Web Servers
and
Java Class Loaders

Pr. Olivier Gruber

Université Grenoble Alpes

Web Server

- Serving static pages
  - HTTP GET → download a resource
  - A resource may be an HTML page, but it may be some other document (like a PDF file)
- Serving dynamic pages
  - Servlet concept: singleton object associated to a URL prefix
  - Multiple servlets can co-exist
- The notion of web applications
  - Each web application: {servlets} and {resources}
  - The servlets are working together to provide one consistent user experience

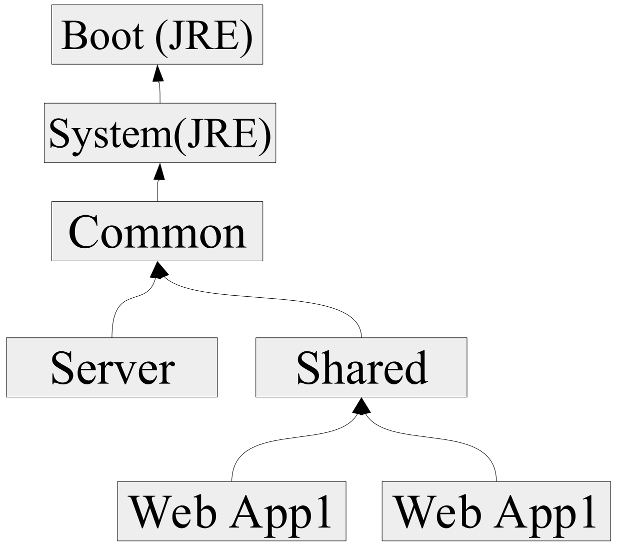
Web Server with Multiple Web Applications

- We could use different processes, each process running its own web server
  - But then we would need to use different ports in URLs
- We could use different machines
  - But then we would need to use different URLs altogether
- We could use a single web server
  - *Sounds great! But how can a single web server host several applications?*
  - *Safety concerns when running on a single Java Runtime Environment (JRE)*
    - *Same object graph… same static variables… same classpath… same classes…*
    - *Same security (access rights)...*
    - *Do we want to rely on developers doing the right thing…*

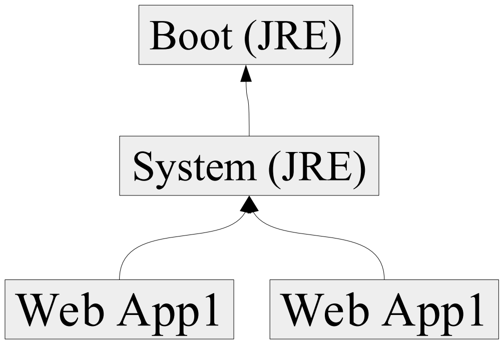
Does not sound so good, does it?

Using Class Loaders

- The idea is simple
  - Use one class loader per web application
  - Each application will load its own classes
- The reality is more complex
  - With Tomcat for example…
  - Even with our simpler use of class loaders...



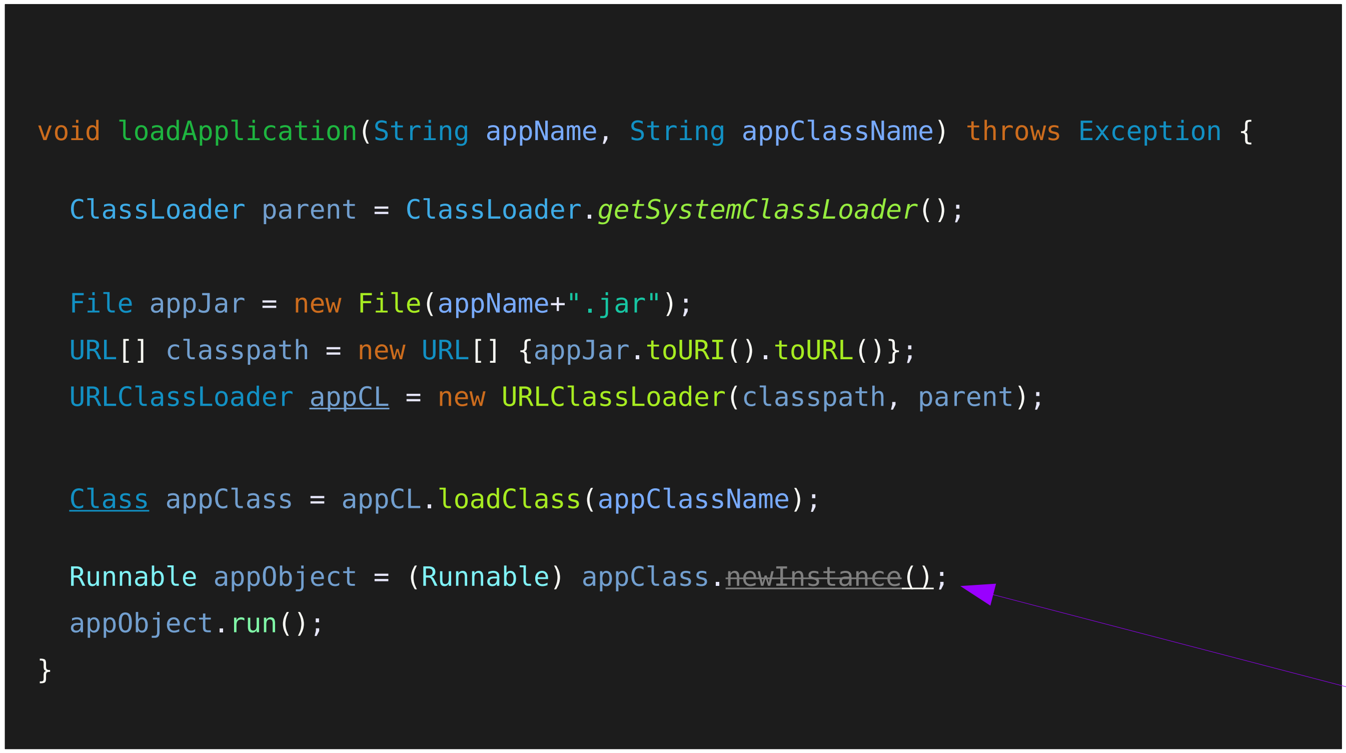
Tomcat Class Loaders [1]



(1) https://tomcat.apache.org/tomcat-9.0-doc/class-loader-howto.html

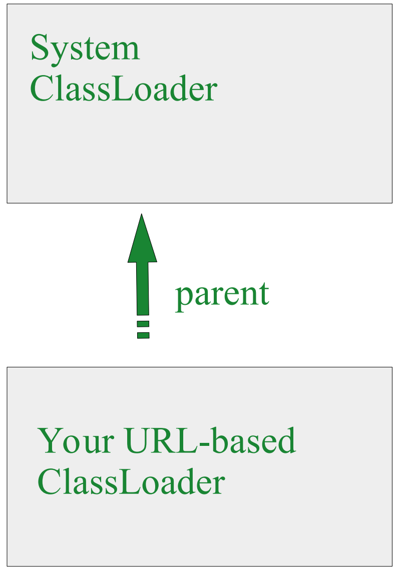
Class Loaders and Web Applications

Creating a URL-based class loader and using it to load a class
and then using that class to create an instance of your web application



```
void loadApplication(String appName, String appClassName) throws Exception {
  ClassLoader parent = ClassLoader.getSystemClassLoader();

  File appJar = new File(appName+".jar");
  URL[] classpath = new URL[] {appJar.toURI().toURL()};
  URLClassLoader appCL = new URLClassLoader(classpath, parent);

  Class appClass = appCL.loadClass(appClassName);

  Runnable appObject = (Runnable) appClass.newInstance();
  appObject.run();
}
```

