Business Tier

Benoît Garbinato
distributed object programming lab

Outline

- □ Separation of concerns
- □ Enterprise Java Beans
- □ Resource pooling
- □ Transactions
- □ Persistence
- □ Asynchronous Invocations

Facts

- Distributed enterprise applications have critical requirements, such as availability, reliability, security, scalability, etc.
- These requirements are orthogonal to the business domain, i.e., they can be found in almost any application
- To address these needs, software architects have usually to rely on an existing hardware & software infrastructure
- A flexible software architecture aims at achieving reuse of both application code and technical code

Problems (1)

- ❑ Heterogeneity: existing infrastructures are usually heterogeneous (different technologies, standards & products)
  - ➥ To solve this problem, we need a portable platform that encapsulates existing technologies, standards and products, e.g., Java & its Enterprise APIs (Java EE)

Problems (2)

- □ Skills Needs: software architects must be experts in all these technical domains, in addition to the business domain underlying the application they build
- □ Software engineering: achieving code reuse both at the technical and the business level is difficult when all concerns (business & technical) are tightly interwoven

Solutions: overview

- □ Skills Needs: we should define distinct roles in developing, assembling, deploying and managing enterprise applications
- □ Software engineering: we should be able to separate the various concerns (business & technical) in distinct reusable components

Software engineering

```
void transfer(  float money,
                Account source,
                Account destination,
                User user ) {
    check whether this user is allowed to perform the transfer   security
    begin transaction                                         consistency
    load source & destination accounts from database(s)       persistence

    withdraw money from source
                                                                 business
    credit money to destination

    store source & destination accounts to database(s)        persistence
    end transaction                                           consistency
}
```

Separation of concerns (1)

Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, [...] occupying oneself only with one of the aspects.

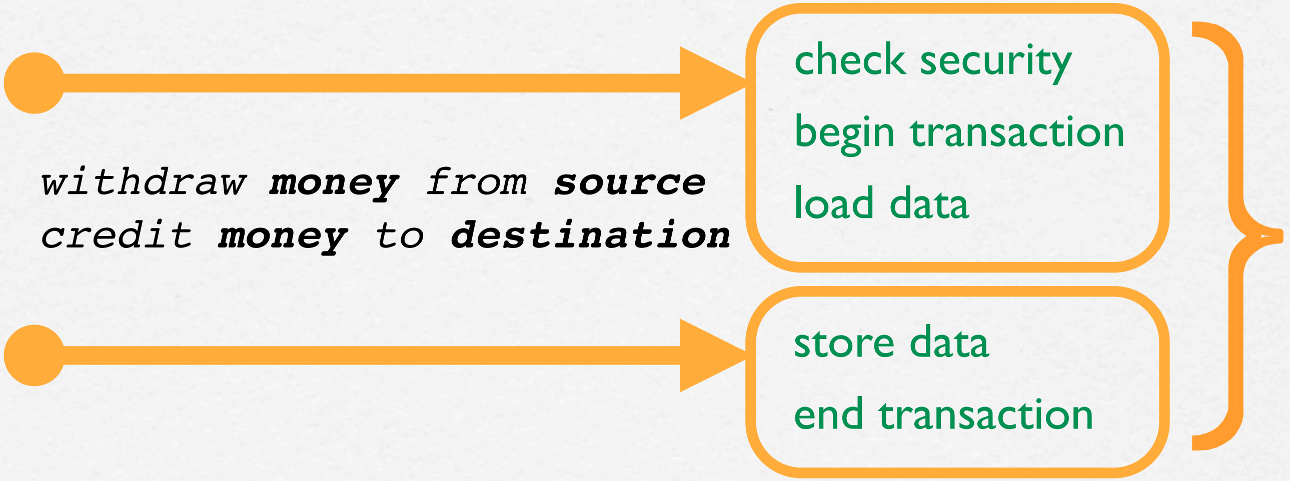
We know that a program must be correct and we can study it from that viewpoint only; we also know that is should be efficient and we can study its efficiency on another day [...] But nothing is gained – on the contrary – by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns" [...]

A scientific discipline separates a fraction of human knowledge from the rest: we have to do so, because, compared with what could be known, we have very, very small heads.

E.W. Dijkstra, On the role of scientific thought
EWD 477, 30th August 1974, Neuen, The Netherlands

Separation of concerns (2)

```
void transfer(float money, Account source, Account destination) {

    withdraw money from source
    credit money to destination

}
```



check security

begin transaction

load data

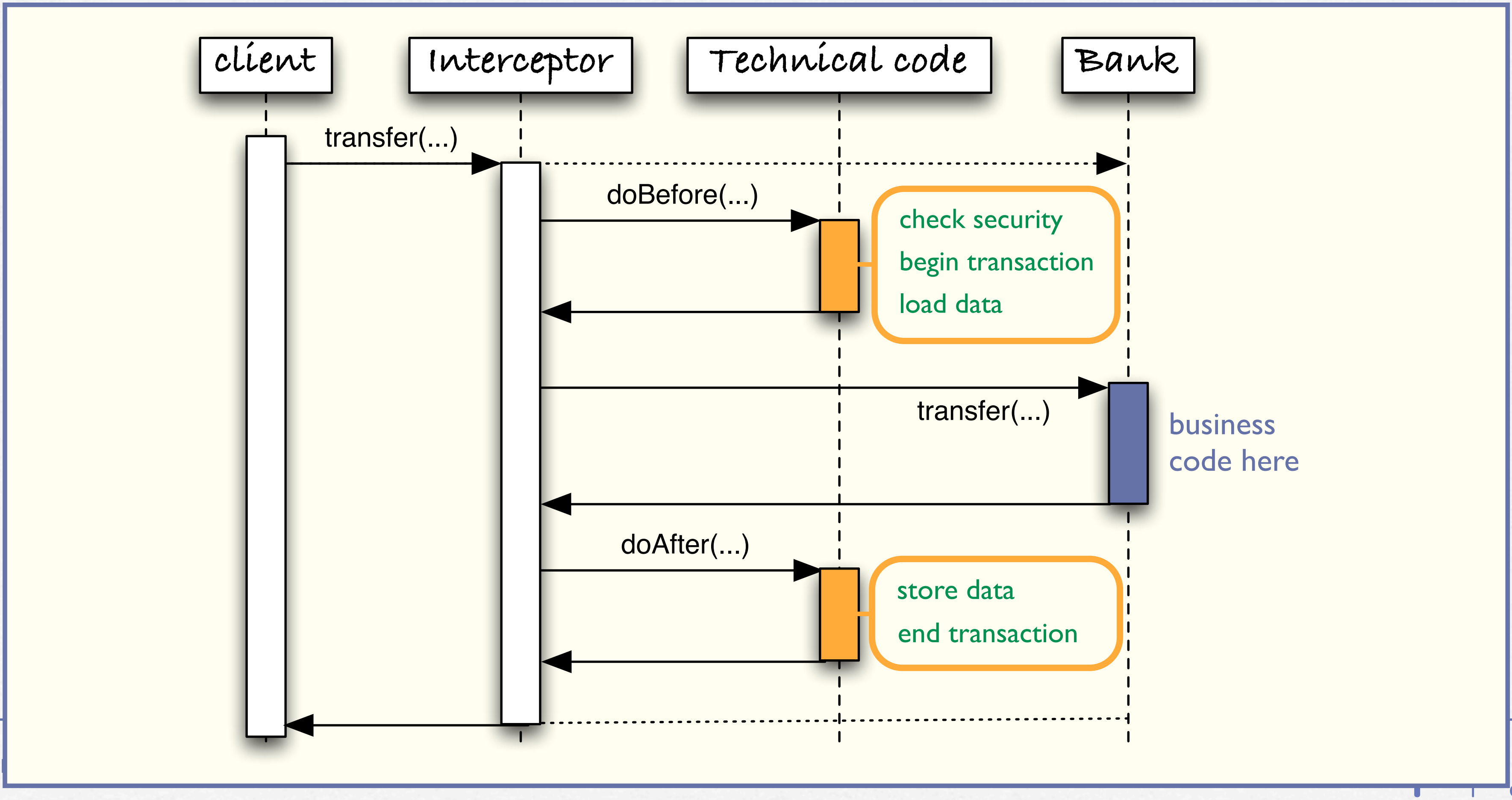
store data

end transaction

technical concerns should be separated from business concerns

Basic mechanism

□ All solutions to support separation of concerns are based on the same basic mechanism: automatic invocation interception



client

Interceptor

Technical code

Bank

transfer(...)

doBefore(...)

check security
begin transaction
load data

transfer(...)

business
code here

doAfter(...)

store data
end transaction

Separation of concerns: variants

- □ When does interception occur ?
  - □ At compile-time
  - □ At run-time
- □ How are technical concerns dealt with?
  - □ By coding/assembling technical objects
  - □ Declaratively, e.g., using deployment descriptors or annotations (metadata)

Examples

- □ AspectJ - Aspect-oriented programming
  - ➥ When? At compile-time.
  - ➥ How? By coding/assembling.
- □ GARF - Génération d'Applications Résistantes aux Fautes
  - ➥ When? At run-time.
  - ➥ How? By coding/assembling.
- □ EJB - Enterprise JavaBean
  - ➥ When? At compile-time.
  - ➥ How? Declaratively.

AspectJ

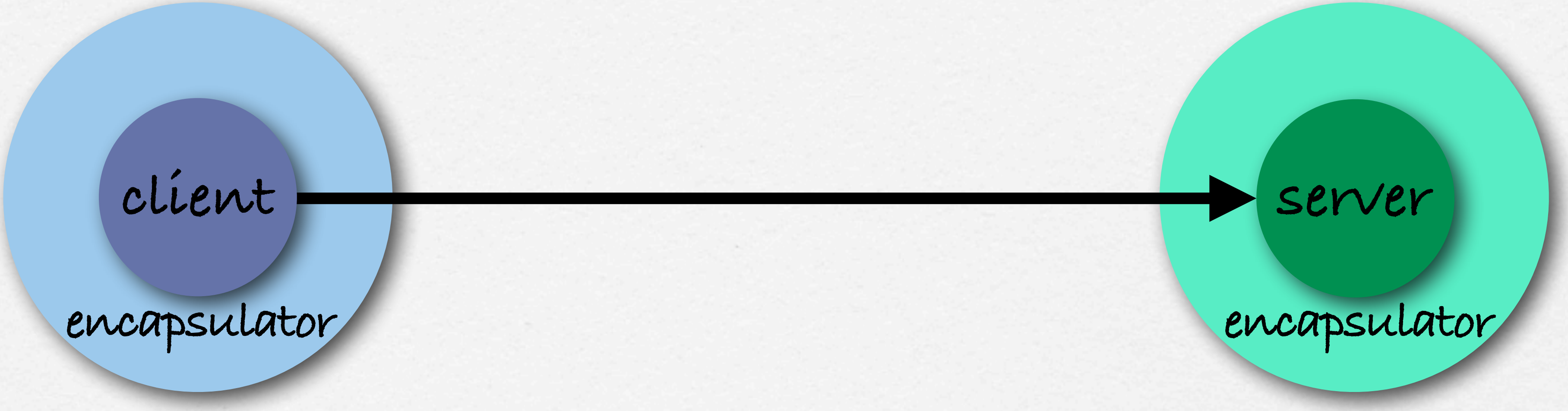
Assume we have some Bank class :

```
public class Bank {
    ...
    void transfer(float money, Account src, Account dest,User user ){ ... }
}
```

We add the technical code as follows :

```
aspect techCode
{   pointcut callTransfer() : call(void Bank.transfer(float, Account, Account, User));

    before() : callTransfer() {
        check security
        begin transaction
        load data
    }

    after() returning : callTransfer() {
        store data
        end transaction
    }
}
```

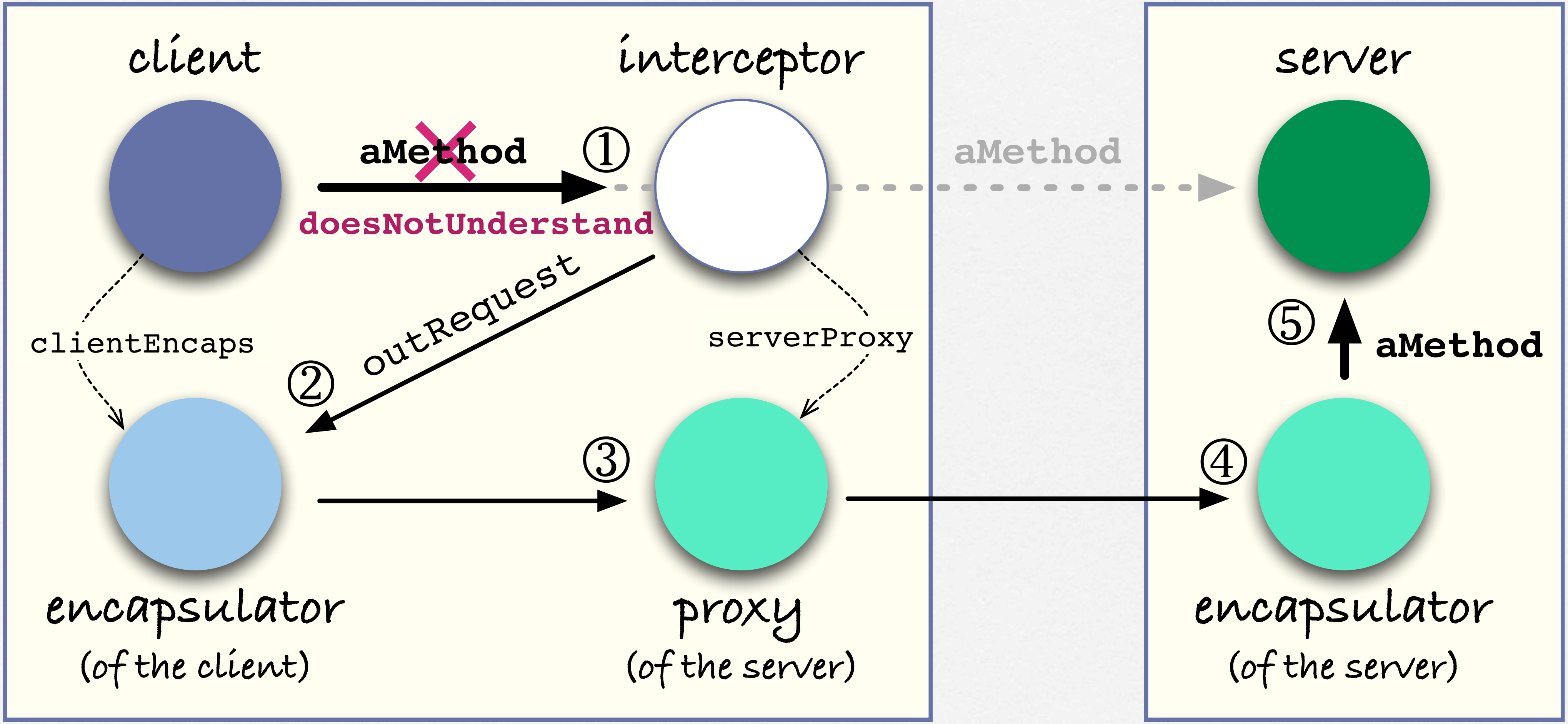
The GARF system (1)



client and server ⇔ component

encapsulator ⇔ container

The GARF system (2)



