Java Reflection
Exercises, Correction, and FAQs

Yann-Gaël Guéhéneuc
Pierre Cointe
Marc Ségura-Devillechaise

École des Mines de Nantes
4, rue Alfred Kastler – BP 20722
44307 Nantes Cedex 3
France
{cointe, guehene, msegura}@emn.fr

December 15, 2001
Last revision: January 28, 2002

Contents

1 Introduction

This document summarizes the exercices given to the EMOOSE students for the practical sessions of the course on Reflection. These practical sessions took place on the 18^{th}, 19^{th}, and 20^{th} of December 2001.

During the practical sessions, the students met the following topics:

- `ExtendedObject`, `ClassDescription`, and `MetaObject`: How to add reflective capabilities to the Java programming language?
- `Proxy`: What is this new feature of the Java programming language and how to use it?
- `ClassLoader`: What are `ClassLoaders` and how to use them to modify the application being run *on-the-fly*?
- Block: How is it possible to implement blocks *à la* Smaltalk in Java?
- HotSwap: What is this new feature of the Java Virtual Machine (JVM) and how to use it?

The sections of this document present the different questions asked to the students and a possible solution[1]. The last section contains answers to the frequently asked questions.

[1] Do not hesitate to contact the authors for comments or questions on the solutions.

2 ExtendedObject

See appendix A page 26 for the source code.

The `ExtendedObject` class is an extension of the Java class `Object`. The goal of the `ExtendedObject` class is to provide the same capabilities as the Smalltalk class `Object`. The `ExtendedObject` class provides methods to access and to modify directly class and instance variables. The `ExtendedObject` class also provides helper methods to reify message sends and methods to clone instances.

Example 1 (ExtendedObject):
Here is an overview of the results expected from the methods of the `ExtendedObject` class. We assume the existence of a `Point` class, subclass of class `ExtendedObject`.

```
p = new Point(1,2) = 1@2@false

p.instVarAt("x") = 1
p.instVarAt("y") = 2

p.instVarAt("ZERO")  = null
p.classVarAt("ZERO") = 0@0@false
p.classVarAt("x")    = null

p.storeString() = new Point(2, 1)
p.copy()        = 1@2@false

p.instVarAtPut("y", new Integer(200)) = 1@200@false
p.instVarAtPutInt("x", 100)           = 100@200@false

Point.class.getMethod("getClass", null)
public final native java.lang.Class java.lang.Object.getClass()
Point.class.getMethod("hashCode", null)
public native int java.lang.Object.hashCode()

p.receive("toString")              = 100@200@false
p.receive("storeString")           = new Point(200, 100)
p.receive("add", p)                = 200@400@false
p.receive("ad" + "d", p)           = 200@400@false
p.receive("add", new Integer(100)) = 400
p.receive("getClass")              = class Point
p.receive("hashCode")              = 26507614
p.getMessageCount()                = 7
p.receive("zero")                  = 0@0@false
p.getMessageCount()                = 8
Message message = new Message("setX", new Object[] { new Integer(6) })
p.receive(message)                 = 6@200@false

p.receive("setX", new Integer(4))                 = 4@200@false
p.receive("setY", new Object[] { new Integer(22) } = 4@22@false
p.copy()                                          = 4@22@false
p.duplicate()                                     = 4@22@false
```

3 ClassDescription

See appendix B page 34 for the source code.

We associate an instance of the `ClassDescription` class with any instance of a Java class. The instance of `ClassDescription` provides methods to compute high-level information about its associated instance: Number of public, protected, and private methods; Number of declared methods; Number of super-classes and super-interfaces (depth in the hierarchy); Simple metrics... The instance of `ClassDescription` also provide pretty-printing methods.

Example 2 (ClassDescription):
The `ClassDescription` class provides helper methods and pretty-printing methods.

```
new ClassDescription(Point.class).getShortName()
Point

new ClassDescription(Point.class).getName()
java.awt.Point

new ClassDescription("java.applet.Applet").printHierarchy()
class java.applet.Applet
    extends java.awt.Panel
        implements javax.accessibility.Accessible
        extends java.awt.Container
            extends java.awt.Component
                implements java.awt.image.ImageObserver
                implements java.awt.MenuContainer
                implements java.io.Serializable
                extends java.lang.Object

new ClassDescription("java.awt.Point").printHierarchy()
class java.awt.Point
    implements java.io.Serializable
    extends java.awt.geom.Point2D
        implements java.lang.Cloneable
        extends java.lang.Object
```

Example 3 (ClassDescription):
The `ClassDescription` class also provides a method to compute simple metrics, using other methods defined in class `ClassDescription`.

```
new ClassDescription(Point.class).metrics()
Analysis of Point
    Number of super-classes: 2
    Number of instance variables: 9
    Number of class variables: 1
    Number of constructors: 2
```

Example 4 (ClassDescription):
The `ClassDescription` class offers a `javap`-like method. `javap` is a tool that displays the content of any Java class file in a user-friendly way. The `javap`-like method returns the following (partial) result:

```
new ClassDescription("Point.class").javap()

javap-like output of class Point
class Point extends ExtendedObject {
    // Declared fields.
    private int Point.y;
    private int Point.x;
    private static boolean Point.TRACE;
    private static Point Point.ZERO;

    // Declared methods.
    public String Point.toString();
    public Point Point.add(Point);
    public Integer Point.add(Integer);
    public void Point.setY(Integer);
    public void Point.setY(int);
    public int Point.getY();
    public void Point.setX(Integer);
    public void Point.setX(int);
    public int Point.getX();
    public static Point Point.zero();

    // Declared constructors.
    public Point(int,int);
    public Point();

    // Public fields only.

    // All public methods, including inherited methods.
    // public static void ExtendedObject.main(String[]);
    // public static Point Point.zero();
    // public final native Class Object.getClass();
    // public native int Object.hashCode();
    ...
    // public String ExtendedObject.classVarAt(int);
    // public Object ExtendedObject.instVarAt(String);
    // public void ExtendedObject.instVarAtPut(String,Object);
    ...
    // public Object ExtendedObject.copy();
    // public ExtendedObject ExtendedObject.duplicate();
    // public String ExtendedObject.storeString();
    // public int Point.getX();
    // public void Point.setX(int);
    ...
    // public Integer Point.add(Integer);
    // public Point Point.add(Point);

    // All public constructors.
    // public Point(int,int);
    // public Point();
}
```

4 MetaObject

See appendix C page 41 for the source code.

The `MetaObject` class implements a simple system of meta-object. We associate one or more instances of `MetaObject` with an instance of a Java class. Then, we use the methods understood by the `MetaObject` instance to talk with its associated instance. The meta-object intercepts the message sends and displays additional information about them. The main problem with this solution is the `this`-problem: We are not talking with the instance of a Java class anymore, we are talking with an instance of class `MetaObject`, with all the expected problems of typing.

Example 5 (MetaObject):

```
MetaObject metaObject           = new MetaObject(new Point(5, 10))
Message zeroMessage             = metaObject.reify("getX")
metaObject.receive(zeroMessage) =
    Before invoking method: public int emn...examples.Point.getX()
    Result of the invocation: 5
    After invoking method: public int emn...examples.Point.getX()
5
```

5 Proxy

See appendix D page 43 for the source code.

The new feature of the Java programming language version 1.3, called `Proxy`, allows developers to create wrappers around instances of their Java class *on-the-fly*. A wrapper implements one or more given interfaces and references one instance of class `InvocationHandler`. The wrapper forwards any message to its associated instance of class `InvocationHandler`. The instance of class `InvocationHandler` is in charge of performing the method invocation, if desired, and any other pre-/post-treatments.

The `Proxy` wrapper is a kind of meta-object. Compared with the `MetaObject` implementation proposed in section 4, the `Proxy` mechanism resolves the `this`-problem.

Example 6 (Proxy):
We create an instance of class `Point`, which implements the `IPoint` interface. Then, we create a `Proxy` on the `IPoint` interface. The `Proxy` references an instance of a sub-class of class `InvocationHandler`. The instance of the sub-class of class `InvocationHandler` references the firstly created instance of class `Point` and the `Proxy` forwards any message to the instance of class `InvocationHandler`, which prints pre- and post-messages and forwards in turn the message to the firstly created instance of class `Point`.

```
IPoint p1 = new Point(10, 15)
IPoint p2 =
    (IPoint) Proxy.newProxyInstance(
        IPoint.class.getClassLoader(),
        new Class[] { IPoint.class },
        new InvocationHandler(p1))
p1 ?                            p2 ?
10@15                           Before
                                Result shall be: 10@15
                                After
                                10@15
p1 == p2 ?
false

p1.getClass() ?
class emn.course.reflection.proxy.examples.Point
p2.getClass() ?
class $Proxy0

p1.equals(p2) ?                 p2.equals(p1) ?
Before                          Before
Result shall be: 10             Result shall be: true
After                           After
Before                          true
Result shall be: 15
After
true
```

The implementation of the `Proxy` mechanism depends on the new version of the compiler.

When the compiler encounters the creation of a new `Proxy` class, it adds a new class variable to the class that creates a new `Proxy`. The new class variable possesses the package visibility, and has for name the pattern `class$`x, where $x \in \mathbb{N}$. The new class variable is of type `Class` and references the interface implemented by the `Proxy`. The compiler adds such a class variable for each `Proxy`, regardless of possible redundancy, if two `Proxies` implement the same interface. In the case of redundancy, the redundant class variable does reference the associated implemented interface but the declaring class (bug?).

In addition to the class variable, with each `Proxy`, the compiler creates a new class with the pattern name `Proxy$`x, where $x \in \mathbb{N}$. The new class implements the interface associated with the `Proxy`. The compiler creates a unique class with each same interface implemented by one or more `Proxies`. The new class extends the `java.lang.reflect.Proxy` class. The new class forwards any message to the instance of class `InvocationHandler` associated with the `Proxy`.

The user defines as many sub-classes of class `InvocationHandler` as required, and associates their instances with the `Proxies`.

