JAVA DYNAMICS

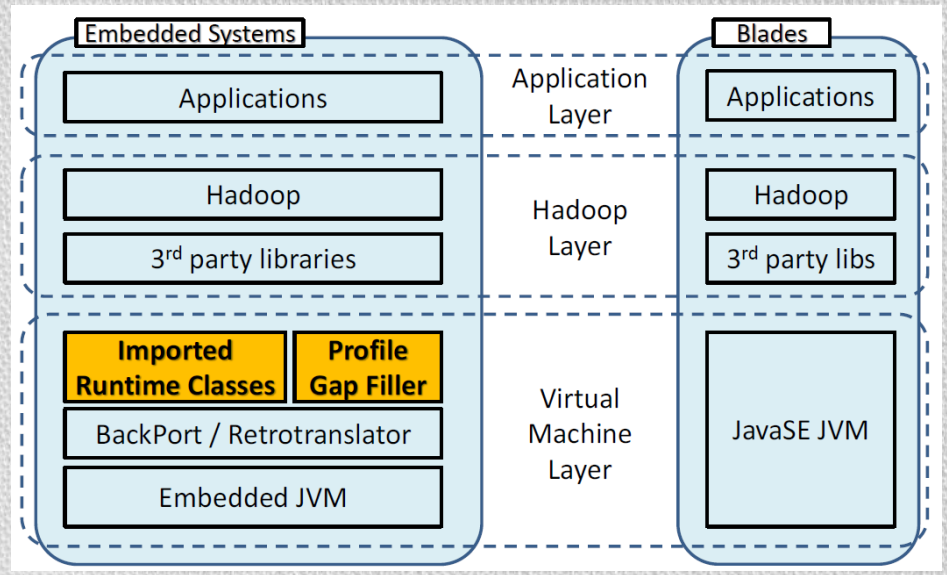
Reflection and a lot more

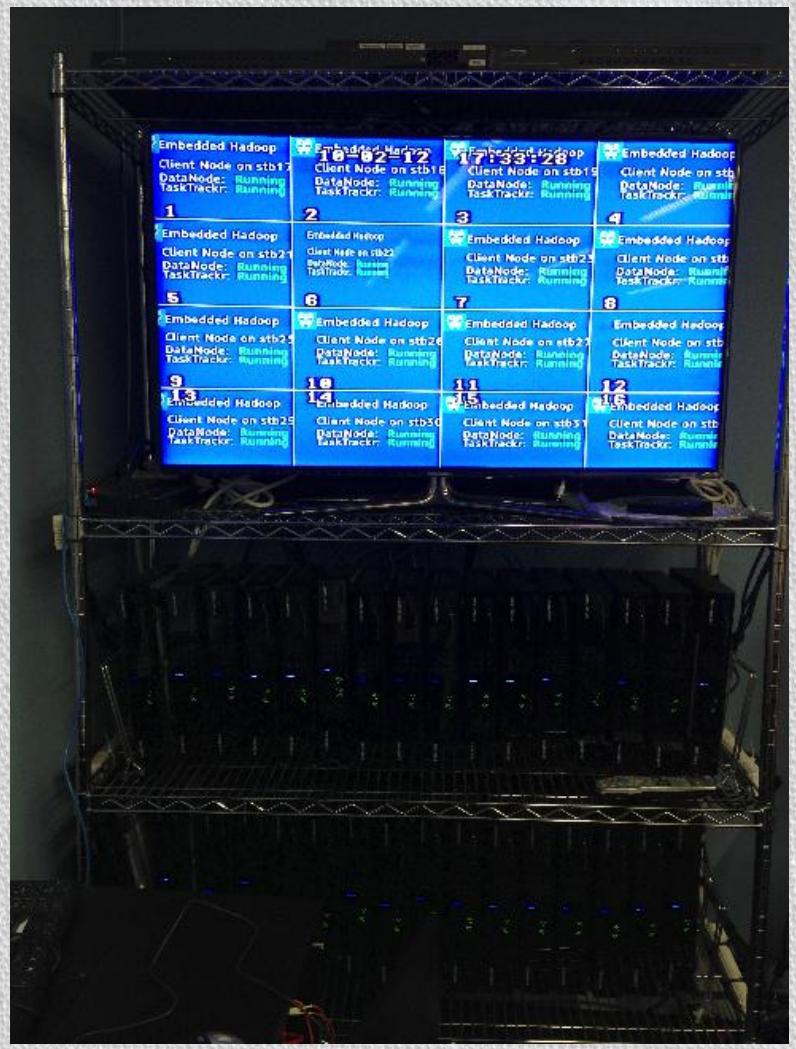
YoungHoon Jung
(jung@cs.columbia.edu)

Recent Publication

A Broadband Embedded Computing System for MapReduce Utilizing Hadoop

- Hadoop ported to STBs
- to be presented at IEEE CloudCom2012 in Dec. (http://2012.cloudcom.org)





Contents

- Terminology
- Dynamic Code Modification
- Why Java?
- Java Dynamics
  - Class loading
  - Reflection
  - Introspection
- Performance
- Javassist / BCEL
- Reflection in Research
- Thoughts for Project

Terminology

- Dynamic Code Modification / Self-Modifying Code
  - Add / Overwrite / Remove code in memory at runtime
- Dynamic Programming
  - An algorithm that pre-calculates immediate values
- Reflection
  - To examine and modify the structure and behavior of an object at runtime
- Introspection
  - To examine the type or properties of an object at runtime
- Self-Hosting
  - To use a toolchain or OS that produces new versions of the same program
- Bootstrapping
  - To write a compiler in the target programming language which it compiles

