, "client_1");
        Client runnableObj2=new Client(shared_counter);
            Thread t2=new Thread(runnableObj2, "client_2");

        t1.start();
        t2.start();

        t1.join();
        t2.join();
        //main thread has to wait until t1 t2 complete
        //their executions
        System.out.println

             ("Counter="+ shared_counter.getCounterValue());
    }
}
```

Problem with shared resource

Access to counter is not exclusive



Solution


t1
t2
temp=counter→ temp=0
c.Increase()
delay
counter= temp+1→ counter=1
temp=counter→ temp=1
c.Increase()
delay
counter= temp+1 → counter=2

Solution – Synchronized method

```
public synchronized void increase(){

int temp=visits;

//We use Thread.sleep(100) to simulate
//a time delay in update
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    visits=temp+1;
}
```

Note: constructors cannot be synchronized

A synchronized method is used to prevent one thread from modifying a shared resource while another thread is updating It.

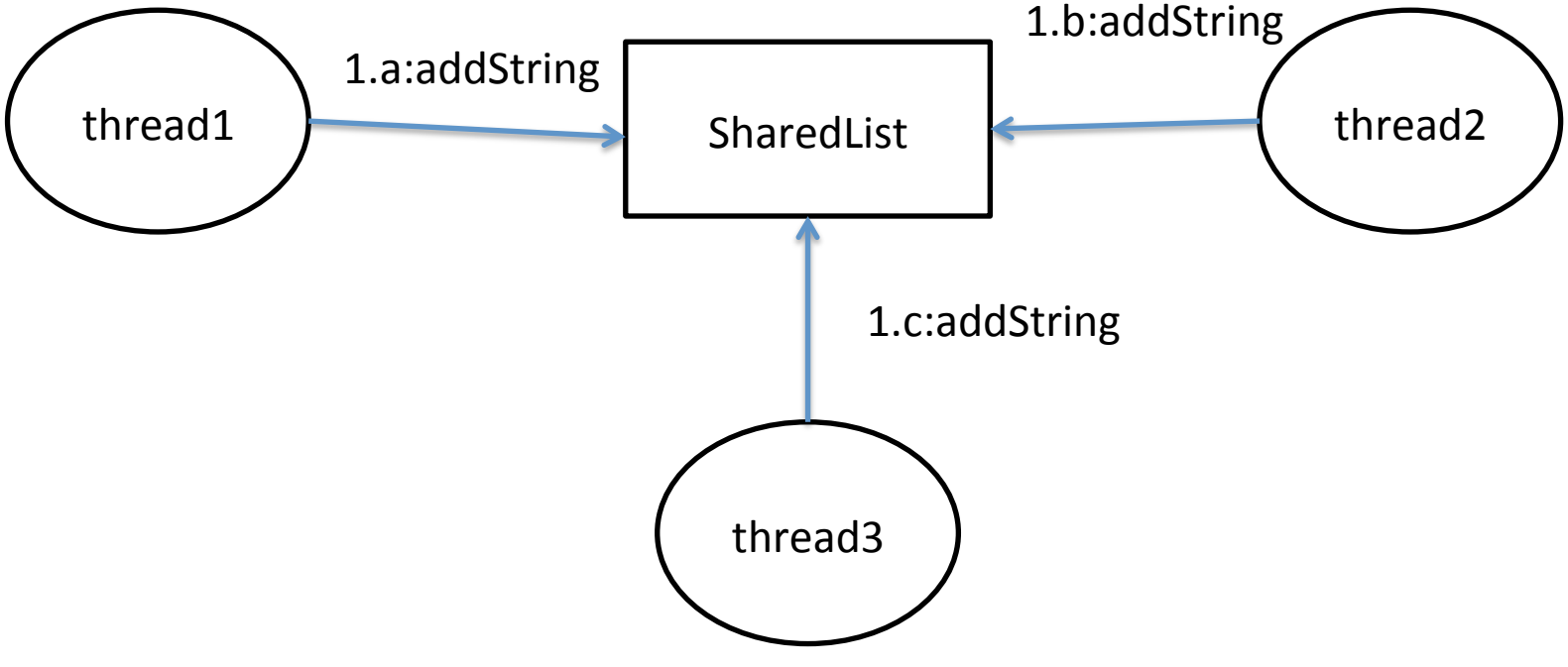
Only ONE thread at a time can execute inside a synchronized instance method.

If more than one threads exist, then one thread at a time can execute inside a synchronized instance method per instance.

Sharing resources among threads

Example 2:

Create and start 3 threads and each thread should add a string "**Hello from thread XX**" to a shared list. (it doesn't matter in which order you insert the string to the list)



Note: synchronized access problem is not considered in this example

Sharing resources among threads

Step (1)

Identify passive objects and create a corresponding class for shared resource

Identify active objects and implement its runnable interface

Why?

```
package CO3090.basic.controls;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class SharedList {

    //Returns a synchronized (thread-safe) ArrayList
    //Each of the methods of this ArrayList is synchronized
     List<String> list=
         Collections.synchronizedList(new ArrayList<String>());

    // Map<Integer,String> map=
    //    Collections.synchronizedMap(new HashMap<>());

     //add a new string to the list
     public synchronized void addStringToList(String str){
         list.add(str);
     }
     //print all strings in the list
     public void printAllStrings(){
         for(String str:list){
             System.out.println(str);
         }
     }
}
```

Step 1.1

Find appropriate data structure to store the information. If shared resource is a Java Collection (List, Map, Set etc) then obtain a synchronized version

Step 1.2

Make sure that any changes to this object can only be done via synchronized methods

Note: concurrent collection does not guarantee thread safe.

Sharing resources among threads

Step 1.1 Choose appropriate thread-safe collection types

The Collections package in Java can give us some more options to wrap existing collections, all accesses to the collection should be wrapped in synchronized methods.

Primitive data type:

```
Int counter=0
StringBuffer sb=new StringBuffer();
```

Synchronized lists/sets/maps:

```
List<String> list=
        Collections.synchronizedList(new ArrayList<String>());

Set<String> set=
     Collections.synchronizedSet(new HashSet<String>());

Map<Integer,String> map=
     Collections.synchronizedMap(new HashMap<Integer,String>());
```

Sharing resources among threads

Step (1)

Identify passive objects and create a corresponding class for shared resource

Identify active objects and implement its runnable interface

Why?

```
package CO3090.basic.controls;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class SharedList {

    //Returns a synchronized (thread-safe) ArrayList
    //Each of the methods of this ArrayList is synchronized
     List<String> list=
         Collections.synchronizedList(new ArrayList<String>());

    // Map<Integer,String> map=
    //    Collections.synchronizedMap(new HashMap<>());

     //add a new string to the list
     public synchronized void addStringToList(String str){
         list.add(str);
     }
     //print all strings in the list
     public void printAllStrings(){
         for(String str:list){
             System.out.println(str);
         }
     }
}
```

Step 1.1

Find appropriate data structure to store the information. If shared resource is a Java Collection (List, Map, Set etc) then obtain a synchronized version

Step 1.2

Make sure that any changes to this object can only be done via synchronized methods

Sharing resources among threads

Step (2)

Pass shared object(s) to threads through constructors (or setter methods).

```
package CO3090.basic.controls;

public class MyThread5 implements Runnable{

    SharedList list;

    public MyThread5(SharedList sl){
        list=sl;
    }

    @Override
    public void run() {
        list.addStringToList
         ("Hello from "+ Thread.currentThread().getName());
    }

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

        SharedList list=new SharedList();
        Thread[] ts=new Thread[3];

        for(int i=0;i<3;i++){
            MyThread5 runnableObj=new MyThread5(list);
            ts[i]=new Thread(runnableObj, "thread_"+i);
            ts[i].start();
        }
        for(int i=0;i<3;i++){
            ts[i].join();
        }

        System.out.println("Strings in the list");
        list.printAllStrings();
    }

}
```

Step 2.1
Creating a new property of shared object type.
e.g. SharedList

Step 2.2
Adding parameter to the constructor so that shared object can be passed to the thread

Step 2.3
In run method, update shared object ONLY through object's synchronized methods

Sharing resources among threads

```
package CO3090.basic.controls;

public class MyThread5 implements Runnable{

    SharedList list;

    public MyThread5(SharedList sl){
        list=sl;
    }

    @Override
    public void run() {
        list.addStringToList
         ("Hello from "+ Thread.currentThread().getName());
    }

    public static void main(String args[])
                        throws InterruptedException{
        SharedList list=new SharedList();
        Thread[] ts=new Thread[3];

        for(int i=0;i<3;i++){
            MyThread5 runnableObj=new MyThread5(list);
            ts[i]=new Thread(runnableObj, "thread_"+i);
            ts[i].start();
        }
        for(int i=0;i<3;i++){
            ts[i].join();
        }

        System.out.println("Strings in the list");
        list.printAllStrings();
    }

}
```

```
package CO3090.basic.controls;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class SharedList {

    //Returns a synchronized (thread-safe) ArrayList
    //Each of the methods of this ArrayList is synchronized
     List<String> list=
         Collections.synchronizedList(new ArrayList<String>());

    // Map<Integer,String> map=
    //    Collections.synchronizedMap(new HashMap<>());

     //add a new string to the list
     public synchronized void addStringToList(String str){
         list.add(str);
     }
     //print all strings in the list
     public void printAllStrings(){
         for(String str:list){
             System.out.println(str);
         }
     }
}
```

Output:

```
Strings in the list
Hello from thread_0
Hello from thread_1
Hello from thread_2
```

Surgery 1

Exercise

You are asked to implement a thread-safe program for online registration that is able to handle concurrent user creation.

Before adding new accounts to the SharedList, your program should double check if:

(1) Account id already exists on the list
(2) Another account is already registered with this email address

- Assume that the user account has two fields `account_id` and `email`

http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html

So far...

- threads in OO programming
- Threads and Runnable definition/
- execution of JAVA threads
- review of some JAVA primitives for concurrent programming

- interference
- mutual exclusion
- deadlock

Thread interference

An extremely efficient way for threads' communication is through resource sharing...but there are two main possible errors:

- race conditions: thread interference (errors are introduced when multiple threads access shared data)
- memory consistency errors (errors from inconsistent views of (some) objects' state)

Remark 3 Therac-25 Race conditions are very subtle and dangerous. Therac-25 was a radiation therapy machine.

- At least six known accidents have happened between 1985 and
- 1987 some patients were given massive overdoses of radiation
- In some accidents patients died of the rays overdoses

Among the main causes

"The equipment control task did not properly synchronize with the operator interface task, so that race conditions occurred if the operator changed the setup too quickly. This was evidently missed during testing, since it took some practice before operators were able to work quickly enough for the problem to occur!" [Ter]

Memory inconsistency

```
//thread-safe ArrayList
//all methods including add()
//remove() are synchnorinzed

List<String> list=
  Collections.synchronizedList
        (new ArrayList<String>());
……..

public void run(){
  1 if(!list.isEmpty()){
  2     String value=list.get(0);
  3     System.out.println(value);
  4 }
}
……
```

Question:
Is this program thread-safe?

run() method did not modify any shared variable

No .
This program might throw ***ArrayIndexOutofBoundsException***

The program is NOT guaranteed to be thread-safe even the program only use thread safe collection classes.
(e.g. Another thread might be updating the list between line 1 and line 2.)

Interference

"Interference happens when two operations, running in different threads, but acting on the same data, interleave. This means that the two operations consist of multiple steps, and the sequences of steps overlap." [Sunb]

Consider the following example taken from [MK06] (Chapt. 4)

```
class Counter {
    int value=0;
    Counter(int value) {
        this.value=value;
    }
    void increment() {
        int temp = value;
        Thread.yield();
        value=temp+1;
    }
}
```

```
class Turnstile extends Thread
    { Counter people;
    Turnstile(Counter c) { people = c; }
    public void run() {
        try{
            for (int i=1;i<=Garden.MAX;i++){
                Thread.sleep(500); //0.5 second
                people.increment();
            }
        } catch (InterruptedException e) {}
    }
}
```

Calling Thread.yield() is similar to calling Thread.sleep(0)

Interference (2)

"In real concurrent programming, interference bugs are extremely difficult to locate. They occur infrequently, perhaps due to some specific combination of device interrupts and application I/O requests. They may not be found even after extensive testing." [MK06]

The problems to solve are:

- concurrent access to shared resources
- atomic access

Exercise 15 Consider the following class

```
class Counter2 {

    private int c = 0;

    public void increment()
        { c++;
    }
}
```

Is it possible that Counter2 causes thread interference in the garden example?

Is it possible that Counter2 causes thread interference in the garden example?

Answer: Yes

It is possible that self-increment operator in could compile to the java bytecode similar to the following assembly code, which consists of more than one instruction:

```
mov    eax, data
inc    eax
mov    data, eax
……
```

The point is, even a single-line self-increment statement might still contain many instructions.

Mutual exclusion (mutex)

- Mutual exclusion mechanisms are used to avoid the simultaneous use of resources by more than one execution flow that can run concurrently.
- Such parts of code are called critical sections.

The general idea of mutual exclusion is to avoid that 2 (or more threads) can concurrently execute a part of code where one of them modifies a shared resource.

Remark 4 (Semaphores) You have heard of semaphores from courses on O.S. (e.g., CO2017). Semaphores are a low-level mechanism for ensuring mutual exclusion. We shall not discuss them, but it is worth to (re)consider how they works (see [MK06] Chapt. 5, Sec. 2).

Mutual exclusion (mutex)

Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*. To ensure mutual exclusion, the mechanism offered by Java are **synchronized methods** and **synchronized statements.**

Method (1)

Synchronized methods

public void synchronized method(){ … }

"…When you call any *synchronized method*, **that object** is **locked** and no other synchronized method of **that object** can be called until the first one finishes and releases the lock. "

Method (2)

Synchronized statements (or synchronized blocks)

Synchronized(object){….. }

"… Another way to create synchronized code is with *synchronized statements*. Unlike synchronized methods, synchronized statements must **specify the objec**t that provides the intrinsic lock…"

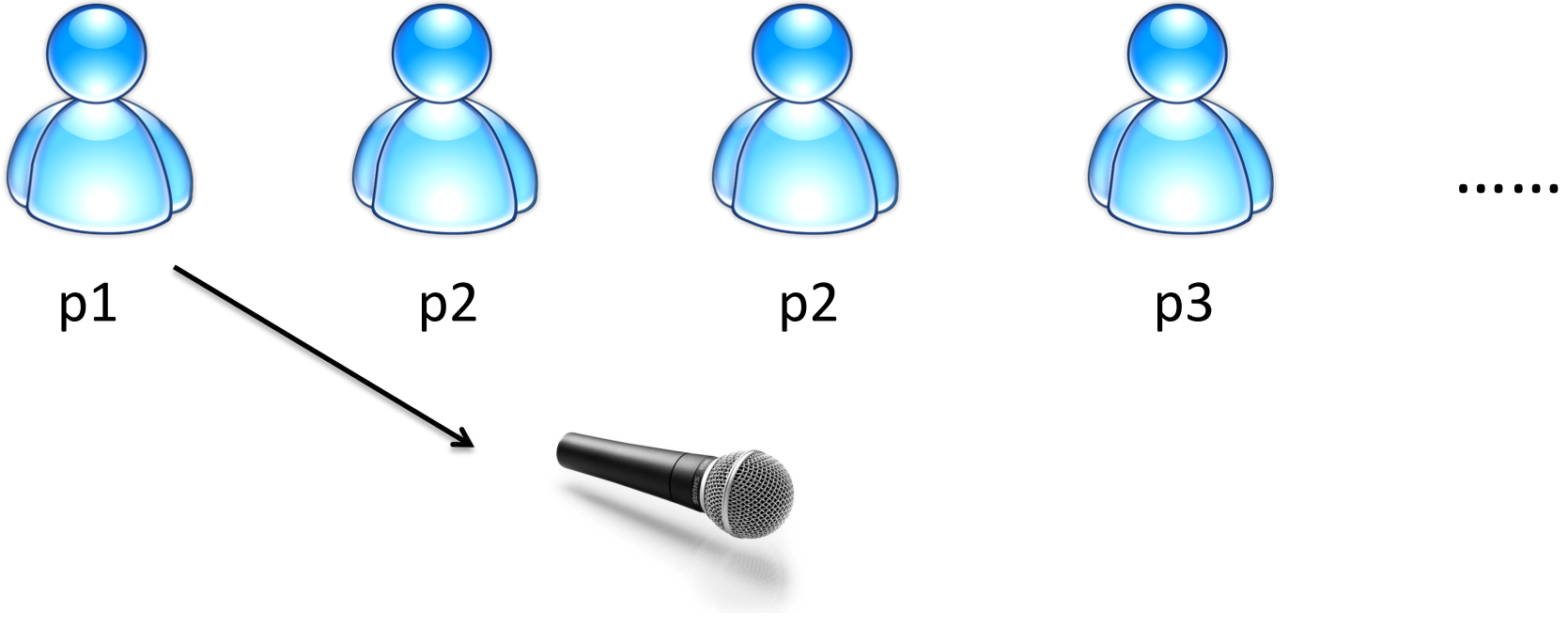
Synchronization is always associated to an object (lock)

Recall Lab exercise 2 .....

How to ensure only one person speaks at a time?

Rule (2): Only the person holding the microphone (or talking stick) is allowed to speak.

Rule (2): No two people will share one talking stick.



(Talking stick) microphone can be thought of as a "Lock object"

Synchronized methods

Usage:

```
public void synchronized method(){ … }
```

Synchronisation is based on (implicit) locks.

A synchronized method is synchronized on the instance (object) owning the method (AKA “this” in Java)

The entire body of a method is synchronized .

Synchronized keyword will prevent **any synchronized methods of the same object** from getting called at the same time.

Synchronized methods

*"..When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for **the same object** will block until invocation is finished...."*

```
public class Counter {

    int number;

    public synchronized void increase() {
        number++;
    }
    public synchronized void decrease() {
        number--;
    }

}
```

```
Counter c1=new Counter();
Counter c2=new Counter();
```

Question:
Can you call the these methods at the same time from different threads?

(A) *c1.increase()*
c1.increase()

(B) *c1.increase()*
c1.decrease()

(C) *c1.increase()*
c2.increase()

(D) *c1.increase()*
c2.decrease()

Synchronisation is based on (implicit) locks.
The entire body a method is synchronized using the current object lock.

Synchronized methods

*"..When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for **the same object** will block until invocation is finished...."*

```
public class Counter {

    int number;

    public synchronized void increase() {
        number++;
    }
    public synchronized void decrease() {
        number--;
    }

}
```

```
Counter c1=new Counter();
Counter c2=new Counter();
```

Question:
Can you call the these methods at the same time from different threads?

(A) *c1.increase()*
c1.increase()

(B) *c1.increase()*
c1.decrease()

(C) *c1.increase()*
c2.increase()

(D) *c1.increase()*
c2.decrease()

Synchronisation is based on (implicit) locks.
The entire body a method is synchronized using the current object lock.

Synchronized methods

Calling a synchronized method from another synchronized method ..

```
package CO3090.basic.control;

public class SyncMethodDemo{

    public static void main(String args[]){
        SyncMethodDemo obj=new SyncMethodDemo();
        obj.methodA();
    }

    public synchronized void methodA(){
        System.out.println("A");
        methodB();
    }

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

Reentrant Synchronization

A thread cannot acquire a lock owned by another thread.
But a thread *can* acquire a lock that it already owns.

Result:

A
B

Yes

Synchronized methods

Static synchronized method …

*"..A **static synchronized method is associated with a class, not an object**. The thread acquires the intrinsic lock for the Class object associated with the class ..".*

```
package CO3090.basic.control;

public class StaticSyncMethod {

    public static synchronized void methodA(){}
    public static synchronized void methodB(){}
    public synchronized void methodC(){}

}
```

Question:
Can you call the these methods at the same time from different threads?

(A) *StaticSyncMethod.methodA()*
StaticSyncMethod.methodA()

(B) *StaticSyncMethod.methodA()*
StaticSyncMethod.methodB()

(C) *StaticSyncMethod.methodA()*
obj1.methodC()

(D) *obj1.methodA()*
obj1.methodC()

Bad practice
should use call methodA() from class

Mutual exclusion (mutex)

Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*. To ensure mutual exclusion, the mechanism offered by Java are **synchronized methods** and **synchronized statements.**

Method (1)

Synchronized methods

public void synchronized method(){ … }

"…When you call any *synchronized method*, **that object** is **locked** and no other synchronized method of **that object** can be called until the first one finishes and releases the lock. "

Method (2)

Synchronized statements (or synchronized blocks)

Synchronized(object){….. }

"… Another way to create synchronized code is with *synchronized statements*. Unlike synchronized methods, synchronized statements must **specify the objec**t that provides the intrinsic lock…"

Synchronized statements (Synchronized block)

Usage:

```
synchronized(object){
    .....
}
```

Synchronisation is based on **explicit** locks.

A synchronized method is synchronized on the object provided

Only code between {} is synchronized .

A synchronized statement must acquires a mutual-exclusion object on behalf of the executing thread before executing the statements enclosed between { }.

* we should not choose a non-final field or String as lock object. (Why?)

Synchronized statements

```
synchronized(object){
    .....
}
```

Choosing lock object

Should use	***Avoid***
final field	non-final field String object

Why?

```
Object obj=new Object()

Synchronized(obj){
//something
}

Obj=new Object();
....
Synchronized(obj){
//something
}
```

```
//not recommended, should not synchronized on String
public final String lock2="Lock";
//not recommended, should not synchronized on non-final field
public Object lock3=new Object();
//OK
public final Object lock=new Object();
```

Synchronized statements

```
synchronized(object){
    .....
}
```

Choosing lock object

Should use	***Avoid***
final field	non-final field String object

Why?

```
private final String lock
 = new String("ABC");
        synchronized(lock){
        //something
       }
```

```
//not recommended, should not synchronized on String
public final String lock2="Lock";
//not recommended, should not synchronized on non-final field
public Object lock3=new Object();
//OK
public final Object lock=new Object();
```

Synchronized statements

Example 1: Lab exercise 2 – make sure people aren't interrupting others.

No synchronization:

```
public void run() {

      for (int i = 1; i <=3; i++) {
         Thread.yield();
         System.out.println("Person"+pid+" said "+ i);
      }

}
```

Sample output:

```
Person(0) said 1
Person(2) said 1
Person(0) said 2
Person(2) said 2
Person(0) said 3
Person(2) said 3
Person(1) said 1
Person(1) said 2
Person(1) said 3
```

Using synchronized block:

```
public static final Object microphone=new Object();
.............
public void run() {
  synchronized(microphone){
     for (int i = 1; i <=3; i++) {
        Thread.yield();
        System.out.println("Person"+pid+" said "+ i);
     }
  }
}
```

Sample output:

```
Person(0) said 1
Person(0) said 2
Person(0) said 3
Person(2) said 1
Person(2) said 2
Person(2) said 3
Person(1) said 1
Person(1) said 2
Person(1) said 3
```

* Result might vary

Synchronized statements

Example 2:

In (A) , all methods are synchronized but can two threads access the same instance of class SyncCounterDemo performing x.addC1() and x.addc2() at the same time? How to improve it?

```
package CO3090.basic.syncthis;

public class SyncCounterDemo {

    final Counter c1=new Counter();
    final Counter c2=new Counter();

    public synchronized void addC1(){
        c1.increase();
    }

    public synchronized void addC2(){
        c2.increase();
    }

}
```

(A)

Synchronized statements

Example 2:

In (A) , all methods are synchronized but can two threads access the same instance of class SyncCounterDemo performing x.addC1() and x.addc2() at the same time? How to improve it?

```
package CO3090.basic.syncthis;

public class SyncCounterDemo {

    final Counter c1=new Counter();
    final Counter c2=new Counter();

    public synchronized void addC1(){
        c1.increase();
    }

    public synchronized void addC2(){
        c2.increase();
    }

}
```

(A)

```
public void addC1(){
    synchronized(c1){
        c1.increase();
    }
}

public  void addC2(){
    synchronized(c2){
     c2.increase();
    }

}
```

(B)

Synchronized statements vs synchronized methods

Synchronized method

```
synchronized….  methodName(){
   //body
}
```

The following code does exactly the same thing

```
public  methodName(){
  synchronized(this){
   //body
  }
}
```

Synchronized method locks on the current object

"this"

Synchronized statements

Example 3:

```
public synchronized String getName()

public  void  setName(String  name) {
    synchronized(this) {
        this.name = name;
        counter++;
    }
    listOfNames.add(name);
}
```

Threads are NOT able to execute them concurrently

setName must synchronise changes to this.name and counter

- the ellipsis stands for some code that does not touch the fields shared by other threads (i.e., this.name and counter)

notice that the invocation to methods of other objects is outside the critical section (synchronized code invoking other objects' methods can easily lead to deadlocks or starvation)

Mutual exclusion in JAVA

The mechanisms that are offered by JAVA are

synchronized methods and **synchronized statements**

- Both of them can prevent thread interference and memory consistency errors
  Synchronisation based on (implicit) locks
- The synchronized modifier can be used in method declarations or for determining critical sections.
- A method declared synchronized cannot be executed at the same time by more than one thread
- If more than 2 threads try to invoke a synchronised method, only one of them actually access the object, while the other is blocked
- synchronized(obj){stm}: acquires the lock on obj, executes stm and releases the lock;stm is the critical section on the shared resource obj

Uses of synchronized

The following equation holds in JAVA:

```
synchronized ... methodName(...) { body; } = ... methodName(...){synchronized(this) { body; }}
```

Remark 6 Notice that this implies that synchronized is not part of the method signature

- As said, a thread cannot acquire a lock owned by another thread
- However, a thread can acquire a lock (i.e., acquire a resource) that it already
- owns This enables the so-called reentrant (or recursive) synchronisation, namely, when synchronised code invokes a method containing synchronised code that locks a resource already locked by the invoker.
- Notice that, without reentrant synchronisation, the invoker would have caused its own deadlock!

Telling a thread to wait….

Put the current thread to sleep for the specified time

Thread.sleep(milliseconds)

*By calling sleep(milliseconds), the current thread will not be run for the **specified time.***

What if we want to put one thread to sleep until a condition is resolved in another thread?

obj.wait() *"wait() causes the current thread to wait **until***
obj.notify() *another thread invokes the notify() method or*
obj.notifyAll() *the notifyAll() method for this object."*

* These methods are non-static method s of java.lang.Object

http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

Telling a thread to wait….

Useful methods:

- public final void wait() throws InterruptedException
  - The thread is suspended and it is put on the object waiting list
- public final void wait(long timeout) throws InterruptedException
  - The thread is suspended until another wakes it up or until the time is
- elapsed

  public final void notify()
  - Chooses and wakes up a single thread among those waiting on the object
  - monitor Which thread is chosen depends on the implementation of the JVM
  - This method should only be called by the “owner thread”, namely the thread that has acquired the lock on the object
  - Throws: IllegalMonitorStateException when invoked by a non-owner thread
- public final void notifyAll()
  - Like notify, but awakes all the waiting threads

Wait(), notify() and notifyAll()

Example: suspend the current thread using wait()
A wait can be "woken up" by another thread calling notify

```
package CO3090.basic.wait;

public class MainProgram {

    public static final Object lock=new Object();

    public static void main(String args[]) {

        synchronized(lock){
            try {
                lock.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println("End");
    }
}
```

Java object used as “monitor”

The *wait()*, *notify()*, and *notifyAll()* methods should be called for an object only when the current thread has already locked the object's **lock**

IllegalMonitorStateException ill occur if we don’t call them from synchronized context

Wait indefinitely until notification from another thread.

Wait(), notify() and notifyAll()

Example: suspend the current thread (timeout: 5 seconds)
A wait can be "woken up" by another thread calling notify

```
package CO3090.basic.wait;

public class MainProgram {

    public static final Object lock=new Object();

    public static void main(String args[]) {

        synchronized(lock){
            try {
                lock.wait(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println("End");
    }
}
```

The program will wake up if:

* It receives a notification from another thread.

Or

* Timeout expires (5 seconds)

Wait(), notify() and notifyAll()

Example: wakes up main thread when counter reaches 10

```
package CO3090.basic.wait;

public class MainProgram {

    public static final Object lock=new Object();

    public static void main(String args[]) {

        Counter counter= new Counter();
        counter.start();

        synchronized(lock){
            try {
                lock.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println("Main thread ends");
    }
}
```

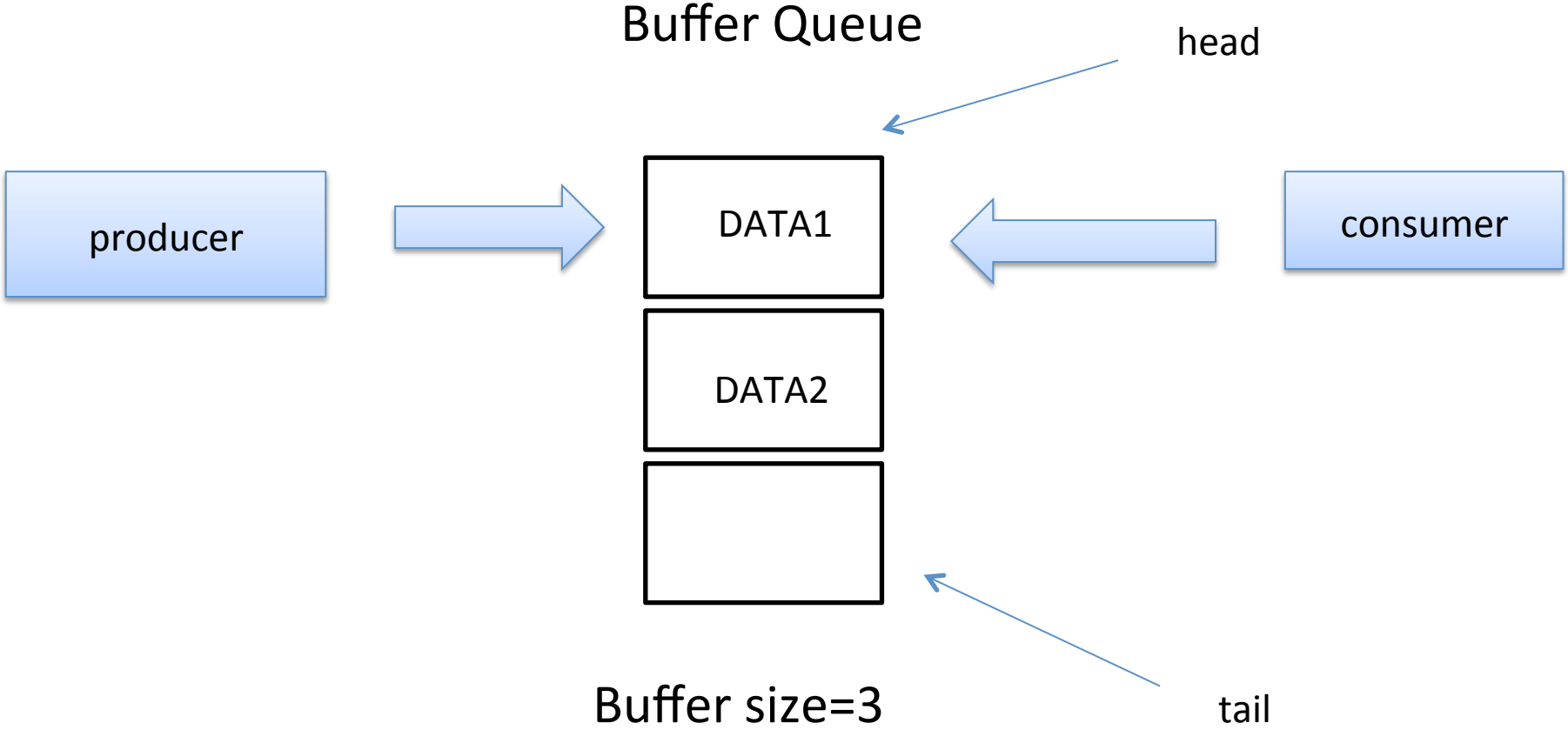
When counter reaches 10, wakes up the main thread

```
package CO3090.basic.wait;

public class Counter extends Thread{
    int visits=0;
    public synchronized void auto_increment(){
    while(true){
        int temp=visits;
            try {
                Thread.sleep(1000);
                System.out.println
                    ("counter="+getCounterValue());
            if(getCounterValue()==10){
                synchronized(MainProgram.lock){
                    MainProgram.lock.notifyAll();
                }
            }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            visits=temp+1;
        }
    }
    public synchronized int getCounterValue()
       {return visits;}
    public void run() {
        auto_increment();
    }
}
```

Producer & Consumer problem (bounded-buffer problem)

How to make sure

* The producer won't try to add data into a buffer if it is full.
* The consumer won't try to get data from an empty buffer.



