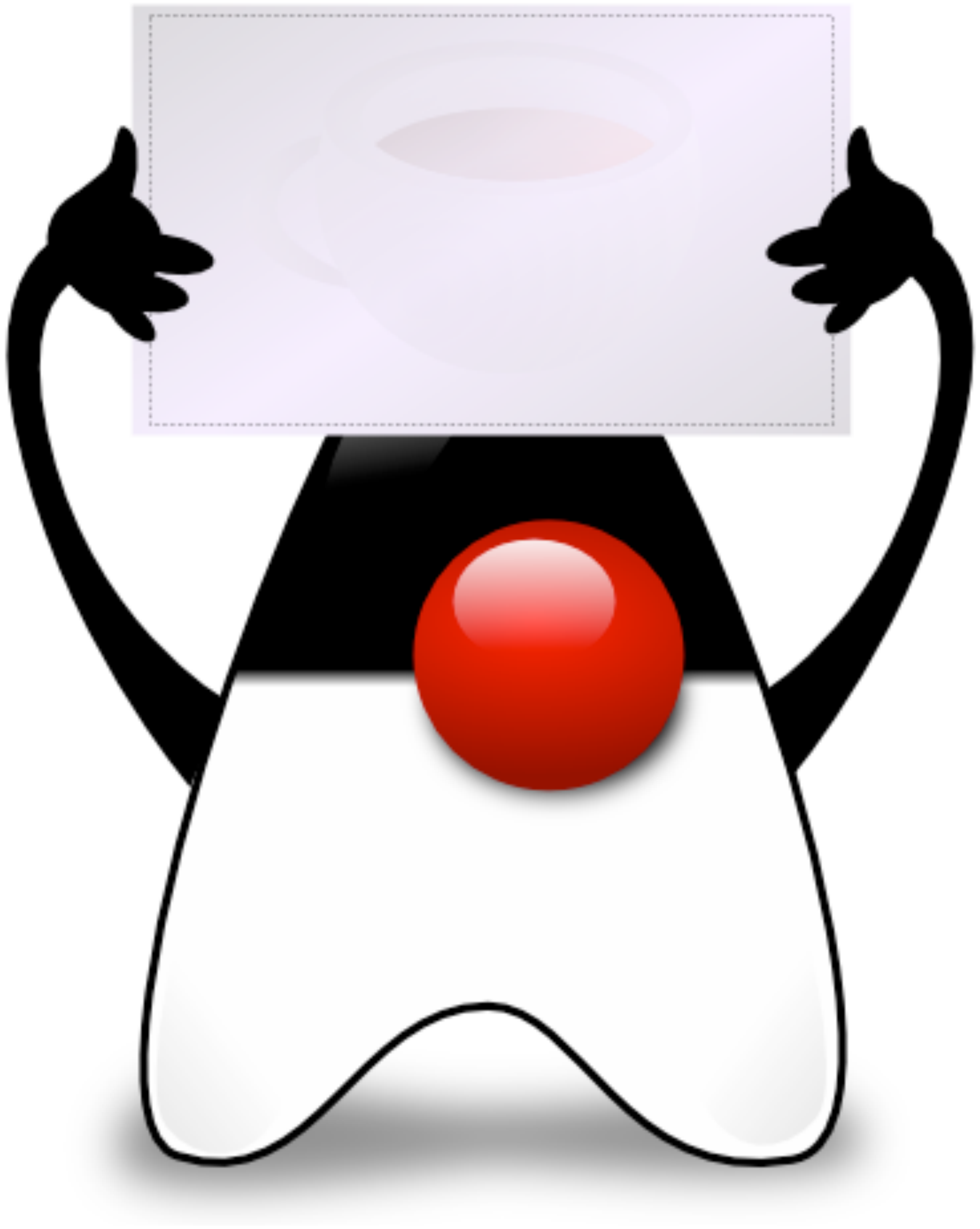
# Lecture 5: Transactions

Olivier Liechti
AMT

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



# Transactions

# Transactions

Imagine that you have the following code in a business service:

```
accountA.debit(100);
accountB.credit(100);
```

What happens is the application crashes here?
Is my data corrupted?
Has money vanished in cyberspace?

# Transactions

```
transaction.start();

accountA.debit(100);
accountB.credit(100);

transaction.commit();
```

Transactions give us an “whole or nothing” semantic
(we often speak about a unit of work)

# Transactions

```
transaction.start();
accountA.debit(100);
try {
  accountB.credit(100);
} catch (AccountFullException e) {
  transaction.rollback();
}
transaction.commit();
```

We can also deal with application-level errors and leave the data in a consistent state.