The following code excerpt mixes the Java code needed to create an instance of a `Proxy` on the `IPoint` interface and the byte-codes obtained after compilation. We can clearly see the use of the `class$`x class variable by comparing lines 21–27 with lines 39–78.

```
package emn.course.reflection.proxy.examples;

import emn.course.reflection.proxy.InvocationHandler;
import java.io.PrintStream;
import java.lang.reflect.Proxy;

// Referenced classes of package emn.course.reflection.proxy.examples:
//            Point, IPoint

public class Main {                                                    10

    // static java.lang.Class class$0;

    public Main() {
        //    0    0:aload_0
        //    1    1:invokespecial   #12  <Method void Object()>
        //    2    4:return
    }

    public static void main(String args[]) {                          20
        IPoint p1 = new Point(10, 10);
        //    0    0:new             #20  <Class Point>
        //    1    3:dup
        //    2    4:bipush          10
        //    3    6:bipush          10
        //    4    8:invokespecial   #23  <Method void Point(int, int)>
```

```
//    5   11:astore_1

IPoint p2 = (IPoint) Proxy.newProxyInstance(
    emn.course.reflection.proxy.examples.IPoint.class.getClassLoader(),                30
    new Class[] { emn.course.reflection.proxy.examples.IPoint.class},
    new InvocationHandler(p1));
//    6   12:getstatic       #25  <Field Class class$0>
//    7   15:dup
//    8   16:ifnonnull       44
//    9   19:pop
//   10   20:ldc1            #27  <String "emn.course.reflection.proxy.examples.IPoint">
//   11   22:invokestatic    #33  <Method Class Class.forName(String)>
//   12   25:dup
//   13   26:putstatic       #25  <Field Class class$0>                                 40
//   14   29:goto            44
//   15   32:new             #35  <Class NoClassDefFoundError>
//   16   35:dup_x1
//   17   36:swap
//   18   37:invokevirtual   #41  <Method String Throwable.getMessage()>
//   19   40:invokespecial   #44  <Method void NoClassDefFoundError(String)>
//   20   43:athrow
//   21   44:invokevirtual   #48  <Method ClassLoader Class.getClassLoader()>
//   22   47:iconst_1
//   23   48:anewarray       Class[]                                                    50
//   24   51:dup
//   25   52:iconst_0
//   26   53:getstatic       #25  <Field Class class$0>
//   27   56:dup
//   28   57:ifnonnull       85
//   29   60:pop
//   30   61:ldc1            #27  <String "emn.course.reflection.proxy.examples.IPoint">
//   31   63:invokestatic    #33  <Method Class Class.forName(String)>
//   32   66:dup
//   33   67:putstatic       #25  <Field Class class$0>                                 60
//   34   70:goto            85
//   35   73:new             #35  <Class NoClassDefFoundError>
//   36   76:dup_x1
//   37   77:swap
//   38   78:invokevirtual   #41  <Method String Throwable.getMessage()>
//   39   81:invokespecial   #44  <Method void NoClassDefFoundError(String)>
//   40   84:athrow
//   41   85:aastore
//   42   86:new             #50  <Class InvocationHandler>
//   43   89:dup                                                                        70
//   44   90:aload_1
//   45   91:invokespecial   #53  <Method void InvocationHandler(Object)>
//   46   94:invokestatic    #59  <Method Object Proxy.newProxyInstance(
                                     ClassLoader,
                                     Class[],
                                     java.lang.reflect.InvocationHandler)>
//   47   97:checkcast       #61  <Class IPoint>
//   48  100:astore_2

...                                                                                    80
```

6 ClassLoader

See appendix E page 47 for the source code.

The `ClassLoader` mechanism allows developers to specify the behavior of the JVM when the JVM loads a class file. The JVM associates an instance of the `ClassLoader` class with any class it loads and defines; except for the classes in the Java class libraries, which have no associated instance of `ClassLoader`. At startup, the JVM creates a default instance of the `ClassLoader` class, and it uses it to load the required class files and to define the associated classes. It is possible for the developer to define sub-classes of the `ClassLoader` class to modify the standard loading and defining behaviors.

By specializing the `ClassLoader` class, it is possible to load class files from different sources than the disk (for example from the Internet, from a data-base, from pre-defined byte-codes...) and to modify the byte-codes before defining the associated classes.

Somewhat like in C++, `ClassLoaders` define name-spaces.

The standard use of the `ClassLoader` mechanism requires to define a sub-class of the `ClassLoader` class and a `Launcher` class. The `Launcher` class creates an instance of the sub-class of class `ClassLoader` and uses it to load and to define all the classes required by the program to run. Once all the classes are defined, the `Launcher` class executes the `main(String[])` method of the main class.

To modify the behavior of the program to run, the sub-class of the `ClassLoader` class only needs to give to the JVM modified byte-codes.

Thus, using the `ClassLoader` mechanism, we can easily implement a reflective system at load-time: We only need to defined sub-classes of the `ClassLoader` class and associate them with a `Launcher` class and with a text file describing which `ClassLoader` must load and define which class file.

For more information on the `ClassLoader` mechanism (i.e., the difference between the loading class loader and the defining class loader), please refer to the article [1];

Example 7 (ClassLoaders):
We assume the existence of a class `emn.course.reflection.classloader.examples.Point`. First, we load and define the `Point` class with the default instance of class `ClassLoader`; We display the corresponding instance of class `Class` and the associated instance of class `ClassLoader`. Second, we display the instance of the class `Class` corresponding to the `java.awt.Point` class (from the standard Java class libraries) and the associated instance of class `ClassLoader`. Third, we use our own `ClassLoader` to load and to define the `PointMain` class, which uses our `Point` class, and we call its `main()` method. Fourth, we use another `ClassLoader` to load and to define again the `PointMain` class. This `ClassLoader` defines a `PointMain` class with tracing capabilities directly from stored byte-codes. We call the `main()` method of this "new" `PointMain` class.

```
Point.class = class emn.course.reflection.classloader.examples.Point
    (sun.misc.Launcher$AppClassLoader@55f210a1)
PointClass  = class java.awt.Point
    (null)
Point.class == PointClass? false
PointMain.main(new String[0])
    Point.class = class emn.course.reflection.classloader.examples.Point
        (sun.misc.Launcher$AppClassLoader@55f210a1)
1@1
classLoader.loadClass("emn.course.reflection.classloader.examples.PointMain")
PointMainClass.getMethod("main", new Class[] { String[].class }).invoke(...)
Looking for C:\...examples/PointMain.class
    Point.class = Looking for C:\...examples/Point.class
class emn.course.reflection.classloader.examples.Point
        (emn.course.reflection.classloader.PointClassLoader@55f490a1)
1@1
classLoader.loadClass("emn.course.reflection.classloader.examples.PointMain")
PointMainClass.getMethod("main", new Class[] { String[].class }).invoke(...)
Looking for C:\...examples/PointMain.class
    Point.class = class emn.course.reflection.classloader.examples.Point
        (emn.course.reflection.classloader.ModifyingPointClassLoader@55f710a1)
    getX() == 1
    getY() == 1
1@1
```

7 Block

See appendix F page 52 for the source code.

In Smalltalk, blocks (*a.k.a.* block closures) are like λ-expressions in Lisp and are used to implement lazy evaluation. A block is a sequence of statements, separated by periods, and delimited by two brackets. A block might have parameters. A block specifies an algorithm but does not execute it. The developer requests the execution of the block by explicitly sending the message `value` (or an analogous message).

```
[:index | index := index + 1. sum := sum + index]
```

In the Java programming language, blocks does not exist. The Java programming language does not offer block closures. However, the Java programming language offer inner classes. Inner classes are somewhat like block closure, although there is much debate on this subject. Using inner classes, Java dynamic class loading capabilities, and Sun's Java compiler in Java (`javac`), it possible to simulate blocks *à la* Smalltalk in Java.

Example 8 (Blocks):
Here are some example of blocks. These examples show the use of different messages: `Object value()`; `Object value(Object)`; `void applyAVectorFromTo(Vector, int, int)`...

```
    for (int i = 1; i < 5; System.out.println(i++));
    1
    2
    3
    4
    void

    for (int i = 1; i < 5; System.out.println(i++)); return "Hello";
    1
    2
    3
    4
    Hello

    int k | for (int i = k; i < 5; System.out.println(i++)); return "Hello";
    3
    4
    Hello

    boolean b | return new Boolean(b);
    true

    Object o | System.out.print(o);
    Hellovoid

    method applyAVectorFromTo()
        Object o | System.out.print(o);
        new Vector() with 'H', 'e', 'l', 'l', and 'o'
    Hello

    String a, String b | System.out.println(a.equals(b));
        with: new String("Hi"), new String("Bye")
    false
    void

    String a, String b | a = b;
        with: new String("Hi"), new String("Bye")
    void
    Hi
    Bye

    Object[] o | o[0] = o[1];
        with: new Object[] { "Hi", "Bye" }
    Hi
    Bye
    void
    Bye
    Bye
```

8 HotSwap

See appendix G page 58 for the source code.

In the latest release of Sun's JDK, version 1.4 beta 2, Sun improved the Java Debug Interface. The `com.sun.jdi.VirtualMachine` interface now provides a method `redefineClasses(Map)`. The `redefineClasses(Map)` method replaces all the classes given in parameter with the definitions supplied. This method, together with the new methods `com.sun.jdi.Method.isObsolete()` and `com.sun.jdi.ThreadReference.popFrames(com.sun.jdi.StackFrame)` implement the new HotSwap mechanism.

This method does not cause any initialization: Redefining a class does not cause the JVM to run its initializers. The values of preexisting static variables remain as they were prior to the method call. The JVM assigned their default values to completely uninitialized (new) static variables.

If a redefined class has instances, then all those instances possess the fields defined by the redefined class at the completion of the method call. Preexisting fields retain their previous values. Any new fields have their default values; The JVM does not call instance initializers or constructors. If any redefined method has active stack frames, those active frames continue to run the byte-codes of the previous methods. The JVM uses the redefined methods on new invocations.

Threads need not be suspended.

The following two figures show a simple application of the HotSwap mechanism. The tool *Euclid's GCD algorithm redefiner* launches a simple application displaying the result of a Greatest Common Denominator (GCD) algorithm. The tool provides a graphic user interface to call the `redefineClasses(Map)` method to replace the standard GCD algorithm by an algorithm returning the opposite value.

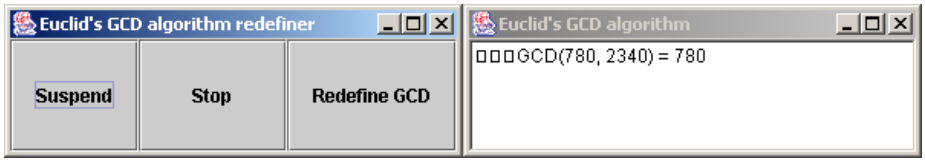

Figure 1: *A Greatest Common Denominator (GCD) implementation and a tool to hot swap the standard GCD algorithm by a GCD algorithm that returns the opposite value.*

The HotSwap mechanism is still in its infancy. The authors have been unable to pop frames out of the stack without the raise of exceptions. The exceptions prevented us to replace the GCD algorithm in a clean way.

However, the HotSwap mechanism is most promising and allows frameworks to implement sophisticated classes and instances management.

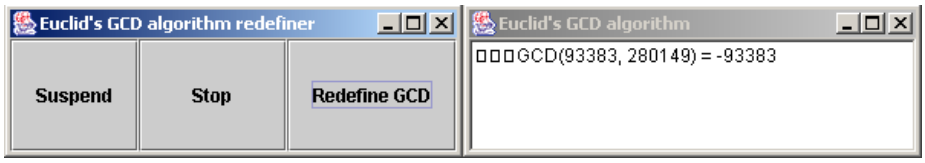

Figure 2: *After hot swaping the GCD algorithm: The GCD implementation keeps on running, it now displays the opposite value.*

9 Frequently Asked Questions

9.1 How do I use the Java Reflection API?

The Java Reflection API is defined in the package `java.lang.reflect`. The Java Reflection API allows introspection (the capability of a system to observe itself) and limited intercession (the capability of a system to modify its own state). For security reasons, developers must explicitly ask for the rights to get or to set the value of a field, to invoke a method... using the methods provided by the `AccessibleObject` class (super-class of all the reification classes: `Field`, `Method`...). The methods provided by the class `AccessibleObject` ask permission to the `SecurityManager` class through its `checkPermission(Permission)` method. The `SecurityManager` may deny the right to suppress security checks by raising a `SecurityException`.

