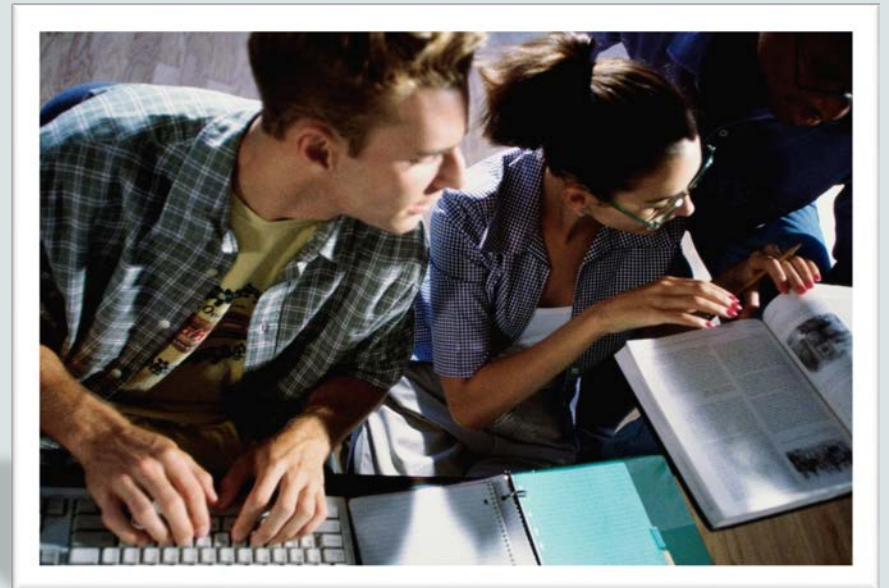




Java Programming

3-2

ClassLoader



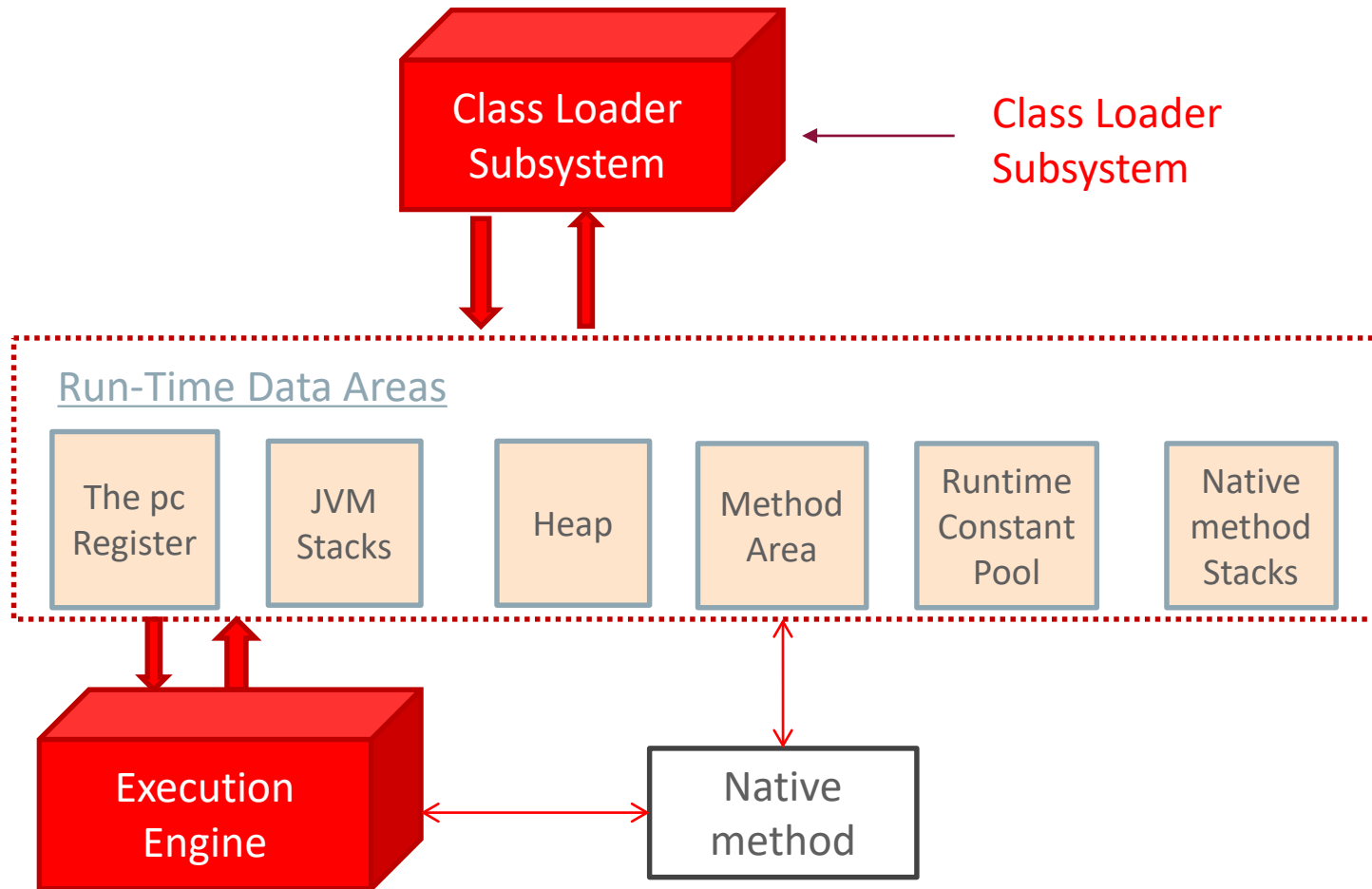
Objectives

This lesson covers the following topics:

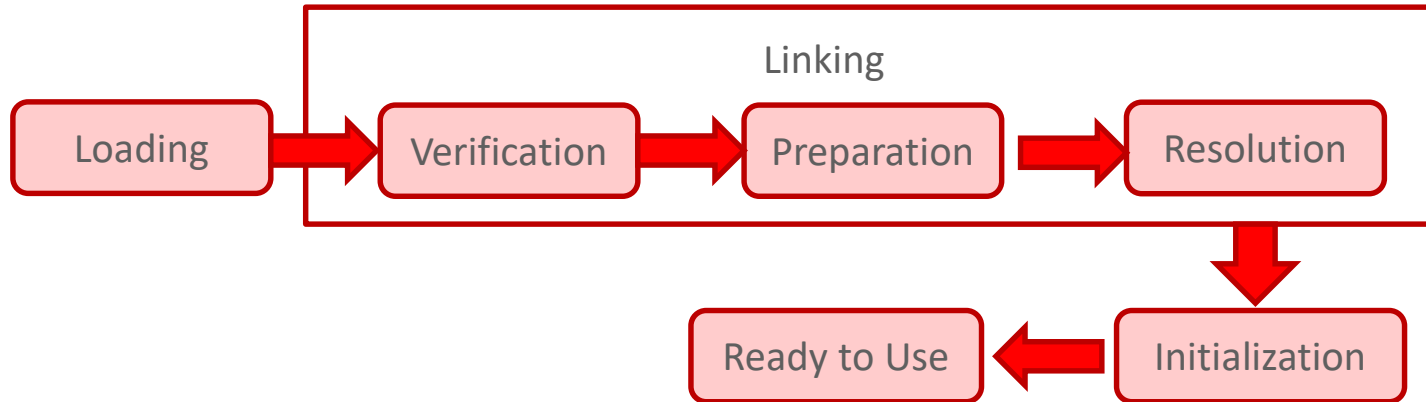
- The Class Loading Overview
- ClassLoader Loading Procedure
- JDK ClassLoader Class
- ClassLoader Hierarchy
- Custom ClassLoader
- Class Linking



Java Virtual Machine Runtime Structure



Class Loader Subsystem



- The Java Runtime starts up by creating an initial class, for example:
C:\Java ExampleClass
- The Java Runtime then links the initial class, initializes it, and invokes the public class.
Method void main(String []).

The Class Loading Overview

- Class Loading is one of the most important mechanisms provided by the Java platform.
- Understanding the concept of how to load the Java class can help debug exceptions such as the `ClassNotFoundException` and `ClassCastException`.
- In the Java runtime, a class will usually have its code contained in a `.class` file, though we can obtain the class from the network or generate on the fly.
- `.class` files are not loaded into memory all at once, but rather are loaded on demand, as needed by the program.

ClassLoader SubClass Example

- The ClassLoader is the part of the JVM that loads classes into memory.
- Suppose you develop a program :

```
class SuperClass{}  
Class InClass{}  
public class SubClass extends SuperClass{  
    public static void main(String[] args){  
        System.out.println("welcome to the SubClass main method and the  
        classloader used is: \n" + SubClass.class.getClassLoader() );  
        InClass ic=new InClass();  
    }  
}
```

ClassLoader SubClass Example

- When we try to run the following command:
`# java SubClass`
- The initial attempt to execute the main method of the class SubClass discovers that the Test class (InClass) is not loaded - that is, the Java Virtual Machine does not currently contain a binary representation for this class.
- The Java Virtual Machine then uses a class loader to attempt to find such a binary representation.