# Transactions

## ACID

# Transactions

## ACID

Atomicity: “all or noting”

# Transactions

## ACID

Consistency: “business data integrity”

# Transactions

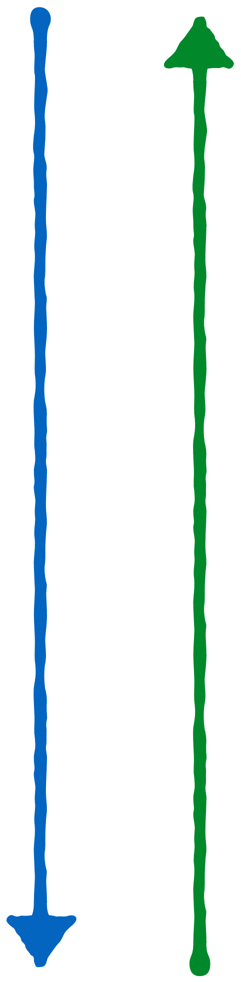
# ACID

Isolation: “deal with concurrent transactions”

*There are different isolation levels!*

# Isolation levels

Increasing isolation between transactions



Increasing performance in the cas of concurrent access

| Isolation level | Potential issues |
|---|---|
| **Read Uncommitted**<br>(no locks) | **Dirty Reads**<br>(no isolation) |
| **Read Committed**<br>(write locks) | **Non-repeatable Reads** |
| **Repeatable Reads**<br>(read & write locks) | **Phantom reads** |
| **Serializable**<br>(range locks) | |

# Isolation levels

- "A **dirty read** occurs when a transaction is allowed to read data from a row that has been modified by another running transaction and not yet committed."
- "A **non-repeatable read** occurs, when during the course of a transaction, a row is retrieved twice and the values within the row differ between reads."
- "A **phantom read** occurs when, in the course of a transaction, two identical (SELECT) queries are executed, and the **collection** of rows returned by the second query is different from the first."

See for **example scenarios**, see:

https://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html#transactions_data_integrity

https://en.wikipedia.org/wiki/Isolation_(database_systems)#Read_phenomena

# Transactions

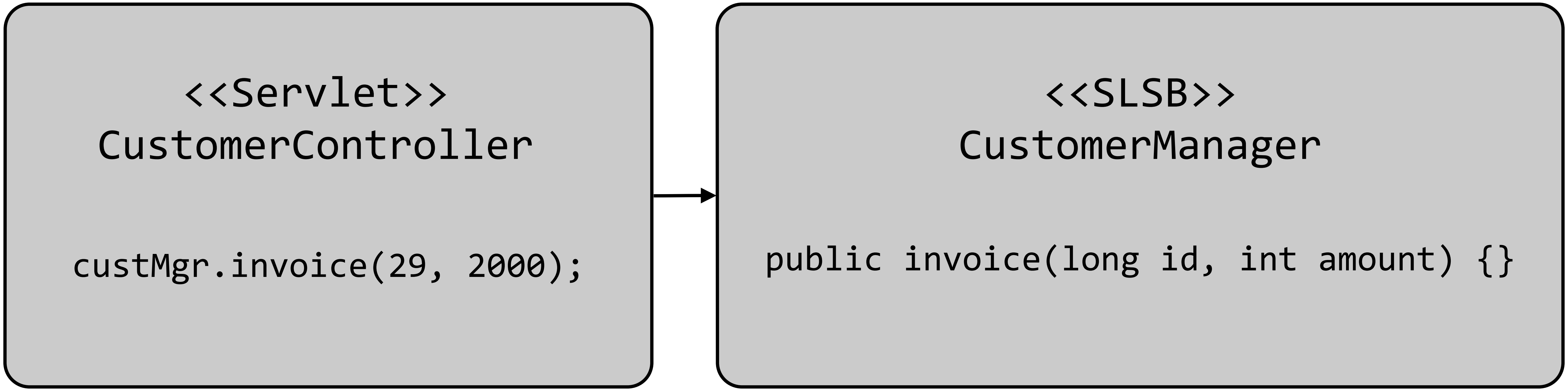
# ACID

Durability: “once it’s done, it’s done”