Dynamic Code Modification

- Modifying the program's code at runtime
- Not Hacking, but smells similar
- Purposes
  - Performance (Fast Paths)
  - Camouflage
  - Self-referential Machine Learning Systems
- Disadvantages
  - Hard to understand
  - Sometimes slightly slower because of cache flushing

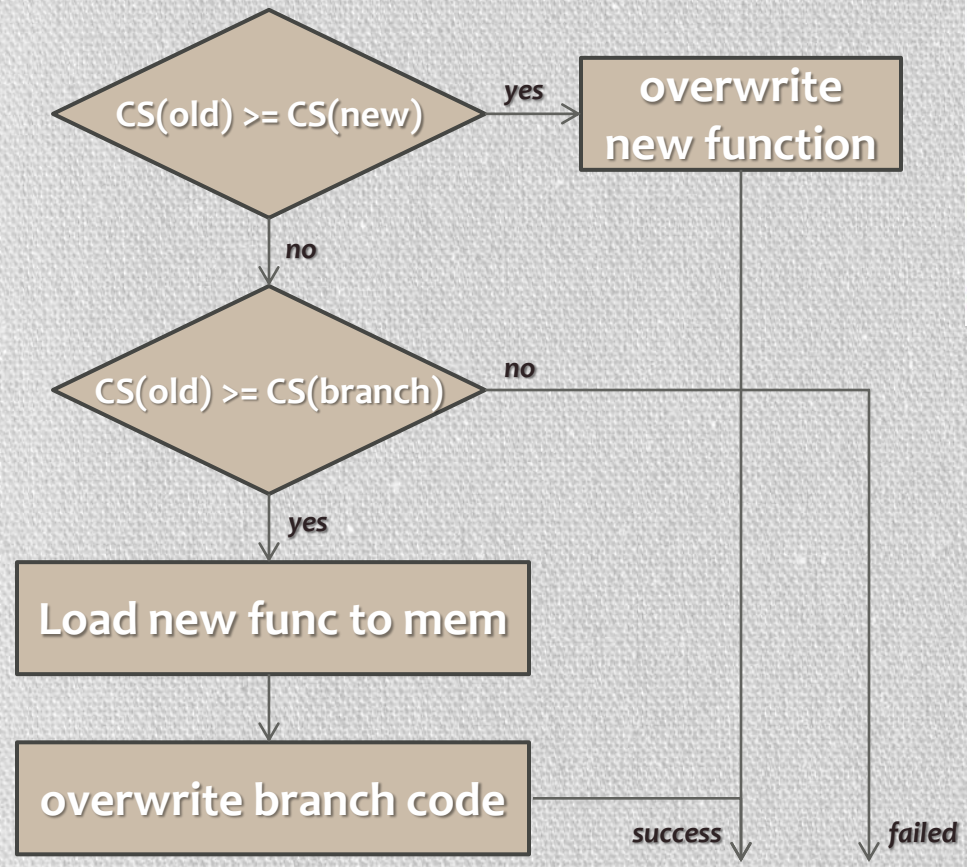
Dynamic Code Modification (cont.)

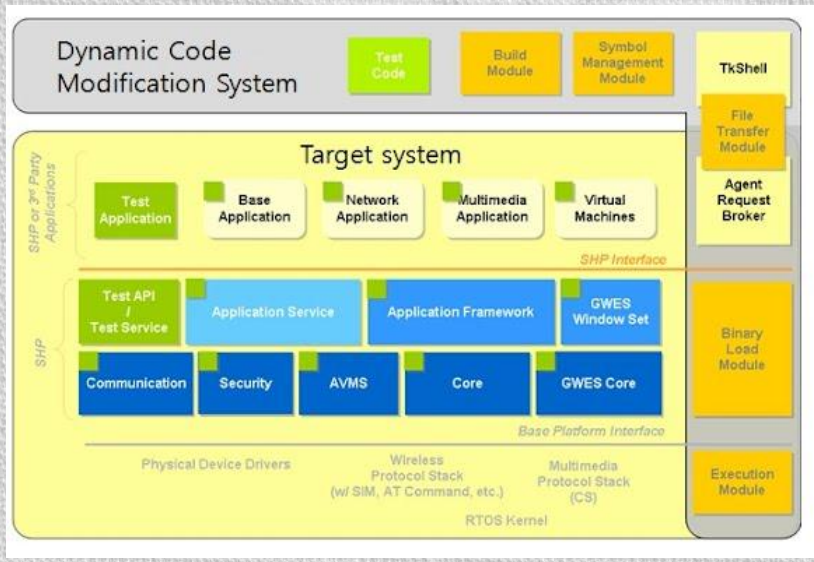
- Still extensively used by some JIT compilers
- OS support
  - efforts to distinguish from attacks or accidental errors ➔ W^X security policy
  - available in many OSs - Linux
- Massalin's Synthesis Kernel (PhD at Columbia University in 1992)
  - designed using self-modifying code
  - extremely fast
  - but written entirely in assembly

Dynamic Code Modification for Embedded Oss

Apparatus and method for developing programs and a method of updating programs

- CS(x) = code size of x
- old = old function, new = new function
- branch = branch code



