Java Class Loading and Bytecode Verification

- Every object is a member of some class.

- The Class class: its members are the (definitions of) various classes that the JVM knows about.

- The classes can be dynamically loaded by the JVM by reading local or remote class files.

- Loading of classes is done by class loaders which are objects of the ClassLoader class.

- The class loader coordinates with the security manager and the access controller to provide the sandbox functions.

```
public class getClassTest {
  public static void main (String args[]) {
    String s = "abc";
    Class c1 = s.getClass();
    System.out.println ("string \"" + s + "\" is of class " + c1.getName());
    Class c2 = c1.getClass();
    System.out.println ("class " + c1.getName() + " is of class " + c2.getName());
    Class c3 = c2.getClass();
    System.out.println ("class " + c2.getName() + " is of class " + c3.getName());
  }
}
```

```
public class getClassTest {
  public static void main (String args[]) {
    String s = "abc";
    Class c1 = s.getClass();
    System.out.println ("string \"" + s + "\" is of class " + c1.getName());
    Class c2 = c1.getClass();
    System.out.println ("class " + c1.getName() + " is of class " + c2.getName());
    Class c3 = c2.getClass();
    System.out.println ("class " + c2.getName() + " is of class " + c3.getName());
  }
}
```

```
string "abc" is of class java.lang.String
class java.lang.String is of class java.lang.Class
class java.lang.Class is of class java.lang.Class
```

An example involving dynamic class loading

```
import java.lang.reflect.*;

public class runhello {
  public static void main (String args[]) {
    Class c = null;
    Method m = null;

    // First we load the required class into the JVM
    try { c = Class.forName ("hello");
    } catch (ClassNotFoundException e) {
      System.out.println ("The class was not found");
    };
```

```
    // Get the main method of the class
    Class argtypes[] = new Class[] { String[].class };
    try { m = c.getMethod ("main", argtypes);
    } catch (NoSuchMethodException e) {
      System.out.println ("The main method was not found");
    };

    // Invoke the method
    Object arglist[] = new Object[1];
    try { m.invoke (null, arglist);
    } catch (Exception e) {
      System.out.println ("Error upon invocation" + e);
    };
} }
```

Hello!

The forName function finds, loads and links the class specified by the name.

forName(String name, boolean initialize, ClassLoader loader)

tries to find the class specified by the name, load it using the specified class loader and link it. The class is initialized if asked for.

forName ("hello")

above is equivalent to

forName ("hello", true, this.getClass().getClassLoader())

Security and the class loader

The security manager and access controller allow or prevent various operations depending upon the context of the request.

This information is provided by the class loader.

The class loader has information about

- origin: where the class was loaded from
- whether the class comes from the local filesystem or from the network
- whether the class comes with a digital signature

- Suppose we visit a website www.site1.com which uses a class named C1. Then we visit a second website www.site2.com which also uses a class C1. How to distinguish between the two classes?

- Worse, www.site2.com could be untrusted and provide some malicious code in class C1

- A class loaded from an untrusted site should not be put in the same package as a class loaded from a trusted site.

Each class loader defines a name space.

All classes loaded by particular class loader belong to its name space.

class loader cl1	class loader cl2	
C1	C1	...
C2	C3	
	...	

Web-browsers typically create different class loaders for loading classes from different sites.

Hence classes from different sites can be put in different name spaces.

Hence the class C1 provided by www.site1.com is different from the class C1 provided by www.site2.com.

If class C1 from a particular name space needs a class C2, then the associated class loader is looked up.

The class loader associated with that namespace provides the required class C2.

