

# Java Class Loading im Detail

Felix Becker

Scala User Group Köln

8. Februar 2017

# Vortragsinhalte

- 1 Class Loading Grundlagen
  - Class startup life cycle
  - ClassLoader delegation model
- 2 ClassLoader-Implementierungen
- 3 Fortgeschrittene ClassLoader-Techniken
  - Der ContextClassLoader & die WebApps
  - Loading Synthetic Classes

# Motivation

- Grundlage für Vortrag von Mathias Kub
- ClassLoader-Hölle bei vielen Entwicklern gefürchtet
- Verständnis vom class loading bei komplexen Fehlersituationen notwendig
- Vorbereitung für Folgetalks über Scala Bytecode / JVM Internals + Scala

# Was tun ClassLoader eigentlich?

- Laden Klassen und Ressourcen aus beliebigen Quellen
- Trennen die Datenquelle von der Applikation, Applikation muss nur den ClassLoader nutzen (Plattformunabhängigkeit!)
- Klassen werden im JVM-Bytecode-Format geladen
- Ressourcen können beliebige Daten sein

# Startup life cycle einer Javaklasse (JVM Spec §12.1.2)

- ❶ Loading §12.2.1
  - Laden des Bytecodes in die JVM über den ClassLoader
- ❷ Verify §12.3.1
  - Validierung der Klassenstruktur / Daten (Opcodes gültig, branch instructions check, signature check, ...)
- ❸ Prepare §12.3.2
  - Storage Allocation, Initialisierung von static fields (default values, keine static Initializer)
- ❹ (Resolve §12.3.3)
  - Optionaler Schritt - Symbolic link resolution
- ❺ Initialization §12.4.1
  - Aufruf von Static-Initializern, Initialisierung von static fields

# Initialization

Initialisierung der Klasse ist die erste "aktive" Codeausführung von Klassencode (falls static fields / initializers vorhanden). Initialisierung findet statt, wenn das erste Mal:

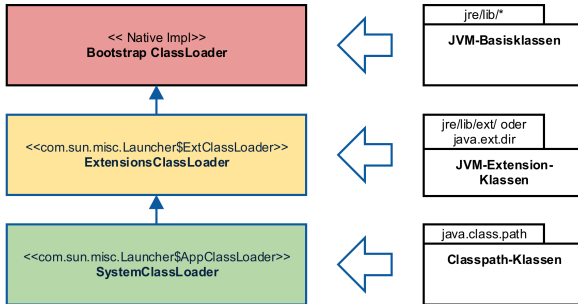
- eine Instanz der Klasse erstellt wird
- eine static-Methode der Klasse aufgerufen wird
- auf static-Variablen der Klasse zugegriffen wird (Zuweisung, Lesen)

Die Initialisierung einer Klasse erzwingt die Initialisierung aller Super-Classes (gilt nicht für Interfaces!)

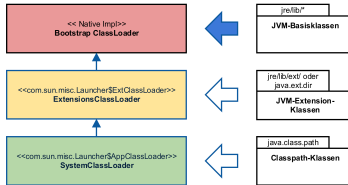
Initialisierungen sind Threadsafe - sichere Erzeugung von Singletons!

Die Initialisierung ist intern in der JVM über ein initialization lock abgesichert. Die JVM garantiert, dass die Initialisierung einer Klasse genau 1x stattfindet.

# Standard-ClassLoader JVM 8 (Oracle)



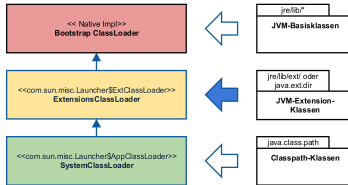
# Bootstrap-ClassLoader



- Lädt Java-Basisklassen (z.B. `java/lang/String`)
- Nativ implementiert
- `java.lang.ClassLoader`: private nativ `Class` `findBootstrapClass`