# Transactions & EJBs

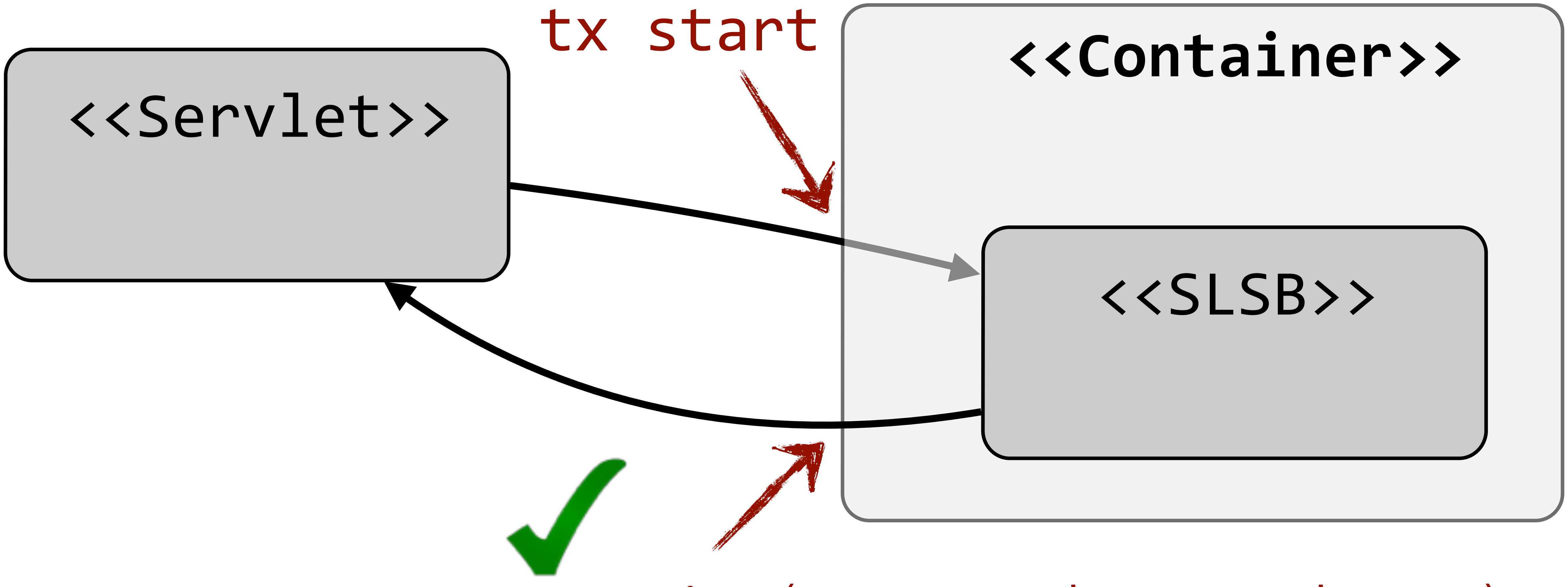
heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- By default, the EJB container handles calls to `commit` and `rollback`.
- Methods defined on EJBs provide demarcation points.
- This is the **default behavior**.



