Enterprise Programmering 1

Lesson 05: EJB

Prof. Andrea Arcuri

About these slides

- These slides are just high level overviews of the topics covered in class
- The details are directly in the code comments on the Git repository

EJB Types

- 3 types, using @ annotations on the class
- *@Stateless*
- *@Stateful*
- *@Singleton*

@Stateless

- A EJB which is not supposed to have own state, ie fields
  - eg, “*private int x = 0;*”
  - can still inject objects, like *EntityManager*
- Technically, you can declare field variables, but there is no guarantee call on proxyed EJB is always on same instance
- For a given EJB, Container can have a pool of instances, and each time you use an injected proxy it can call method on different instance

@Stateful

- Can have state, ie local variables
- *@Stateful* EJB are linked to users (to sessions, to be more precise)
- If you have many requests (eg web page visits) from different users, need to have a EJB instance for each of them
    - eg, 50,000 different users asking for a page using a *@Stateful* EJB? Then you need to keep 50,000 instances in memory
- JEE Container can automatically store EJB instances to disk when running out of space (and resume when those are needed)

@Singleton

- A EJB that can have state
- Only one instance exists in the whole Container
- Every time you inject a *@Singleton*, is always the same instance
- As the same singleton can be used by different threads (eg handling concurrent web page requests), each method invocation is automatically *synchronized* in the proxy class to avoid concurrency issues

Injection

- You can inject a EJB inside another EJB by declaring a variable annotated with *@EJB*
  - eg, *"@EJB private A a;"*
  - recall that you cannot instantiate a EJB directly with "new"
- Note: you can also use *@Inject*, but that is part of the **CDI** (Contexts and Dependency Injection) specs, which is a more general framework for injection, not just EJB
  - Note: we will not see the details of CDI specs in this course

@PostConstruct

- The Container, before doing dependency injection, needs to create an instance of the EJB with “new”
- This means that the code of the *constructor* is called BEFORE dependency injection (DI) is done
- If you need to access an injected variable in the constructor, you will hence get a null pointer exception
- A method marked with *@PostConstruct* will be executed AFTER the constructor and DI
    - so, useful when you need initializing code relying on injected variables

Container Deployment

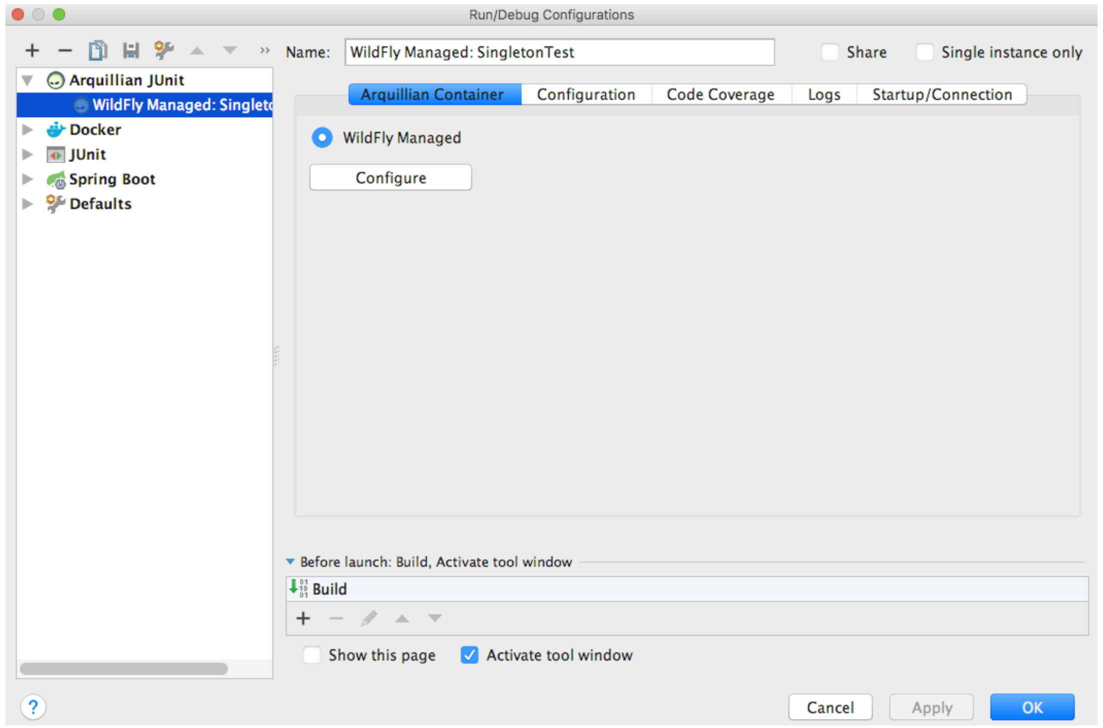
- To use EJBs, we need to run them in a JEE Container
  - WildFly, GlassFish, Payara, etc.
- We would need to package the JAR/WAR with our code, install it on a running container
- But before that, we would need to download, install and start a container
- But how to test the methods of EJBs directly from a JUnit test?