The GARF system (3)

The Interceptor class holds a reference to the serverProxy and redefines method doesNotUnderstand as follows:

```
doesNotUnderstand: aMethod
      |client clientEncaps|

      client ← currentStackFrame getCaller.
      clientEncaps ← client getEncapsulator.

      ↑clientEncaps outRequest:  aMethod
                    to:          serverProxy.
```

```
public void doesNotUnderstand(Method aMethod) {
        Object client; Encapsulator clientEncaps;

        client = currentStackFrame.getCaller();
        clientEncaps = client.getEncapsulator();

        return clientEncaps.outRequest(  aMethod,
                                         serverProxy);
}
```

Important: we must also make sure doesNotUnderstand is called for all methods, including inherited ones

The GARF system (3)

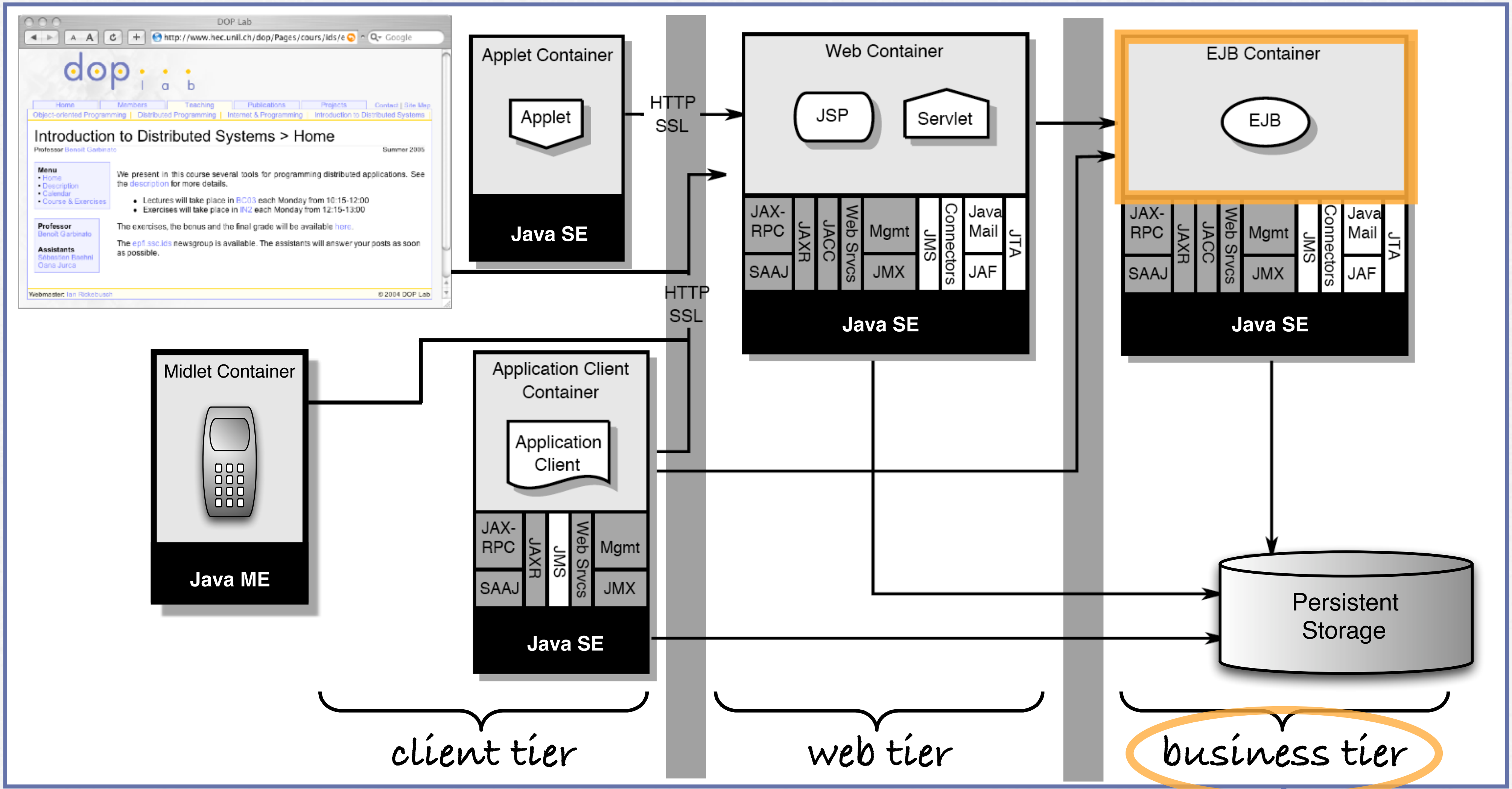
B. Garbinato, R. Guerraoui, and K. Mazouni. 1993. Distributed Programming in GARF. In *Proceedings of the Workshop on Object-Based Distributed Programming (ECOOP '93)*. Springer-Verlag, London, UK, 225-239.

B. Garbinato, R. Guerraoui, and K.R. Mazouni. Implementation of the GARF Replicated Object Platform. *Distributed Systems Engineering Journal*, 2:14–27, 1995.

The EJB model (1)

- The Enterprise JavaBeans model relies on two key notions:
  - Component: server-side software unit encapsulating business logic and deployed into a container; this is the actual Enterprise JavaBean (EJB).
  - Container: hosting environment interfacing the EJB with its clients and with the low-level platform services, and ultimately managing all technical aspects for the EJB; it is also known as the EJB Container.

The EJB model (2)



DOP Lab

http://www.hec.unil.ch/dop/Pages/cours/ids/e

Home | Members | Teaching | Publications | Projects | Contact | Site Map

Object-oriented Programming | Distributed Programming | Internet & Programming | Introduction to Distributed Systems

Introduction to Distributed Systems > Home

Professor Benoit Garbinato — Summer 2005

Menu
- Home
- Description
- Calendar
- Course & Exercises

Professor
Benoit Garbinato

Assistants
Sébastien Baehni
Oana Jurca

We present in this course several tools for programming distributed applications. See the description for more details.

- Lectures will take place in BC03 each Monday from 10:15-12:00
- Exercises will take place in IN2 each Monday from 12:15-13:00

The exercises, the bonus and the final grade will be available here.

The epfl.ssc.ids newsgroup is available. The assistants will answer your posts as soon as possible.

Webmaster: Ian Rickebusch — © 2004 DOP Lab

EJB 2 versus EJB 3

- The EJB specification has been drastically revised from version 2 to version 3
- The execution model is basically the same
- The programming model however has been deeply revisited
  - In version 2, the programming model is more explicit but also more complex, as it relies on multiple files
  - In version 3, the programming model is simpler but somehow more opaque, as it heavily relies on annotations and dependency injection

Annotations

□ An annotation is a portion of text that expresses information about the code directly in the code

□ An annotation does not directly modify the semantics of your code but the way it is treated by tools and library from

□ Java always had ad hoc annotation, e.g., Java comments, the transient keyword, etc.

□ Since Java SE 5, Java supports general and extensible annotations mechanism (@...)

□ In Java EE 5, annotations are used as a lighter alternative to deployment descriptors

```
@Stateless
@Stateful
@LocalBean
@Remote
@Resource
@EJB
@Remove
@PostConstruct
@PreDestroy
@PrePassivate
@PostActivate
...
```

Dependency injection

- Dependency injection is an alternative to having an object set its dependencies to other objects itself
- With dependency injection, an object's field can be set by an external actor, in our case the container
- Dependency injection is expressed by the programmer via annotations
- Dependency injection allows us to decouple various components at the code level

Types of EJBs (1)

There exists three types of Enterprise JavaBeans

Session: performs actions for the client, manages a conversation with it

EJB 2.1

Entity: represents a persistent business object, usually accessed within a transaction

Message-driven: acts as a JMS MessageListener and processes messages asynchronously

Types of EJBs (2)

- A session bean can be either :
  - stateless: it belongs to a client only during a method call
  - stateful: it belongs to a client for the whole conversation this client holds with the application

EJB 2.1

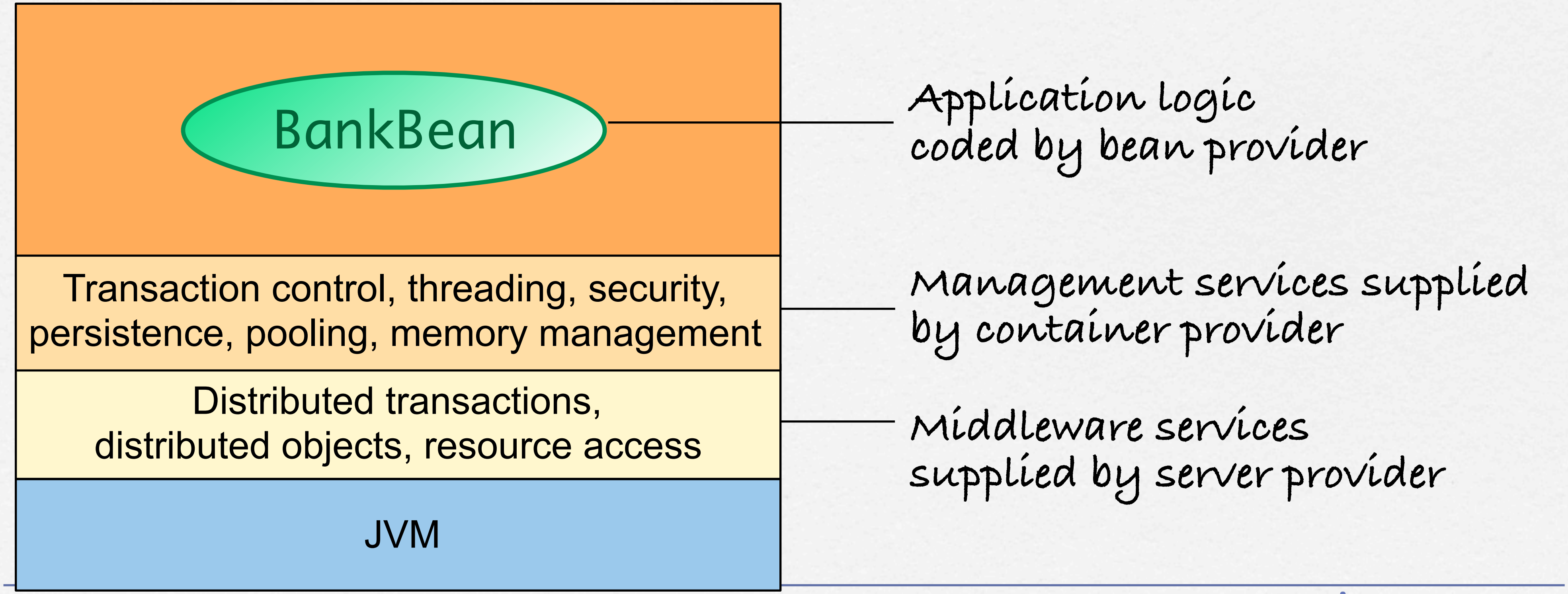
- An entity bean can have its persistence either:
  - bean-managed: the developer writes SQL code to retrieve, store and update persistent information (in the database)
  - container-managed: the developer provides a relational mapping, which is used by the container to automatically manages the persistence of the entity bean

Managing skills needs

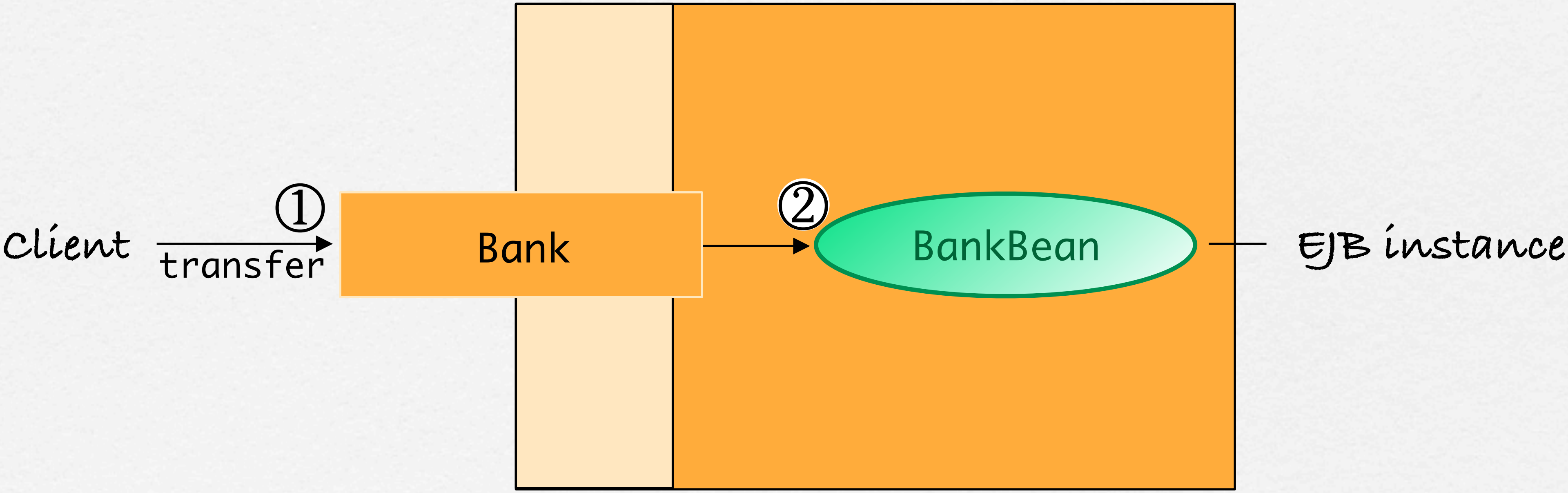
- The Bean Provider develops enterprise beans and produces an ejb-jar containing one or more EJBs (hereafter bean ⇔ EJB).
- The Application Assembler combines several EJBs into larger deployable units, still as ejb-jars.
- The Deployer takes one ore more ejb-jars and deploys them in a specific operational environment (application server/container).
- The Container Provider provides tools for deploying EJBs and runtime support for the deployed EJBs, in the form of a container.
- The Server Provider provides the low-level system services on which the container relies, e.g., transactions, persistence, etc.
- The System Administrator manages the computing & networking infrastructure, including the container & server.