Producer & Consumer problem (bounded-buffer problem)

a consumer invokes

```
public synchronized consume() {
      while(queue.isEmpty()) {
            try { wait(); } catch (InterruptedException e) {}
      }
      // Assume that getElement() notifies to producers!
      el = queue.getElement();
      ...
}
```

"Guarded Block"

a producer invokes

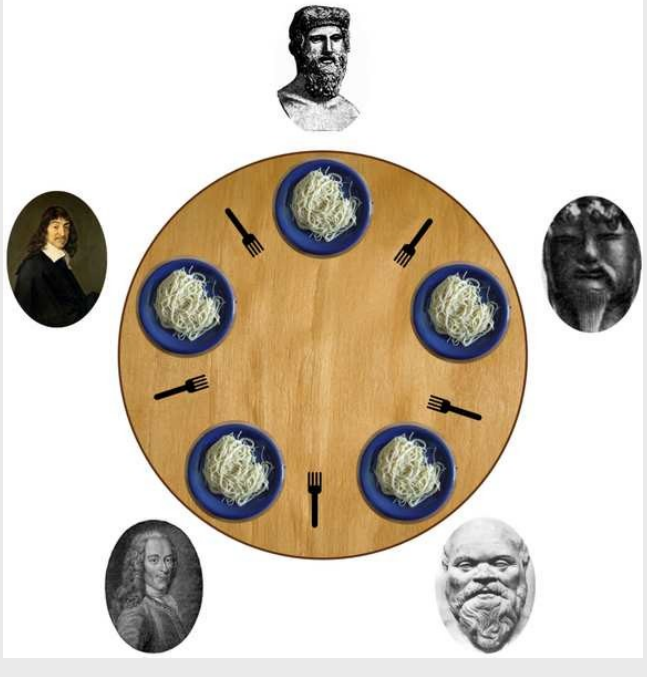
```
public synchronized produce() {
 while(queue.isFull()) {
       try { wait(); } catch (InterruptedException e) {}
 }
 el = new Element();
 // computes the info to insert in the queue
 ...
 queue.addElement(el);
 notifyAll();
 }
```

"Guarded Block"

Deadlock

"The life of a philosopher consists of an alternation of thinking and eating:

```
cycle begin
    think;
    eat
end
```



"Five philosophers sit around a circular table. Each philosopher spends his life alternatively thinking and eating. There is one plate of noodles in front of each philosopher. A philosopher needs two chopsticks to eat a helping of noodles. no two neighbours may be eating simultaneously."

"

[Dij71]

Dining philosophers (2)

There are some underling assumptions to consider for the dining philosopher problem:

- each philosopher cannot "see" what the others are doing: he can only realise whether he can pick up the left or right chopsticks
- philosophers are very polite: they behave according to the following:

  same abstraction level do not eat with their hands!

  no-preemption do not grab chopsticksalready hold by others

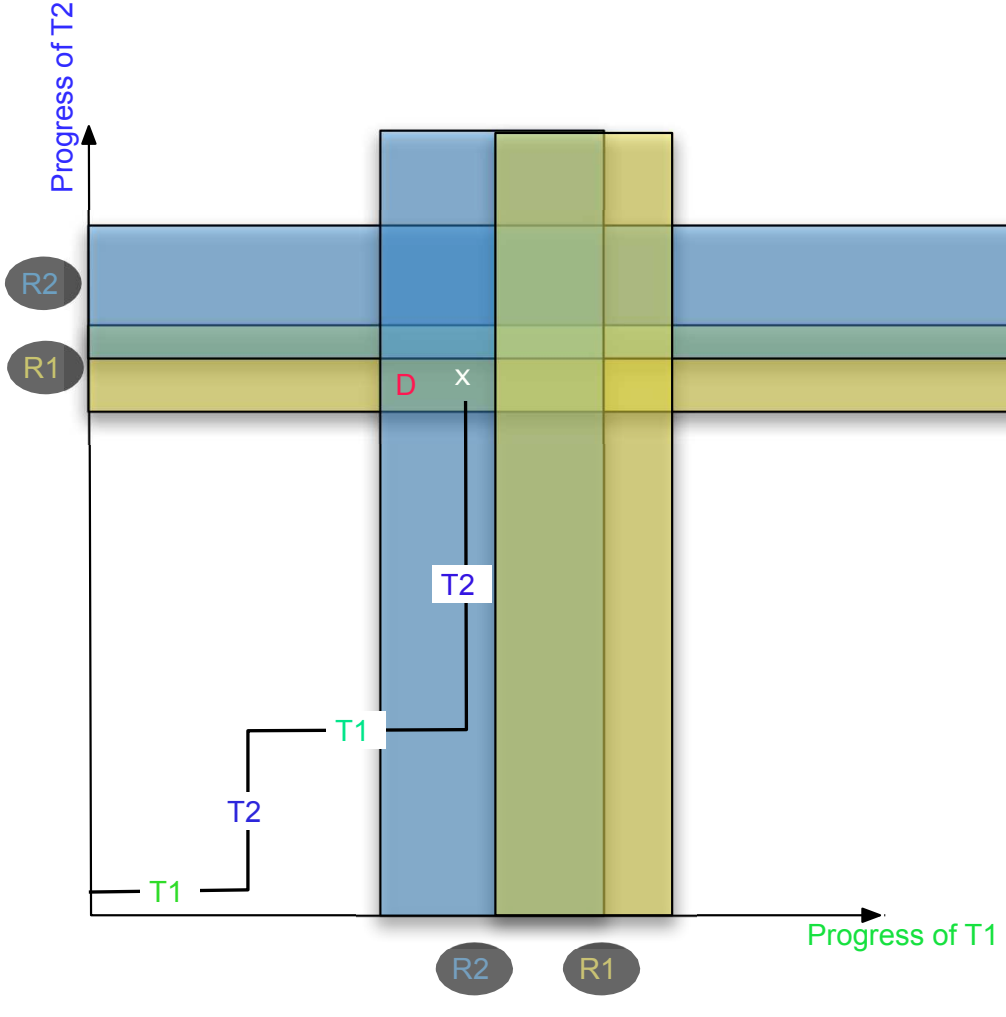
  respect the protocol use the same ceremonial (e.g., each philosopher always starts by picking up one chopsticks and then the other)

The dining philosopher problem shows two issues that can occur in concurrent (or distributed) applications:

- starvation: a process is perpetually denied the necessary resources for its computation so that it cannot finish its task
- deadlock: two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain.

Example 9 A db client requires exclusive access to a table (it locks the table) and then needs to obtain the lock on a second table. A deadlock occurs if another client locked first the second table and then tries to lock the first one.

Consider tasks T1 and T2. The combined progress of two tasks T1 and T2 has been represented by Dijkstra with a diagram as below where a point represent the number of instructions executed by T1 and T2. Tasks hold resources already allocated to them while waiting for additional resources ("wait for" condition).



- T1 and T2 require exclusive access to R1 and R2
- either T1 or T2 in execution (single CPU)
- trajectories are not under the control of T1 or T2 (it is the O.S. that decides who runs)

Fixing deadlocks

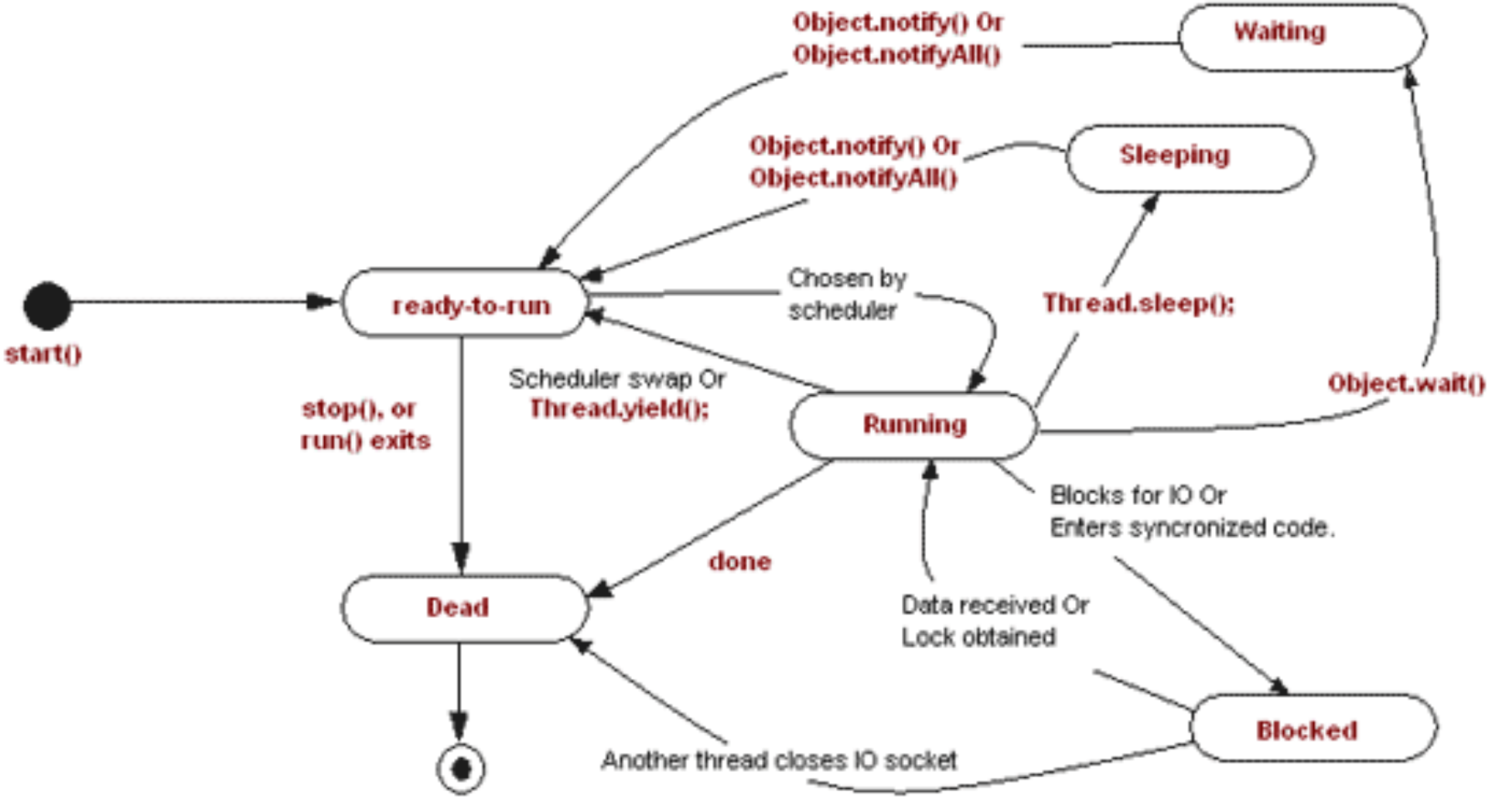
When deadlocks occur, the 4 Coffman conditions [CES71] hold

- Mutual exclusion: a shared resource is either available or owned by a thread
- Hold & Wait condition: a thread requests for resources, while holding some other resources
- No preemption condition: a resource cannot be preemptively taken away from a process (philosophers well-behave), namely, only the thread owning a resource can release it
- Circular Wait condition: A circular chain of threads one waiting for a resource owned by the next one must happen

Since these are necessary conditions for a deadlock to happen, it suffices to falsify one of them in order to prevent deadlocks.

Remark 5 In the JAVA solution for the Philosophers problem, the circular wait condition is broken.

Thread life cycle in Java



*borrowed from [REF]

Threads pools with the Executor Framework

ExecutorService

Threads pools

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

"Runnable object can be passed to a thread Pool, as soon as pool has any ideal threads then the runnable object will be executed "

execute(Runnable command)

Executes the given command at some time in the future.

shutdown()

Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.

isShutdown()

Returns true if this executor has been shut down.

awaitTermination(long timeout, WtimeUnit unit)

Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html

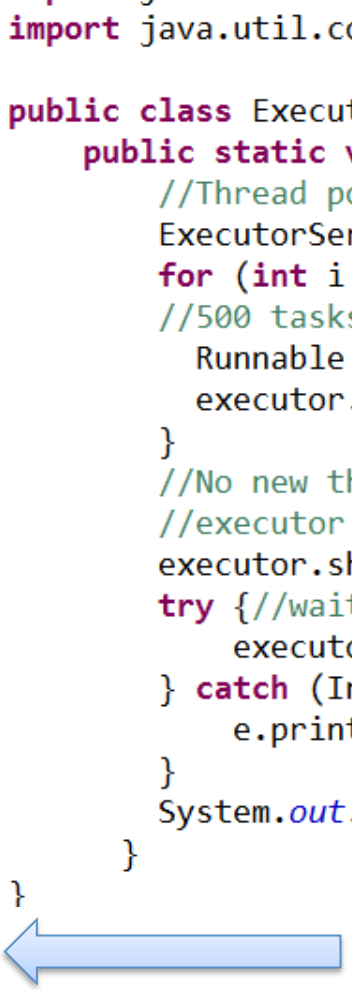
Threads pools with the Executor Framework

Limited the number of threads running at the same time.

```java
package CO3090.advanced;

public class PrintThread
         implements Runnable{

    String threadName;
    public String getThreadName() {
        return threadName;
    }
    public PrintThread(String name){
        this.threadName=name;
    }
    @Override
    public void run() {
        // TODO Auto-generated method stub
        System.out.println
    ("Hello from "+ this.getThreadName());
    }

}
```

```java
package CO3090.advanced;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class ExecutorServiceDemo {
    public static void main(String[] args) {
        //Thread pool size=10
        ExecutorService executor = Executors.newFixedThreadPool(10);
        for (int i = 0; i < 500; i++) {
        //500 tasks (runnableObject)
          Runnable task = new PrintThread("thread_t"+i);
          executor.execute(task);
        }
        //No new threads will be accepted
        //executor will finish all existing threads
        executor.shutdown();
        try {//wait until all threads are finish
            executor.awaitTermination(20000, TimeUnit.MILLISECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("All threads have terminted");
      }
}
```

Thread pool capacity



Surgery 2

Exercise 1

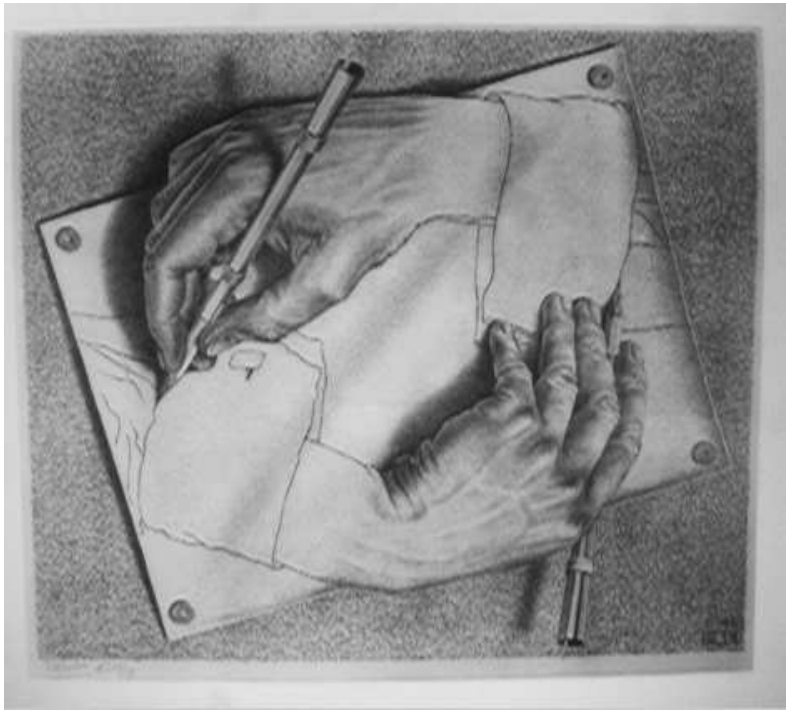
What is the problem with this program?

(1) Can you write a scenario which leads to possible deadlock situations?

(2) Modify the program to avoid potential deadlocks (Suppose we are not allowed to change ***transfer*** to a static synchronized method)

```java
package CO3090.deadlock;

public class BankAccount {

    double balance;

    public String getAccountName() {
        return accountName;
    }

    public void setAccountName(String accountName) {
        this.accountName = accountName;
    }

    String accountName;

    public BankAccount(String accountName, int init_amount){
        this.accountName=accountName;
        balance=init_amount;
    }

    public synchronized double getBalance() {
        return balance;
    }

    public synchronized void withdraw(double amount) {
        this.balance = this.balance-amount;
    }

    public synchronized void deposit(double amount) {
        this.balance = this.balance+amount;
    }


}
```

Concurrency (in JAVA)



Explicit Lock objects

Non-reentrant lock

```
package CO3090.advanced.lock;

import java.util.concurrent.Semaphore;

public class SemaphoreLock {
    Semaphore semaphore = new Semaphore(1);

     public void A() throws InterruptedException{
         semaphore.acquire();
         System.out.println("acquired the semaphore");
         System.out.println("A");
         B();
         semaphore.release();
         System.out.println("released the semaphore");
    }

     public void B() throws InterruptedException{
         semaphore.acquire();
         System.out.println("acquired the semaphore");
         System.out.println("B");
         semaphore.release();
         System.out.println("released the semaphore");
    }

     public static void main(String args[])
             throws InterruptedException{
         SemaphoreLock obj=new SemaphoreLock();
         obj.A();
     }
}
```

Non-reentrant lock

java.util.concurrent. ***Semaphore***

Non-reentrant lock dose not allow one thread to acquire the same lock more than once.

e.g. Invocation to method B() from A() is blocked

Output:

```
acquired the semaphore
A
```

blocked 

Explicit Lock objects

Reentrant lock

```
package CO3090.advanced.lock;

import java.util.concurrent.locks.*;

public class ReentrantLockDemo {

    Lock lock = new ReentrantLock();

     public void A() throws InterruptedException{
         lock.lock();
         System.out.println("A() acquired the lock");
         System.out.println("A");
            B();
         lock.lock();
         System.out.println("A() released the lock");
    }

     public void B() throws InterruptedException{
         lock.lock();
         System.out.println("B() acquired the lock");
         System.out.println("B");
         lock.unlock();
         System.out.println("B() released the lock");
    }

     public static void main(String args[])
             throws InterruptedException{
         ReentrantLockDemo obj=new ReentrantLockDemo();

         obj.A();
     }
}
```

Output:

```
A() acquired the lock
A
B() acquired the lock
B
B() released the lock
A() released the lock
```

Reentrant lock.

java.util.concurrent.locks. ***ReentrantLock***

Reentrant lock allows one thread to acquire the same lock more than once.

e.g. No problem to call B() from A()

Deadlock Prevention

Conditions for deadlock

Mutual exclusion

Hold-and-wait

No preemption

Circular wait

Prevention

Avoid mutual exclusion (could be very difficult…)

Avoid deadlock by acquiring all locks at once

“try lock”

Povide a total ordering on lock acquisition.
make sure locks are obtained in the same order

Deadlock Prevention

*assume ***a*** and ***b*** are 2 different locks

Hold-and-wait

```
Synchronized(a){
    Synchronized(b){
      //something
    }
}
```

Break **Hold-and-wait** condition by **acquiring all locks at once**

```
Synchronized(preventionLock)
   Synchronized(a){
    Synchronized(b){
      //something
     }
   }
}
```

Circular wait

Break **Circular wait** condition by **providing a total ordering on lock acquisition.** Making sure the locks are always obtained in the same order in different threads**.**

```
Synchronized(a){
    Synchronized(b){
      //something
    }
}
```

```
if(a.getID() >b.getID()){
            lockA=a;
          lockB=b;
   }
Synchronized(lockA){
    Synchronized(lockB){
      //something
    }
}
```

You own "***getID()***" method should generate an unique number for each lock objects.
e.g. UUID
or hashCode()*.

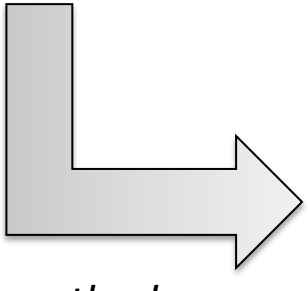
* In exceptional cases, two different object might still have the same hashCode

Deadlock Prevention

No preemption

Use "***tryLock***" method to check if the lock is free with the given waiting time

```
synchronized(lock){
   //critical section
}
```



*"**tryLock()** method acquires the lock if it is free within the given waiting time and the current thread has not been interrupted."*

```
public final Lock lock = new ReentrantLock();
....
//Acquires the lock if it is free within 10 seconds
 boolean acquired
      = lock.tryLock(10, TimeUnit.SECONDS);

 if(acquired){
 try{
       //locked successfully acquired
       //critical section
 } finally {
   lock.unlock();
}
 }else{
 //failure code
}
```

RPC & RMI

What an architecture is supposed to do?

> “The main purpose of defining an architecture is to try to impose order on chaos, or potential chaos” [BB04](chapt. 1, p. 5)

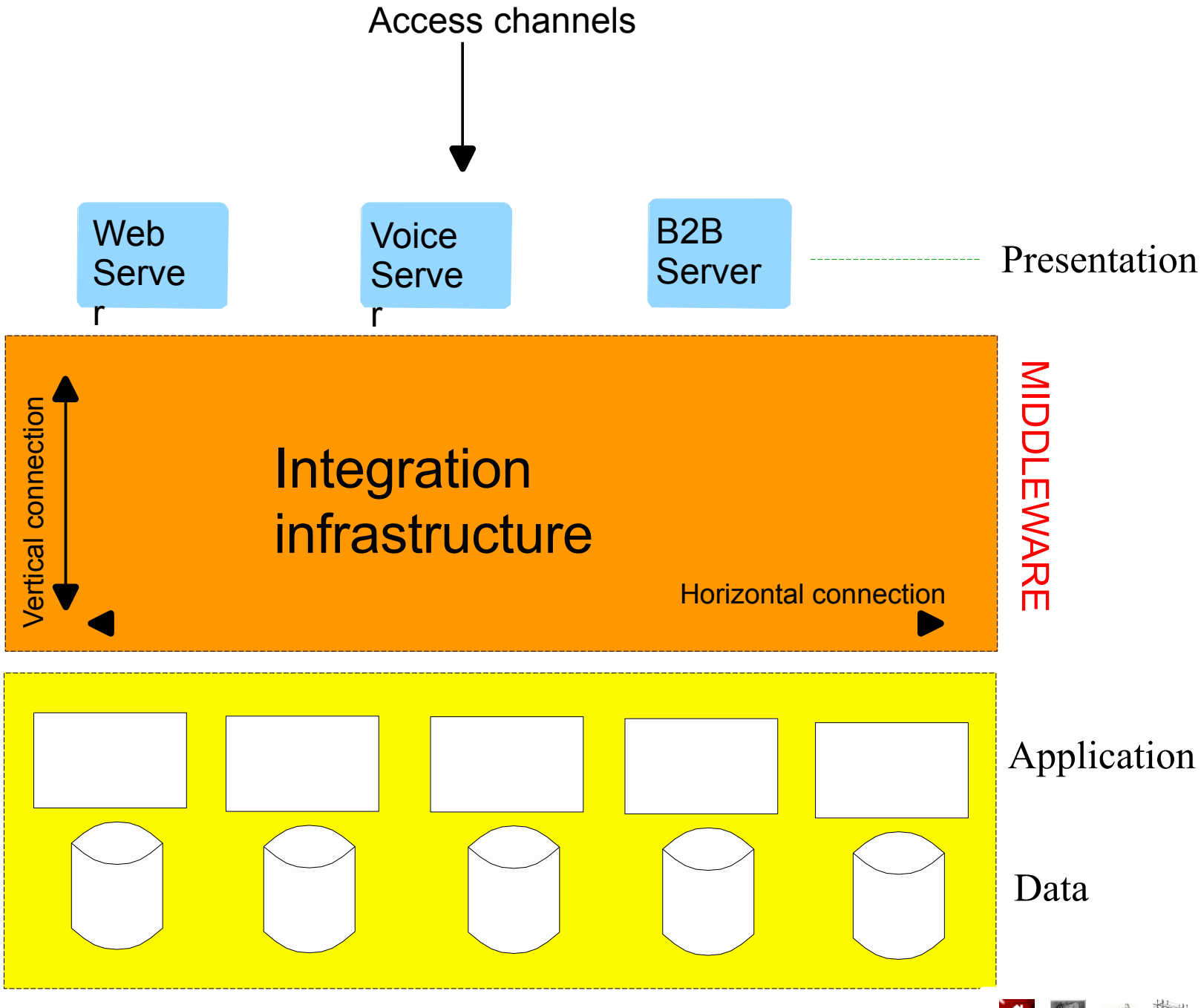
A well-designed architecture

- permits to solve current (and unanticipated) problems easily undertake evolving
- requirements
- expand the system in a systematic way facilitate reuse of code
- make teams work better
- improve maintainability
- ...
-

An example [BB04]

- A company wants to sell some products over the Internet and asks you to develop a website.
- If the initial architecture of the system is not “good” enough, soon
  - the system will become chaotic and hard to manage
  - usability will be poor
- Later (e.g., a few months after the web site has been deployed) the company decides to add a new functionality (e.g., use a Wireless Application Protocol (WAP) to get orders over mobile phones,...)
  - ♪ A bad architecture would make this extension
    - patchy (e.g., incomplete or inconsistent)
    - hard because it should very likely modify many parts of the existing
    - system hard to integrate with existing functionalities

A look to 3-tier architectures



Role of middlewares

In general, the role of middlewares in IT architectures is to ease programming by removing the burden of low-level on non application-specific details from the programmer.

- Middlewares for distributed applications typically
  - make it easier to program interactions among components of the business logic (horizontal connection)
  - make it easier to program interactions across different layers of the architecture (vertical connection)
  - allow to address distribution-related issues without modifying the logic of the application; for instance, J2EE distinguishes
    - administrative mechanisms
    - programmatic mechanisms
- The result is that programmers can focus on the applications and avoid bothering about low level or non application-specific details

Role of middlewares

In general, the role of middlewares in IT architectures is to ease programming by removing the burden of low-level on non application-specific details from the programmer.

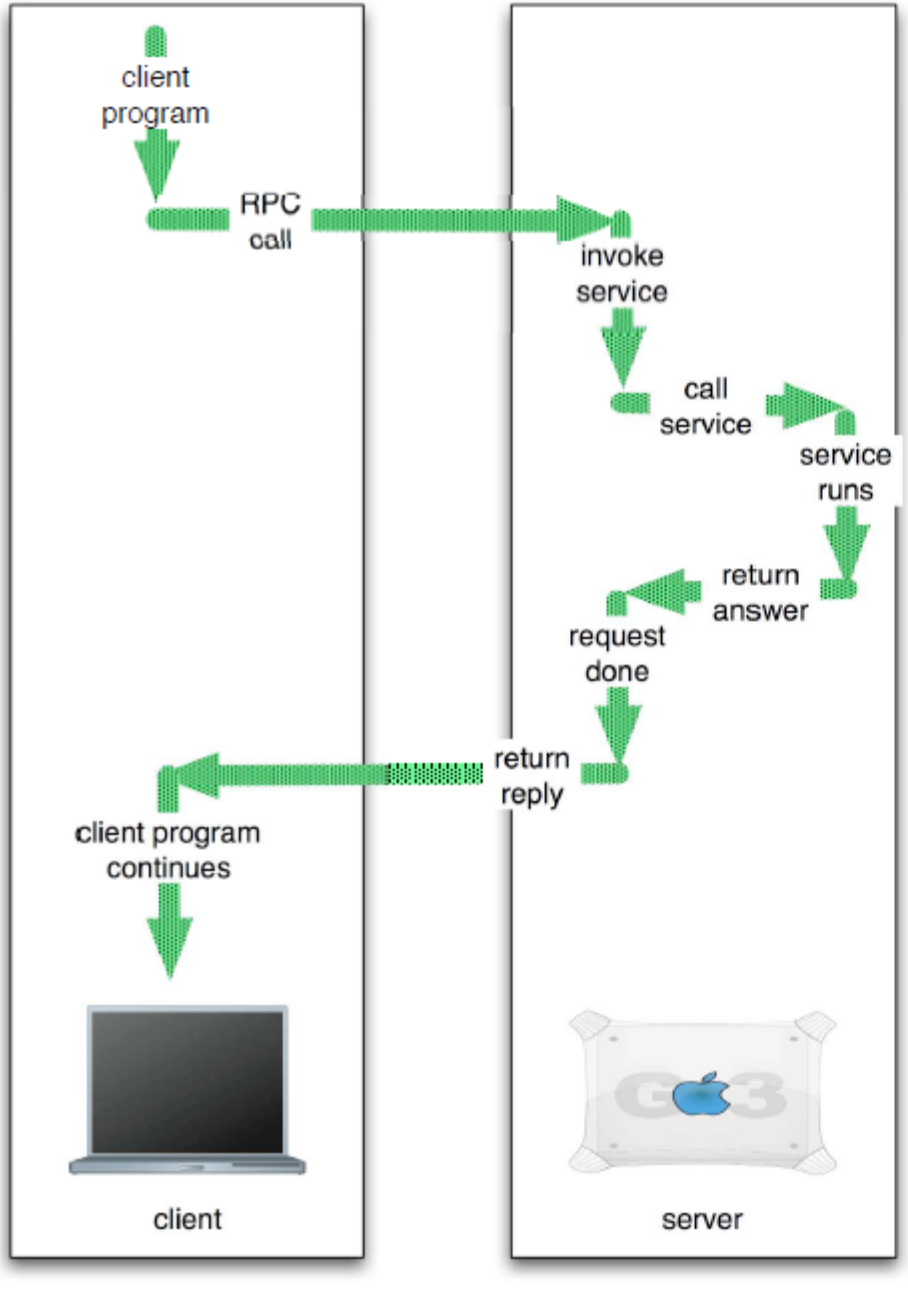
- Middlewares for distributed applications typically
  - make it easier to program interactions among components of the business logic (horizontal connection)
  - make it easier to program interactions across different layers of the architecture (vertical connection)
  - allow to address distribution-related issues without modifying the logic of the application; for instance, J2EE distinguishes
    - administrative mechanisms
    - programmatic mechanisms
- The result is that programmers can focus on the applications and avoid bothering about low level or non application-specific details

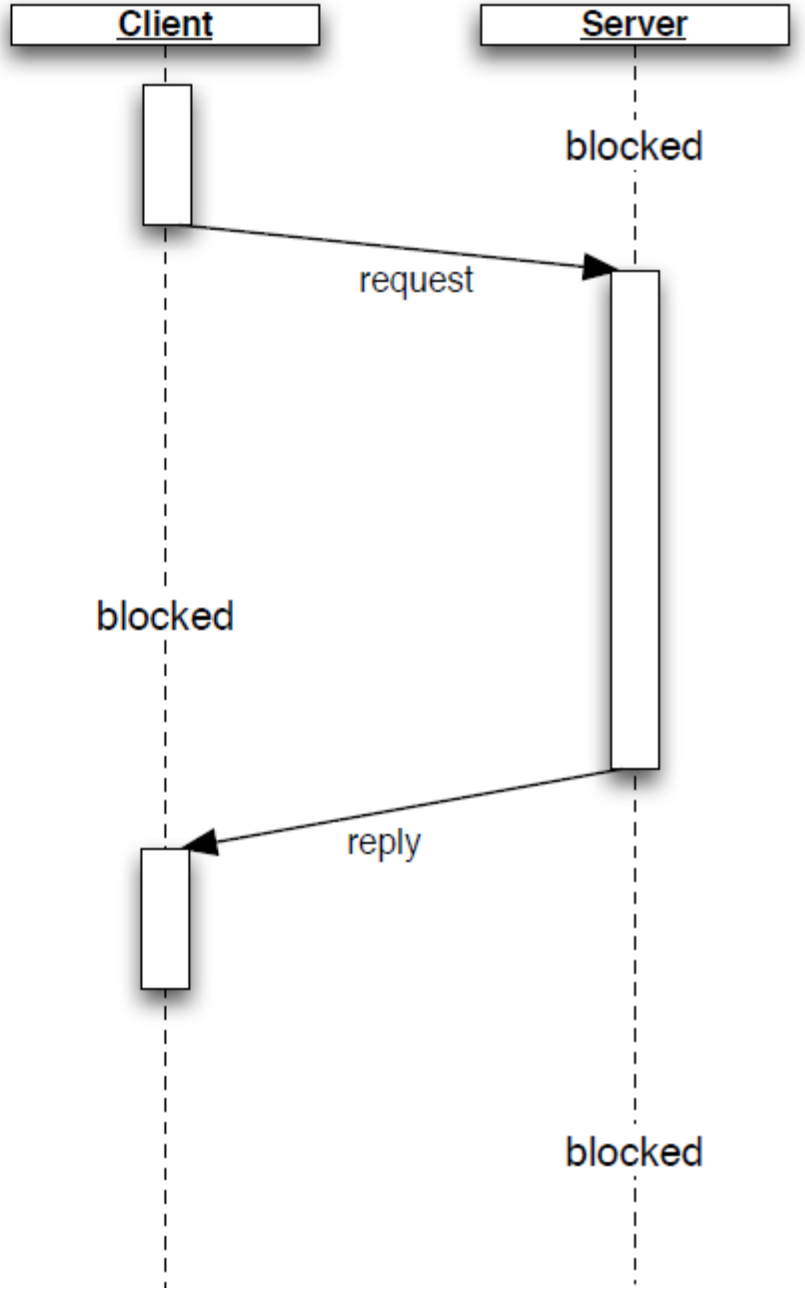
Popularity of RPC

RPC is very popular as an easy way of distributing applications. The main reasons are:

