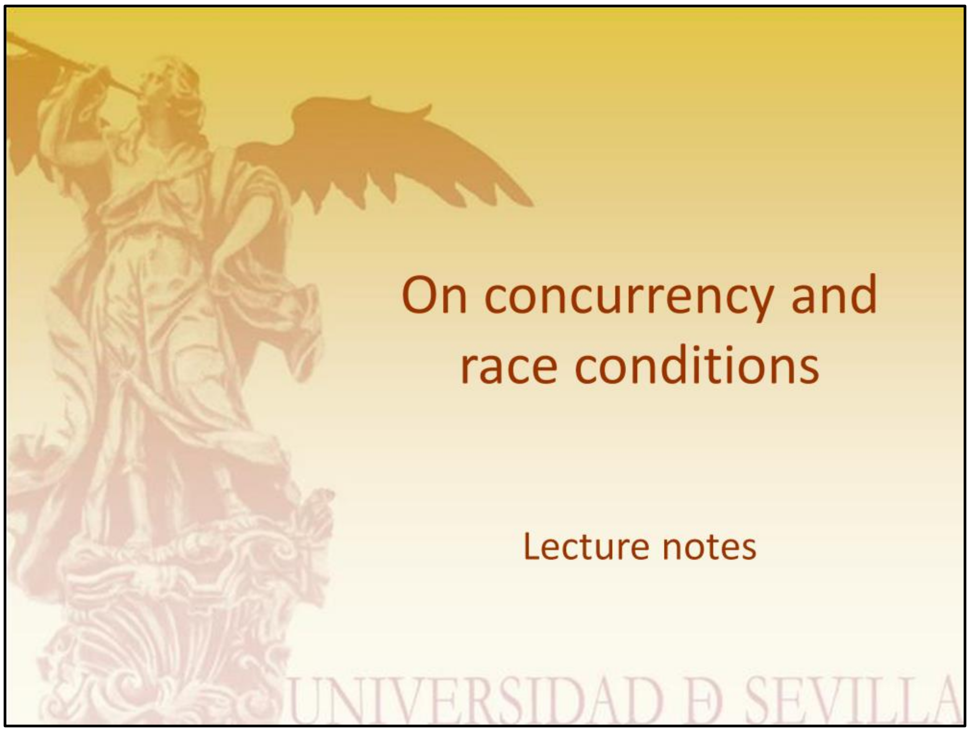


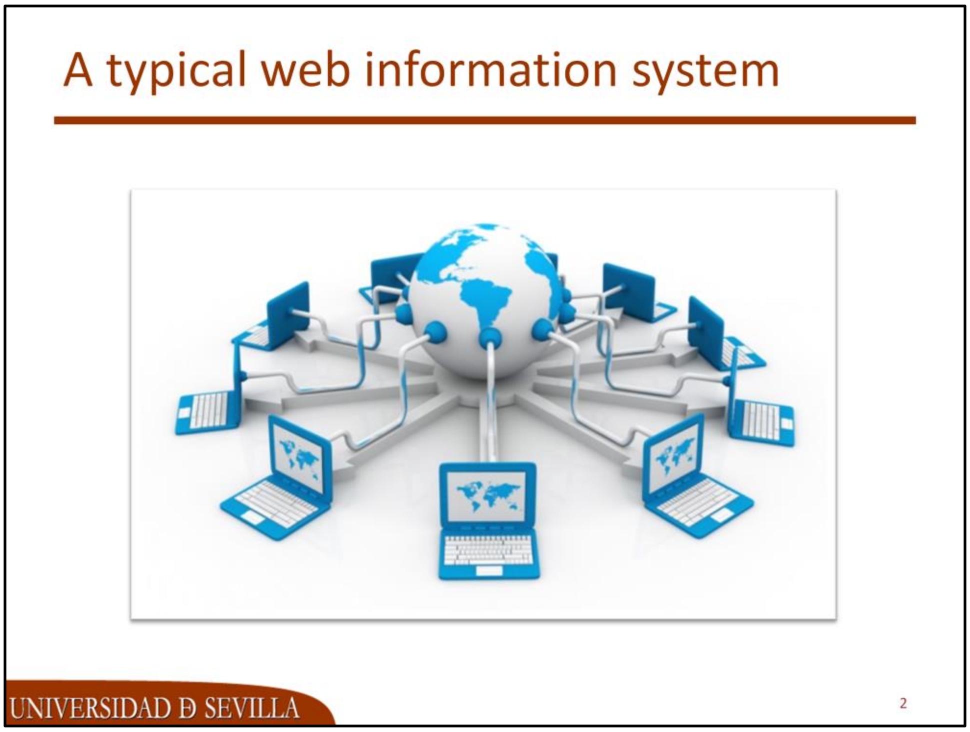
On concurrency and race conditions

Lecture notes

UNIVERSIDAD Ð SEVILLA

These notes are a complement to the theory lecture. Race conditions are a common problem in concurrent systems. These notes are intended to clarify