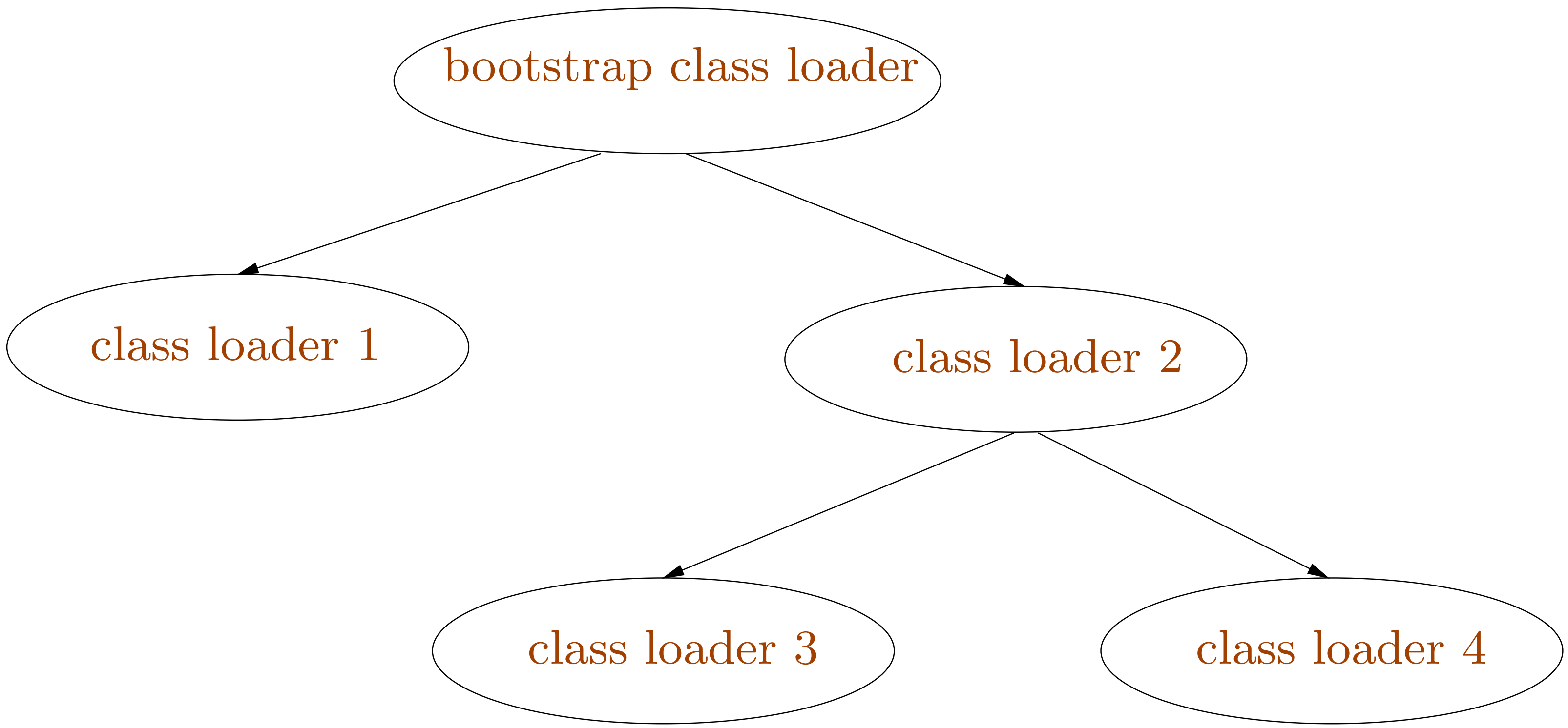
Hence the class C2 provided will be from the same namespace.

In this way, name spaces provide a way to prevent untrsuted classes from accessing trusted classes.

A class from an untrusted site cannot pretend to be a trusted class from the java API.

Hierarchy of class loaders

- The bootstrap class loader (primordial class loader, internal class loader) is responsible for loading a few initial classes when the JVM is launched.
- All new user defined class loaders have a parent class loader.



Typical class loading mechanism

1. return already existing class object, if found
2. ask the security manager for permission to access this class
3. attempt to load the class using the parent class loader
4. ask the security manager for permission to create this class
5. read the class file into an array of bytes
6. perform bytecode verification
7. create the class object
8. resolve the class

Using a class loader

Class loaders are members of (subclasses of) the abstract ClassLoader class.

Classes are loaded using the loadClass function of class loaders:

protected Class loadClass (String name, boolean resolve)

where name is the name of the class, and resolve tells us whether the class should be resolved or not.