Arquillian

- A library extending JUnit that allows you to package JAR/WAR files directly from tests and deploy them on a container
- The tests themselves are run in the container, so can use dependency injection *@EJB*
- Configuration in special resource file called *arquillian.xml*
- Limitations: cannot just right-click in IDE to run tests, need some manual settings first…
- … plus, you still need to download and install a JEE Container

Test Configuration

- Arquillian “WildFly Managed”
- “Working directory” -> “$MODULE_DIR$”



Run/Debug Configurations

Name: WildFly Managed: SingletonTest ☐ Share ☐ Single instance only

- Arquillian JUnit
  - WildFly Managed: Singleto
- Docker
- JUnit
- Spring Boot
- Defaults

Arquillian Container | Configuration | Code Coverage | Logs | Startup/Connection

● WildFly Managed

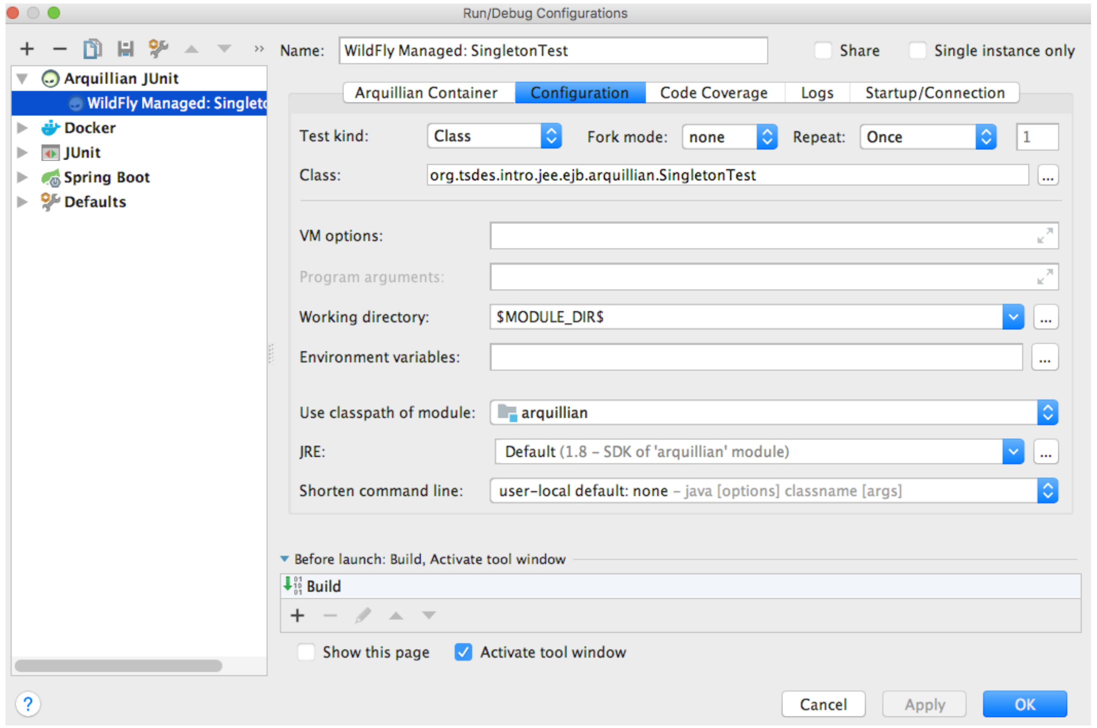
Configure

Before launch: Build, Activate tool window

Build

☐ Show this page ☑ Activate tool window

Cancel Apply OK



Run/Debug Configurations

Name: WildFly Managed: SingletonTest ☐ Share ☐ Single instance only

- Arquillian JUnit