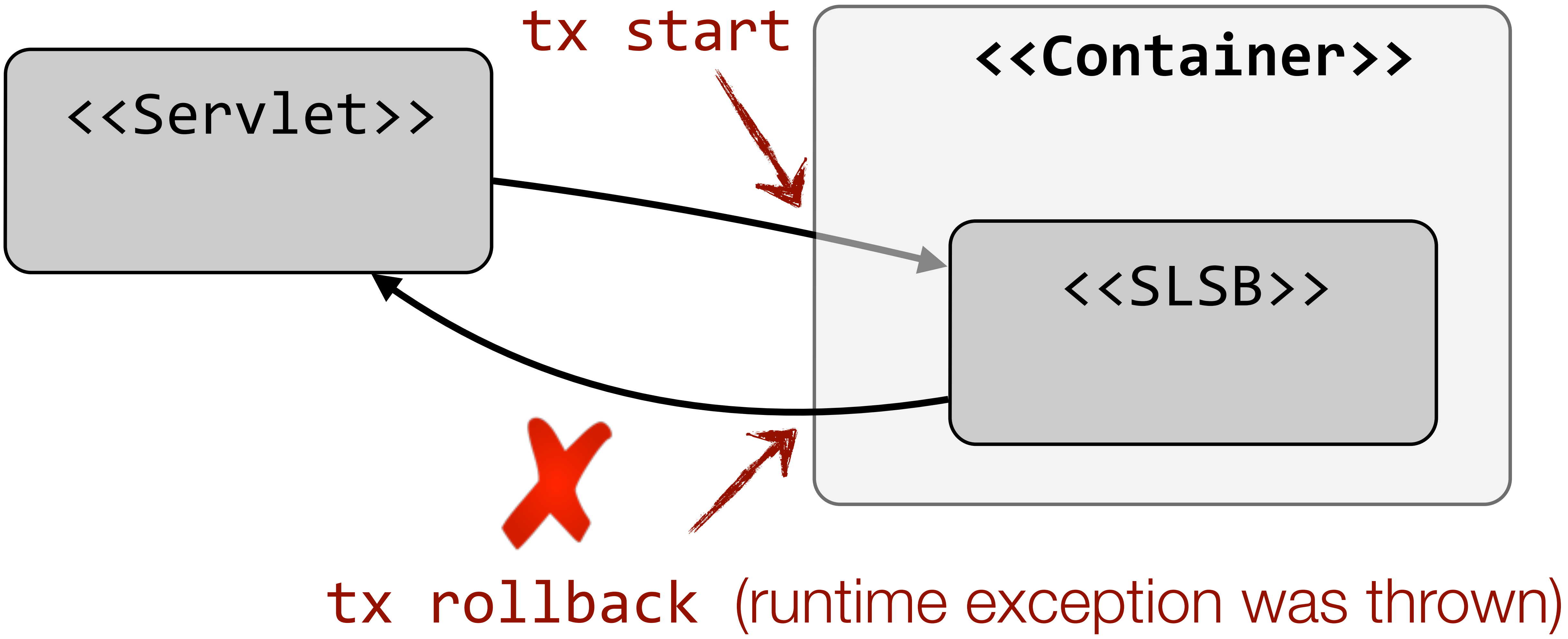
<<Servlet>>
CustomerController
custMgr.invoice(29, 2000);
<<SLSB>>
CustomerManager
public invoice(long id, int amount) {}

# Transactions & EJBs



tx commit (no exception was thrown)

# Transactions & EJBs



tx start

<<Container>>

<<Servlet>>

<<SLSB>>

tx rollback (runtime exception was thrown)

# Transaction Scope

What happens when a **client** calls a method on a session bean,

which calls a method on a session bean,
which calls a method on a session bean,
which calls a method on a session bean,
which calls a method on a session bean,
which calls a method on a session bean,
which calls a method on a session bean,

which **throws an exception**?

# Transaction Scope

What happens when a **client** calls a method on a sess

session bean,

whic on a session bean,

which calls a method on a session bean

which calls a method on a

which calls a method o

which calls a method o

which **throws an exception**?

*Opinion1*

**Everything** should be rolled back!

*Opinion2*

**No!** Only changes incurred by the last method should be rolled back!

# Transaction Scope

What happens when a **client** calls a method on a

sess

whi

which calls a method

which calls a method on a

which calls a method o

which calls a method o

which **throws an exception**?