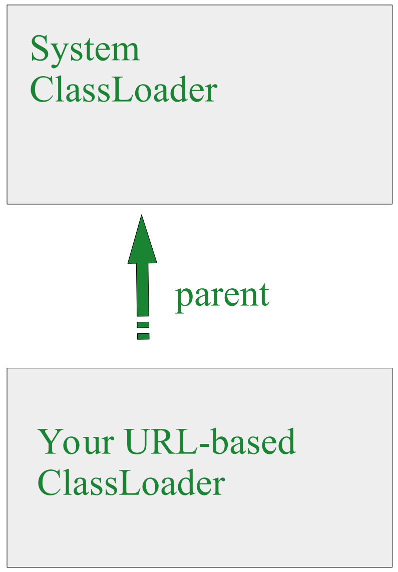
Deprecated method...



Class Loaders and Web Applications

The proper way using meta-programming...

```java
void loadApplication(String appName, String appClassName) throws Exception {

  ClassLoader parent = ClassLoader.getSystemClassLoader();

  File appJar = new File(appName+".jar");
  URL[] classpath = new URL[] {appJar.toURI().toURL()};
  URLClassLoader appCL = new URLClassLoader(classpath, parent);

  Class appClass = appCL.loadClass(appClassName);


  Class params[] = new Class[] {};
  Constructor ctor = appClass.getConstructor(params);


  Runnable appObject = (Runnable) ctor.newInstance();
  appObject.run();
}
```



Class Loaders - Rules and Pitfalls

- Classical Pitfalls
  - Two class loaders loading the “same class” yields two classes
    - Even when using the same class file!
  - Beware of equivalent names
    - Name equivalence does not mean a thing between class loaders
    - **Same class name** does not mean **the same class**
  - Debugging
    - The debugger does not show class loaders and class objects… just names...
    - So you can have class cast exceptions although the class names seem OK...
- Important Rules

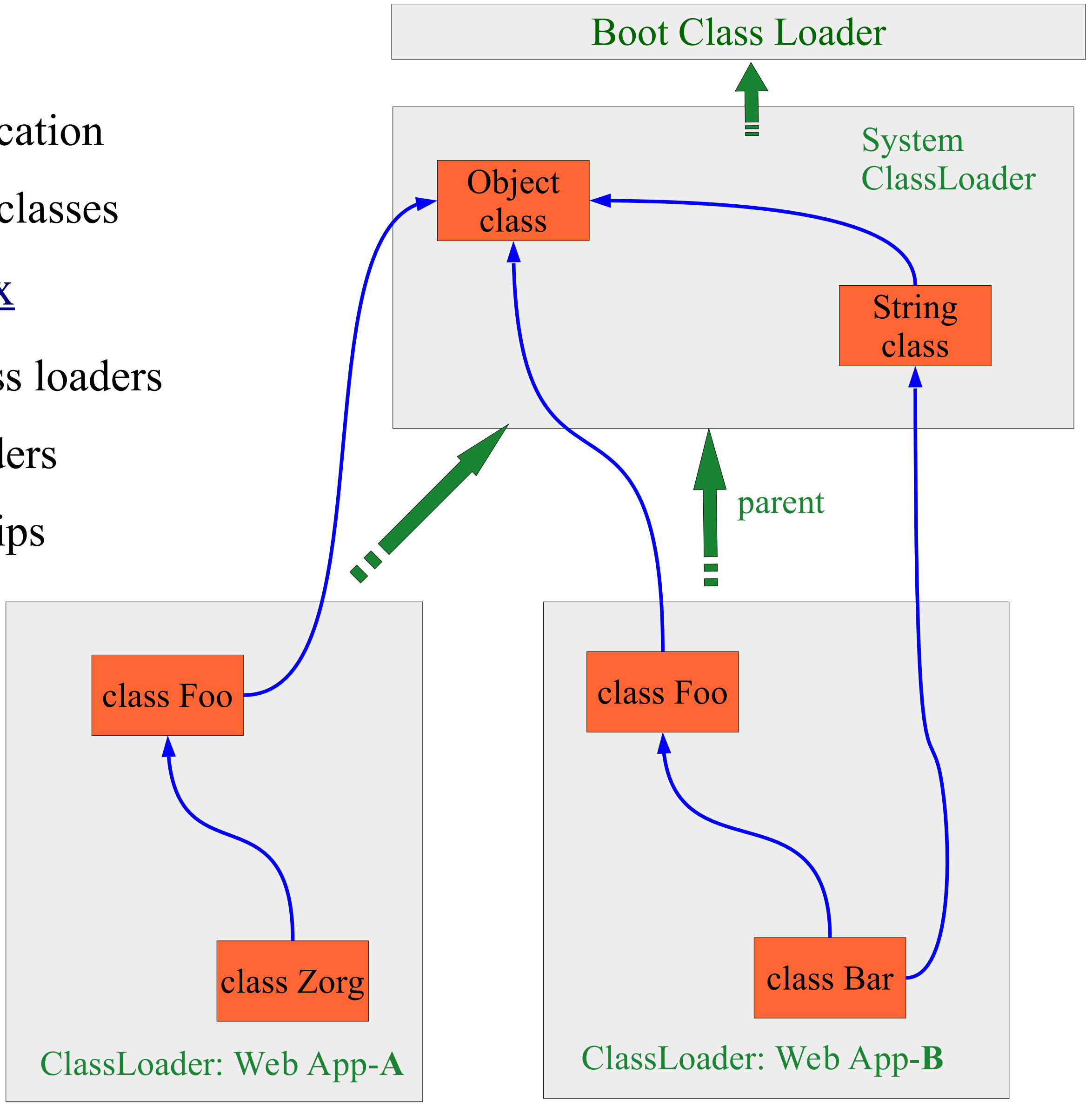
Rule 1: two classes are the same if and only if they are the same class object

Rule 2: one class object belongs to one and only one classloader

Rule 3: lazy loading… the runtime loads classes as it needs them for the execution