Container responsibilities

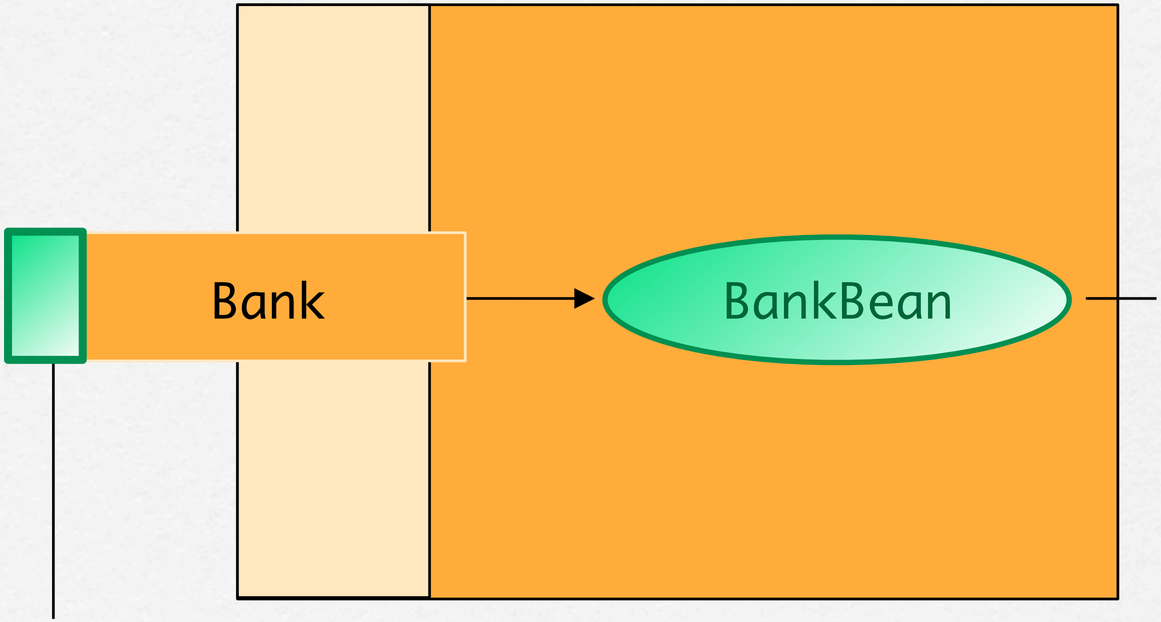
The container intercepts client calls to manage the EJB lifecycle and its technical needs


BankBean
Transaction control, threading, security, persistence, pooling, memory management
Distributed transactions, distributed objects, resource access
JVM
Application logic coded by bean provider
Management services supplied by container provider
Middleware services supplied by server provider

Container as interceptor



Bean provider tasks

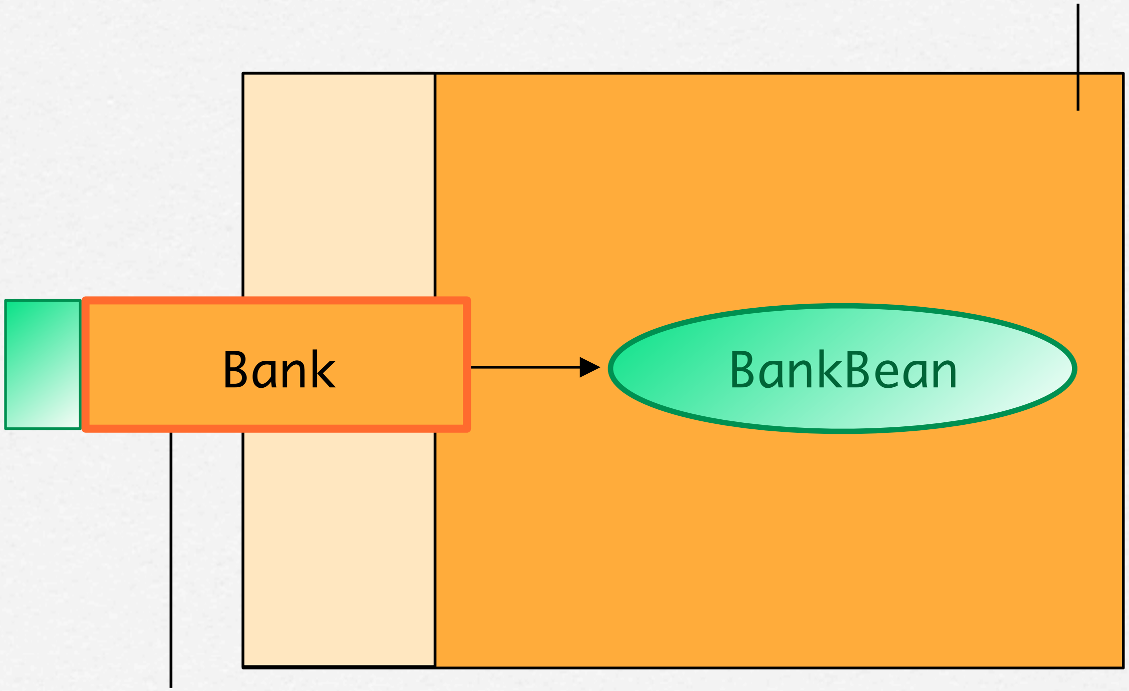


creates the bean class, coding business methods

creates the remote interface

Container provider tasks

provide an EJB-compliant container



implements the remote interface,
i.e., provides the interceptor object

A typical session bean

```
@Remote
public interface BankRemote {
    public void transfer( Account source, Account destination,double amount )
    throws BankingException;
    void initialize();
}
```

```
@Stateless
public class BankBean implements BankRemote {
    @Resource
    SessionContext ctx;

    public void transfer( Account source, Account destination,double amount )
    throws BankingException { ... }

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

dependency injection

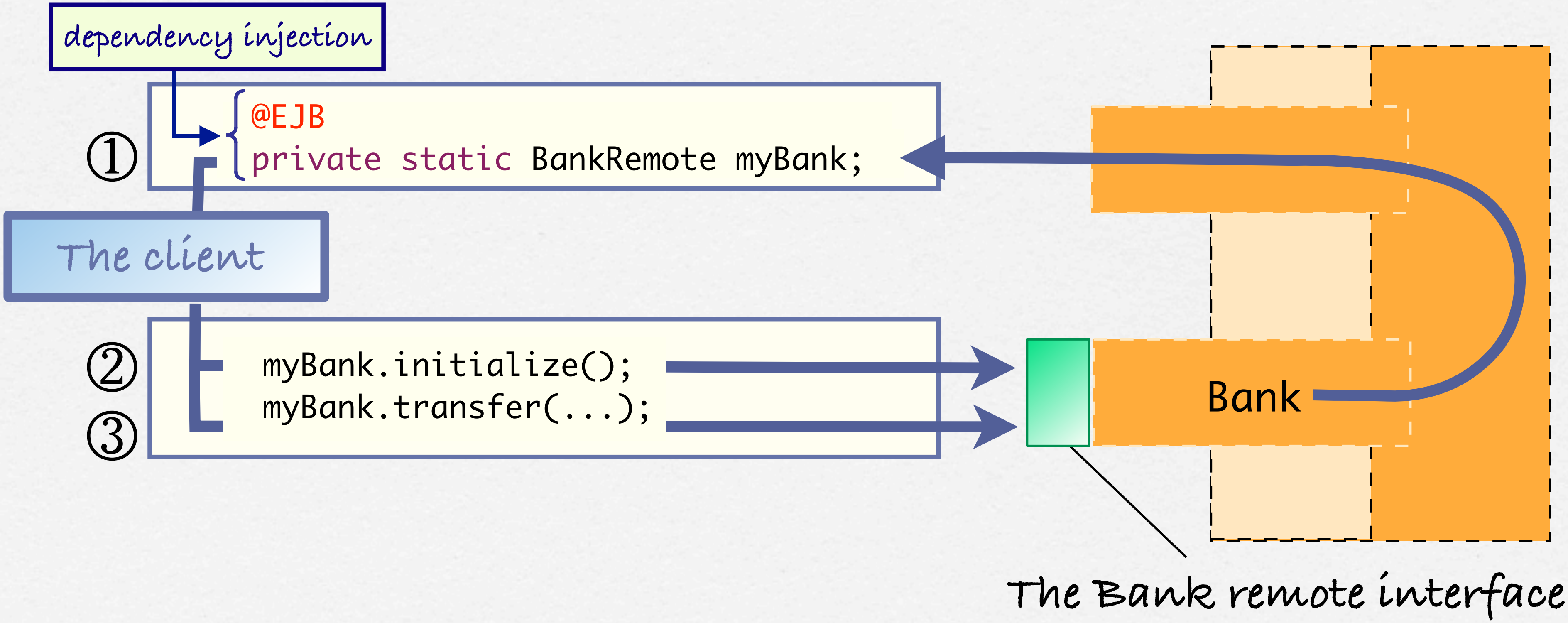
Local beans

- □ A bean can also provide a local interface, marked by the @Local annotation, in order to expose methods to components deployed in the same address space, e.g., another bean or a servlet (deployed together with the bean)
- □ While it is possible for a bean to provide both a local interface and a remote interface, this is usually considered bad practice
- □ A bean marked by the @LocalBean annotation can only be invoked locally and you do not need to provide a separate Java interface for that bean

Singleton beans

- □ In software engineering, the singleton pattern is used to implement the mathematical concept of a singleton, by restricting the instantiation of a given type of object to one and one instance only
- □ To make a given type of bean a singleton, simply mark the corresponding class with the `@Singleton` annotation
- □ As a consequence, the container ensures that any reference to a bean of that class point to the same instance
- □ A singleton bean is stateful by definition

Client developer tasks



dependency injection

①

```
@EJB
private static BankRemote myBank;
```

The client

②
③

```
myBank.initialize();
myBank.transfer(...);
```

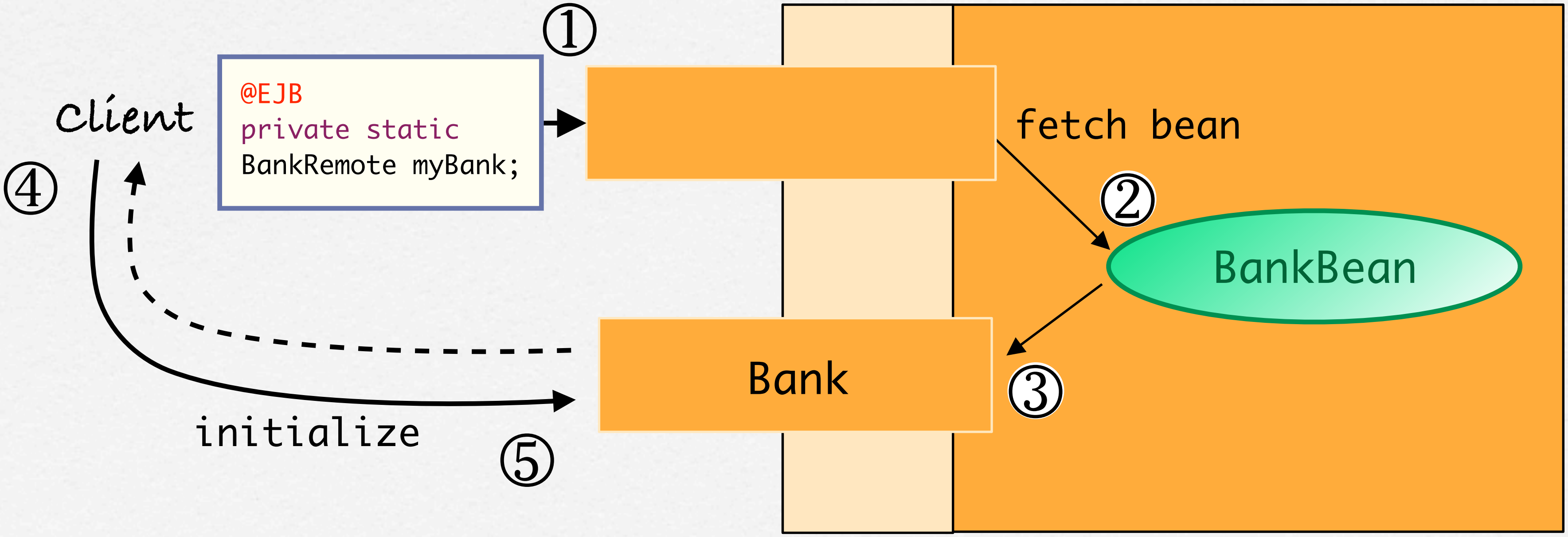
Bank

The Bank remote interface

Creating session beans

Stateless bean: no need for an initialization method

Stateful bean: one or more initialization methods (business method)



```
@EJB
private static
BankRemote myBank;
```

Creating session beans

Stateless bean: no need for an initialization method

Stateful bean: one or more initialization methods (business method)