*Opinion1*

**Everything** should

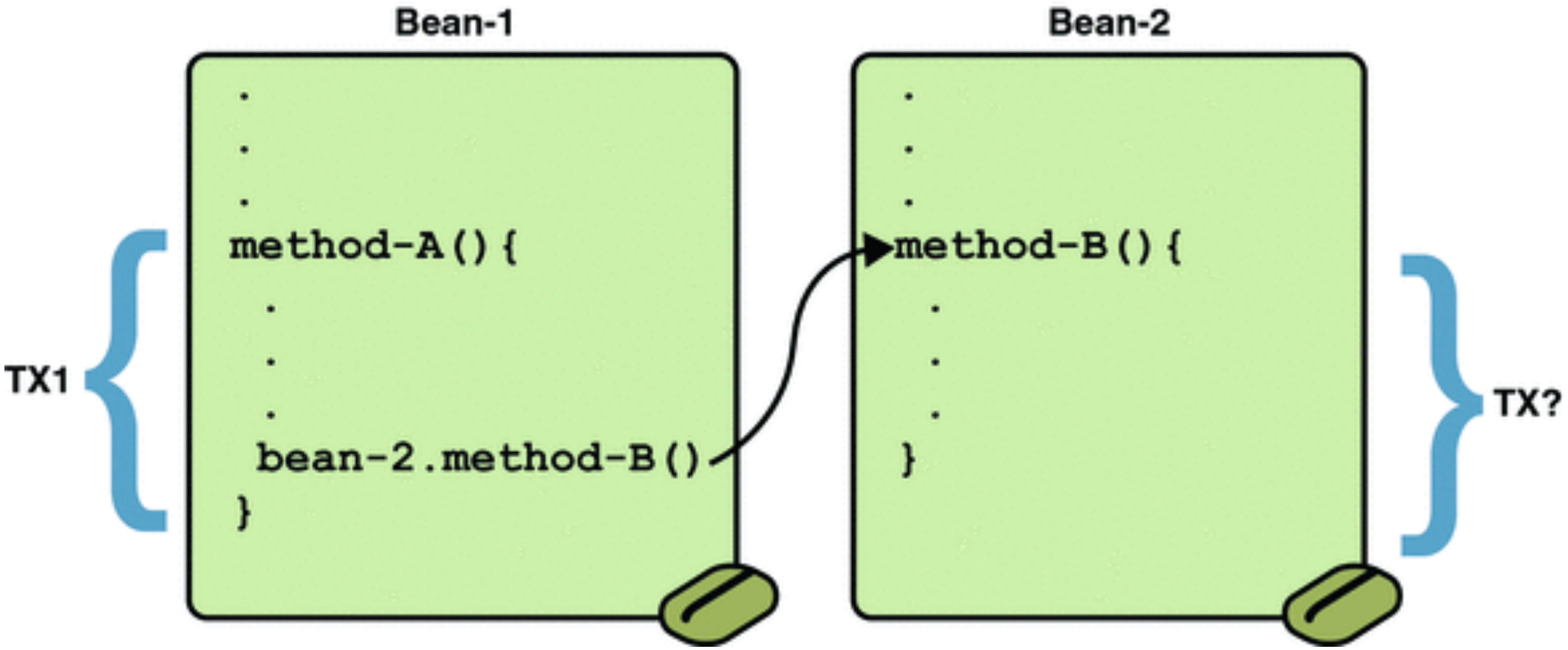
rolled back!

It is **up to the application** to specify intended behavior. The developer must specify transaction scope, typically with **annotations**.

