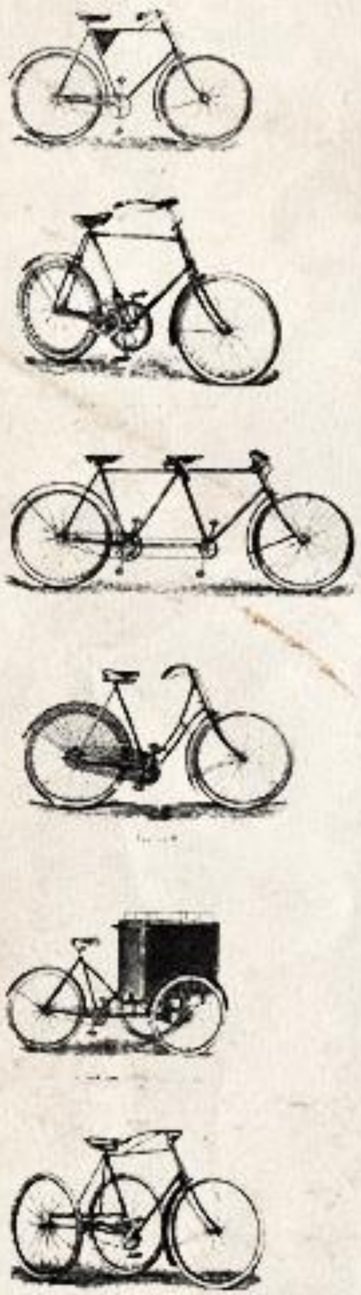
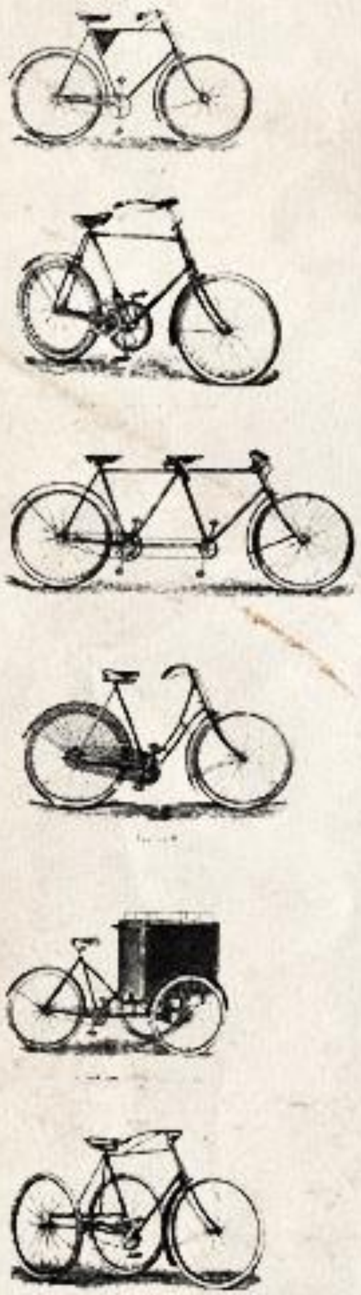


Dynamic reloading of java-classes

Vitaliy Savkin

VitaliySavkin@epam.com



Objective

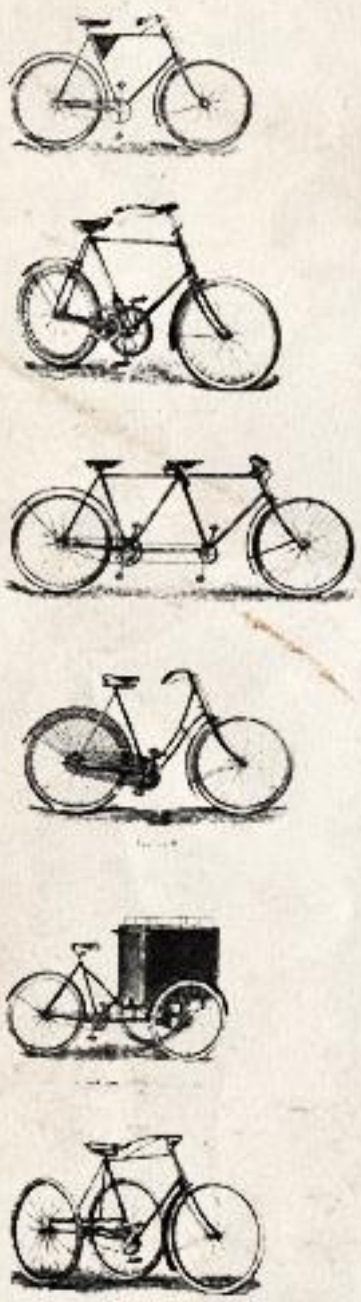
- Implementation of plugin-based system
- Plugins may be changed on the fly