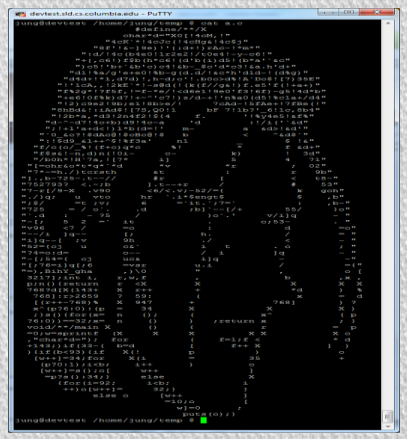


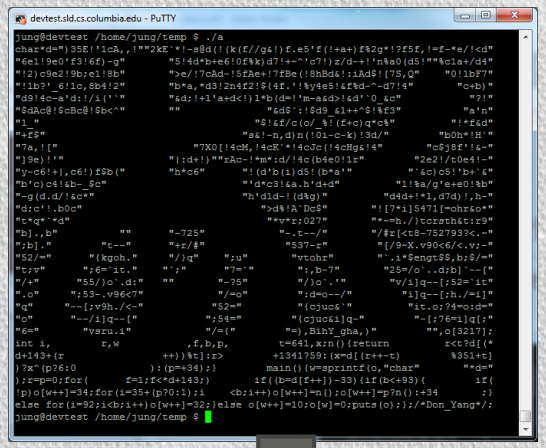
Branch Code (ARM)

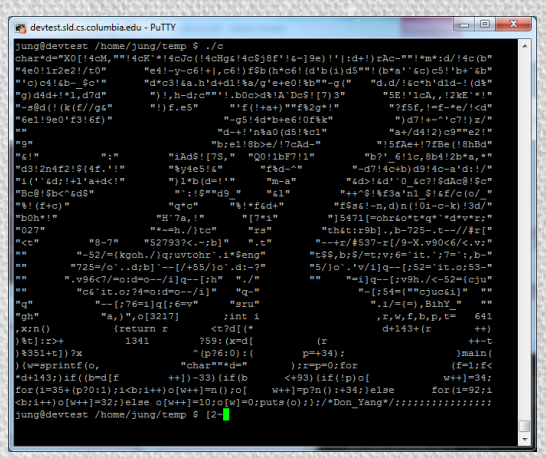
```
        PUSH {r0, r1, r2}
        MOV r1, sp
        ADD r1, r1, #8
        LDR r0, =ADDR_LBL
        STR r0, [r1]
        POP {r0, r1, pc}
ADDR_LBL: DCD [Address of New Code]
```

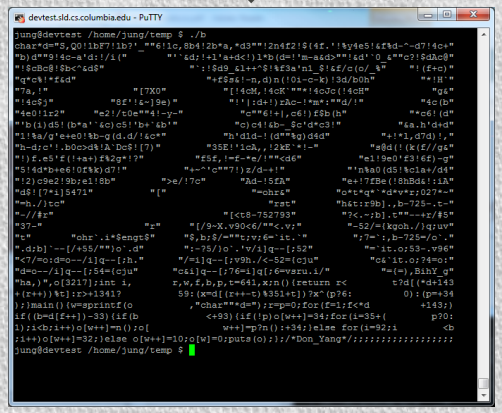
Self-Generating Code

- Saitou Hajime
  - IOCCC (The International Obfuscated C Code Contest) [link]









Why Java?

- Portability
  - handsets, smartphones, STBs, TVs, DVDs, PlayStation, …
- Still one of the most popular programming languages in use (Orace: 9M, Wikipedia: 10M)
- Wide support for Cloud services (Hadoop)
- MIPL has a Java Bytecode-generating Backend.

Class Loading

- Late Dynamic Binding
  - The JRE does not require that all classes are loaded prior to execution
    - Different from most other environments
  - Class loading occurs when the class is first referenced
  - Late Dynamic Binding is…
    - Important for polymorphism
      - Message propagation is dictated at runtime
      - Messages are directed to the correct method
    - Essential for reflection to be possible

The Order of Class Loading

```
class A {
    static {
        System.out.println("Class A loaded");
    }
}

class B extends A {
    static {
        System.out.println("Class B loaded");
    }
}

class C extends B {
    static {
        System.out.println("Class C loaded");
    }
}

class D {
    static {
        System.out.println("Class D loaded");
    }
}
```

```
public class E {
    static D d;
    static C c;

    static {
        System.out.println("Class E loaded");
    }

    public static void main(String[] args) {
        System.out.println("before create an instance of C");
        c = new C();
        System.out.println("after create an instance of C");

        System.out.println("before create an instance of D");
        d = new D();
        System.out.println("after create an instance of D");
    }
}
```

Credit by YoungHoon Jung

1. A,B,C,D,E
2. E, D, C, B, A
3. E, C, B, A, D
4. E, A, B, C, D
5. E, D, A, B, C

The Order of Class Loading

```
class A {
    static {
        System.out.println("Class A loaded");
    }
}

class B extends A {
    static {
        System.out.println("Class
    }
}

class C extends B {
    static {
        System.out.println("Class
    }
}

class D {
    static {
        System.out.println("Class
    }
}
```

```
public class E {
    static D d;
    static C c;

    static {
ass E loaded");


ng[] args) {
efore create an instance of C");

ter create an instance of C");

efore create an instance of D");

ter create an instance of D");
```

Class E loaded
before create an instance of C
Class A loaded
Class B loaded
Class C loaded
after create an instance of C
before create an instance of D
Class D loaded
after create an instance of D

Credit by YoungHoon Jung

1. A,B,C,D,E
2. E, D, C, B, A
3. E, C, B, A, D
4. E, A, B, C, D
5. E, D, A, B, C

Java ClassLoaders

- All ClassLoaders are a subclass of the class “ClassLoader”
- Every ClassLoader has a parent class loader (or often null)
  - Create a tree
  - Delegation Class Loading Model
- Many JVM has three default ClassLoaders
  - Bootstrap class loader
    - Loads the core Java libs in <JAVA_HOME>/lib
    - Part of the core JVM
    - Written in native code
  - Extensions class loader
    - Loads the code in the Extension directories <JAVA_HOME>/lib/ext or specified directories by “java.ext.dirs”
    - Cryptographic, Secure Socket, Management, …
  - System class loader
    - Loads classes found on CLASSPATH