> **Example 9 (Java Reflection API):**
> A typical example of introspection/intercession is:
>
> ```
> final Class clazz = Class.forName("MyClass");
> final Field[] fields = clazz.getDeclaredFields();
> fields[0].setAccessible(true);
> // The SecurityManager may deny the right to change the accessibility.
> System.out.println(fields[0].get(null));
> // If and only if fields[0] is a static field.
> ```

9.2 What is Java Security and how does it affect the Java Reflection API?

Java Security is a PhD thesis subject *per se*. Java security describes all the mechanisms in charge of preventing malicious code to run. For Java Security and Reflection, please see the previous sub-section (section 9.1).

9.3 Can I use the ExtendedObject class to create a new class from scratch?

No. You cannot use directly `ExtendedObject` to create a new class from scratch; i.e., by defining a new class, by adding to it new fields, new methods...

Using the Java programming language, you must use `ClassLoader` to define new classes, in the sense of the Java programming language; i.e., to make the classes available to the JVM at run-time. Only `ClassLoaders` offer the capability to define a new class.

Using a `ClassLoader`, the only way to define a new class is to use the methods `defineClass(...)` or `loadClass(...)` (the latter uses the former). These methods assume the existence of valid byte-codes, either produced by a compiler or written by hand.

Thus, the only way to define a new class from scratch is:

- By writing byte-codes by hand from scratch.
- By writing Java source code (possibly another language) from scratch and by compiling it to obtain byte-codes.

9.4 What is a modifier?

A modifier is a keyword that modifies the semantics of a declaration, such as a method declaration, a class declaration... For instance, the `final` modifier put in front of a class declaration indicates that the class cannot have sub-classes; the same modifier in front of a variable declaration (either class, instance, or local) indicates that the reference held by the variable cannot change.

9.5 How does `Modifier.isStatic(f.getModifiers())` work?

The `Modifier` class reflects all the possible values for the modifiers, as described in [2]. The JVM associates a unique integer value with each modifier, such that each integer value corresponds to a unique binary number: 2^x, where $x \in \mathbb{N}$. For instance, the JVM associates:

- $2^0 = 1$ with the `public` modifer.
- $2^1 = 2$ with the `private` modifer.
- ...
- $2^4 = 16$ with the `final` modifer.
- ...

The use of binary numbers helps in constructing, in extracting, and in checking the appearance of a given modifier. The use of binary numbers also helps in reducing the complexity and the foot-print of programs: A unique integer holds several values. For example, let assume $m = 17$, an integer, that represents the set of modifiers for a method `m`. Using the logical *and* operation, we can check:

- If the method `m` is `public`: $m \ \& \ (2^0) \stackrel{?}{\neq} 0$, yes, the method is `public`.

- If the method `m` is `private`: $m \;\&\; (2^1) \stackrel{?}{\neq} 0$, no, the method is not `private`.
- ...
- If the method `m` is `final`: $m \;\&\; (2^4) \stackrel{?}{\neq} 0$, yes, the method is `final`.
- ...

9.6 How to create a new instance of a class at runtime?

You must use the `Class.newInstance()` method. The `Class.newInstance()` method creates a new instance of the class represented by the `Class` instance. This method instantiates the class as if by a `new` expression with an empty argument list; This method requires the existence of a constructor with an empty argument list or it throws an exception.

You must use the `Constructor.newInstance()` method to create instance of a class using a specific constructor of this class:

```
final Constructor pointClass1Constructor =
    pointClass1.getConstructor(
        new Class[] { int.class, int.class });
System.out.println(
    pointClass1Method.invoke(
        pointClass1Constructor.newInstance(
            new Object[] { new Integer(1), new Integer(2)}),
        new Object[0]));
```

9.7 Should I catch or propagate any exception that may arise?

It all depends on the thrown exception and on the purpose of your program.

If you write a prototype tool to demonstrate quickly your ideas, it might be a okay to propagate some exceptions to the JVM. Indeed, propagating some exceptions to the JVM keeps your code short and simple.

If you write a program for an exam, for a demonstration, for sharing, for commercial purposes, it is a *very* bad idea to propagate *a single* exception to the JVM. Indeed, you are entitled to meet certain quality characteristics, one being sure that your program does not crash unexpectedly in front of the user!

9.8 Why bother to use one `System.print(...)` followed by a `System.println(...)` when a `+` would do?

The Java programming language provides special support for the string concatenation operator `+`, and for conversion of other objects into `String` objects.

In Sun's JDK, string concatenation is implemented through the `StringBuffer` class and the `append(...)` method. Each time you use the `+` operator to concatenate a string and another object (or another primitive type), the compiler

calls the `String valueOf(Object)` method, and creates an instance of class `StringBuffer` to hold the result of the concatenation. This involves the creation of several temporary objects.

In opposition, several `System.print(...)` method calls in a row suppress the need for the `+` concatenation operator and thus suppress the creation of temporary instances.

Example 10 (+ concatenation operator):
Consider the following source code:

```
public static void main(String[] args) {
    String a = "A";
    String b = "B";
    System.out.print(a + b);
}
```

The associated byte-codes are:

```
Method void main(java.lang.String[])
       ...
     6 getstatic #25 <Field java.io.PrintStream out>
     9 new #27 <Class java.lang.StringBuffer>
    12 dup
    13 aload_1
    14 invokestatic #33 <Method java.lang.String valueOf(java.lang.Object)>
    17 invokespecial #36 <Method java.lang.StringBuffer(java.lang.String)>
    20 aload_2
    21 invokevirtual #40 <Method java.lang.StringBuffer append(java.lang.String)>
    24 invokevirtual #44 <Method java.lang.String toString()>
    27 invokevirtual #49 <Method void println(java.lang.String)>
    30 return
```

Now, consider the following source code, which as the same effects as the one above:

```
public static void main(String[] args) {
    String a = "A";
    String b = "B";
    System.out.print(a);
    System.out.println(b);
}
```

The associated byte-codes are:

```
Method void main(java.lang.String[])
       ...
     6 getstatic #25 <Field java.io.PrintStream out>
     9 aload_1
    10 invokevirtual #31 <Method void print(java.lang.String)>
    13 getstatic #25 <Field java.io.PrintStream out>
    16 aload_2
    17 invokevirtual #34 <Method void println(java.lang.String)>
    20 return
```

9.9 What is the `this` problem?

The `this`-problem occurs when using wrappers to implement meta-objects. The wrapper is different from the wrapped object. This causes two problems:

- The wrapper must *look like* the wrapped object. We must be able to use the wrapper in the stead of the wrapped object. We typically solve this problem of identity using an interface in Java.
- When the wrapped object references itself (i.e., when the wrapped object calls one of its own methods, or accesses one of its own fields), the call explicitly or implicitly references the `this`. In theory, the access to `this` must go to the wrapper, which in turn calls the appropriate method of the wrapped object. In practice, the method is directly invoked on the wrapped object. This problem is difficult to resolve. For instance, the `Proxy` mechanism does not solve the `this`-problem, see example 6 page 8.

9.10 Are constructors inherited from the super-class?

No. Constructors are not inherited from the super-class.

Example 11 (Constructor inheritance):
Let consider the following class. It has a default constructor and it defines a constructor with a `String` as parameter.

```
public class SuperA {
    public SuperA() {
        ...
    }
    public SuperA(String s) {
        ...
    }
}
```

Now, let consider the following class `A`, sub-class of class `SuperA`. The compilation of this class leads to a compilation error: The constructor `A(String)` is undefined.

```
public class A extends SuperA {
    public A() {
        ...
    }
    public static void main(String[] args) {
        A a = new A();      // Okay.
        A a = new A("");    // Compilation error:
                            // The constructor A(String) is undefined.
    }
}
```

However, static initializers ("class constructors") are inherited from the super-class to the sub-class. A static initializer initialize the values of the static fields of the class. Static fields are inherited from the super-class to the sub-class and thus the static initializers must be inherited.

9.11 What is the difference between `Class.getFields()` and `Class.getDeclaredFields()`?

The method `Class.getDeclaredFields()` returns an array of instances of class `Field`, reflecting all the fields declared by the class or the interface represented by the instance of class `Class`. This includes public, protected, default (package) access, and private fields; excluding inherited fields.

The method `Class.getDeclaredFields()` returns an array containing instances of class `Field`, reflecting all the public fields of the class or the interface represented by the instance of class `Class`; including inherited fields.

The elements in the returned arrays are not sorted and are not in any particular order. The two methods return arrays of length 0 if the class or the interface declares no fields, or if the instance of class `Class` represents a primitive type or an array class.

9.12 What is a MOP?

The purpose of a Meta-Object Protocol (MOP) is to provide users with a powerful mechanism for extending and for customizing the basic behavior of a programming language. This implies the complete reification of the interpretation mechanism of the language.

"*Meta-Object Protocol are interfaces to the language that give the users the ability to incrementally modify the language's behavior and implementation, as well as the ability to write programs within the language. [...] The Meta-Object Protocol approach is based on the idea that one can and should "open languages up", allowing users to adjust the design and implementation to suit their particular needs. [...] The resulting implementation does not represent a single point in the overall space of language designs, but rather an entire region within that space.*" [3] page 1.

9.13 What is the difference between explicit and implicit MOP?

In an explicit MOP, the instantiation of the entities takes place at the base-level. The base-level contains some *explicit* code that instantiates and uses the entities.

The following source code, in the Java programming language, represents the reification of a method. This code exists at the base-level, and thus defines an explicit MOP.

```
class X {
    public void m() {
        Method m = DummyClass.getMethod("dummyMethod");
        m.invoke(dummyInstance, new Object[0]);
        ...
    }
}
```

In an implicit MOP, the base-level code is not aware of the existence of the meta-level. In the Smalltalk programming language, instance creation happens through the invocation of the `new` message. The `new` message is defined at the meta-level. Any change to the `new` message modifies the semantics of the base-level, without the base-level being aware of the semantic change.

9.14 What is a Meta-Object good for?

A MOP is a powerful mechanism to implement separation of concern, because you can modify the semantics of the interpretation of the language. Using MOPs, you can implement transparently trace mechanisms, distribution mechanisms, new keywords, semantic changes...

9.15 How to implement a MOP in Java?

After the course on Reflection and after reading this document, you should be able to answer this question!?

9.16 How am I suppose to live with primitive types?

As best as you can! Unlike the C♯ programming language, which provides implicit boxing/unboxing, the Java programming language imposes a strong dichotomy between first-class objects and primitive types. The Java programming language provides wrapper classes, such as: `java.lang.Byte`, `java.lang.Void`... However, developers must take care of most conversions from primitive types to wrapped instances (and vice-versa) and make those conversions explicit it their source code.

9.17 The `javadoc` of method `Class.getFields()` says that the method returns fields in any particular order, am I supposed to sort them out myself?

No. Now, if you want to make your code surprise-proof, you better order them yourself, in case the default ordering is different from one JVM to another.