Solution

- Represent each module type as a class and each module instance as an object
- (Re)load class and instantiate instance when needed

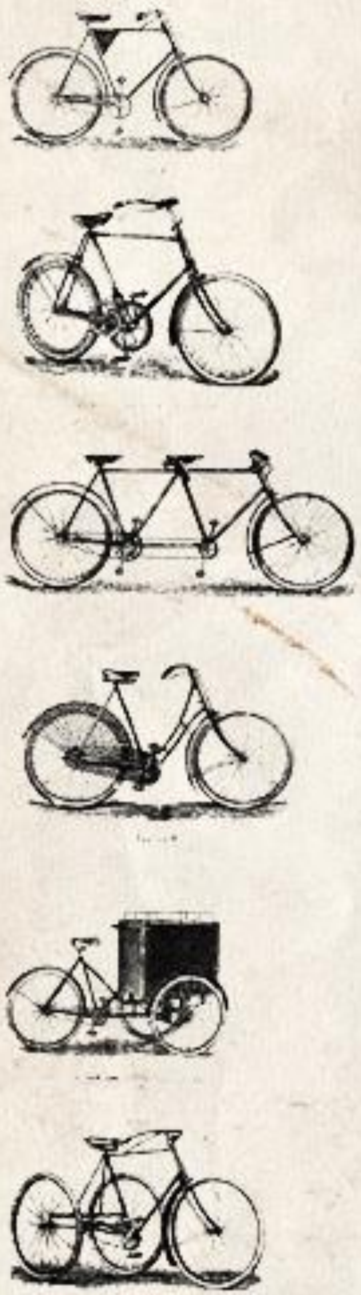
Problem

- How to (re)load a class?



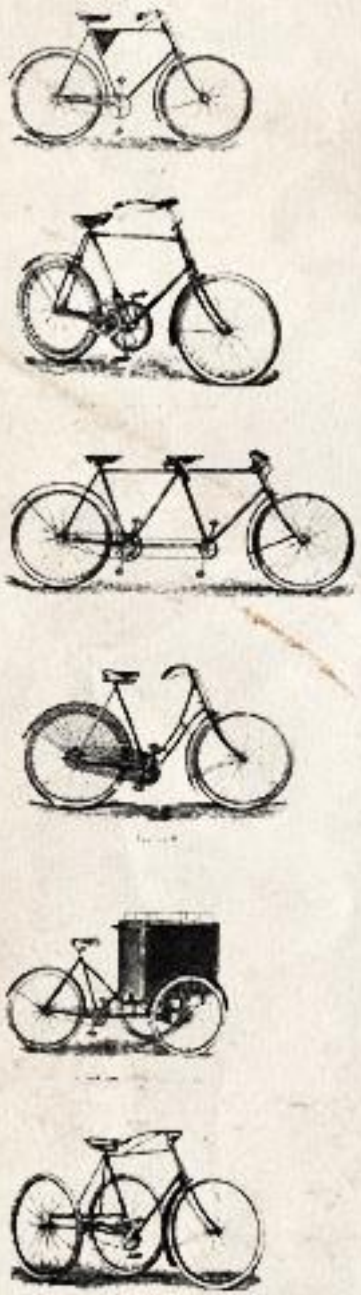
```
// file BasicModule.scala
package com.epam
class BasicModule { def method(): Unit = println(“B1") }
// TestNaive.scala
package com.epam
object TestNaive extends App {
    // first time we load class
    val c1 = getClass.getClassLoader.loadClass("com.epam.BasicModule")
    var i: BasicModule = c1.newInstance().asInstanceOf[BasicModule]
    i.method() // B1
    // wait, change and recompile BasicModule.scala(println(“B2"))
    readLine()
    // then we try to reload
    // name is the same, but class has been changed
    val c2 = getClass.getClassLoader.loadClass("com.epam.BasicModule")
    i = c2.newInstance().asInstanceOf[BasicModule]
    i.method() // B1 - WTF???
}
```