  - WildFly Managed: Singleto
- Docker
- JUnit
- Spring Boot
- Defaults

Arquillian Container | Configuration | Code Coverage | Logs | Startup/Connection

Test kind:	Class; Fork mode: none; Repeat: Once; 1
Class:	org.tsdes.intro.jee.ejb.arquillian.SingletonTest
VM options:	
Program arguments:	
Working directory:	$MODULE_DIR$
Environment variables:	
Use classpath of module:	arquillian
JRE:	Default (1.8 – SDK of 'arquillian' module)
Shorten command line:	user-local default: none – java [options] classname [args]

Before launch: Build, Activate tool window

Build

☐ Show this page ☑ Activate tool window

Cancel Apply OK

Download/Install WildFly

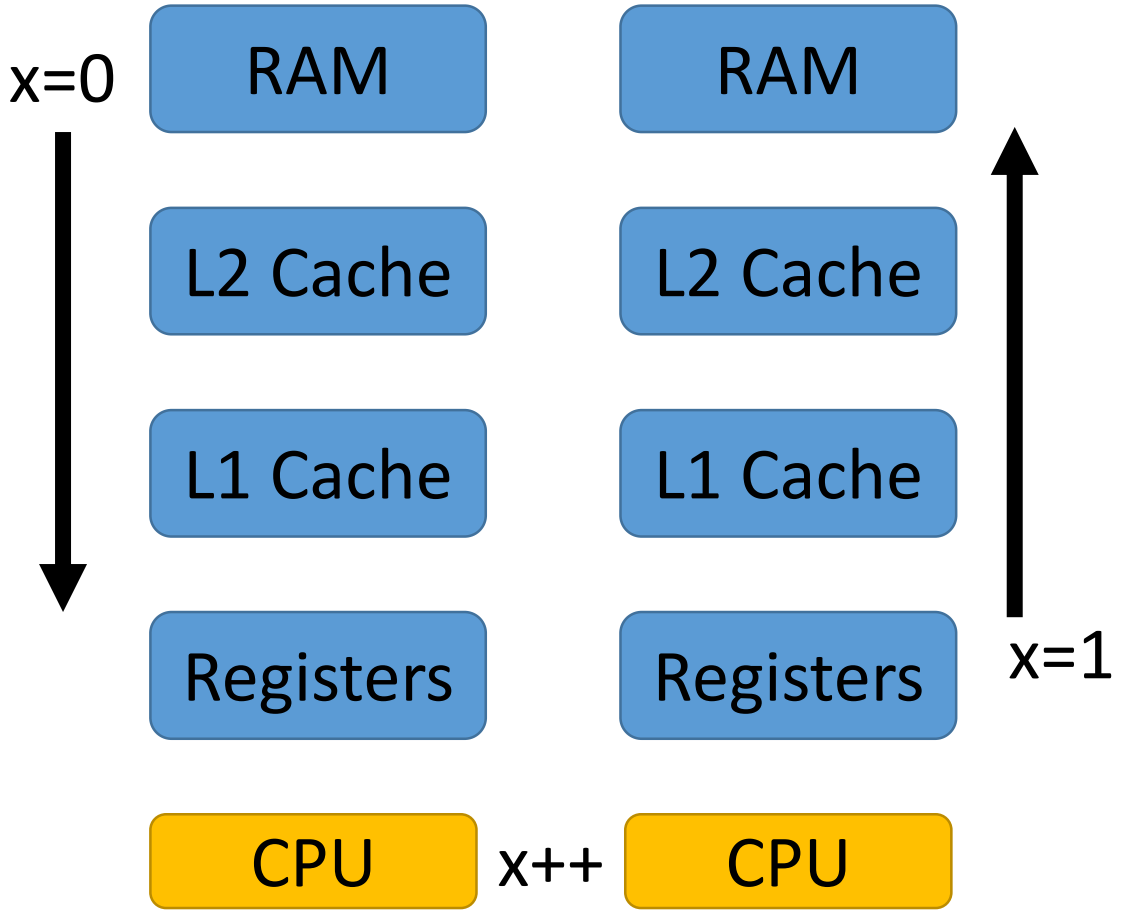
- We do it with a Maven plugin, as part of the build
  - Note: we could use Docker… but here we just want to see how Maven plugins can be used to do several different things during the build
- WildFly installed under the "*target*" folder
  - So it would be deleted when running "*mvn clean*"
- Need to run "*mvn test*" at least once to download/install WildFly BEFORE you can run tests in IntelliJ