9.18 Why do I get a `StackOverflowError` when using my implementation of a Java `InvocationHandler`?

This exception arises because of a common mistake when using the `Proxy` mechanism. The interface `InvocationHandler` declares a unique method `invoke(Object, Method, Object[])`. The method is called whenever a method is invoked on the `Proxy` instance. The first argument of this method corresponds to the `Proxy` instance. It is tempting to forward the method call to this `Proxy` instance:

```
public Object invoke(Object proxy,
                     Method method,
                     Object[] arguments) throws Throwable {
    System.out.println("Before");
    Object result = method.invoke(proxy, arguments);
    System.out.println("After");

    return result;
}
```

However, we now invoke *again* the method on the `Proxy` instance, which calls the `invoke(Object, Method, Object[])` method of the `InvocationHandler` instance, which forwards the method call to the `Proxy` instance, which calls the... and so forth, until the JVM stack is overflown.

To prevent the `StackOverflowError`, the `InvocationHandler` must forward the method call to an appropriate referenced instance:

```
public class InvocationHandler
    implements java.lang.reflect.InvocationHandler {

    private Object baseObject;

    public InvocationHandler(Object baseObject) {
        this.baseObject = baseObject;
    }
    public Object invoke(Object proxy,
                         Method method,
                         Object[] arguments) throws Throwable {
        System.out.println("Before");
        Object result = method.invoke(this.baseObject, arguments);
        System.out.println("After");

        return result;
    }
}
```

9.19 What is the difference between an inner class and a block anyway?

Hot topic... Please refer to section 7 page 13 for the beginning of an answer.

9.20 What are `ClassLoaders` made for?

Did you carefully read this whole document? Well... Then, please have a look at section 6 page 11.

9.21 What if I take a method in one class and apply it on the same class but defined in a different class loader?

You will get an exception:

```
java.lang.IllegalArgumentException: object is not an instance of
declaring class
    at java.lang.reflect.Method.invoke(Native Method)
    at emn...ClassLoaderTests.main(ClassLoaderTests.java:17)
Exception in thread "main"
```

The following source code demonstrates the exception:

```
package emn.course.reflection.misc;

import java.lang.reflect.Method;
import emn.course.reflection.classloader.PointClassLoader;

public class ClassLoaderTests {
   public static void main(final String[] args) throws Exception {
      final String pointClassName = "emn.course.reflection.misc.Point";

      // Two instances of the class Class, representing                    10
      // the class emn.course.reflection.misc.Point,
      // from two different class loaders.
      final Class pointClass1 =
         ClassLoaderTests.class.getClassLoader().loadClass(pointClassName);
      final Class pointClass2 = new PointClassLoader().loadClass(pointClassName);

      // The getX() method of the first representation
      // of the Point class...
      final Method pointClass1Method = pointClass1.getMethod("getX", new Class[0]);
                                                                           20
      // ... is applied on an instance of the second
      // representation of the Point class.
      pointClass1Method.invoke(pointClass2.newInstance(), new Object[0]);
   }
}
```

9.22 What is the problem between a super class loader and a parent class loader?

There is no real problem between a super class loader and a parent class loader. There only exists a need to be very careful!

Two class loaders are chained if and only if one is the parent class loader of the other; i.e., if only if they are chained using the one argument constructor: `ClassLoader(ClassLoader)`.

There is a very important distinction between association and specialization: A class loader may specialize another (to factorize code), and be chained with yet another class loader (to load and to define classes).

9.23 Why not use subclasses to modify the behavior of a class?

This is actually a very interesting approach to meta-programming. Several projects exist that use this approach. For example, [4] propose Java// (now known as ProActive). Java// only requires changing instantiation code to transform a standard object into an active one. The following source code presents the technique for turning a passive instance of a class `A` into an active one:

```
A a = new A("foo", 7);

A a = (A) Javall.newActive("A", ...);
```

References

[1] Sheng Liang and Gilad Bracha ; *Dyanmic Class Loading in the Java Virtual Machine* ; Proceedings of OOPSLA, pp. 36–44, October 1998.

[2] *The Java Virtual Machine Specification* ; Available at: java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

[3] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow ; *The Art of the Metaobject Protocol* ; MIT Press, 1991.

[4] Denis Caromel, Wilfried Lauser, and Julien Vayssière ; *Towards Seamless Computing and Metaprogramming in Java* ; Proceedings of Concurrency Practice and Experience 10(11–13), pp. 1043–1061, September-November 1998, Editor Geoffrey C. Fox, Published by Wiley & Sons, Ltd.

A ExtendedObject Source Code

```
package emn.course.reflection.extension;

import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.Vector;

import emn.course.reflection.extension.examples.Integer;
import emn.course.reflection.extension.examples.Point;

/**
 * This class is an extension <I>α la</I> Smalltalk of the class Object.
 * This class provides:
 *     - helpers to build inspectors (instVarAt, instVarAtPut, ...).
 *     - explicit messages and control.
 *     - explicit copy and persistence.
 */

public class ExtendedObject {
    public static void main(String[] args) {
        Point p = new Point(1, 2);
        try {
            System.out.print("p.copy()                                    = ");
            System.out.println(p.copy());
            System.out.print("p.duplicate()                                = ");
            System.out.println(p.duplicate());
            System.out.println();

            System.out.print("p = new Point(1,2) = ");
            System.out.print(p);
            System.out.println('\n');

            System.out.print("p.instVarAt(\"x\") = ");
            System.out.println(p.instVarAt("x"));
            System.out.print("p.instVarAt(\"y\") = ");
            System.out.print(p.instVarAt("y"));
            System.out.println('\n');

            System.out.print("p.instVarAt(\"ZERO\")  = ");
            System.out.println(p.instVarAt("ZERO"));
            System.out.print("p.classVarAt(\"ZERO\") = ");
            System.out.println(p.classVarAt("ZERO"));
            System.out.print("p.classVarAt(\"x\")    = ");
            System.out.print(p.classVarAt("x"));
            System.out.println('\n');

            System.out.print("p.storeString() = ");
            System.out.println(p.storeString());
            System.out.print("p.copy()        = ");
            System.out.print(p.copy());
            System.out.println('\n');
```

```
    p.instVarAtPut("y", new java.lang.Integer(200));
    System.out.print("p.instVarAtPut(\"y\", new Integer(200)) = ");
    System.out.println(p);
    p.instVarAtPutInt("x", 100);
    System.out.print("p.instVarAtPutInt(\"x\", 100)          = ");                60
    System.out.print(p);
    System.out.println('\n');

    System.out.println("Point.class.getMethod(\"getClass\", null)");
    System.out.println(Point.class.getMethod("getClass", null));
    System.out.println("Point.class.getMethod(\"hashCode\", null)");
    System.out.println(Point.class.getMethod("hashCode", null));
    // The clone() method is not public.
    // System.out.println(Point.class.getMethod("clone", null));
    System.out.println();                                                        70

    System.out.print("p.receive(\"toString\")            = ");
    System.out.println(p.receive("toString"));
    System.out.print("p.receive(\"storeString\")         = ");
    System.out.println(p.receive("storeString"));
    System.out.print("p.receive(\"add\", p)              = ");
    System.out.println(p.receive("add", p));
    System.out.print("p.receive(\"ad\" + \"d\", p)         = ");
    System.out.println(p.receive("ad" + "d", p));
    System.out.print("p.receive(\"add\", new Integer(100)) = ");                  80
    System.out.println(p.receive("add", new Integer(100)));
    System.out.print("p.receive(\"getClass\")            = ");
    System.out.println(p.receive("getClass"));
    System.out.print("p.receive(\"hashCode\")            = ");
    System.out.println(p.receive("hashCode"));
    System.out.print("p.getMessageCount()              = ");
    System.out.println(p.getMessageCount());
    System.out.print("p.receive(\"zero\")               = ");
    System.out.println(p.receive("zero"));
    System.out.print("p.getMessageCount()              = ");                     90
    System.out.println(p.getMessageCount());
    Message message =
        new Message("setX", new Class[] { int.class }, new Object[] { new Integer(6)});
    System.out.println(
        "Message message = new Message(\"setX\", new Object[] { new Integer(6) })");
    System.out.print("p.receive(message)               = ");
    p.receive(message);
    System.out.println(p);
    System.out.println();
                                                                                 100
    System.out.print("p.receive(\"setX\", new Integer(4))              = ");
    p.receive("setX", new Integer(4));
    System.out.println(p);
    System.out.print("p.receive(\"setY\", new Object[] { new Integer(22) } = ");
    p.receive("setY", new Object[] { new Integer(22)});
    System.out.println(p);
    System.out.print("p.copy()                                       = ");
    System.out.println(p.copy());
    System.out.print("p.duplicate()                                   = ");
    System.out.println(p.duplicate());                                           110
}
```

```
    catch (Exception e) {
        System.err.println("ExtendedObject.Main");
        e.printStackTrace();
    }
}

private static boolean VERBOSE = false;
private int messageCount = 0;
                                                                    120
public Vector instVarIndexes() {
    final Field[] fields = this.getClass().getDeclaredFields();
    final Vector tmp = new Vector();
    try {
        for (int i = 0; i < fields.length; i++) {
            final int m = fields[i].getModifiers();
            if (!Modifier.isStatic(m))
                tmp.addElement(new Integer(i));
        }
    }                                                               130
    catch (Exception e) {
        System.out.println("MetaObject.instVarAtIndexes()");
    }
    return tmp;
}
public Vector classVarIndexes() {
    final Field[] fields = this.getClass().getDeclaredFields();
    final Vector tmp = new Vector();
    try {
        for (int i = 0; i < fields.length; i++) {               140
            int m = fields[i].getModifiers();
            if (Modifier.isStatic(m))
                tmp.addElement(new Integer(i));
        }
    }
    catch (Exception e) {
        System.out.println("ExtendedObject.classVarAtIndexes()");
        e.printStackTrace();
    }
    return tmp;                                                     150
}

public Object instVarAt(int i) {
    final Field[] fields = this.getClass().getDeclaredFields();
    Object tmp = null;
    try {
        final int m = fields[i].getModifiers();
        if (!Modifier.isStatic(m)) {
            fields[i].setAccessible(true);
            tmp = fields[i].get(this);                              160
        }
    }
    catch (Exception e) {
        System.out.print("ExtendedObject.instVarAt(");
        System.out.print(i);
        System.out.println(')');
        e.printStackTrace();
    }
```

```
    return tmp;
}
public String classVarAt(int i) {
    Field[] fields = this.getClass().getDeclaredFields();
    String tmp = "n'existe pas";
    try {
        Field f = fields[i];
        int m = f.getModifiers();
        if (Modifier.isStatic(m))
            tmp = f.get(null).toString();
    }
    catch (Exception e) {
        System.out.println("ExtendedObject.classVarAt(" + i + ")" + e);
    }
    return tmp;
}
public Object instVarAt(final String name) {
    Object result = null;
    try {
        final Field f = this.getClass().getDeclaredField(name);
        f.setAccessible(true);
        int m = f.getModifiers();
        if (!Modifier.isStatic(m)) {
            result = f.get(this);
        }
    }
    catch (Exception e) {
        System.err.print("ExtendedObject.instVarAt(");
        System.err.print(name);
        System.err.println(")");
        e.printStackTrace();
    }
    finally {
        return result;
    }
}
public void instVarAtPut(String name, Object value) {
    try {
        Field f = this.getClass().getDeclaredField(name);
        f.setAccessible(true);
        f.set(this, value);
    }
    catch (Exception e) {
        System.err.println("ExtendedObject.instVarAtPut(String, Object)");
        e.printStackTrace();
    }
}
public void instVarAtPutInt(String name, int value) {
    try {
        Field f = this.getClass().getDeclaredField(name);
        f.setAccessible(true);
        f.setInt(this, value);
    }
    catch (Exception e) {
        System.err.println("ExtendedObject.instVarAtPutInt()");
        e.printStackTrace();
    }
```

```
}
public void instVarAtPutString(int i, Object value) {
    Field[] fields = this.getClass().getDeclaredFields();
    String tmp = null;
    try {                                                                    230
        Field f = fields[i];
        f.setAccessible(true);
        f.set(this, value);
    }
    catch (Exception e) {
        System.err.println("ExtendedObject.instVarAtPutString()");
        e.printStackTrace();
    }
}
public void instVarAtPutObject(int i, Object value) {                       240
    final Field[] fields = this.getClass().getDeclaredFields();
    try {
        final int m = fields[i].getModifiers();
        if (!Modifier.isStatic(m)) {
            fields[i].setAccessible(true);
            fields[i].set(this, value);
        }
    }
    catch (Exception e) {
        System.out.println("ExtendedObject.instVarAtPutObject()");           250
        e.printStackTrace();
    }
}
public Object classVarAt(final String name) {
    Object result = null;
    try {
        Field f = this.getClass().getDeclaredField(name);
        f.setAccessible(true);
        int m = f.getModifiers();
        if (Modifier.isStatic(m)) {                                          260
            result = f.get(null);
        }
    }
    catch (Exception e) {
        System.err.print("ExtendedObject.classVarAt(");
        System.err.print(name);
        System.err.println(')');
        e.printStackTrace();
    }
    finally {                                                                270
        return result;
    }
}

public int getMessageCount() {
    return this.messageCount;
}
public Object receive(final String selector, final Object[] args) {
    Class[] classes = new Class[0];
    Object result = null;                                                    280

    if (args != null) {
```

```
        final int lo = args.length;
        classes = new Class[lo];
        for (int i = 0; i < lo; i++) {
            classes[i] = args[i].getClass();
        }
    }

    try {                                                                    290
        final Method method = this.getClass().getMethod(selector, classes);
        if (VERBOSE) {
            System.out.print("Before invoking method: ");
            System.out.println(method);
        }
        this.messageCount++;
        result = method.invoke(this, args);
        if (VERBOSE) {
            System.out.print("Result of the invocation: ");
            System.out.println(result);                                      300
        }
        if (VERBOSE) {
            System.out.print("After invoking method: ");
            System.out.println(method);
        }
    }
    catch (Exception e) {
        e.printStackTrace();
    }
                                                                             310
    return result;
}
public Object receive(String selector) {
    return this.receive(selector, null);
}
public Object receive(String selector, Object arg1) {
    return this.receive(selector, new Object[] { arg1 });
}
public Object receive(String selector, Object arg1, Object arg2) {
    return this.receive(selector, new Object[] { arg1, arg2 });              320
}

public Object receive(Message aMessage) {
    return this.receive(aMessage.getSelector(), aMessage.getArgumentValues());
}

public void initialize(Object[] initArgs) {
    try {
        final Vector instVarIndexes = this.instVarIndexes();
        for (int i = 0; i < instVarIndexes.size(); i++) {                    330
            int j = ((Integer) instVarIndexes.elementAt(i)).intValue();
            instVarAtPutObject(j, initArgs[i]);
        }

    }
    catch (Exception e) {
        System.out.println("initialize(Object[])");
        e.printStackTrace();
    }
```

```
}

public Object copy() {
    Object tmp = null;
    try {
        tmp = this.getClass().newInstance();
        Field[] fields = this.getClass().getDeclaredFields();
        for (int i = 0; i < fields.length; i++) {
            fields[i].setAccessible(true);
            fields[i].set(tmp, fields[i].get(this));
        }
    }
    catch (Exception e) {
        System.out.println("ExtendedObject.copy()");
        e.printStackTrace();
    }
    finally {
        return tmp;
    }
}
public ExtendedObject duplicate() {
    ExtendedObject result = null;
    try {
        result = (ExtendedObject) this.getClass().newInstance();
        Iterator iterator = this.instVarIndexes().iterator();
        while (iterator.hasNext()) {
            final int i = ((Integer) iterator.next()).intValue();
            result.instVarAtPutObject(i, instVarAt(i));
        }
    }
    catch (Exception e) {
        System.err.println("ExtendedObject.duplicate()");
        e.printStackTrace();
    }
    return result;
}
public String storeString() {
    // One assumption:
    //      Static fields (a.k.a. class variables)
    //      are used only to store constant values.

    final StringBuffer buffer = new StringBuffer();
    final Field[] fields = this.getClass().getDeclaredFields();

    // I compute the list of the non-static fields.
    final ArrayList nonStaticFieldList = new ArrayList();
    for (int i = 0; i < fields.length; i++) {
        if (!Modifier.isStatic(fields[i].getModifiers())) {
            fields[i].setAccessible(true);
            nonStaticFieldList.add(fields[i]);
        }
    }

    try {
        buffer.append("new ");
        buffer.append(this.getClass().getName());
        buffer.append('(');
```

```
            final int size = nonStaticFieldList.size();
            for (int i = 0; i < size; i++) {
                buffer.append(((Field) nonStaticFieldList.get(i)).get(this));
                if (i < size - 1) {
                    buffer.append(", ");
                }
            }
            buffer.append(')');
        }
        catch (Exception e) {
            System.out.println("ExtendedObject.storeString()");
            e.printStackTrace();
        }
        return buffer.toString();
    }
}
```