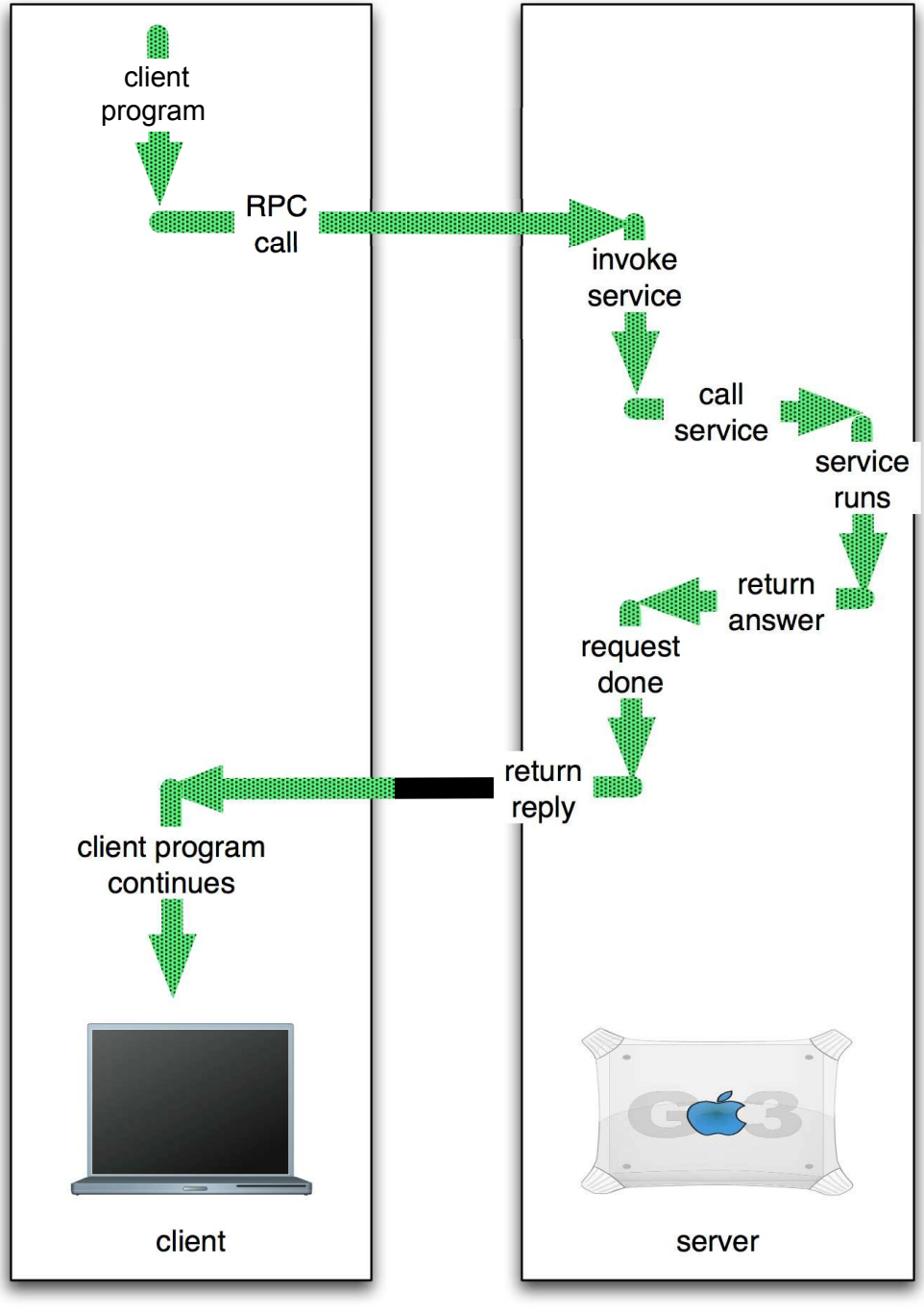
- The client-server paradigm is the most renowned among the paradigms for distributed programming as well-structured programs use procedure call.
- RPC allows one to distribute program without requiring many changes (provided that the client-server model is adopted)
- RPC is available in any language and at different levels of abstraction
- High-level RPC hides most of the underlying communication infrastructure
  - clients' remote calls send requests to a dispatcher routine on the server side
  - the dispatcher routine invokes the required local service and sends back the results
  - ...apart from minor issues, this is similar to what happens in local calls

Synchrony


client program
RPC call
invoke service
call service
service runs
return answer
request done
return reply
client program continues
client
server


Client
Server
blocked
request
blocked
reply
blocked

A few issues of distribution (1)



Remark 2 It is different to compile and link programs invoking local or remote procedures! For instance, the linker assumes that there is only one main program when the application is not distributed (also, there is a single address space for the main program and the invoked procedures).

Exercise 6 Figure out where vertical and horizontal connections are in the RPC middleware?

A few issues of distribution (2)

Consider the following programs (written in pseudo-code):

client side

```
program A {
...
   res1 =
   add(x,y);
...
   res2 =
   mul(u,v);
}
```

server side

```
...
public int add(int a, int b)
{
  return a + b;
}
public int mul(int a, int b)
{
  return a * b;
}
...
```

Client and server programs are written separately (may be from different programmers) and run in different execution environments!

- How the programmer of A knows “how to invoke” add and mul?
- What the invocations of add and mul will execute?
- Where add and mul are made “available”?

So far...

- IT architectures
- Basics of RPC

...now

The Ingredients of RPC

In order to tackle the problems of distribution in the compilation of RPC-based applications, the following mechanisms have been introduced:

- Stubs
  - client stubs
  - server stubs (skeleton)
- IDL
- Data marshalling/unmarshalling
- Naming service

We will now catch a glimpse of them and later on we will precisely look at their counterpart for JAVA RMI.

Principles of RPC Architecture: stubs

In order to make an RPC-based application executable, stubs are used.

- A client stub is a dummy procedure that
  yields the same signature of the corresponding function (e.g., add) provided by the server
  instead of the actual code of the function (e.g., sum of integers), the code of
  the stub handles the connection with the server stub in order to send the data and receive the results of invocations

- A server stub (or skeleton) is basically the main program of the server side!
  - it allows the creation of an executable entity consisting of the procedures specified on the server
  - dispatches the invocations received from clients to the actual procedures to be executed. This means that the server stub
    - receives, “translates” and passes the parameters of the invocations to the procedures
    - receives, “translates” and passes (if any) the result to the client (stub)

Exercise 7 Stubs are automatically generated.
Figure out which information is required
to generate them.

Principles of RPC Architecture: IDL

For the client and the server stub to communicate, an “agreement” must be reached. This is obtained by means of the Interface Description which is written in the Interface Description Language (IDL).

An interface description provides (together with other information necessary for generating the stubs):

uuid: unique universal identification number	[uuid (xxx–xxx–xxx–xxx) version y.y]
the name of the server interface	interface math_server
the signature of the published services	int add([in] int a, [in] int b [out] res),...

Remark 3 The input parameters ([in]) are passed by value while the output ones ([out]) by reference. Some IDLs allow declarations of input/output parameters ([in out]).

Principles of RPC Architecture: (un)marshalling

The activity of "translating" parameters and results, consists of two stages:

- Marshalling: (or in more modern terminology, serialisation) is the encoding of data in a standard form
- Unmarshalling: is, of course, the decoding of data from the chosen standard to the internal representation in the language(s) of the client and the server.

Remark 4 There exist many standards, for instance,

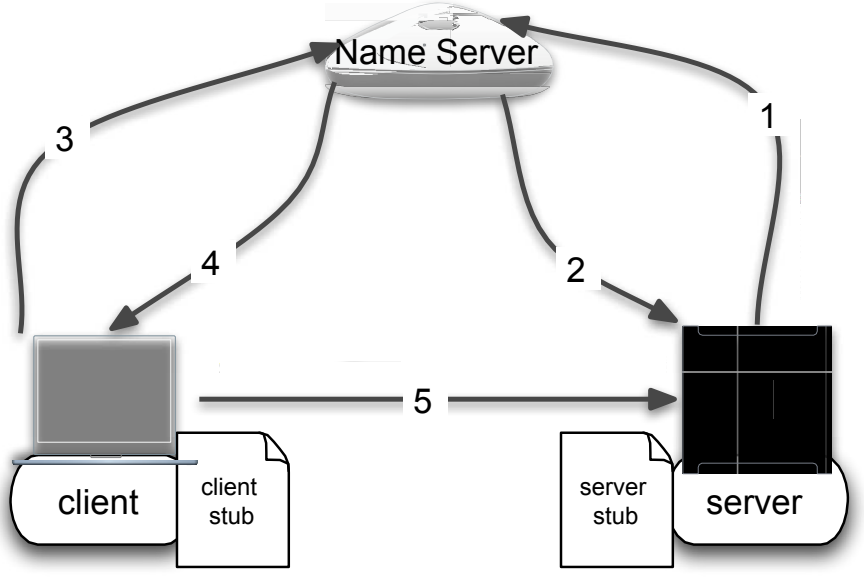
- Open Software Foundation DEC RPC adopts the Network Data Representation(NDR),
- Sun RPC uses External Data Representation (XDR).

It is not very important which is the actual format adopted, as long as client and server stubs are able to encode data in such common format.

Exercise 8 Why marshalling and unmarshalling are necessary/important?

Have a look to this link (6th bullet of §2.1).

Principles of RPC Architecture: Name server



1. the server stub registers with the name server and communicates the interface and physical address of the server
2. the name server confirms the registration; the server starts waiting for requests to serve
3. the client stub makes a request to the name server for a physical address of a server providing the name of the server and the number of the interface (uuid)
4. the name server returns the physical address of such a server (if any)
5. the client connects to the server

So far...

The principles of RPC middleware have been introduced

- client/ server stubs
- marshalling/ unmarshalling
- IDL
- Naming service

...now

Remote Method Invocation (RMI) is the OO correspondent of RPC. We see now how the principles and the concepts of RPC middleware are reflected in JAVA RMI

- JAVA RMI implements calls to methods of remote JAVA objects
- The main differences between RPC and RMI are:
  1. JAVA RMI allows objects written in JAVA to invoke each others methods, while RPC (in general) a communication middleware for programs written in different languages
  2. RPC can be seen as data-based, while
  3. in JAVA RMI you can communicate objects, namely data & behaviour!

Remark 6 Observe that JAVA objects

- can be passed as parameters in and returned
- as results of remote method calls

.

JAVA RMI has to deal with code mobility!

Distributed Objects: some terminology

The following terminology can be usually found in JAVA tutorials:

Distributed object an object whose methods can be remotely invoked. A distributed object is provided (or exported) by the object server.

Remote method a public method of a distributed object.

Object registry is the equivalent of the RPC name server. Namely, it is used by object servers to register their services and by object clients to look up for service references.

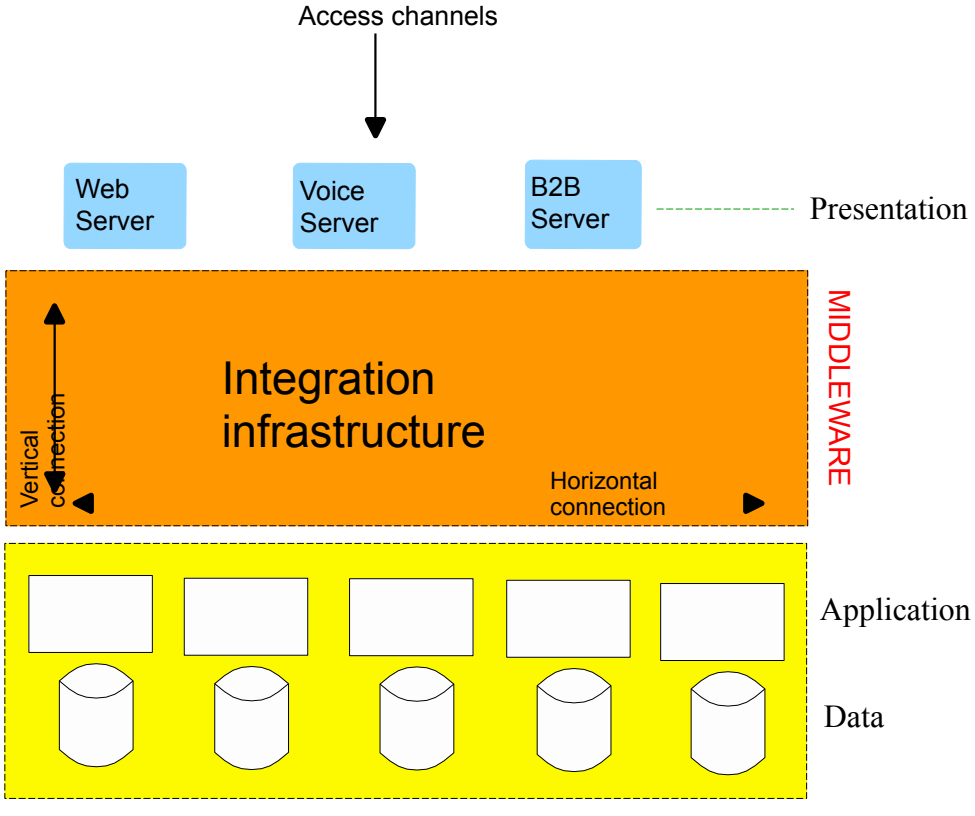
Client/server proxy is the equivalent of client/server stubs in RPC.

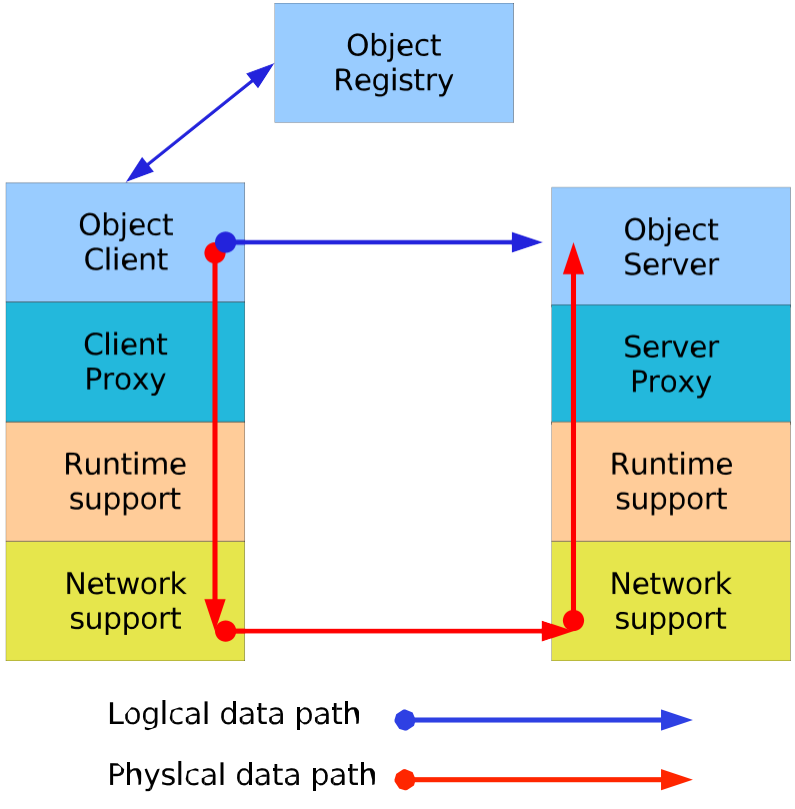
Invocations to remote methods "look" like normal calls to the programmer...but

- the client proxy (on the client host) interacts with the software providing runtime support for the distributed object system
- the runtime support transmits the actual call to the remote host (it also marshals the parameter to be transmitted)
- on the object server side, the runtime support for the distributed object system handles the incoming messages (and their unmarshalling), and forwards the call to the server proxy

RPC/RMI vertical and horizontal connections

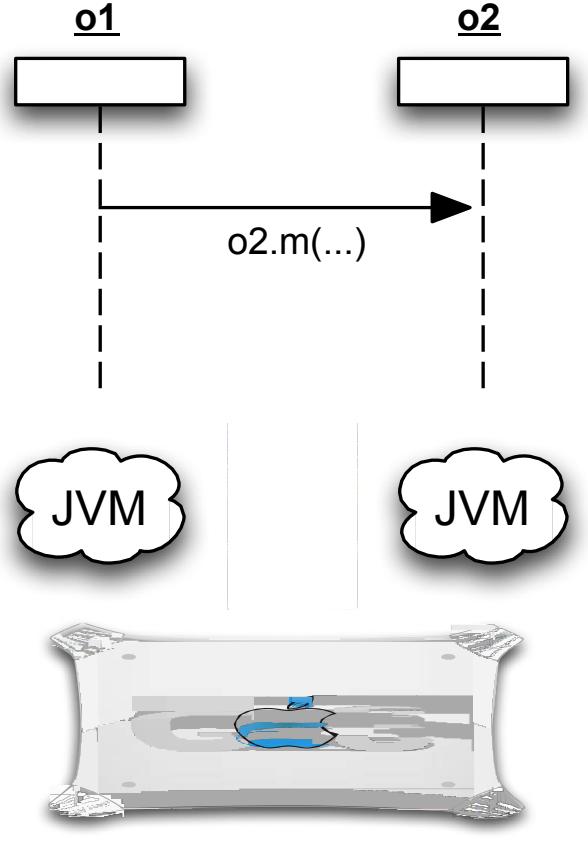
The RPC/RMI middleware implements vertical and horizontal connections as in the right-hand-side figure:



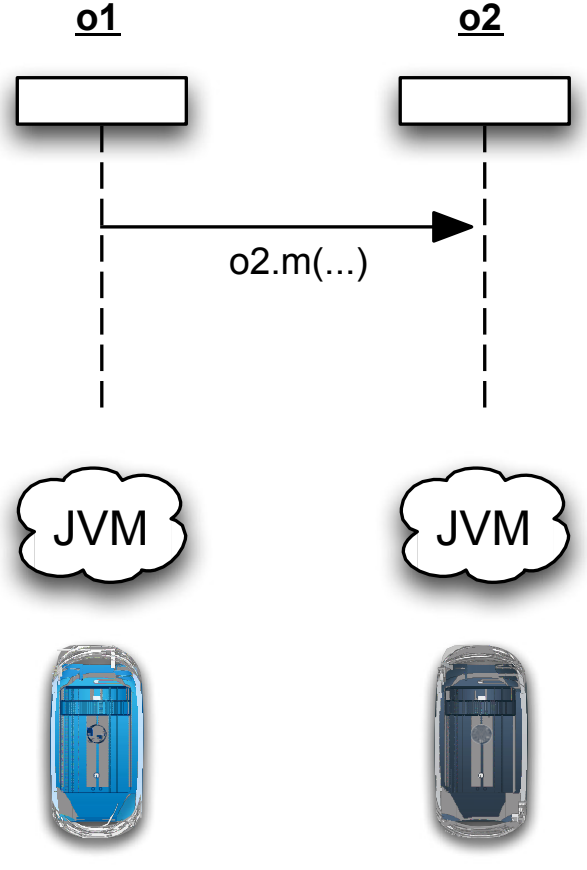


The logical data path from the client to the server corresponds the the physical data path from (the abstraction level of) the application to the network, then over the network and, finally, from the network to the invoked service.

Remark 7 JAVA RMI allows an object running in a JVM to invoke methods of other (JAVA) objects running in different JVMs...but where JVMs are running is immaterial!



Logical distribution



Physical distribution

In both cases we have a distributed application!

1. Design and implement RMI distributed application

First, give an initial architecture for your application (this might require some revision at a later stage) and determine which components are local objects and remote objects.

This phase consists of:

remote interfaces definition this determines the remote methods and their remote interfaces so that (local) objects that will be used as parameters or return values in remote invocations are also spotted.

remote objects implementation generally, remote objects have to implement several remote interfaces.

Remark 11 A remote object may implement other non-remote interfaces and define methods available only locally.

Any local classes used in remote method invocations (as parameters or return values) must be implemented as well.

clients implementation clients invoking remote objects can be implemented at any time after the definition of remote interfaces (or after deployment of remote objects).

In JAVA, remote objects are those implementing the java.rmi.Remote remote interface.

Remark 8 Basically, interfaces play the role of IDL; the IDL of JAVA RMI is java.rmi.Remote

- the object server
  - implements a remote interface
  - uses the remote interface for generating stub and skeleton
  - registers a distributed object implementing the interface
- an object client accesses the object server by invoking its remote methods according to the corresponding ID

Remark 9 Within RMI, remote objects are treated differently from non-remote objects. Fo instance, when a reference to a remote object r_obj should be passed in a remote invocation, what is actually sent is a stub for r_obj.
The stub acts as a local proxy for r_obj so that calls to r_obj are mediated by the stub.

.

Applications relying on distributed objects must:

1. Locate remote objects
   - by using the object registry or
   - by passing remote object references
2. Communicate with remote objects
3. Load bytecode of the classes of objects passed around

Since RMI allows a caller to pass objects within calls to remote methods, RMI yields the necessary mechanisms for loading an object's code and for transmitting its data.

Remark 10 One of the central features of RMI is the possibility of dynamically downloading the bytecode of the class of an object when it is not defined in the caller's JVM.

- Types and behaviours of objects can be transmitted to remote JVMs
- RMI guarantees that behaviour remains unchanged when objects are executed in another JVM
- new classes are introduced into a remote JVMs at run-time, so that applications can be dynamically extended.

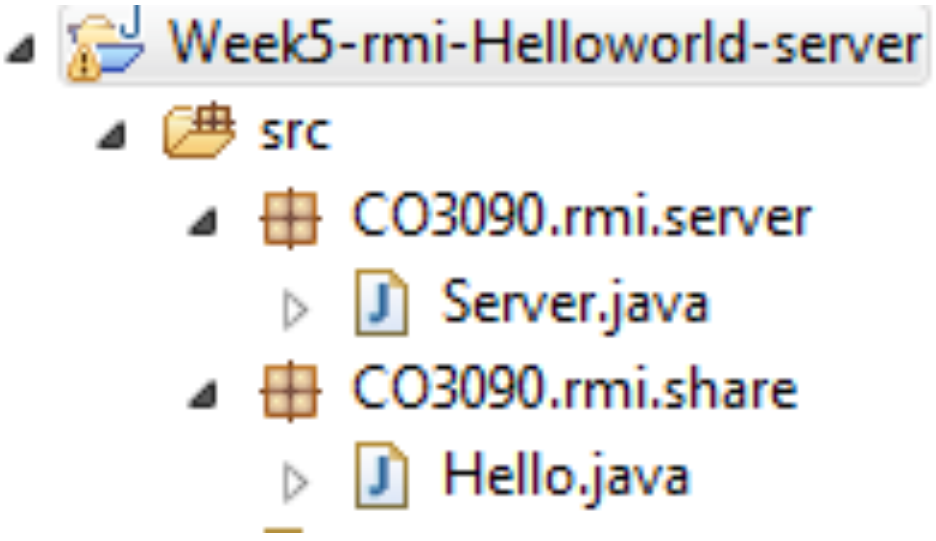
The development of distributed applications with RMI requires programmers to follow these general steps:

1. Design and implement the distributed application components
2. Compile sources and generate stubs
3. Make classes network accessible: in this step
   - the class files associated with remote interfaces
   - stubs
   - other classes needed by clients because of the remote invocations

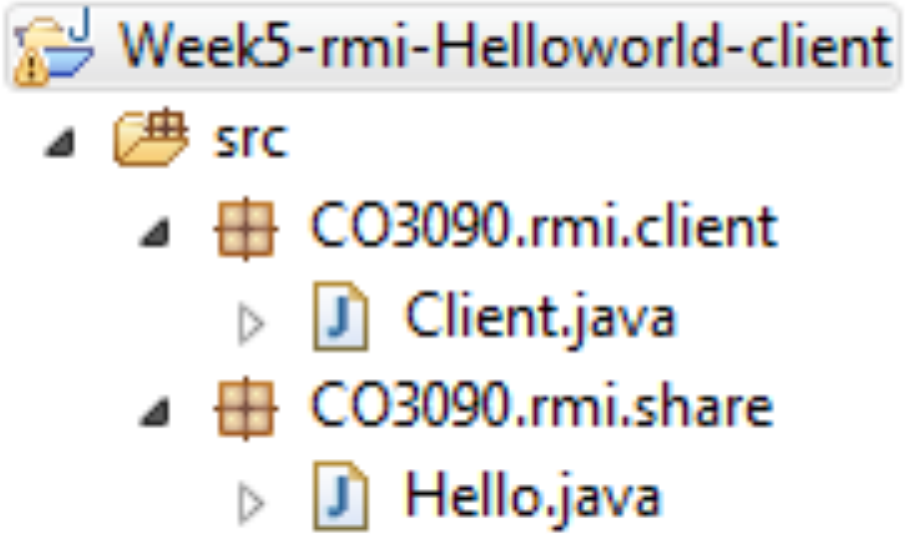
   are made accessible (e.g., via a Web server).
4. Start the application:
   (a) the RMI remote object registry
   (b) the server
   (c) the client

Exercise 10 Can the order of the steps to start the application be changed?

RMI Server



RMI Client



Step 1.1 - Remote interface

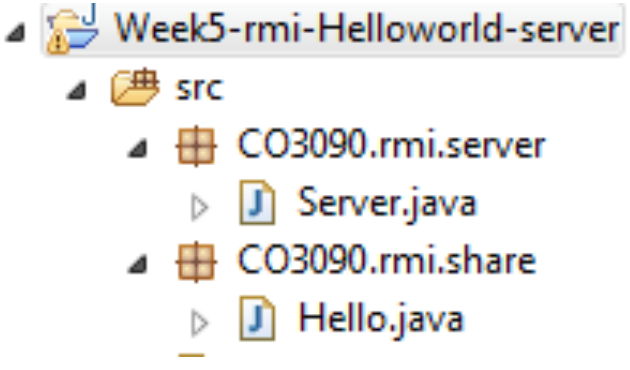
Declaring the Remote Interfaces Being Implemented

```
package CO3090.rmi.share;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

must extend java.rmi.Remote

must throw RemoteException()

Server



"RMI applications often comprise two separate programs, a server and a client.

A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects.

In JAVA, remote objects are those implementing the java.rmi.Remote remote interface.

JAVA remote interface

A remote interface is as any other JAVA interface...but

- each method signature must throw RemoteException.
- RemoteException exceptions are raised if errors occur when processing remote method call. The exception must be caught by the caller.

  Exercise 11 Explain why RemoteException exceptions cannot be handled by callees.

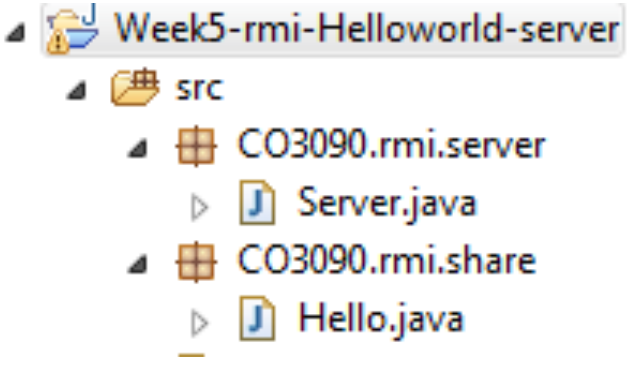
- RemoteException can be caused
  - by exceptions that may occur during communications (e.g., access or connection failures)
  - by problems in remote method invocations (e.g., errors resulting from object, stub, or skeleton not being found)

An example:

```
import java.rmi.*;

public interface ARemoteInterface extends Remote {
    String aRemoteMethod1( ... ) throws
    RemoteException;
    int aRemoteMethod2( ... ) throws
    RemoteException;
}
```

Step 1.2 - Remote object implementation Server

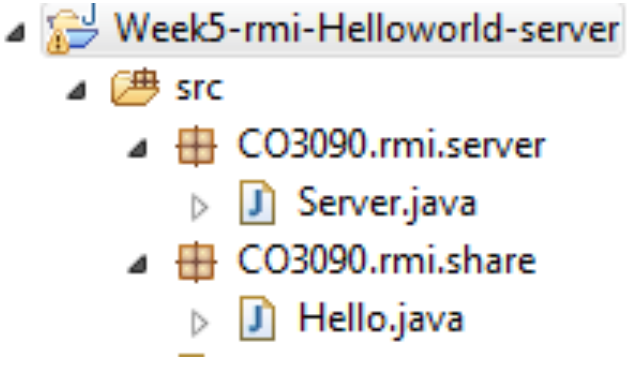
```
package CO3090.rmi.server;
import java.rmi.RMISecurityManager;

public class Server implements Hello {

    public String sayHello() throws RemoteException {
        return "Hello, world!";
    }
    public static void main(String args[]) {

        try {
            if (System.getSecurityManager() == null) {
                System.setSecurityManager(new RMISecurityManager());
            }
            String name = "rmi://localhost/Hello";
            Server obj = new Server();
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
            Registry registry = LocateRegistry.getRegistry();
            // Bind the remote object's stub in the registry
            registry.rebind(name, stub);

            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```



Providing Implementations for Each Remote Method

Step 1.2 - Remote object implementation Server

```
package CO3090.rmi.server;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

import CO3090.rmi.share.Hello;

public class Server implements Hello {

    public String sayHello() throws RemoteException {
        return "Hello, world!";
    }
    public static void main(String args[]) {

        try {
            if (System.getSecurityManager() == null) {
                System.setSecurityManager(new RMISecurityManager());
            }
            String name = "rmi://localhost/Hello";
            Server obj = new Server();
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
            Registry registry = LocateRegistry.getRegistry("localhost");
            // Bind the remote object's stub in the registry
            registry.rebind(name, stub);

            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```



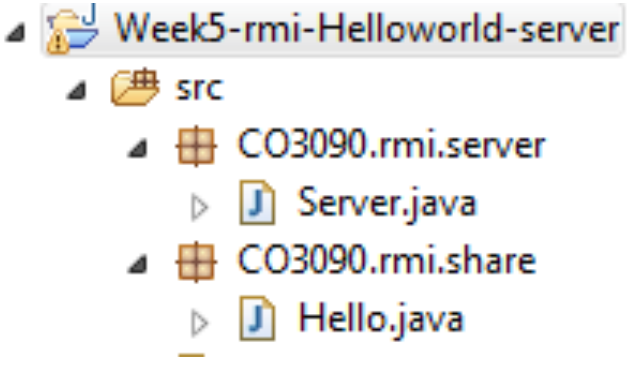
Create a Security Manager

Export a server stub(skeleton)

Get RMI registry on specified host (e.g. localhost)

Step 1.2 - Remote object implementation Server

```
package CO3090.rmi.server;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

import CO3090.rmi.share.Hello;

public class Server implements Hello {

    public String sayHello() throws RemoteException {
        return "Hello, world!";
    }
    public static void main(String args[]) {

        try {
            if (System.getSecurityManager() == null) {
                System.setSecurityManager(new RMISecurityManager());
            }
            String name = "rmi://localhost/Hello";
            Server obj = new Server();
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
            Registry registry = LocateRegistry.getRegistry("localhost");
            // Bind the remote object's stub in the registry
            registry.rebind(name, stub);

            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

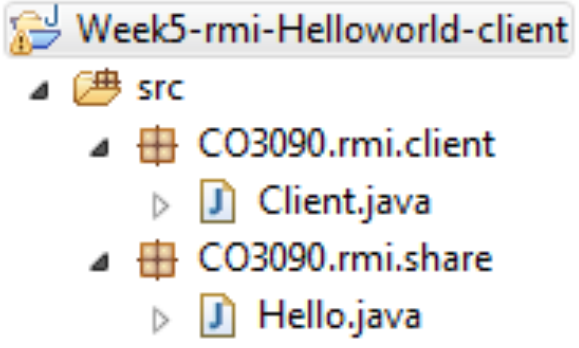


Make the Remote Object available to Clients

Step 2.1 - Client Implementation

Import the remote Interfaces
(Hello.java is the same as the interface defined on the server side)