https://github.com/Vitaliy-Savkin/lt_reloading

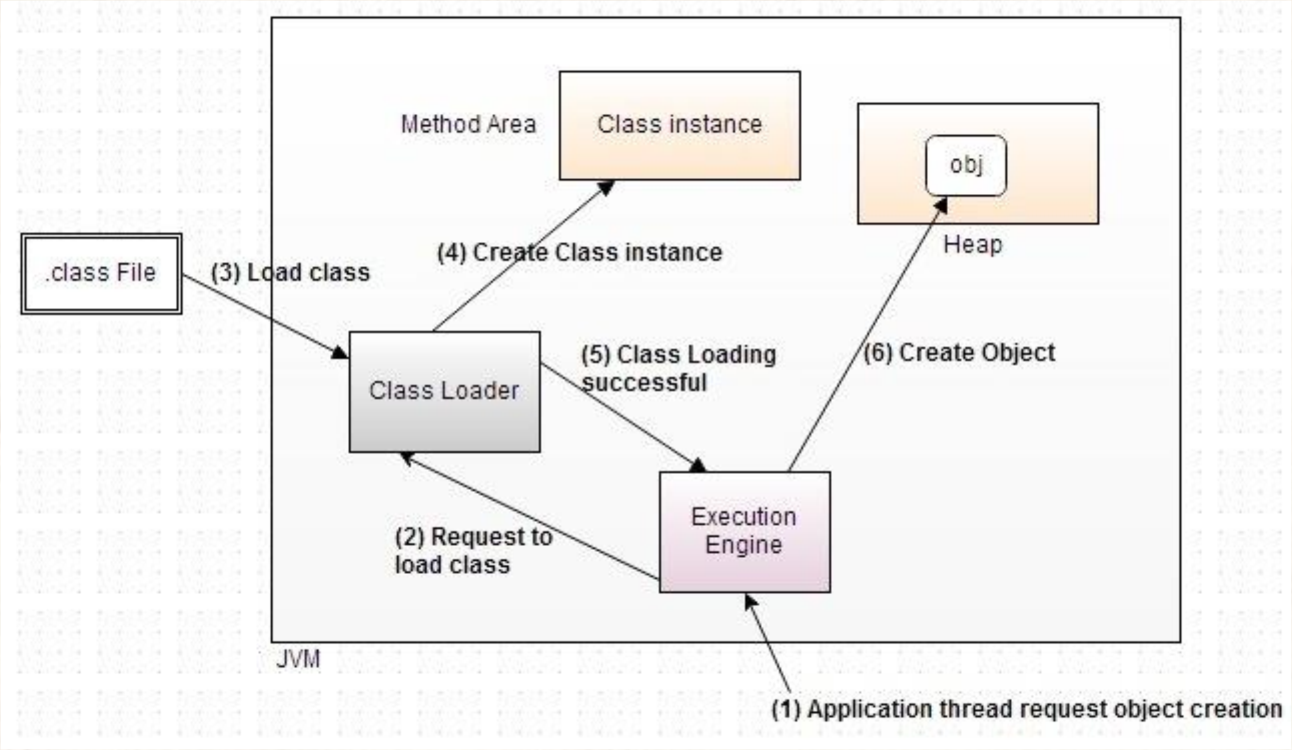


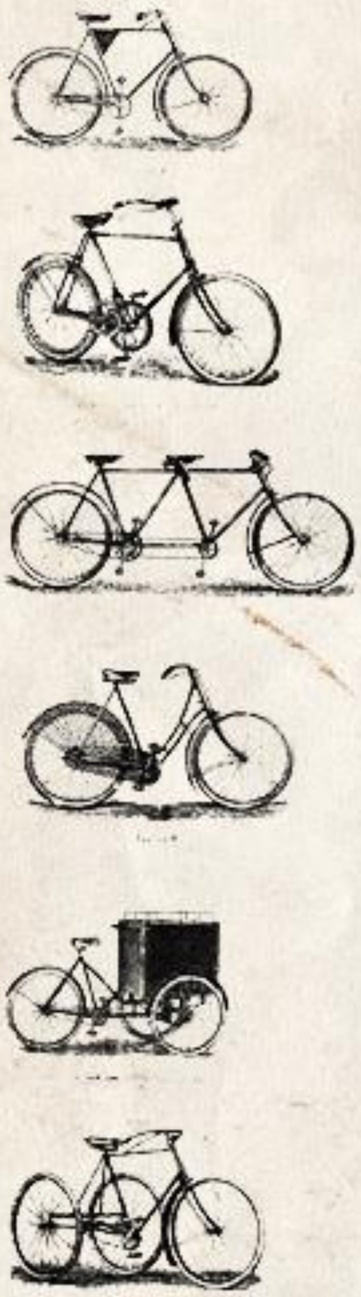
What has happened?

- Built-in class loaders cache classes they have loaded
- Therefore reloading of a class is not possible using Java's built-in class loaders
- So we have to implement own class loader