B ClassDescription Source Code

```
package emn.course.reflection.extension;


import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.util.Iterator;
import java.util.Vector;
                                                                                10
import emn.course.reflection.extension.examples.Point;


/**
 * This class defines extra methods to extend the class Class.
 * This class provides method for:
 *     - Metrics.
 *     - Class signature.
 *     - Class member signature.
 *                                                                              20
 * The class Class is final in Java. We use the delegation to add
 * the missing methods.
 */


public class ClassDescription {
    public static void main(String[] args) throws Exception {
        System.out.println("new ClassDescription(Point.class).getShortName()");
        System.out.println(new ClassDescription(Point.class).getShortName());
        System.out.println();                                                   30

        System.out.println("new ClassDescription(Point.class).getName()");
        System.out.println(new ClassDescription(Point.class).getName());
        System.out.println();

        System.out.println("new ClassDescription(\"java.applet.Applet\").printHierarchy()");
        new ClassDescription("java.applet.Applet").printHierarchy();
        System.out.println();

        System.out.println("new ClassDescription(\"java.awt.Point\").printHierarchy()");  40
        new ClassDescription("java.awt.Point").printHierarchy();
        System.out.println();

        System.out.println("new ClassDescription(\"Point.class\").javap()");
        new ClassDescription(Point.class).javap();
        System.out.println();

        System.out.println("new ClassDescription(Point.class).metrics()");
        new ClassDescription(Point.class).metrics();
    }                                                                           50

    private final static String[] BASIC = { "class", "interface" };
    private final static String[] EXTENDED = { "extends", "implements" };
```

```
private final Class currentClass;

public ClassDescription(final Class aClass) {
    this.currentClass = aClass;
}
public ClassDescription(final String className) {
    Class temporaryClass = null;
    try {
        temporaryClass = Class.forName(className);
    }
    catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    this.currentClass = temporaryClass;
}

public String getName() {
    return currentClass.getName();
}
public String getShortName() {
    final String fullName = this.getName();
    if (fullName.lastIndexOf('.') > -1) {
        return fullName.substring(fullName.lastIndexOf('.') + 1);
    }
    else {
        return fullName;
    }
}
public String getProperties() {
    return Modifier.toString(this.currentClass.getModifiers());
}
public boolean isAbstract() {
    return Modifier.isAbstract(this.currentClass.getModifiers());
}
public boolean isFinal() {
    return Modifier.isFinal(currentClass.getModifiers());
}
public Object newInstance() throws Exception {
    return this.currentClass.newInstance();
}
public Vector getMethods() {
    final Vector allMethods = new Vector();
    final Method[] arrayOfAllMethods = this.currentClass.getMethods();
    for (int i = 0; i < arrayOfAllMethods.length; i++) {
        allMethods.addElement(arrayOfAllMethods[i]);
    }
    return allMethods;
}
public Vector getDeclaredMethods() {
    final Vector allMethods = new Vector();
    final Method[] arrayOfAllMethods = this.currentClass.getDeclaredMethods();
    for (int i = 0; i < arrayOfAllMethods.length; i++) {
        allMethods.addElement(arrayOfAllMethods[i]);
    }
    return allMethods;
}
public void printHierarchy() {
```

```
    this.printHierarchy(currentClass, 0);
}
private void printHierarchy(Class type, int depth) {
    // The current type has no super-type.
    if (type == null) {
        return;
    }

                                                                                120
    // I print the type information.
    for (int i = 0; i < depth; i++) {
        System.out.print('\t');
    }
    String[] labels =
        (depth == 0 ? ClassDescription.BASIC : ClassDescription.EXTENDED);
    System.out.print(labels[type.isInterface() ? 1 : 0]);
    System.out.print(" ");
    System.out.println(type.getName());
                                                                                130

    // I print the super-type information.
    depth = depth + 1;
    Class[] interfaces = type.getInterfaces();
    for (int i = 0; i < interfaces.length; i++) {
        this.printHierarchy(interfaces[i], depth);
    }
    this.printHierarchy(type.getSuperclass(), depth);
}
public void javap() {                                                           140
    System.out.print("\njavap-like output of ");
    System.out.println(this.currentClass);


    Field[] fields;
    Method[] methods;
    Class[] interfaces;
    Constructor[] constructors;

                                                                                150
    System.out.print(this.currentClass.toString());
    if (this.currentClass.getSuperclass() != null) {
        System.out.print(" extends ");
        System.out.print(this.currentClass.getSuperclass().getName());
    }
    interfaces = this.currentClass.getInterfaces();
    if (interfaces.length > 0) {
        System.out.print(" implements ");
        for (int j = 0; j < interfaces.length; j++) {
            System.out.print(interfaces[j].getName());                          160
            if (j < interfaces.length) {
                System.out.print(", ");
            }
        }
    }


    System.out.println(" {\n\t// Declared fields.");
```

```
    fields = this.currentClass.getDeclaredFields();
    for (int j = 0; j < fields.length; j++) {
        System.out.print('\t');
        System.out.print(fields[j]);
        System.out.println(';');
    }


    System.out.println("\n\t// Declared methods.");
    methods = this.currentClass.getDeclaredMethods();
    for (int j = 0; j < methods.length; j++) {
        System.out.print('\t');
        System.out.print(methods[j]);
        System.out.println(';');
    }


    System.out.println("\n\t// Declared constructors.");
    constructors = this.currentClass.getDeclaredConstructors();
    for (int j = 0; j < constructors.length; j++) {
        System.out.print('\t');
        System.out.print(constructors[j]);
        System.out.println(';');
    }


    System.out.println("\n\t// Public fields only.");
    fields = this.currentClass.getFields();
    for (int j = 0; j < fields.length; j++) {
        System.out.print("\t// ");
        System.out.print(fields[j]);
        System.out.println(';');
    }


    System.out.println("\n\t// All public methods, including inherited methods.");
    methods = this.currentClass.getMethods();
    for (int j = 0; j < methods.length; j++) {
        System.out.print("\t// ");
        System.out.print(methods[j]);
        System.out.println(';');
    }


    System.out.println("\n\t// All public constructors.");
    constructors = this.currentClass.getConstructors();
    for (int j = 0; j < constructors.length; j++) {
        System.out.print("\t// ");
        System.out.print(constructors[j]);
        System.out.println(';');
    }
    System.out.println('}');
}
public String toString() {
    // I use the delegation to improve
    // the toString() method of the class Class.
    final StringBuffer buffer = new StringBuffer();
```

```
    if (isFinal()) {
        buffer.append("final ");
    }
    if (isAbstract()) {
        buffer.append("abstract ");
    }
    buffer.append(currentClass);
    return buffer.toString();
}


public Vector getSuperclasses() {
    return this.getSuperclasses(this.currentClass, new Vector());
}
private Vector getSuperclasses(final Class aClass, Vector result) {
    if (aClass == null) {
        return result;
    }
    else {
        result.addElement(aClass);
        this.getSuperclasses(aClass.getSuperclass(), result);
        return result;
    };
}
public int instanceSize() {
    return this.currentClass.getDeclaredFields().length;
}
public void metrics() {
    System.out.print("Analysis of ");
    System.out.println(this.currentClass.getName());
    System.out.print("\tNumber of super-classes: ");
    System.out.println(this.getSuperclasses().size() - 1);


    System.out.print("\tNumber of instance variables: ");
    System.out.println(this.getDeclaredInstanceMethods().size());


    int nbClassMethods = getDeclaredClassMethods().size();
    System.out.print("\tNumber of class variables: ");
    System.out.println(this.getDeclaredClassMethods().size());


    System.out.print("\tNumber of constructors: ");
    System.out.println(this.numberOfConstructors());
}
public int numberOfPublicMethods() {
    return this.currentClass.getMethods().length;
}
public int numberOfConstructors() {
    return this.currentClass.getDeclaredConstructors().length;
}
public int numberOfMethods() {
    return this.currentClass.getDeclaredMethods().length;
}
public int numberOfDeclaredPublicMethods() {
    int publicMethods = 0;
```

```
    final Iterator iterator = getDeclaredMethods().iterator();
    while (iterator.hasNext()) {
        int i = ((Method) iterator.next()).getModifiers();
        if (Modifier.isPublic(i)) {
            publicMethods = publicMethods + 1;
        }
    }
    return publicMethods;                                                    290
}
public Vector classVarIndexes() {
    final Field[] fields = currentClass.getDeclaredFields();
    final Vector tmp = new Vector();
    try {
        for (int i = 0; i < fields.length; i++) {
            Field f = fields[i];
            if (f.isAccessible()) {
                int m = f.getModifiers();
                if (Modifier.isStatic(m) & !Modifier.isPrivate(m)) {         300
                    tmp.addElement(f.getName());
                    tmp.addElement(f.getType());
                    tmp.addElement(new Integer(i));
                }
            }
        };
    }
    catch (Exception e) {
        e.printStackTrace();
    }                                                                        310
    return tmp;
}
public Vector getDeclaredClassMethods() {
    final Vector allMethods = new Vector();
    final Method[] arrayOfAllMethods = currentClass.getDeclaredMethods();
    for (int i = 0; i < arrayOfAllMethods.length; i++) {
        Method m = arrayOfAllMethods[i];
        if (Modifier.isStatic(m.getModifiers())) {
            allMethods.addElement(m);
        }                                                                    320
    }
    return allMethods;
}
public Vector getDeclaredInstanceMethods() {
    final Vector allMethods = new Vector();
    final Method[] arrayOfAllMethods = currentClass.getDeclaredMethods();
    for (int i = 0; i < arrayOfAllMethods.length; i++) {
        Method m = arrayOfAllMethods[i];
        if (!Modifier.isStatic(m.getModifiers())) {
            allMethods.addElement(m);                                        330
        }
    }
    return allMethods;
}
public Vector instVarIndexes() {
    final Field[] fields = currentClass.getDeclaredFields();
    final Vector tmp = new Vector();
    try {
        for (int i = 0; i < fields.length; i++) {
```

```
                Field f = fields[i];                                                    340
                if (f.isAccessible()) {
                    int m = f.getModifiers();
                    if (!Modifier.isStatic(m) & !Modifier.isPrivate(m)) {
                        tmp.addElement(f.getName());
                        tmp.addElement(f.getType());
                        tmp.addElement(new Integer(i));
                    }
                }
            };
        }                                                                               350
        catch (Exception e) {
            e.printStackTrace();
        }
        return tmp;
    }
    public int numberOfAllMethods() {
        Vector v = getSuperclasses();
        int result = 0;
        Iterator iterator = v.iterator();
        while (iterator.hasNext()) {                                                    360
            int i = (((Class) iterator.next()).getDeclaredMethods()).length;
            result += i;
        }
        return result;


    }
}
```