Multi-Threading

- A server like WildFly will have a *pool of threads*
- Each incoming HTTP request could be handled by a different thread, possibly in parallel on 2 or more CPUs
- Issue when different threads are working on same data
  - example, state in a *@Singleton*
  - recall that threads in the same process share the same *heap,* ie the objects and their state declared with *new* keyword, but each thread has its own *method-call-stack*

CPU and Caches

- Assume a thread is manipulating a variable **x**, eg, by doing **x++**
- **x** needs to be loaded from the RAM, all the way down to the CPU registers
- It might take a while before changes in registers are propagated back to the RAM

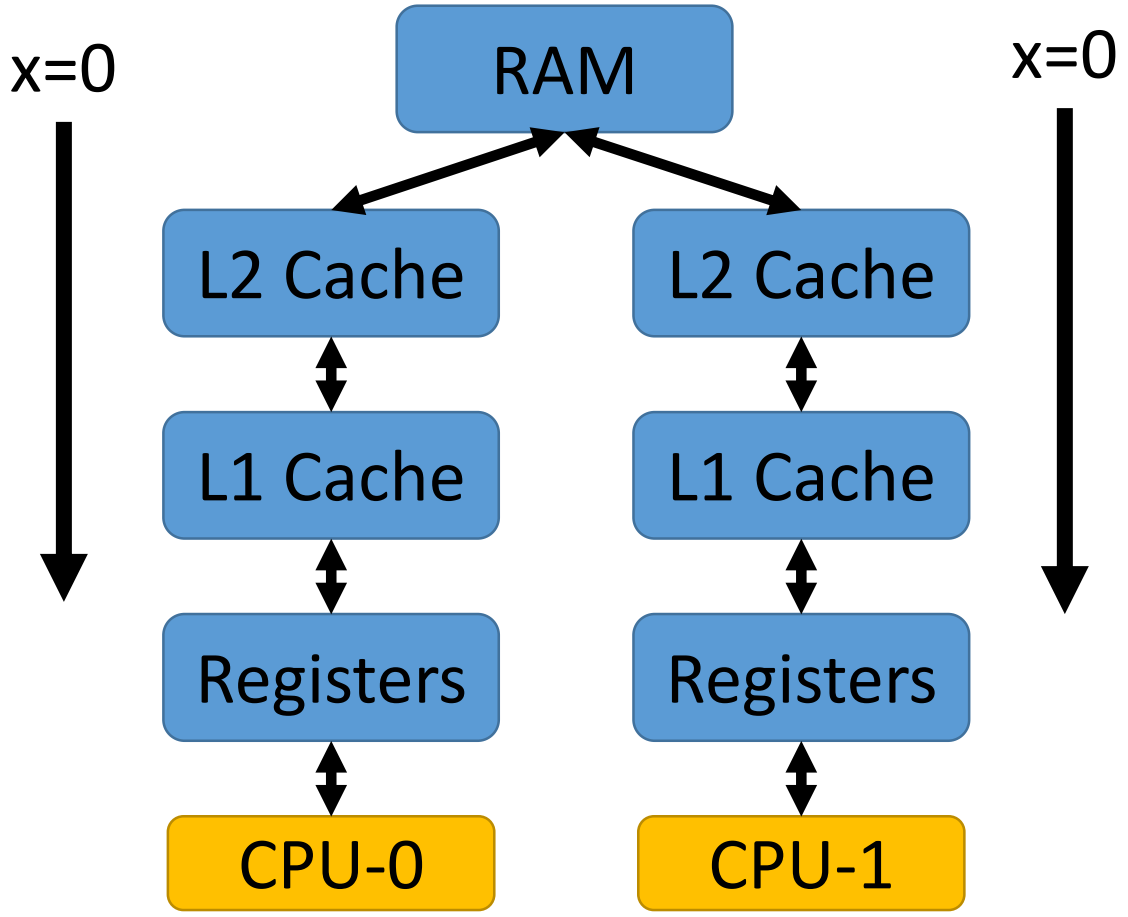


Performance

- *RAM* are *large* but *expensive* and *slow*
- *Caches* are *faster* but (much) *smaller*
- A computation will use data, and we want to have such data as close as possible to the CPU
- But registers/caches cannot hold all the data needed for the computation (ie the code executed by the thread)
- So, page swaps between caches to retrieve needed data
- *If data already in cache, do not load it again from RAM*