Client



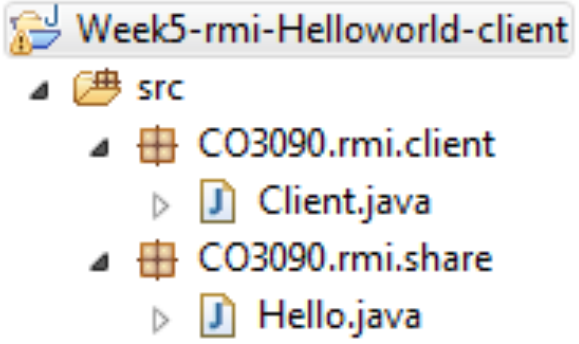
"A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them."

```
package CO3090.rmi.client;

import java.rmi.registry.LocateRegistry;

public class Client {

    private Client() {}

    public static void main(String[] args) {

        String rmiregistry_host="127.0.0.1";

      try {
            Registry registry = LocateRegistry.getRegistry(rmiregistry_host);
            Hello stub = (Hello) registry.lookup("rmi://localhost/Hello");
            String response = stub.sayHello();
            System.out.println("response: " + response);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Step 2.1 - Client Implementation

Import the remote Interfaces
(Hello.java is the same as the interface defined on the server side)

Client




```
package CO3090.rmi.client;

import java.rmi.registry.LocateRegistry;

public class Client {

    private Client() {}

    public static void main(String[] args) {

        String rmiregistry_host="127.0.0.1";

      try {
            Registry registry = LocateRegistry.getRegistry(rmiregistry_host);
            Hello stub = (Hello) registry.lookup("rmi://localhost/Hello");
            String response = stub.sayHello();
            System.out.println("response: " + response);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Obtain a registry on given host

Obtain reference **bound to "Hello"**

Invoke the remote method()

Step 3.1 - Compile source codes

Use javac to compile all server-side and client-side classes

(Note: If you use Eclipse, all binary classes are located inside \bin\ folder)

Step 3.2 -create client and server binary archive (JAR)

CD to “Week5-rmi-Helloworld-server” directory

jar cvf server.jar ./bin/CO3090/

CD to “Week5-rmi-Helloworld-client” directory

jar cvf client.jar ./bin/CO3090/

Step 3.3 -Make JARs accessible

Ideally JARs should be uploaded to a HTTP server. For example, since server and client are located on the same machine , we simply copy them to a shared folder (e.g. file:///C:/Export/)

Step 4 - Run rmi registry

Step 4.1 start rmiregistry

Win

```
start rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false
```

Linux/Mac

```
rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false
```

Step 4.2 grant security permissions

Create the a access policy file named `policy.permission`

```
grant {
    permission java.security.AllPermission;
};
```

Step 5 - Run server and client

Step 5.1 start server

```
java
-Djava.rmi.server.codebase="file:///C:/Export/server.jar"
-Djava.rmi.server.useCodebaseOnly=false
-Djava.security.policy="policy.permission"
CO3090.rmi.server.Server
```

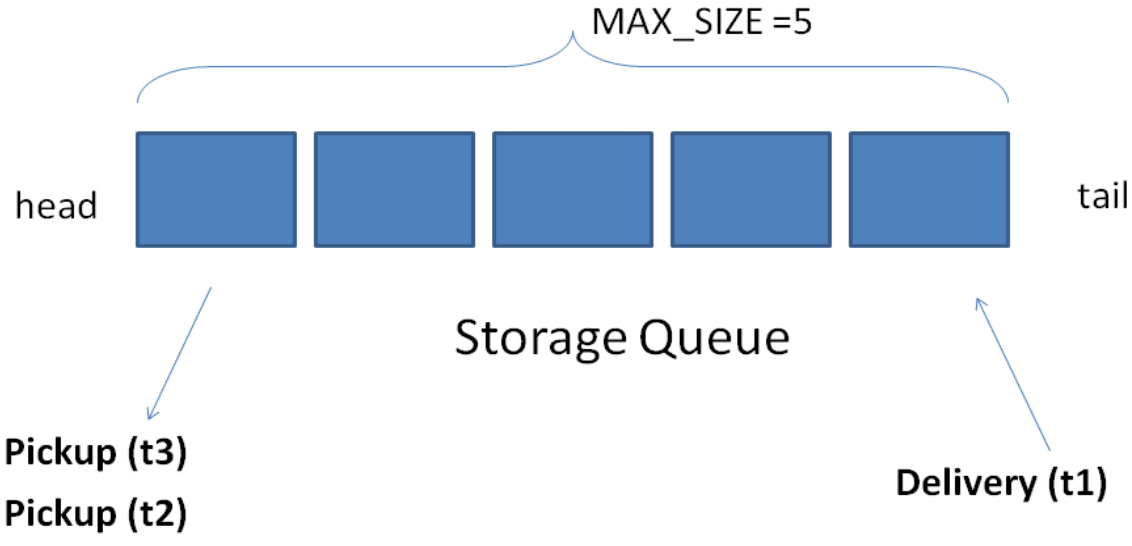
Step 5.2 start client

```
java
-Djava.rmi.server.codebase="file:///C:/Export/client.jar"
-Djava.rmi.server.useCodebaseOnly=false
-Djava.security.policy="policy.permission"
CO3090.rmi.client.Client
```

VM arguements

VM argument	
`Java.rmi.server.codebase` **Trouble shooting:** Incorrect codebase will usually cause **java.rmi.UnmarshalException** to be thrown	This property specifies the locations from which classes that are published by this VM (for example: stub classes, custom classes that implement the declared return type of a remote method call, or interfaces used by a proxy or stub class) may be downloaded. The value of this property is a string in URL format or a space-separated list of URLs that will be the codebase annotation for all classes loaded from the CLASSPATH of (and subsequently marshalled by) this VM.
`-Djava.security.policy` **Trouble shooting:** Double check this argument the following exception: occurred: j**ava.security.AccessControlException:** access denied	We must specify a policy file to grant access to the the directory. policy file content: `grant { permission java.security.AllPermission };`
`-Djava.rmi.server.useCodebaseOnly`	IMPROTANT: from JDK 7, The behavior of loading classes from locations specified by the remote end of the RMI connection, is disabled when the java.rmi.server.useCodebaseOnly is set to true**. We must set this property to false** so that server can load the classes from the location specified

Surgery 3



One Delivery threads (t1) and two Puckup thread (t2, t3) are sharing a fixed-size buffer (MAX_SIZE=5) . Delivery thread t1 generates a random number (1~10) every few seconds and put the number into StorageQueue while Pickup thread (t2, t3) can take a number from the Queue at a time.

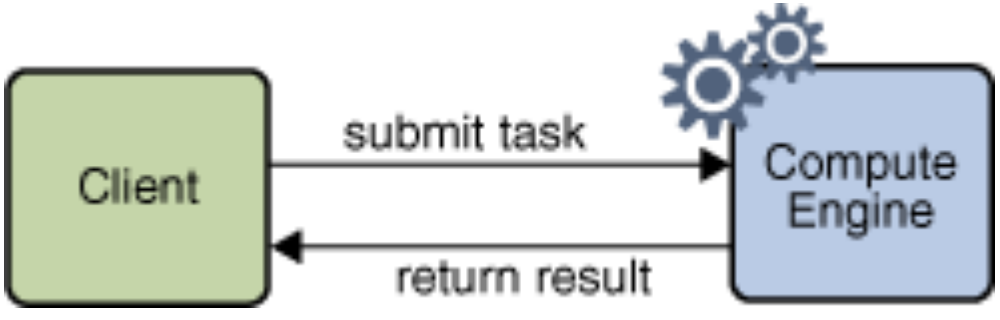
You tasks:

(1) Complete **put()** and **take()** methods in Storage.java, use wait() and notifyAll(), make sure that

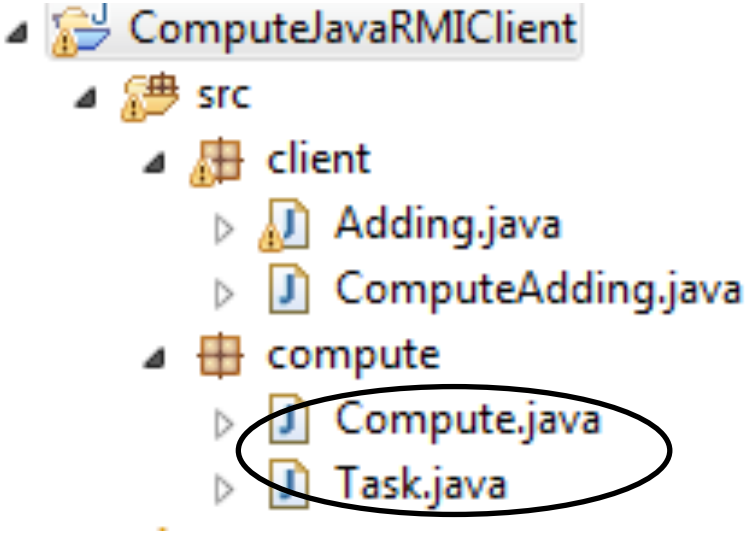
t1 do not try to put number into the queue when the queue is full.
t2, t3 do not try to remove data from an empty storage queue.

(2) Complete printMessage() in Main.java so that printMessage()should print “Great!” when any pickup thread (t2 or t3) receives an even number (2,4,6,8) from the queue.

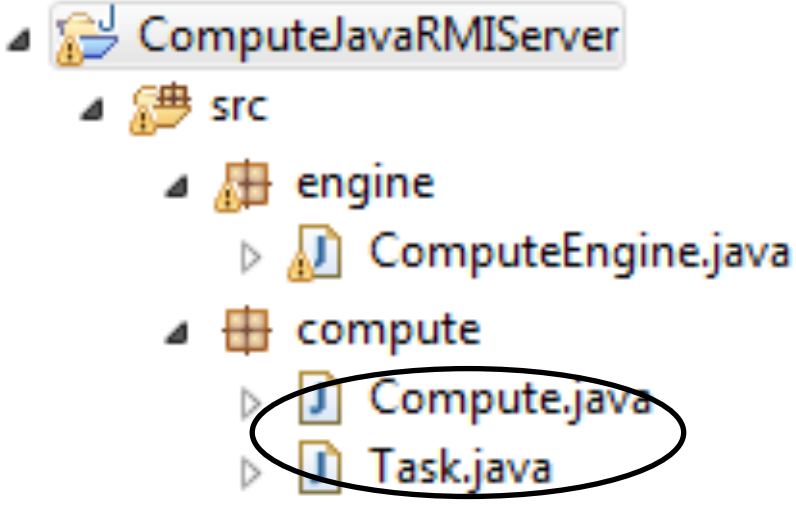
An example: the compute engine



Client



Server



An example: the compute engine

"At the heart of the compute engine is a protocol that allows jobs to be submitted to the compute engine, the compute engine to run those jobs, and the results of the job to be returned to the client. This protocol is expressed in interfaces supported by the compute engine and by the objects that are submitted to the compute engine[...]" [Sun]

```
import java.rmi.Remote;
import java.rmi.RemoteException;
/*
 * The 2 lines above can be replaced by
 * import java.rmi.*;
 */
public interface Compute extends Remote
{
    public Object executeTask(Task t)
        throws RemoteException;
}
```

The remotely accessible part is the compute engine itself, whose remote interface has a single method.
By extending java.rmi.Remote, Compute allows executeTask to be called from any JVM. Any object implementing Compute becomes a remote object.
Notice that executeTask

- takes a Task
- returns any Object
- throws RemoteException

An example: the compute engine (2)

An interface for Task objects must be defined.

```
import
java.io.Serializable;

 public  interface  Task  extends
                    Serializable  {


      public  Object  execute();


  }
```

Different kinds of tasks can be run by a Compute object provided that they implement Task. It is possible to add further methods (or data) needed for the computation of the task.

Exercise 12 execute is not required to throw RemoteException , why?

Remark 13 The Task interface extends the java.io.Serializable interface to let the RMI middleware serialise objects so that they can be transported from a JVM to another.

"Implementing Serializable marks the class as being capable of conversion into a self-describing byte stream that can be used to reconstruct an exact copy of the serialised object when the object is read back from the stream." [Sun]

Passing objects in RMI

Parameters or return values in remote method invocations can be of almost any type (e.g., local objects, remote objects, and primitive types):

- any entity of any type can be exchanged in remote method invocations provided that one of the following conditions applies
  - it is primitive data type or
  - it is a remote object
  - it is a serialisable object (i.e., it implements the interface java.io.Serializable)
- Some types do not fulfil any of the above criteria and, therefore, they cannot be passed in remote method invocations or returned as parameters.

  Remark 14 Basically, the objects of such types encapsulate information that has no sense in a remote JVM (for instance, a file descriptor)
- However, many of the core classes (e.g., those in java.lang and java.util) do implement Serializable and can be used when invoking remote methods.

How does JAVA RMI work?

Here is how RMI makes the simple compute engine possible

- RMI can assume that Task objects are written in JAVA
- Objects of a previously unknown Task type are downloaded by RMI into the Compute JVM as needed
- Clients do not need to install the code for Task on the server machine
- executeTask can return any object (the server does not need to know the class of the returned result)

```
package compute;

public interface Compute extends java.rmi.Remote {

    public Object executeTask(Task t) throws java.rmi.RemoteException;

}
```

```
package compute;
import java.io.Serializable;

public interface Task extends Serializable {

    public Object execute();

}
```

So far...

- Relations between RPC and JAVA RMI
- Steps for defining JAVA RMI applications
- Parameter passing in JAVA RMI
- Remote exceptions

http://docs.oracle.com/javase/tutorial/rmi/overview.html

For further reading

Main differences between local & remote objects (1)

	Local Object	Remote Object
Definition	by a JAVA class	by a Remote interface
Implementation	by its Java class	by a JAVA class that implements the remote interface
Creation	by new operator	new but executed on the server host: a client cannot directly create a new remote object
...	...	...

Main differences between local & remote objects(2)

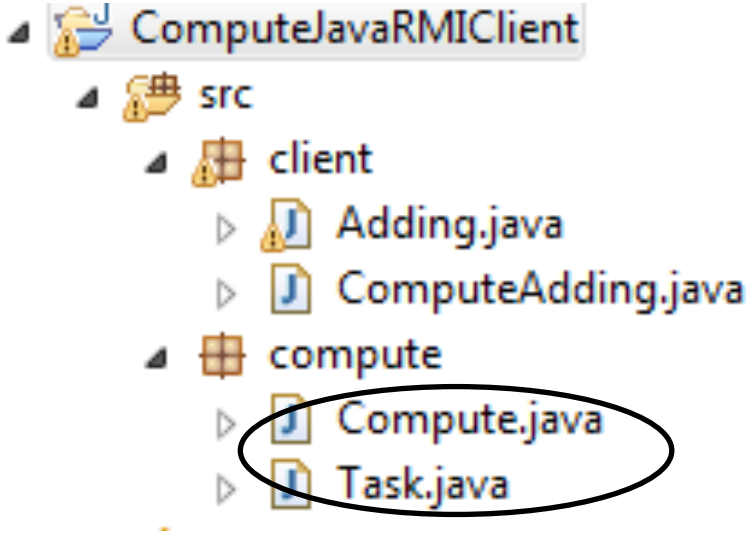
	Local Object	Remote Object
...	...	...
Access	directly via a reference	via a reference pointing to a stub that act as a proxy to an object that implements the remote interface
Passed by	(deep) copy	(remote) reference (to stubs)
Exceptions	Runtime exceptions or Exceptions	In order to ensure robustness of distributed applications, RMI forces to deal with any possible RemoteException

Implementing a remote interface

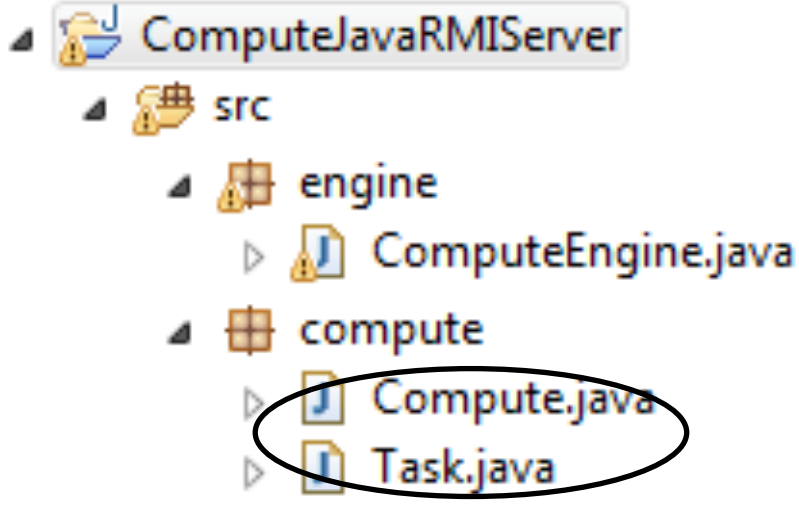
In order to implement a remote object it is necessary to:

1. declare the remote interfaces to be implemented
2. define constructors for the remote object
3. provide an implementation (for each remote method) of the remote interfaces
4. The server has to create and install the remote objects.

Client



Server



The object server class

The object server lies in a class that

- implements the remote interface (e.g., see ComputeEngine later on)
- requires to explicitly define constructors for the remote object because they must throw RemoteException.

  Remark 15 This is necessary even if the super-class constructor is "enough" (in which case the constructor's body is just super)
- can contain other methods, however only those in the remote interface are available to the client.

  Remark 16 Clients will not have a reference to (objects of) the class that implements a remote interface, but only references to interfaces
- extends UnicastRemoteObject...(see next slides)

```
public ComputeEngine() throws RemoteException {
    super();
}
```

.....

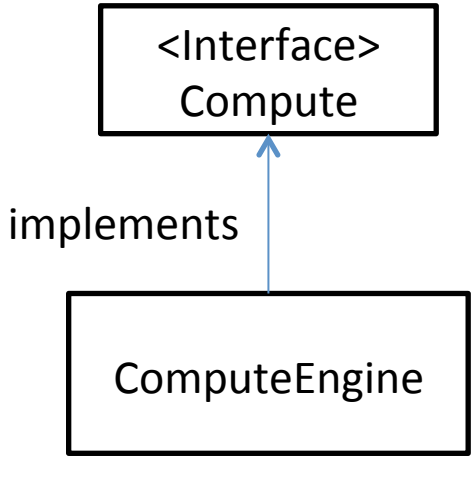
The object server class

The object server lies in a class that

- implements the remote interface (e.g., see ComputeEngine later on)
- requires to explicitly define constructors for the remote object because they must throw RemoteException.

  Remark 15 This is necessary even if the super-class constructor is "enough" (in which case the constructor's body is just super)
- can contain other methods, however only those in the remote interface are available to the client.

  Remark 16 Clients will not have a reference to (objects of) the class that implements a remote interface, but only references to interfaces
- extends UnicastRemoteObject...(see next slides)



Client:

ComputeEngine comp = (ComputeEngine) Naming.*lookup(name);*

Compute comp = (Compute) Naming.*lookup(name);*

The object server class

The object server lies in a class that

- implements the remote interface (e.g., see ComputeEngine later on)
- requires to explicitly define constructors for the remote object because they must throw RemoteException.

  Remark 15 This is necessary even if the super-class constructor is "enough" (in which case the constructor's body is just super)
- can contain other methods, however only those in the remote interface are available to the client.

  Remark 16 Clients will not have a reference to (objects of) the class that implements a remote interface, but only references to interfaces
- extends UnicastRemoteObject...(see next slides)

public class Server implements Hello {….}

```
Server obj = new Server();
Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
```

public class Server extends UnicastRemoteObject implements Hello

```
Hello stub= new Server();
```

preferable

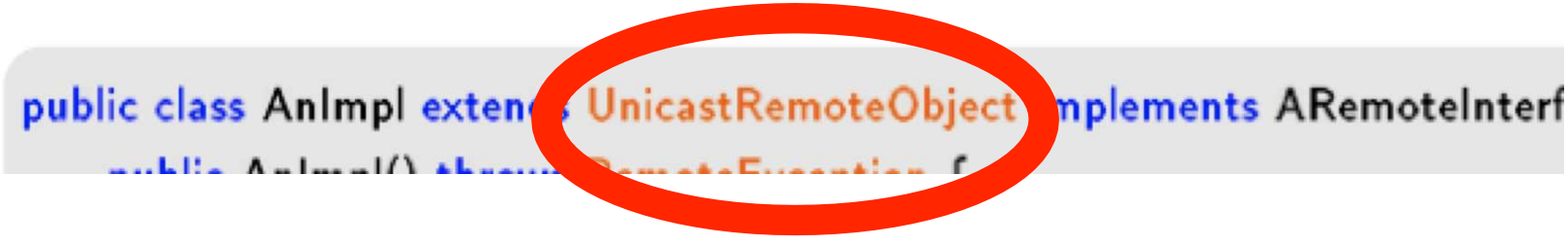
Implementing a remote interface

```
public class AnImpl extends UnicastRemoteObject implements ARemoteInterface {
    public AnImpl() throws RemoteException {
        super( );
    }
    public String aRemoteMethod1( ) throws RemoteException {
      ...
    }
    public int aRemoteMethod2( ) throws RemoteException {
      ...
    }
}
```

- Since remote objects can potentially throw a java.rmi.RemoteException,
  - a constructor that throws a RemoteException must be defined, even if the
  - constructor does nothing but invoking super
  - If such a constructor is missing, the compiler will produce an error message like

```
....: Exception java.rmi.RemoteException must be caught,
or it must be declared in the throws clause of this method.
```

The UnicastRemoteObject JAVA class



Remark 17 The implementation of a remote object r_obj does not have to extend UnicastRemoteObject (see next slides) but, in such case, r_obj's implementation must supply appropriate implementations of the java.lang.Object methods. Moreover, r_obj's implementation must explicitly call one of UnicastRemoteObject's exportObject methods in

order to make the RMI runtime aware of r_obj and letting it to accept incoming calls.

The class UnicastRemoteObject is part of the JAVA RMI API and it

- makes remote objects available and able to receive incoming invocations
- can include constructors and static methods used when exporting remote objects
- supplies several java.lang.Object methods (e.g, equals, toString...) in order to make them appropriately defined for remote objects

A closer look to UnicastRemoteObject

The constructors of UnicastRemoteObject create and export a new UnicastRemoteObject object using:

- an anonymous port:

```
protected UnicastRemoteObject() throws RemoteException
```

- a specific port:

```
protected UnicastRemoteObject(int port) throws RemoteException
```

The first constructor let the JAVA RMI runtime choose which port should be used, while in the latter the provided port is used.

Remark 18 The port is used for a server socket implicitly run by the constructors to be listening and catch the remote invocations directed to the object server.

Other methods in UnicastRemoteObject

- export a remote object on an anonymous port:

```
public static RemoteStub exportObject(Remote obj) throws RemoteException
```

- export a remote object on a provided port:

```
public static RemoteStub exportObject(Remote obj, int port) throws RemoteException
```

- remove the remote object, obj, from the RMI runtime:

```
public static boolean unexportObject(Remote obj, boolean force) throws NoSuchObjectExceptior
```

  - If unexportObject(obj, force) is successful, remote object obj will no longer accept incoming remote calls
  - If force is true, obj is withdrawn even if there are pending calls or it is still serving calls
  - If force is false, obj is unexported only if there are no pending calls
  - unexportObject(obj, force) returns true if successful and false otherwise

Remark 19 When a remote interface is implemented without extending UnicastRemoteObject, then the exportObject(...) method must be invoked.

Avoid extending UnicastRemoteObject

When implementing a remote interface:

- usually it suffices to extend UnicastRemoteObject
- it is easier to extend UnicastRemoteObject...but there might be drawbacks
  - remote objects cannot inherits from other classes
  - sometimes remote objects should not be immediately exported to clients!
- In order to tackle these problems, if UnicastRemoteObject is not suitable for your application, you should specify a server that does not extend UnicastRemoteObject.

Remark 20 If you take the latter option, you must override

- equals
- hashCode

(these are methods defined in the java.lang.Object).

So far...

- Comparison/ relations between remote and local objects
- Key points of server object classes
- UnicastRemoteObject

...now

The compute engine example

Remind that

> "[...] the compute engine [...] allows jobs to be submitted to the compute engine [...] to run those jobs, and the results of the job to be returned to the client[...] " [Sun]

and the interfaces are

```
public interface Compute extends
Remote {
  public Object executeTask(Task t)
        throws RemoteException;
}
```

```
public interface Task extends
Serializable {
     public Object execute();

}
```

- The remotely accessible part is specified by the Compute interface and has a single method
- throws RemoteException

- takes a Task which is not a remote interface

Implementing a remote interface: compute engine

```
public class
    ComputeEngine
    extends
    UnicastRemoteObject
    implements Compute{
    public ComputeEngine()
    throws RemoteException {
        super();
    }
    public Object executeTask(Task t)
    throws RemoteException {
        return t.execute();
    }

    public static void main(String[] args)
    { [...];
    }
```

The most involved method of the implementation is the main program which

- is not a remote method (indeed, it is static)
- starts the ComputeEngine
- prepares the server for accepting calls from clients
- makes the remote object available to clients

Compute engine & remote objects

```
public static void main(String[] args) {
    ...
    String host = "localhost"; // just an example
    String name = "rmi://" + host + "/Compute";
    try {
        Compute engine = new ComputeEngine();
        Naming.rebind(name, engine);
        System.out.println("ComputeEngine␣bound");
    } catch (Exception e) {
        System.err.println("ComputeEngine␣exception:␣"
        +
                           e.getMessage());
        e.printStackTrace();
    }
}
```

- the creation of the remote object exploits the constructor of the UnicastRemoteObject constructor, which exports the newly created object to the RMI runtime
- then, the object is bound in the registry
- this has to be done in a try-catch block

Remark 21 Notice the type of the engine object! Clients will have remote references to the Compute interface and (its methods), not to objects in the ComputeEngine class.

The main method creates/installs a security manager

```
public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new RMISecurityManager());
    }
    String host = "localhost"; // just an example
    String name = "rmi://" + host + "/Compute";
    try {
        Compute engine = new ComputeEngine();
        Naming.rebind(name, engine);
        System.out.println("ComputeEngine␣bound");
    } catch (Exception e) {
        System.err.println("ComputeEngine␣exception:␣" +
                           e.getMessage());
        e.printStackTrace();
    }
}
```

Compute engine & security

The main method creates/installs a security manager

```
public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new RMISecurityManager());
    }
    String host = "pc104.mcs.le.ac.uk"; // just an example
    String name = "rmi://" + host + "/Compute";
    try {
        Compute engine = new ComputeEngine();
        Naming.rebind(name, engine);
        System.out.println("ComputeEngine␣bound");
    } catch (Exception e) {
        System.err.println("ComputeEngine␣exception:␣" +
                           e.getMessage());
        e.printStackTrace();
    }
}
```

- protects local resources (e.g., the file system) from unauthorised accesses
- must be installed by any RMI application, otherwise no class is downloaded
- RMISecurityManager is a default manager applying a very strict security policy

- ♪ We do not discuss security issues of distributed applications
- ♪ For the purpose of this module, this is all you need to know about security

  * http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html

Making the Remote Object Available to Clients

After being created (and exported), an instance of ComputeEngine must be registered in a registry so that clients can obtain a reference to the remote object and invoke it.

```
public static void main(String[] args) {
    String host = "localhost"; // just an example
    String name = "rmi://" + host + "/Compute";
    ...
    try {
        Compute engine = new ComputeEngine();
        Naming.rebind(name, engine);
        System.out.println("ComputeEngine␣bound");
    } catch (Exception e) {
        System.err.println
        ("ComputeEngine␣exception:␣"
        +e.getMessage());
        e.printStackTrace();
    }
}
```

The class java.rmi.Naming is the JAVA registry; Its methods

- allow clients to get a reference to a remote object by name
- bind, register, and look-up for remote objects in the RMI registy

Naming Remote Objects on the server side

We consider now a few features of the JAVA RMI registry

- Binding a remote object to a name means to register the object under a name which can be used later to look up for references to the remote object
- In general, a naming (or directory) service allows clients to find remote services

Remark 22 Is this approach a vicious circle? How can a client locate a service by using another service? Actually, the solution to this paradoxical situation is

- to treat the naming service as a special service that runs on a known host and port number
- many directory services can be used

A very simple naming service, called RMI Registry, is embedded in JAVA RMI

- an RMI registry is executed on each JVM were an object server is running
- once a remote object is exported, RMI creates a listening service waiting for clients' invocations which corresponds to the remote reference sent to the client by the registry

The java.rmi.Naming class

The Naming class is defined as `public final class Naming extends Object` and

- yields methods for storing (references to) remote objects in an RMI registry yields
- methods for retrieving (references to) remote objects in an RMI registry

Each method of java.rmi.Naming takes string in URL format of the form

rmi://[<host_name>[:<name_service_port>]]/<service_name> where

host_name is the host (remote or local!) where the registry is located

registry_port is the port number on which the registry accepts calls

service_name is a simple string (not interpreted by the registry)

Both host_name and registry_port are optional

- if host_name is omitted, the default host is the local host if
- registry_port is omitted, the default port is 1099

e.g. rmi://localhost/Compute

URL in the ComputeEngine example

The main method of ComputeEngine assigns a name to the remote object engine

```
String name = "rmi://"+ host + "/Compute";
```

where

- the name of the host is (i.e.,rmi://localhost)
- The service name for the remote object in the registry is the string “Compute”

Exercise 13 Modify ComputeEngine so that a port different from the default one is used.

An object server can

- use a registry common to other servers running on its host
- create and use its own registry

Binding names using Naming

java.rmi.Naming offers two methods for binding names:

to bind name to obj	`static void bind(String name, Remote obj)`
to reassign name to obj	`static void rebind(String name, Remote obj)`

Both bind and rebind throw MalformedURLException, RemoteException while bind may throw AlreadyBoundException.

Exercise 14 Is it possible to register two instances of the same server object? If yes how? Is it possible to register a server object on two different registries?

java.rmi.Naming provides also a method for unbinding names:

unbind the specified name	`static void unbind(String name)`

throws RemoteException, NotBoundException, MalformedURLException

Remark 23 All the strings must have the URL format for a service.

So far...

- An example of server object
- Security issues
- RMI registry

...now

Take a walk on the client side

- The ComputeEngine server is quite simple: it just has to get a Task object and invoke its execute method
- clients for ComputeEngine are more complex as they need to define the actual task to be performed by invoking a ComputeEngine server

Suppose that we want to write a client whose Task computes:

x+y

We can proceed by defining 2 separate classes:

Adding	actually implements the Task interface
ComputeAdding	looks up and calls a Compute object

Remark 24 Observe that Adding is the actual class that defines the work to be done by the compute engine, while ComputeAdding simply invokes a Compute server.