how they can be avoided with the technology that we're using.

--

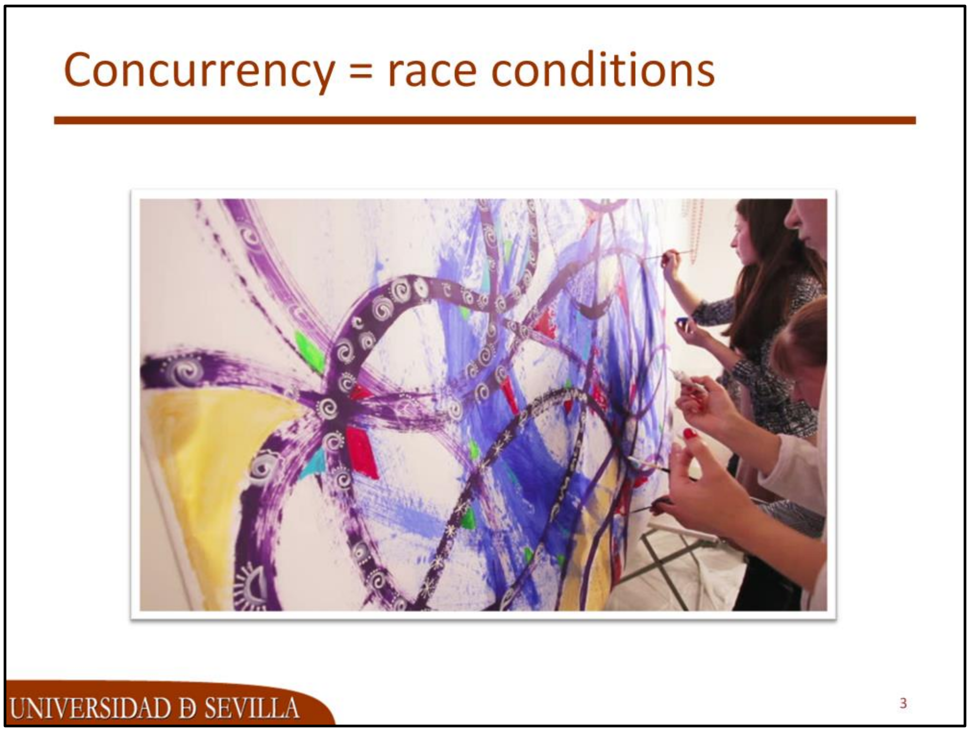


A typical web information system

UNIVERSIDAD D SEVILLA

2

This slide illustrates a typical web information system, which serves dozens if not hundreds or thousands of users concurrently.