User-Defined ClassLoaders

- Written in Java by users
- Support various way to get bytecode (e.g. from HTTP)
- Can decode specific bytecode (e.g. encrypted)
- Allows multiple namespaces (e.g. CORBA / RMI)
- Can modify the loaded bytecode (e.g. AOP)

- Implemented by overriding two methods:
  - protected synchronized Class<?> loadClass(String name, boolean resolve)
    - Determines the class has already been loaded, otherwise call findClass()
  - Protected Class<?> findClass(String name)
    - Actually tries to find the contents of the designated class

Typical loadClass()

```
protected synchronized Class<?> loadClass (String name, boolean resolve) throws
                        ClassNotFoundException {

        // First check if the class is already loaded
        Class c = findLoadedClass(name);
        if (c == null) {
                try {
                        if (parent != null) {
                                c = getParent().loadClass(name, false);
                        } else {
                                c = Class.forName(name, resolve, null);
                        }
                } catch (ClassNotFoundException e) {
                        // If still not found, then invoke findClass to find the class.
                        c = findClass(name);
                }
        }
        if (resolve) {
                resolveClass(c);
        }
        return c;
}
```

Examples of Class Loader I – Plugin

- Class Loading from specific directories in configuration

```
public Class<?> findClass(String name) throws ClassNotFoundException {
    // load pluginDirs from xml configuration
    …

    try {
      while (String dir : pluginDirs) {
        String classPath =  dir +  className.replace('.',File.separatorChar)+".class";
        classByte = loadClassData(classPath);
        result = defineClass(className,classByte,0,classByte.length,null);
        classes.put(className,result);
        return result;
      }
    } catch (Exception e) {
         return null;
    }
}
```

Examples of Class Loader II – Jar

- Class Loading from jar files

```
public Class<?> findClass(String name) throws ClassNotFoundException {
    …

    for (String jarFilename : jarsList) {
        try {
            JarFile jarFile = new JarFile(jarFilename);
            ZipEntry entry = jarFile.getEntry(className);
            if (entry == null) continue;
            InputStream classStream = jarFile.getInputStream(entry);
            byte[] theClass = … // fully read from classStream
            loadedClass = defineClass(name, theClass, 0, theClass.length);
            classList.put(name, loadedClass);
        } catch (IOException e) { … }
    }
    …
    return loadedClass;
}
```

Examples of Class Loader III - Network

- Class Loading through HTTP

```
public class HttpClassLoader extends ClassLoader {
        String host;
        int port;
        …

        public Class findClass(String name) {
                byte[] b = downloadClassData(name);
                return defineClass(name, b, 0, b.length);
        }

        private byte[] downloadClassData(String name) {
                // load the class data from the connection  . . .
                //
        }
}
```

Another Class Loading Example

```
public class A {
}
```

1. InstantiationException
2. ClassNotFoundException
3. NullPointerException
4. InvocationTargetException
5. None of above

```
import java.net.URL;
import java.net.URLClassLoader;
import java.net.MalformedURLException;

public class TwoClassLoaders {
   public static void main(String[] args) {
      ClassLoader cl = null;
      try {
         cl = new URLClassLoader(new URL[] {new URL("file:///.../")}, null);
      } catch (MalformedURLException mue) { }

      try {
         Class clsA = cl.loadClass("A");
         A a = (A) clsA.newInstance();
         System.out.println("Finished");
      } catch (InstantiationException ie) {
         System.out.println("InstantiationException:" + ie);
      } catch (ClassNotFoundException cnfe) {
         System.out.println("ClassNotFoundException:" + cnfe);
      } catch (Exception e) {
         System.out.println("Exception:" + e);
      }
   }
}
```