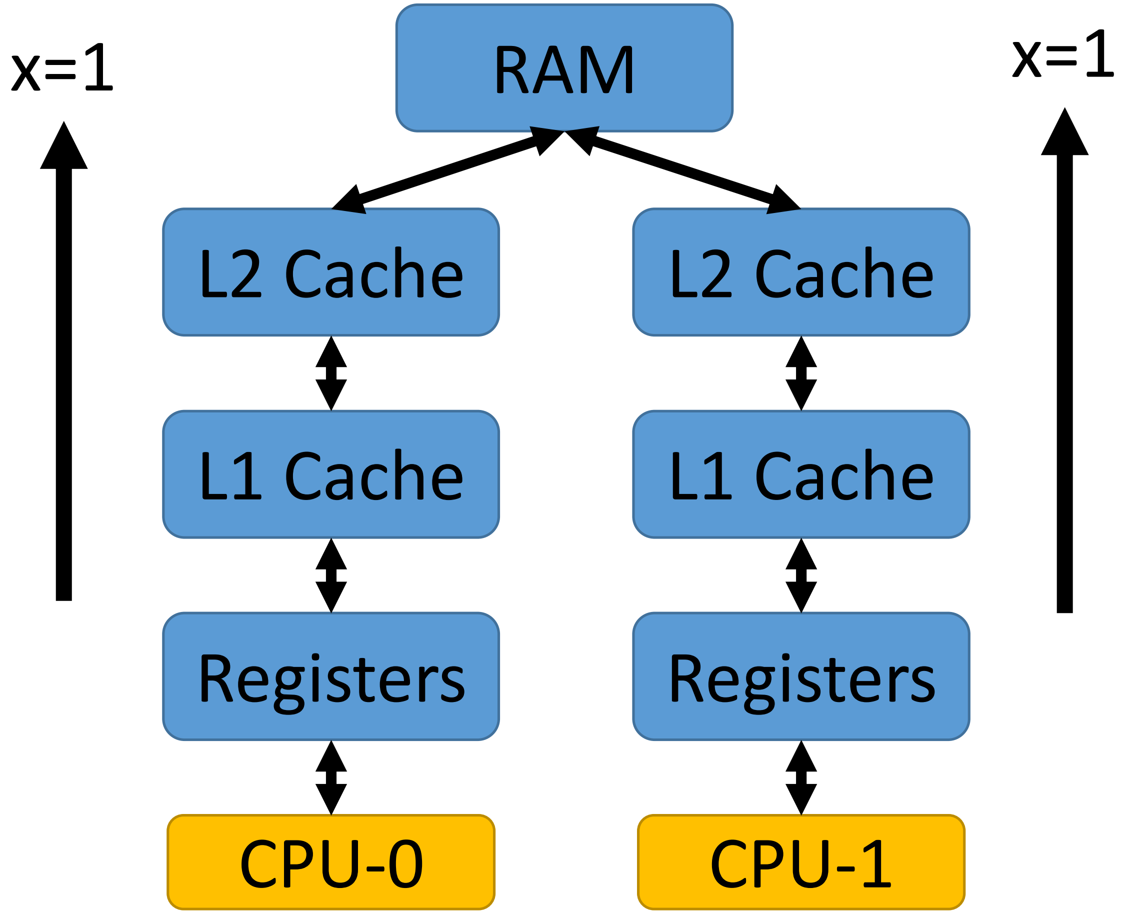
Issue: 2 Threads on 2 CPUs

- Assume both threads are reading the same **x** from the heap
- Registers in both CPUs will see the same value, eg **x=0**
- But what if CPU-0 does a **x++**?
  - CPU-1 will not see it, as still using the cached **x=0**
- What if both CPUs do a **x++**?
  - only one of them might effect the RAM



Cont.

- If threads are executed at same time, both will change **x** from 0 to 1
- **x=1** will be in the cache, and later on propagated up to the RAM
- But if 2 threads do not run at exactly same time, the result could be **x=2**, as second thread could use the updated **x=1** from the RAM