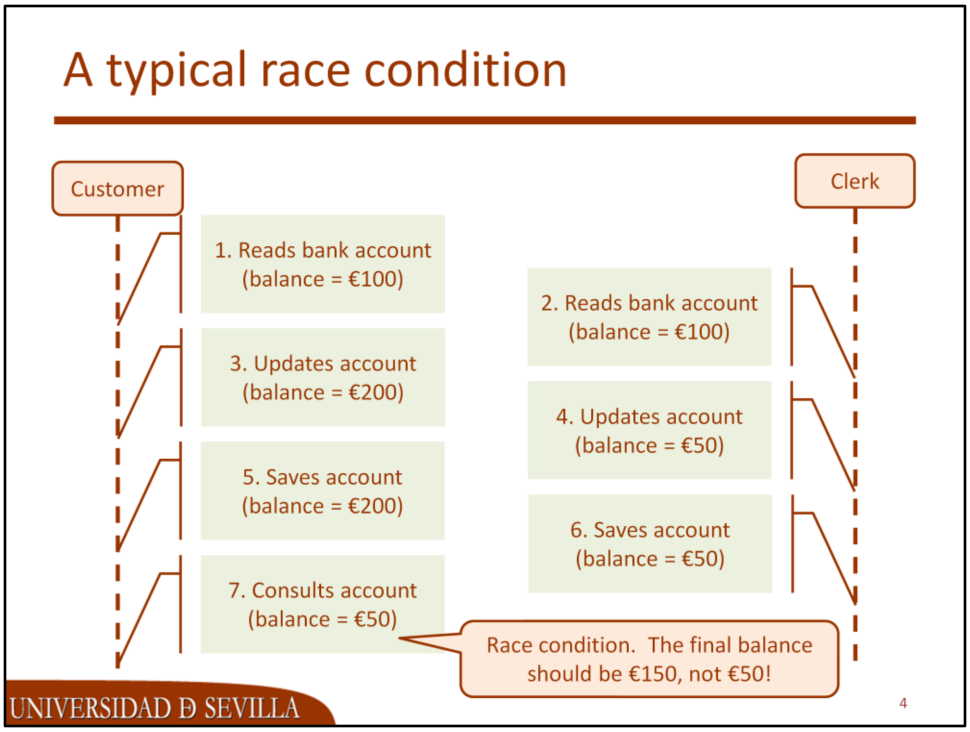


Concurrency = race conditions

UNIVERSIDAD Ð SEVILLA

3

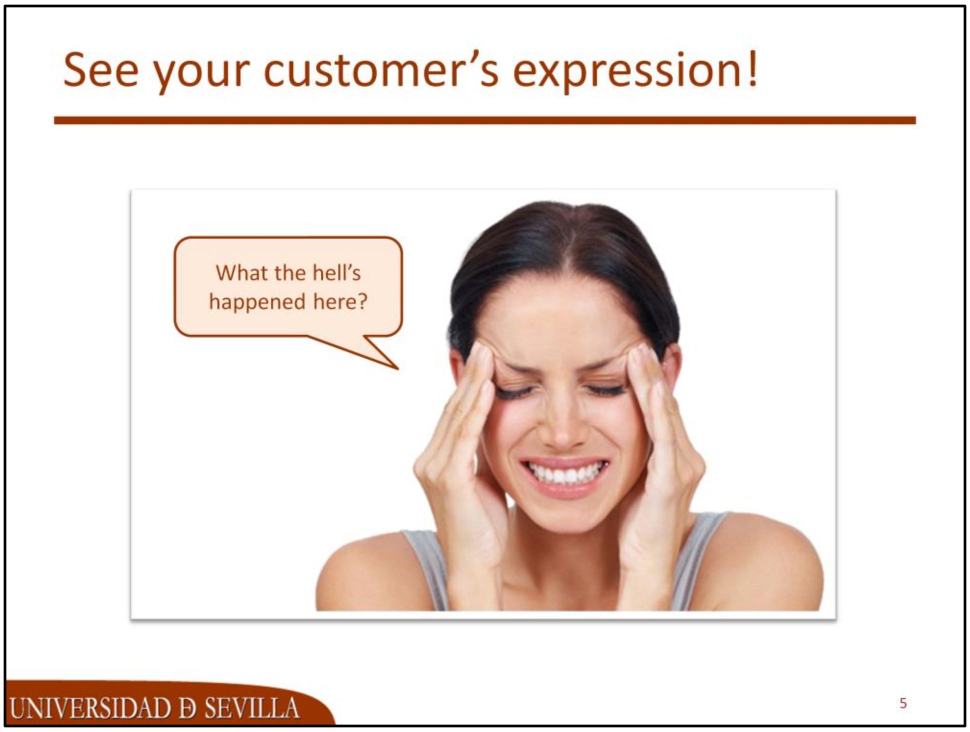
Unfortunately, concurrency means trouble if two users update the same entities more or less at the same time and this leads to a race condition.