```
...
Context c = new InitialContext();
BankRemote theBank = (BankRemote) c.lookup("java:global/ubs-app/Bank");
theBank.initialize();
...
```

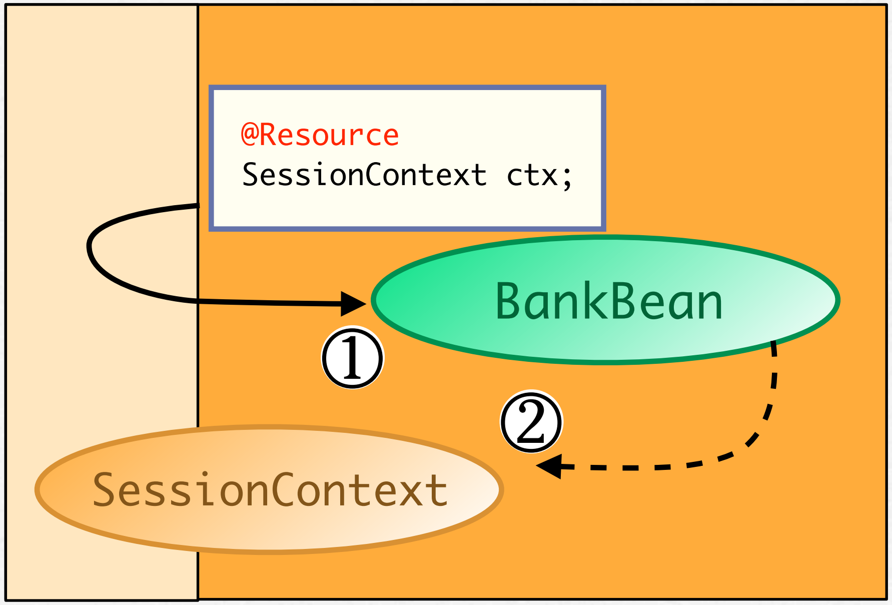
assuming we have:

```
@Stateful(mappedName = "java:global/ubs-app/Bank")
public class BankBean implements BankRemote {
...
}
```

Session context

The SessionContext object provides access to container services, e.g., to:

- □ the interceptor object
- □ the transaction context
- □ the security context

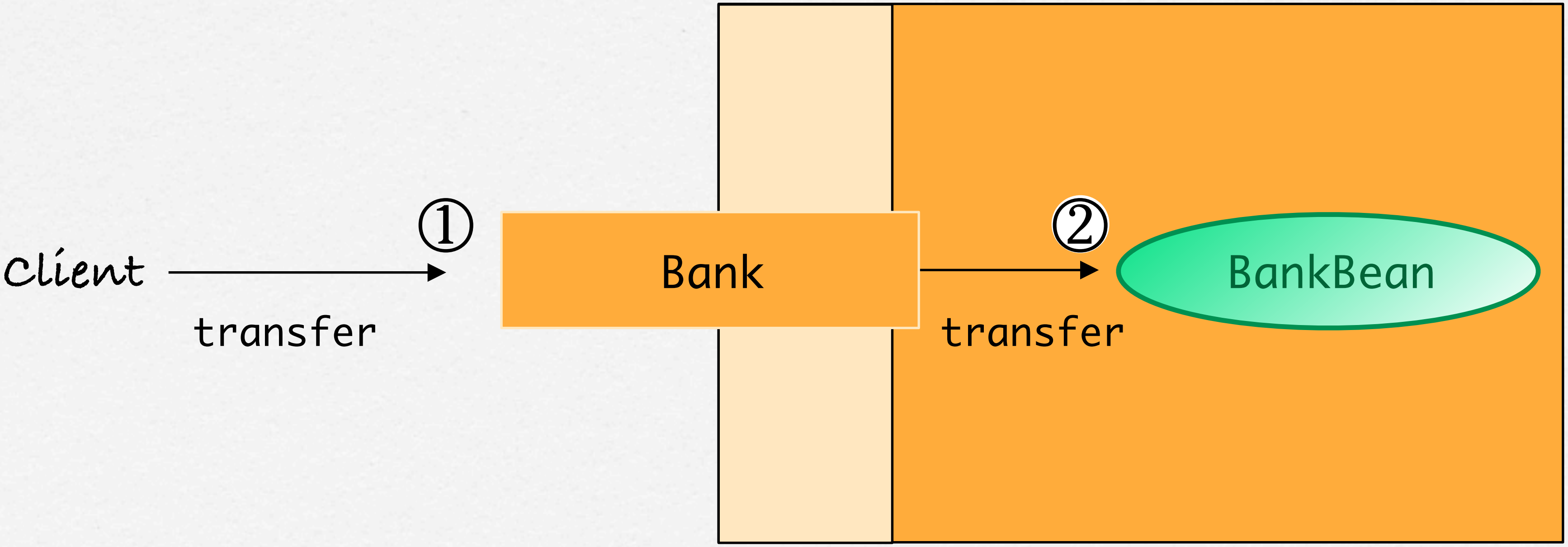


```
@Resource
SessionContext ctx;
```

Business methods

The BankBean object is not a remote object, but its interceptor object (implementing the Bank interface) is,

so this object throws java.rmi.RemoteException



Removing a session bean...

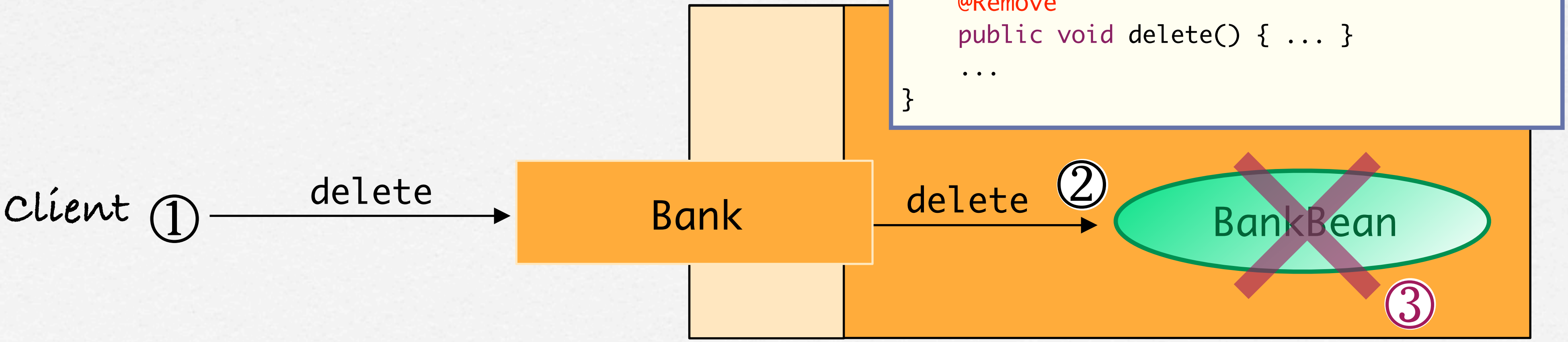
□ ... is useful to perform some house cleaning before stopping to use that bean

□ ... is useful to indicate to the container that we no longer need that bean

□ ... is performed:

1. in the bean code by marking a method using the @Remove annotation
2. in the client code by calling that method on the bean

```
@Stateful
public class BankBean implements BankRemote {
    ...
    @Remove
    public void delete() { ... }
    ...
}
```



Resource pooling

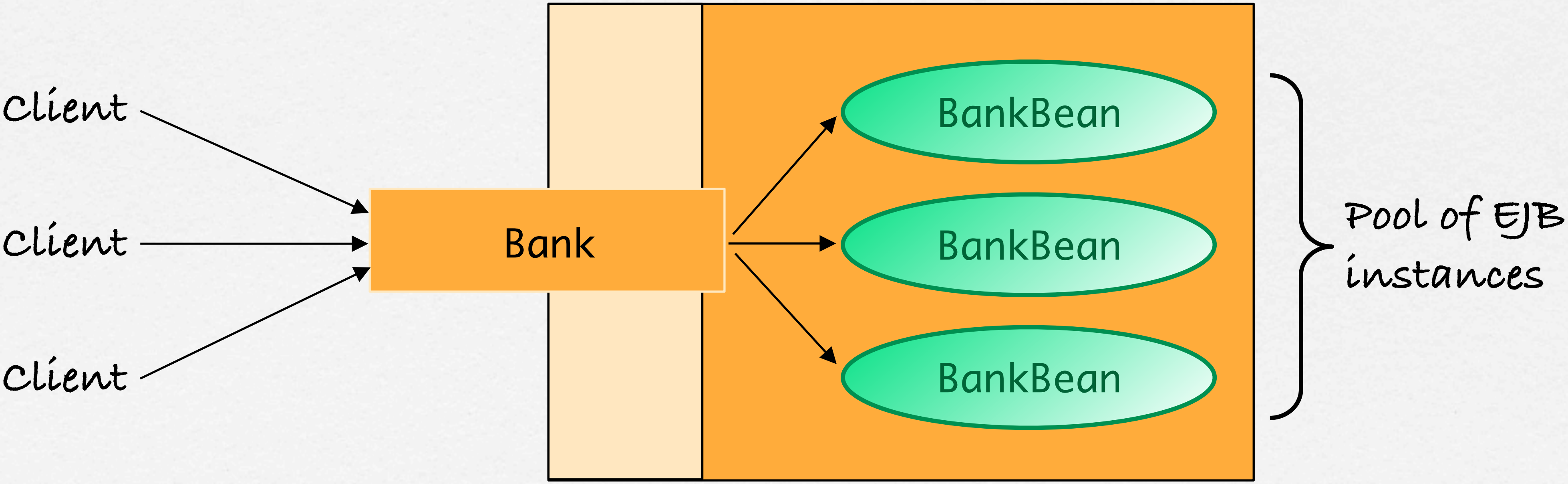
- □ Among the various resources managed by the container, we find connections (to databases, to moms, etc.), threads, memory, etc., and the EJBs themselves

- □ To ensure adequate performance & scalability, the container uses various pooling strategies to manage resources

Session bean pooling (1)

How does the container manage stateless session beans?

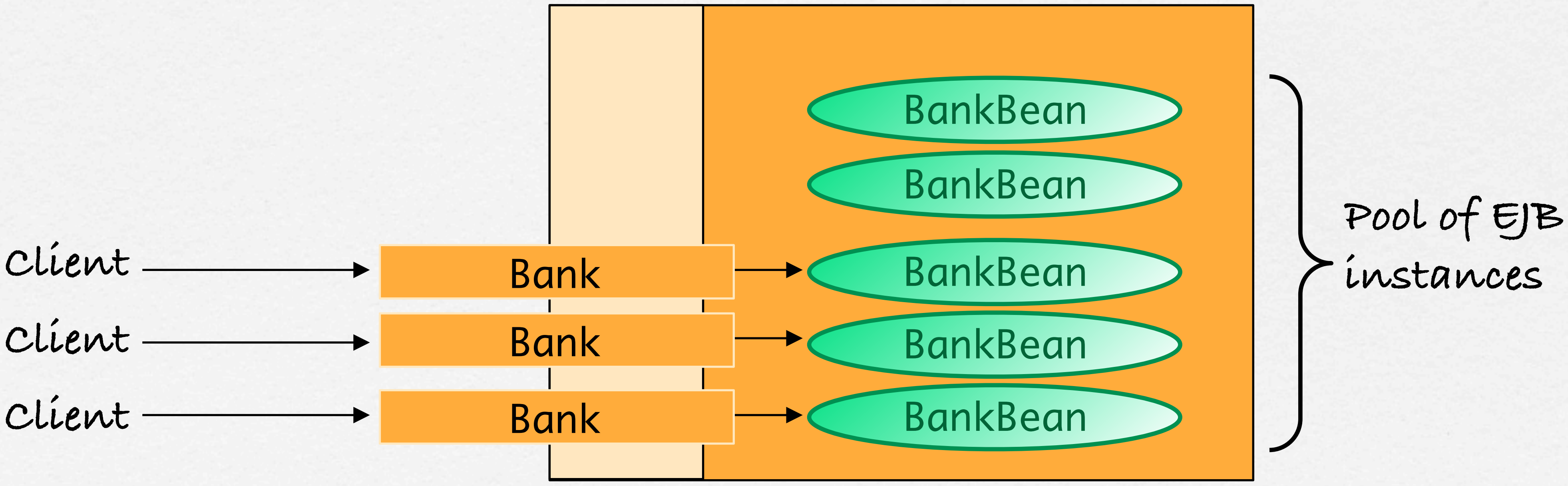
➥ It fetches any bean from the pool for any call



Session bean pooling (2)

How does the container manage stateful session beans?

➡ It dedicates a specific bean to each client session



Passivation/Activation (1)

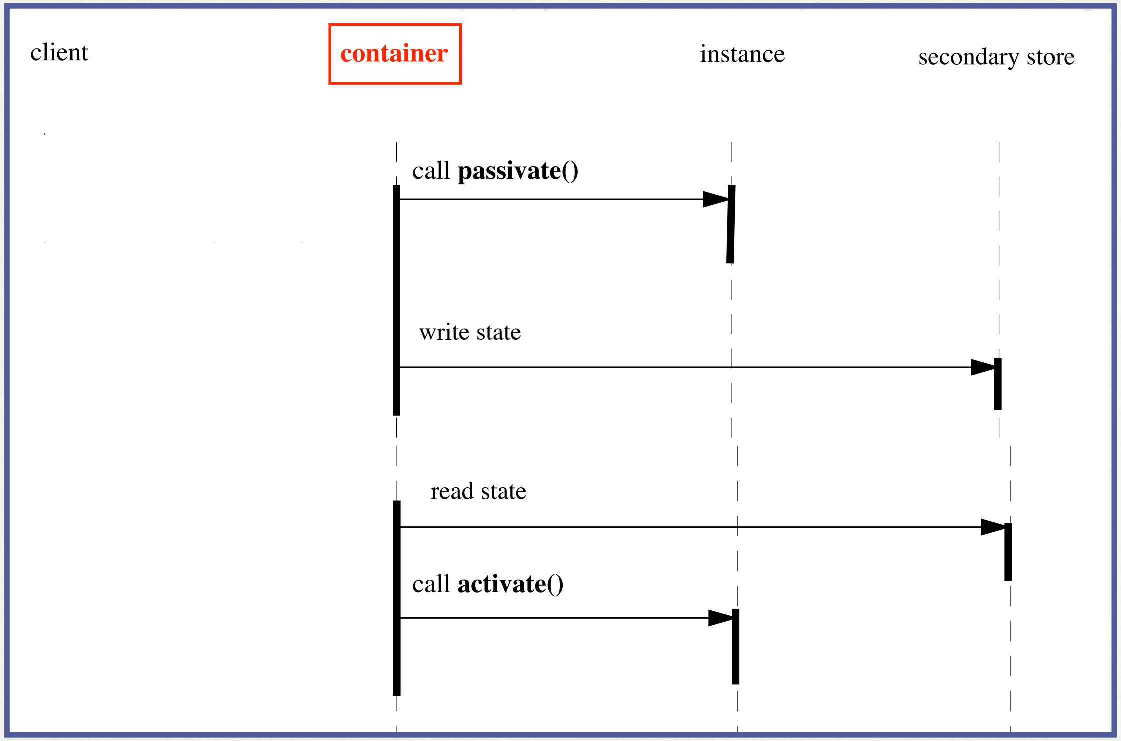
- □ A container can only host a limited number of session beans in memory

- □ When more stateful session beans are needed, the container uses an passivation/activation strategy
  - Passivation: write a bean to disk and remove it (swap out)
  - Activation: read a bean from disk and recreate it (swap in)
  - usually follows a Least Recently Used (LRU) policy

- □ The container can only manage part of the state of a passivated/activated session bean, i.e., primitive types, serializable objects, context objects, etc.

- □ For state (fields) outside this category, the bean provider must manage activation/passivation programmatically

Passivation/Activation (2)



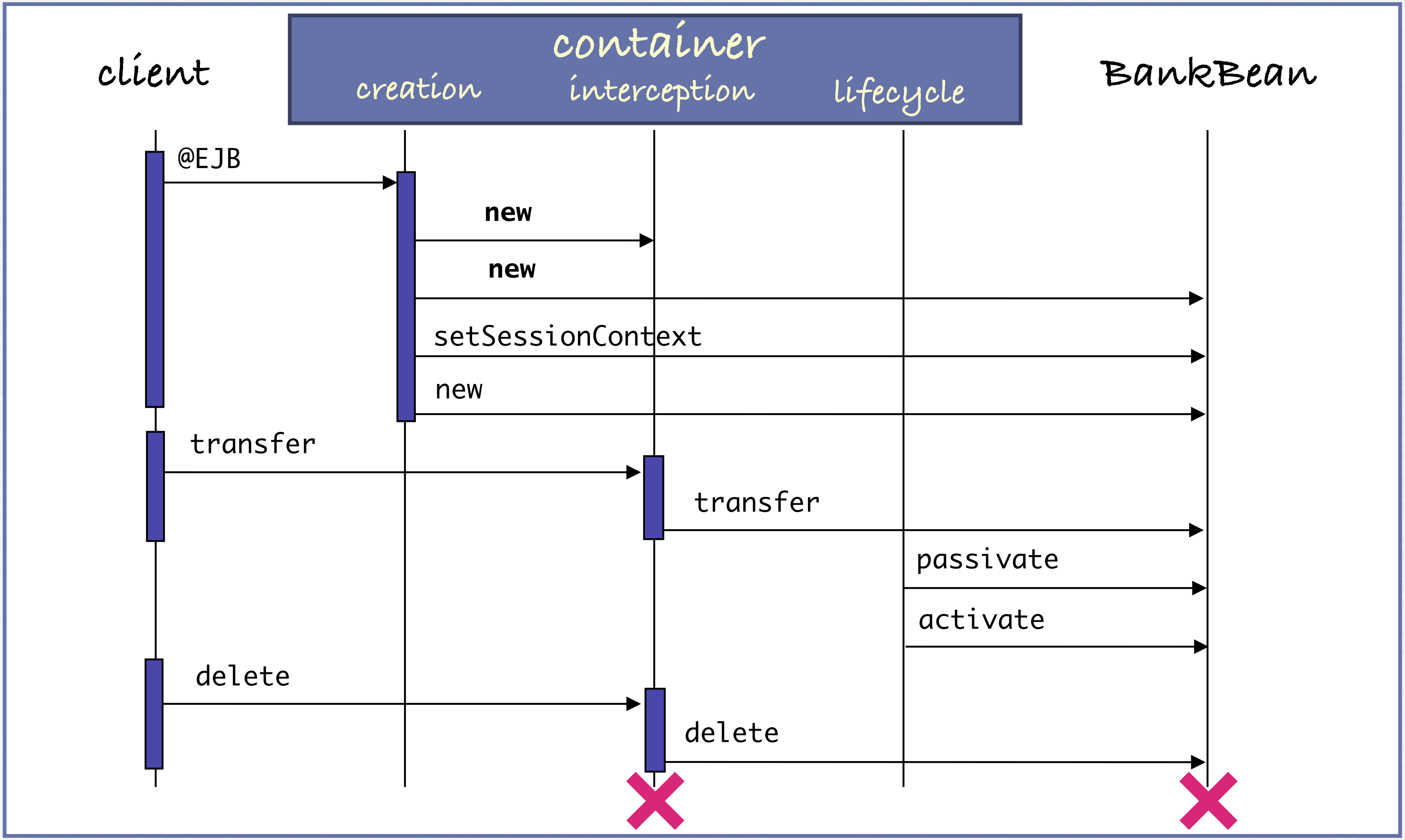
```
@Stateful
public class BankBean implements BankRemote {
    ...
    @PrePassivate
    public void passivate() { ... }
    @PostActivate
    public void activate() { ... }
}
```

Session bean contract

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.annotation.Resource;
import javax.ejb.PostActivate;
import javax.ejb.PrePassivate;
import javax.ejb.Remove;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;

@Stateful
public class BankBean implements BankRemote {
    @Resource
    SessionContext ctx;
    public void initialize() { ... }
    @Remove
    public void delete() { ... }
    @PostConstruct
    public void construct() { ... }
    @PreDestroy
    public void destroy() { ... }
    @PrePassivate
    public void passivate() { ... }
    @PostActivate
    public void activate() { ... }
}
```