Typically new classes of class loaders are defined by extending standard ones like SecureClassLoader or URLClassLoader.

Defining a new class of class loader

Either extend ClassLoader or one of its subclasses.

```
import java.net.*;
// a trivial extension of URLClassLoader
public class myClassLoader extends URLClassLoader {
  myClassLoader (URL url) { super (new URL[] {url}); }

  protected Class loadClass (String name, boolean resolve) {
    Class c = null;
    try { c = super.loadClass(name, resolve);
    } catch (ClassNotFoundException e) { System.out.println ("Class not found"); }
    return c;
  }
}
```

Using a class loader

```
import java.lang.reflect.*;
import java.net.*;

public class runClass {
  public static void main (String args[]) {

    // Create a class loader
    URL url = null;
    try { url = new URL ("file:/home/userxyz/classes");
    } catch (MalformedURLException e) { }
    myClassLoader cl = new myClassLoader(url);
```

```
    Class c = null; Method m = null;
    c = cl.loadClass (args[0]); // Load the class

    //Compute the argument vector and invoke the main method
    Class argtypes[] = new Class[] { String[].class };
    try { m = c.getMethod ("main", argtypes); } catch (NoSuchMethodException e) {
      System.out.println ("The main method was not found"); };
    Object arglist[] = new Object[1];
    arglist[0] = new String[args.length - 1];
    for (int i=0; i < args.length - 1; i++) ((String[])arglist[0])[i] = args[i+1];
    try { m.invoke (null, arglist); } catch (Exception e) {
        System.out.println ("Error upon invocation" + e); };
  }
}
```

Java Bytecode Verification

Static analysis of the bytecodes to ensure security properties like

- operations follow typing rules
- no illegal casts
- no conversion from integers to pointers
- no calling of directly private methods of another class
- no jumping into the middle of a method
- no confusion between data and code

The JVM

- Stack based abstract machine: operations pop arguments and push results
- A set of registers, typically used for local variables and parameters: accessed by load and store instructions
- Stack and registers are preserved across method calls
- For each method, the number of stack slots and registers is specified in the bytecode
- unconditional, conditional and multiway (switch) intra-procedural branches
- Exception handlers table of entries (pc_1, pc_2, C, h): if exception of class C is raised between locations pc_1 and pc_2, then handler is at location h.
- Most JVM instructions are typed.

Example bytecode

The source code:

```
public class test {
  public static int factorial (int n) {
    int res;
    for (res = 1; n > 0; n--) res = res * n;
    return res;
  }
}
```

and the JVM bytecode (shown by running javap on the class file)...

```
...
public static int factorial(int); 2 stack slots, 2 registers
    0:  iconst_1       // push integer constant 1
    1:  istore_1       // store it in register 1 (res)
    2:  iload_0        // push register 0 (n)
    3:  ifle 16        // if negative or zero, goto 16
    6:  iload_1        // push register 1 (res)
    7:  iload_0        // push register 0 (n)
    8:  imul           // multiply
    9:  istore_1       // store in register 1 (res)
    10: iinc 0, −1     // increment register 0 (n) by -1
    13: goto 2         // goto beginning of loop
    16: iload_1        // load register 1 (res)
    17: ireturn        // return this value
...
```

Some properties to be verified

- Type correctness: the arguments of an instruction are always of the right type.
- No stack overflow or underflow
- Code containment: the PC points within the code for the method, at the beginning of an instruction
- Register initialization before use
- Object initialization before use

Minimize runtime checks $\Longrightarrow$ efficient execution

Verification idea: type level abstract interpretation

Use types as the abstract values.

The partial ordering $\sqsubseteq$ on types is the subtype relation.

Hence for example $C \sqsubseteq D \sqsubseteq \mathsf{Object}$ if class C extends class D.

We introduce special types Null and $\top$ to abstract null pointers and uninitialized values. Also $T \sqsubseteq \top$ for every T.

An abstract stack S is a sequence of types.

The sequence $S = \mathsf{Int} \cdot \mathsf{Int} \cdot \mathsf{String}$ abstracts a stack having a string at the bottom of the stack and just two integers above it.