ClassLoader SubSystem

- The Virtual machine loads only those class files that are needed for the execution of a program.
- Here are the steps the Java Virtual Machine executes:
 1. The JVM has a mechanism for loading the Class. In this case, the Class is read from the file in the local disk.
 2. If the SubClass class has fields or have an inheritance relationship with other classes, ie SuperClass, the SuperClass is loaded also.

ClassLoader SubSystem

(cont):

3. After SubClass is loaded, it must be initialized before the main method can be used. SubClass must also be linked before it is initialized. Linking involves verification, preparation, and (optionally) resolution.
4. The JVM then executes the main method in Subclass
5. If the main method or a method that main method calls requires additional classes, these classes will be loaded next.

ClassLoader Loading Procedure

- We can check the class load order for the previous example using the command :

Java -verbose:class SubClass

```
[Loaded SuperClass from file:/C:/test/]
[Loaded SubClass from file:/C:/test/]
[Loaded sun.launcher.LauncherHelper$FXHelper from C:\Program Files\Java\jre1.8.0_10
[Loaded java.lang.Class$MethodArray from C:\Program Files\Java\jre1.8.0_101\lib\rt.
[Loaded java.lang.Void from C:\Program Files\Java\jre1.8.0_101\lib\rt.jar]
welcome to the SubClass main method and the classloader used is:
sun.misc.Launcher$AppClassLoader@2a139a55
[Loaded InClass from file:/C:/test/]
```

- You can see that SuperClass will be loaded first and SubClass next, the InClass will be loaded last.

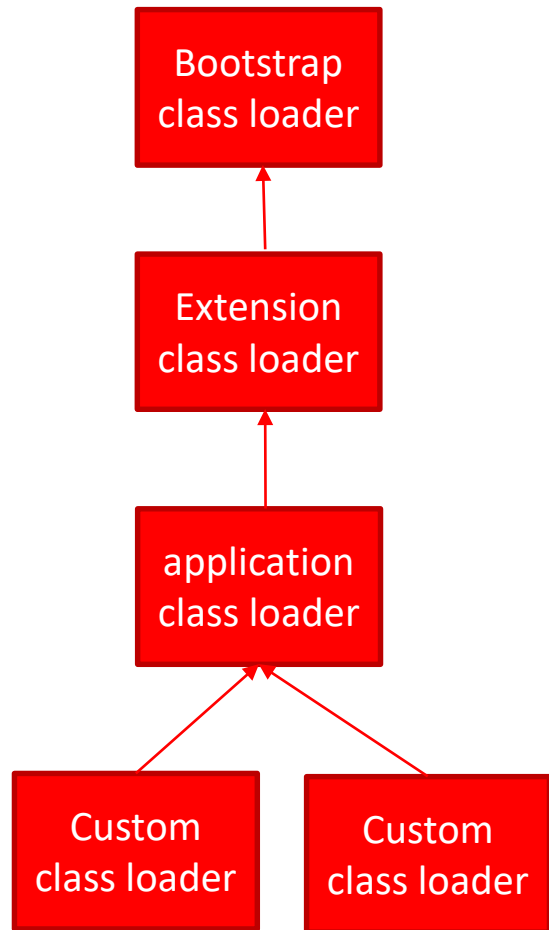
JDK ClassLoader Class

- Java Class loading mechanism has three basic class loaders and we can develop new class loading methods following the requirements.
- Every java program has at least three class loaders:
 - The bootstrap class loader
 - The extension class loader
 - The application class loader

JDK ClassLoader Class

- The bootstrap class loader loads the system class, usually from the rt.jar included in the JDK.
- The extension class loader loads the extension jar file from the jre/lib/ext directory.
- The application class loader loads the application class which can come from a directory or JAR file on the class path.
- In the Oracle JDK, both the extension and application class loaders are implemented from URLClassLoader class.

ClassLoader Hierarchy



- Class loaders have a parent/child relationship.
- Every class except for the bootstrap loader has a parent class loader.
- A class loader is supposed to give its parent a chance to load any given class and only try to load in case the parent failed to load.

ClassLoader in Java Runtime

From the JVM point of view:

- There are two kinds of class loaders: the bootstrap class loader supplied by the Java Virtual Machine, and user-defined class loaders.
- Every user-defined class loader is an instance of a subclass of the abstract class `ClassLoader`.
- The loading of class process is implemented by the class `ClassLoader` and its subclasses.

