Announcements

Java's Dynamic Loading Mechanism

JVM Dynamic Classfile Loading

- Runtime (re)installation of components
- Different from previous/similar loaders (Oberon,Com.Lisp)
  - Lazy loading (wait until the last possible minute to load)
    - Reduces memory usage/bandwidth/system response time
  - Type-safe linkage
    - Additional **link-time** checks enable type-safety guarantees in the presence of lazy loading
  - User-definable class loaders
    - Users have complete (?) control over class loading
    - Users can dictate where files are loaded from and what security measures must be taken for different sources
  - Multiple namespaces
    - Different class loaders define different **runtime** namespaces
    - Enables componentization of applications

Java Class Loader

```
package java.lang;
public abstract class ClassLoader {
   public Class loadClass(String name);
   protected final Class defineClass(String name, byte[]buf,
      int off, int len);
   protected final Class findLoadedClass(String name);
   protected final Class findSystemClass(String name);
   . . .
}
```

- The JVM class loader is the **primordial** or **system loader**
  - Java library
  - Loads initial application class and all system classes
- **Defining** loader of class C
  - The loader that actually loads C (calls defineClass)
  - C's defining loader will load ALL of the unloaded classes that C accesses

Java Class Loader

```
package java.lang;
public abstract class ClassLoader {
   public Class loadClass(String name);
   protected final Class defineClass(String name, byte[]buf,
      int off, int len);
   protected final Class findLoadedClass(String name);
   protected final Class findSystemClass(String name);
   . . .
}
```

- Users can define their own class loaders
  - All user-defined class loaders are subclasses of ClassLoader
  - Enables namespace separation
  - Class-level security & loading strategies
  - Program instrumentation and dynamic class re-loading
- Delegation - Class loader A asks class loader B to load a class on behalf of class loader A

Java Class Loaders

- **Delegation** is used to enable type safe sharing of classes
  - All system classes are loaded by the **system class loader**
  - A user-defined class loader, A, must delegate the loading of a system class, C, to the system loader, S
    - In such a case, A is an **initiating** loader of class C and S is the **defining** loader of class C
- The static type of a class corresponds to the class name
- The ***runtime or dynamic* type** of a class is the class name **and its defining class loader**

User-defined Class Loader Example

```
import java.io.*;
Class MyClassLoader extends ClassLoader {
private String directory;
public MyClassLoader(String dir) { directory = dir; }

public synchronized Class loadClass(String name)
   throws ClassNotFoundException {
   Class c = findLoadedClass(name);
   if (c!= null) return c;
   try { //will load any class if found(check name)
      c = findSystemClass(name); /*delegate to the system */
      return c;
   } catch (ClassNotFoundException e) { }
   try {
      byte[] data = getClassData(directory, name);
      return defineClass(name,data,0,data.length);
   } catch (ClassFormatError e){
      throw new ClassNotFoundException();   }
   }
}
/* for now, its not implemented */ protected
byte[] getClassData(String dir, String nm) { return null; }
```

Java Class Loaders (review)

- Dynamic class type = Namespace separation
  - Multiple apps in the same VM can use the same names
  - Each independent application/applet running in a VM has a unique class loader
    - Prohibits the interference of one program with another
    - Applets behavior is restricted (for security) - separate class loaders enable this
  - Enables components of the application to be updated *while* the program is running

Security for Java Programs

- The security of a sandbox relies on the fact that Applets
  - Use bytecode - an abstract, architecture-independent program representation
    - Enable manipulation of more secure data abstractions
    - Ex: Object references instead of memory addresses
  - Access low-level devices through a set of well-defined APIs
    - Allowing access control to be implemented/validated
  - And that the code is **type-safe (or memory safe)**
    - **Verification is performed to check each class**
    - **However, we must ensure that class loading does not violate type safety**

Type Safety & Class Loading

- To maintain type safety, the VM must consistently
  - Obtain the same class type for a given class name/loader
  - A user-defined loadClass(…) cannot be trusted to return the same type for a given name
  - Loaded class cache is used
    - Maps class names and initiating loaders to class types
  - The value returned from a user-defined loadClass is checked
    - The name of the class is checked against the name passed to loadClass (error if not the same, cached if it is the same)
    - Prior to calling loadClass, the cache is checked to prohibit repeated loadClass invocations on the same class name