*Opinion2*

**No!** Only changes incurred by the last method should be rolled back!

# Transaction Scope



http://java.sun.com/javaee/5/docs/tutorial/doc/bncij.html

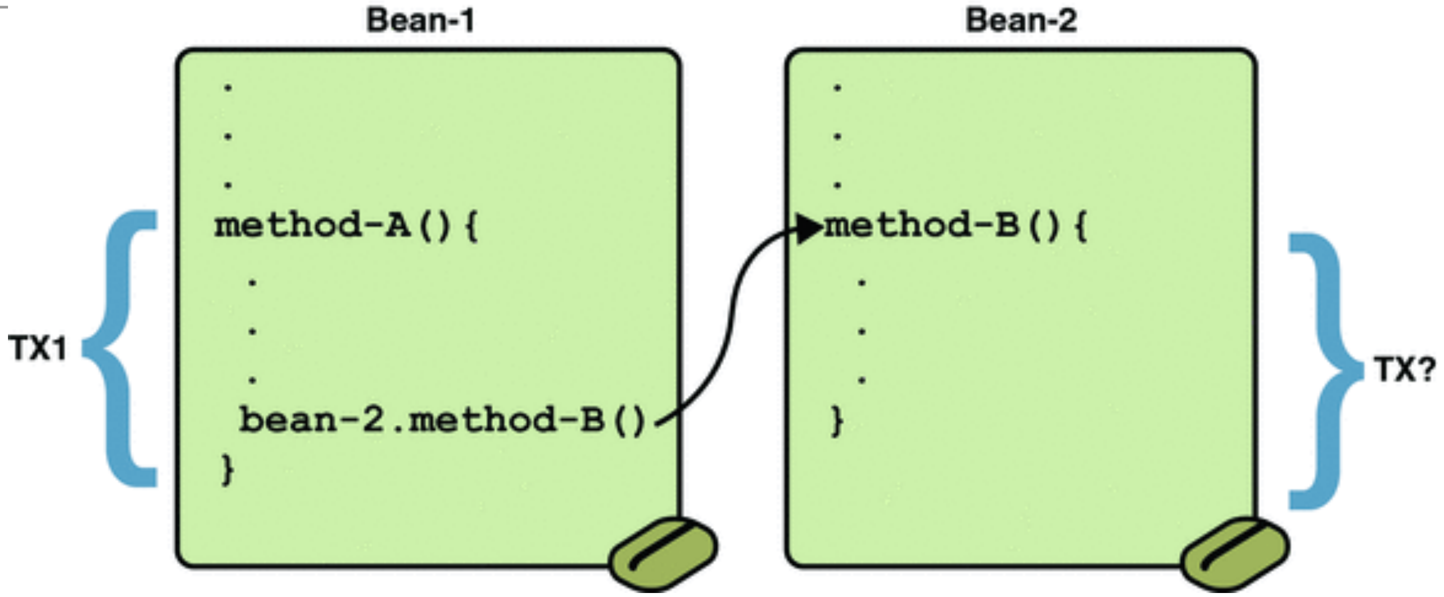
# Transaction Scope

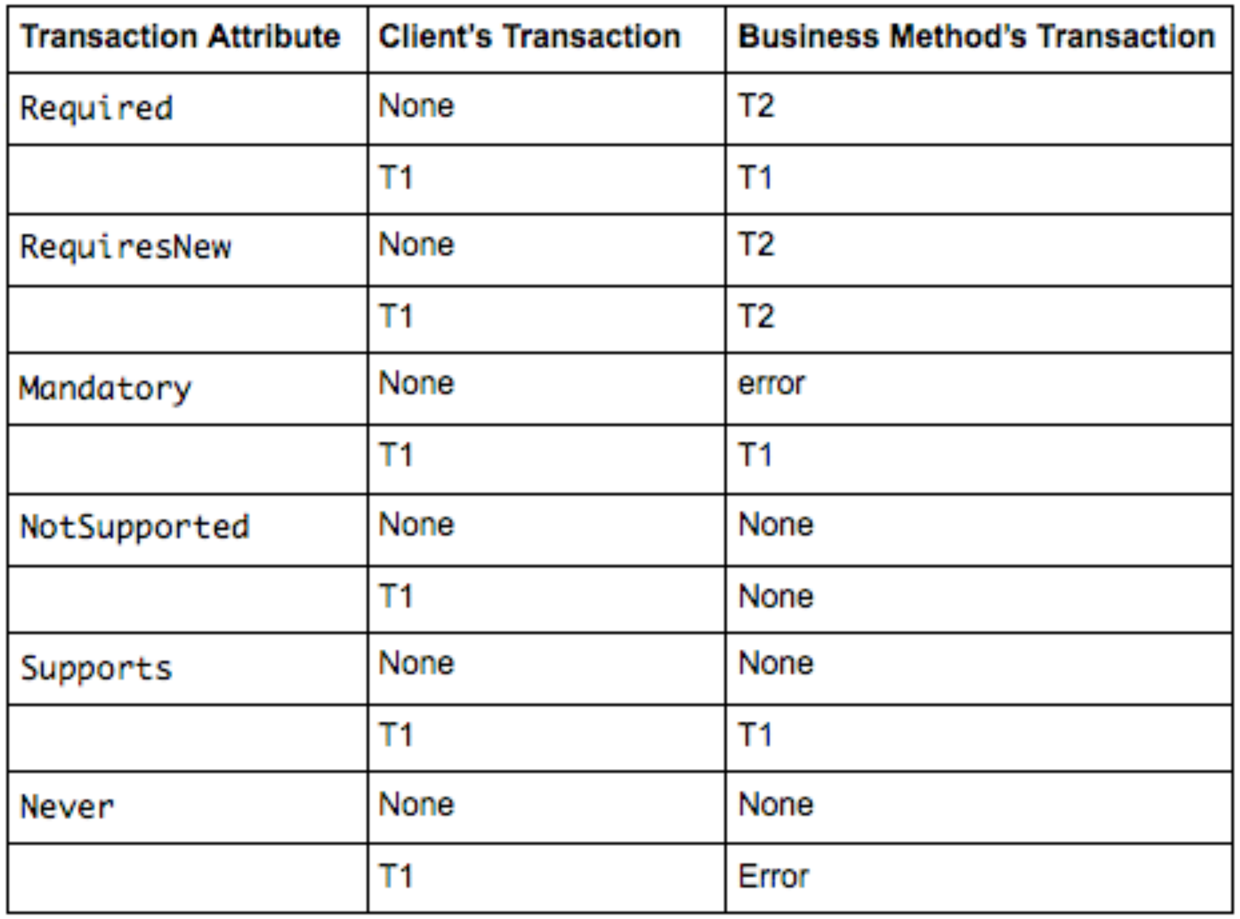
```
@TransactionAttribute(NOT_SUPPORTED)
@Stateless
public class TransactionBean implements
Transaction {
...
    @TransactionAttribute(REQUIRES_NEW)
    public void firstMethod() {...}

    @TransactionAttribute(REQUIRED)
    public void secondMethod() {...}

    public void thirdMethod() {...}

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





| Transaction Attribute | Client's Transaction | Business Method's Transaction |
|---|---|---|
| Required | None | T2 |
| | T1 | T1 |
| RequiresNew | None | T2 |
| | T1 | T2 |
| Mandatory | None | error |
| | T1 | T1 |
| NotSupported | None | None |
| | T1 | None |
| Supports | None | None |
| | T1 | T1 |
| Never | None | None |
| | T1 | Error |

# Transactions & Exceptions

- There are **two ways to roll back a container-managed transaction**
- Firstly, if a **system exception** is thrown, the container will automatically roll back the transaction.
- Secondly, by invoking the **setRollbackOnly** method of the EJBContext interface, the bean method instructs the container to roll back the transaction.
- If the bean throws an **application exception**, the rollback is not automatic but can be initiated by a call to **setRollbackOnly**.
- Note: you can also annotate your Exception class with **@ApplicationException(rollback=true)**

# Transaction scope & JPA