A typical race condition

Customer
Clerk
1. Reads bank account (balance = €100)
2. Reads bank account (balance = €100)
3. Updates account (balance = €200)
4. Updates account (balance = €50)
5. Saves account (balance = €200)
6. Saves account (balance = €50)
7. Consults account (balance = €50)
Race condition. The final balance should be €150, not €50!
UNIVERSIDAD Ð SEVILLA
4

Let’s illustrate what a race condition is with a standard example: there are a customer and a clerk who are working concurrently on the same bank account: first, the customer reads the bank account, whose balance is €100; then the clerk reads it (note that the account is the same in the database, but the reading operations result in two different objects); then the customer pays in €100 and updates the balance to €200; then the clerk pays an invoice and withdraws €50 and updates the balance to €50 (recall that the customer and the clerk are working with different objects in memory); then the customer saves his or her account object (with a balance of €200); then the clerk saves his or her account object (with a balance of €50).



See your customer's expression!

UNIVERSIDAD Ð SEVILLA

5

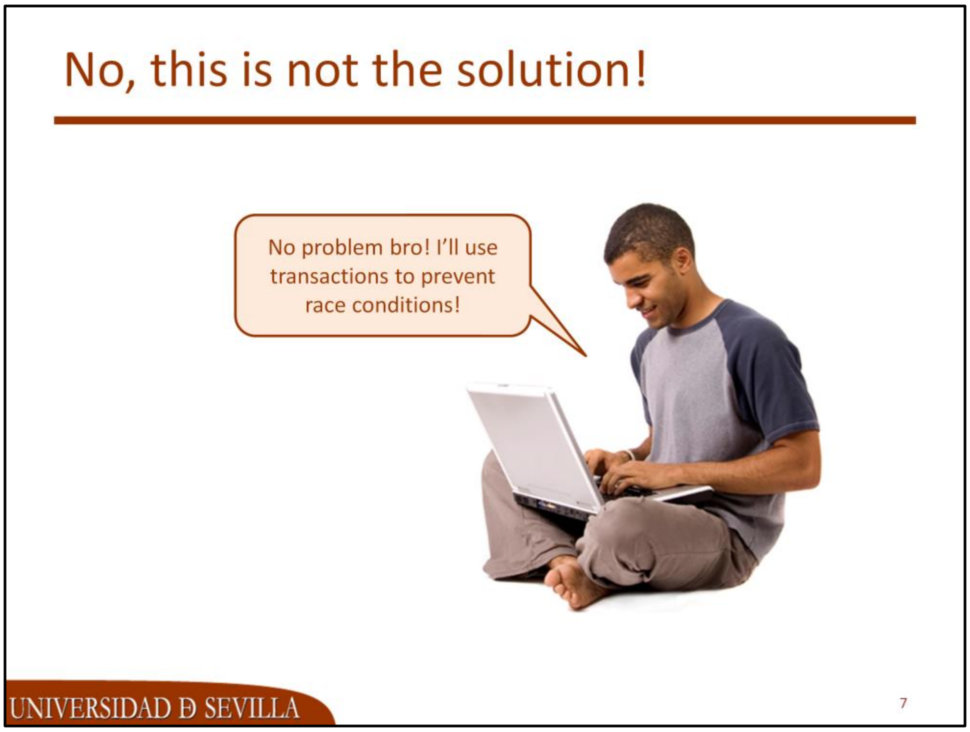
Your customer expected the balance to be €200, but it's €50. Note that he or she might jump in surprise if he or she saw €150, but he or she would soon realise that an invoice was paid concurrently; he or she wouldn't be happy, but he or she wouldn't complain. But the balance is €50 and that doesn't make sense at all! More than that: there's no record of his or her paying in €100!