Using Class Loaders

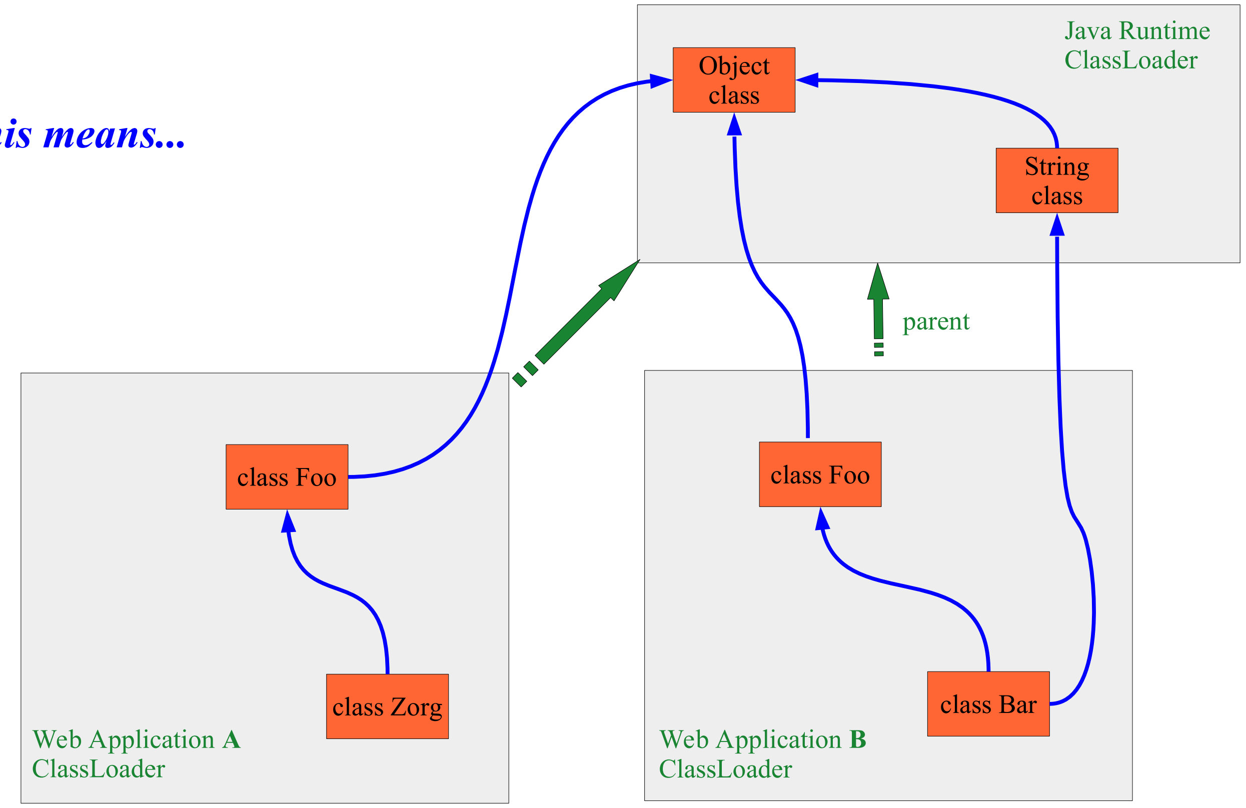
- The idea is simple
  - Use one class loader per web application
  - Each application will load its own classes
- How it works is bit more complex
  - Classes are loaded by different class loaders
  - Classes are linked across class loaders
  - Based on ***extend***/***import*** relationships



Using Class Loaders

- What the heck is a class loader?
- What is this graph of classes all about?
- Are we saying that classes are objects?
- As in the object graph contains objects and classes?

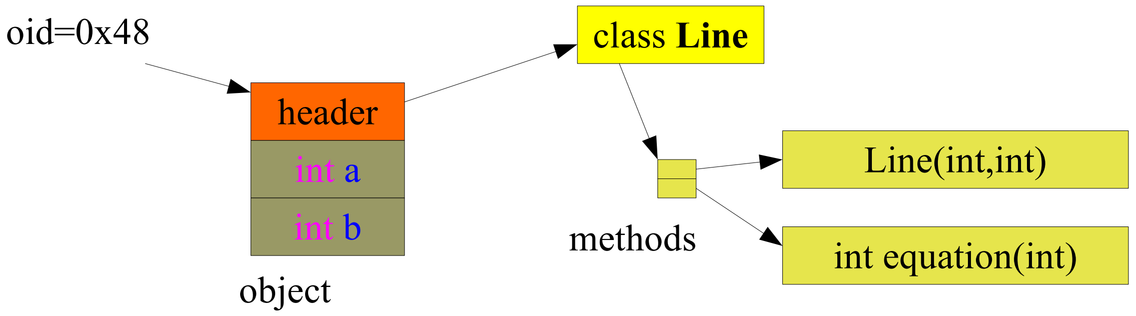
Yup, so let's understand what this means...



Java Classes @ Runtime

- A class is an object at runtime
- That describes the structure and behavior of its instances
- Created when loading a class file

```
public class Line {
  int a;
  int b;

  Line(int a, int b) {
    this.a = a;
    this.b = b;
  }

  int equation(int x) {
    return a * x + b;
  }

  static void snippet() {
    int x,y;
    Line line = new Line(2,3);
    x = 5;
    y = line.equation(x);
  }
}
```



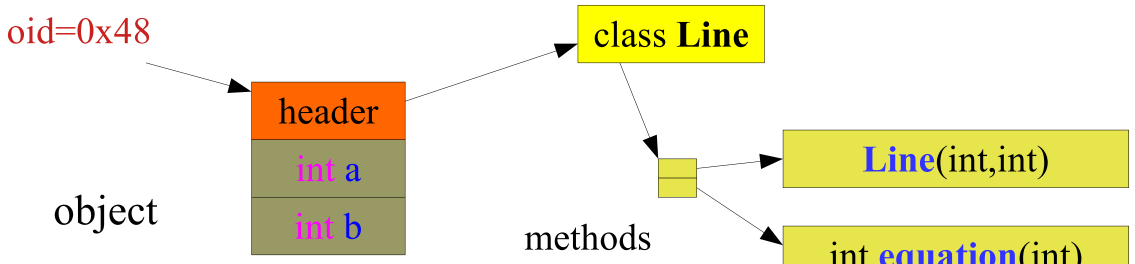
A simple graph of objects, with one object and its class...

Java Classes @ Runtime

- Classes are used during the execution...

```
static void snippet();
  0  new Line                         ; new Line → oid=0x48
  3  dup                              ; duplicate the oid
  4  iconst_2                         ; load  #2
  5  iconst_3                         ; load  #3
  6  invokespecial Line(int, int)     ; invoke constructor Line(2,3)
  9  astore_2                         ; store in 'line'
  10  iconst_5                        ; load 5
  11  istore_0                        ; store in 'x'
  12  aload_2                         ; load 'line'
  13  iload_0                         ; load [x]
  14  invokevirtual Line.equation(int) : int  ; invoke virtual method
  17  istore_1                        ; store in 'y'
  18  return
```

```java
public class Line {
  int a;
  int b;

  Line(int a, int b) {
    this.a = a;
    this.b = b;
  }

  int equation(int x) {
    return a * x + b;
  }

  static void snippet() {
    int x,y;
    Line line = new Line(2,3);
    x = 5;
    y = line.equation(x);
  }
}
```



