

# Do you really get classloaders?

Jevgeni Kabanov  
CEO & Founder of ZeroTurnaround

[twitter.com/ekabanov](https://twitter.com/ekabanov)



2011 DUKE'S CHOICE  
AWARD

ORACLE

Free!  
[social.jrebel.com](http://social.jrebel.com)

(how awesome is that?)



**Over 50 million builds, redeploys & restarts  
prevented for 30,000+ Java developers**

# To create JRebel we...

- Hooked into class loading on the JVM level
- Integrated with the class loading mechanism in more than 10 different servers
- Solved hundreds of issues connected to class loading
- Learned a lot more about class loaders than we wanted to 😊

# Inspirations

- **Vijay Saraswat**, *Java is not type-safe*, 1997
- **Sheng Liang** and **Gilad Bracha**, *Dynamic Class Loading in the Java Virtual Machine* in Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications, October 1998.

# Overview

- Basics

- What is class loading?
- How was it meant to work?

- Problems and solutions

- How do class loaders leak?

- OSGi, Spring dm, JBoss and others

- Conclusions



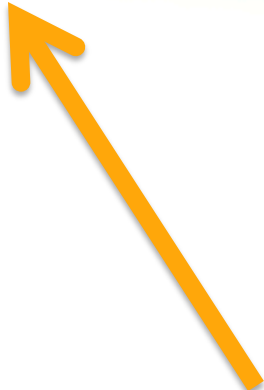
# BASICS

# Class loader API

```
public abstract class ClassLoader {  
    public Class loadClass(String name);  
    protected Class defineClass(byte[] b);  
  
    public URL getResource(String name);  
    public Enumeration getResources(String  
name);  
  
    public ClassLoader getParent()  
}
```

# Class loading

```
public class A {  
    public void doSmth() {  
        B b = new B();  
        b.doSmthElse();  
    }  
}
```



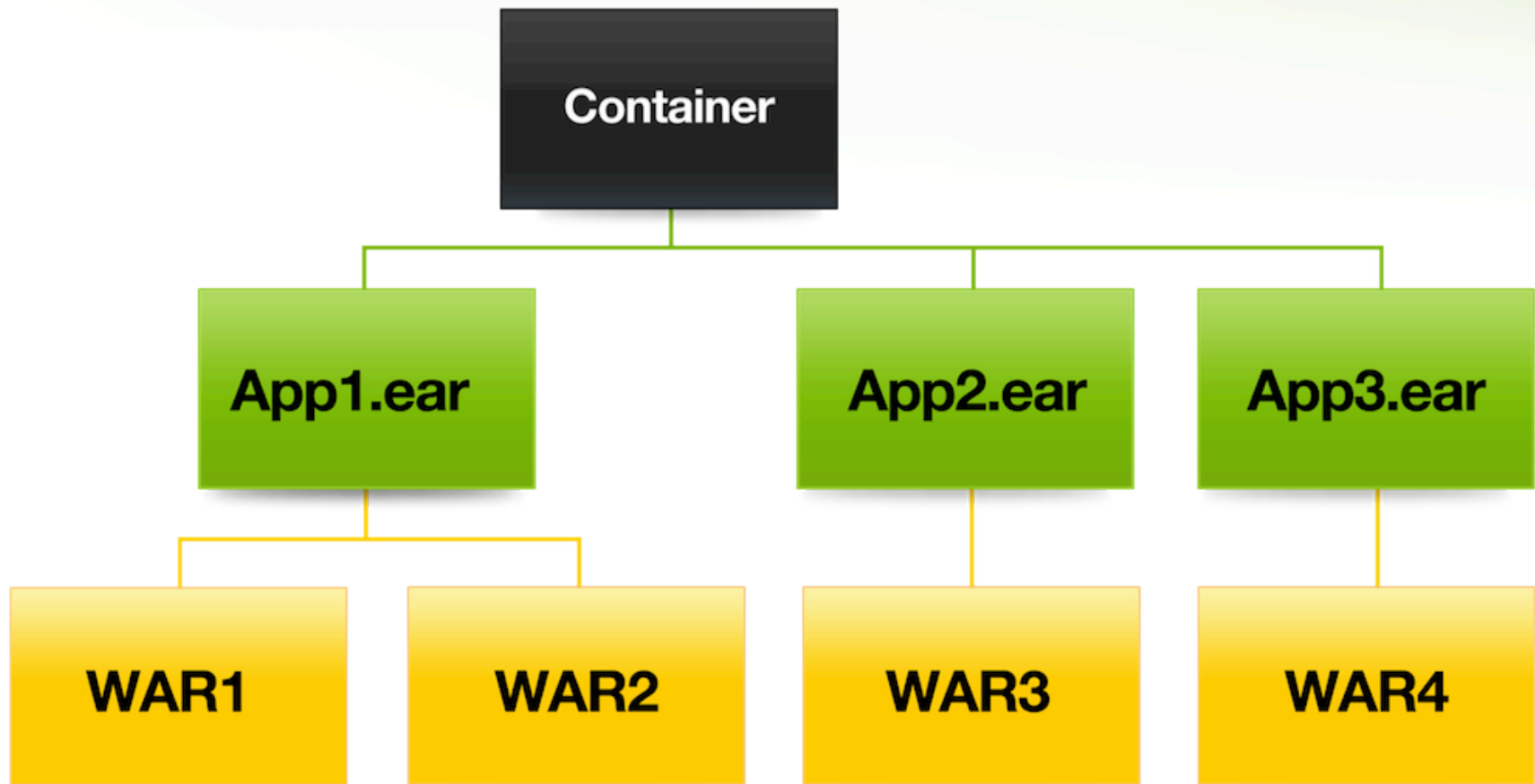
Causes a call to  
`A.class.getClassLoader().loadClass("B");`