An abstract register assignment R maps registers to types.

$$R : \{0, \ldots, M_{reg} - 1\} \rightarrow \mathcal{T}$$

where M_{reg} is the maximum number of registers and $\mathcal{T}$ is the set of types.

An abstract state is either $\perp$ (unreachable state) or (S, R) where S is an abstract stack and R is an abstract register assignment.

Executing instructions modifies the abstract state.

$$(S, R) \xrightarrow{\mathsf{iconst}\ n} (\mathsf{Int} \cdot S, R)$$

if $|S| < M_{stack}$ where M_{stack} is the maximum size of the stack

Executing instructions modifies the abstract state.

$$(S, R) \xrightarrow{\mathsf{iconst}\ n} (\mathsf{Int} \cdot S, R)$$

if $|S| < M_{stack}$ where M_{stack} is the maximum size of the stack

$$(\mathsf{Int} \cdot \mathsf{Int} \cdot S, R) \xrightarrow{\mathsf{iadd}} (\mathsf{Int} \cdot S, R)$$

Executing instructions modifies the abstract state.

$$(S, R) \xrightarrow{\text{iconst } n} (\mathsf{Int} \cdot S, R)$$

if $|S| < M_{stack}$ where M_{stack} is the maximum size of the stack

$$(\mathsf{Int} \cdot \mathsf{Int} \cdot S, R) \xrightarrow{\text{iadd}} (\mathsf{Int} \cdot S, R)$$

$$(S, R) \xrightarrow{\text{iload } n} (\mathsf{Int} \cdot S, R)$$

if $0 \leq n < M_{reg}$ and $R(n) = \mathsf{Int}$ and $|S| < M_{stack}$

Executing instructions modifies the abstract state.

$$(S, R) \xrightarrow{\mathsf{iconst}\ n} (\mathsf{Int} \cdot S, R)$$

if $|S| < M_{stack}$ where M_{stack} is the maximum size of the stack

$$(\mathsf{Int} \cdot \mathsf{Int} \cdot S, R) \xrightarrow{\mathsf{iadd}} (\mathsf{Int} \cdot S, R)$$

$$(S, R) \xrightarrow{\mathsf{iload}\ n} (\mathsf{Int} \cdot S, R)$$

if $0 \leq n < M_{reg}$ and $R(n) = \mathsf{Int}$ and $|S| < M_{stack}$

$$(\mathsf{Int} \cdot S, R) \xrightarrow{\mathsf{istore}\ n} (S, R\{n \mapsto \mathsf{Int}\})$$

if $0 \leq n < M_{reg}$

$$(\mathsf{Int} \cdot S, R) \xrightarrow{\mathsf{ifle}\ n} (S, R)$$

if n is a valid instruction location

$$(S, R) \xrightarrow{\mathsf{goto}\ n} (S, R)$$

if n is a valid instruction location

$$(S, R) \xrightarrow{\textsf{aconst\_null}} (\textsf{Null} \cdot S, R)$$

$$\text{if } |S| < M_{stack}$$

$$(S, R) \xrightarrow{\textsf{aconst\_null}} (\textsf{Null} \cdot S, R)$$

$$\text{if } |S| < M_{stack}$$

$$(S, R) \xrightarrow{\textsf{aload } n} (R(n) \cdot S, R)$$

$$\text{if } 0 \leq n < M_{reg} \text{ and } R(n) \sqsubseteq \textsf{Object} \text{ and } |S| < M_{stack}$$

$$(S, R) \xrightarrow{\textsf{aconst\_null}} (\textsf{Null} \cdot S, R)$$

$$\text{if } |S| < M_{stack}$$

$$(S, R) \xrightarrow{\textsf{aload}\ n} (R(n) \cdot S, R)$$

$$\text{if } 0 \leq n < M_{reg} \text{ and } R(n) \sqsubseteq \textsf{Object} \text{ and } |S| < M_{stack}$$