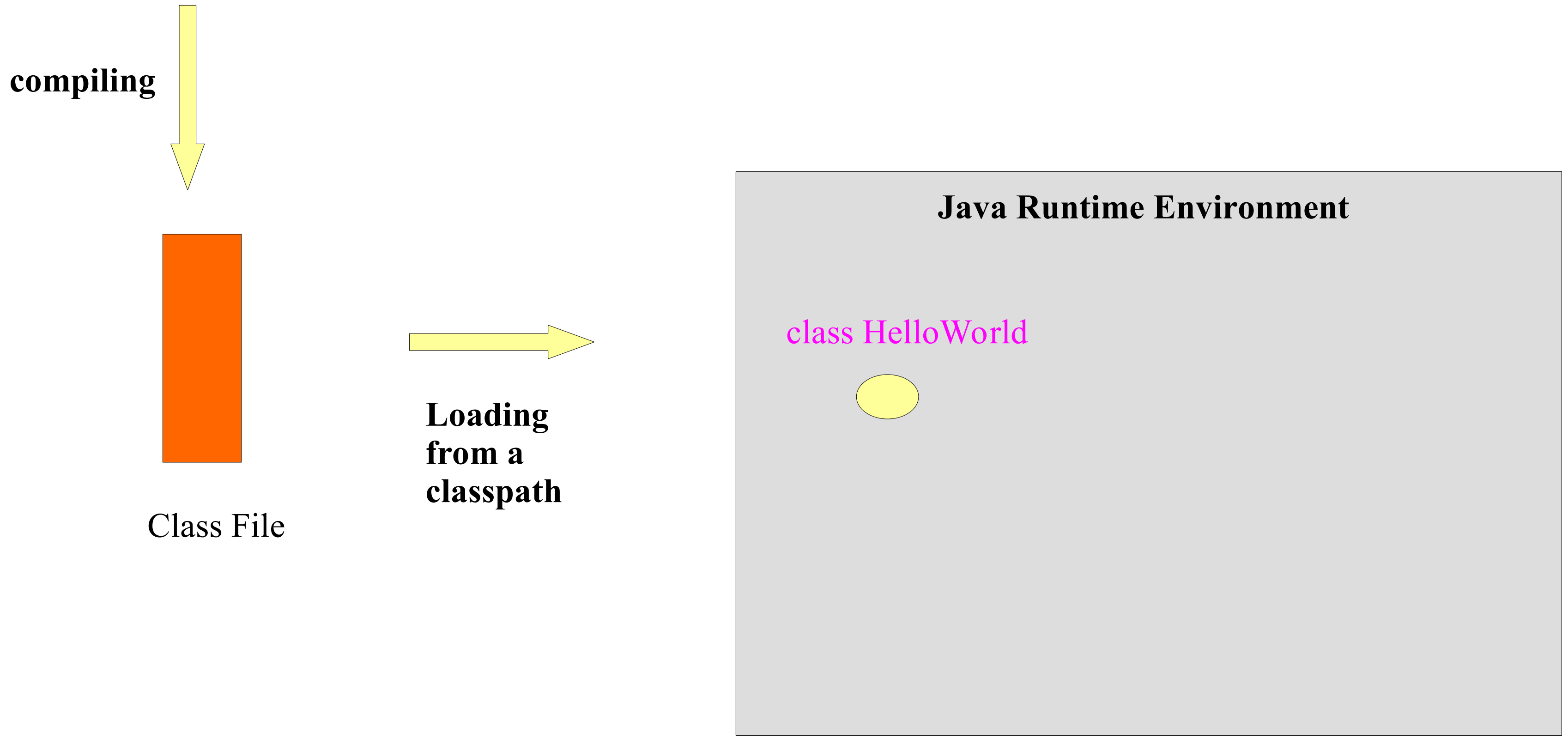
Java Class *Compiling* and *Loading*

```
class HelloWorld {
  public static void main(String args[]) {
     HelloWord hw = new HelloWorld();
  }
   HelloWorld() {
     System.out.println("Hello World!");
  }
}
```

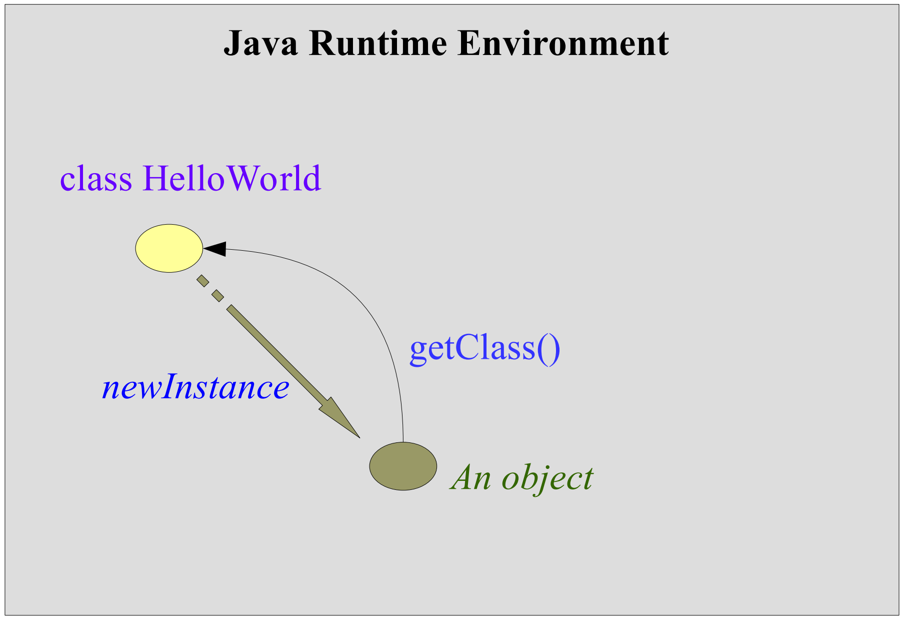
So what is next after loading the class HelloWorld?

To execute the method "main"…

That will create an instance of that class...



Java Class Loading



```
class Object {
  Class getClass();
  ...
}

class Class extends Object {
  Object newInstance();
}
```

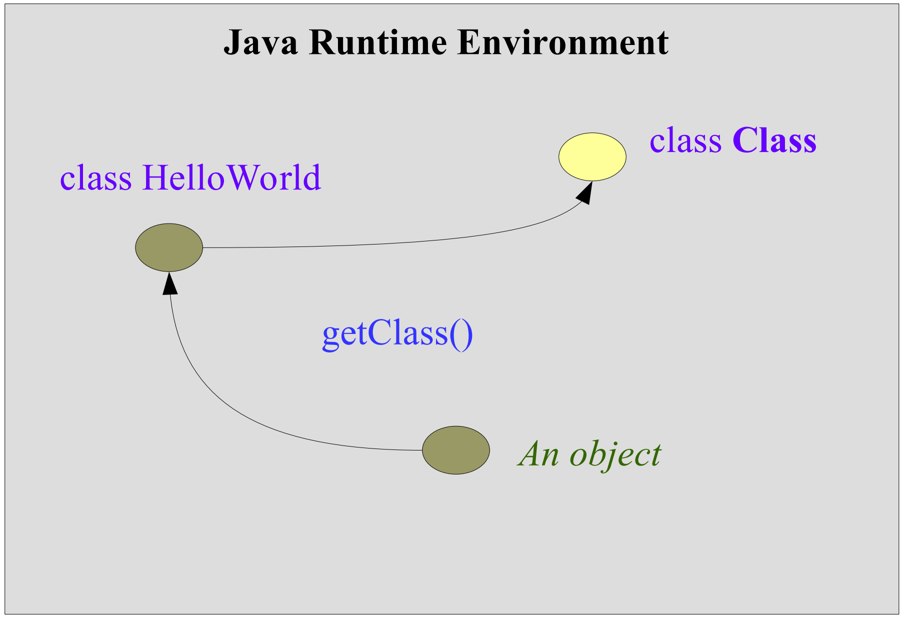
Any Java object knows its class.
A class is an object as well.