# Delegation

- Class loaders have a *parent* class loader
- The parent is usually consulted first
  - Avoids loading same class several times
  - However in a Java EE web module local classes are searched first
- In Java EE each WAR module of an EAR gets its own class loader
  - This allows separate namespaces for applications in same container

# Java EE Delegation





# PROBLEMS AND SOLUTIONS

# No class found

## • Variants

- ClassNotFoundException
- ClassNoDefFoundException

## • Helpful

- IDE class lookup (Ctrl+Shift+T in Eclipse)
- *find \*.jar -exec jar -tf '{}' \; | grep MyClass*
- URLClassLoader.getUrls()
- Container specific logs

# Wrong class found

## • Variants

- IncompatibleClassChangeError
  - AbstractMethodError
  - NoSuch(Method|Field)FoundError
- ClassCastException, IllegalAccessException

## • Helpful

- -verbose:class
- `ClassLoader.getResource()`
- *javap -private MyClass*

# More than one class found

## • Variants

- LinkageError (class loading constraints violated)
- ClassCastException, IllegalAccessException

## • Helpful

- -verbose:class
- `ClassLoader.getResource()`

# More than one class found

## Shared ClassLoader

Util3

Factory3

`new Util3()`

`Factory3.instanceUntyped();`

Util3

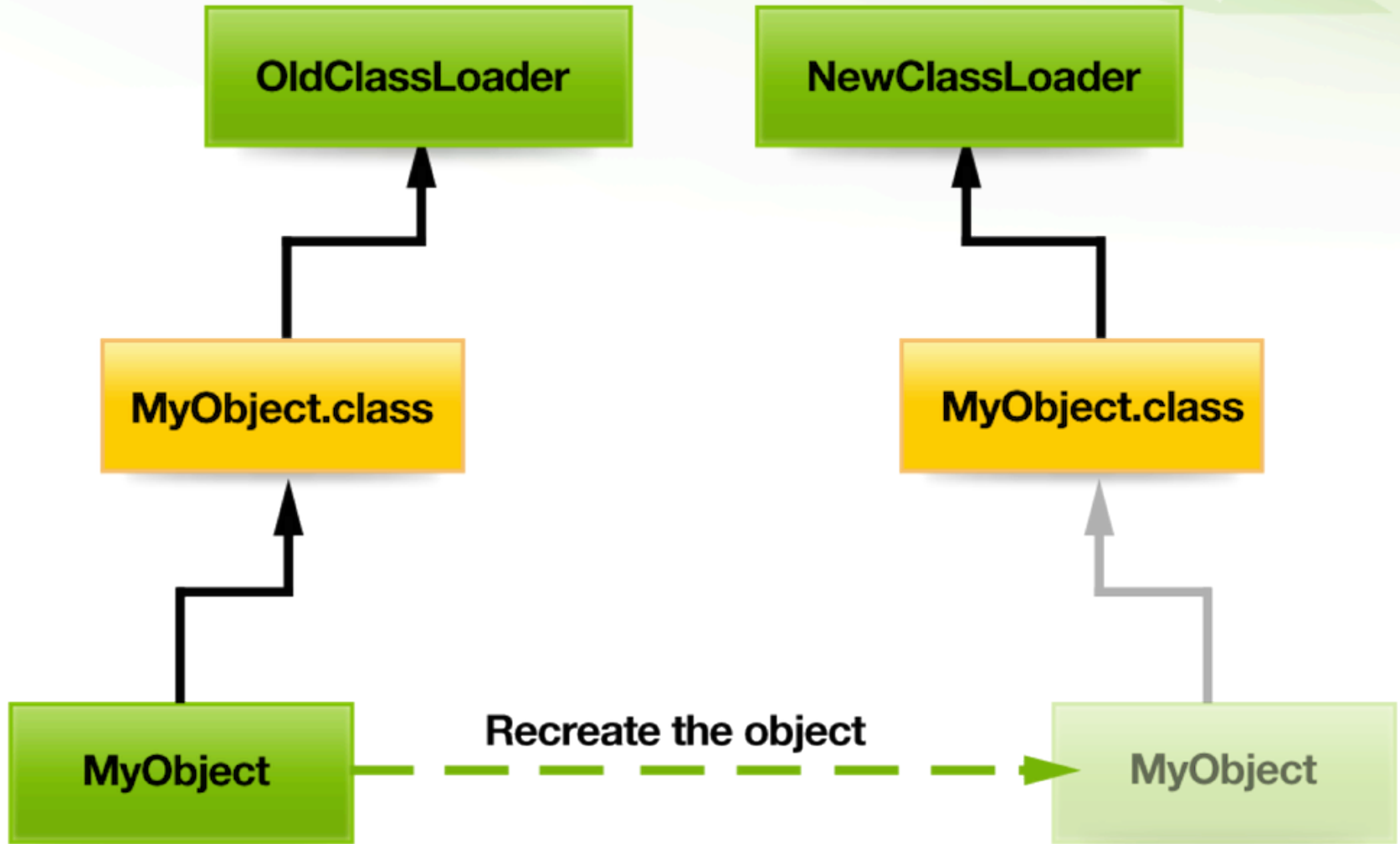
Test3

## Web ClassLoader

`Util3 u = (Util3) Factory3.instanceUntyped();`

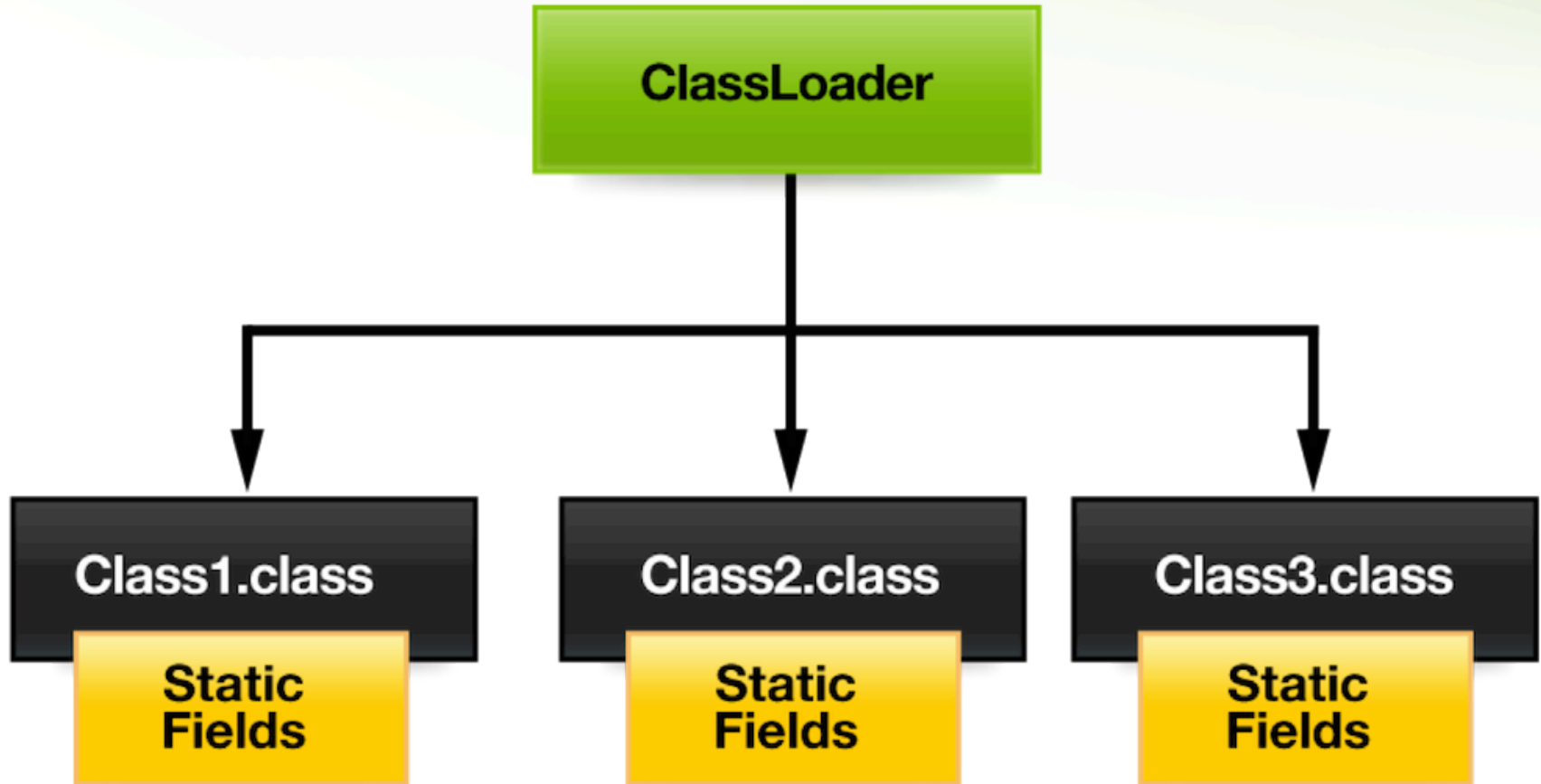


# Reloading an Object

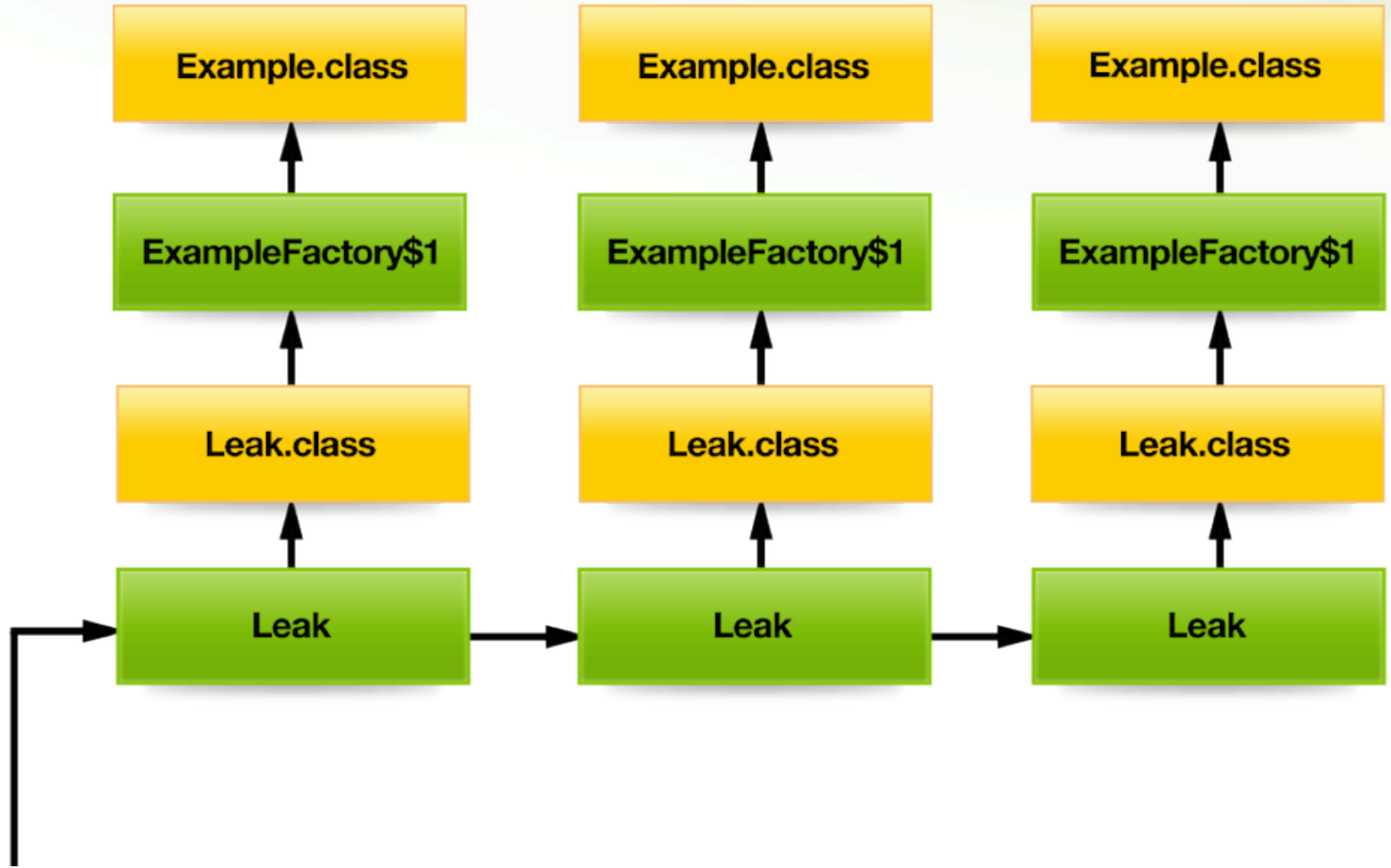




# Leaking ClassLoaders



# Leaking ClassLoaders





# STATE OF THE ART

# Hierarchy is not enough?

- Isolation
  - Different versions of the same library
- Performance
  - Class lookup is very slow
- Restricted
  - Why siblings can't see each other's classes?
- OSGi, JBoss, NetBeans and others implement a different system

# The Modern Way

- Each JAR has own class loader
- All class loaders are siblings, with one central repository
- Each JAR explicitly declares
  - Packages it exports
  - Packages it imports
- Repository can find relevant class loaders by package

# Modern Filtering

```
class MClassLoader extends ClassLoader {  
    // Initialized during startup from imports  
    Set<String> imps;  
  