This is a good question!

UNIVERSIDAD Ð SEVILLA

6

This is a good question. How can race conditions be avoided? Keep reading, please.



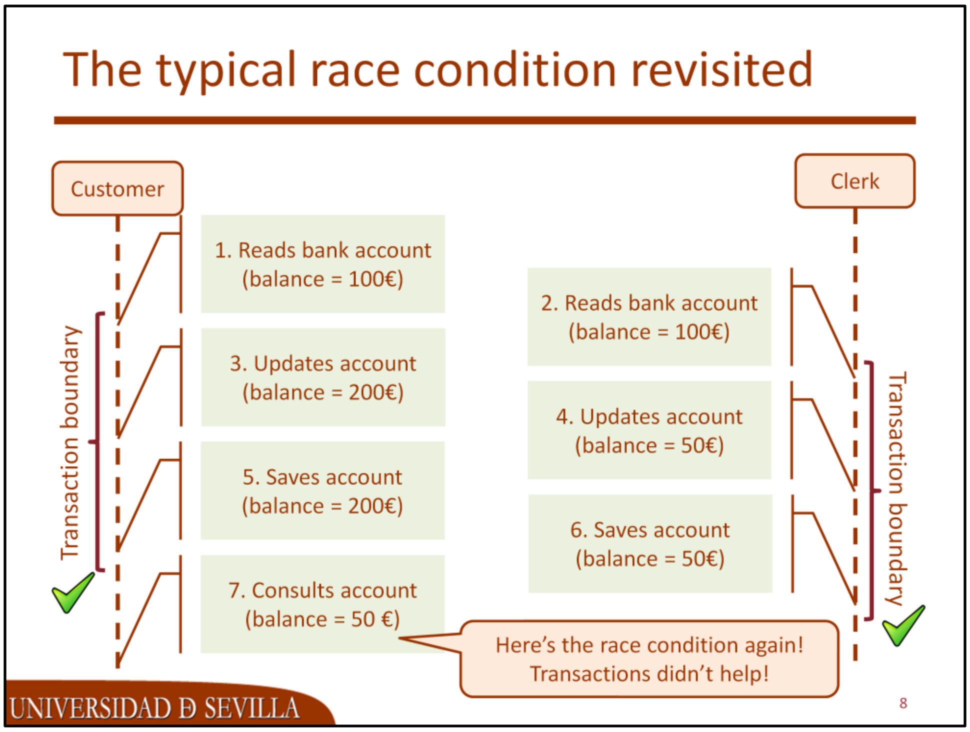
No, this is not the solution!

No problem bro! I'll use transactions to prevent race conditions!

UNIVERSIDAD D SEVILLA

7

Unfortunately, our students think that making their services transactional is enough to avoid race conditions. But it is not!