called by container
(optional)

Lifecycle of a session bean



Deployment descriptor (1)

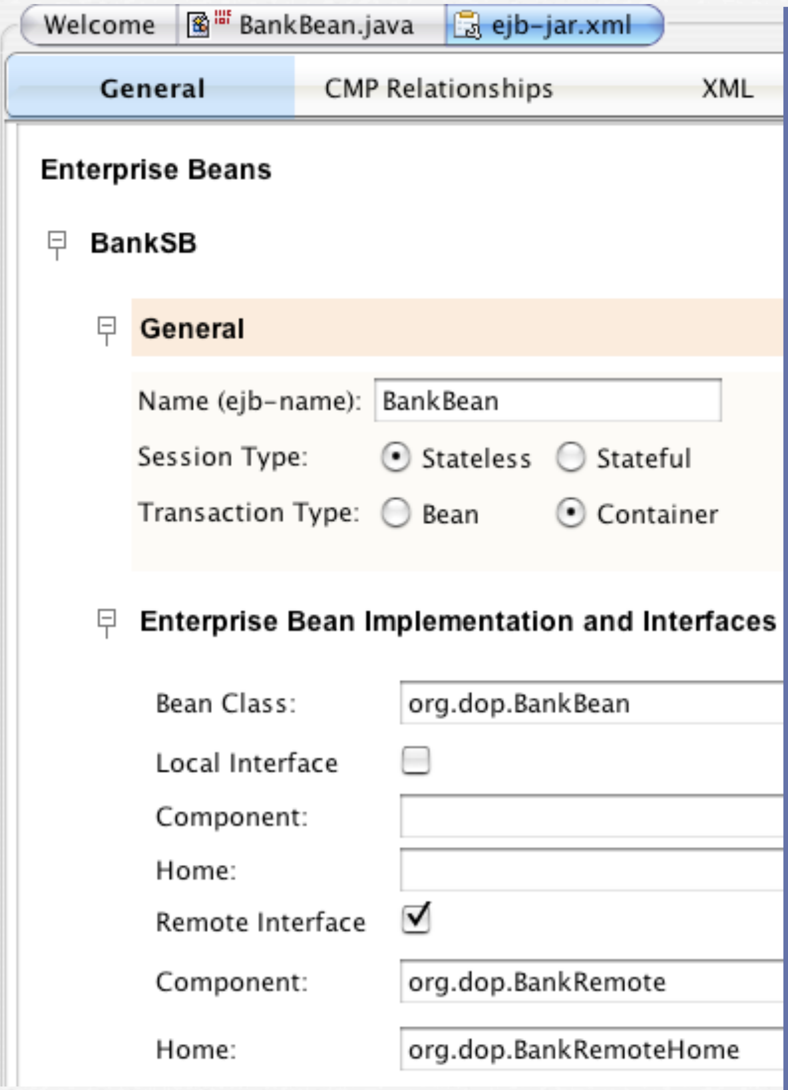
□ A deployment descriptor is associated with one or more EJBs, within the corresponding ejb-jar file

□ It expresses how the container should handle the technical aspects with respect to these EJBs, e.g., security, transactions, persistence, etc.

□ It is written in XML and its format is standardized by the EJB specification

□ In EJB 3, the deployment descriptor is optional and supersedes annotations

Deployment descriptor (2)



Enterprise Beans

BankSB

General

Name (ejb-name): BankBean

Session Type: ● Stateless ○ Stateful

Transaction Type: ○ Bean ● Container

Enterprise Bean Implementation and Interfaces

Bean Class: org.dop.BankBean

Local Interface □

Component:

Home:

Remote Interface ☑

Component: org.dop.BankRemote

Home: org.dop.BankRemoteHome

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="2.1" ... >
  <display-name>BankApplication-EJBModule</display-name>
  <enterprise-beans>
    <session>
      <display-name>BankSB</display-name>
      <ejb-name>BankBean</ejb-name>
      <home>org.dop.BankRemoteHome</home>
      <remote>org.dop.BankRemote</remote>
      <ejb-class>org.dop.BankBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>BankBean</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

Atomic transactions

A transaction T ensures the four ACID properties:

Atomicity. T appears either committed or aborted with respect to failures

Consistency. T does not compromise the consistency of the data it manipulates

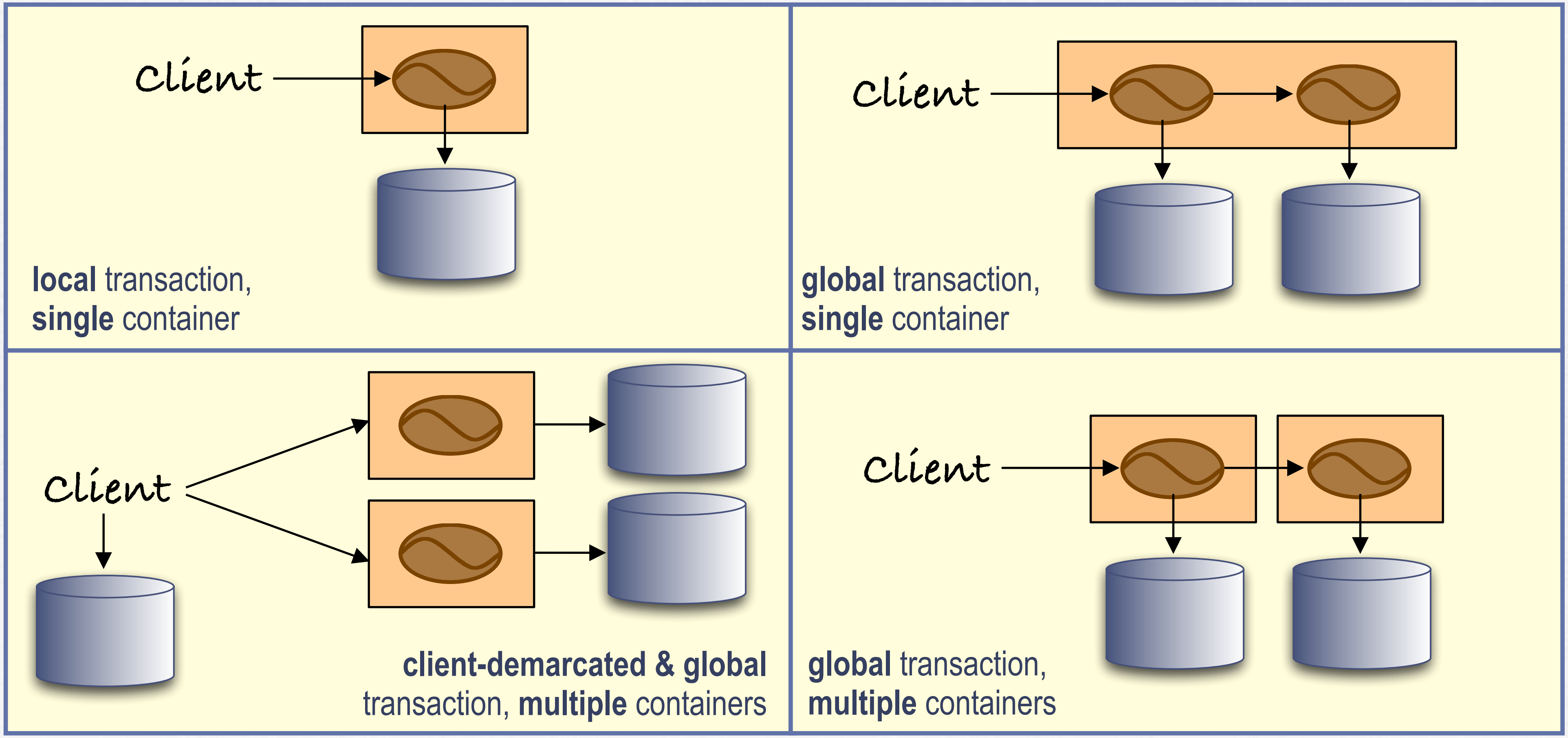
Isolation. T appears indivisible with respect to all other transactions

Durability. T being committed, its effects will survive subsequent crashes

Transactions with EJBs

- □ The EJB transactional model supports various scenarios
- □ The EJB model offers two ways to express transactional needs:
  - □ programmatically (⇔bean-managed)
  - □ declaratively (⇔container-managed)