Another Class Loading Example

```
public class A {
}
```

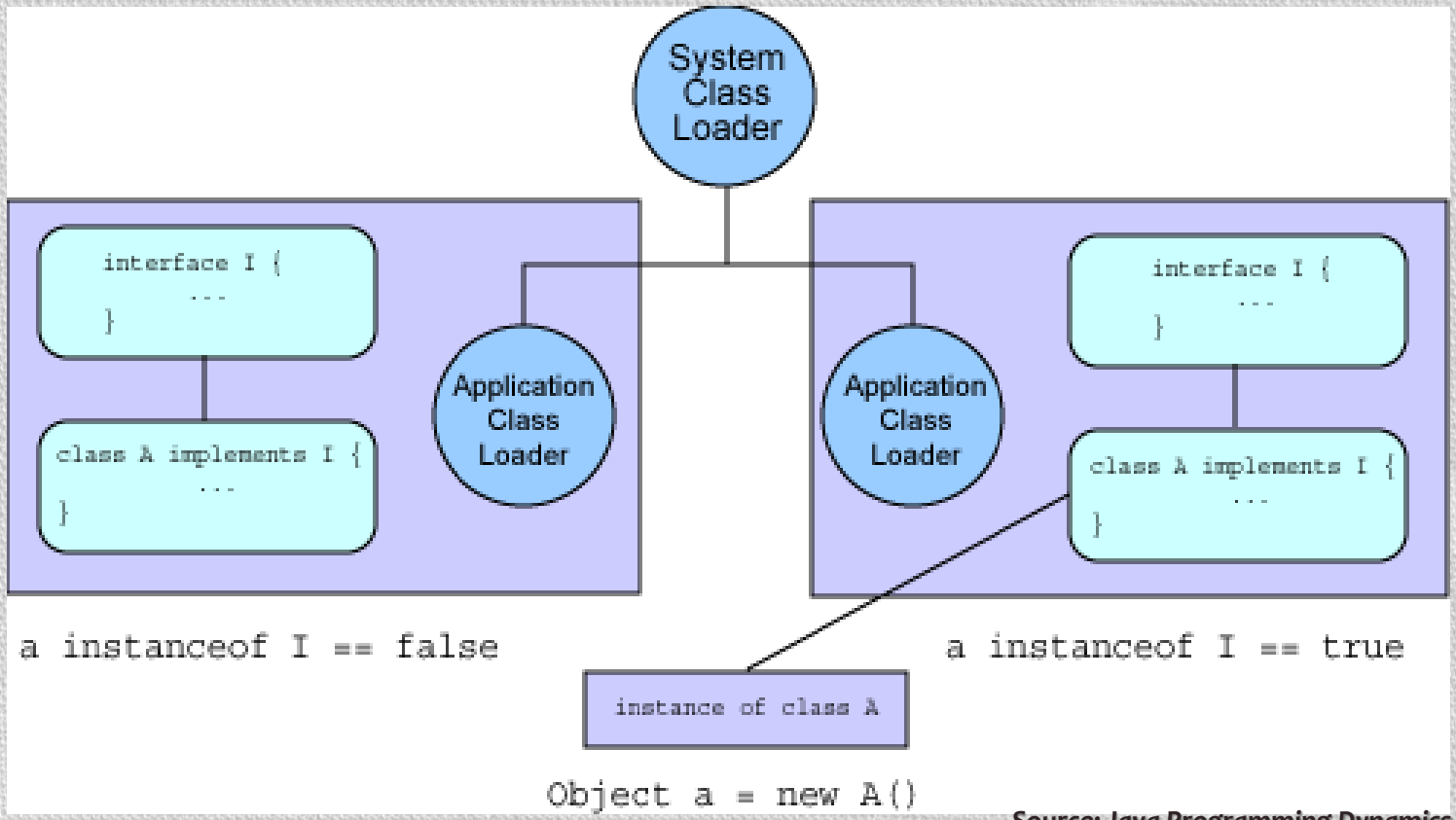
1. InstantiationExc
2. ClassNotFoundExc
3. NullPointerExc
4. InvocationTargetException
5. None of above

Exception:
java.lang.ClassCastException:
A cannot be cast to A

```
import java.net.URL;
import java.net.URLClassLoader;
import java.net.MalformedURLException;

public class TwoClassLoaders {
  public static void main(String[] args) {
    ClassLoader cl = null;
    try {
                                    URL[] {new URL("file:///.../")}, null);
                                    n mue) { }
                                    ;
                                    e) {
                                    onException:" + ie);
    } catch (ClassNotFoundException cnfe) {
      System.out.println("ClassNotFoundException:" + cnfe);
    } catch (Exception e) {
      System.out.println("Exception:" + e);
    }
  }
}
```

Class Identity Crisis

- The same class loaded by two different Class Loaders is identified as two different classes.



Source: Java Programming Dynamics

Reflection

- The ability to observe and/or manipulate the inner workings of the environment programmatically
- The reflection API represents, or reflects, the classes, interfaces, and objects in the current Java™ virtual machine
- Reflection can be used for observing and/or modifying program execution (not code!) at runtime.
- example:

```
// Without reflection
new Foo().hello();

// With reflection
Class<?> clazz = Class.forName("Foo");
clazz.getMethod("hello").invoke(clazz.newInstance());
```

Reflection (cont.)

- Reflection is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language.

- With that caveat in mind, reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible.

(from Oracle's official Java Tutorial)

History of Reflection

- Invented by Brian Smith in June 1976 at the Xerox Palo Alto Research Center.
  - Designed for a way to learn a language, MANTIQ.
  - Worked on the initial versions of the language for five years.

- 1982
  - Brian Cantwell Smith writes a doctoral dissertation at MIT introducing the notion of computational reflection. 3-LISP is the first official programming language to use reflection.
- 1983
  - Smalltalk v1.0 has 75% of the standard Reflection command language.
- Oct 1996
  - Visual J++ and C# has reflections. Python v1.4
- Feb 1997
  - Java Reflections (JDK v1.1).

Source: Java Reflection

The Reflection Classes

- java.lang.reflect
  - The reflection package
  - Introduced in JDK 1.1 release
- java.lang.reflect.AccessibleObject
  - The superclass for ***`Field`***, ***`Method`***, and ***`Constructor`*** classes
  - Suppresses the default Java language access control checks
  - Introduced in JDK 1.2 release
- java.lang.reflect.Array
  - Provides static methods to dynamically create and access Java arrays
- java.lang.reflect.Constructor
  - Provides information about, and access to, a single constructor for a class

The Reflection Classes (cont.)

- java.lang.reflect.Field
  - Provides information about, and dynamic access to, a single field of a class or an interface
  - The reflected field may be a class (static) field or an instance field
- java.lang.reflect.Member
  - Interface that reflects identifying information about a single member (a field or a method) or a constructor
- java.lang.reflect.Method
  - Provides information about, and access to, a single method on a class or interface
- java.lang.reflect.Modifier
  - Provides static methods and constants to decode class and member access modifiers

The Reflection Classes (cont.)