```
@Stateless
public class CarService {

  @PersistenceContext
  EntityManager em;

  @EJB
  PartsService partsService;

  public Car buildCar() {
    Engine e = getEngine();
    SapinVanille s = getSapin();
    Car c = new Car(e, s);
    em.persist(c);
  }

}
```

What happens if there is strike and a **NullPointerException** is thrown in the constructor?

```
@Stateless
public class PartsService {

  @PersistenceContext
  EntityManager em;

  public Engine getEngine() {
    Engine e = new Engine();
    em.persist(e);
    return e;
  }

  public SapinVanille getSapin() {
    SapinVanille s = new SapinVanille();
    em.persist(s);
    return(s);
  }

}
```

What happens if there is a shortage of vanilla and a **NullPointerException** is thrown in the constructor?

# Transaction scope & JPA

- The default transaction scope for EJB methods is “REQUIRED”. This means that we will have one single transaction for the whole process (and one JPA persistence context).

- **Scenario 1**: no exception thrown. The 3 rows will be inserted when the container commits.

| T1 | Persistence Context |
|---|---|
| [CarService]<br>  Engine e = getEngine(); | **PC[t1]= {}** |
| [PartsService]<br>  Engine e = new Engine();<br>  em.persist(e); | **PC[t1]= {e}** |
| [CarService]<br>  SapinVanille s = getSapin(); | **PC[t1]= {e}** |
| [PartsService]<br>    SapinVanille s = new SapinVanille();<br>    em.persist(s); | **PC[t1]= {e, s}** |
| [CarService]<br>    Car c = new Car(e, s);<br>    em.persist(c); | **PC[t1]= {e, s, c}** |

# Transaction scope & JPA

- The default transaction scope for EJB methods is "REQUIRED". This means that we will have one single transaction for the whole process (and one JPA persistence context).

- **Scenario 2**: an exception is thrown in the SapinVanille constructor. No row is added to the database (not even the engine which was successfully persisted).

| T1 | Persistence Context |
|---|---|
| [CarService]<br>Engine e = getEngine(); | **PC[t1]= {}** |
| [PartsService]<br>Engine e = new Engine();<br>em.persist(e); | **PC[t1]= {e}** |
| [CarService]<br>SapinVanille s = getSapin(); | **PC[t1]= {e}** |
| [PartsService]<br>SapinVanille s = new SapinVanille();<br>NullPointerException is thrown | **PC[t1]= {e}** |
| Transaction is rolled back by the container | |

# Transaction scope & JPA

- Vanilla shortage should not block the production line!
- The Car constructor is ok with a null value anyway. Let's execute the getSapin() method in its own transaction.

```
@Stateless
public class CarService {

  @PersistenceContext
  EntityManager em;

  @EJB
  PartsService partsService;

  public Car buildCar() {
    Engine e = getEngine();
    try {
      SapinVanille s = getSapin();
    } catch (Exception e) {
      logException(e);
    }
    Car c = new Car(e, s);
    em.persist(c);
  }

}
```

```
@Stateless
public class PartsService {

  @PersistenceContext
  EntityManager em;

  public Engine getEngine() {
    Engine e = new Engine();
    em.persist(e);
    return e;
  }