ClassLoader in Java Runtime

- Loading refers to the process of finding the binary form of a class or interface type with a particular name.
- Perhaps by computing it on the fly, but more typically by retrieving a binary representation previously computed from source code by a Java compiler.
- And then constructing from that binary form, a Class object to represent the class or interface.

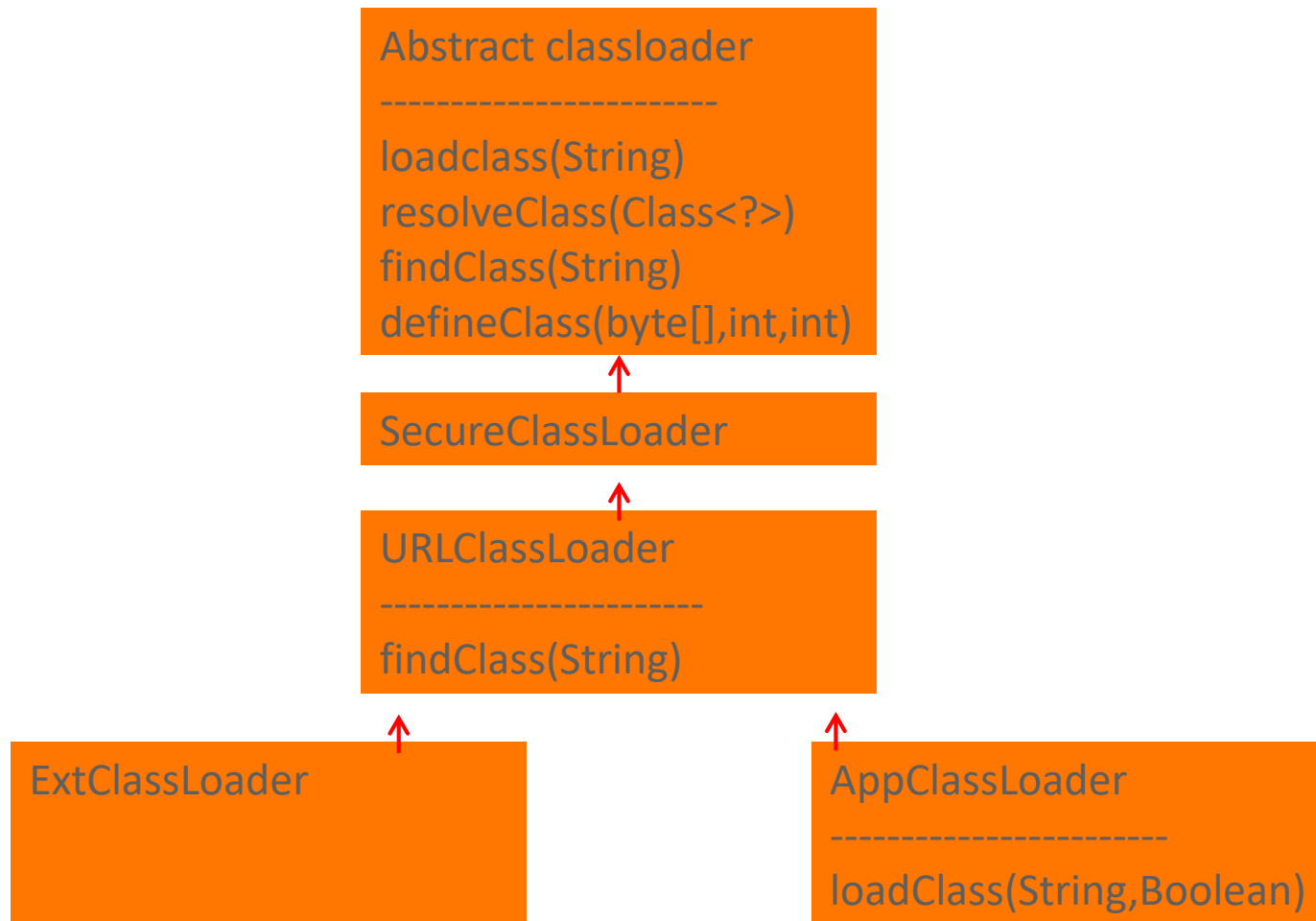
ClassLoader Class Example

- From the SubClass example, All of SubClass, SuperClass and InClass will be loaded by the ApplicationClassLoader.

```
welcome to the SubClass main method and the classloader used is:  
sun.misc.Launcher$AppClassLoader@2a139a55
```

- This Loader will check the -classpath option or CLASSPATH variable to search and load the classes.