C MetaObject Source Code

```
package emn.course.reflection.extension;

import java.lang.reflect.Method;

import emn.course.reflection.extension.examples.Point;

public class MetaObject extends Object {
    public static void main(final String[] args) {
        final MetaObject metaObject = new MetaObject(new Point(5, 10));
        System.out.println("MetaObject metaObject          = new MetaObject(new Point(5, 10))");
        final Message zeroMessage = metaObject.reify("getX");
        System.out.println("Message zeroMessage            = metaObject.reify(\"getX\")");
        System.out.print("metaObject.receive(zeroMessage) = ");
        System.out.println(metaObject.receive(zeroMessage));
    }

    private final Object baseObject;

    public MetaObject(final Object baseObject) {
        this.baseObject = baseObject;
    }

    public Object getBaseObject() {
        return this.baseObject;
    }
    public Message reify(final String unarySelector) {
        return this.reify(unarySelector, Message.NO_ARGUMENT_TYPE);
    }
    public Message reify(final String unarySelector, final Class[] arguments) {
        Message message = null;
        try {
            final Method method =
                this.getBaseObject().getClass().getDeclaredMethod(unarySelector, arguments);
            message = new Message(method.getName(), arguments);
        }
        catch (NoSuchMethodException nsme) {
            nsme.printStackTrace();
        }
        return message;
    }
    public Object receive(final Message message) {
        Object result = null;

        try {
            final Method method =
                this.baseObject.getClass().getMethod(
                    message.getSelector(),
                    message.getArgumentTypes());

            System.out.print("\n\tBefore invoking method: ");
            System.out.println(method);

            result = method.invoke(this.getBaseObject(), message.getArgumentValues());
```

```
            System.out.print("\tResult of the invocation: ");
            System.out.println(result);
            System.out.print("\tAfter invoking method: ");
            System.out.println(method);
        }
        catch (Exception e) {                                                    60
            System.out.print("Selector      : ");
            System.out.println(message.getSelector());
            System.out.print("Argument types : ");
            System.out.println(message.getArgumentTypes());
            System.out.print("Argument values: ");
            System.out.println(message.getArgumentValues());
            e.printStackTrace();
        }

        return result;                                                           70
    }
}
```

D Proxy Source Code

```
package emn.course.reflection.proxy;

import java.lang.reflect.Method;

public class InvocationHandler implements java.lang.reflect.InvocationHandler {
    private Object baseObject;

    public InvocationHandler(Object baseObject) {
        this.baseObject = baseObject;
    }

    public Object invoke(Object proxy, Method method, Object[] arguments)
        throws Throwable {

        System.out.println("Before");
        Object result = method.invoke(baseObject, arguments);
        System.out.print("Result shall be: ");
        System.out.println(result);
        System.out.println("After");

        return result;
    }
}
```

```
package emn.course.reflection.proxy.examples;

public interface ILine {
    int getX1();
    int getY1();
    int getX2();
    int getY2();
}

public class Line implements ILine {
    private IPoint origin, destination;

    public Line(IPoint origin, IPoint destination) {
        this.origin = origin;
        this.destination = destination;
    }

    public int getX1() {
        return this.origin.getX();
    }
    public int getY1() {
        return this.origin.getY();
    }
    public int getX2() {
        return this.destination.getX();
    }
    public int getY2() {
        return this.destination.getY();
```

```
    }
}

public interface IPoint {
    public int getX();
    public int getY();
    public void setX(int x);
    public void setY(int y);
}

public class Point implements IPoint {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return this.x;
    }
    public int getY() {
        return this.y;
    }
    public void setX(int x) {
        this.x = x;
    }
    public void setY(int y) {
        this.y = y;
    }
    public boolean equals(Object o) {
        if (o instanceof IPoint) {
            return this.getX() == ((IPoint) o).getX() & this.getY() == ((IPoint) o).getY();
        }
        return false;
    }
    public String toString() {
        final StringBuffer buffer = new StringBuffer();

        buffer.append(this.getX());
        buffer.append('@');
        buffer.append(this.getY());

        return buffer.toString();
    }
}
```

```
package emn.course.reflection.proxy.examples;

import java.lang.reflect.Proxy;

import emn.course.reflection.extension.ClassDescription;
import emn.course.reflection.proxy.InvocationHandler;

public class Main {
    public static void main(String[] args) throws Exception {
```

```
System.out.println("IPoint p1 = new Point(10, 15)");
final IPoint p1 = new Point(10, 15);

System.out.println(
    "IPoint p2 =\n\t(IPoint) Proxy.newProxyInstance(. . .)");
final IPoint p2 =
    (IPoint) Proxy.newProxyInstance(
        IPoint.class.getClassLoader(),
        new Class[] { IPoint.class },
        new InvocationHandler(p1));
System.out.println();

System.out.println(
    "IPoint p3 =\n\t(IPoint) Proxy.newProxyInstance(. . .)");
final IPoint p3 =
    (IPoint) Proxy.newProxyInstance(
        IPoint.class.getClassLoader(),
        new Class[] { IPoint.class },
        new InvocationHandler(p2));
System.out.println();

System.out.println("ILine p1 = new Line(10, 15)");
final ILine l1 = new Line(p1, p2);

System.out.println(
    "IPoint l2 =\n\t(ILine) Proxy.newProxyInstance(. . .)");
final ILine l2 =
    (ILine) Proxy.newProxyInstance(
        ILine.class.getClassLoader(),
        new Class[] { ILine.class },
        new InvocationHandler(l1));
System.out.println();

System.out.println("p1 ?");
System.out.println(p1);
System.out.println();

System.out.println("p2 ?");
System.out.println(p2);
System.out.println();

System.out.println("l1 ?");
System.out.println(l1);
System.out.println();

System.out.println("l2 ?");
System.out.println(l2);
System.out.println();

System.out.println("p1 == p2 ?");
System.out.println(p1 == p2);
System.out.println();

System.out.println("p1.getClass() ?");
System.out.println(p1.getClass());
System.out.println("p2.getClass() ?");
System.out.println(p2.getClass());
```

```
        System.out.println();

        System.out.println("p1.equals(p2) ?");
        System.out.println(p1.equals(p2));
        System.out.println();

        System.out.println("p2.equals(p1) ?");
        System.out.println(p2.equals(p1));
        System.out.println();

        System.out.println("Main.class.getDeclaredField(\"class$0\").get(null) ?");
        System.out.println(Main.class.getDeclaredField("class$0").get(null));
        System.out.println("Main.class.getDeclaredField(\"class$1\").get(null) ?");
        System.out.println(Main.class.getDeclaredField("class$1").get(null));
        System.out.println("Main.class.getDeclaredField(\"class$2\").get(null) ?");
        System.out.println(Main.class.getDeclaredField("class$2").get(null));
        System.out.println();

        System.out.println("Main.class.getDeclared... == IPoint.class ?");
        System.out.println(Main.class.getDeclaredField("class$0").get(null) == IPoint.class);
        System.out.println();

        final Class proxy0Class = Class.forName("$Proxy0");
        final ClassDescription proxy0ClassDescription =
            new ClassDescription(proxy0Class);
        proxy0ClassDescription.printHierarchy();
        // proxy0ClassDescription.javap();

        final Class proxy1Class = Class.forName("$Proxy1");
        final ClassDescription proxy1ClassDescription =
            new ClassDescription(proxy1Class);
        proxy1ClassDescription.printHierarchy();
        // proxy0ClassDescription.javap();
    }
}
```