Transactional scenarios



local transaction,
single container

global transaction,
single container

client-demarcated & global
transaction, **multiple** containers

global transaction,
multiple containers

Programmatic transactions

```
@Resource(name=”jdbc/EmployeeAppDB”, type=javax.sql.DataSource)
@Stateless public class WarehouseBean implements SessionBean {
    private DataSource ds;
    private Connection cn;
    @Resource SessionContext ctx;
    public void ship(String productId, String orderId, int quantity) {
        try {
            ds = (javax.sql.DataSource) ctx.lookup("jdbc/EmployeeAppDB");
            cn = ds.getConnection();
            cn.setAutoCommit(false);
            updateOrderItem(productId, orderId);
            updateInventory(productId, quantity);
            cn.commit();
        } catch (Exception ex) {
            try {
                cn.rollback();
                throw new EJBException("Transaction failed: " + ex.getMessage());
            } catch (SQLException sqx) {
                throw new EJBException("Rollback failed: " + sqx.getMessage());
            }
        } finally {
            cn.close();
        }
    ...
    }
```

local transaction

Programmatic transactions

```
@Stateless
@TransactionManagement(javax.ejb.TransactionManagementType.BEAN)
public class TellerBean implements TellerRemote {                ⬆
    ...
    public void withdrawCash(double amount) {
      ➡ UserTransaction ut =
          context.getUserTransaction();
        try {
          ➡ ut.begin();
            updateChecking(amount);
            machineBalance -= amount;
            insertMachine(machineBalance);
          ➡ ut.commit();
        } catch (Exception ex) {
            try {
              ➡ ut.rollback();
            } catch (SystemException syex) {
                throw new Exception("Rollback failed: " + syex.getMessage());
            }
            throw new Exception("Transaction failed: " + ex.getMessage());
        }
    }
}
```

global transaction

Declarative transactions (1)

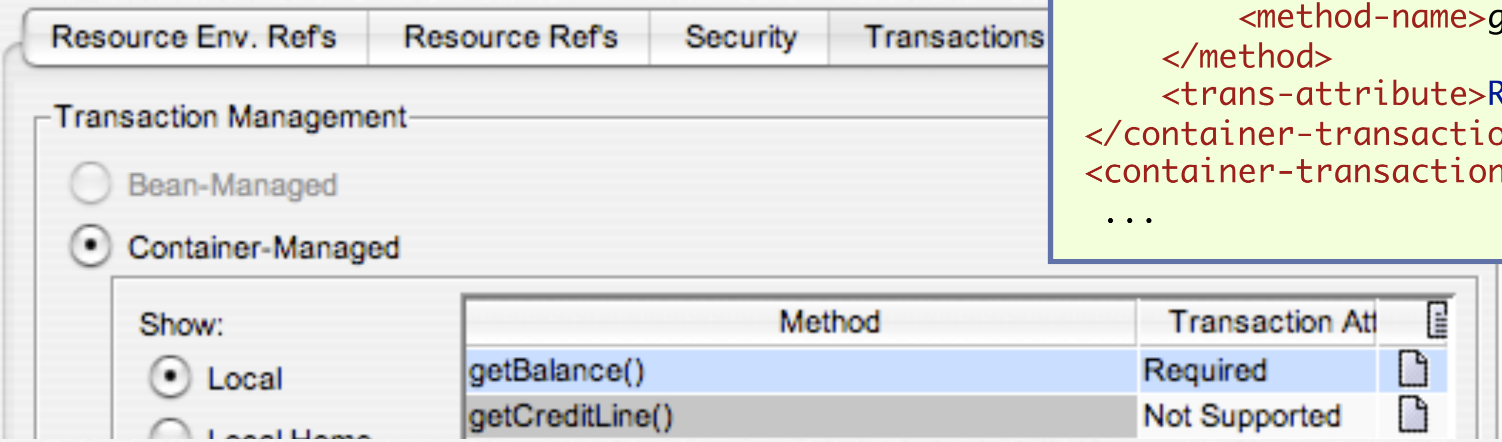
A transactional attribute is associated with each method via annotations or deployment descriptors

Attribute	Meaning
NotSupported	If a client's transaction exists, it is suspended
Supports	If a client's transaction exists, it is continued
Required	If a client's transaction exists, it is continued; otherwise, the container starts a new transaction
RequiresNew	The container always starts a new transactions; if a client's transaction exists, it is suspended first
Mandatory	The client must be in a transaction when calling
Never	The client must not be in a transaction when calling

Declarative transactions (2)

```
@Stateless
@TransactionManagement(javax.ejb.TransactionManagementType.CONTAINER)
public class AccountBean implements AccountLocal {
    ...

    @TransactionAttribute(javax.ejb.TransactionAttributeType.SUPPORTS)
    public double getBalance() { ... }
}
```

```
 ...
<container-transaction>
    <method>
        <ejb-name>AccountBean</ejb-name>
        <method-intf>Local</method-intf>
        <method-name>getBalance</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
 ...
```



Transaction Management

○ Bean-Managed

● Container-Managed

Show:

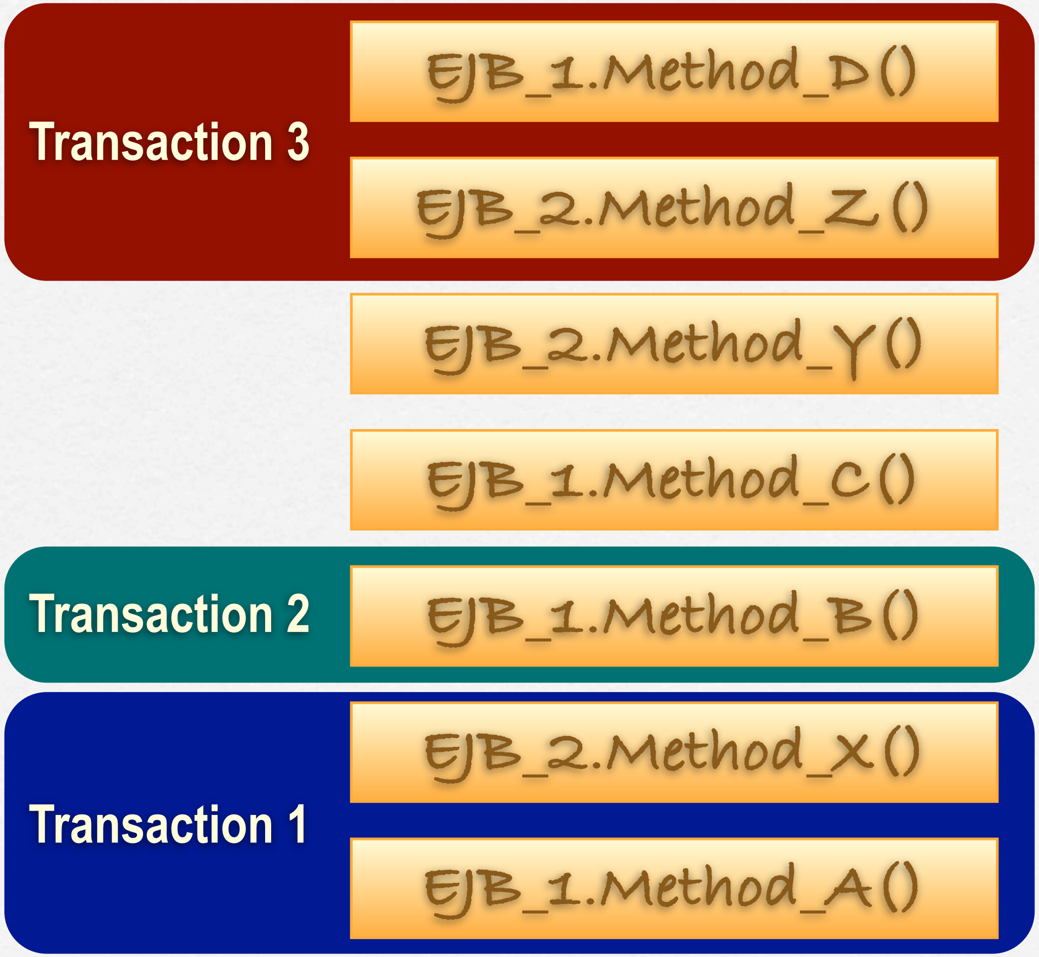
● Local

○ Local Home

Method	Transaction Att
getBalance()	Required
getCreditLine()	Not Supported

Declarative transactions (3)

call stack



transactional attributes

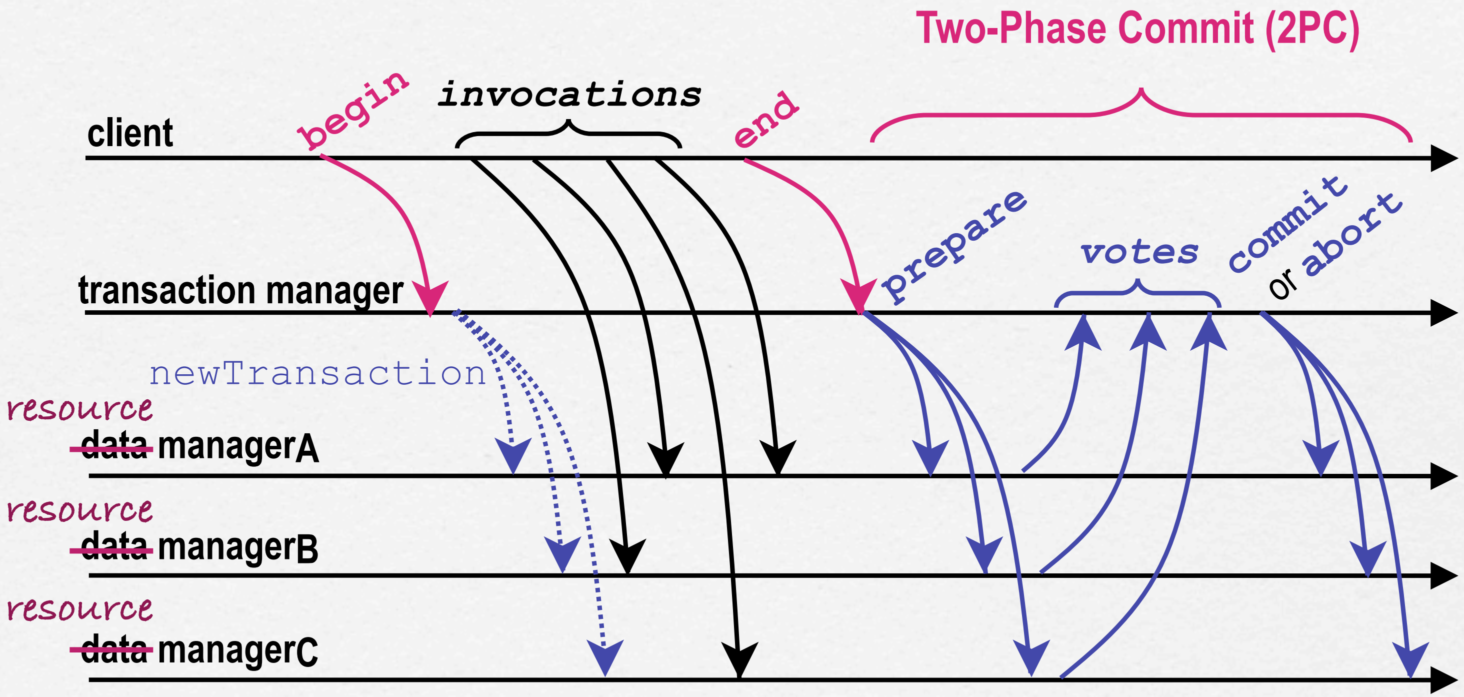
EJB_1.Method_D	Mandatory
EJB_2.Method_Z	Required
EJB_2.Method_Y	Supports
EJB_1.Method_C	NotSupported
EJB_1.Method_B	RequiresNew
EJB_2.Method_X	Supports
EJB_1.Method_A	Required

Rolling back transactions

How can we tell the container to rollback a transaction, because of some applicative problem occurred?

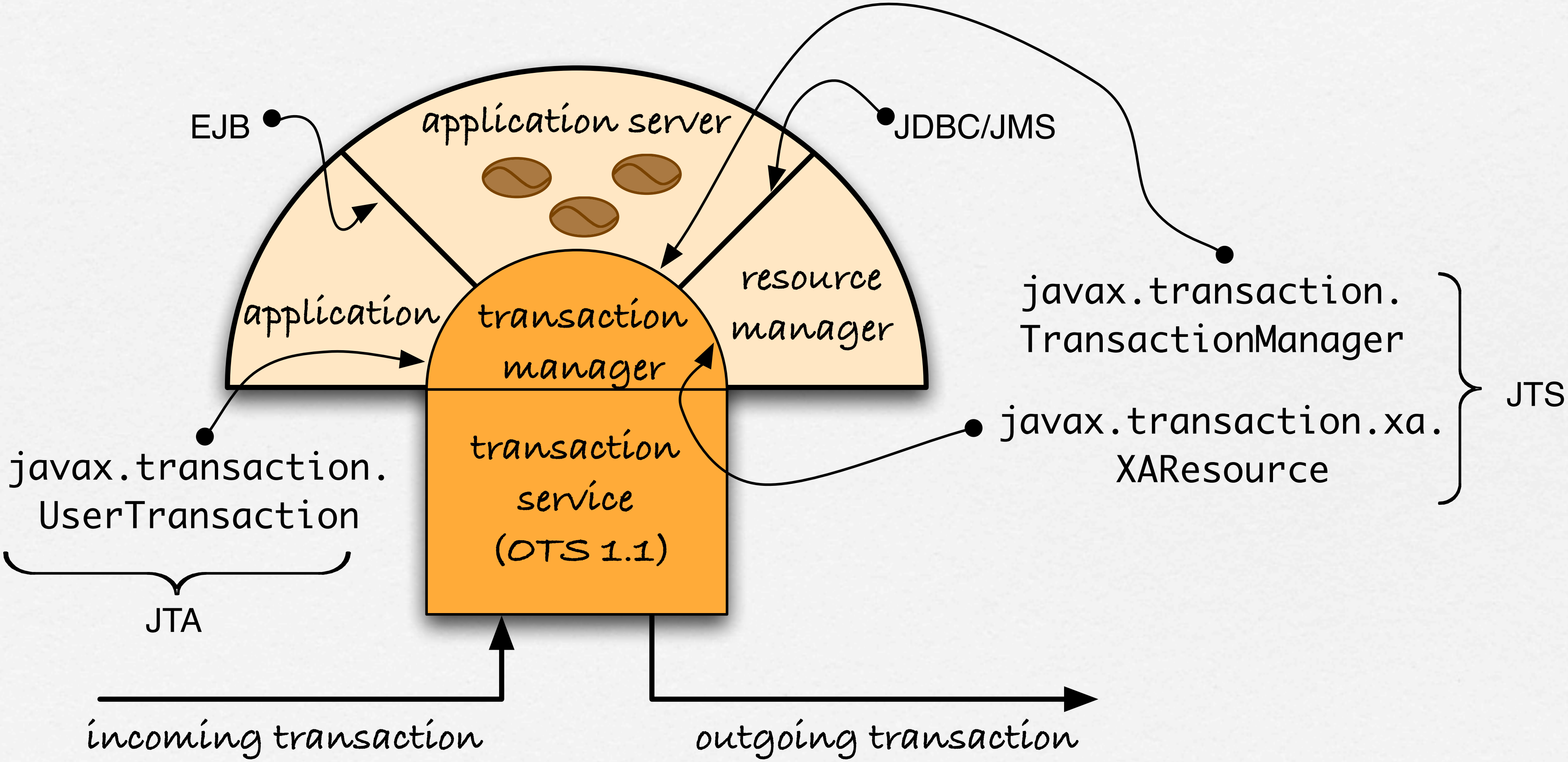
```
public void transferToSaving(double amount) throws InsufficientBalanceException {
    checkingBalance -= amount;
    savingBalance += amount;

    if (checkingBalance < 0.00) {
      ➡ context.setRollbackOnly();
         throw new InsufficientBalanceException();
    }

    updateChecking(checkingBalance);
    updateSaving(savingBalance);
    ...
}
```