    public Class loadClass(String name) {  
        String pkg = name.substring(0,  
            name.lastIndexOf('.'));  
  
        if (!imps.contains(pkg))  
            return null;  
  
        return repository.loadClass(name);  
    }  
}
```

# Modern Lookup

```
class MRepository {  
    // Initialized during startup from exports  
    Map<String, List<MClassLoader>> exps;  
  
    public Class loadClass(String name) {  
        String pkg = name.substring(0,  
            name.lastIndexOf('.'));  
        for (MClassLoader cl : exps.get(pkg)) {  
            Class result = cl.loadLocalClass(name);  
            if (result != null) return result;  
        }  
        return null;  
    }  
}
```

# Troubleshooting

- The same tricks also work with Modern class loading systems
  - `ClassLoader.getResource();`
  - `-verbose:class`
- Often can be supplemented with custom tools
- Need to think in terms of export/import in addition to classpath
  - Looking at the pseudocode can help



# Problems

- Too restrictive
  - Import is a one-way street
  - If you want to use Hibernate, you import it, but it cannot access your classes
- Easy to leak
  - Any references between class loaders are leaks waiting to happen
- Deadlocks
  - JVM enforces a global lock on `loadClass()`



# HOW CAN WE FIX IT?

# Processes

- Processes are a natural abstraction for **isolation**
- Widely used outside Java: .NET, Dynamic Languages and even browsers
- **JSR 121: Application Isolation API Specification**
  - Created in 2001
  - Last update in 2005

# In-App Updates

- Update the app code and resources, while keeping it running
- Like Databases
  - But without transactions
  - May need to handle structural updates
- **LiveRebel** is our product that does that
  - Makes small updates cheap!
  - Also automates rolling restarts!

# Conclusions

- When solving classloading problems it's very important to **validate assumptions**
- To leak a ClassLoader **it's enough to leak any object** of class loaded in that ClassLoader
- **Processes** are the only isolation abstraction that works for updates
- **In-app updates** a (symbiotic) alternative



# Visit our booth at 5<sup>th</sup> floor



Q?