Task objects

```
public class Adding implements Task{
    private double x = 0.0;
    private double y = 0.0;
    public Adding (double x, double y){
        this.x = x;
        this.y = y;
    }
    public Double execute(){
        return Double.valueOf(x+y);
    }
}
```

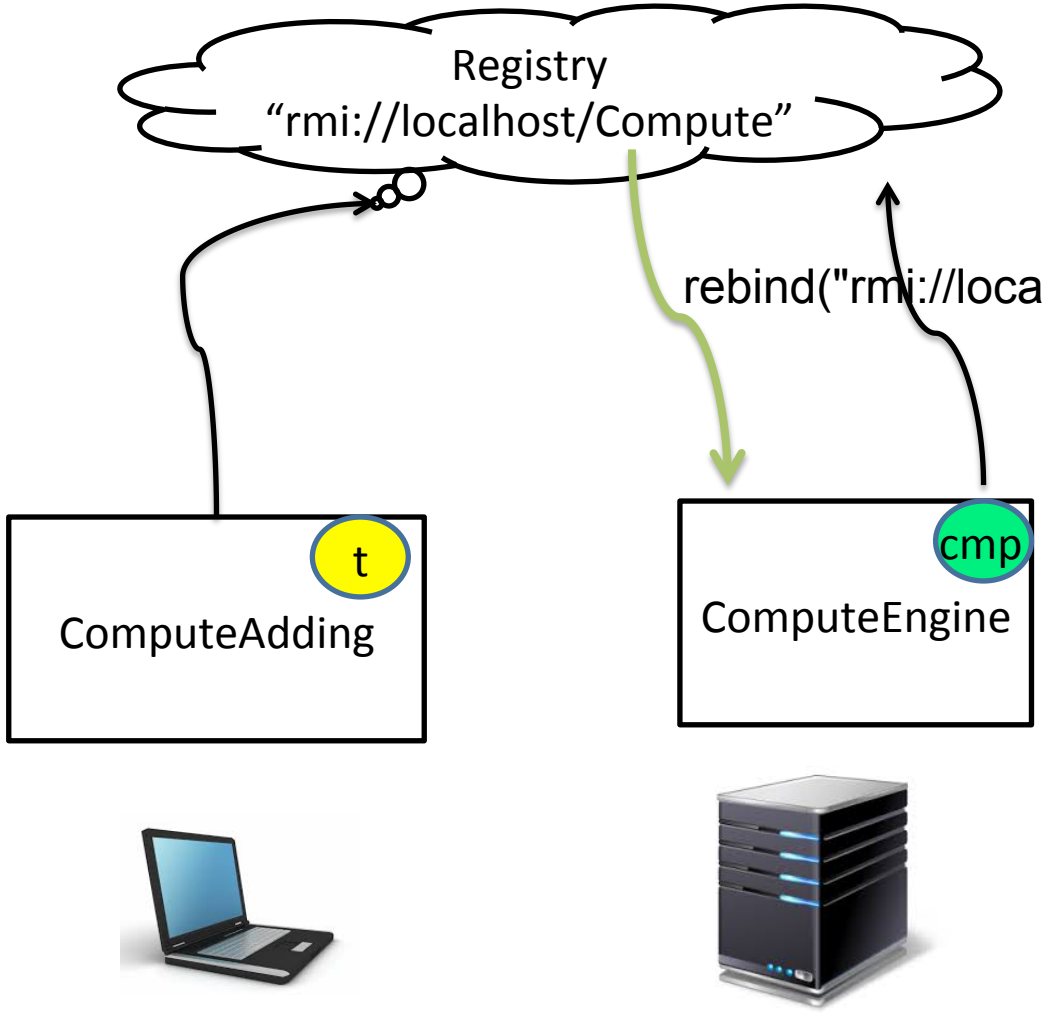
- The class Adding implements Task
- The arguments are passed in the constructor
- The execution of the task is trivial
- The conversion is necessary

Remark 25 What the supplied Task object computes is immaterial to the ComputeEngine remote object. For instance, implementing a task that multiplies two doubles would have been quite similar.

Exercise 15 Write a class ComputeStringCat that concatenates two strings and is suitable to be used by a ComputeEngine server.

Finding and invoking the remote server (1)

The Compute object knows only that each object it receives implements the execute method; it does not (need to) know what the implementation actually does.



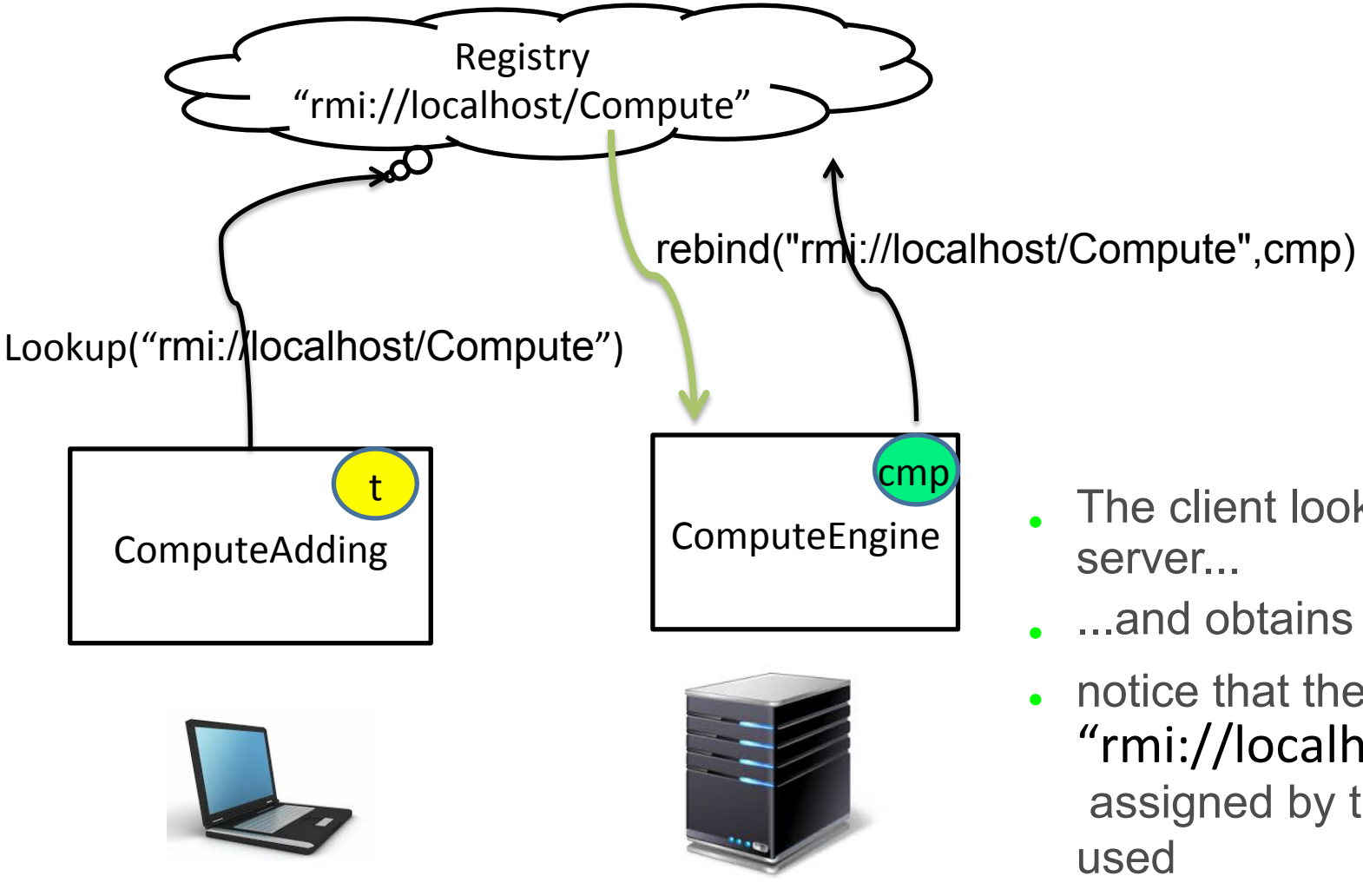
- Server and client must use the same registry
- The server has bound the remote object

t is an object of the class Task

cmp is a remote object

Finding and invoking the remote server (1)

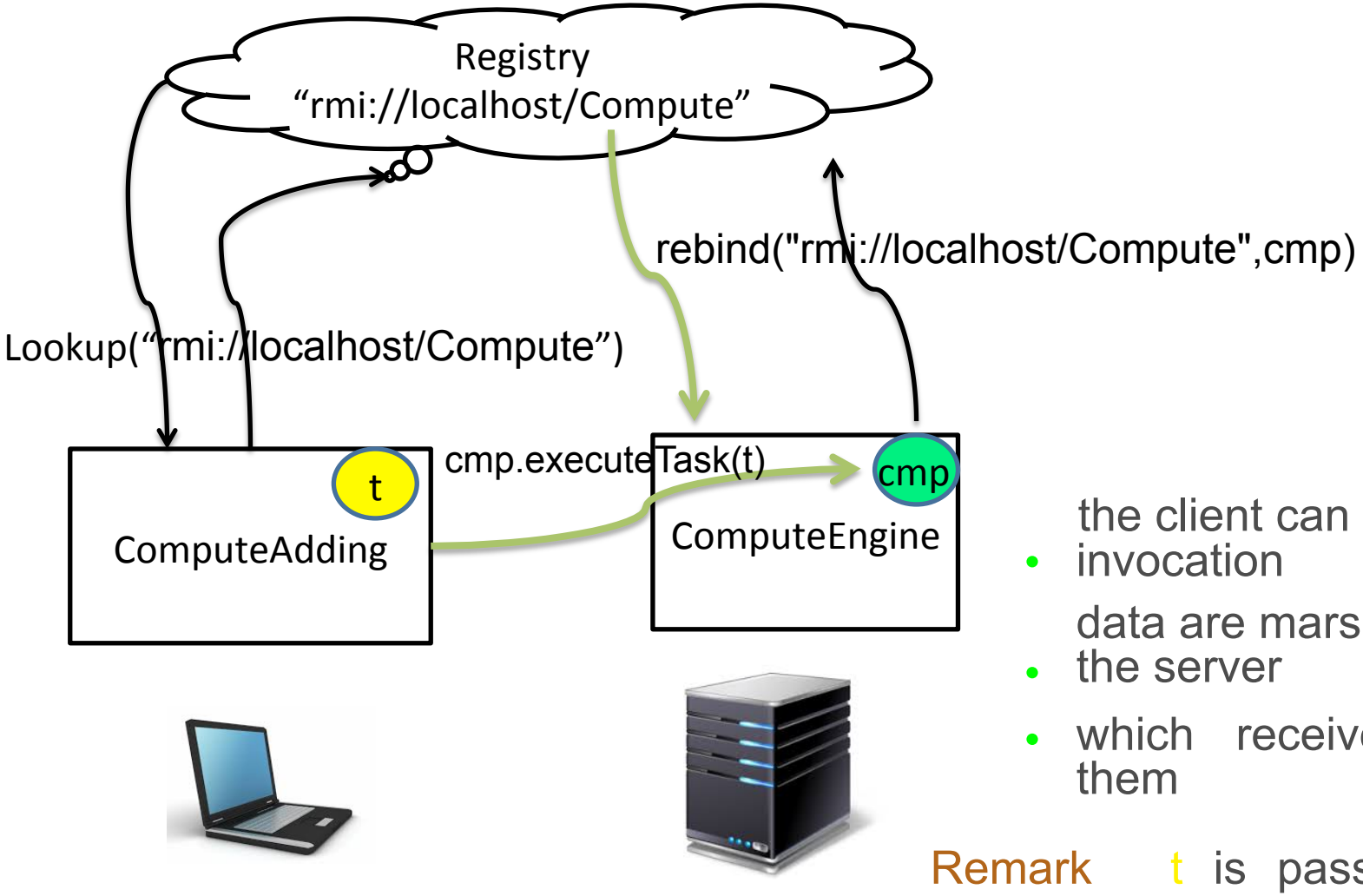
The Compute object knows only that each object it receives implements the execute method; it does not (need to) know what the implementation actually does.



- The client looks up for a Compute server...
- ...and obtains a remote object
- notice that the string “rmi://localhost/Compute” assigned by the server must be used

Finding and invoking the remote server (1)

The Compute object knows only that each object it receives implements the execute method; it does not (need to) know what the implementation actually does.



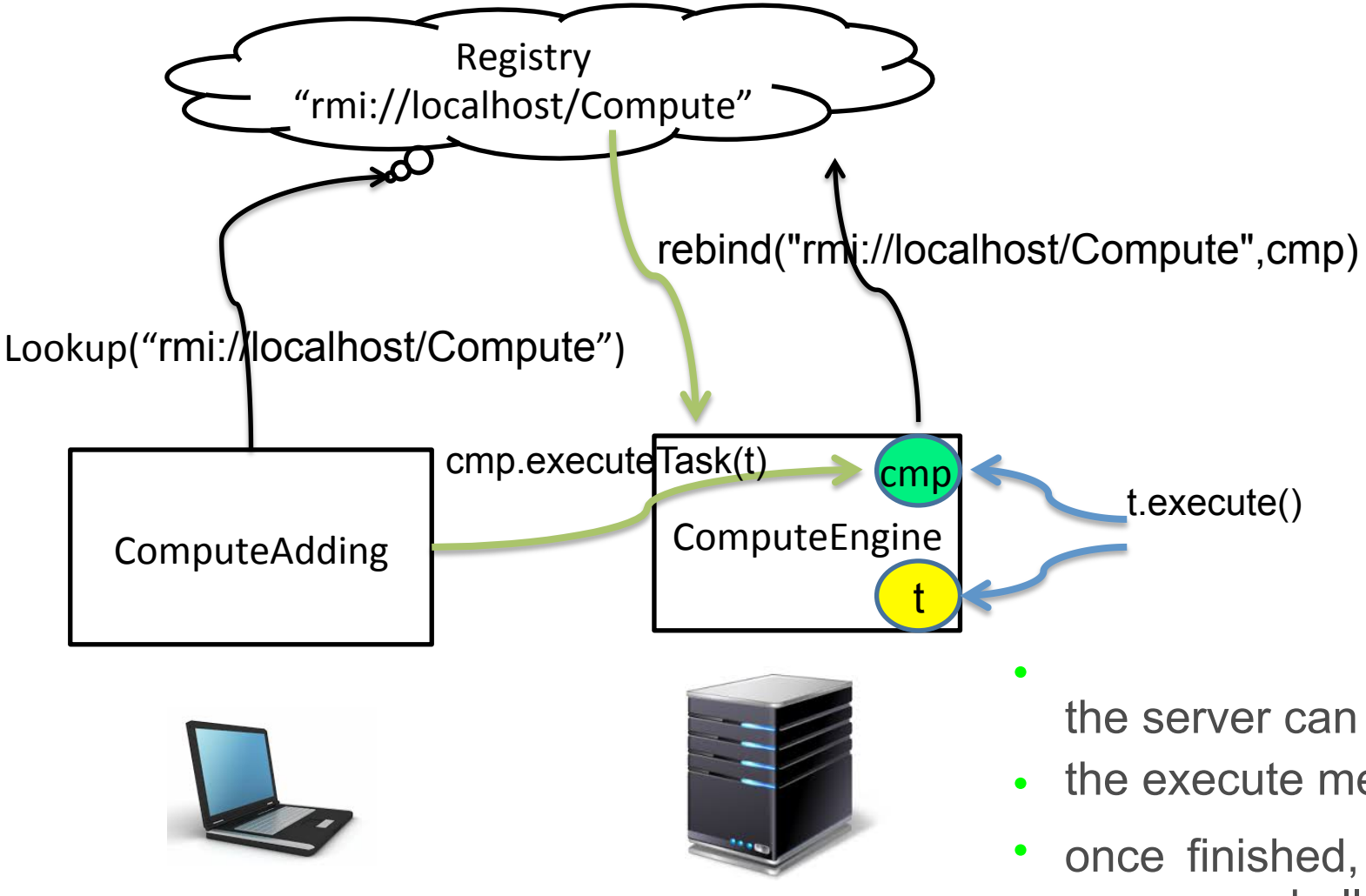
- the client can now make a remote invocation
- data are marshalled and sent to the server
- which receives and unmarshalls them

Remark cmp is passed by reference

Remark t is passed by copy and obtained by unmarshalling the data sent by the client

Finding and invoking the remote server (1)

The Compute object knows only that each object it receives implements the execute method; it does not (need to) know what the implementation actually does.



- the server can now use data
- the execute method of p is invoked
- once finished, the result is marshalled and sent back to the client

Finding and invoking the remote server (2)

```
public class ComputeAdding {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            String name = "rmi://" + args[0] + "/Compute";
            Compute comp = (Compute) Naming.lookup(name); Adding
            task =new Adding (
                                Integer.parseInt(args[1],In
                                teger.parseInt(args[2])
                                  );
            double result= ((Double)
                (comp.executeTask(task))).doubleValue();
            System.out.println("The result is" + result);
        } catch (Exception e) {
            System.err.println("ComputeAdding␣exception:␣" +
                                  e.getMessage());
            e.printStackTrace();
        }
    }
}
```

- The client installs a security manager

  Exercise 16 Why?

- The registry host name is passed as parameter to the main lookup queries the registry for a Compute server

- A Task is created and a remote method invocation to the executeTask method of the remote object is done

  Remark 32 Notice that the remote method invocation to Compute triggers the uploading of the Adding class on the server host.

  Remark 33 All the conversions are necessary because executeTask returns Objects members.

Naming Remote Objects on the client side

On the client side, the RMI Registry is accessed through the class java.rmi.Naming that offers two methods for looking-up names:

- to get a reference to the remote object associated with a specified name:

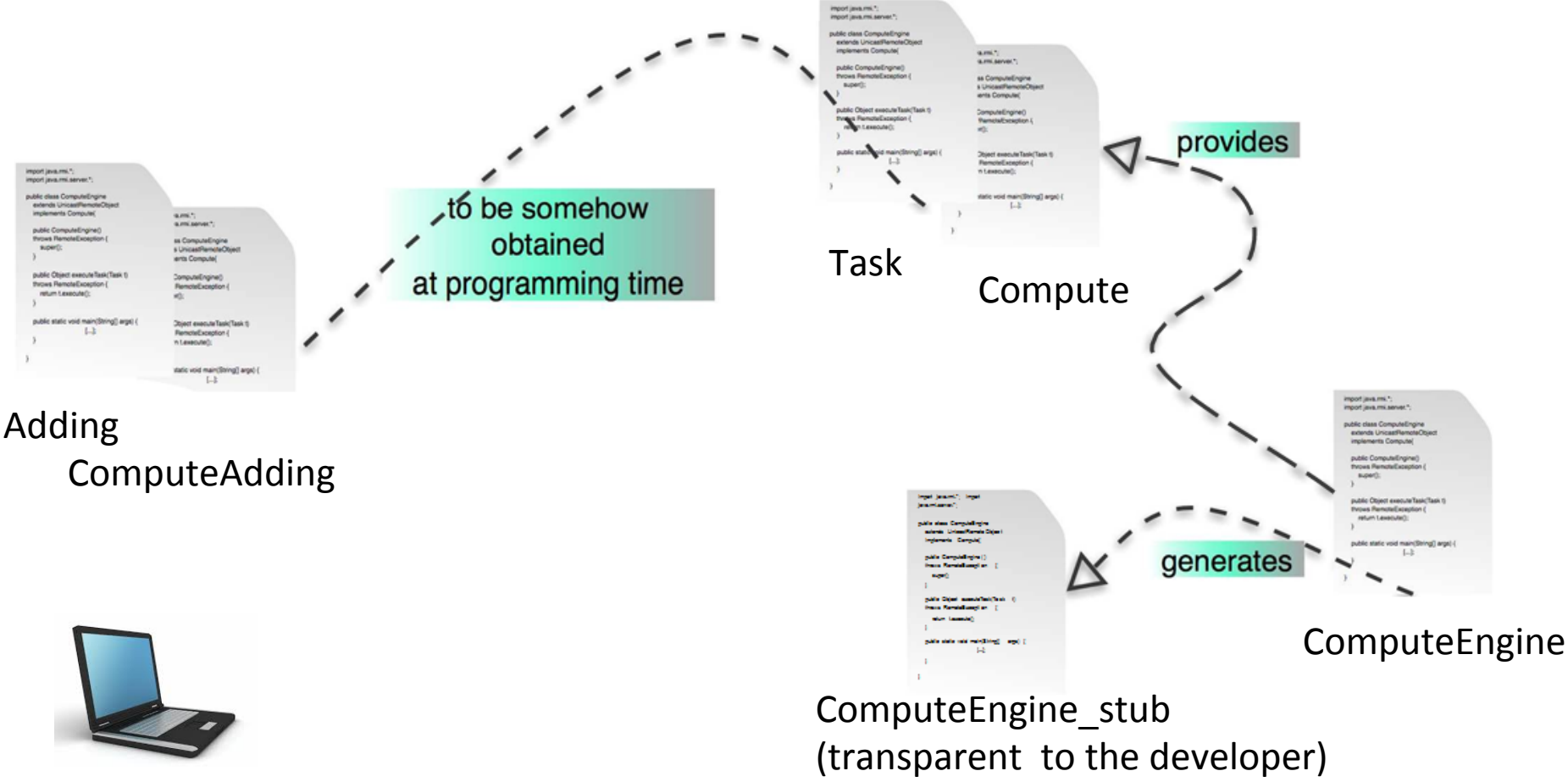
```
public static Remote lookup(String registry_name)
```

  - lookup() takes the URL of the service written as
    rmi://[<host_name>[:<name_service_port>]]/<service_name> lookup()
  - returns a reference to a remote object

Both these methods throw RemoteException, MalformedURLException while lookup throws also NotBoundException.

Remark 34 All the strings must be a URL to a registry and it is necessary to specify the name_service_port in the URL only if it is different from 1099.

Summary of files of the compute engine example



Java EE

Enterprise Beans

JavaBeans

*"**JavaBeans** technology is the component architecture for the Java Platform, Standard Edition (JavaSE). JavaBeans components (beans) are reusable software programs that you can develop and assemble easily to create sophisticated applications. …. JavaBeans technology is based on the Java Bean specification" – Sun Microsystem*

- A **bean** is a Java class with method names that follow the JavaBeans guidelines.
- All you have to do is make your class *look* like a bean!
- A Java object class must obey certain **conventions** to become a bean

JavaBean conventions

To make a Java Bean, this class must follow these guideline

- have a public default constructor .
- properties must be accessible using getter and setter
- The class should be serializable

```
package CO3090.transation;

import java.io.Serializable;

public class PersonBean implements Serializable{

    public PersonBean() {
        super();
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    String name;
    int age;
}
```

Implements java.io.Serializable

Public default constructor

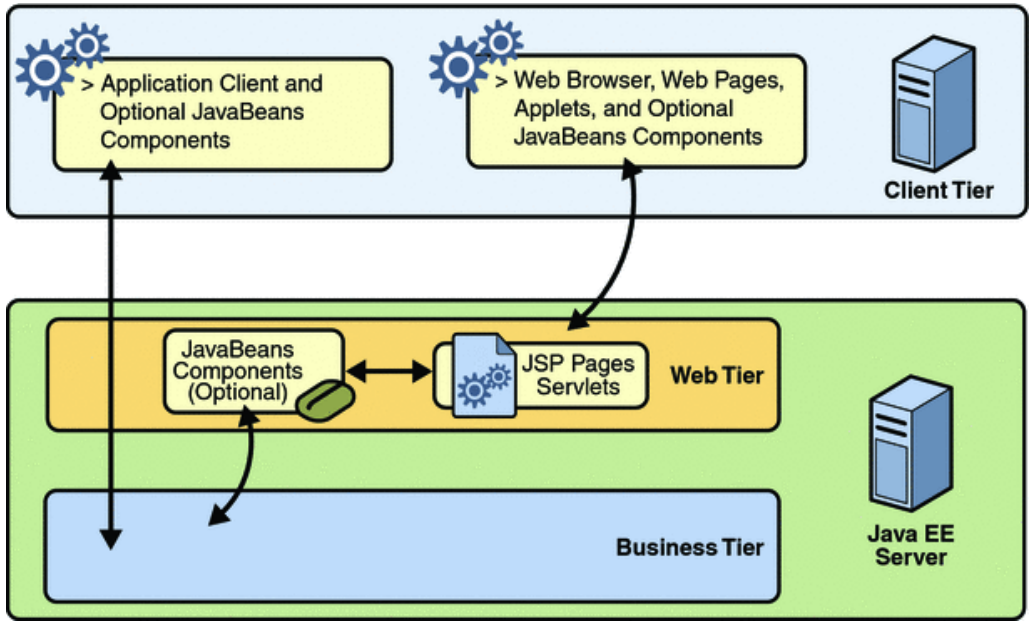
Getter and setter

Example of a Javabean

http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html

Use of Java bean in JSP

```
package CO3090.transation;

import java.io.Serializable;

public class PersonBean implements Serializable{

    public PersonBean() {
        super();
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    String name;
    int age;
}
```



Use JavaBean in a JSP page:

```
<jsp:useBean id="person" class="PersonBean" scope="page"/>
<jsp:setProperty name="person" property="age" value="10"/>
……
<b> Age: <jsp:getProperty name="person" property="age"/></b>
<%
  int age= person.getAge();
%>
```

Example of a Javabean

Enterprise JavaBeans (EJB)

What is Enterprise JavaBean (EJB)?

"An **Enterprise Bean (EJB)** is a server-side component that encapsulates the business logic of an application."

EJB technology supports application development based on a multiplier, distributed object architecture in which most of application's logic is moved from the client to the server.

EJB is a specification

It 's not a product

As is XML, TCP/IP

EJB is Java based

Defined by Sun Microsystems

Applications are written in Java

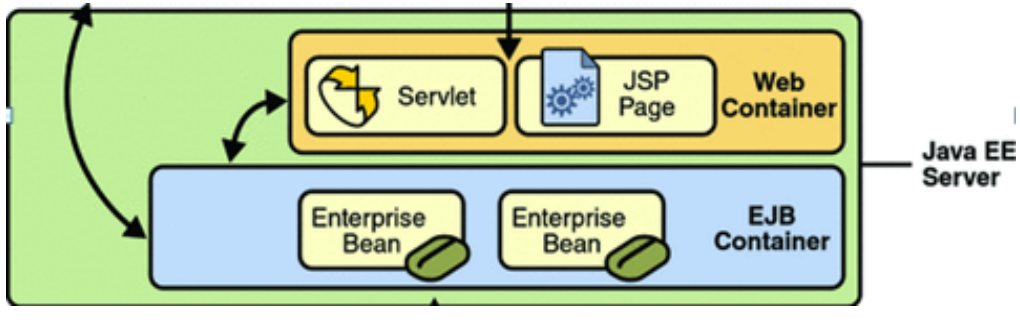
EJBs are distributed objects

Similar to Java RMI

EJBs are components

Similar to regular JavaBeans

EJB products are Transactional Monitor



EJB specification

- EJB supports
- Remote interfaces
- RMI-IIOP, CORBA, Web Service
- Transaction processing
- Java Transaction API (JTA)
- Data persistence
- Java Persistence API (JPA)
- Container-managed persistence
- Bean-managed persistence
- Message-Driven programming
- Java Message Service (JMS)
- Naming and directory service
- JNDI
- Concurrency control

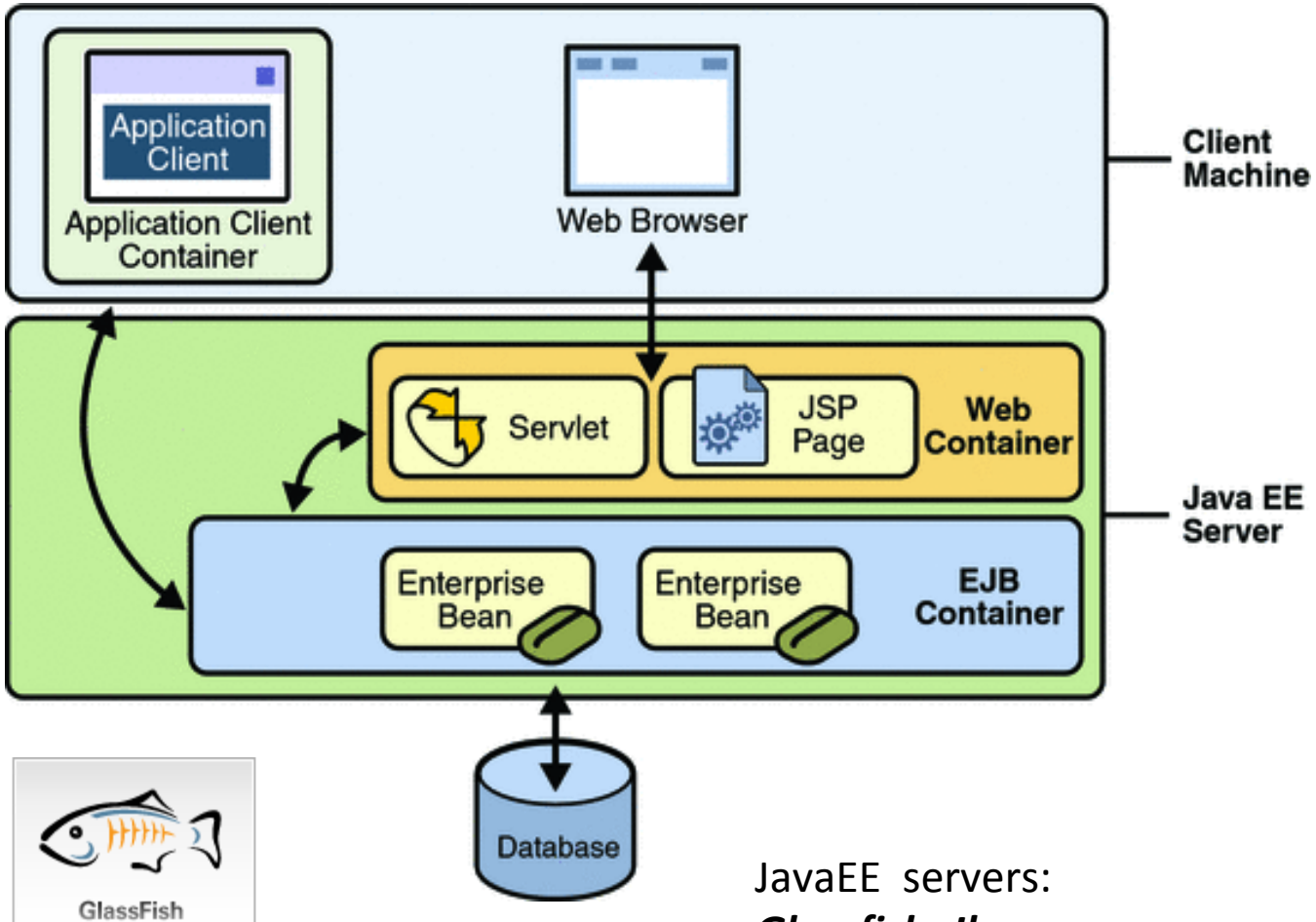
Main advantages

- Distributed
- Multithreaded
- Transactional
- Persistent

http://www.oracle.com/technetwork/java/docs-135218.html

Java EE servers

A Java EE server is a server application that the implements the Java EE platform APIs and provides the standard Java EE services.

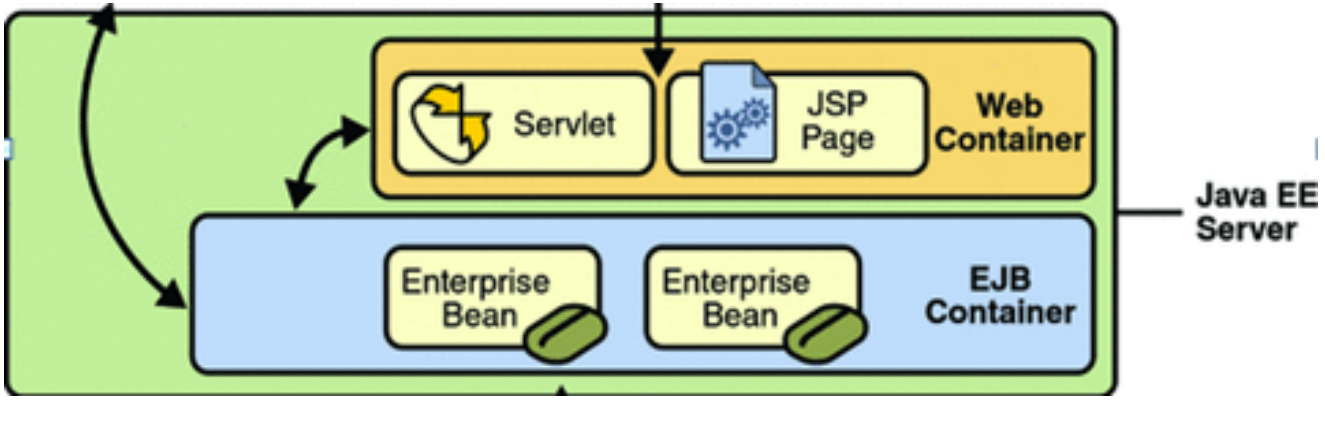


JavaEE servers:
Glassfish, Jboss

- Java EE servers are sometimes called application servers
- Java EE servers host several application component types that correspond to the tiers in a multi-tiered application.
- The Java EE server provides

Web container vs EJB container

	Web container	EJB container
Jobs	Manages the execution of web pages, JSP, servlets	Manages the execution of EJB, JMS, JTA run on the Java EE server
handles	A server that handles HTTP protocol.	Serve enterprise based applications
Runs on	web server (e.g. Apache Tomcat, Jetty) Many application servers may contain a web server internally.	Application server (e.g. JBoss, Glassfish)



Types of Enterprise Beans

Session Beans: prerforms a task for a client, may be implemented as a web service

Stateful Session Beans
Stateless Session Beans
Singleton Session Beans
~~Entity beans (deprecated in EJB 3.2)~~

Message-Driven Beans: (MDBs) acts as a listener for a particular messaging type

Message-driven beans (asynchronous)

Session beans

"A Session Bean represents a single client inside the Application Server. It represents a single conversation with a client. …"

Session Beans are

- created and used by a single client.
- not shared
- not recoverable—if the EJB server fails, it may be destroyed.
- not persistent or typically, persists only for the life of the conversation with the client.
- not re-entrant.