- JDK 1.3 release additions
  - java.lang.reflect.Proxy
    - Provides static methods for creating dynamic proxy classes and instances
    - The superclass of all dynamic proxy classes created by those methods
  - java.lang.reflect.InvocationHandler
    - Interface
    - Interface implemented by the invocation handler of a proxy instance

What Reflection Does?

- Literally everything that you can do if you know the object's class
  - Load a class
  - Determine if it is a class or interface
  - Determine its superclass and implemented interfaces
  - Instantiate a new instance of a class
  - Determine class and instance methods
  - Invoke class and instance methods
  - Determine and possibly manipulate fields
  - Determine the modifiers for fields, methods, classes, and interfaces
  - Etc.

Reflection Howto

- Load a class

```
Class c = Class.forName ("Classname")
```

- Determine if a class or interface

```
c.isInterface ()
```

- Determine lineage
  - Superclass

```
Class c1 = c.getSuperclass ()
```

  - Superinterface

```
Class[] c2 = c.getInterfaces ()
```

Reflection Howto

- Determine implemented interfaces

```
Class[] c2 = c.getInterfaces ()
```

- Determine constructors

```
Constructor[] c0 = c.getDeclaredConstructors ()
```

- Instantiate an instance
  - Default constructor

```
Object o1 = c.newInstance ()
```

  - Non-default constructor

```
Constructor c1 = c.getConstructor (class[]{…})
Object i = c1.newInstance (Object[] {…})
```

Reflection Howto

- Determine methods

```
Methods[] m1 = c.getDeclaredMethods ()
```

- Find a specific method

```
Method m = c.getMethod ("methodName",
                        new Class[] {…})
```

- Invoke a method

```
m.invoke (c, new Object[] {…})
```

Reflection Howto

- Determine modifiers

```
Modifiers[] mo = c.getModifiers ()
```

- Determine fields

```
Class[] f = c.getDeclaredFields ()
```

- Find a specific field

```
Field f = c.getField("name")
```

- Modify a specific field
  - Get the value of a specific field

```
f.get (o)
```

  - Set the value of a specific field

```
f.set (o, value)
```

Three Myths of Reflection

- "Reflection is only useful for JavaBeans™ technology-based components"
- "Reflection is too complex for use in general purpose applications"
- "Reflection always reduces performance of applications"

Source: Using Java Technology Reflection to Improve Design

“Reflection Is Only Useful for JavaBeans™ Technology-based Components”

- False
- Reflection is a common technique used in other pure object oriented languages like Smalltalk and Eiffel
- Benefits
  - Reflection helps keep software robust
  - Can help applications become more
    - Flexible
    - Extensible
    - Pluggable

“Reflection Is Too Complex for Use in General Applications”

- False
- For most purposes, use of reflection requires mastery of only several method invocations
- The skills required are easily mastered
- Reflection can significantly…
  - Reduce the footprint of an application
  - Improve reusability

“Reflection Always Reduces the Performance of Applications”

- False
- Reflection can actually increase the performance of code
- Benefits
  - Can reduce and remove expensive conditional code
  - Can simplify source code and design
  - Can greatly expand the capabilities of the application

Reflection - Drawbacks

- Performance Overhead
  - reflective operations have slower performance than their non-reflective counterparts
- Security Restrictions
  - Reflection requires a runtime permission which may not be present when running under a security manager.
- Exposure of Internals
  - can result in unexpected side-effects, which may render code dysfunctional and may destroy portability. Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

Why Use Reflection

- Reflection solves problems within object-oriented design:
  - Flexibility
  - Extensibility
  - Pluggability
- Reflection solves problems caused by…
  - The static nature of the class hierarchy
  - The complexities of strong typing

Use Reflection With Design Patterns

- Design patterns can benefit from reflection
- Reflection can …
  - Further decouple objects
  - Simplify and reduce maintenance

Design Patterns and Reflection

- Many of the object-oriented design patterns can benefit from reflection
- Reflection extends the decoupling of objects that design patterns offer
- Can significantly simplify design patterns

- Factory
- Factory Method
- State
- Command
- Observer
- Others

Factory Without Reflection

```
public static Shape getFactoryShape (String s)
{
  Shape temp = null;
  if (s.equals (“Circle”))
    temp = new Circle ();
  else
    if (s.equals (“Square”))
      temp = new Square ();
    else
      if (s.equals (“Triangle”)
        temp = new Triangle ();
      else
        // …
        // continues for each kind of shape
  return temp;
}
```

Factory With Reflection

```
public static Shape getFactoryShape (String s)
{
    Shape temp = null;
    try
    {
        temp = (Shape) Class.forName (s).newInstance ();
    }
    catch (Exception e)
    {
    }
    return temp;
}
```

Design Pattern Implications

- Product classes can be added, changed, or deleted without affecting the factory
  - Faster development (one factory fits all)
  - Reduced maintenance
  - Less code to develop, test, and debug

Reflective Programming Languages

- APL
- Befunge
- BlitzMax
- ColdFusion MX
- Curl
- Delphi
- JavaScript
- Eiffel
- Forth
- Go
- Io
- Java
- Lisp
- Logo
- Logtalk
- Lua
- Mathematica
- Maude system
- .NET Common Language Runtime
- Oberon
- Objective-C
- Perl
- PHP
- Pico
- PL/SQL
- POP-11
- Poplog
- Prolog
- Python
- R
- REBOL
- Ruby
- Scheme
- Smalltalk
- SuperCollider
- Snobol
- Tcl