Java Class Loaders

- **Delegation** is used to enable type safe sharing of classes
  - All system classes are loaded by the **system class loader**
  - A user-defined class loader, A, must delegate the loading of a system class, C, to the system loader, S
    - In such a case, A is an **initiating** loader of class C and S is the **defining** loader of class C
- Type-safety is violated without delegation of system classes
  - An application could load java.lang.String from /tmp/bogus
  - The system could load java.lang.String from /usr/src
  - The application & system might not have a uniform notion of the type of this class and its object instances

Type Safety & Class Loading

- The static type of a class corresponds to the class name
- However, class loaders introduce multiple namespaces
- The ***runtime* type** of a class is the class name **and its defining class loader**
- Hence, namespaces may be inconsistent with the namespace managed by the compiler (static type checker) and jeopardize type safety

```
class C {
   void f(X x) { . . . }
   . . .
   void g() { . . . f(new X()); . . . }
```

If the 2 occurences of X map to 2 different types, the type-safety of the method call would be compromised.

Example

```
class C {                          class S {
   void f() {                          private int secret_value;
      S x = D.g();                     private int forged_pointer;
   }                                  }
}

class D {                          /* L1 loads C, S, and D */
   S g(){ . . .}
}
```

- L1 is a user-defined class loader
- C is a class
  - L1 intiates and loads C
  - L1 therefore will initiate loading of all classes accessed by C
    EX: classes S and D
  - No delegation
    - L1 is the **intiating and defining loader** for C, S, and D
- No problems

Violating Type Safety with Delegating Loaders

```
class C {                          class S {
   void f() {                          private int secret_value;
      S x = D.g();                     private int forged_pointer;
   }                                  }
}

class D {                          /* L1 loads C
   S g(){ . . .}                       L2 loads D (L1 initiates it)
}                                      Which loader loads S?
                                    */
```

- L1 is a user-defined class loader
- C is a class
  - L1 intiates and loads C
  - L1 **delegates** the loading of D to L2
  - L2 loads D
  - L1 loads S (because S is accessed in C)
- L2 is the defining loader of D & initiates loads of all classes D accesses
  - L2 loads S (because S is accessed D)
- Now there exists S with loader L1, and S with loader L2

Violating Type Safety with Delegating Loaders

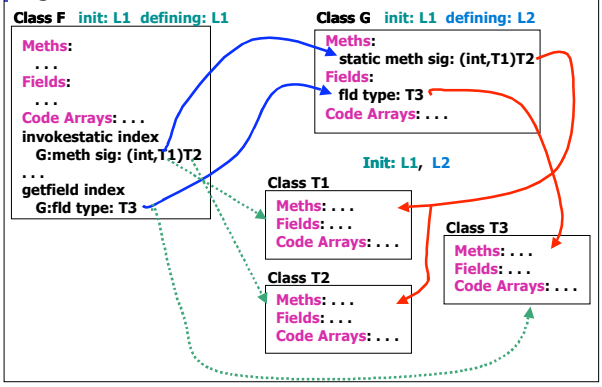
```
class C { /*defd by L1*/           class S { /* defd by L1 */
   void f() {                          /* SPOOFED CLASS */
      S x = D.g();          ──►        public int secret_value;
   }                                    public int[] forged_pointer;
}                                    }

class D { /*defd by L2*/           class S { /* defd by L2 */
   S g(){ . . .}                       private int secret_value;
}                            ──►     private int forged_pointer;
                                     /* intended class */
                                    }
```

- We are ok if the class that L1 loads is the same as L2 loads
- **However, if they are not . . .**

- In the above, method f can access secret_value and forged_pointer in x
- However, the application's actual (true) implementation is for these fields to be **private!**
- In addition, **an integer can be deferenced: x.forged_pointer[0]**
  - L2/S defines an int[] called forged_pointer which is a reference
  - f() can access x.forged_pointer as an integer - **a type-safety violation**

Type Safety Violation by JDK1 Class Loaders
Big Picture of the Problem: Indirect Class Access



Enabling Type-Safety & Dynamic Class Loading

- This was allowed by all JVMs under the 1.0.x & 1.1.x spec
  - Many researchers offered solutions & holes were found
  - It was questioned whether a solution existed (that didn't involve adding runtime checks, eliminating lazy loading or user-defined class loaders)
- However there is a solution:
  - In previous specifications, the type of a class was checked only by its name
  - Java 2 - defines the dynamic type of a class to be its class name and the defining loader of the class
  - **Dynamic type checks** that check this type guarantee type-safety in the presence of dynamic class file loading . . .

Checking Class Types to Enable Type-Safety

- Goal: Load-time checking only
  - If the same name is loaded by two different loaders, there is a check that ensures that the resulting type is the same
- Implementation . . .