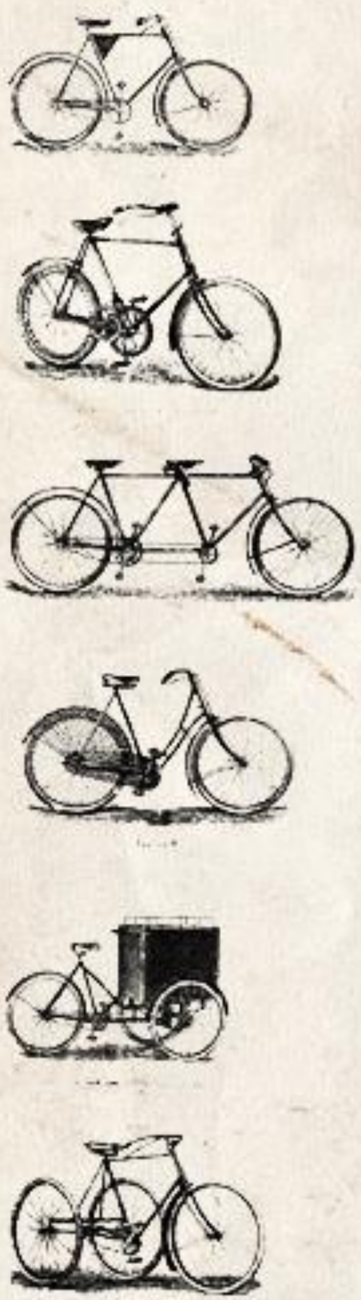
Class loading


Method Area
Class instance
obj
Heap
.class File
(3) Load class
(4) Create Class instance
Class Loader
(5) Class Loading successful
(6) Create Object
Execution Engine
(2) Request to load class
JVM
(1) Application thread request object creation