Source: Wikipedia

Summary of Java Reflection

- Reflection is what makes the language dynamic
- An advanced and powerful feature but easy to use
- Java Reflection APIs provide access to every part of a class
  - Field, Method, Constructors, …
  - Load, Check, Create, Invoke, Manipulate, …
- Disadvantages:
  - Performance, Security, and Exposure
- Advantages
  - Flexibility, Extensibility, and Pluggability

Introspection

- Focused on Type Checking
  - instanceof

```
if(obj instanceof Person)
{
   Person p = (Person) obj;
   p.walk();
}
```

  - getName()

```
System.out.println(obj.getClass().getName());
```

Comparison to Reflections in Other Languages

- versus C#
  - Reflection in C# is done at assembly the level
  - while in Java is done at the class level

(Source: A Comparison of Microsoft's C# Programming Language to Sun Microsystems' Java Programming Language)

- versus Python
  - Python supports Reflection (or Introspection in Python) without using APIs
  - Thus, some argue it's easier in Python

(Source: Why is Python more fun than Java?)

BCEL: The Byte Code Engineering Library (Apache Commons BCEL™)

- intended to give users a convenient way to analyze, create, and manipulate (binary) Java class files (those ending with .class).

- Objects can be
  - read from an existing file
  - transformed by a program (e.g. a class loader at run-time)
  - written to a file

- One can create classes from scratch at run-time

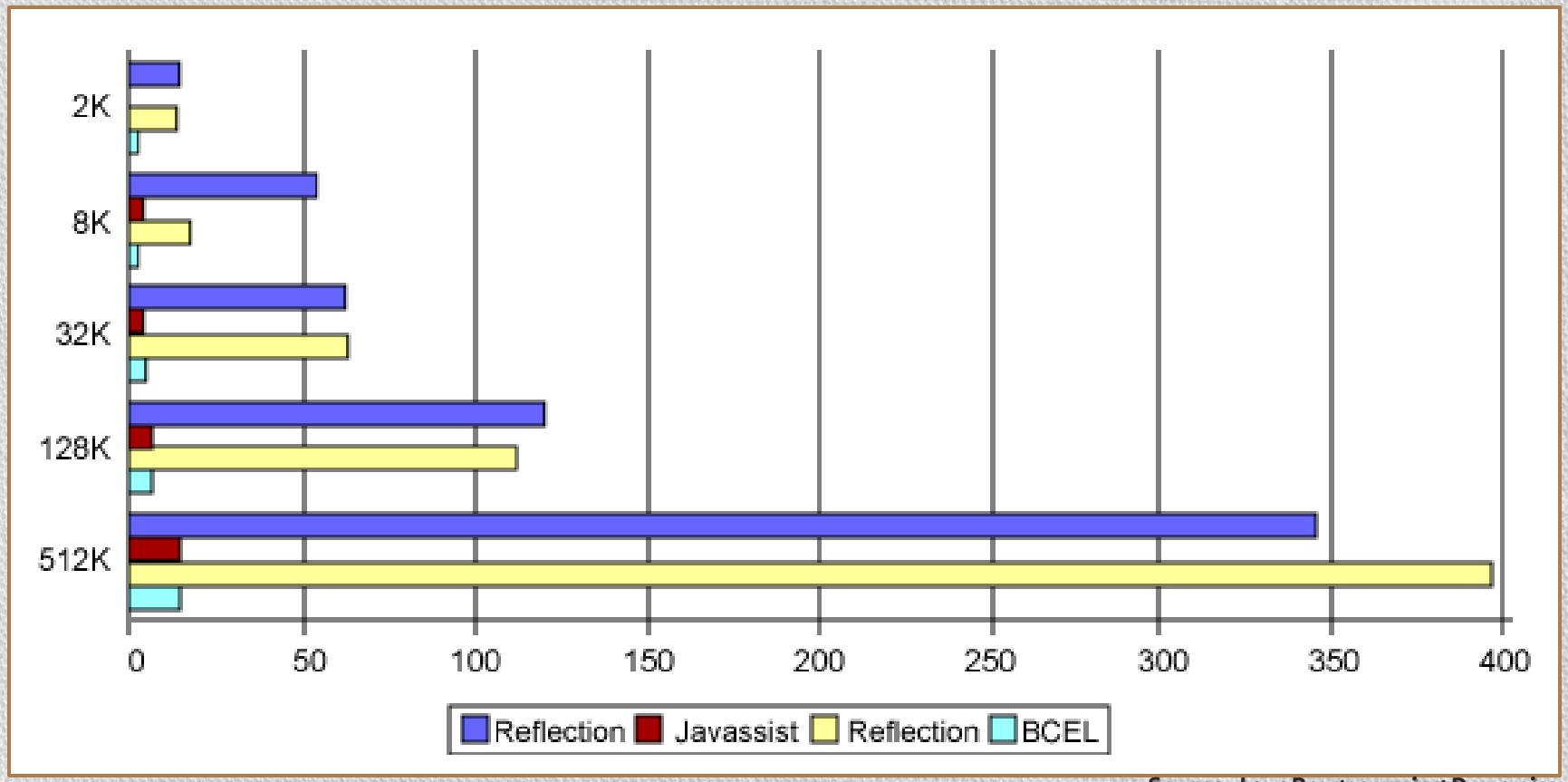
- being used successfully in several projects such as compilers, optimizers, obfuscators, code generators and analysis tools
  - MIPL generates Java byte code using BCEL

BCEL Example

```
import org.apache.bcel.Repository;
import org.apache.bcel.classfile.JavaClass;
import org.apache.bcel.generic.ClassGen;
import java.io.IOException;

public class SomeBcelClass {

 public static void main(String[] args) {

  ClassGen myClassGen;
  try {
   JavaClass myClass =
    Repository.lookupClass("MyClass");
   myClassGen = new ClassGen(myClass);
  }
  catch(ClassNotFoundException ex) {
   ex.printStackTrace();
   return;
  }

  // this is where you mess
  // around with the classes

  try {
   myClassGen.getJavaClass()
    .dump("MyClass.class");
  }
  catch(IOException ex) {
   ex.printStackTrace();
  }
 }
}
```

Performance Comparison

- Code Generation is 5-24 times faster!