So… a class has a class?

*Yes, the class **Class***

*And a class is a **factory** for its intances*

Java Class Loading



```
class Object {
  Class getClass();
  ...
}

class Class extends Object {
  Object newInstance();
}
```

Any Java object knows its class.
A class is an object as well.

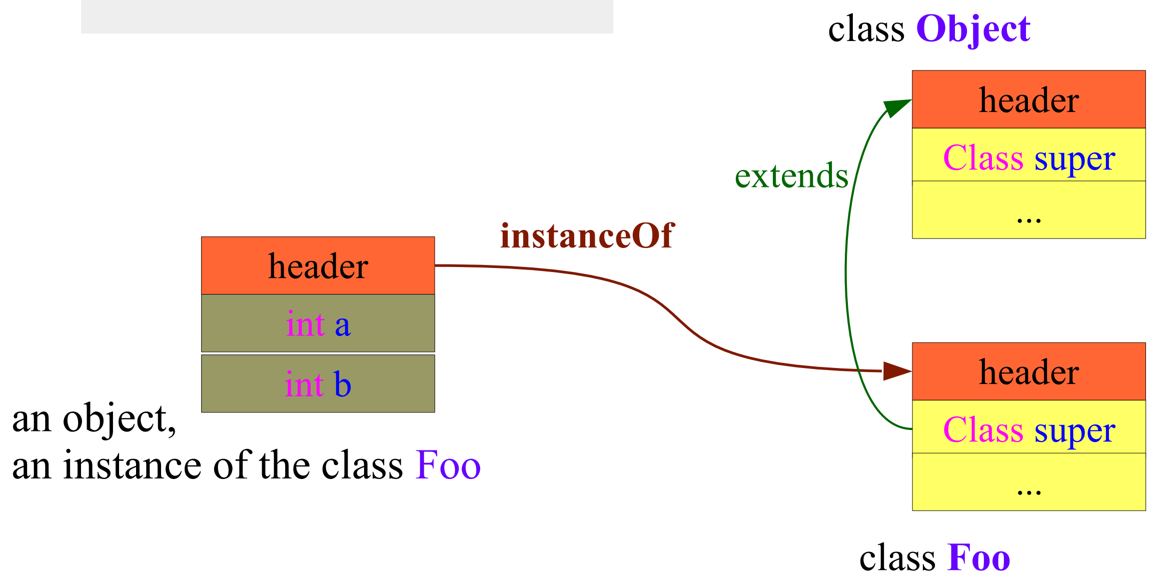
So… a class has a class?

*Yes, the class **Class***

*And a class is a **factory** for its intances*

Object Graph and Classes

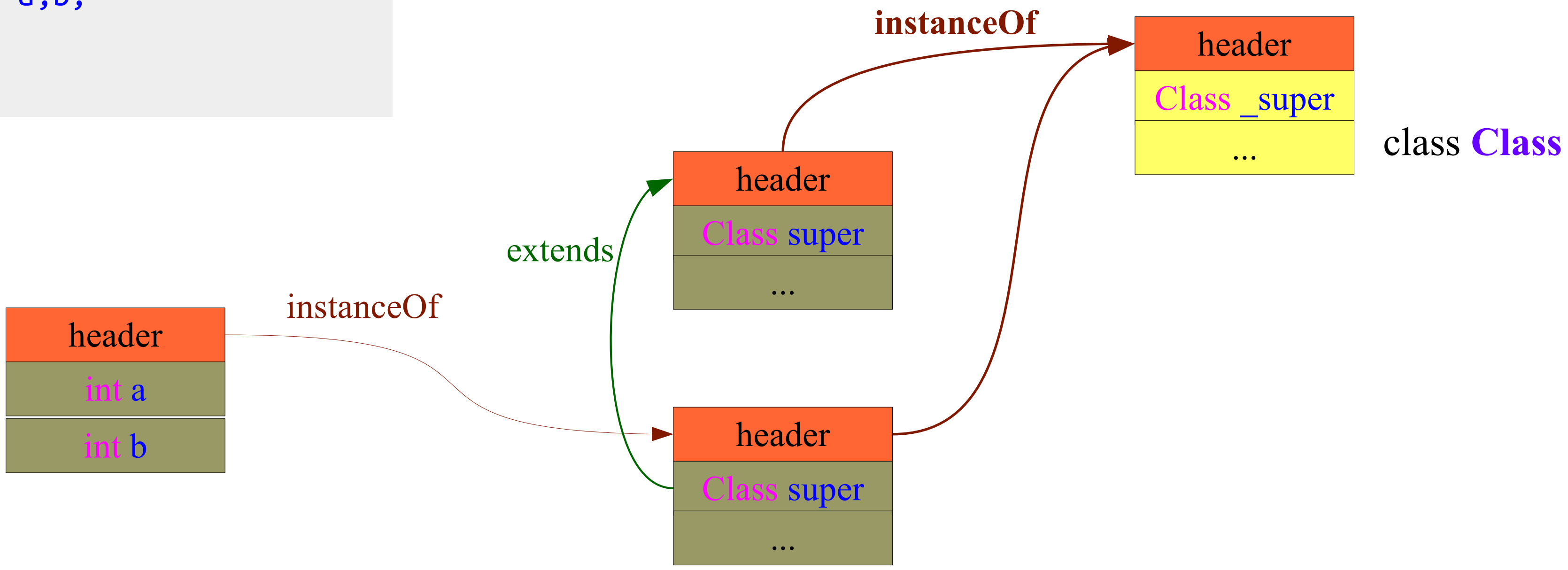
```
class Object {
  Class getClass();
  ...
}

class Foo extends Object {
  int a,b;
  ...
}
```



Object Graph and Classes

```
class Object {
  Class getClass();
  ...
}

class Foo extends Object {
  int a,b;
  ...
}
```

```
class Class extends Object {
  Object newInstance();
}
```