$$(\tau \cdot S, R) \xrightarrow{\textsf{astore}\ n} (S, R\{n \mapsto \tau\})$$

$$\text{if } 0 \leq n < M_{reg} \text{ and } \tau \sqsubseteq \textsf{Object}$$

Accessing fields and methods

$$(\tau' \cdot S, R) \xrightarrow{\mathsf{getfield}\ C.f.\tau} (\tau \cdot S, R)$$

$$\text{if } \tau' \sqsubseteq C$$

Accessing fields and methods

$$(\tau' \cdot S, R) \xrightarrow{\mathsf{getfield}\ C.f.\tau} (\tau \cdot S, R)$$

$$\text{if } \tau' \sqsubseteq C$$

$$(\tau_1 \cdot \tau_2 \cdot S, R) \xrightarrow{\mathsf{putfield}\ C.f.\tau} (S, R)$$

$$\text{if } \tau_1 \sqsubseteq \tau \text{ and } \tau_2 \sqsubseteq C$$

Accessing fields and methods

$$(\tau' \cdot S, R) \xrightarrow{\textsf{getfield } C.f.\tau} (\tau \cdot S, R)$$
$$\text{if } \tau' \sqsubseteq C$$

$$(\tau_1 \cdot \tau_2 \cdot S, R) \xrightarrow{\textsf{putfield } C.f.\tau} (S, R)$$
$$\text{if } \tau_1 \sqsubseteq \tau \text{ and } \tau_2 \sqsubseteq C$$

$$(\tau'_n \cdot \ldots \cdot \tau'_1 \cdot S, R) \xrightarrow{\textsf{invokestatic } C.m.\sigma} (\tau \cdot S, R)$$
$$\text{if } \sigma = \tau(\tau_1, \ldots, \tau_n), \tau'_i \sqsubseteq \tau_i \text{ for } 1 \leq i \leq n \text{ and } |\tau \cdot S| \leq M_{stack}$$

Accessing fields and methods

$$(\tau' \cdot S, R) \xrightarrow{\textsf{getfield}\ C.f.\tau} (\tau \cdot S, R)$$
$$\text{if } \tau' \sqsubseteq C$$

$$(\tau_1 \cdot \tau_2 \cdot S, R) \xrightarrow{\textsf{putfield}\ C.f.\tau} (S, R)$$
$$\text{if } \tau_1 \sqsubseteq \tau \text{ and } \tau_2 \sqsubseteq C$$

$$(\tau'_n \cdot \ldots \cdot \tau'_1 \cdot S, R) \xrightarrow{\textsf{invokestatic}\ C.m.\sigma} (\tau \cdot S, R)$$
$$\text{if } \sigma = \tau(\tau_1, \ldots, \tau_n), \tau'_i \sqsubseteq \tau_i \text{ for } 1 \leq i \leq n \text{ and } |\tau \cdot S| \leq M_{stack}$$

$$(\tau'_n \cdot \ldots \cdot \tau'_1 \cdot \tau' \cdot S, R) \xrightarrow{\textsf{invokevirtual}\ C.m.\sigma} (\tau \cdot S, R)$$
$$\text{if } \sigma = \tau(\tau_1, \ldots, \tau_n), \tau' \sqsubseteq C, \tau'_i \sqsubseteq \tau_i \text{ for } 1 \leq i \leq n \text{ and } |\tau \cdot S| \leq M_{stack}$$

Another example

```
public class testclass {
  public testclass () { }
  public Class testfunction (String s) {
    Class c = s.getClass();
    return c;
  }
}
```

```
public java.lang.Class testfunction(java.lang.String); 1 stack slots, 3 registers
   0: aload_1
   1: invokevirtual #2; //Method java/lang/Object.getClass:()Ljava/lang/Class;
   4: astore_2
   5: aload_2
   6: areturn
```

Our analysis on this example

```
public java.lang.Class testfunction(java.lang.String); 1 stack slots, 3 registers
                                        // stack, R(0), R(1), R(2)
                                        // ϵ, (testclass, String, ⊤)
   0:    aload_1                        // String, (testclass, String, ⊤)
   1:    invokevirtual #2;              // Class, (testclass, String, ⊤)
   4:    astore_2                       // ϵ, (testclass, String, Class)
   5:    aload_2                        // Class, (testclass, String, Class)
   6:    areturn
```

In case of several paths to a node, we need to compute least upper bounds $\sqcup$.