E ClassLoader Source Code

```
package emn.course.reflection.classloader.examples;

import java.lang.reflect.InvocationTargetException;

import emn.course.reflection.classloader.ModifyingPointClassLoader;
import emn.course.reflection.classloader.PointClassLoader;

public class Launcher {
    public static void main(final String[] args) {
        try {                                                                    10
            final String[] mainArguments = new String[0];

            final PointClassLoader classLoader = new PointClassLoader();
            final Class PointClass =
                classLoader.loadClass("java.awt.Point");
            System.out.print("Point.class = ");
            System.out.println(Point.class);
            System.out.print("\t(");
            System.out.print(Point.class.getClassLoader());
            System.out.println(')');                                             20
            System.out.print("PointClass  = ");
            System.out.println(PointClass);
            System.out.print("\t(");
            System.out.print(PointClass.getClassLoader());
            System.out.println(')');
            System.out.print("Point.class == PointClass? ");
            System.out.println(Point.class == PointClass);
            System.out.println();

            System.out.println("PointMain.main(new String[0])");                 30
            PointMain.main(mainArguments);
            System.out.println();

            System.out.println(
                "classLoader.loadClass(\"emn.course.reflection.classloader.examples.PointMain\")");
            System.out.println(
                "PointMainClass.getMethod(\"main\", new Class[] { String[].class }).invoke(null, new Object[] { new S
            final Class PointMainClass =
                classLoader.loadClass("emn.course.reflection.classloader.examples.PointMain");
            PointMainClass.getMethod("main", new Class[] { String[].class }).invoke(     40
                null,
                new Object[] { mainArguments });
            System.out.println();

            System.out.println(
                "classLoader.loadClass(\"emn.course.reflection.classloader.examples.PointMain\")");
            System.out.println(
                "PointMainClass.getMethod(\"main\", new Class[] { String[].class }).invoke(null, new Object[] { new S
            final ModifyingPointClassLoader modifyingClassLoader =
                new ModifyingPointClassLoader();                                 50
            final Class ModifiedPointMainClass =
                modifyingClassLoader.loadClass(
                    "emn.course.reflection.classloader.examples.PointMain");
            ModifiedPointMainClass.getMethod(
```

```
                "main",
                new Class[] { String[].class }).invoke(
                null,
                new Object[] { mainArguments });
        }
        catch (ClassNotFoundException cnfe) {
            cnfe.printStackTrace();
        }
        catch (IllegalAccessException iae) {
            iae.printStackTrace();
        }
        catch (InvocationTargetException ite) {
            ite.printStackTrace();
        }
        catch (NoSuchMethodException nsme) {
            nsme.printStackTrace();
        }
    }
}
```

```
package emn.course.reflection.classloader.examples;

public class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        // System.out.print("\tgetX() == ");
        // System.out.println(this.x);
        return this.x;
    }
    public int getY() {
        // System.out.print("\tgetY() == ");
        // System.out.println(this.x);
        return this.x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public void setY(int y) {
        this.y = y;
    }
    public boolean equals(Point p) {
        return this.getX() == p.getX() & this.getY() == p.getY();
    }
    public String toString() {
        final StringBuffer buffer = new StringBuffer();

        buffer.append(this.getX());
        buffer.append('@');
        buffer.append(this.getY());
```

```
        return buffer.toString();
    }
}

public class PointMain {
    public static void main(String[] args) {
        System.out.print("\tPoint.class = ");
        System.out.println(Point.class);
        System.out.print("\t\t(");
        System.out.print(Point.class.getClassLoader());
        System.out.println(')');

        final Point p = new Point(1, 5);
        System.out.println(p);
    }
}
```

```
package emn.course.reflection.classloader;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class PointClassLoader extends ClassLoader {
    private static final String Directory =
        System.getProperty("java.class.path").substring(
            0,
            System.getProperty("java.class.path").indexOf(';'))
            + File.separator;
    private static final ClassLoader Parent =
        PointClassLoader.class.getClassLoader();

    protected Class findClass(String name) {
        Class newClass = null;
        String osName = PointClassLoader.Directory + name.replace('.', '/') + ".class";
        System.out.print("Looking for ");
        System.out.println(osName);

        try {
            FileInputStream fis = new FileInputStream(osName);
            int length = (int) new File(osName).length();
            byte[] bytes = new byte[length];
            fis.read(bytes);
            fis.close();

            newClass = this.defineClass(name, bytes, 0, length);
            this.definePackage(
                newClass.getName().substring(0, newClass.getName().lastIndexOf('.')),
                "",
                "",
                "",
                "",
                "",
                "",
                "",
```

```
                null);
        }
        catch (FileNotFoundException fnfe) {
            // fnfe.printStackTrace();
        }
        catch (IOException ioe) {
            // ioe.printStackTrace();
        }
        catch (IllegalArgumentException iae) {
            // The package already exists.
            // iae.printStackTrace();
        }
        catch (ClassFormatError cfe) {
            System.err.print("The file ");
            System.err.print(osName);
            System.err.println(" produces an error.");
            cfe.printStackTrace();
        }

        return newClass;
    }
    public Class loadClass(String name, boolean resolve)
        throws ClassNotFoundException {

        Class c = null;
        if (name.startsWith("emn")) {
            c = this.findClass(name);
        }
        else {
            c = PointClassLoader.Parent.loadClass(name);
        }
        if (resolve) {
            this.resolveClass(c);
        }
        return c;
    }
}
```

```
package emn.course.reflection.classloader;

public class ModifyingPointClassLoader extends PointClassLoader {
    public Class loadClass(String name, boolean resolve)
        throws ClassNotFoundException {

        Class c = null;
        if (name.equals("emn.course.reflection.classloader.examples.Point")) {
            c = this.defineNewPointClass();
        }
        else {
            c = super.loadClass(name, resolve);
        }
        return c;
    }
    private Class defineNewPointClass() {
        final byte[] bytes = new byte[] {
```

```
            −54, −2, −70, −66, 0, 3, 0, 45, 0, 72,
            1, 0, 48, 101, 109, 110, 47, 99, 111, 117,
            ...                                                    20
            68, 0, 69, 0, 1, 0, 1, 0, 70, 0,
            0, 0, 2, 0, 71};
        return this.defineClass(
            "emn.course.reflection.classloader.examples.Point",
            bytes,
            0,
            bytes.length);
    }
}
```