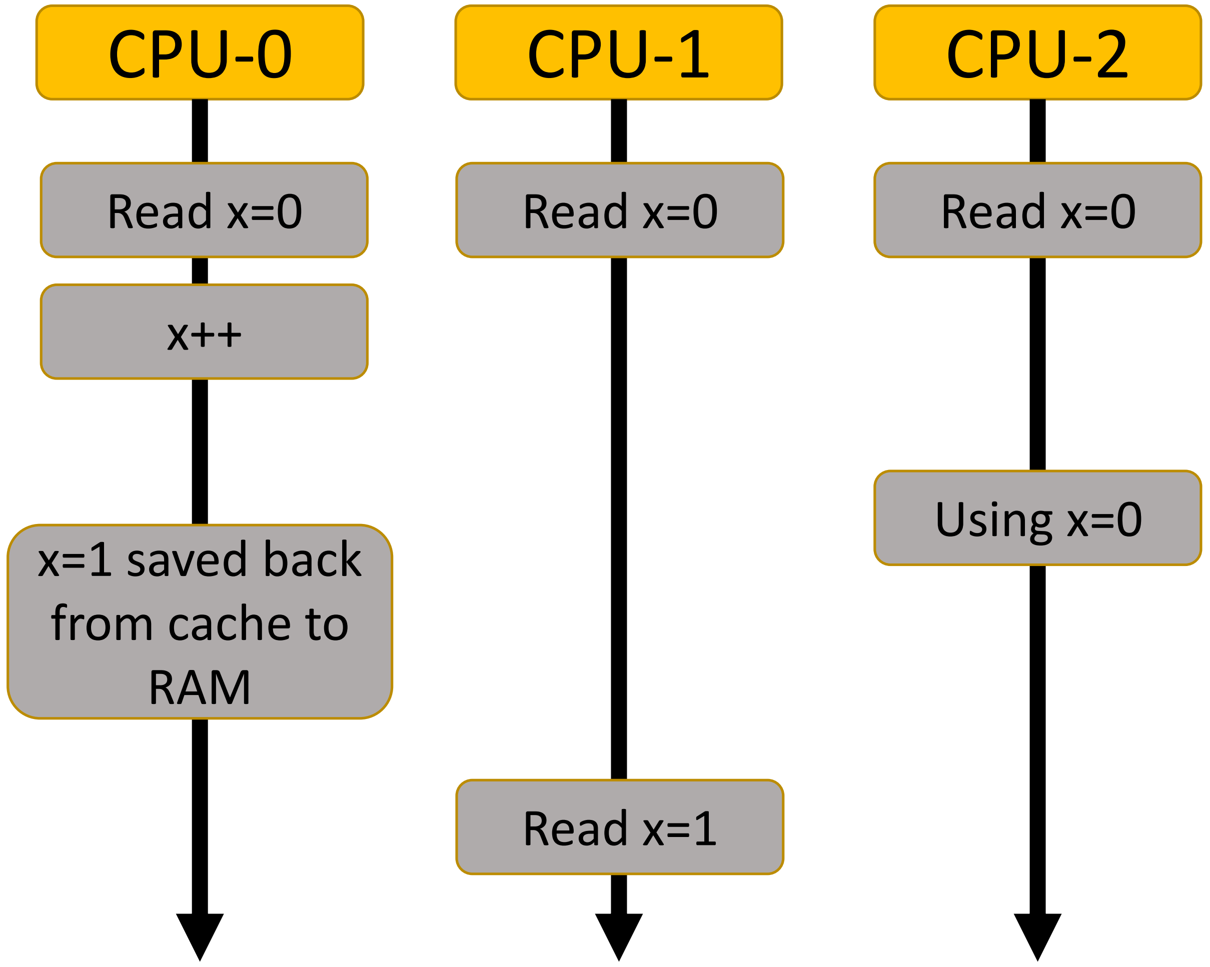


Volatile

- In Java, variables can be declared with **volatile** keyword
  - eg, **volatile int x = 0**;
- A volatile variable is always read from RAM, and not used from cache
- Useful to get most recent update in RAM, when local values in cache could become stale if other threads do modify such values
- Good when you just need to *read* such values, but not to *write* them, as read/write is **not** atomic

Volatile Issue

- Assume **x** being *volatile*
- 3 threads on 3 CPUs read **x=0** at the same time from RAM
- CPU-0 does a **x++**
- CPU-1 reads value back once CPU-0 modification saved in RAM: it will use **x=1** instead of relaying on cached **x=0**
- CPU-2 stills uses **x=0**, even if *volatile* and CPU-0 did **x++**, as update not in RAM yet



Atomicity

- When we need to READ and then WRITE a shared variable between different threads, we want it *atomic*
  - eg, if thread does **x++**, no other thread should be able to read and use x until the **+1** is saved back in RAM
- Java provides ways to execute code blocks atomically, with **synchronized** keyword, putting a **lock** on an object, released once block’s execution is completed
- Any other thread trying to execute the same code *has to wait* (ie put on hold) until the lock is released
- *Issue*: thread waiting and context-switch are computationally expensive

Git Repository Modules

- *NOTE: most of the explanations will be directly in the code as comments, and not here in the slides*
- **intro/jee/ejb/singleton**
- **intro/jee/ejb/arquillian**
- **intro/jee/ejb/multithreading**
- **intro/jee/ejb/stateful**
- **intro/jee/ejb/callback**
- Exercises for Lesson 05 (see documentation)