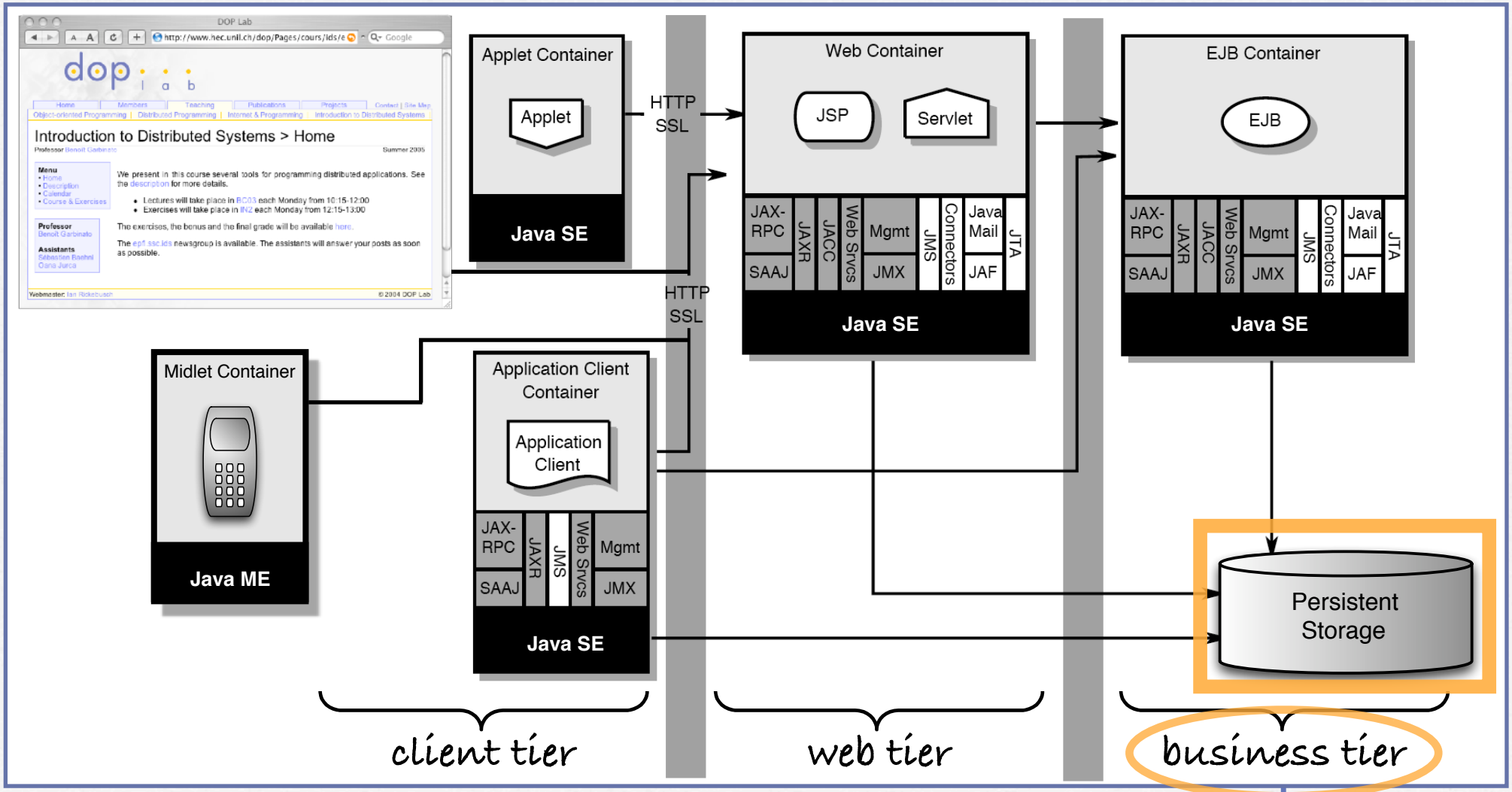
Business Tier
Data Persistence

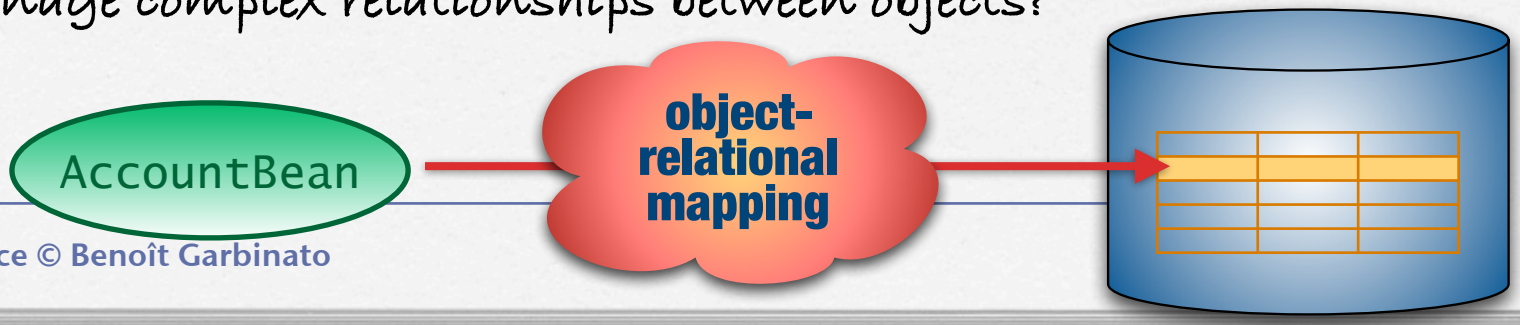
Unil | HEC | dop lab
Benoît Garbinato
distributed object programming lab