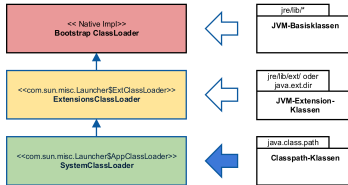


# Extension-ClassLoader



- Lädt Extension-Klassen
- Beispiel: `com/sun/nio/zipfs/ZipPath`
- lädt Extension-Klassen aus `jre/lib/ext/` bzw. `java.ext.dirs`
- `com.sun.misc.Launcher$ExtClassLoader`

# System-ClassLoader



- Lädt alle Klassen aus dem `java.class.path` (`java -cp ..`)
- `java.class.path` (`com.sun.misc.Launcher$AppClassLoader`)

# Relevante Funktionen

- Aus Applikationssicht:
  - `Class.forName(String)`
  - `Class.getClassLoader()`
  - `ClassLoader.loadClass(String)`
- Im `ClassLoader`:
  - `findLoadedClass(String)`
  - `findBootstrapClassOrNull(String)`
  - `resolveClass`
  - `defineClass`

## Live-Demo 1: First Steps

# ClassLoader.loadClass

```
1  protected Class<?> loadClass(String name, boolean resolve)
2      throws ClassNotFoundException {
3
4      synchronized (getClassLoadingLock(name)) {
5
6          Class<?> c = findLoadedClass(name);
7
8          if (c == null) {
9              try {
10                 if (parent != null){
11                     c = parent.loadClass(name, false);
12                 } else {
13                     c = findBootstrapClassOrNull(name);
14                 }
15             } catch (ClassNotFoundException e) {}
16
17             if (c == null){ c = findClass(name); }
18         }
19
20         if (resolve){ resolveClass(c); }
21
22         return c;
23     }
24 }
```

- Class Loading ist synchronized
- Lookup-Schritte:
  - 1 findLoadedClass (native)
  - 2 parent / bootstrap lookup
  - 3 findClass
- findClass wird von eigenen ClassLoader-Implementierungen überschrieben

# ClassLoader.resolveClass

```
1 protected Class<?> loadClass(String name, boolean resolve)
```

*Lädt die Klasse und bindet sie mit resolveClass() ein, wenn resolve gleich true ist. - (Java ist auch eine Insel)*

*The variable resolve is a flag to tell the class loader that classes referenced by this class name should be resolved (that is, any referenced class should be loaded as well). - JavaWorld (The basics of Java class loaders)*

*As I mentioned previously, loading a class can be done partially (without resolution) or completely (with resolution). When we write our version of loadClass, we may need to call resolveClass, depending on the value of the resolve parameter to loadClass. - IBM Developer Works (Understanding the Java ClassLoader)*

## Live-Demo 2: resolveClass

# ClassLoader.resolveClass

```
1  protected Class<?> loadClass(String name, boolean resolve)
```

Lädt die Klasse und bindet sie mit `resolveClass()` ein, wenn `resolve` gleich `true` ist. - (Java ist auch eine Insel)

The variable `resolve` is a flag to tell the class loader that classes referenced by this class name should be resolved (that is, any referenced class should be loaded as well). - JavaWorld (The basics of Java class loaders)

As I mentioned previously, loading a class can be done partially (without resolution) or completely (with resolution). When we write our version of `loadClass`, we may need to call `resolveClass`, depending on the value of the `resolve` parameter to `loadClass`. - IBM Developer Works (Understanding the Java ClassLoader)

## Live-Demo 2: resolveClass

```
1  // http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/tip/src/share/vm/prims/jvm.cpp
2
3  // 732 - 735
4  JVM_ENTRY(void, JVM_ResolveClass(JNIEnv* env, jclass cls))
5      JVMWrapper("JVM_ResolveClass");
6      if (PrintJVMWarnings) warning("JVM_ResolveClass not implemented");
7  JVM_END
```

# ClassLoader.getResource

```
1  public URL getResource(String name) {  
2  
3      URL url;  
4  
5      if (parent != null) {  
6          url = parent.getResource(name);  
7      } else {  
8          url = getBootstrapResource(name);  
9      }  
10  
11     if (url == null) {  
12         url = findResource(name);  
13     }  
14  
15     return url;  
16 }
```