~~Distributed~~ global transactions



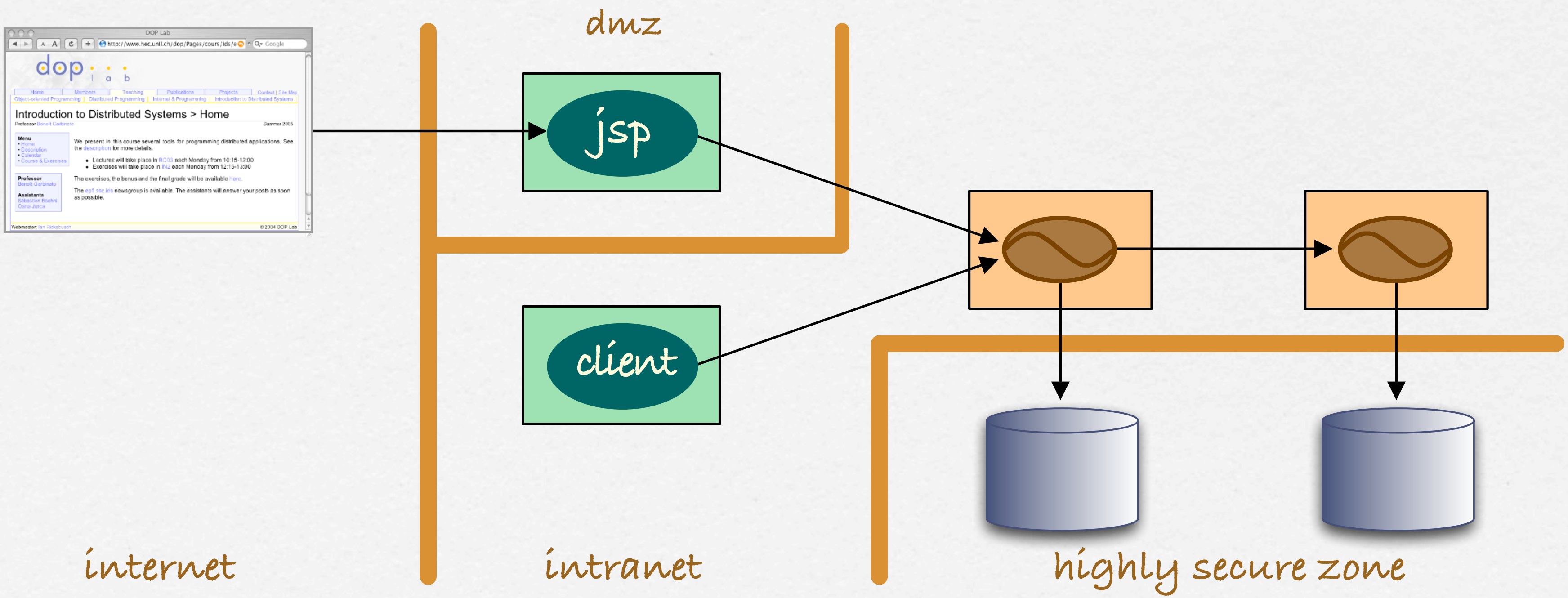
The transaction manager and all ~~data~~ resource managers must at least run compatible protocols (JTS/OTS)

Global transactions (APIs)



Context propagation

The various containers play a key role in propagating the context across tiers, typically security & transaction contexts



DOP Lab

http://www.hec.unil.ch/dop/Pages/cours/ids/e

Google

dop l a b

Home | Members | Teaching | Publications | Projects | Contact | Site Map

Object-oriented Programming | Distributed Programming | Internet & Programming | Introduction to Distributed Systems

Introduction to Distributed Systems > Home

Professor Benoit Garbinato — Summer 2005

Menu
- Home
- Description
- Calendar
- Course & Exercises

Professor
Benoit Garbinato

Assistants
Sébastien Baehni
Oana Jurca

We present in this course several tools for programming distributed applications. See the description for more details.

- Lectures will take place in BC03 each Monday from 10:15-12:00
- Exercises will take place in IN2 each Monday from 12:15-13:00

The exercises, the bonus and the final grade will be available here.

The epfl.ssc.ids newsgroup is available. The assistants will answer your posts as soon as possible.

Webmaster: Ian Rickebusch — © 2004 DOP Lab

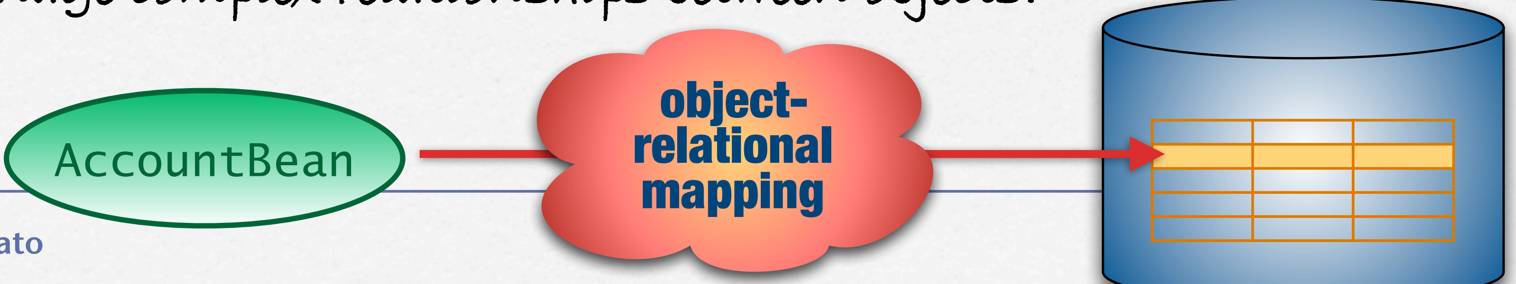
Message-driven beans

- □ A message-driven bean is a bean that can receive asynchronous messages
- □ It is invoked by the container upon arrival of a message at a given destination
- □ It is decoupled from clients, stateless and single-threaded

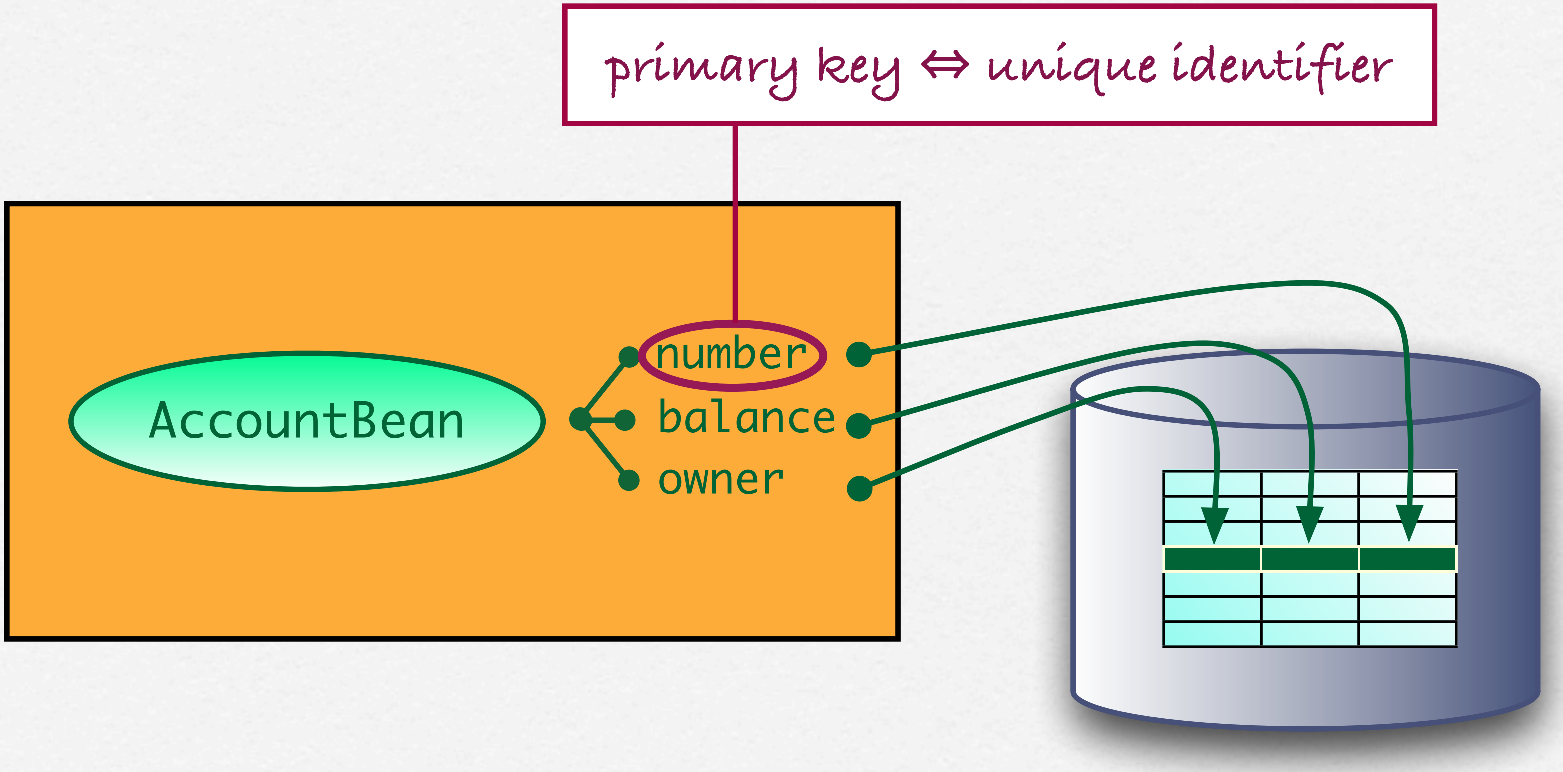
```
@MessageDriven(mappedName = "jms/OrderQueue", activationConfig =  {
    @ActivationConfigProperty(propertyName = "acknowledgeMode",
                              propertyValue = "Auto-acknowledge"),
    @ActivationConfigProperty(propertyName = "destinationType",
                              propertyValue = "javax.jms.Queue")  })
public class OrderListenerBean implements MessageListener {
    public void onMessage(Message message) { ... }
    ...
}
```

Persisting objects

- □ To ensure persistence basically means to ensure the durability property of transactions
- □ It can be done via object serialization but:
  - ▸ no easy navigation and querying of the object graph
  - ▸ no support of legacy persistent data, stored in relational databases
- □ The object-relational mapping approach:
  - ▸ How should we persist a graph of objects into a relational database, and what is the reference model?
  - ▸ What happens to fields, constructors & methods ?
  - ▸ How do manage complex relationships between objects?



Object-relational mapping



Question: how is the mapping done ?

Solutions in the Java platform

- □ The Java Persistence API…
  - … is more recent (2006) and merges several previous efforts
  - … is available in both the Java Standard Editions (Java SE) and the Java Enterprise Edition (Java EE) platforms
  - … is portable across operating systems
  - … relies on the notion of entities

EJB 2.1

- The entity bean model…
  - … came first, as part of the EJB programming model
  - … is also portable across operating systems
  - … is still valid, i.e., not deprecated

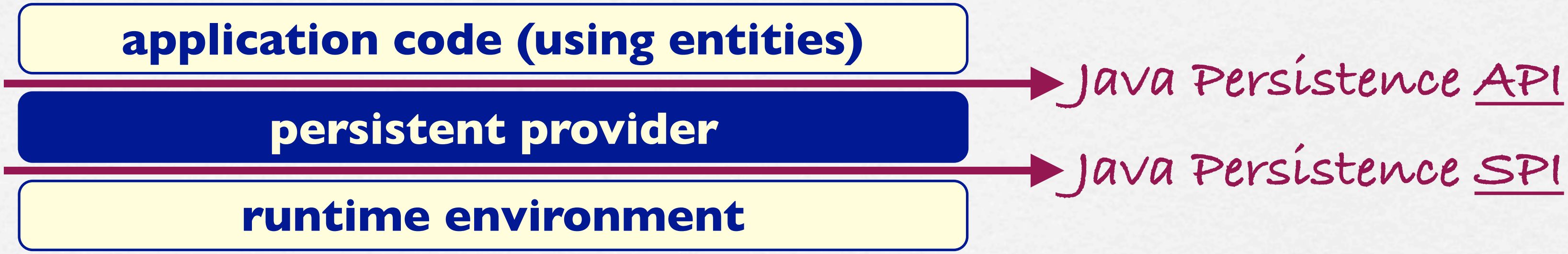
entities ≠ entity beans

What is an entity?