ClassLoader Class Diagram



Custom ClassLoader

```
import java.lang.reflect.Method;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;
public class ClassLoaderDemo
{
    final static String DiskURL = "file:/// " + System.getProperty("user.dir") + "/classdemo/";
    public static void main(String[] args)
    {
        try
        {
            URL[] urls = new URL[] { new URL(DiskURL) };
            URLClassLoader loader = new URLClassLoader(urls);
            Class<?> clazz = Class.forName("LoadingExample", true, loader);
            System.out.println(clazz.getClassLoader());
            runTest(clazz);
        }
        catch (Exception e)
        { }
    }
}
```

Custom ClassLoader

```
static void runTest(Class<?> clazz)
{
    try
    {
        Method main = clazz.getMethod("main", new Class[] { String[].class });
        Object[] args = new Object[] { new String[0] };
        main.invoke(null, args);
    }
    catch (Exception e)
    {
        System.err.println(e.getMessage());
    }
}
```

Using the Java reflection API to invoke the method defined in the class.

Custom ClassLoader

```
public class LoadingExample{  
  
    public LoadingExample(){  
    }  
  
    public static void main(String[] args){  
        System.out.println("LoadingExample class loaded and instantiated");  
    }  
}
```

```
PS C:\test> java ClassLoaderDemo  
Picked up JAVA_TOOL_OPTIONS: -Duser.language=en  
java.net.URLClassLoader@7852e922  
LoadingExample class loaded and instantiated
```

Class Linking

- Linking is the process of taking a binary form of a class or interface type and combining it into the run-time state of the Java Virtual Machine, so that it can be executed.
- A class or interface type is always loaded before it is linked.
- Three different activities are involved in linking:
 - Verification
 - Preparation
 - Resolution of symbolic references

Class Linking - Verification

- When a class loader presents the newly loaded binary representation of a Java Class to the Java Runtime, these binary representations are first inspected by a verifier which checks that the binary contents cannot perform actions that are damaging to the JVM.
- Verification ensures that the binary representation of a class or interface is structurally correct.

Class Linking - Verification

- If the binary representation of a class or interface fails to satisfy the requirement of the Verifier, then a Error will be thrown.
- Here are some of checks:
 - The opcode of the first instruction in the code array begins at index 0
 - Ensure the final classes are not subclasses

Class Linking - Preparation

- Preparation involves the allocation of static storage and any data structures that are used internally by the implementation of the Java Virtual Machine, such as method tables.
- Preparation involves creating the static fields (class variables and constants) for a class or interface and initializing such fields to the default values.

The Linking - Resolution

- Resolution is the process of dynamically determining concrete values from symbolic references in the run-time constant pool.
- The following Java Virtual Machine instructions make symbolic references to the run-time constant pool :
 - anewarray, checkcast, getfield, getstatic, instanceof, invokedynamic, invokeinterface, invokespecial, invokestatic, invokevirtual, ldc, ldc_w, multianewarray, new, putfield, putstatic
- Execution of any of these instructions requires resolution of its symbolic reference.

Class Initialization

- Linking involves verification, preparation, and, where required, resolution.
- Initialization consists of execution of any class variable initializers and static initializers of the class Test, in textual order.
- Verification checks that the loaded representation of Test is well-formed, with a proper symbol table.
- Verification also checks that the code that implements Test obeys the semantic requirements of the Java programming language and the Java Virtual Machine.

Class Initialization Example

```
public class HelloClass{  
    static{  
        System.out.println("HelloClass has been initialized");  
    }  
    public void sayHello(){  
        System.out.println("Hello Class Loader");  
    }  
}
```

Class Initialization Example - Cont

```
static {};  
  descriptor: ()V  
  flags: ACC_STATIC  
  Code:  
    stack=2, locals=0, args_size=0  
    0: getstatic    #2  
    3: ldc          #5  
    5: invokevirtual #4  
    8: return  
  LineNumberTable:  
    line 4: 0  
    line 5: 8
```

- The static initializers in the Java source code are translated into static {} methods in the class file.
- The block of bytecode will be executed during the class initialization phase following the Class Link process.

Summary

In this lesson, you should have learned:

- The Class Loading Overview
- ClassLoader Loading Procedure
- JDK ClassLoader Class
- ClassLoader Hierarchy
- Custom ClassLoader
- Class Linking