- Zuerst wird beim Parent / Bootstrap-ClassLoader nach der Resource gesucht
- findResource wird aufgerufen, falls Parent / Bootstrap keine Resource gefunden haben
- findResource wird von eigenen ClassLoader-Implementierungen überschrieben

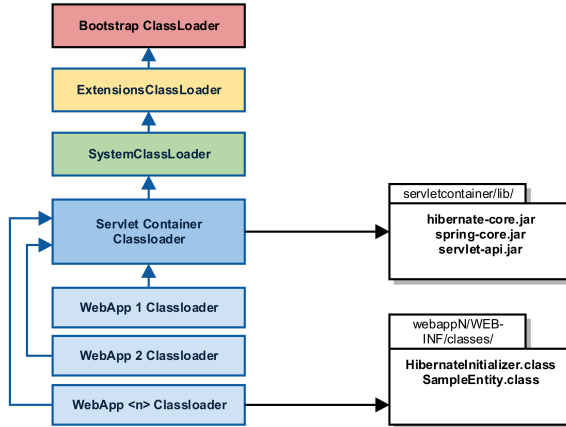


# URL-ClassLoader

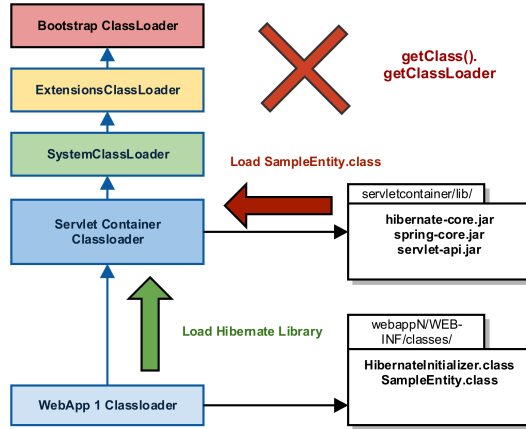
```
1  // Removed SecurityManager Stuff
2  protected Class<?> findClass(final String name) throws ClassNotFoundException {
3      String path = name.replace('.', '/').concat(".class");
4      Resource res = ucp.getResource(path, false);
5      if (res != null) {
6          try {
7              return defineClass(name, res);
8          } catch (IOException e) {
9              throw new ClassNotFoundException(name, e);
10         }
11     } else {
12         throw new ClassNotFoundException(name);
13     }
14 }
```

## Live-Demo 3: URL-ClassLoader

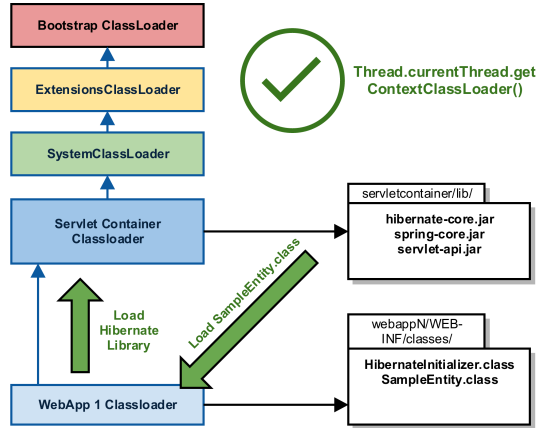
# WebApp ClassLoader



# WebApp ClassLoader - Context ClassLoader



# WebApp ClassLoader - Context ClassLoader



## Live-Demo 4: Doppeltes Laden von Klassen über getrennte ClassLoader

# Synthetic Classes

- Synthetische Klassen sind "künstliche" Klassen, die nicht im source code abgebildet werden
- Werden vom Compiler oder zur Runtime erzeugt
- Stark eingesetzt im AOP-Umfeld - z.B Transaktions-Proxy-Klassen
- Nutzung in Testframeworks wie PowerMock
- Können neben dem Proxy-API von Java auch aus Bytes direkt im ClassLoader erzeugt werden

## Live-Demo 5: Erzeugung synthetischer Klassen



# Ende

Vielen Dank für eure Aufmerksamkeit!  
Vortrag und Sourcen auf [github.com/fbe/classloader-vortrag](https://github.com/fbe/classloader-vortrag)