  @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
  public SapinVanille getSapin() {
    SapinVanille s = new SapinVanille();
    em.persist(s);
    return(s);
  }
}
```

# Transaction scope & JPA

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- **Scenario 1**: no exception thrown. 1 row is committed in the SapinVanille table first, 2 rows are committed in the Engine and Car tables later.

**s** is not a managed entity

| T1 | T2 | Persistence Context |
|---|---|---|
| [CarService]<br>Engine e = getEngine(); | | **PC[t1]= {}** |
| [PartsService]<br>Engine e = new Engine();<br>em.persist(e); | | **PC[t1]= {e}** |
| [CarService]<br>SapinVanille s = getSapin(); | | **PC[t1]= {e}** |
| | [PartsService]<br>SapinVanille s = new SapinVanille();<br>em.persist(s); | **PC[t2]= {s}** |
| | The container commits T2 | |
| [CarService]<br>Car c = new Car(e, s);<br>em.persist(c); | | **PC[t1]= {e, c}** |
| | The container commits T1 | |

# Transaction scope & JPA

- **Scenario 2**: an exception is thrown in the SapinVanille constructor. No row is committed in the SapinVanille table, BUT 2 rows are committed in the Engine and Car tables!

| T1 | T2 | Persistence Context |
|---|---|---|
| `[CarService]`<br>`Engine e = getEngine();` | | **`PC[t1]= {}`** |
| `[PartsService]`<br>`Engine e = new Engine();`<br>`em.persist(e);` | | **`PC[t1]= {e}`** |
| `[CarService]`<br>`SapinVanille s = getSapin();` | | **`PC[t1]= {e}`** |
| | `[PartsService]`<br>`SapinVanille s = new SapinVanille();`<br>`NullPointerException is thrown` | **`PC[t2]= {}`** |
| | `The container rollbacks T2` | |
| `[CarService]`<br>`Car c = new Car(e, s);`<br>`em.persist(c);` | | **`PC[t1]= {e, c}`** |
| | `The container commits T1` | |

# Transactions & concurrency control

- If several transactions are processed **concurrently**, unexpected results may occur. There are different strategies and mechanisms for dealing with that.

```
@Stateless
public class TransactionProcessor {

  @EJB
  AccountDAO accountDao;

  public void processTransaction(Transaction t) {
    Account a = accountDao.findById(t.getAccountId());
    long previousBalance = a.getBalance();
    a.setBalance(previousBalance + t.getAmount();
  }

}
```

What happens if another transaction modifies the account balance between these two statements?

# Optimistic concurrency control

- In many applications, there is a **high ratio of "read to write" operations** (many transactions read data, few update data). Moreover, there is a "small" likelihood that two concurrent transactions try to update the same data.
- In this case, for performance and scalability reasons, it is often recommended to implement an optimistic concurrency control mechanism.
- The mechanism works as follows:
  - When a program **reads** a record, it gets its "**version number**" (the number of previous updates) in a table column.
  - When it **updates** this record, it makes sure that the version number has not been incremented (this would indicate a conflict with another transaction).
- The developer has to write the logic to execute when a conflict is notified (retry, notify the user, etc.)

https://blogs.oracle.com/enterprisetechtips/entry/locking_and_concurrency_in_java

# Optimistic concurrency control with JPA

- **JPA supports optimistic concurrency control.**
- To use it, the first step is to annotate one field of the entity with the **@version** annotation. JPA will ensure that this value is incremented with every update.
- The second step is to catch the **OptimisticLockException** that may be thrown by JPA when the transaction commits.
- This is where the developer specifies what to do if a conflict has been detected. In some cases, it is possible to immediately and silently retry the transaction.

```
@Entity
public class Account {

  @Id
  long accountId;

  @Version
  long version;
}
```

# Pessimistic concurrency control

- When an optimistic concurrency control is not appropriate, then it is possible to implement **pessimistic concurrency control with locks**.
- RDBMS support different types of locks (read lock, write lock).
- When a transaction obtains a **read lock** on a record, it cannot be modified by other transactions. However, it can be read by other transactions.
- When a transaction obtains a **write lock** on a record, it cannot be modified, nor read by other transactions.
- Locking database records **introduce issues**: scalability, performance, deadlocks. It can be tricky to decide when to obtain a lock and for how long.

https://blogs.oracle.com/enterprisetechtips/entry/locking_and_concurrency_in_java

# Pessimistic concurrency control with JPA

- **JPA supports pessimistic concurrency control since version 2.0**
- It is possible to **lock a record** with **em.lock**(entity, LOCK_TYPE).
- It is also possible to lock a record at the time of retrieval with **em.find**(class, id, LOCK_TYPE)

```
Account a = em.find(Account.class, id);
em.lock(a, PESSIMISTIC_WRITE);
```

Be aware that we still have a risk of stale data here!

```
Account a = em.find(Account.class, id, PESSIMISTIC_WRITE);
```