- □ An entity is a POJO (Plain Old Java Object), not an EJB
- □ It is not remotely accessible (unlike session or entity beans)
- □ It represents data stored in a relational database
- □ It provides basic methods to manipulate that data
- □ It has a persistent identity (primary key) that is distinct from its object reference (in memory)
- □ Its lifetime may be completely independent of the application lifetime in which it is used
- □ The persistence aspect is managed via annotations and calls to the persistence provider API

Persistence provider

- □ The Java Persistence API defines the notion of persistence provider, which…
  - … is responsible for the object-relational mapping
  - … complies with a Service Provider Interface (SPI)



- □ The SPI is what makes the persistence provider pluggable into both the Java SE and EE runtime environments
- □ In Java EE, the runtime is simply the EJB 3.0 container
- □ The object-relational mapping is transparent to entities

A typical entity

why is it serializable ?

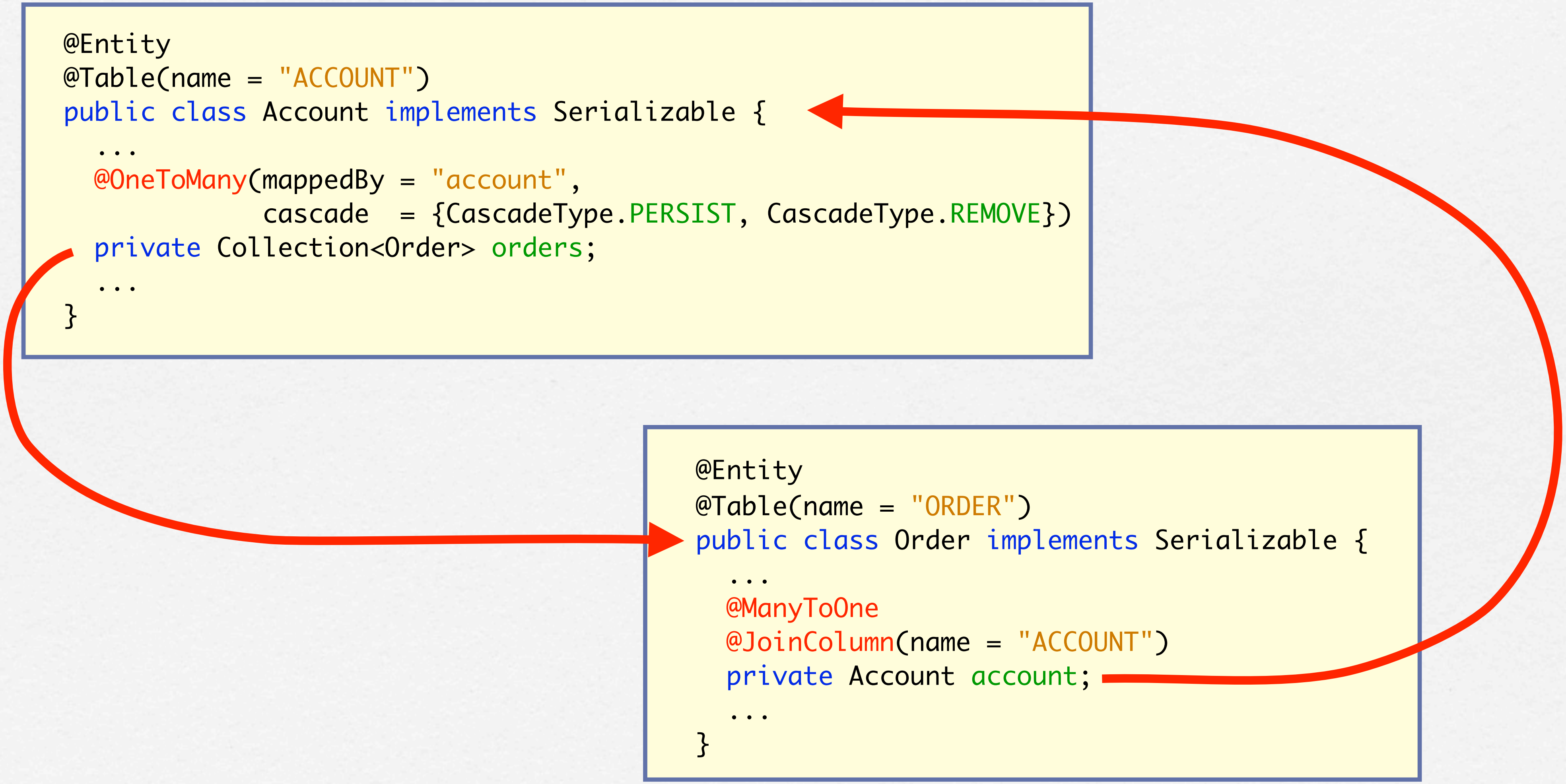
primary key

```java
@Entity
@Table(name = "ACCOUNT")
public class Account implements Serializable {
    @Id
    @Column(name = "ACCTNUMBER", nullable = false)
    private Integer acctnumber;

    @Column(name = "NAME")
    private String name;

    @Column(name = "BALANCE")
    private Integer balance;

    public Account() {
        this.acctnumber =
            (int) System.currentTimeMillis();
        this.balance = 0;
    }

    public Integer getAcctnumber() {
        return acctnumber;
    }
    ...
```

```java
    ...

    public Integer getAcctnumber() {
        return acctnumber;
    }

    public void setAcctnumber(Integer acctnumber) {
        this.acctnumber = acctnumber;
    }

    public void deposit(int amount) {
        balance += amount;
    }

    public int withdraw(int amount) {
        if (amount > balance) return 0;
        else {
            balance -= amount;
            return amount;
        }
    }
}
```

```sql
CREATE TABLE ACCOUNT(ACCTNUMBER INT  PRIMARY KEY, NAME VARCHAR(256), BALANCE INT);
```

Relationship management



```
@Entity
@Table(name = "ACCOUNT")
public class Account implements Serializable {
  ...
  @OneToMany(mappedBy = "account",
             cascade  = {CascadeType.PERSIST, CascadeType.REMOVE})
  private Collection<Order> orders;
  ...
}
```

```
@Entity
@Table(name = "ORDER")
public class Order implements Serializable {
  ...
  @ManyToOne
  @JoinColumn(name = "ACCOUNT")
  private Account account;
  ...
}
```

Using an entity (1)

- □ Since entities cannot be accessed remotely, they are typically deployed together with EJBs using them
- □ Before using an entity, an EJB must first retrieve it from the persistence context
- □ The persistence context is part of the persistence provider API and responsible for the connection with the database
- □ The persistence context is materialized via the `EntityManager` interface (API)

Using an entity (2)

```
@Stateless
@TransactionManagement(javax.ejb.TransactionManagementType.CONTAINER)
public class BankBean implements BankRemote {
    ...
    @PersistenceContext
    private EntityManager manager;

    public Account openAccount(String ownerName) {
        Account account = new Account();
        account.setName(ownerName);
        manager.persist(account);
        return account;
    }
    ...
    public void deposit(int accountNumber, int amount) {
        Account account = manager.find(Account.class, accountNumber);
        account.deposit(amount);
    }
    public void close(int accountNumber) {
        Account account = manager.find(Account.class, accountNumber);
        manager.remove(account);
    }
}
```

dependency injection

why do we have to find the entity in every method ?

Transaction boundaries

- After the `manager.persist(account)` call, the `account` entity is scheduled for being synchronized (written) to the database
- The entity will actually be written when the current transaction commits
- Until then, we say that the entity is in managed state

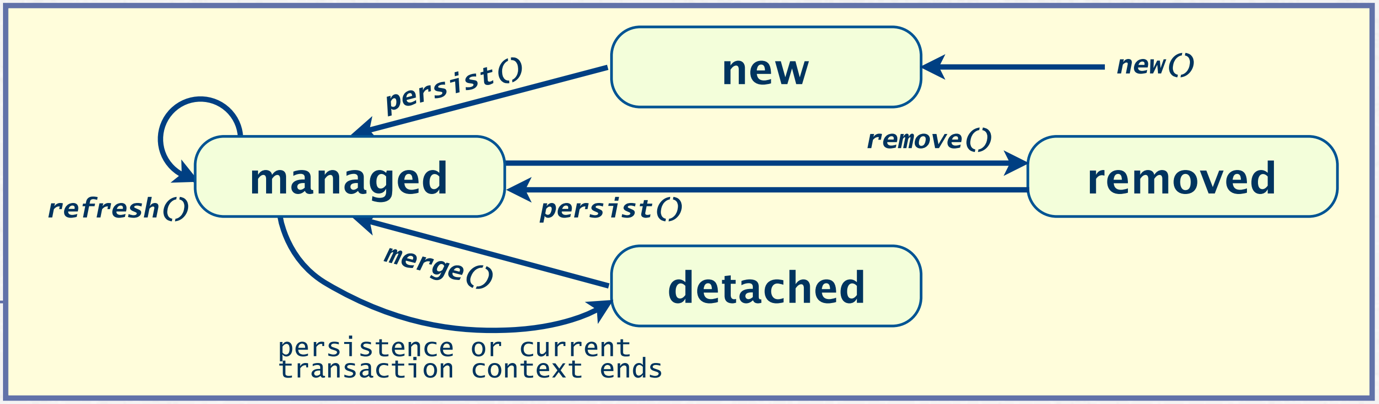
Entity possible states

new The entity was just created but is not yet bound to a persistent identity in the database or to a persistent context

managed The entity has a persistent identity in the database, is currently bound to a persistent context and is scheduled to be synchronized with the database.

detached The entity has a persistent identity but is not currently bound to a persistent context.

removed The entity is currently bound to a persistent context and scheduled for removal from the database.



Entity lifecycle callbacks

```
@Entity
@Table(name = "ACCOUNT")
public class Account {
    @PrePersist
    void prePersist()   { ... }

    @PostPersist
    void postPersist()  { ... }

    @PreRemove
    void preRemove()    { ... }

    ...

    ...

    @PostRemove
    void postRemove()   { ... }

    @PreUpdate
    void preUpdate()    { ... }

    @PostUpdate
    void postUpdate()   { ... }

    @PostLoad
    void postLoad()     { ... }
}
```

Entity lookup and queries

- Apart from the straightforward find-by-primary-key query, automatically managed via the `EntityManager.find()` method, we can perform more general queries to find entities
- This is done via the Query interface, another key element of the persistence provider API
- Queries are expressed using the Java Persistence Query Language (JP-QL), inspired from EJB-QL (EJB 2.1)
- JP-QL has a syntax similar to SQL but :
  - it manipulates objects rather than rows & columns
  - it is really portable across various implementations

Examples of queries

- □ Queries can either be dynamic or static
- □ Static queries are also known as named queries

dynamic query

```
@Stateless
@TransactionManagement(javax.ejb.TransactionManagementType.CONTAINER)
public class BankBean implements BankRemote {
    ...
    @PersistenceContext
    private EntityManager manager;

    public List<Account> listAccounts() {
      ➡Query query = manager.createQuery("SELECT a FROM Account a");
        return query.getResultList();
    }
}
```

named query

```
@Entity
@Table(name = "ACCOUNT")
@NamedQueries({
➡ @NamedQuery(name = "findByAcctnumber", query = "SELECT a FROM Account a WHERE a.acctnumber = :acctnumber"),
➡ @NamedQuery(name = "findByName",       query = "SELECT a FROM Account a WHERE a.name = :name"),
➡ @NamedQuery(name = "findByBalance",    query = "SELECT a FROM Account a WHERE a.balance = :balance")})
public class Account implements Serializable {
    ...
}
```

Extended persistent context

- □ Until now, we only saw transaction-scoped persistent contexts, i.e., ones that end when the enclosing transaction ends

- □ At this point, all entities in the persistent context become detached (from the database)

- □ Transaction-scoped persistent contexts are fine for stateless session beans, because the stateless bean cannot keep references to entities across method calls, and hence does a lookup prior to any entity manipulation

- □ For stateful session beans however, we need an extended persistent context, i.e., one where entities remain managed across methods calls

The session facade pattern

```
@Stateful
public class AccountBean implements AccountRemote {
    @PersistenceContext(type = PersistenceContextType.EXTENDED)
    private EntityManager manager;

    private Account account = null;

    public void open(int accountNumber) {
        account = manager.find(Account.class, accountNumber);
        if (account == null) {
            account = new Account();
            manager.persist(account);
        }
    }
    public void deposit(int amount) {
        if (account == null) throw new IllegalStateException();
        account.deposit(amount);
    }
    public String getName() {
        if (account == null) throw new IllegalStateException();
        return account.getName();
    }
    ...
}
```

This pattern consists in having a (remote) stateful session bean act as front-end for a non-remote entity

Persistence units

- Entities are packaged and deployed in persistence units
- A persistence unit is a logical grouping of entity classes, object-relational mapping metadata, and possibly database configuration information
- If there is more than one persistence units in an application, we need to explicitly reference it in the @PersistenceContext annotation

```
@Stateful
public class AccountBean implements AccountRemote {

    private Account account = null;
    @PersistenceContext(type = PersistenceContextType.EXTENDED, unitName = "Banking")
    private EntityManager manager;
    ...
}
```

Asynchronous invocations (1)

- □ A session bean can implement asynchronous methods, in order to increase throughput and response time, typically in the case of processor-intensive computation
- □ With an asynchronous method, the container returns the control to the client before the method is actually invoked and executes it in the background (asynchronously)
- □ An asynchronous method must return void or a Future<V> object; if it returns void it cannot declare exceptions
- □ The client can use the Future<V> object to retrieve the actual result or to cancel the invocation

Asynchronous invocations (2)

```
@Remote
public interface PortfolioRemote {
  ...
  public Future<Double> computeValue();
}
```

```
@Stateful
public class Portfolio implements PortfolioRemote {
  @Resource
  SessionContext context;
  ...
  @Asynchronous
  public Future<Double> computeValue() {
    double value = ...;  // Processor-intensive computation
    return new AsyncResult<Double>(value);
  }
}
```

Asynchronous invocations (3)

```
Future<Double> value = myPortfolio.computeValue();
...   // Some time goes by...
System.out.println("Portfolio is worth $" + value.get());
```

```
Future<Double> value = myPortfolio.computeValue();
try {
  System.out.println("Portfolio is worth $" + value.get(5, TimeUnit.SECONDS));
} catch (TimeoutException ex) {
  value.cancel(true);
  System.err.println("Timeout: operation was cancelled");
}
```

```
@Asynchronous
public Future<Double> computeValue() {
  if (context.wasCancelCalled()) {
    System.err.println("Call to computeValue() was cancelled");
    return null;
  }
  double value = ...;  // Processor-intensive computation
  return new AsyncResult<Double>(value);
}
```