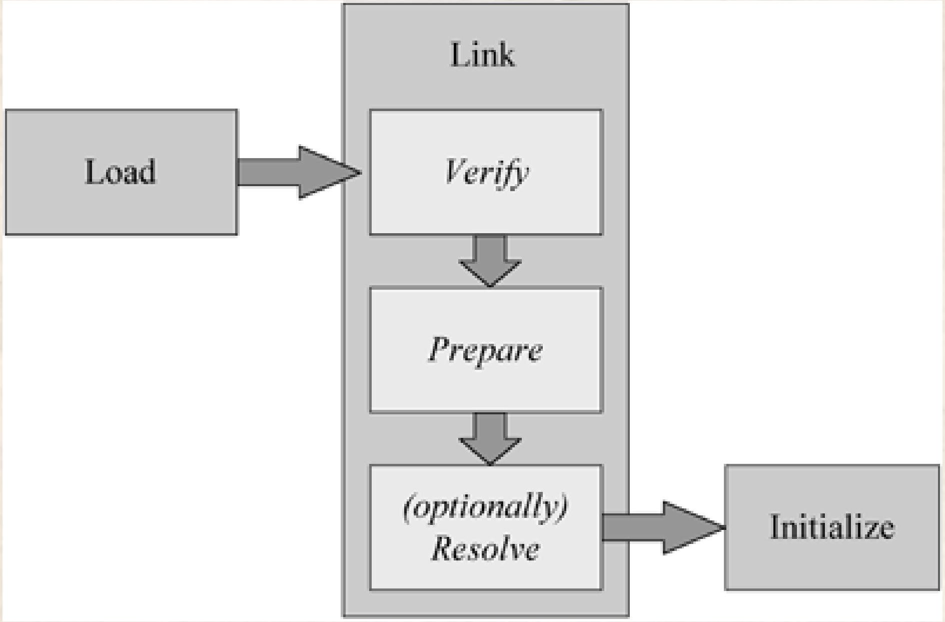


Class Loaders

- Any class in a Java application is loaded using some subclass of java.lang.ClassLoader
- Class Loaders are organized in a hierarchy - each class loader has a "parent" class loader, except of bootstrap (system) class loader
- Class Loader can delegate loading of classes to their parents in two ways:
    - parent-first
    - self-first
- Class loader loads classes only when they are needed
- Class loader lives until all the classes it has loaded are in use



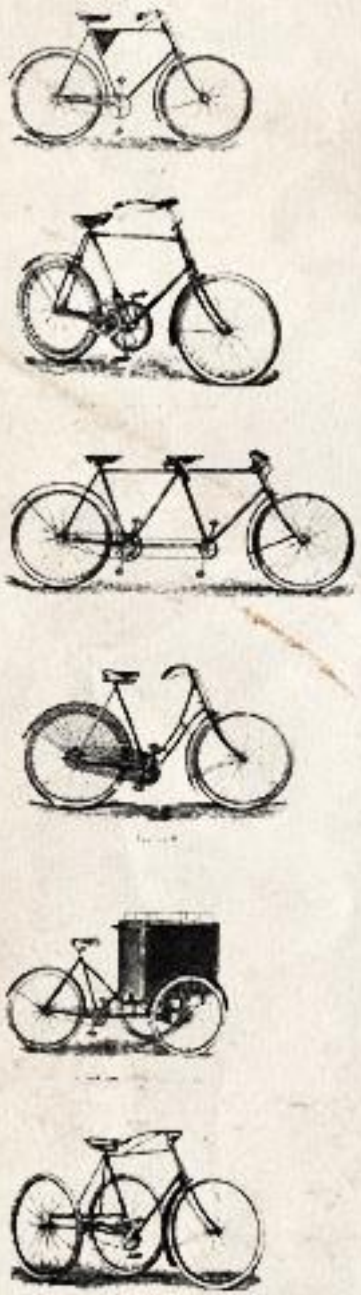
Type's lifetime



- Resolution is done using the **final** ClassLoader.resolve() method.
- The resolve() method will not allow any given ClassLoader instance to link the same class twice.
- Therefore, every time you want to reload a class you must use a new instance of your ClassLoader subclass.