Stateful session beans

Example of stateful session bean:

```
import javax.ejb.LocalBean;
import javax.ejb.Stateful;

/**
 * Session Bean implementation class Counter
 */
@Stateful
@LocalBean
public class Counter implements CounterRemote {
    /**
     * Default constructor.
     */
    public Counter() {
        // TODO Auto-generated constructor stub
    }

    public void increase(){
        count++;
    }

    public int  getCounter(){
        return count;
    }

    int count=0;

}
```

Stateful Session Bean

The state is retained for the duration of the client-bean session. If the client removes the bean or terminates, the session ends and the state disappears.

Stateless session beans

Example of stateless session bean:

```
import javax.ejb.LocalBean;

/**
 * Session Bean implementation class
 * Calculator
 */
@Stateless
@LocalBean
public class Calculator
        implements CalculatorRemote {

    /**
     * Default constructor.
     */
    public Calculator() {
        // TODO Auto-generated
    }

    public int add(int a, int b){
        return a+b;
    }

}
```

A stateless session bean **does not maintain a conversational state** with the client.

When a client invokes the methods of a stateless bean, the bean's instance variables may contain a state specific to that client, but only for the duration of the invocation.

Singleton Session Beans

Example of singleton session bean:

```
import javax.ejb.LocalBean;
import javax.ejb.Lock;
import javax.ejb.Singleton;
import javax.ejb.LockType;
import javax.ejb.ConcurrencyManagementType;
import javax.ejb.ConcurrencyManagement;

@Singleton
@LocalBean
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
public class ShareCounter implements ShareCounterRemote {

    int counter;

    public ShareCounter() {
       counter=0;
    }

    @Lock(LockType.WRITE)
    public void intcrease() {
        counter++;

    }

    @Lock(LockType.READ)
    public int getCounter() {
        // TODO Auto-generated method stub
        return counter;
    }

}
```

Singleton Session Beans

they are objects having a global shared state within a JVM. Concurrent access to the one and only bean instance can be controlled by the container (Container-managed concurrency, CMC) or by the bean itself (Bean-managed concurrency, BMC).

Singleton Session Beans

Example of singleton session bean:

```
import javax.ejb.LocalBean;
import javax.ejb.Lock;
import javax.ejb.Singleton;
import javax.ejb.LockType;
import javax.ejb.ConcurrencyManagementType;
import javax.ejb.ConcurrencyManagement;

@Singleton
@LocalBean
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
public class ShareCounter implements ShareCounterRemote {

    int counter;

    public ShareCounter() {
       counter=0;
    }

    @Lock(LockType.WRITE)
    public void intcrease() {
        counter++;

    }

    @Lock(LockType.READ)
    public int getCounter() {
        // TODO Auto-generated method stub
        return counter;
    }

}
```

CMC

Container-managed concurrency (CMC) can be tuned using the ***@Lock*** annotation, that designates whether a read lock or a write lock will be used for a method call.

Message-Driven Beans (MDBs)

*"A **message-driven bean** is an enterprise bean that allows Java EE applications to process messages asynchronously. It normally acts as a JMS message listener, which is similar to an event listener except that it receives JMS messages instead of events."*

Message-driven bean

- are stateless
- Invoked asynchronously
- are instances retain no data or conversational state for a specific client.
- all instances of a message-driven bean are equivalent, MDB pool allows messages to be processed concurrently.
- A single message driven beans can process messages from multiple clients

Message-Driven Beans (MDBs)

Example of singleton session bean:

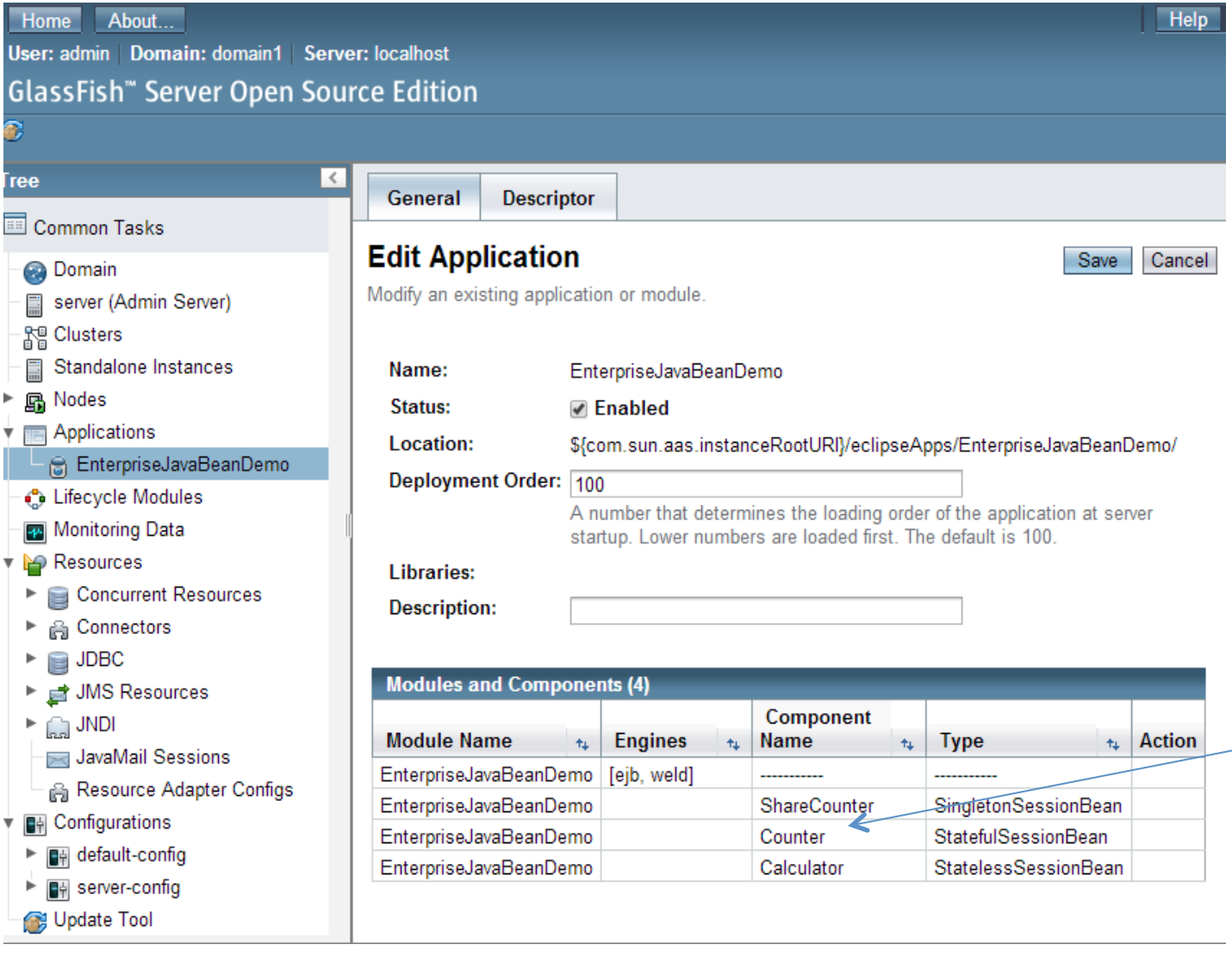
```
import javax.ejb.ActivationConfigProperty;

@MessageDriven(
    activationConfig = { @ActivationConfigProperty(
        propertyName = "destinationType",
        propertyValue = "javax.jms.Queue"),
                         @ActivationConfigProperty(
        propertyName = "destination",
        propertyValue ="/queue/MsgQueue")
})
public class MessageDrivenBean implements MessageListener {

    public MessageDrivenBean() {
        // TODO Auto-generated constructor stub
    }

    public void onMessage(Message message) {
        ObjectMessage objectMessage = null;
        try {
           objectMessage = (ObjectMessage) message;
           Object msg= objectMessage.getObject();

        // do something here
        // after receiving messages

        } catch (JMSException ex) {
           ex.printStackTrace();
        }
     }

    }
```

MessageListener

When a message arrives, the container calls the message-driven bean's onMessage method to process the message.

Enterprise Beans deployment

Open source Application Server and EJB container - **Glassfish**



https://glassfish.java.net/

EJB components deployed

EJB invocation from clients


```
package CO3090.beantest;
import CO3090.bean.*;

import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;

public class BeanTest {

public static void main(String[] args){

  try{
     Properties props = new Properties();
     props.put(Context.INITIAL_CONTEXT_FACTORY,
                       "com.sun.enterprise.naming.SerialInitContextFactory");
    props.setProperty("org.omg.CORBA.ORBInitialHost", "localhost");

              // glassfish default port value will be 3700,
    props.setProperty("org.omg.CORBA.ORBInitialPort", "3700");

    CalculatorRemote calculator =
         (CalculatorRemote) new InitialContext(props).lookup("ejb/CalculatorRemote");

      System.out.println(calculator.add(1, 2));

        }catch(Exception ex){
            ex.printStackTrace();
        }
    }
}
```

CORBA

Example:

Invoking ***add(int a, int b)*** of Stateless session bean ***Calculator***

From a client

Expose stateless session EJB as JAX-WS Web Service

```java
import javax.ejb.LocalBean;
import javax.ejb.Stateless;
import javax.jws.WebMethod;
import javax.jws.WebService;

/**
 * Session Bean implementation class
 * Calculator
 */
@Stateless(mappedName="ejb/CalculatorRemote")
@WebService(targetNamespace="http://127.0.0.1")
@LocalBean
public class Calculator
       implements CalculatorRemote {

    public Calculator() {
        // TODO Auto-generated
    }

    @WebMethod
    public int add(int a, int b){
        return a+b;
    }
    @WebMethod
    public int mul(int a, int b){
        return a*b;
    }
}
```

(JAX-WS) = Java API for XML Web Services
Web Services are **platform-independent** and **language-independent**

@WebService annotation is used to mark a class as a web service end point

@WebMethod annotation is used to mark a method as part the remote interface

When to use…

Stateful session beans

- The bean's state represents the interaction between the bean and a specific client.
- The bean needs to hold information about the client across method invocations ..

Stateless session beans

- The bean's state has no data for a specific client.
- In a single method invocation, the bean performs a generic task for all clients
- The bean implements a web service.

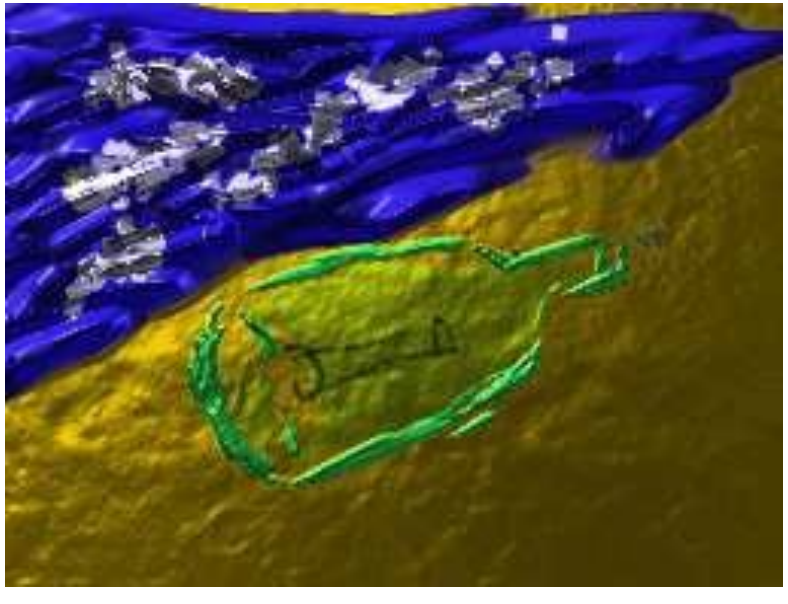
Singleton session beans

- State needs to be shared across the application.
- A single enterprise bean needs to be accessed by multiple threads concurrently
- The application needs an enterprise bean to perform tasks upon application startup and shutdown
- The bean implements a web service

Message driven bean

- The bean needs to receive and process messages asynchronously

Distributed coordination
&
concurrency (in JAVA)



Distribution vs Coordination

One of the most important issues of distributed systems/applications is how to coordinate distributed activities.

- in absence of “centralisation points”, the problem may be unsolvable in its full generality
- we cannot consider all coordination issues therefore we focus on
  - distributed transactions
  - complex communication mechanisms, and specifically message oriented middlewares (MOMs)
- we’ll first present the general principles of distributed transactions and MOMs
- and then consider how they have been “interpreted” in JAVA

Transactions: the problem

Typically, information

- is stored in several DBs
- must be accurate, up to date, and reliable
- integrity is affected by concurrency

Moreover, in a distributed system/application

- integrity depends on the consistency of the data stored in distributed DBs access
- might require many steps involving distributed components (e.g., DBMS)
- it is possible that after several steps, the access to a remote DB creates problems (e.g., network connection is not available)

Hence, recovery policies must be adopted for restoring the status of the information.

What are transactions?

Intuitively, a transaction is a sequence of steps that

- either “ends positively”
- or “fails”

More precisely, a transaction is a sequence of steps that must be logically considered as unique. A transaction can either

- commit: the effects of the transaction becomes “effective” or
- fail: the transaction has to rollback

Example 1 A transaction might look like the following pseudocode:

```
transaction begin
   debit account 1;
   credit account 2;
   update logs;
commit transaction
```

In order to correctly execute a bank transfer

either the sequence of steps must be successfully executed

or the whole computation must be aborted, if any problem arise in one of the steps

Aborting computations is sometime necessary to preserve the integrity of data; for instance, the status of the bank accounts (and of the logs of systems) would be inconsistent without aborting the whole transaction.

Properties of transactions

Transactions have been introduced in DB theory in order to envisage complex interactions among DBMS as unitary computations

- Usually, transactional computations must be A · C · I · D
  - Atomicity: either all or none of the tasks of a transaction are performed
  - Consistency: a transaction can't break integrity constraints
  - Isolation: operations in a transaction appear isolated from all other operations
  - Durability: effects of committed transaction are persistent

Remark 1 A very recent research topic is Long running transactions (LRT), namely transactions with long life span (usually, web based commercial transactions). For LRT some properties must be relaxed; for instance, LRT are not atomic.

The scope of a transaction

In a distributed setting, each object participating in a transactional activity is responsible for rolling-back to a consistent state in case of failure. Thereforere, it is important to determine which is the scope of a transaction.

> The scope of a transaction is the set of objects/methods involved in the transactional activity

Example 2 The scope of the transaction in Example1 involves

```
transaction begin
   debit account 1;
   credit account
   2; update log;
commit transaction
```

- the two accounts (possibly of different
- banks!) the logging files
- the methods/procedure invoked (e.g., debit, credit)

Remark 2 In general, middlewares specify a transaction manager that controls the boundaries of transactions

The 2-Phase commit protocol

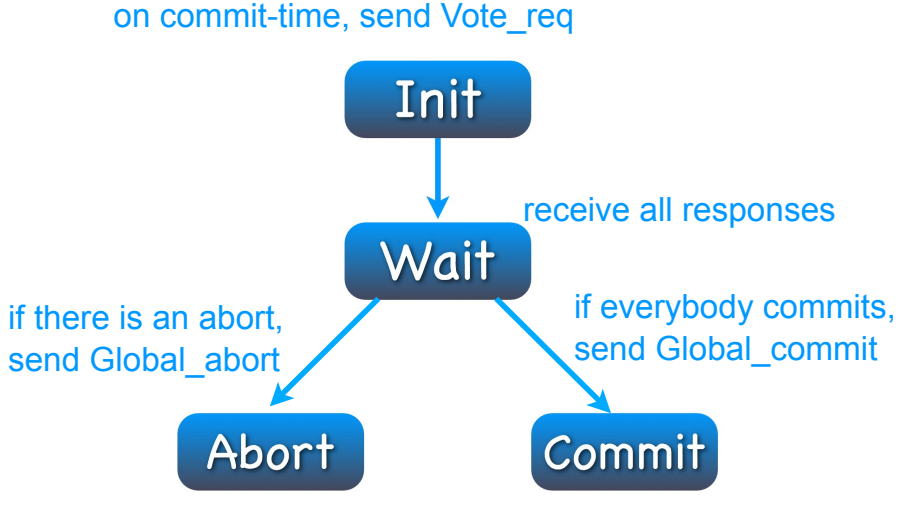
2PC is a protocol (distributed algorithm) used for coordinating the participants to a transaction. 2PC is used for

- deciding if a distributed transaction must
  - commit
  - or fail

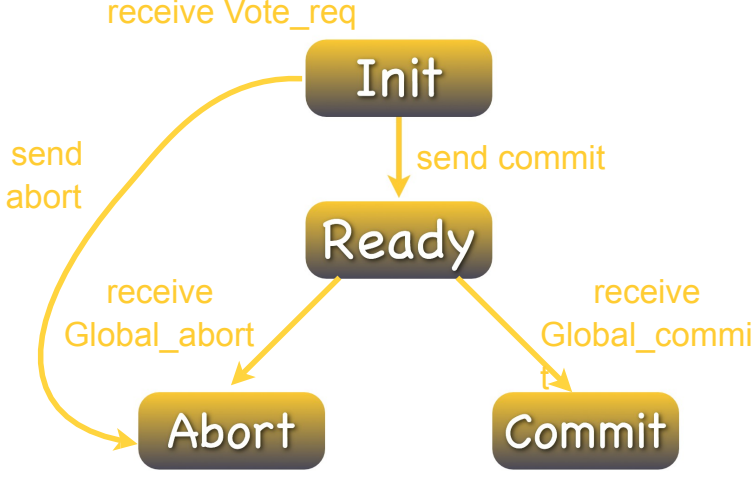
The roles of the protocol are

- Actors/participants (assumed to run in different execution environments)
- A (single) coordinator

Coordinator and actors behaves as follows:

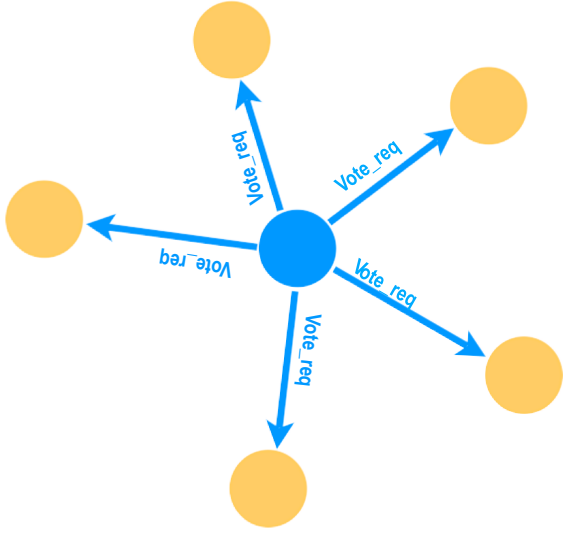


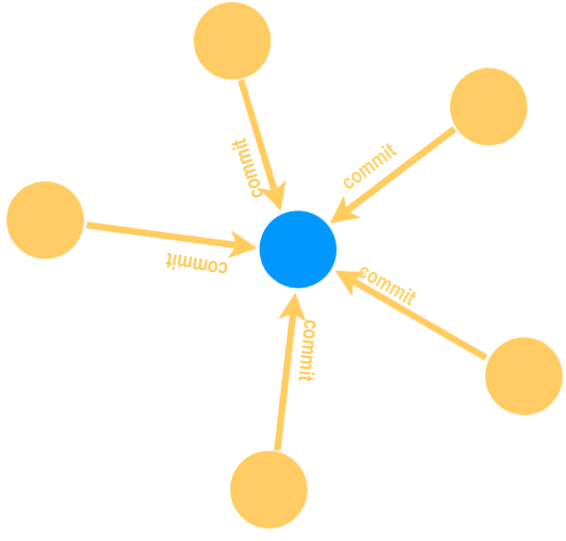
Coordinator



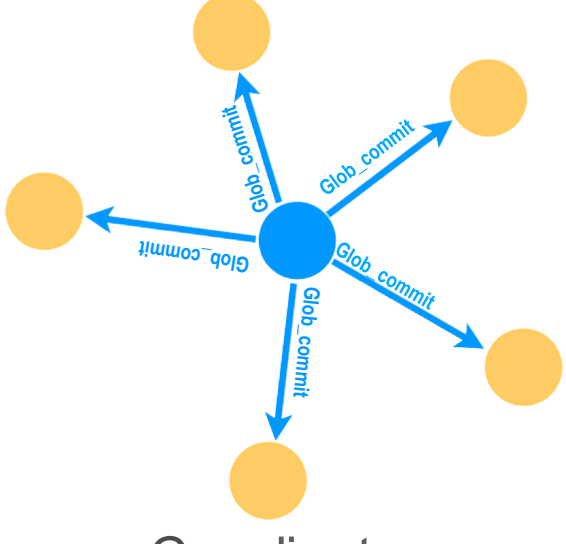
Actors

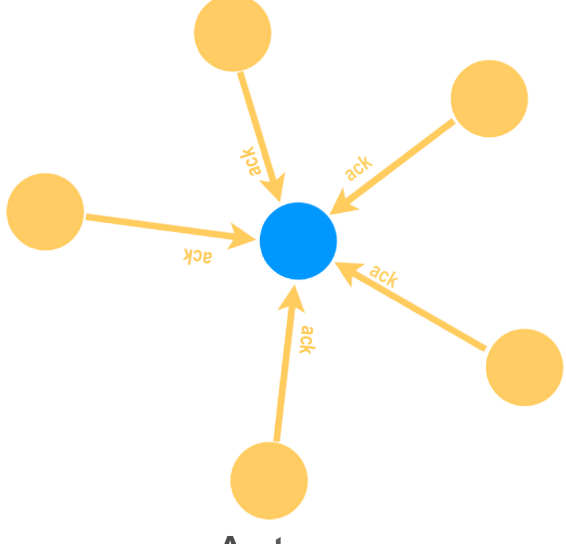
The 2 phases of 2PC: commit





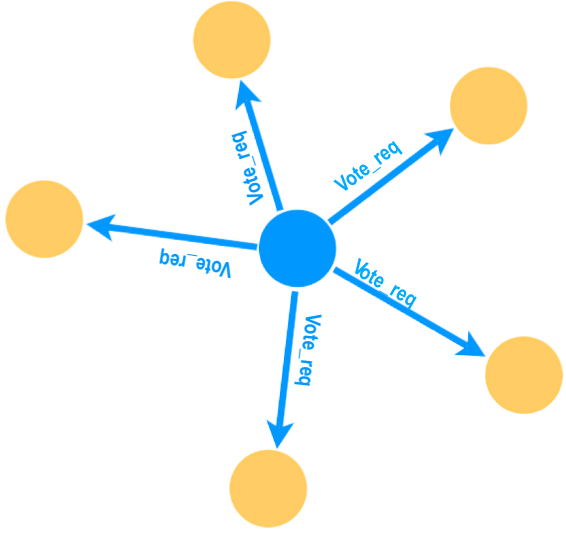
Phase 1

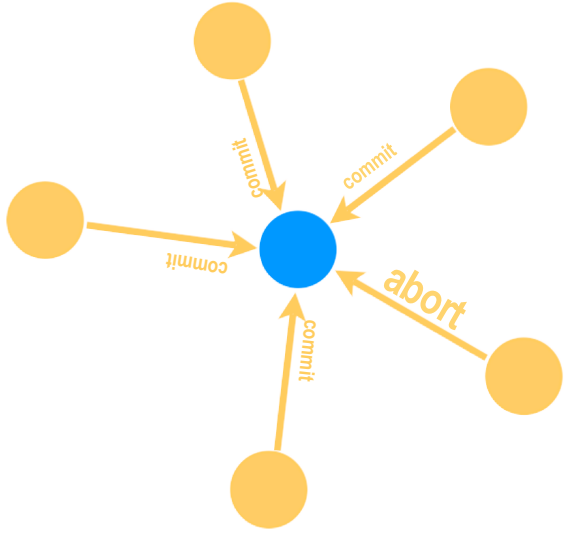




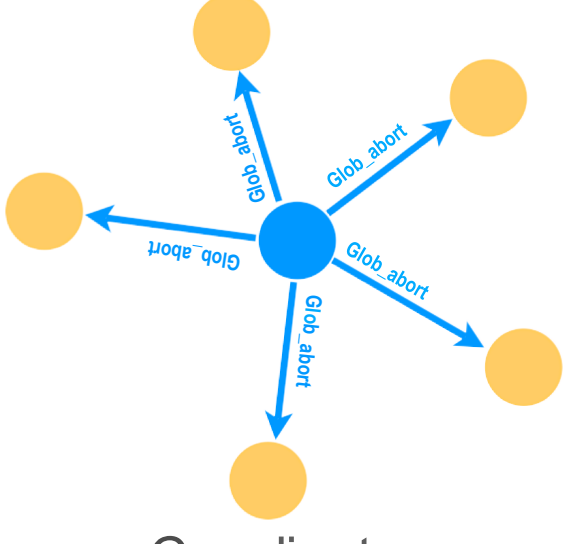
Phase 2

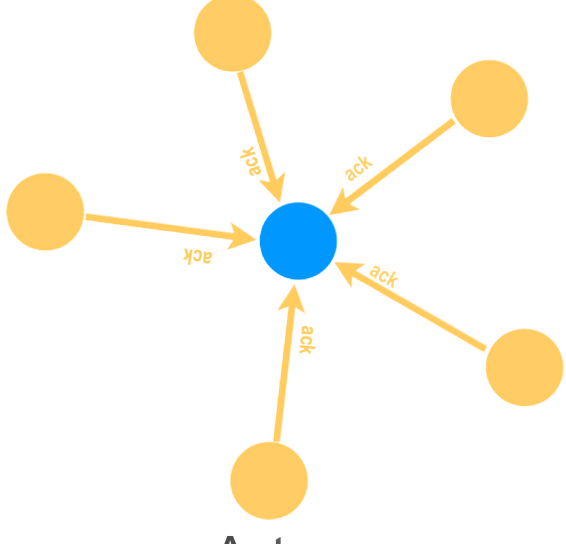
The 2 phases of 2PC: abort





Phase 1





Phase 2

So far...

- Transactions
- Scope of transactions
- 2PC

Remember:

- Properties of transactions
- Scope of transactions
- Transaction attributes
- Further reading [JBC+07] chapt. 33

Container-Managed Transactions

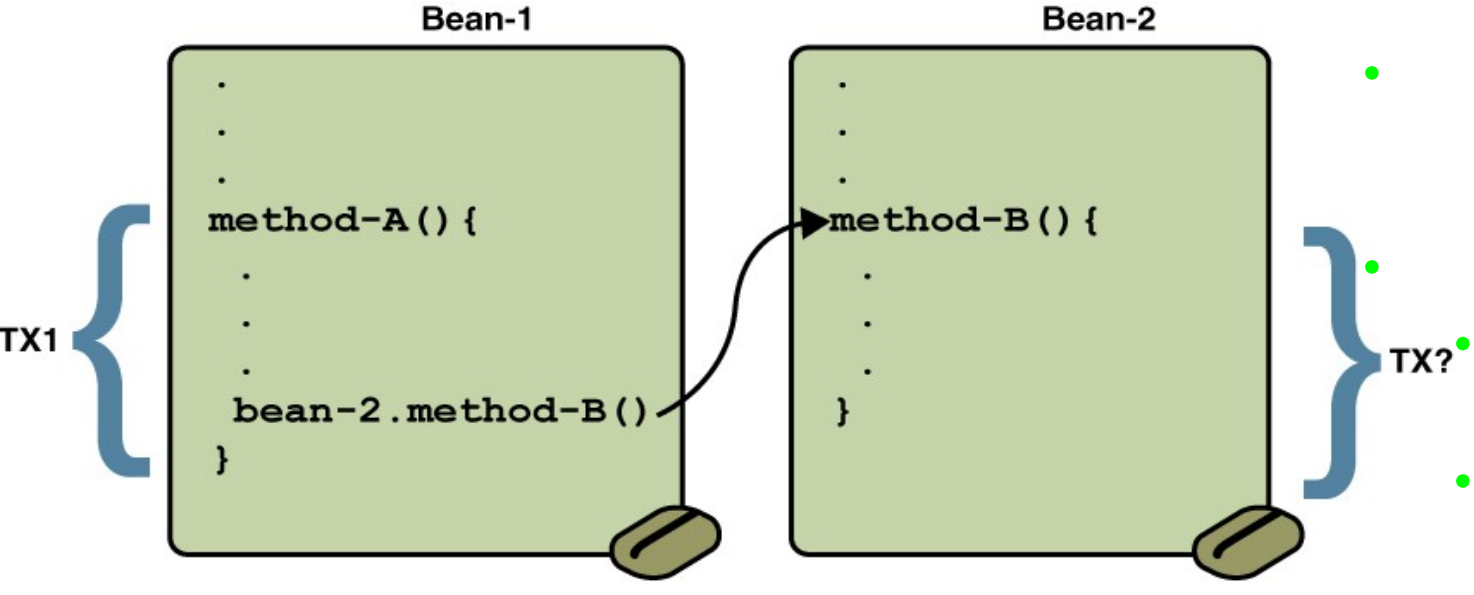
Components relying on container-managed transaction scoping must not use

- any transaction management methods that interfere with the container's transaction demarcation boundaries. Examples of such methods are
  - commit
  - setAutoCommit
  - rollback

  in the java.sql.Connection interface
- the javax.transaction.UserTransaction interface

In container managed demarcation, the scope of a transaction is controlled through transaction attributes.

Transaction attributes

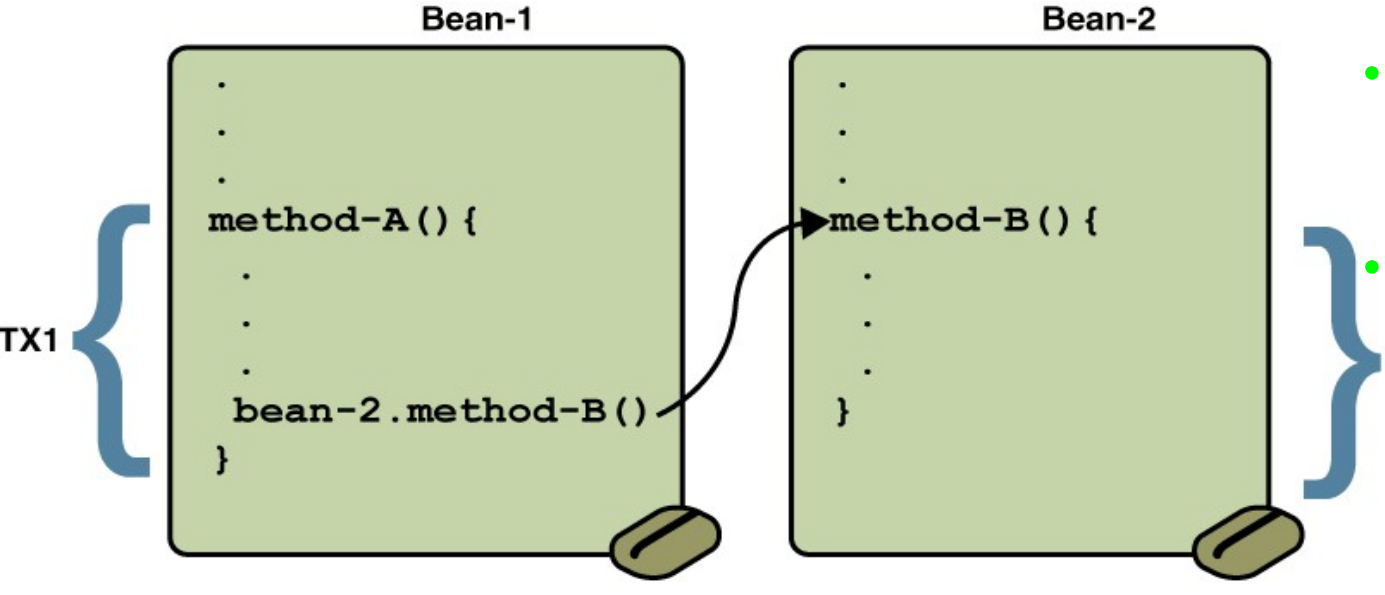


(Borrowed from [JBC+07])

- method−A() begins a transaction and then invokes method-B of Bean-2.
- when method−B() executes

should it belong to the scope of the transaction? should it start a new transaction?