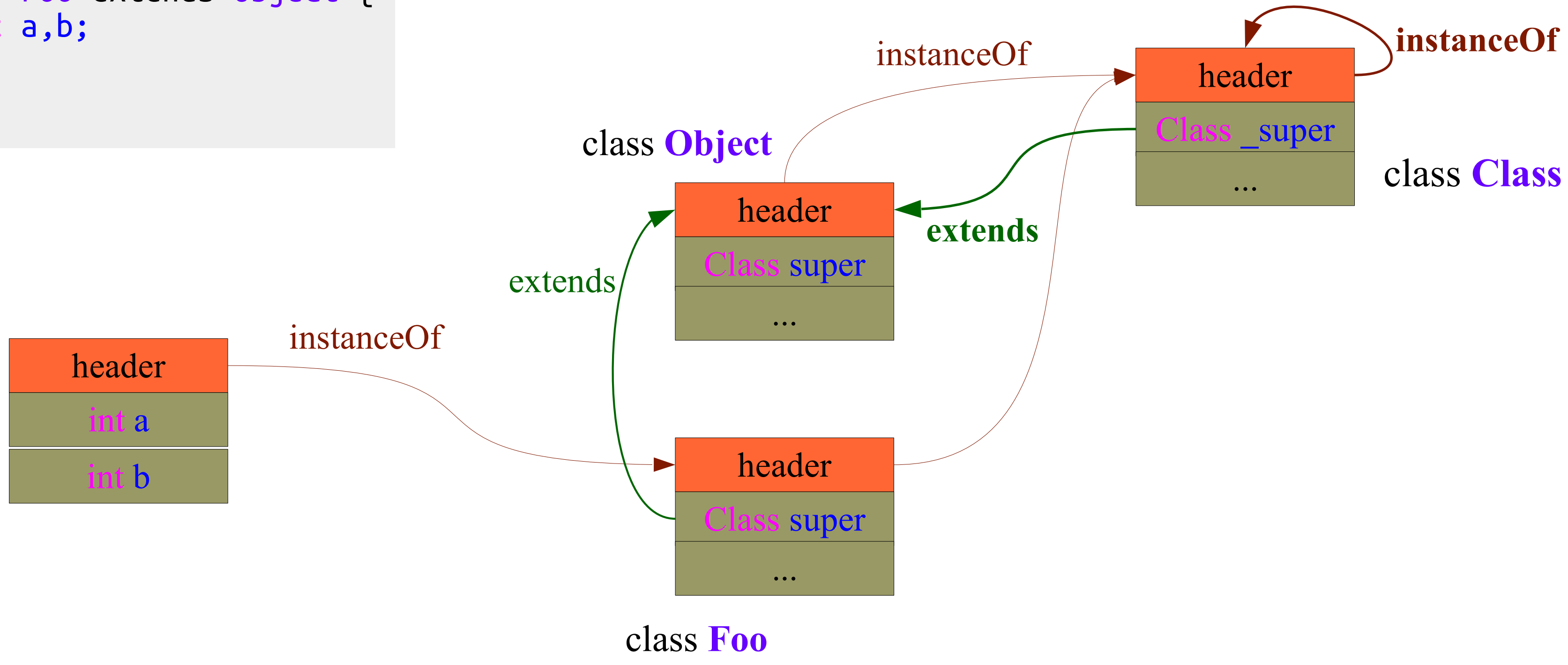


Two *"class"* objects,
instances of the class Class

Object Graph and Classes – "*Everything is an Object*"

```
class Object {
  Class getClass();
  ...
}

class Foo extends Object {
  int a,b;
  ...
}
```

```
class Class extends Object {
  Object newInstance();
}
```

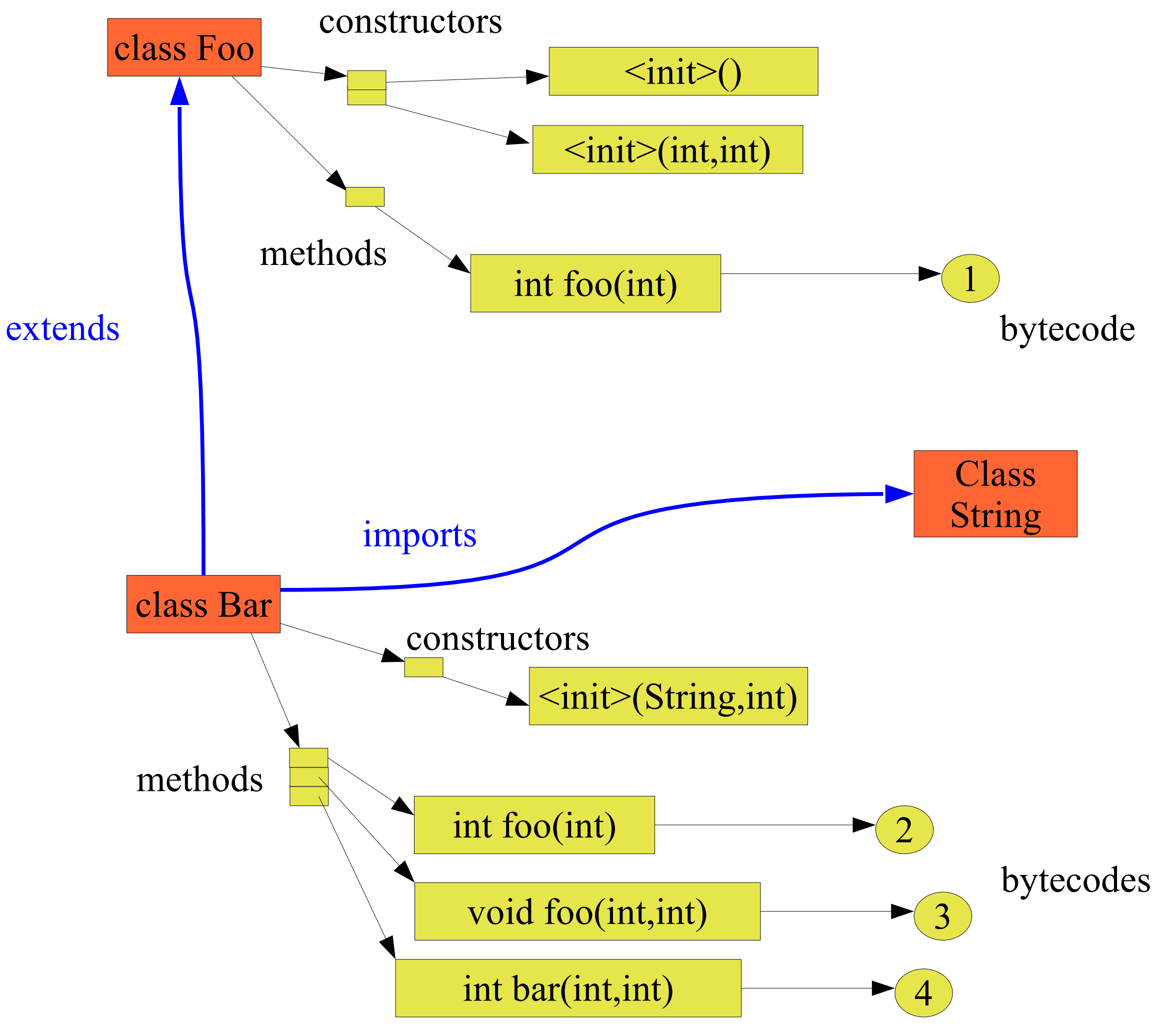


And now everything is an object…
And every object has a class object…
And all classes extends the class Object…
And so everything is an object...