Every class loaded in a Java application is identified by

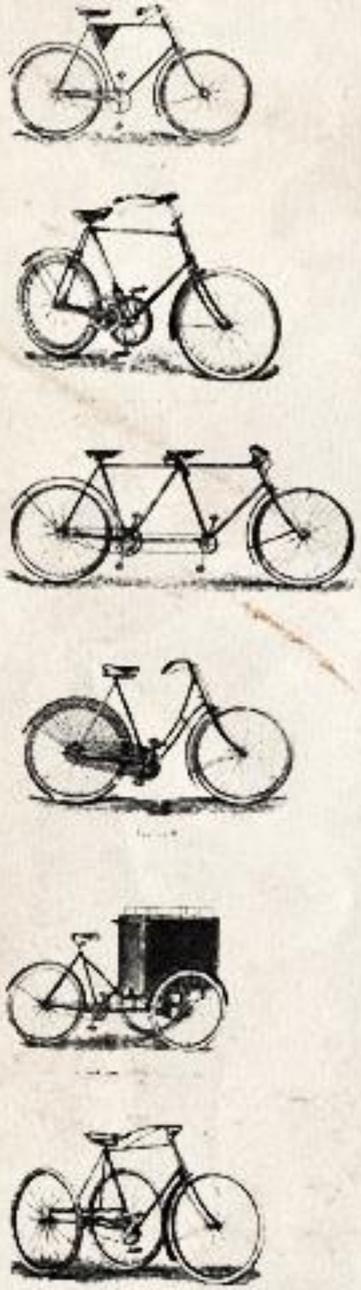
- its fully qualified name (package name + class name), and
- the ClassLoader instance that loaded it.



So we have problem with the following code:

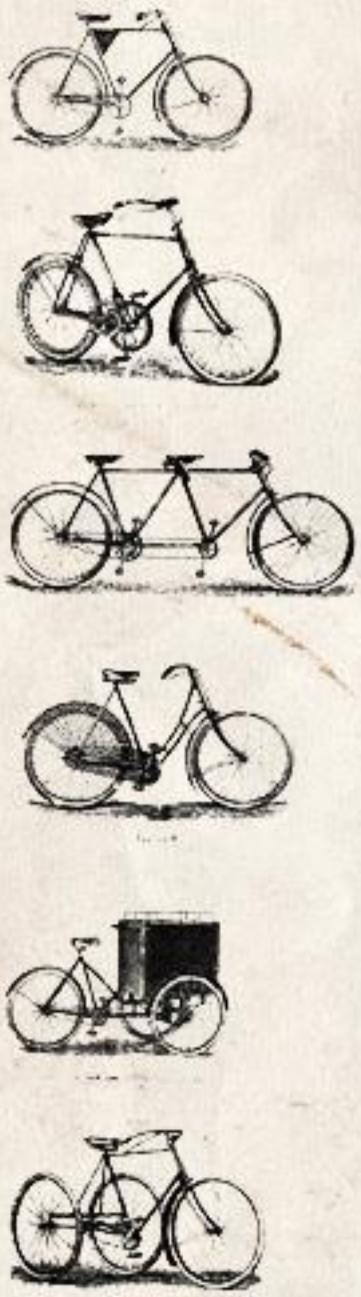
```
val c1 = new CustomClassLoader().loadClass("com.epam.A")
// ClassCastException
var i: A = c1.newInstance().asInstanceOf[A]
```

- The `A` class is referenced in the code, as the type of the `i` variable
- This causes the `A` class to be loaded by the same class loader that loaded the class this code is residing in (built-in class-loader)
- Therefore, `classOf[A]` and `c1.newInstance().getClass` are regarded as different classes