The EJB model

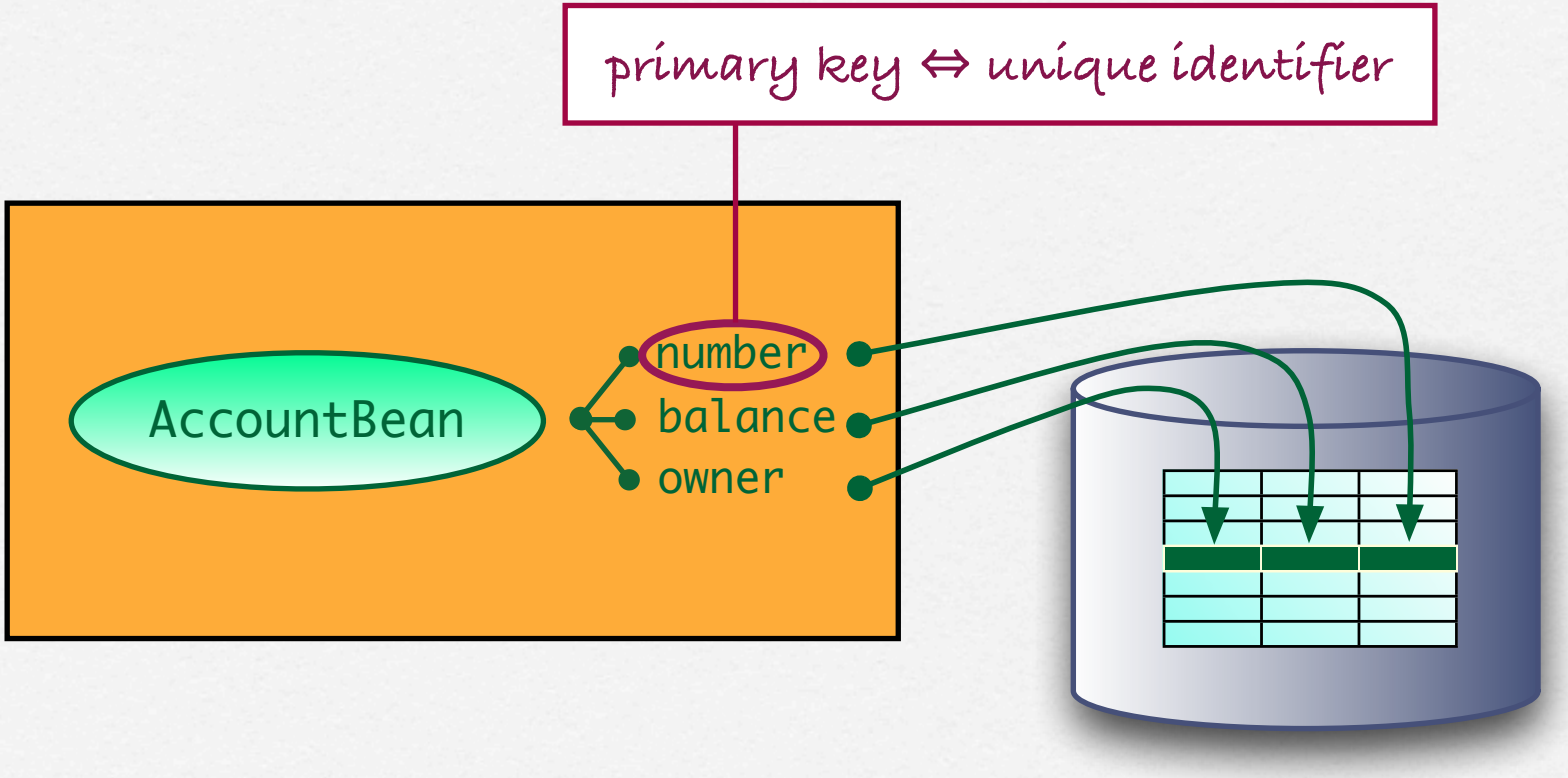


Persisting objects

- 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

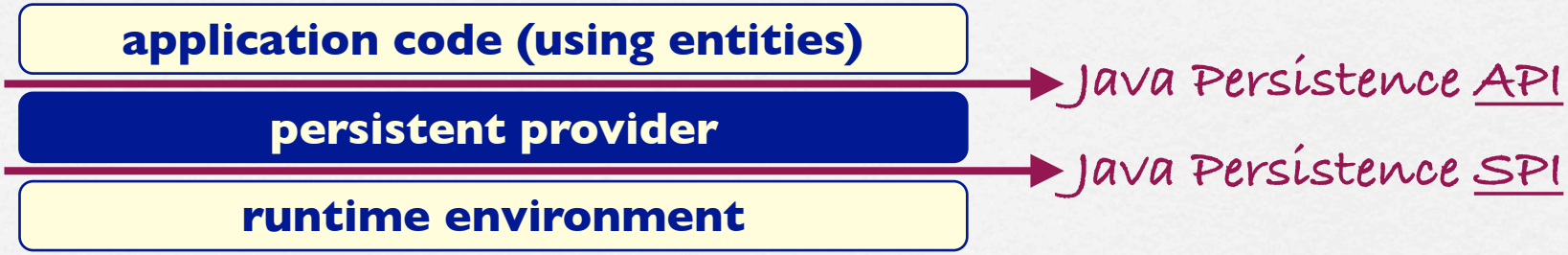
entities ≠ entity beans

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

```
@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;
    }
    ...
```

```
    ...

    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;