Transaction attributes



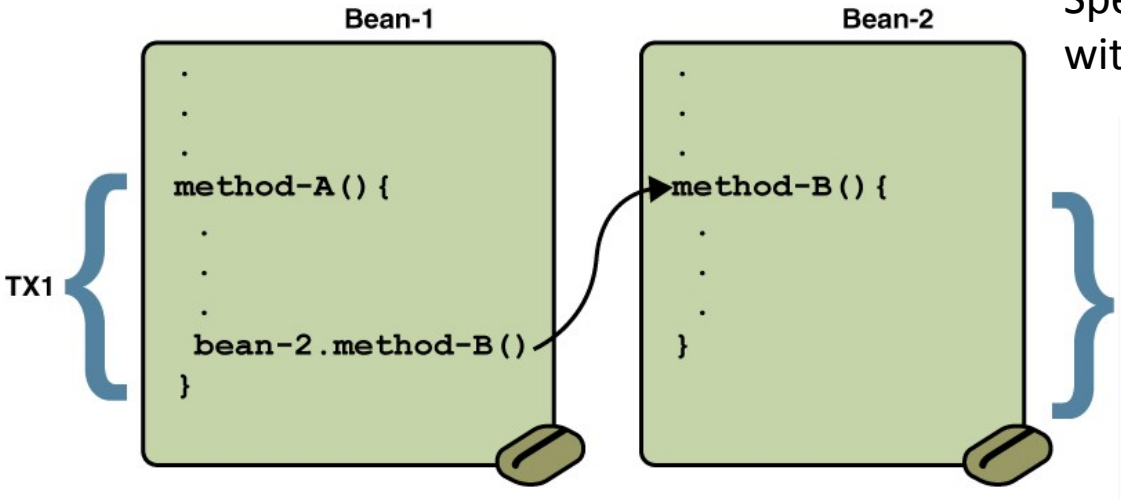
(Borrowed from [JBC+07])

- method−A() begins a transaction and then invokes method-B of Bean-2.
- when method−B() executes
- should it belong to the scope of the transaction?
- should it start a new transaction?

What happens depends on the transaction attribute of method−B(). A transaction attribute can have one of the following values:

- Required
- RequiresNew
- Mandatory
- NotSupported
- Supports
- Never

Semantics of transaction attributes



(Borrowed from [JBC+ 07])

Specify the transaction attribute of methodB with EJB annotation

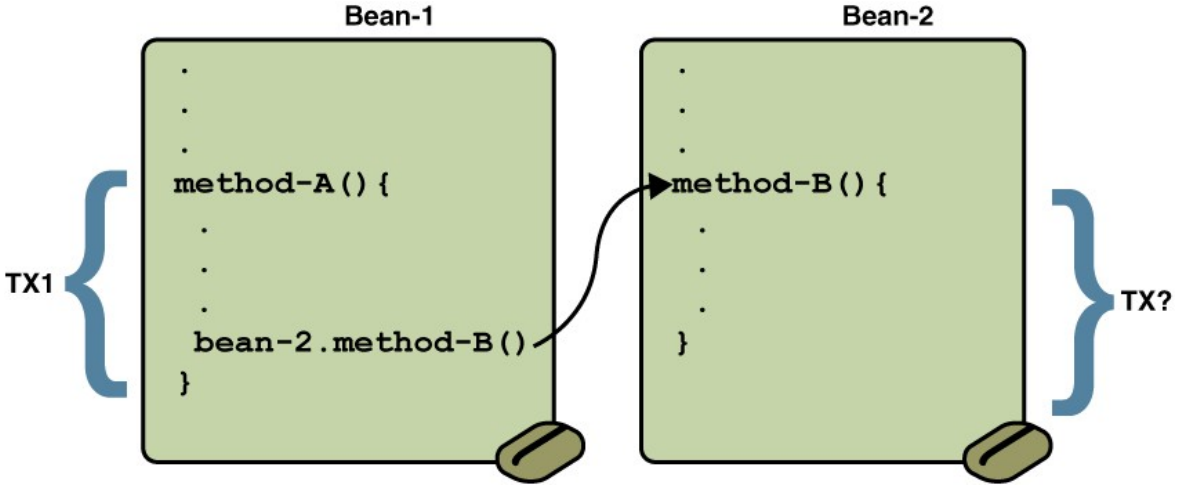
```
package CO3090.beantest;

import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;

    @Stateless
    public class Userbean {

        @TransactionAttribute(TransactionAttributeType.REQUIRED)
        public String methodB(String s) {
        return s;
        }

    }
```

Required: If method−A()

- is running within a transaction, then method−B() executes within the same transaction
- is not running in any transaction, the container of method−B() starts a new transaction before executing it

It is the default value of transaction attributes.

Semantics of transaction attributes



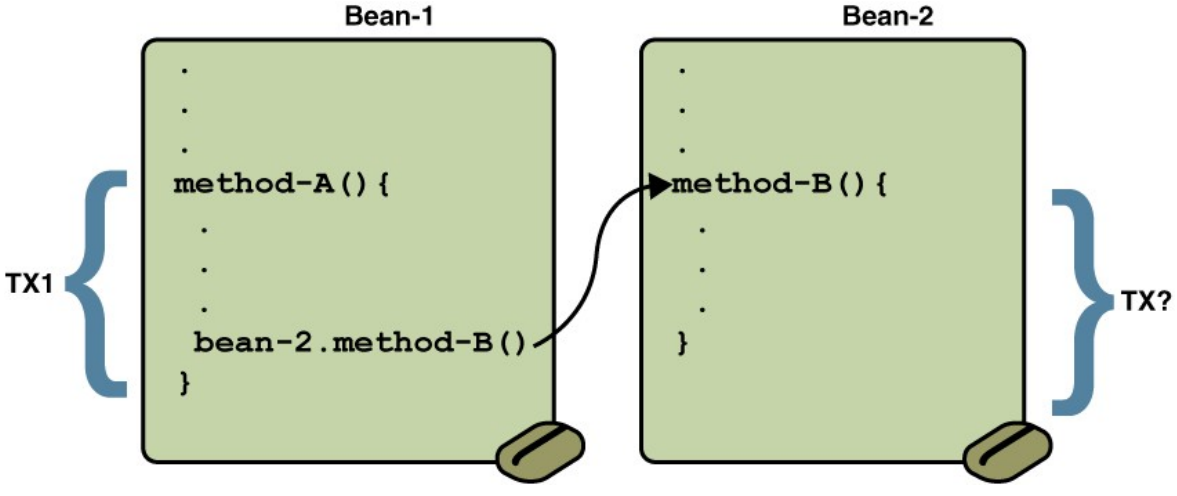
(Borrowed from [JBC+07])

RequiresNew: If method−A()

- runs within a transaction, the container:
  - Suspends the client's transaction
  - Starts a new transaction
  - Delegates the call to the method
  - Resumes the client's transaction after the method completes
- is not running in any transaction, the container of method−B() starts a new transaction before executing it

To be used if method−B() must execute in a transaction different from method−A()'s one (if any).

Semantics of transaction attributes



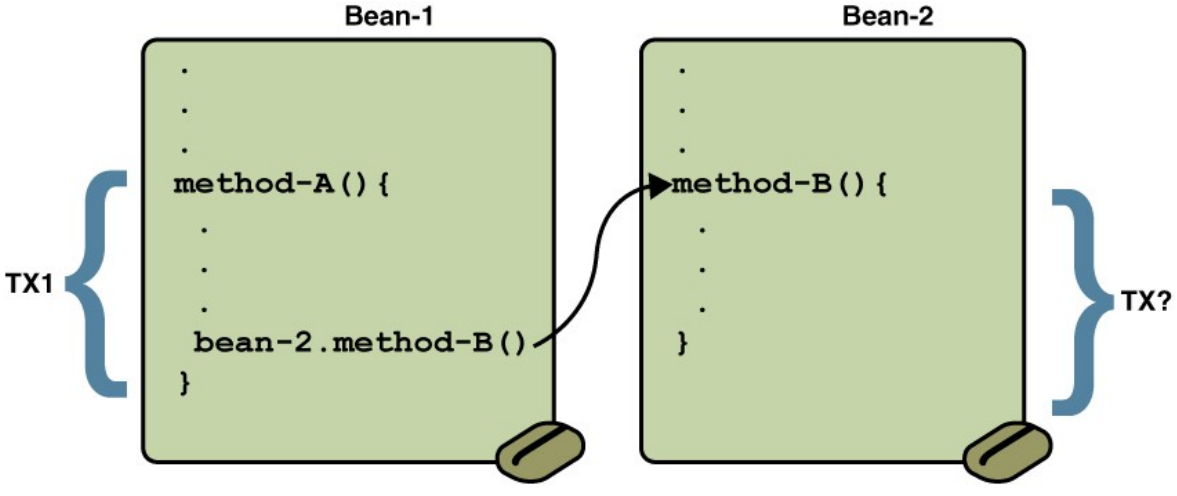
(Borrowed from [JBC+07])

Mandatory: If method−A()

- is running within a transaction, method−B() is executed in the same transaction
- is not running in any transaction, the container throws the TransactionRequiredException

To be used when method−B() must execute in the same transaction of the invoker.

Semantics of transaction attributes



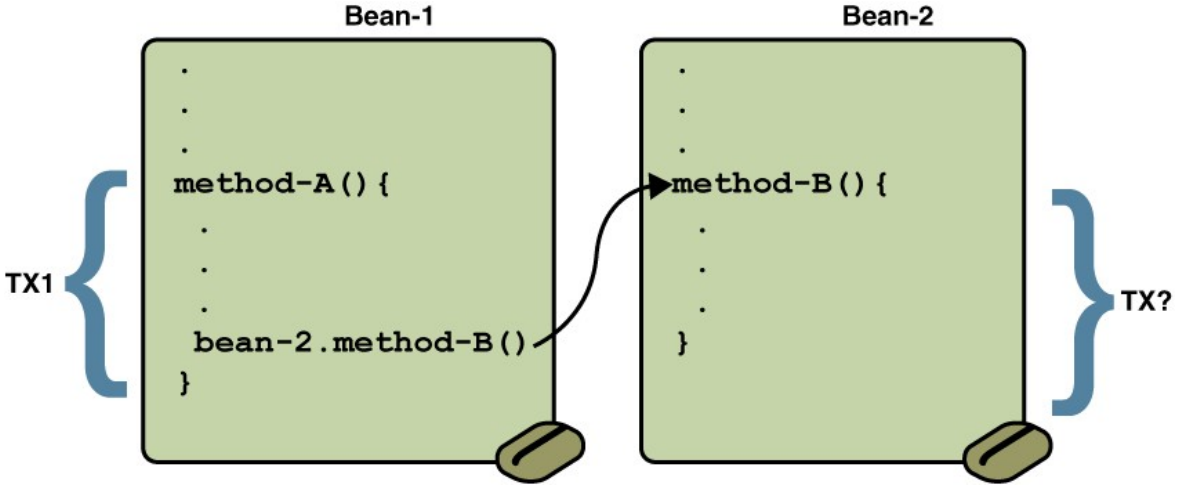
(Borrowed from [JBC+ 07])

NotSupported: If method−A()

- is running within a transaction, the container suspends the transaction of method−A() before invoking method−B(). When method−B() has finished, the container resumes the transaction of method−A()
- is not running in any transaction, the container of method−B() does not start a new transaction before executing it

Transactions have overheads: to be used to improve performance.

Semantics of transaction attributes



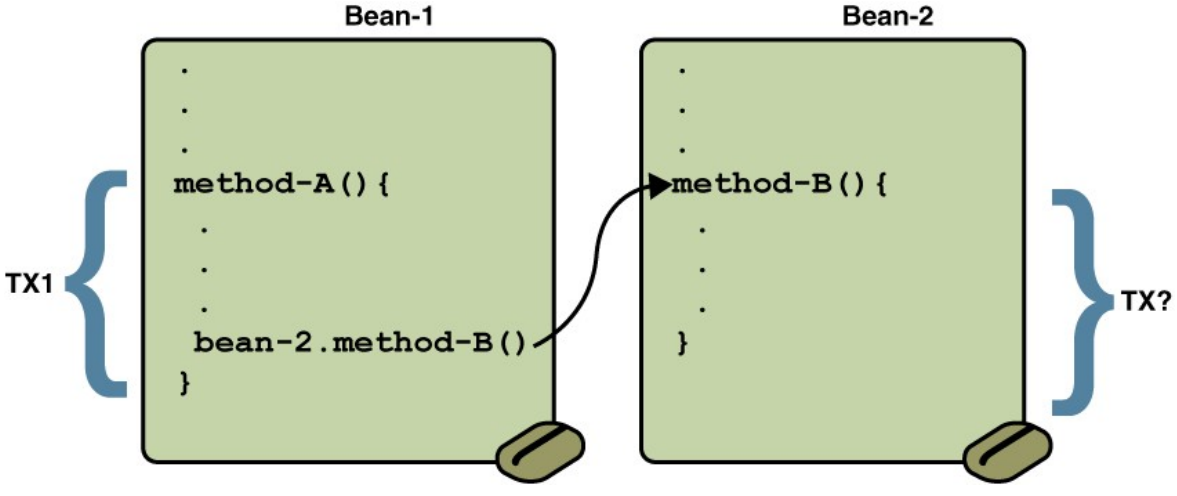
(Borrowed from [JBC+ 07])

Supports: If method−A()

- is running within a transaction method−B() is executed in the transaction of method−A()
- is not running in any transaction, the container of method−B() does not start a new transaction before executing it

To be used when method−B() is to be executed in a transaction whenever method−A() is.

Semantics of transaction attributes



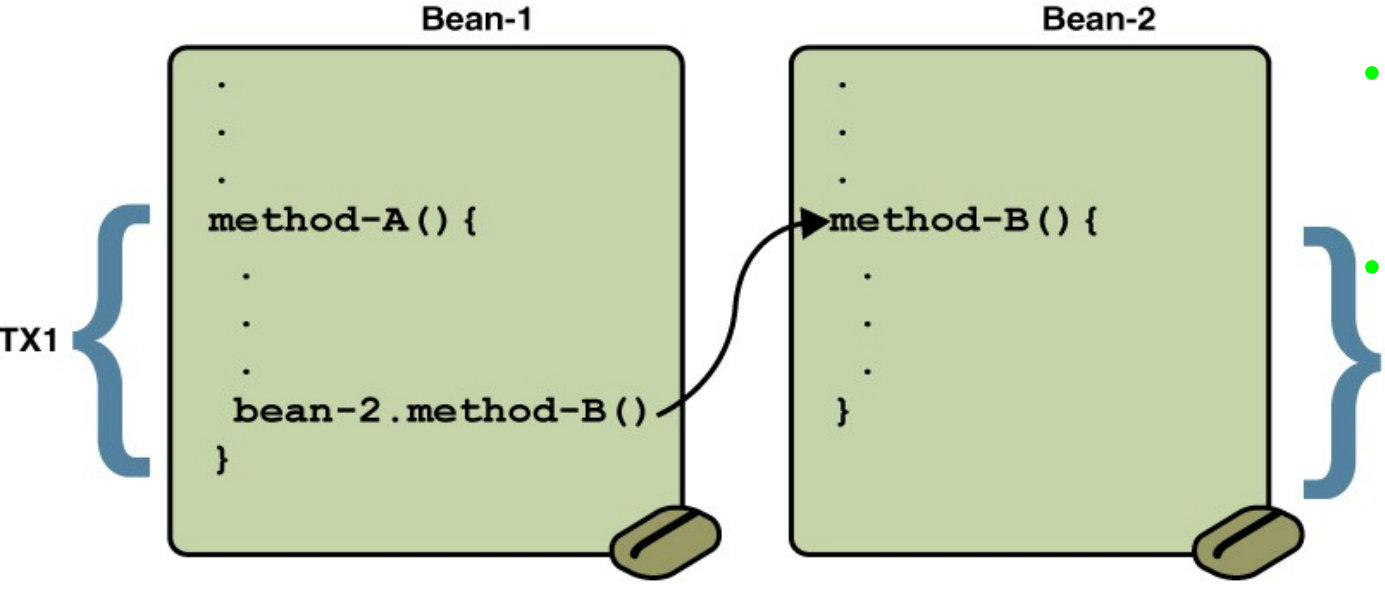
(Borrowed from [JBC+ 07])

Never: If method−A()

- is running within a transaction, the container of method−B() throws a RemoteException
- is not running within a transaction, the container of method−B() does not start a new transaction before running it

To be used when method−B() must never be involved in a running transaction.

Transaction attributes



(Borrowed from [JBC+07])

- method−A() begins a transaction and then invokes method-B of Bean-2.
- when method−B() executes
- should it belong to the scope of the transaction?
- should it start a new transaction?

What happens depends on the transaction attribute of method−B(). A transaction attribute can have one of the following values:

- Required
- RequiresNew
- Mandatory
- NotSupported
- Supports
- Never

Bean-Managed Transactions

The scope of the transaction is explicitly written in the code:

```
begin transaction
    //... do something ...
    if  (condition−x){
        // ... do something else ...
        commit transaction
    }
    else{
        rollback transaction
        begin transaction
        //...
        commit transaction
    }
end transaction
```

To set the scope of a JTA transaction methods Programmers uses the following interface javax.transaction.UserTransaction interface

- begin
- commit
- rollback

The scope of the transaction does not depend on the transaction manager

Java-Transation API (JTA)

```java
@Stateless
@TransactionManagement( TransactionManagementType.BEAN )
public class JTABean
{
    @Resource
    private EJBContext context;

    public void trxnMethod() throws Exception
    {
        UserTransaction utx = context.getUserTransaction();

        try
        {
            utx.begin();
            /**
              * If any error happens before reaching
              * commit, the rollback() method
              * is waiting below at the catch{} clause,
              * before hitting on the commit() method.
             **/
            utx.commit();
        }
        catch(Throwable t)
        {
            //Invokes only if the above throws and
            //Exception or instance of Throwable
            utx.rollback();

            throw new Exception(t.getMessage());
        }
    }
}
```

UserTransaction provides an interface to the transaction manager that allows the application developer to manage the scope of a transaction explicitly.

To begin a transaction, call the **begin** method. When all the entity operations are complete, call the **commit** method to commit the transaction. The **rollback** method is used to roll back the current transaction.

How to obtain a UserTranscation object

(1) Using JNDI

```
…..

Properties props = new Properties();
    props.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.sun.enterprise.naming.SerialInitContextFactory");
    props.setProperty("org.omg.CORBA.ORBInitialHost",
"localhost");
    props.setProperty("org.omg.CORBA.ORBInitialPort", "3700");

Context jndiCntx = new InitialContext();
UserTransaction utx=
(UserTransaction)jndiCntx.lookup("java:comp/UserTransaction");

 utx.begin();
...
utx.commit();
....
```

How to obtain a UserTranscation object

(2) Using field injection in bean class

```
@Resource
UserTransaction ut;
```

Some EJB injection examples:

```
@EJB
private Calculator calculator;

@Resource
UserTransaction ut;

@Resource
(name="jdbc/
MyAppServerDataSourceJNDIName") DataSource
ds;
```

*"EJB 3 specification allows you to use annotations to inject dependencies through annotations on fields or setter methods. … you can use the @EJB and @Resource annotations to set the value of a field or to call a setter method within your session bean with anything registered within JNDI. You can use the **@EJB annotation** to inject EJB references and **@Resource** to access datasources." -- JBoss*

So far...

- JTA
- Scope of transactions in JAVA
- Further reading [JBC+07] chapt. 33

What is messaging?

Message Oriented Middleware (MOM) offers support for (time) decoupled asynchronous messaging

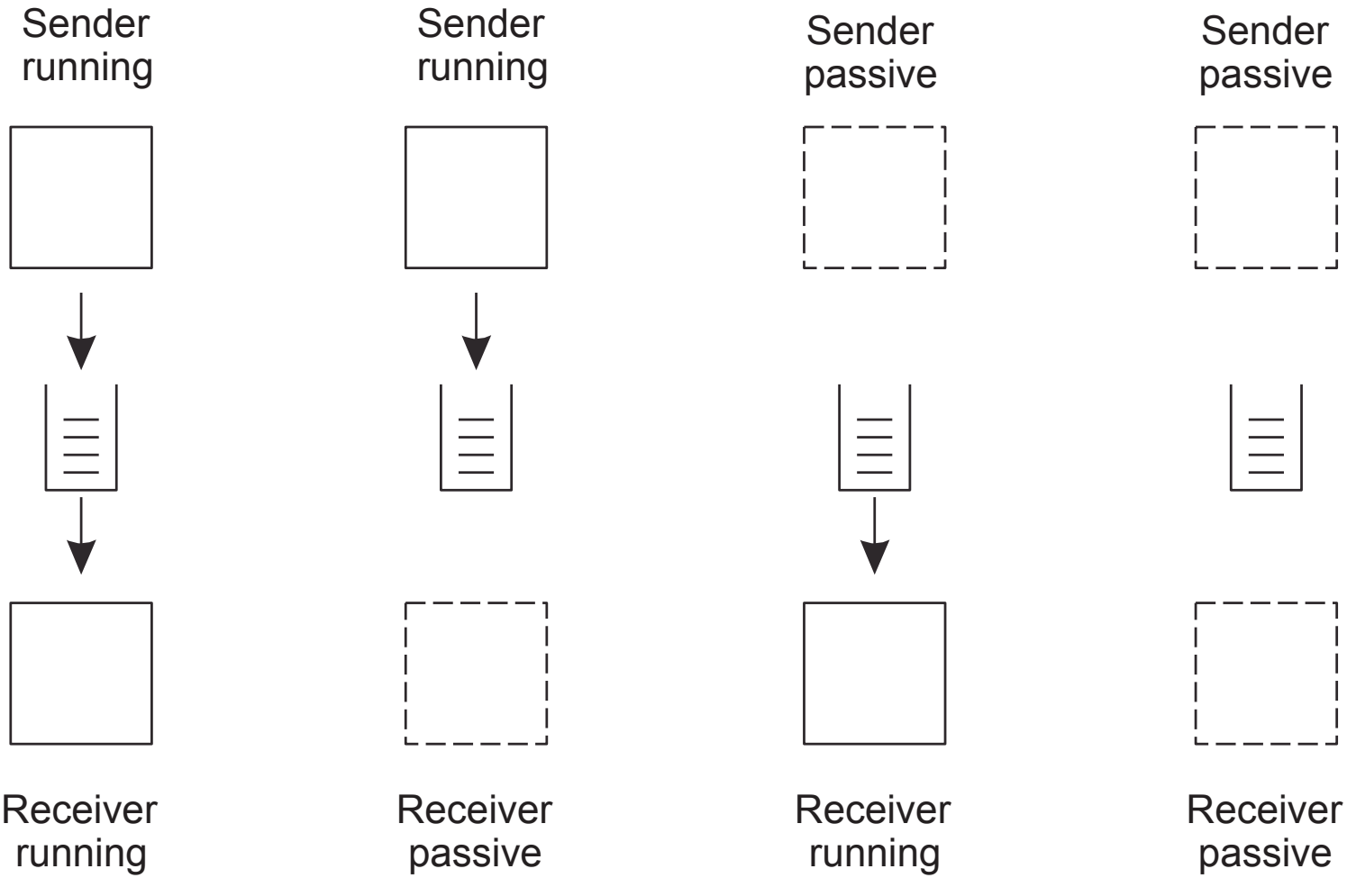
- model of communication among components (or applications)
- based on message queues and peer-to-peer facility:
  - a component can send/receive messages to/from other components by using (specific) queues
  - each client connects to a messaging agent (server) that provides facilities for creating, sending/receiving and reading messages
- enables loosely coupled distributed communication:
- sender and receiver need to know only format of messages and their destinations; they do not
  - need to know anything about each other
  - have to be available at the same time in order to communicate: messages are sent/retrieved to/from some destination

What is messaging?

Message Oriented Middleware (MOM) offers support for (time) decoupled asynchronous messaging

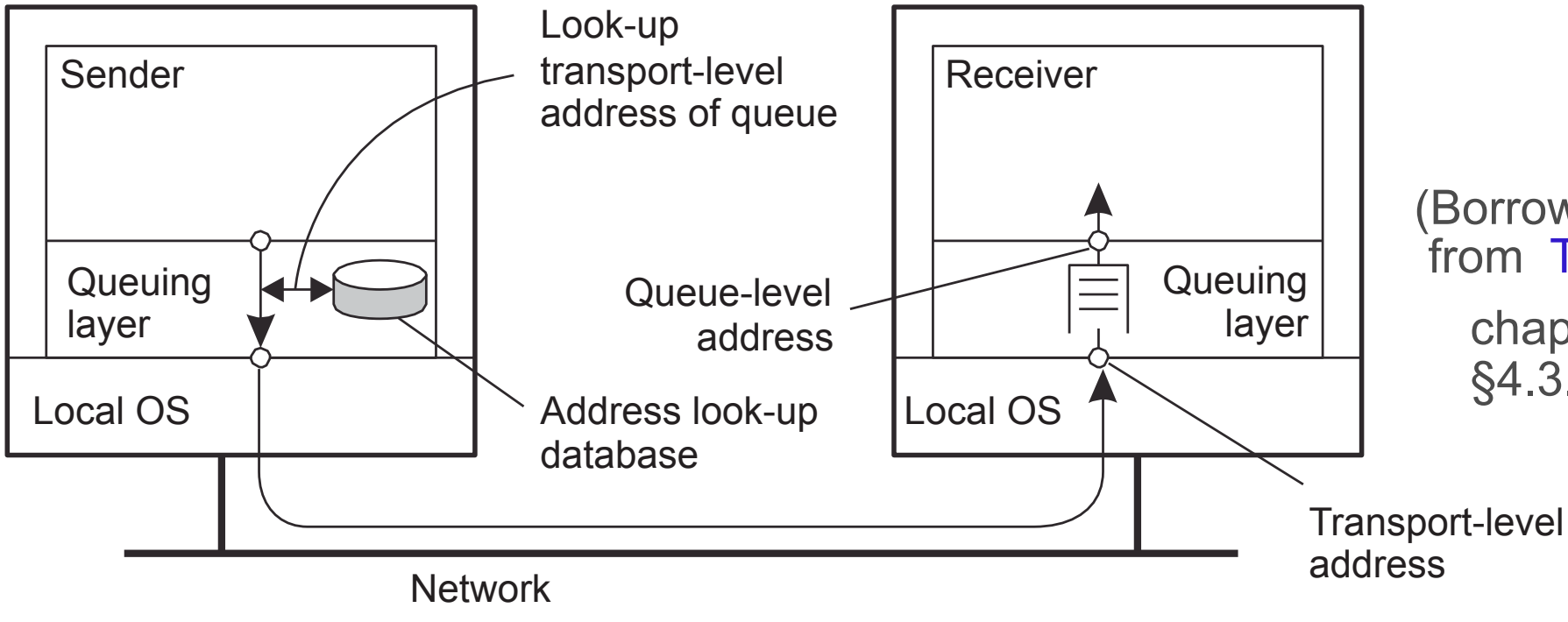
- model of communication among components (or applications)
- based on message queues and peer-to-peer facility:
  - a component can send/receive messages to/from other components by using (specific) queues
  - each client connects to a messaging agent (server) that provides facilities for creating, sending/receiving and reading messages
- enables loosely coupled distributed communication:
- sender and receiver need to know only format of messages and their destinations; they do not
  - need to know anything about each other
  - have to be available at the same time in order to communicate: messages are sent/retrieved to/from some destination

Decoupled communications in MOM



(Borrowed from [TV06] chap. 4, §4.3.2;
do not confuse this terminology/graphical notation with the one used for concurrent application)

Architecture of (point-to-point) MOMs

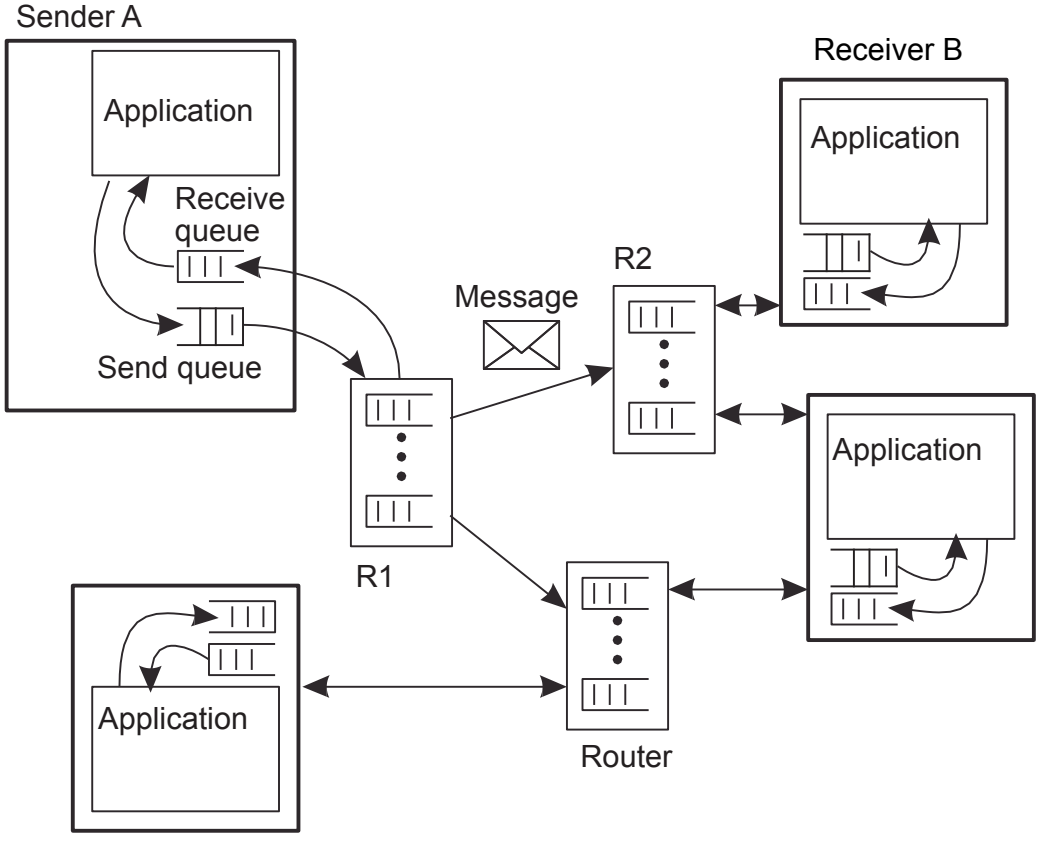


(Borrowed from TV06 chap. 4, §4.3.2)

- sender and receiver only access their local queue messages
- in local queues mention their destination queue
- local queues are transparent to sender/receiver
- queues are distributed and MOMs have to maintain DB of queue names (i.e., MOMs provide a naming service)

MOMs and “routers”

The most sophisticated MOMs can be envisaged as overlay networks

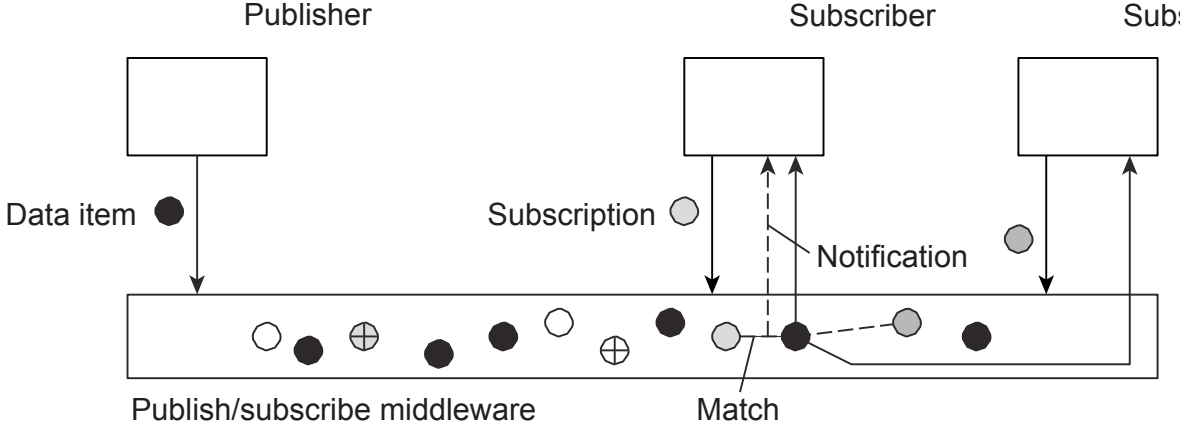


- “A” puts a message for “B” in its local queue
- “R1” gets the message
  - decide its “direction” according to “B” ’s name
  - forwards the message to “R2”
- only routers have to be modified if the topology changes
- marshalling/unmarshalling might be necessary

(Adapted from [TV06] chap. 4, §4.3.2)

Publish/Subscribe MOMs

In Publish/Subscribe MOMs applications "publish" messages on topics of interest and retrieve messages after "subscribing" to their associated topic



(Adapted from [TV06] chap. 4, §4.3.2)

- Generative communication [Gel8
  - data described by a set of attributes
  - subscription specify pairs `(attribute, range)`

- when matching between subscription and data is successful
  either data are sent to subscribers
  or a notification is sent and subscribers have to retrieve data with an explicit "read" operation
- in the former case data are typically removed
- in the latter case data are stored until all the notified subscribers read them (with possible usage of leases attached to data)

So far...