The typical race condition revisited

Customer
Clerk
1. Reads bank account (balance = 100€)
2. Reads bank account (balance = 100€)
3. Updates account (balance = 200€)
4. Updates account (balance = 50€)
5. Saves account (balance = 200€)
6. Saves account (balance = 50€)
7. Consults account (balance = 50 €)
Transaction boundary
Transaction boundary
Here's the race condition again! Transactions didn't help!
UNIVERSIDAD Ð SEVILLA
8

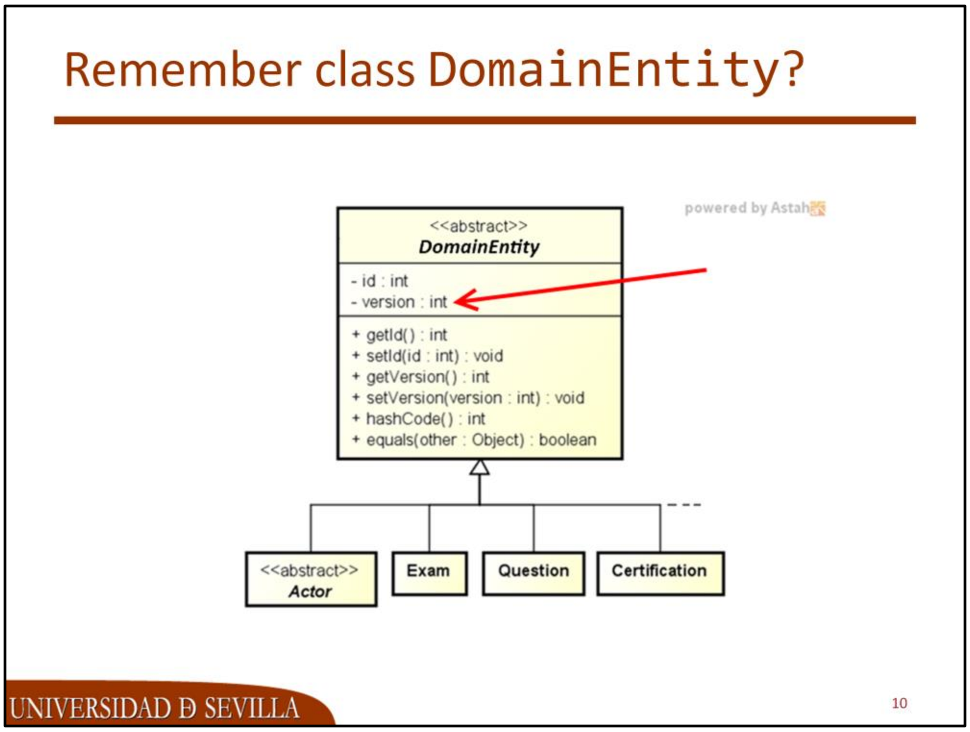
Note that a transaction is a sequence of operations that are executed atomically on a database. That is, they all succeed and then the results are committed to the database or a single failure rolls them back. Note that no operation fails in our example, so adding transactions to the customer or the clerk does not help at all.

Transactions are of no help here

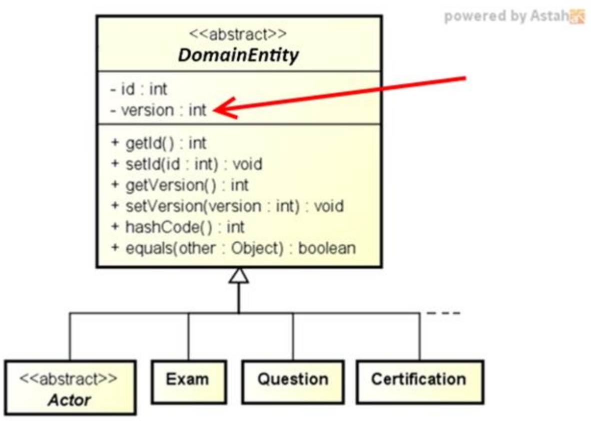
UNIVERSIDAD Ð SEVILLA

9

The conclusion is that transactions are of no help here! But we can help you out! Keep reading.