Checking Class Types to Enable Type-Safety

- Distinguish defining loaders from initiating loaders
  - L1 is C's defining loader <C,L1>
  - L1 is C's initiating loader C/L1
- Solution
  - Develop a set of **loader constraints** for each class
    - Predicates (statements with a true/false value)
    - EX: S/L2 = S/L1 for class S
  - Every entry in the class cache satisfies all loader constraints
  - Each time a new class is loaded and about to be cached
    - All existing loader constraints in the cache are checked for violation by the new class (class loading fails if so)
    - Each time a new loader constraint is added, the cache is check for any classes that violate the constraint (the new constraint fails)

Constraint Rules

- Applied and checked during class file verification
  - If class has already been loaded by 2 loaders in a way that violates the constraint being added, error is raised
- Fields: <C,L1> references a field of type T in <D,L2>
  - T/L1 = T/L2
  - T is referenced in one class and defined in another (the classes could be loaded by different loaders)
- Methods: <C,L1> calls meth, sig: (T1,...,Tn)T0 in <D,L2>
  - T0 methodname(T1, T2, . . ., Tn); // In <D,L2>
  - T0/L1 = T0/L2, T1/L1 = T1/L2, …, Tn/L1 = Tn/L2
  - methodname is **invoked by** one class & defined in another
  - methodname is **overridden** by one class & defined in another

Revisiting Our Violation Example

```
class C { /*defd by L1*/
   static void f() {
      S x = D.g();
      x.F = 1;
   }
}
class D { /*defd by L2*/
   static S g(){
       return new S();
   }
```

```
class S { /* defd by L1 */
    /* SPOOFED CLASS */
    public int secret_value;
    public int[] forged_pointer;
    int F;
}
class S { /* defd by L2 */
    private int secret_value;
    private int forged_pointer;
    /* intended class */
}
```

```
Class C, method f:                          Class D, method g:
    invokestatic #3 //D:g type:()S              new #2
    astore_0                                    dup
    aload_0                                     invokespecial #3
    iconst_1                                       //S:<init>
    putfield #4 //S:F type:int                     //type: ()V
    return                                      areturn
```

Revisiting Our Violation Example

```
Class C, method f:                          Class D, method g:
    invokestatic #3 //D:g type:()S              new #2
    astore_0                                    dup
    aload_0                                     invokespecial #3
    iconst_1                                       //S:<init>
    putfield #4 //S:F type:int                     //type: ()V
    return                                      areturn
```

```
C loaded by L1 (init/def)    add: S/L1 = S/def(D)
f invoked
   invokestatic invoked
   load D with L1 (init)
     delegate to L2 (def)  update/check constraint: S/L1 = S/L2
   g invoked
      new instruction
      load S with L2 (D's defL) (init/def)  check constraints
      return (ref of type S is on tos)
   astore_0, aload_0, iconst_1
   putfield #4
   load S with L1 (C's defL) (init/def)  check constraints
```

Revisiting Our Violation Example

```
public class C {
    private static boolean test = true;
    public static void main(String args[]) {
            C myc = new C(); myc.f();
    }
    void f() {
        D.g();
       //even though no assignment is ever
       //made btwn the two, loadtime error
        S myS2 = new S();
    }
}
```

```
Class C, method f:                          Class D, method g:
    invokestatic #3 //D:g type:()S              new #2
    pop                                         dup
    new #6 <Class S>                               invokespecial #3
    dup                                            //S:<init>
    invokespecial #7 // S()                        //type: ()V
    astore_1                                    areturn
    return
```

Summary

- Example of using class loaders and violation cases
  - www.cs.ucsb.edu/~ckrintz/classes/w06/cs263/classloader_test
- Solution in JDK2
  - Use constraints to dynamically check that each newly loaded class is the same type of loaded classes of the same name (loaded by different class loaders) if they can be accessed in the same context
  - Applied when a class type may be referred to by another class and the two classes are defined by different loaders
  - Check when a class is loaded that it meets all existing constraints
- Componentization can still occur safely
  - As long as a class is loaded by a new/separate/non-delegated class loader, it is different from all classes of the same name

Other things

- findSystemClass() in ClassLoader will load any class
- Loading a java.* class in a user-defined class loader
  - Not allowed: SecurityException-prohibited package name
- Calling loadClass on a class that has already been loaded
  - Not allowed: LinkageError-duplicate class definition
  - You have to check if its cached to avoid this Error