Java Classes – A Complete Description @ Runtime

```
class Foo {
  int a;
  int b;
  Foo() {...}
  Foo(int a, int b) {...}

  int foo(int x) { 1 }
}
```

```
class Bar extends Foo {
  int b;
  String c;

  Bar(String c, int b) { ... }

  int foo(int x) { 2 }

  void foo(int x, int y) { 3 }

  int bar(int x, int y) { 4 }
}
```

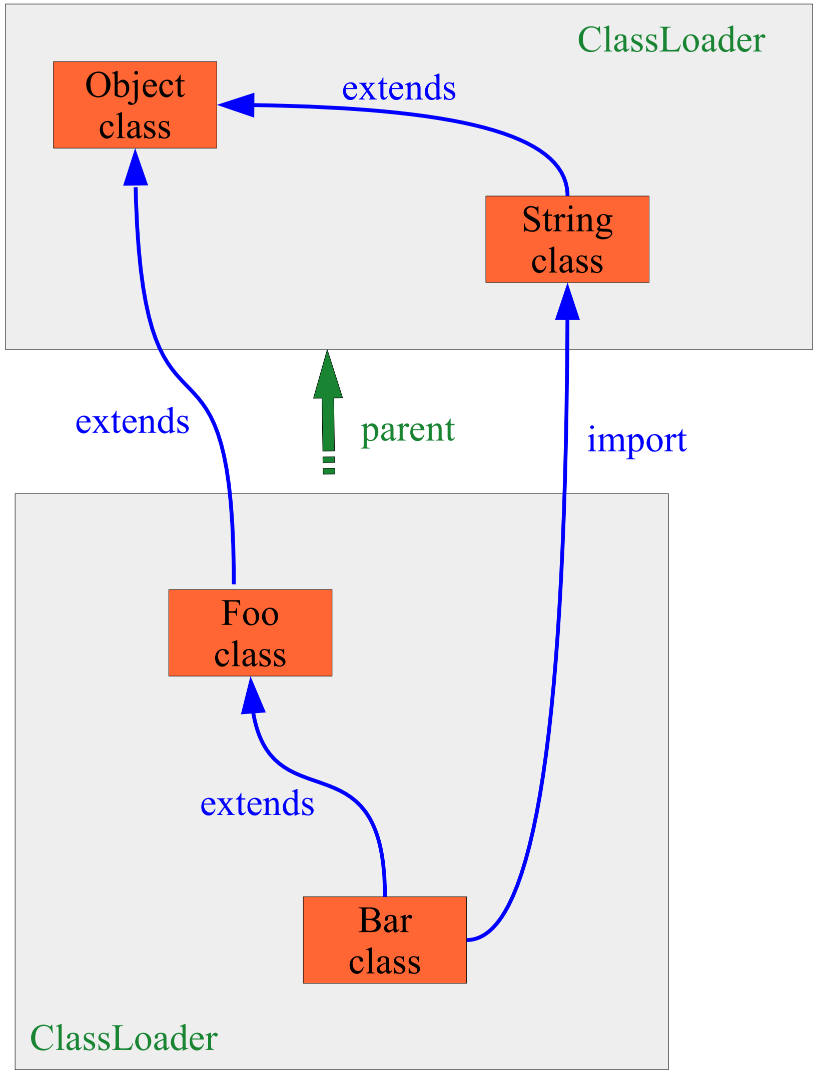


Class Loaders – Grouping Classes

- A tree of class loaders
  - Classes in parent class loaders are visible and they have precedence
  - Classes are linked with classes loaded in the parent class loaders (via the *extend*/*import* relationships)
- Points to Remember
  - Compile-time relationships must agree with runtime relationships (buildpath versus classpath)
  - Class loading is lazy, at runtime, that is, classes are loaded as the execution needs them

Why is that important?

Because class loading errors may show up late in the execution…. Missing classes or Incompatible classes...



Rappel – Class Loader Rules and Pitfalls

- Classical Pitfalls
  - Two class loaders loading the “same class” yields two classes
    - Even when using the same class file!
  - Beware of equivalent names
    - Name equivalence does not mean a thing between class loaders
    - **Same class name** does not mean **the same class**
  - Debugging
    - The debugger does not show class loaders…
    - So you can have class cast exceptions although the class names seem OK...
- Important Rules

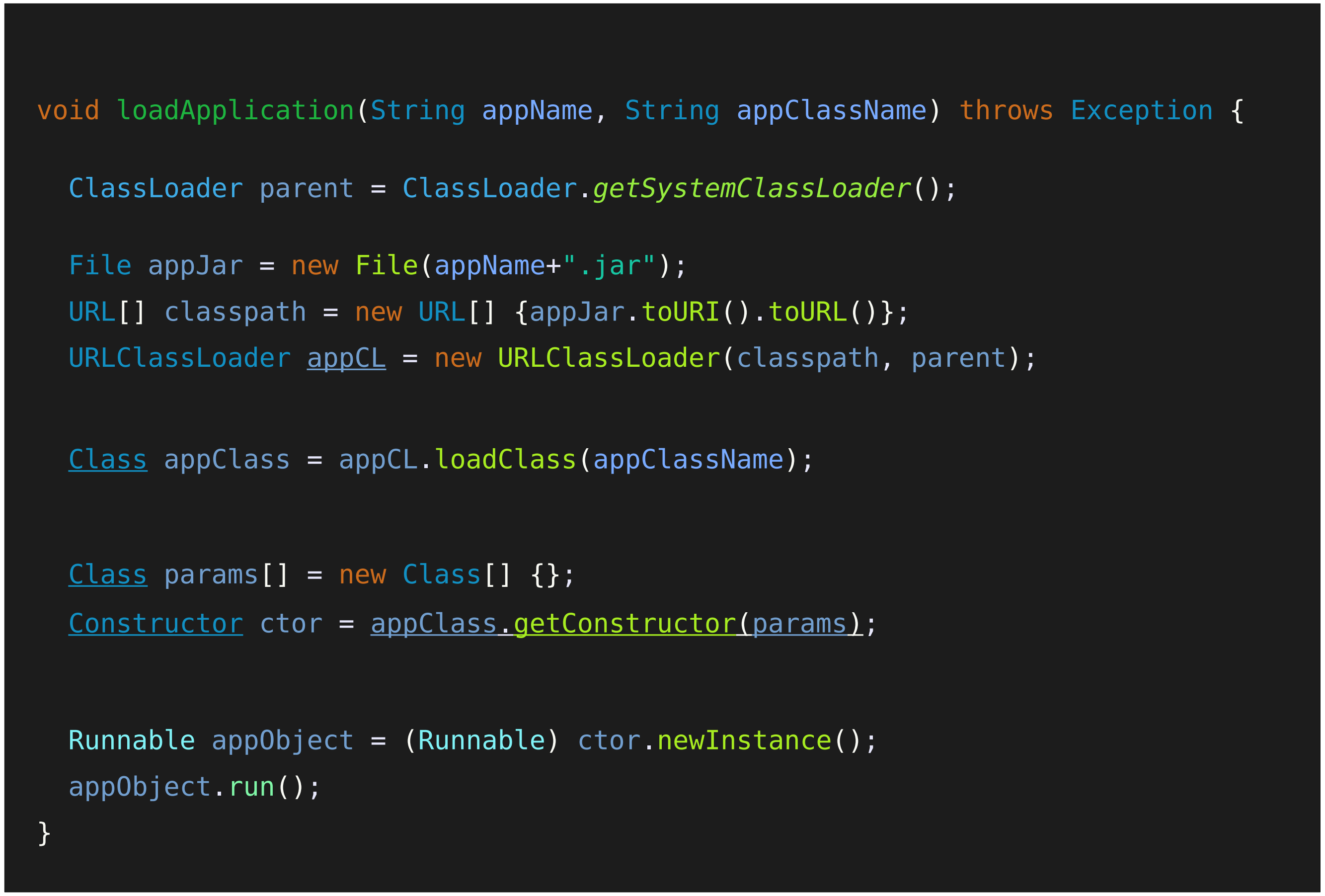
  Rule 1: two classes are the same if and only if they are the same class object

  Rule 2: one class object belongs to one and only one classloader

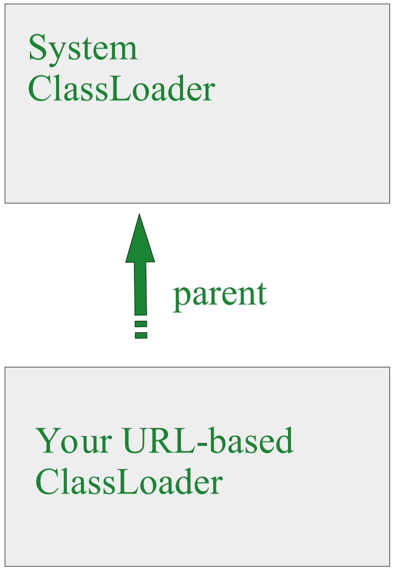
  Rule 3: lazy loading… the runtime loads classes as it needs them for the execution

Rappel – Class Loaders and Web Applications

Creating a class loader and using meta-programming...