        }
    }
}
```

```
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)

dependency injection

```
@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);
    }
}
```

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 <u>managed</u> state

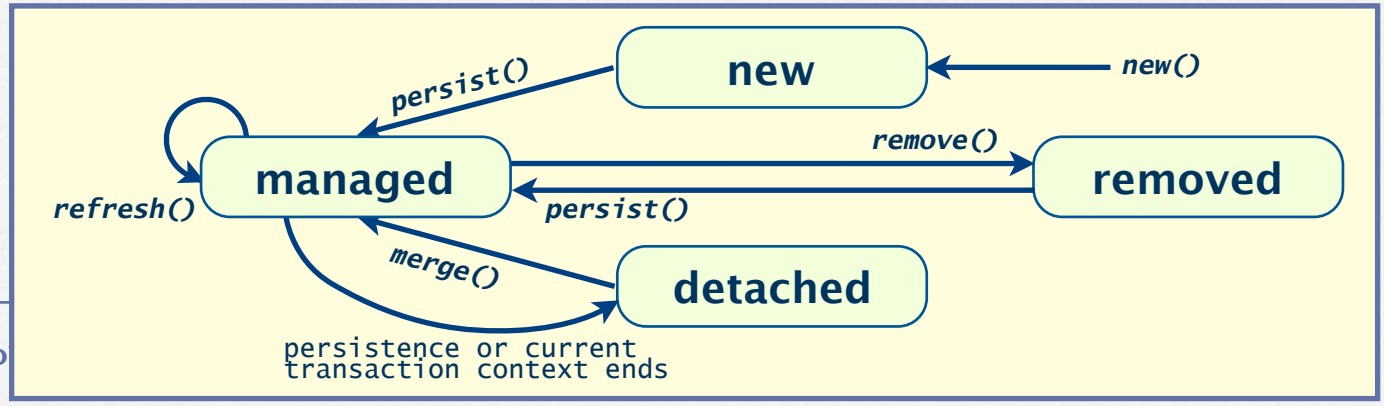
Entity possible states

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;
    ...
}
```