Comparison of abstract stacks:

$$T_1 \cdot \ldots \cdot T_n \sqsubseteq U_1 \cdot \ldots \cdot U_n \quad \text{iff} \quad T_i \sqsubseteq U_i \text{ for } 1 \leq i \leq n.$$

$$T_1 \cdot \ldots \cdot T_n \sqcup U_1 \cdot \ldots \cdot U_n = T_1 \sqcup U_1 \cdot \ldots \cdot T_n \sqcup U_n$$

Comparison of abstract register assignments:

$$R_1 \sqsubseteq R_2 \quad \text{iff} \quad R_1(i) \sqsubseteq R_2(i) \text{ for } 0 \leq i < M_{reg}.$$

$$(R_1 \sqcup R_2)(n) = R_1(n) \sqcup R_2(n)$$

Comparison of abstract states

$$(S_1, R_1) \sqsubseteq (S_2, R_2) \quad \text{iff} \quad S_1 \sqsubseteq S_2 \text{ and } R_1 \sqsubseteq R_2$$

$$(S_1, R_1) \sqcup (S_2, R_2) = (S_1 \sqcup S_2, R_1 \sqcup R_2)$$

Also $\bot \sqsubseteq (R, S)$ and $\bot \sqcup (R, S) = (R, S)$.

Initial abstract state: (S_{start}, R_{start}) where $S_{start} = \epsilon$ is the empty stack and $R_{start}(0), \ldots, R_{start}(n-1)$ are the n arguments, and $R_{start}(i) = \top$ for $i \geq n$

If $\pi : pc_1 \to pc_2$ is a path (possibly with loops) from pc_1 to pc_2 with corresponding instruction sequence $I_1, \ldots, I_k$ and

$$(R_{i-1}, S_{i-1}) \xrightarrow{I_i} (S_i, R_i)$$

for $1 \leq i \leq n$ then we write $\pi : (S_0, R_0) \to (S_k, R_k)$.

For every valid location pc we define

Merge Over All Paths (MOP):

$$\mathcal{S}[pc] = \bigsqcup \{(S, R) \mid \pi : (S_{start}, R_{start}) \to (S, R)\}$$

Example

Suppose classes D and E are defined by extending class C, so that $D \sqcup E = C$.

```
                        // Int, (D, E)
10:  ifle  17           // ε, (D, E)
13:  aload_0            // D, (D, E)
14:  goto 18            // ε, (D, E)
17:  aload_1            // C, (D, E)
18:  areturn
```

(According to our notation, $C, (D, E)$ is the abstract state before the execution of the instruction at location 18.)

Another example

```
                        // ε, (Int, String)
9:   iload_0            // Int, (Int, String)
10:  ifle  17           // ε, (Int, String)
13:  iload_0            // Int, (Int, String)
14:  goto 18            // ε, (Int, String)
17:  aload_1            // ⊤, (Int, String)
18:  areturn
```

The bytecode verification fails because the return value is of unknown type.

```
public static int factorial(int); 2 stack slots, 2 registers
                              // ϵ, (Int, ⊤)
   0:   iconst_1              // Int, (Int, ⊤)
   1:   istore_1              // ϵ, (Int, Int)
   2:   iload_0               // Int, (Int, Int)
   3:   ifle 16               // ϵ, (Int, Int)
   6:   iload_1               // Int, (Int, Int)
   7:   iload_0               // Int · Int, (Int, Int)
   8:   imul                  // Int, (Int, Int)
   9:   istore_1              // ϵ, (Int, Int)
   10:  iinc 0, −1            // ϵ, (Int, Int)
   13:  goto 2                // ϵ, (Int, Int)
   16:  iload_1               // Int, (Int, Int)
   17:  ireturn
```

Other issues to be tackled in the full Java bytecode language:

- initialization of objects
- exception handling