Source: Java Programming Dynamics

Leveraging Java Reflection

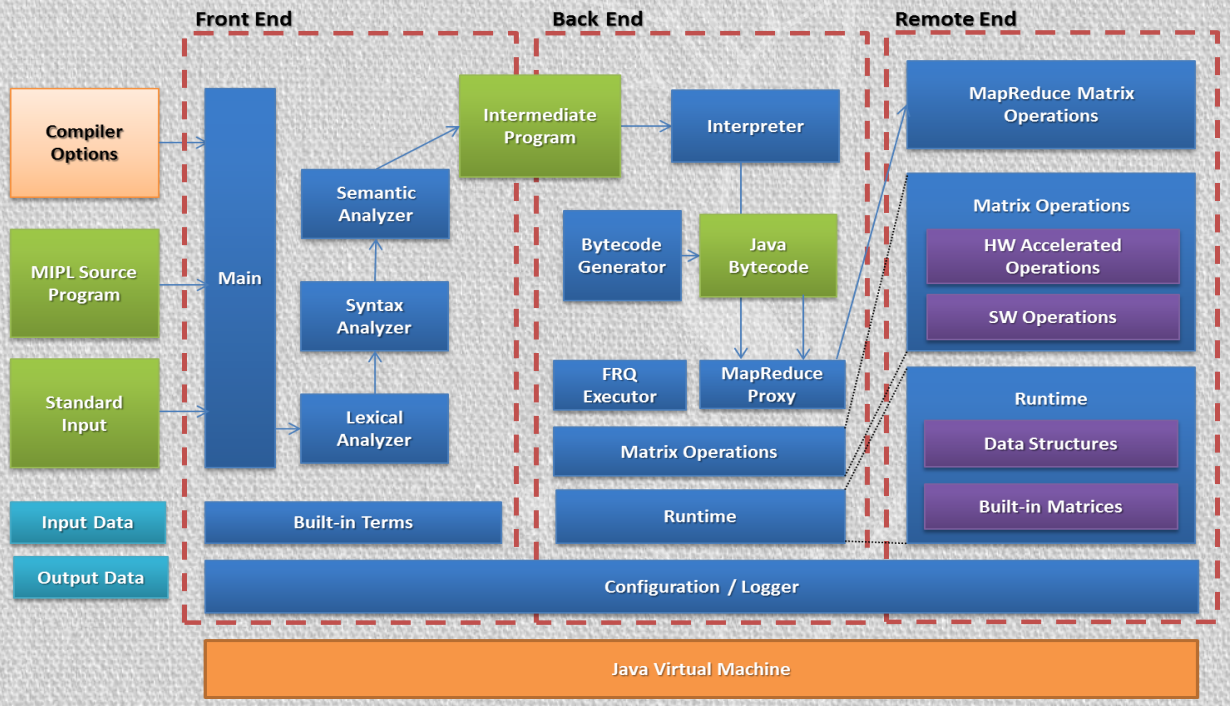
- T. J. Brown, I. Spence, and P. Kilpatrick, "**Mixin Programming in Java with Reflection and Dynamic Invocation**," *Proc. of Inaugural Conf. on Principles and Practice of Programming*, 2002
  - Unlike C++, Java has no template. Instead, Reflection is used for Mixin.
- Viera K. Proulx and Weston Jossey, **"Unit Test Support for Java via Reflection and Annotations," Proc. of 8th Int. Conf. on Principles and Practice of Programming in Java, 2009**
- Mathias Braux and Jacques Noye, **"Towards partially evaluating reflection in Java," Proc. of Partial Evaluation and Semantics-based Program Manipulation, 1999**
- Remi forax, Etienne Duris, and Gilles Roussel, "**Reflection-based implementation of Java extensions: the double-dispatch use-case**," Proc. of ACM symposium on Applied Computing, 2005

Reflection for Performance

- Tharaka Devadithya, Kenneth Chiu, and Wei Lu, "**C++ Reflection for High Performance Problem Solving Environments,**" in Proc. of the 2007 Spring Simulation MultiConference, vol 2, pp 435-440, 2007

Thoughts for Project

- MIPL (Mining Integrated Programming Language) – [link]
  - Compiler written in Java (over 12,000 loc)
  - Pluggable Backend Architecture
    - Java+Hadoop



Thoughts for Project

- Designing front end for dynamic elements in the language
- Applying various Matrix Computation Optimizations
- Connecting Front-end and Middle-end through Dynamic Code Modification

- Using Flexibility for Performance

Reference

- Dennis Sosnoski, Java Programming Dynamics, Developer Works, IBM, 2003
- Michael T. Portwood, Using Java Technology Reflection to Improve Design, Exuberance LLC, 2008
- Ken Cooney, Java Reflection
- Dare Obasanjo, A Comparison of Microsoft's C# Programming Language to Sun Microsystems' Java Programming Language
- Brian Clapper, Why is Python more fun than Java?, 2008

- Wikipedia: http://www.wikipedia.org
- BCEL official site: http://commons.apache.org/bcel
- Java Tutorials: http://docs.oracle.com/javase/tutorial