F Block Source Code

```
package emn.course.reflection.block.examples;

import java.util.Vector;

import emn.course.reflection.block.Block;

public class Test {
    public static void main(String[] args) {
        System.out.println("for (int i = 1; i < 5; System.out.println(i++));");
        Block b1 = new Block(".", "for (int i = 1; i < 5; System.out.println(i++));");
        System.out.println(b1.value());

        System.out.println(
            "\nfor (int i = 1; i < 5; System.out.println(i++)); return \"Hello\";");
        Block b2 =
            new Block(
                ".",
                "for (int i = 1; i < 5; System.out.println(i++)); return \"Hello\";");
        System.out.println(b2.value());

        System.out.println(
            "\nint k | for (int i = k; i < 5; System.out.println(i++)); return \"Hello\";");
        Block b3 =
            new Block(
                ".",
                "int k |for (int i = k; i < 5; System.out.println(i++)); return \"Hello\";");
        System.out.println(b3.value(3));

        System.out.println("\nboolean b | return new Boolean(b);");
        Block b4 = new Block(".", "boolean b | return new Boolean(b);");
        System.out.println(b4.value(true));

        System.out.println("\nObject o | System.out.print(o);");
        Block b5 = new Block(".", "Object o | System.out.print(o);");
        System.out.println(b5.value(new String("Hello")));

        System.out.println("\nmethod applyAVectorFromTo()");
        System.out.println("\tObject o | System.out.print(o);");
        System.out.println("\tnew Vector() with 'H', 'e', 'l', 'l', and 'o'");
        Block b6 = new Block(".", "Object o | System.out.print(o);");
        Vector v = new Vector();
        v.addElement("H");
        v.addElement("e");
        v.addElement("l");
        v.addElement("l");
        v.addElement("o");
        b6.applyAVectorFromTo(v, 0, 5);
        System.out.println();

        System.out.println("\nString a, String b | System.out.println(a.equals(b));");
        Block b7 =
            new Block(".", "String a, String b | System.out.println(a.equals(b));");
        System.out.println("\twith: new String(\"Hi\"), new String(\"Bye\")");
        System.out.println(
```

```
        b7.value(new Object[] { new String("Hello"), new String("Pas coucou")}));

        System.out.println("\nString a, String b | a = b;");
        Block b8 = new Block(".", "String a, String b | a = b;");
        System.out.println("\twith: new String(\"Hi\"), new String(\"Bye\")");
        String b8a = "Hi";
        String b8b = "Bye";
        System.out.println(b8.value(new Object[] { b8a, b8b }));
        System.out.println(b8a);
        System.out.println(b8b);

        System.out.println("\nObject[] o | o[0] = o[1];");
        Block b9 = new Block(".", "Object[] o | o[0] = o[1];");
        System.out.println("\twith: new Object[] { \"Hi\", \"Bye\" }");
        Object[] b9o = new Object[] { "Hi", "Bye" };
        System.out.println(b9o[0]);
        System.out.println(b9o[1]);
        System.out.println(b9.value(new Object[] { b9o }));
        System.out.println(b9o[0]);
        System.out.println(b9o[1]);
    }
}
```

```
package emn.course.reflection.block;

import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.Writer;
import java.io.BufferedWriter;
import java.lang.reflect.Method;
import java.util.Date;
import java.util.Vector;

/**
 * This class acts 'as' a BlockContextTemplate in Smalltalk.
 * To buil a new instance of me, you have to give a piece of
 * Java code, then I will understand some messages, allowing you
 * to use me as a BlockContextTemplate inside your own method.
 *
 * author  Yann-Gal Guhneuc
 * version Version 0.2, 01/12/13
 */

public class Block {
    /* To store a reference of the new created class */
    private Class aClass;
    public Block(String classPath, String corpus) {
        try {
            // Fist, I give a unique name to the new class I will create.
            String className = "Block" + new Date().hashCode();
            className = className.replace('-', '_');

            // Then I open/create a new file.
```

```
        OutputStream os = new FileOutputStream(className + ".java");
        Writer w = new BufferedWriter(new OutputStreamWriter(os));

        // If I can find the symbol |, it means that the user has declared arguments for its block,
        // so I catch them to give them as parameters of the new function, and remove them from the
        // body of the unique function of the new class.
        int index = corpus.indexOf('|');
        String blocksArgs = new String("");                                          40
        if (index > 0) {
            blocksArgs = corpus.substring(0, index);
            corpus = corpus.substring(index + 1);
        }

        // Now, I can write the new class.
        w.write("public class " + className + " {");
        w.write("public static Object blockValue(" + blocksArgs + ") {");
        w.write(corpus);
                                                                                     50
        // If no return can be found into the corpus variable, the I add a Void.TYPE return.
        if (corpus.indexOf("return") == -1) {
            w.write("return Void.TYPE;");
        }

        // Now, I close all the brackets.
        w.write("}}");

        // And I close the streams I openned.
        w.close();                                                                   60
        os.close();

        // I compile the new file and take a reference on it.
        sun.tools.javac.Main compiler = new sun.tools.javac.Main(System.err, "javac");
        compiler.compile(
            new String[] {
                "-classpath",
                classPath,
                "-d",
                System.getProperty("user.dir"),                                      70
                className + ".java" });
        aClass = Class.forName(className);

        // I can now remove the 'temporary' files I have created.
        new File(System.getProperty("user.dir") + File.separator + className + ".java")
            .delete();
        new File(
            System.getProperty("user.dir") + File.separator + className + ".class")
            .delete();
    }                                                                                80
    catch (Exception e) {
        System.err.println("Block( " + corpus + " )");
        e.printStackTrace();
    }
}
public void applyAVectorFromTo(Vector v, int beginindex, int endindex) {
    for (int i = beginindex; i < endindex; i++) {
        this.value(v.elementAt(i));
    }
```

```
}
public Object value() {
    Object r = null;
    try {
        /* I know that ONE and only ONE method is defined */
        Method m = aClass.getDeclaredMethod("blockValue", new Class[0]);
        r = m.invoke(aClass.newInstance(), null);
    }
    catch (Exception e) {
        System.err.println("Block.value()");
        e.printStackTrace();
    }
    if (r == null)
        return Void.TYPE;
    return r;
}
private Object value(
    Class[] blockMethodArguments,
    Object[] blockCallArguments) {
    Object r = null;
    try {
        // I know that ONE and only ONE method is defined.
        Method m = aClass.getDeclaredMethod("blockValue", blockMethodArguments);

        // I can now invoke the method defined into the new classe with the rigth parameter.
        r = m.invoke(aClass.newInstance(), blockCallArguments);
    }
    catch (Exception e) {
        System.err.println("Block.value(Class[], Object[])");
        for (int i = 0; i < blockMethodArguments.length; i++) {
            System.err.print('\t');
            System.err.println(blockMethodArguments[i]);
        }
        System.err.println(this.aClass);
        final Method[] declardMethods = this.aClass.getDeclaredMethods();
        for (int i = 0; i < declardMethods.length; i++) {
            System.err.print('\t');
            System.err.println(declardMethods[i]);
        }
        e.printStackTrace();
    }
    if (r == null)
        return Void.TYPE;
    return r;
}
public Object value(Object[] o) {
    try {
        if (o.length > 0) {
            return this.value(o[0].getClass(), o);
        }
        return this.value();
    }
    catch (Exception e) {
        System.err.println("Block.value(Object[])");
        e.printStackTrace();
        return null;
    }
```

```
}
public Object value(byte b) {
    try {
        return value(Byte.TYPE, new Byte(b));
    }
    catch (Exception e) {
        System.err.println("Block.value(byte)");
        e.printStackTrace();
        return null;
    }
}
public Object value(char c) {
    try {
        return value(Character.TYPE, new Character(c));
    }
    catch (Exception e) {
        System.err.println("Block.value(char)");
        e.printStackTrace();
        return null;
    }
}
public Object value(double d) {
    try {
        return value(Double.TYPE, new Double(d));
    }
    catch (Exception e) {
        System.err.println("Block.value(double)");
        e.printStackTrace();
        return null;
    }
}
public Object value(float f) {
    try {
        return value(Float.TYPE, new Float(f));
    }
    catch (Exception e) {
        System.err.println("Block.value(float)");
        e.printStackTrace();
        return null;
    }
}
public Object value(int i) {
    try {
        return value(Integer.TYPE, new Integer(i));
    }
    catch (Exception e) {
        System.err.println("Block.value(int)");
        e.printStackTrace();
        return null;
    }
}
public Object value(long l) {
    try {
        return value(Long.TYPE, new Long(l));
    }
    catch (Exception e) {
        System.err.println("Block.value(long)");
```

```
            e.printStackTrace();
            return null;
        }
    }
    private Object value(Class blockMethodArgument, Object[] blockCallArguments) {
        Class[] blockMethodArguments = new Class[blockCallArguments.length];
        for (int i = 0; i < blockMethodArguments.length; i++) {                    210
            blockMethodArguments[i] = blockMethodArgument;
        }
        return value(blockMethodArguments, blockCallArguments);
    }
    private Object value(Class blockMethodArgument, Object blockCallArgument) {
        return value(blockMethodArgument, new Object[] { blockCallArgument });
    }
    public Object value(Object o) {
        try {
            return value(Class.forName("java.lang.Object"), o);                    220
        }
        catch (Exception e) {
            System.err.println("Block.value(Object)");
            e.printStackTrace();
            return null;
        }
    }
    public Object value(boolean b) {
        try {
            return value(Boolean.TYPE, new Boolean(b));                            230
        }
        catch (Exception e) {
            System.err.println("Block.value(boolean)");
            e.printStackTrace();
            return null;
        }
    }
}
```

G HotSwap Source Code

```
package emn.course.vm.hotswap;

import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.lang.reflect.Array;
import java.util.HashMap;
import java.util.Iterator;                                                    10
import java.util.Map;

import javax.swing.JButton;
import javax.swing.JFrame;

import com.sun.jdi.Bootstrap;
import com.sun.jdi.InvalidStackFrameException;
import com.sun.jdi.ReferenceType;
import com.sun.jdi.ThreadReference;
import com.sun.jdi.VirtualMachine;                                            20
import com.sun.jdi.VirtualMachineManager;
import com.sun.jdi.connect.LaunchingConnector;
import com.sun.jdi.connect.Connector.Argument;

public final class GCDRedefiner {
    public static void main(String[] args) {
        try {
            final VirtualMachineManager vmManager = Bootstrap.virtualMachineManager();
            final LaunchingConnector launchingConnector = vmManager.defaultConnector();
            final Map arguments = launchingConnector.defaultArguments();      30
            final Iterator iterator = arguments.values().iterator();
            while (iterator.hasNext()) {
                final Argument argument = (Argument) iterator.next();
                if (argument.name().equals("main")) {
                    argument.setValue("emn.course.vm.GCD");
                }
            }
            final VirtualMachine vm = launchingConnector.launch(arguments);
            vm.resume();
                                                                              40
            final JFrame frame = new JFrame("Euclid's GCD algorithm redefiner");
            frame.setLocation(100, 100);
            frame.setSize(300, 100);
            frame.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
            });
            frame.getContentPane().setLayout(new BorderLayout());
                                                                              50
            final JButton suspendResumeButton = new JButton("Suspend");
            suspendResumeButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    JButton button = (JButton) e.getSource();
```

```
            if (button.getText().equals("Suspend")) {
                button.setText("Resume");
                vm.suspend();
            }
            else {
                button.setText("Suspend");                                    60
                vm.resume();
            }
        }
    });
    frame.getContentPane().add(suspendResumeButton, BorderLayout.WEST);

    final JButton stopButton = new JButton("Stop");
    stopButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            vm.exit(0);                                                       70
        }
    });
    frame.getContentPane().add(stopButton, BorderLayout.CENTER);

    final JButton redefineGCDButton = new JButton("Redefine GCD");
    redefineGCDButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            final Iterator allClassesIterator = vm.allClasses().iterator();
            ReferenceType gcdReferenceType = null;
            while (allClassesIterator.hasNext()                               80
                && (gcdReferenceType == null
                    || !gcdReferenceType.name().equals("emn.course.vm.GCDAlgorithm"))) {

                gcdReferenceType = (ReferenceType) allClassesIterator.next();
            }

            Iterator allThreadsIterator = vm.allThreads().iterator();
            ThreadReference gcdThreadReference = null;
            while (allThreadsIterator.hasNext()
                && (gcdThreadReference == null || !gcdThreadReference.name().equals("main"))) {  90

                gcdThreadReference = (ThreadReference) allThreadsIterator.next();
            }

            try {
                HashMap hashMap = new HashMap();
                hashMap.put(gcdReferenceType, emn.course.vm.GCDBytes.Array);
                vm.redefineClasses(hashMap);
            }
            catch (Exception exception) {                                     100
                exception.printStackTrace();
            }
        }
    });
    frame.getContentPane().add(redefineGCDButton, BorderLayout.EAST);

    frame.setVisible(true);
}
catch (Exception e) {
    e.printStackTrace();                                                      110
}
```

```
    }
}
```

```
package emn.course.vm;

import java.awt.Frame;
import java.awt.TextField;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public final class GCD {
    public static void main(String[] args) {
        final Frame frame = new Frame("Euclid's GCD algorithm");
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }

        });
        frame.setLocation(400, 100);
        frame.setSize(300, 100);
        final TextField text = new TextField("Results");
        frame.add(text);
        frame.setVisible(true);

        int a = 6;
        int b = 18;
        final StringBuffer buffer = new StringBuffer();
        while (true) {
            buffer.setLength(0);
            buffer.append("\n\rGCD(");
            buffer.append(a);
            buffer.append(", ");
            buffer.append(b);
            buffer.append(") = ");
            buffer.append(new GCDAlgorithm().computeGCD(a, b));
            text.setText(buffer.toString());
            a = a + 1;
            b = b + 3;
        }
    }
}
class GCDAlgorithm {
    public int computeGCD(int a, int b) {
        if (b == 0) {
            return a;
        }
        else {
            return computeGCD(b, a % b);
        }
    }
}
```