MOMs principles and architecture

- Point-to-Point
- Publish/Subscribe

JAVA Message Service

JAVA Message Service (JMS) API allows messages to be

- Created
- sent/received
- read

JMS yields a communication models which is

- reliable
- asynchronous
- loosely coupled

What is the JMS API?

JAVA Message Service is a JAVA API that

- allows applications to create, send, receive, and read messages
- designed by Sun (and several partner companies)
- specifies interfaces for JAVA MOMs
- besides loosely coupled communication, JMS enables

  Asynchronous: messages are delivered on arrival (clients do not explicitly have to request messages)

  Reliable: messages are delivered “once and only once”

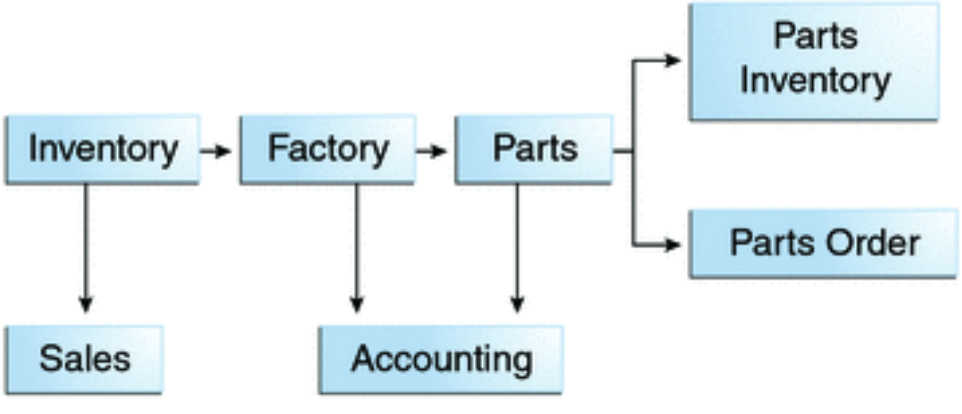
When it is worth using JMS?

Typical situations are when

- components do not depend on (interfaces of) other components
- a component should run also if other components are not running
- communication is mostly asynchronous

Example 3 ([JBC+07] chapt. 31) An application made of several components

an automobile manufacturer



- `Inventory` informs `Factory` when the level of a product is below a given level
- `Factory` sends `Parts` so that `Factory` can assemble the parts it needs
- `Parts` can send messages to its own inventory and make orders from suppliers
- `Factory` informs `Inventory` when level has been re-established
- `Factory` and `Parts` can send messages to `Accounting` to update their budget numbers

JMS features

JMS supports

- asynchronous communication among
  - application clients
  - EJB components
  - web components
- concurrent processing of messages by message-driven beans
- sending/receiving of messages involved in distributed transactions
- interfacing messages in legacy systems
- dynamic integration of message-driven bean with running applications

JMS API Architecture

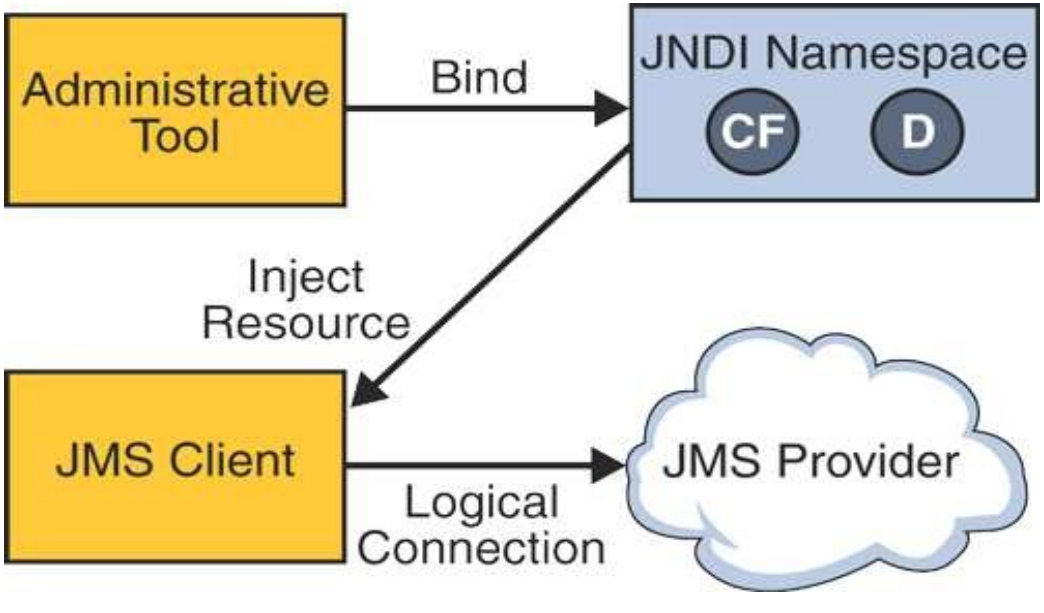
JMS applications consist of:

JMS provider: messaging system implementing the JMS interfaces (its a JAVA MOM)

JMS clients: components that produce/consume messages

Messages: objects conveying information among JMS clients

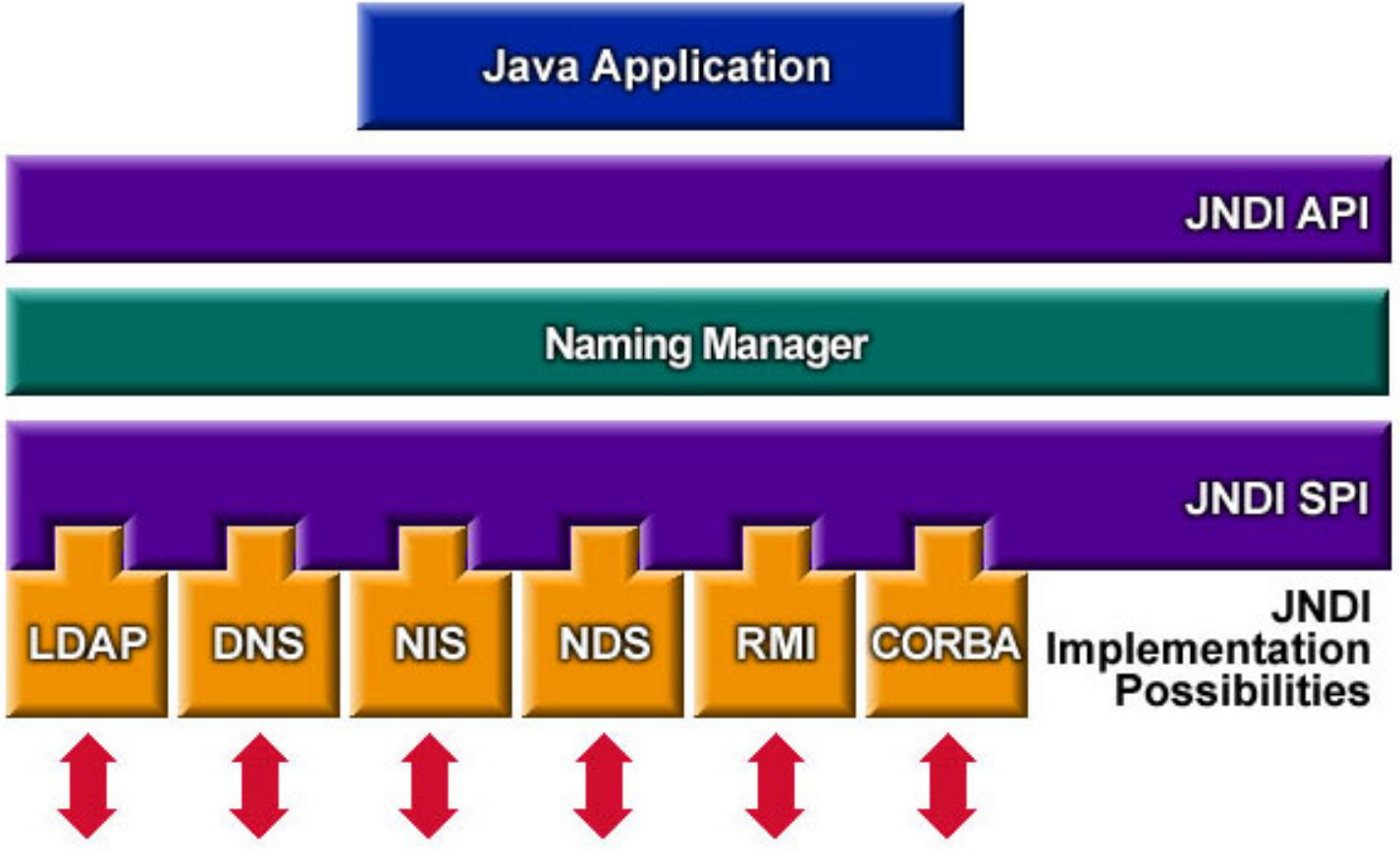
Administered objects: pre-configured JMS objects (created by an administrator for clients); they are distinguished in **destinations** and **connection factories**



- administrative tools bind destinations and connection factories into a JNDI namespace
- JMS clients can then access the administered objects in the JNDI namespace via "resource injection" (similar to RMI remote referencing)
- JMS clients then establish a logical connection to the same objects through the JMS provider

JNDI

The Java Naming and Directory Interface(**JNDI**) is an application programming interface (API) that provides naming and directory functionality to applications written using the JavaTM programming language. It is defined to be independent of any specific directory service implementation. Thus a variety of directories--new, emerging, and already deployed--can be accessed in a common way.



SPI:
the Service Provider Interface

JMS API Architecture

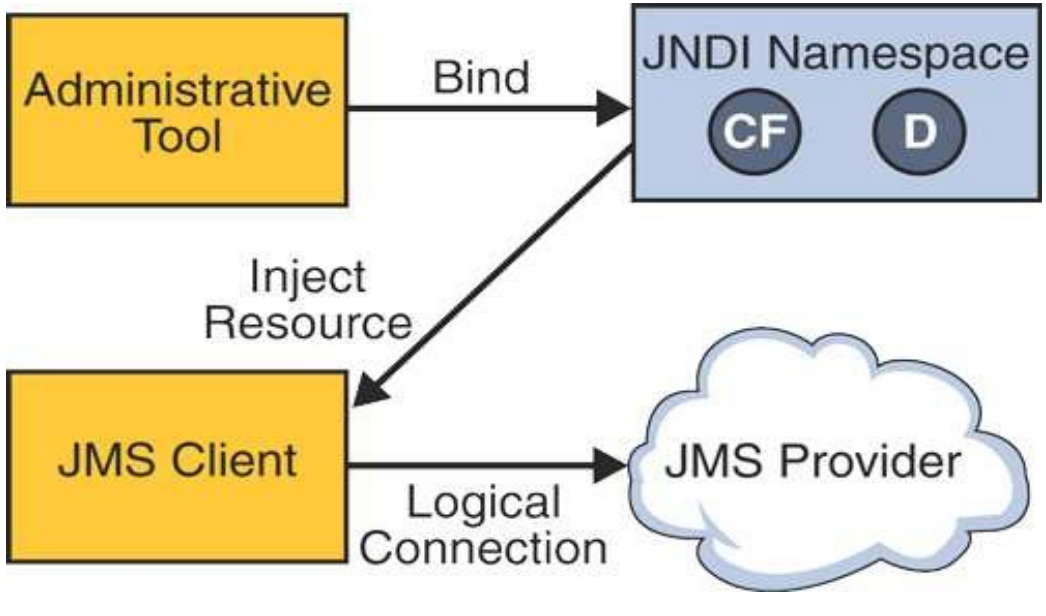
JMS applications consist of:

JMS provider: messaging system implementing the JMS interfaces (its a JAVA MOM)

JMS clients: components that produce/consume messages

Messages: objects conveying information among JMS clients

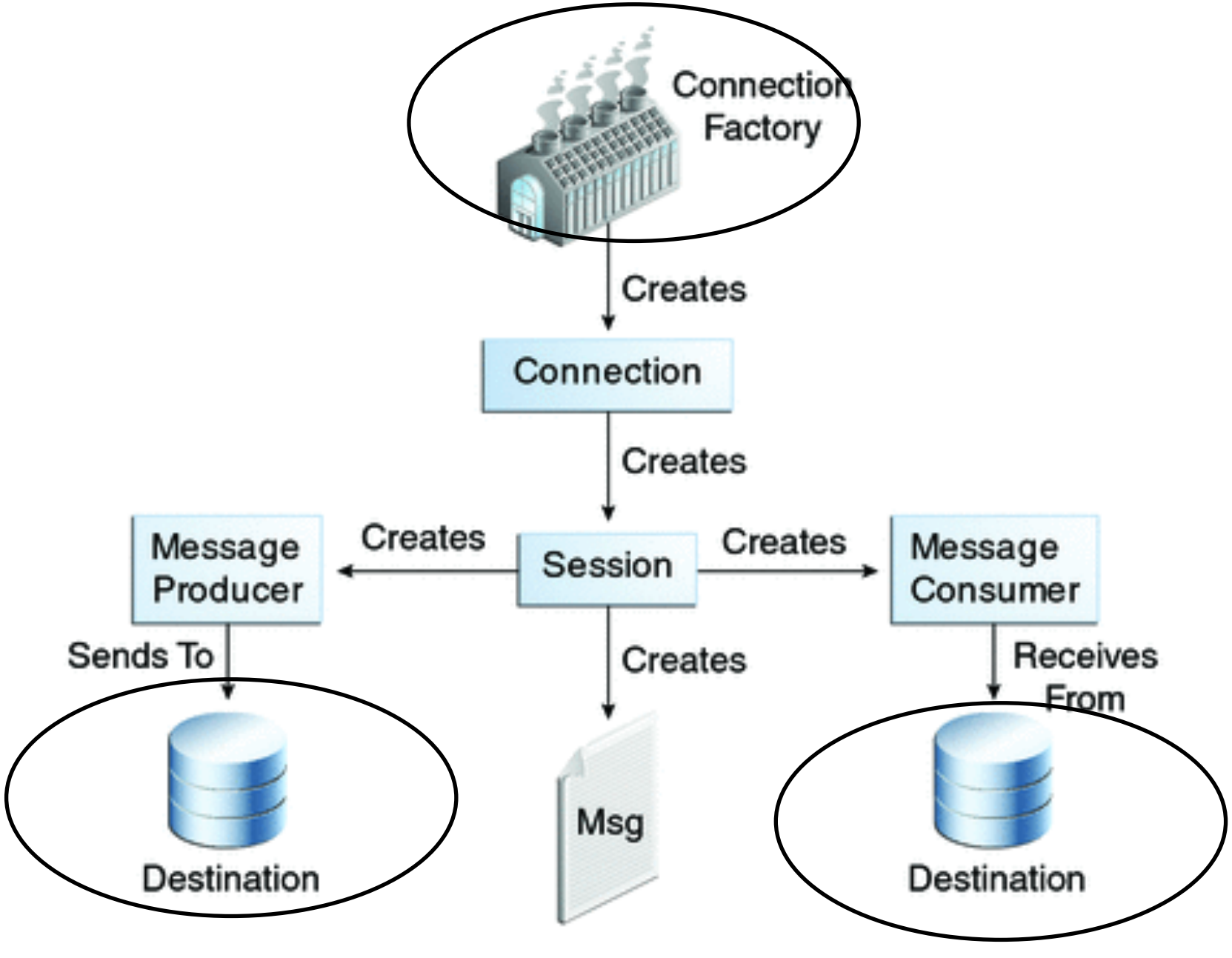
Administered objects: pre-configured JMS objects (created by an administrator for clients); they are distinguished in destinations and connection factories



- administrative tools bind destinations and connection factories into a JNDI namespace
- JMS clients can then access the administered objects in the JNDI namespace via "resource injection" (similar to RMI remote referencing)
- JMS clients then establish a logical connection to the same objects through the JMS provider

```
@Resource(mappedName="jms/Queue")
private static Queue queue;
```

The JMS Programming Model



(Borrowed from [JBC+ 07])

The JMS Programming Model: connection factories and destinations

Connection Factories and Destinations are administered objects (depend on providers):

Connection Factories: objects that are

- used by clients to connect to a JMS provider
- instance of one of the following interfaces:
  - ConnectionFactory
  - QueueConnectionFactory
  - TopicConnectionFactory
- a JMS client program usually injects a connection factory resource into a ConnectionFactory object:

```
@Resource(mappedName="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
```

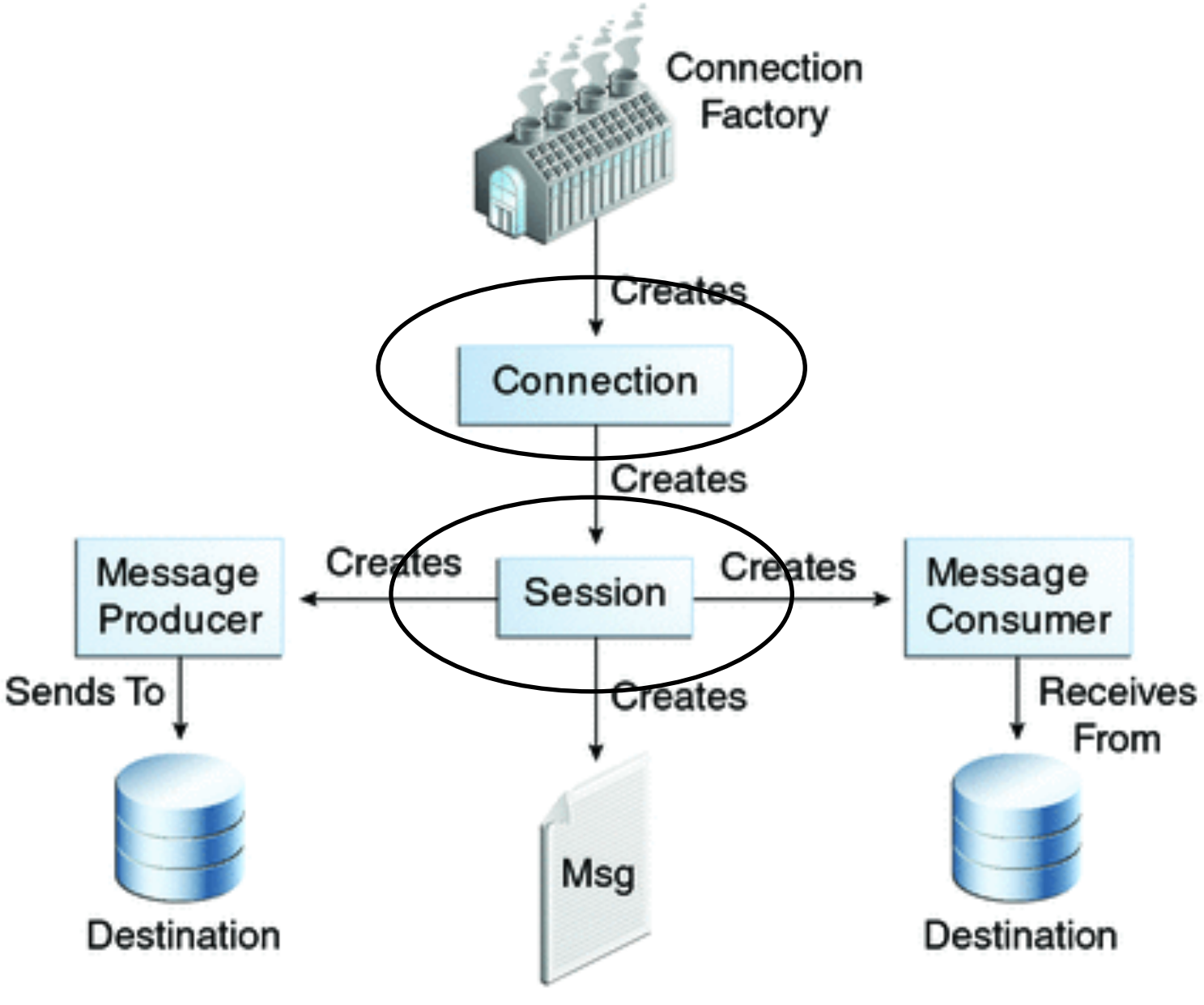
specifies a resource registered as ConnectionFactory and assigns it to the ConnectionFactory object connectionFactory

Destinations: objects used by clients to specify

- the targets of messages produced
- the sources of messages consumed
- a JMS client program usually injects a connection

```
@Resource(mappedName="jms/Queue")
private static Queue queue;
```

The JMS Programming Model



(Borrowed from [JBC+ 07])

The JMS Programming Model: connections and sessions

Connections: encapsulate a (virtual) connections with JMS providers

- connections are created as follows:

```
Connection connection = connectionFactory.createConnection();
```

- must be closed (connection.close();) before application exits in order to release resources (connection.stop() can be used to suspend a connection without closing it)

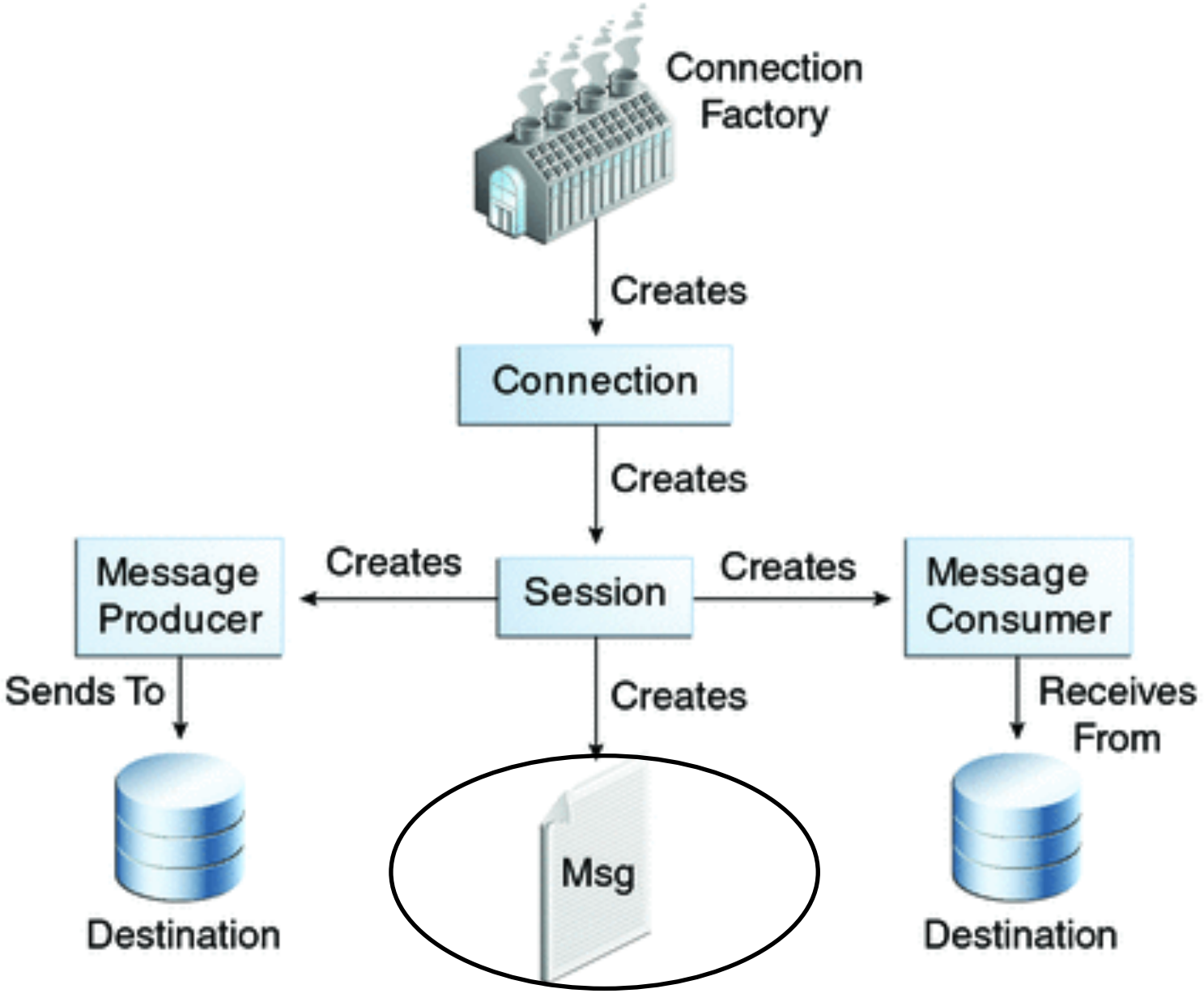
Sessions: single-threaded contexts for producing/consuming messages:

```
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

creates a non-transacted session.

- Sessions are used to create:
  - Message Producers/Consumers
  - Messages
  - Queue Browsers
  - Temporary queues and topics
- a session permits to (transactionally) group message exchanges into an atomic unit of work
- implement the Session interface

The JMS Programming Model



(Borrowed from [JBC+ 07])

The JMS Programming Model: JMS communications

- Messages: consist of three parts

  header (required): used by client and provider to identify and route messages

  properties (optional): used to specify further fields not in the header

  body (optional): specify the format of the message

- Message Producer/Consumer: objects created by sessions that implement the MessageProducer and MessageConsumer interfaces, respectively

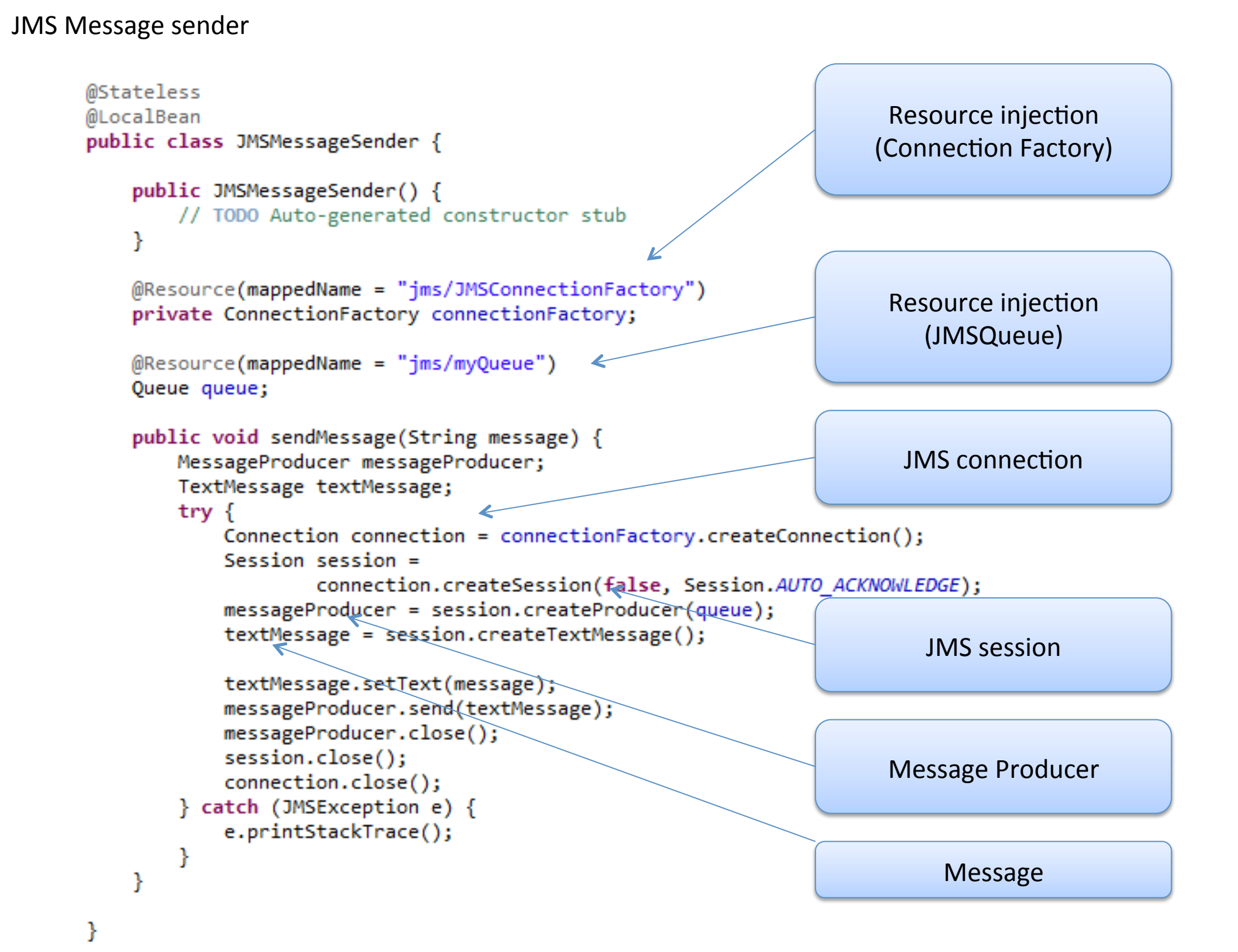
```
MessageConsumer consumer = session.createConsumer(topic);

Message m = consumer.receive();
m.getText("...");
MessageProducer producer = session.createProducer(topic);
 producer.send(m);
```

- Queue Browsers: objects that inspect the messages in a queue

```
QueueBrowser browser = session.createBrowser(queue);
```

JMS Message sender

```java
@Stateless
@LocalBean
public class JMSMessageSender {

    public JMSMessageSender() {
        // TODO Auto-generated constructor stub
    }

    @Resource(mappedName = "jms/JMSConnectionFactory")
    private ConnectionFactory connectionFactory;

    @Resource(mappedName = "jms/myQueue")
    Queue queue;

    public void sendMessage(String message) {
        MessageProducer messageProducer;
        TextMessage textMessage;
        try {
            Connection connection = connectionFactory.createConnection();
            Session session =
                    connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            messageProducer = session.createProducer(queue);
            textMessage = session.createTextMessage();

            textMessage.setText(message);
            messageProducer.send(textMessage);
            messageProducer.close();
            session.close();
            connection.close();
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }

}
```



JMS Message Receiver

```java
@Stateless
public class MessageReceiverSync {

    @Resource(mappedName = "jms/JMSConnectionFactory")
    private ConnectionFactory connectionFactory;

    @Resource(mappedName = "jms/myQueue")
    Queue myQueue;

    private String message;

    public String receiveMessage() {
        try {
            Connection connection = connectionFactory.createConnection();
            Session session =
                connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            QueueBrowser queueBrowser = session.createBrowser(myQueue);
            Enumeration enumeration = queueBrowser.getEnumeration();

            while (enumeration.hasMoreElements()) {
                TextMessage o = (TextMessage) enumeration.nextElement();
               return "Receive " + o.getText();

            }
            session.close();
            connection.close();
        } catch (JMSException e) {
            e.printStackTrace();
        }
        return "";
    }
}
```

Resource injection (Connection Factory)

Resource injection (JMSQueue)

JMS connection

JMS Queue browser

Message

Test Client (Java servelt)

```java
@WebServlet(urlPatterns = {"/TestServlet"})
public class TestServlet extends HttpServlet {

    @EJB JMSMessageSender sender;

    @EJB MessageReceiverSync receiver;

    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            String m = "Hello there";
            sender.sendMessage(m);
            out.format("Message sent: %1$s.", m);
            out.println("Receiving message...");
            String message = receiver.receiveMessage();
            out.println("Message rx: " + message);

        }
    }
```

Resource injection (Sender)

Resource injection (Receiver)

Send message

Receive message

Reliability mechanisms

The basic reliability mechanisms for message delivery are:

- Controls on message acknowledgment
- Message persistence: messages can be declared persistent, i.e., they must not be lost in case of failure of the producer
- Message priority: different messages can have different priority levels in order to control the order in which the messages are delivered
- Messages can expire: an expiration time for a message can be set so that it will not be delivered when becomes obsolete

More advanced reliability mechanisms are:

- Durable subscriptions: durable topic subscriptions allow reception of message published while the subscriber is not active.
- Local transactions: allow messages to be grouped into sequences of send/ receive to form an atomic unit of work.

So far...

- JAVA EE architecture
- Advantages of component based programming
- JMS
- Further reading [JBC+07] chapt. 31 and [RMHC02]

Remember:

- The architecture of JAVA EE
- How the components distribute over the tiers
- The principles of JMS