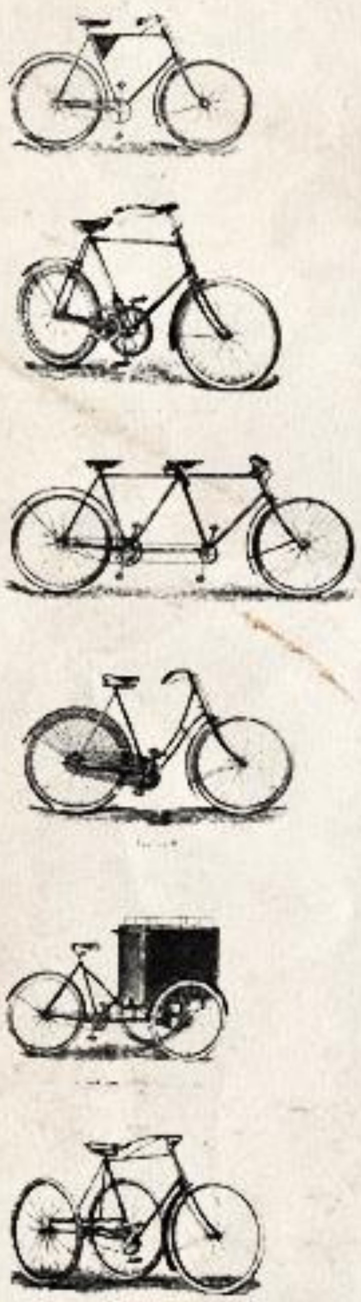
Workaround:

- Use an interface as the variable type, and just reload an implementing class.
- Use a superclass as the variable type, and just reload a subclass.

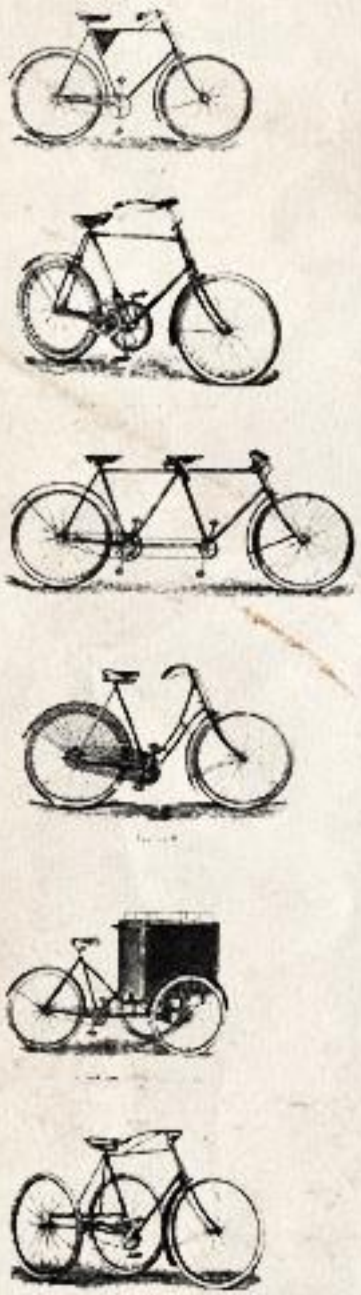


Implementation of class loader

- We use self-first delegation strategy,
- but do not want to reload all the standard library again and again
- so we have to define which classes should be loaded by custom class loader and which should be delegated to built-in class loader