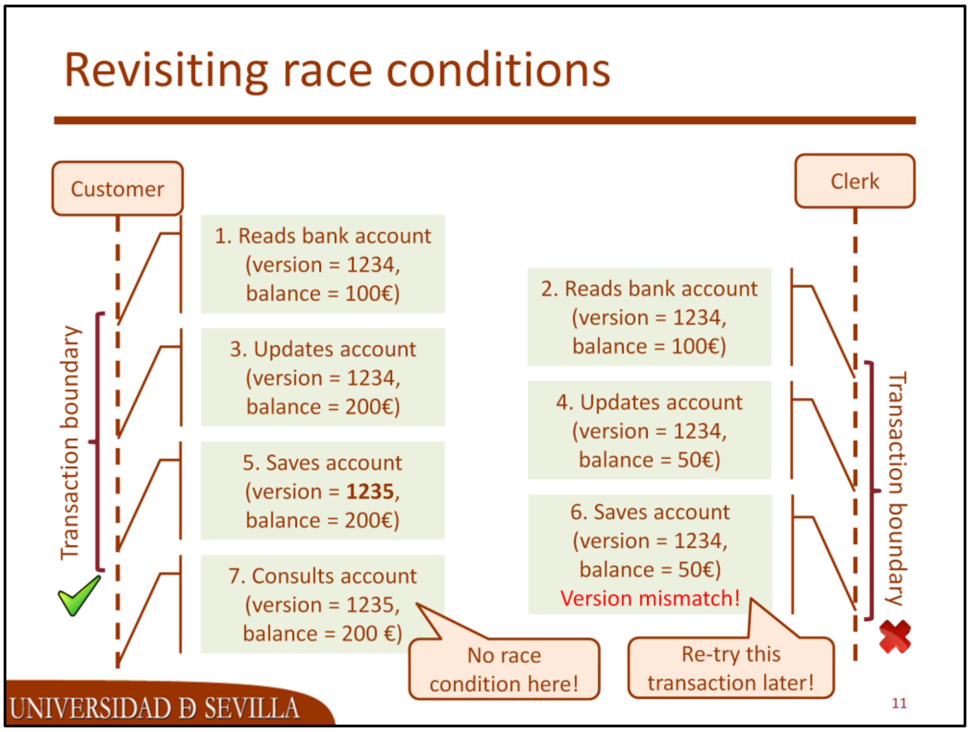
Remember class DomainEntity?



UNIVERSIDAD Ð SEVILLA

10

D’ya remember class “DomainEntity”? It is intended to be the common ancestor of every entity in your Java domain model. You must also know that this class endows every entity with an “id” attribute and a “version” attribute; the former represents its identification and the latter its version, that is, the number of times it’s been modified since it was created. You don’t have to worry about version numbers: JPA repositories care of updating them every time an entity is saved. Let’s revisit the typical race condition example using version numbers.



Revisiting race conditions

Customer
Clerk
1. Reads bank account (version = 1234, balance = 100€)
2. Reads bank account (version = 1234, balance = 100€)
3. Updates account (version = 1234, balance = 200€)
4. Updates account (version = 1234, balance = 50€)
5. Saves account (version = **1235**, balance = 200€)
6. Saves account (version = 1234, balance = 50€) Version mismatch!
7. Consults account (version = 1235, balance = 200 €)
Transaction boundary
Transaction boundary
No race condition here!
Re-try this transaction later!
UNIVERSIDAD Ð SEVILLA
11

Note that the slide now illustrates the version number. Initially, both the user and the clerk read the same bank account, which has version number 1234, that is to say: it's been modified 1234 times in the past. Note that the operations happen in the same order as before, but the sixth step is now different: it raises an exception that informs the clerk that a race condition's happened, that is, that he or she was working with an outdated version of the bank account, i.e., that it's been updated while he or she was working on it. The clerk should repeat the operation later, thus avoiding the race condition.

Thanks!

O snail
Climb Mount Fuji,
But slowly, slowly!

Thanks for reading these lecture notes