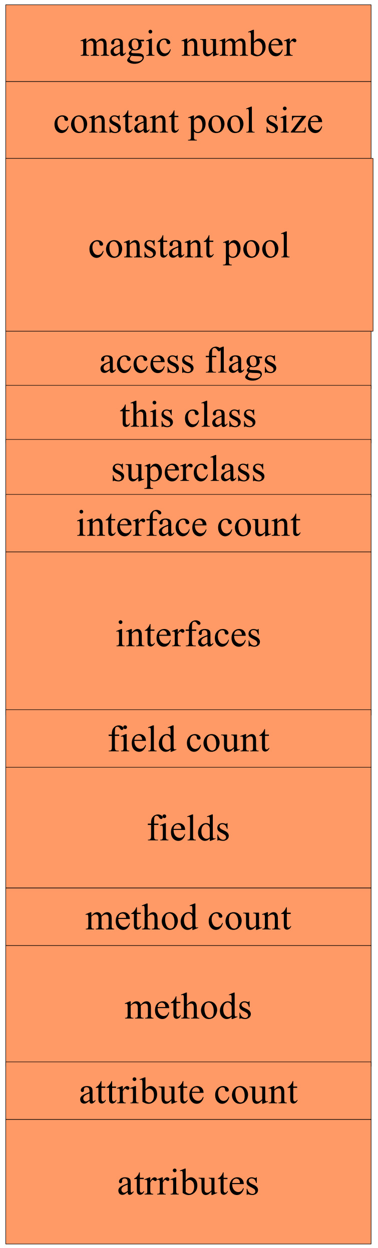
```java
void loadApplication(String appName, String appClassName) throws Exception {

  ClassLoader parent = ClassLoader.getSystemClassLoader();

  File appJar = new File(appName+".jar");
  URL[] classpath = new URL[] {appJar.toURI().toURL()};
  URLClassLoader appCL = new URLClassLoader(classpath, parent);

  Class appClass = appCL.loadClass(appClassName);


  Class params[] = new Class[] {};
  Constructor ctor = appClass.getConstructor(params);


  Runnable appObject = (Runnable) ctor.newInstance();
  appObject.run();
}
```



Extra Slides

Class Loaders – Class Loading

- Class Loading only through the class file format
  - Only the JVM can create classes through a native method
    - The native method **ClassLoader.define(...)**
    - Passing the byte array of a class file to define the described type
  - The class file is an exchange format
    - **Usually produced by Java compilers and consumed by class loaders**
- But a quite open approach to class loading
  - Loaded from the file system
  - Or downloaded from a URL
  - It can be weaved for different purposes
  - Or it can be even synthetic



Classfile Examples

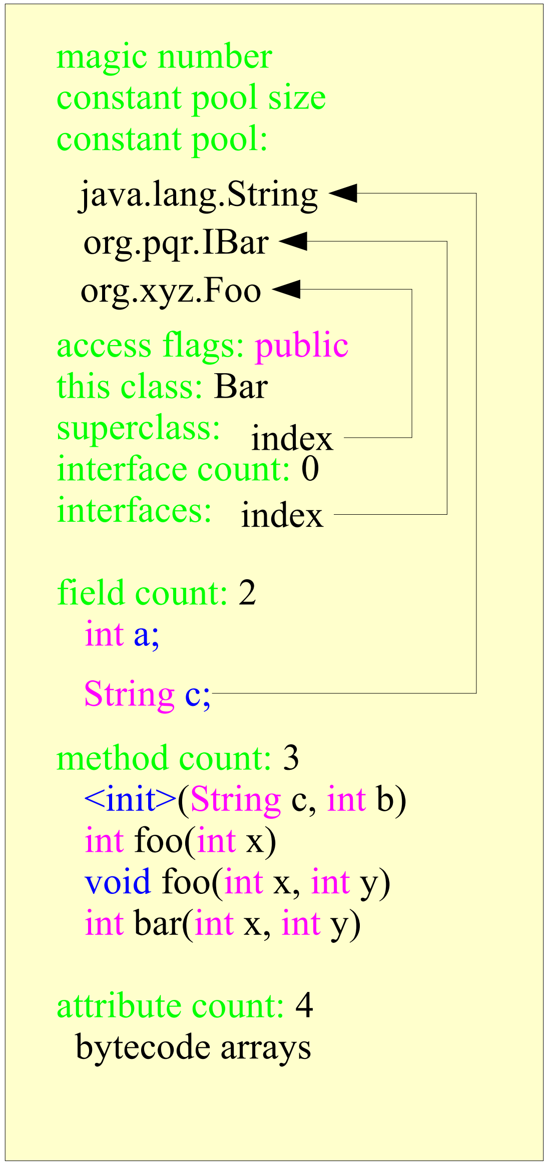
```
public class Line {
  int a;
  int b;
  Line(int a, int b) {
    this.a = a; this.b = b;
  }
  int equation(int x) {
    return a*x+b;
  }
  public String toString() {
    return “a line“;
  }
}
```

```
magic number
constant pool size
constant pool:
  “a line“
  java.lang.Object ◄──────┐
                          │
access flags: public      │
this class: Line          │
superclass:     Index ────┘
interface count: 0
interfaces:

field count: 2
   int a;
   int b;
method count: 3
   <init>(int a, int b)
   int equation(int x)
   public String toString()

attribute count: 3
  bytecode arrays
```

Classfile Examples

```
package org.xyz;

public class Foo {
   int a;
   int b;

   Foo(int a, int b) {...}

   int foo(int x) {...}

}
```

```
package org.pqr;

import org.xyz.Foo;

public class Bar extends Foo
   implements  IBar {
   int b;
   String c;

   Bar(String c, int b) { ... }

   int foo(int x) {... }
   void foo(int x, int y) {... }

   int bar(int x, int y) { ... }
}
```



```
magic number
constant pool size
constant pool:
  java.lang.String
  org.pqr.IBar
  org.xyz.Foo
access flags: public
this class: Bar
superclass:  index
interface count: 0
interfaces:  index

field count: 2
   int a;

   String c;

method count: 3
   <init>(String c, int b)
   int foo(int x)
   void foo(int x, int y)
   int bar(int x, int y)

attribute count: 4
  bytecode arrays
```