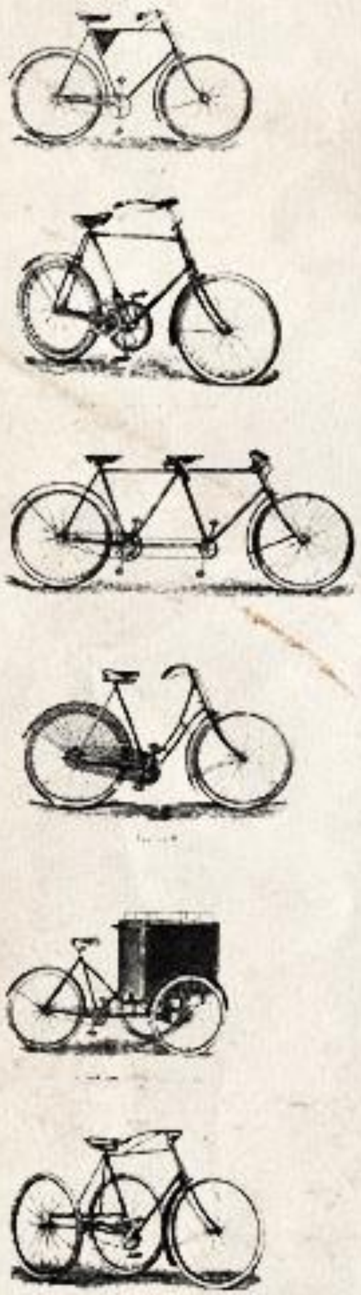
```
class NonCachingClassLoader(classFilter: String => Boolean)
                           (implicit parent: ClassLoader)
  extends ClassLoader(parent){
  protected def classNameToPath(name: String): String = …

  override def loadClass(name: String): Class[_] =
    try {
      if(!classFilter(name)) parent.loadClass(name)
      else {
        val fileURL = parent.getResource(classNameToPath(name))
        val file = new File(fileURL.getFile)
        val classData = Array.ofDim[Byte](file.length.toInt)
        val dis = new DataInputStream(new FileInputStream(file))
        dis.readFully(classData)
        dis.close()
        defineClass(name, classData, 0, classData.length)
      }
    } catch {
      case e @ (_: MalformedURLException | _: IOException) => null
    }
}
```



```scala
// file TestCustomCL.scala
package com.epam
object TestCustomCL extends App {
    implicit val parent = this.getClass.getClassLoader
    // first time we load class
    val c1 = new NonCachingClassLoader(_ startsWith "com.epam.impl")
                                      .loadClass("com.epam.impl.Module")
    var i: BasicModule = c1.newInstance().asInstanceOf[BasicModule]
    i.method() // Module1
    // wait, change and recompile AImpl.scala (println(“Module2"))
    readLine()
    // then we try to reload
    val c2 = // same stuff
    i = c2.newInstance().asInstanceOf[BasicModule]
    i.method() // Module2 - Reloaded!
}
// file Module.scala
package com.epam.impl
class Module extends BasicModule { def method() = println(“Module1") }
```

Caution!

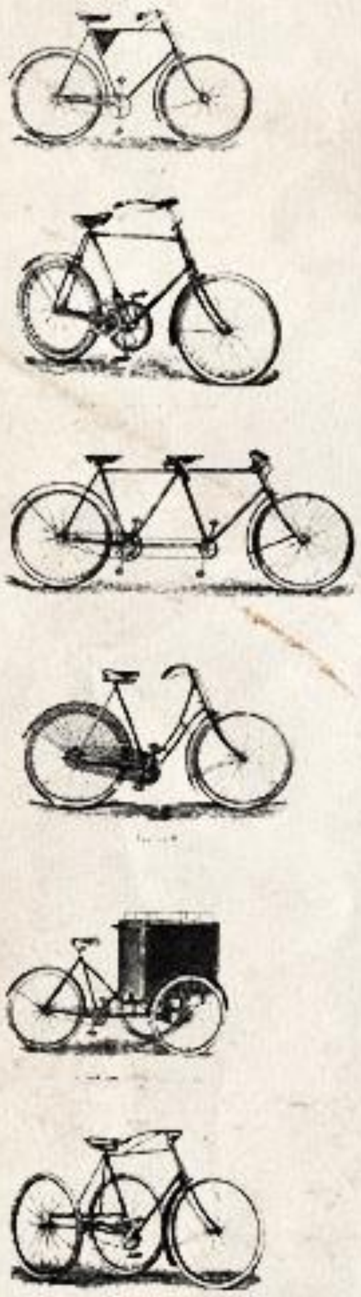
- Your strategy of filtration of classes may be much more complicated then in example
- but you have only string to take decision
- so consider the possibility of redesigning organization of namespaces and naming conventions to make your strategies more clear.
- Be very care with inner classes, take into account how their names are constructed, how are they used, and do you really need them.
- Be care with companion objects in Scala.



Class unloading

- You can not explicitly unload class
- VM can optionally unload the classes after they are no longer referenced by the program
- You must have no references to any instance of a class allow JVM unload definition of this class
- Therefore, there is a danger of memory leaks
- Which is especially noticeable in Java 7 and earlier versions, where classes were stored in Permanent Generation (PermGen)

Questions?



Thanks for attention!