Java Reflection

Object Oriented Programming
Java Reflection

“Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution.” [Demers and Malenfant]

- A Class object is associated for every loaded class by the JVM.
- The Class object reflects the class it represents
  - The primitive Java types are also represented as class objects
Class Object

- Instances of class `Class` stores information about classes:
  - Class name
  - Inheritance
  - Interfaces implemented
  - Methods and fields

- Enables method invocation and field referencing by name.
Accessing the class object

• A class object may be accessed in the following ways:

```java
Class<?> c = "OOP".getClass();
c = String.class;
c = Class.forName("java.lang.String");
```

• Accessing the super-class:

```java
Class<?> superClass = c.getSuperClass();
String name = superClass.getName();
```

• The class object enables referencing fields and methods, as well as querying for implemented interfaces:

```java
Field[] fields = c.getFields();
Method[] methods = c.getMethods();
Class[] interfaces = c.getInterfaces();
```
How many layers in the Java object model?

```java
static void traverse(Object o)
{
    for (int n = 0; ; o = o.getClass()) {
        System.out.println("Level " + ++n + ": " + o + ".getClass() = " + o.getClass());
        if (o == o.getClass())
            break;
    }
}
```

traverse(42):

```
> Level 1: 42.getClass() = class java.lang.Integer
> Level 2: class java.lang.Integer.getClass() = class java.lang.Class
> Level 3: class java.lang.Class.getClass() = class java.lang.Class
```
public static void showType(String className) throws ClassNotFoundException {
    Class<?> thisClass = Class.forName(className);
    String flavor = thisClass.isInterface() ? "interface" : "class";
    System.out.println(flavor + " " + className);

    // print the superclass.
    Class<?> parent = thisClass.getSuperclass();
    if (parent != null) {
        System.out.println("extends " + parent.getName());
    }

    // print all the interfaces implemented.
    Class<?>[] interfaces = thisClass.getInterfaces();
    for (int i = 0; i < interfaces.length; ++i) {
        System.out.println("implements " + interfaces[i].getName());
    }
}
showType("java.lang.Object");
> class java.lang.Object

showType("java.util.HashMap");
> class java.util.HashMap
   extends java.util.AbstractMap
   implements java.util.Map
   implements java.lang.Cloneable
   implements java.io.Serializable

package "mine";
public class Point {}

showType("mine.Point");
> class mine.Point
   extends java.lang.Object
static void showMethods(Object o) {
    Class<?> c = o.getClass();
    Method[] theMethods = c.getMethods();

    for (Method m : theMethods) {
        String methodString = m.getName();
        System.out.println("Name: " + methodString);
        System.out.println(" Return Type: " +
                           m.getReturnType().getName());
        Class<?>[] parameterTypes = m.getParameterTypes();
        System.out.print(" Parameter Types:");
        for (Class<?> p : parameterTypes) {
            System.out.print(" " + p.getName());
        }
        System.out.println();
    }
}

Public only

What is returned?
Outputs...

Input:

```java
Object o = new Object();
showMethods(o);
```

Output:

- Name: `hashCode`
  - Return Type: `int`
  - Parameter Types:

- Name: `getClass`
  - Return Type: `java.lang.Class`
  - Parameter Types:

- Name: `equals`
  - Return Type: `boolean`
  - Parameter Types: `java.lang.Object`
Main Java Reflection Classes

- **Class** (*java.lang.class*)
  - Instances of the class `Class` represent classes and interfaces in a running Java application, every object is represented by a `Class` object.

- **Package java.lang.reflect**
  - Member (*java.lang.reflect.Member*)
    - An Interface that reflects identifying information about a single member (a field or a method) or a constructor.
  - Method (*java.lang.reflect.Method*)
    - Implements Member Interface
    - Provides information about, and access to, a single method on a class or interface.
    - Represents instance methods and class methods (static).
Main Java Reflection Classes (cont).

- **Field** *(java.lang.reflect.Field)*
  - Implements Member interface
  - Provides information about, and dynamic access to, a single field (also for static fields)
  - Provides access and modification (set, get) methods.

- **Constructor** *(java.lang.reflect.Constructor)*
  - Implements Member interface
  - Provides information about, and access to, a single constructor for a class.

- **Package** *(java.lang.Package)*
  - Package objects contain version information about the implementation and specification of a Java package
  - Package name, creator name, implementation version...
Main Reflection Classes (cont.)

Example:

```java
static void packageExploring(String name) {
    Package p = Package.getPackage(name);
    if (p == null)
        return;
    System.out.format("%1$s is %2$compatible with %3$s", name,
                        (p.isCompatibleWith("1.6") ? "" : "not ") , "1.6");
}
```

Modifiers – decodes member access modifiers.

```java
Member m;
//initializing m...
int mod = m.getModifiers();
if (Modifier.isAbstract(mod))
    System.out.println("abstract");
if (Modifier.isFinal(mod))
    System.out.println("final");
if (Modifier.isPublic(mod)) ...
```
Arrays and Reflection

- Reflection can be used to create and manipulate arrays whose size and component type are not known until runtime.

```java
public static void testArray() {
    Class<?> cls = String.class;
    int i = 10;
    Object arr = Array.newInstance(cls, i);
    // arr now points on a String[10]
    Array.set(arr, 5, "this is a test");
    String s = (String)Array.get(arr, 5);
    System.out.println(s);
    System.out.println(arr);
}
```
A Java Reflection Example

- Illustrates Four Issues:
  - Runtime information
  - Introspection
  - Invoking Method Objects
  - Dynamic Instantiation
The Employee Example

• Reminder...

Employee

number
level

print(): void

MonthlyEmployee

print(): void

HourlyEmployee

print(): void
Reflection and Dynamic Binding

• The binding to `getClass()` is dynamic:

```java
Employee me, he;
me = new MonthlyEmployee();
Class<?> c = me.getClass();
System.out.println("class of me = " + c.getName());
he = new HourlyEmployee();
c = he.getClass();
System.out.println("class of he = " + c.getName());
```

Output:

Class of me = MonthlyEmployee
Class of he = HourlyEmployee
• `getFields()` returns all public fields including inherited ones.
  – `getDeclaredFields()` returns all fields declared by the class, but excludes inherited ones.

• Setting and getting a field:

```java
Field f = c.getField("level");
f.setInt(e, f.getInt(e) + 1);
```

```java
MonthlyEmployee e = new MonthlyEmployee();
Field fields[] = e.getClass().getFields();
for (Field f : fields) {
    System.out.println(f.getName() +"="+f.getInt(e));
}
```
We can ask a method object to invoke the method it represents.

Implicit and explicit arguments must be provided.

```java
Employee e = new HourlyEmployee();
Class<?> c = e.getClass();
Method m = c.getMethod("print", PrintStream.class);
m.invoke(e, System.out);
```

Output:
- I’m a Hourly Employee
Dynamic Instantiation

- The universal printer gets the employee type and invokes the print method.

```java
class UniversalPrinter {
    public void print(String empType) {
        Class<?> c = Class.forName(empType);
        Employee emp = (Employee)c.newInstance();
        emp.print(System.out);
    }
}
```

- Instantiating objects by calling non-default constructors

```java
Constructor<?> c = Class.forName(empType).getConstructor(...)
Object[] initArgs = ...
Object o = c.newInstance(initArgs);
```
static void mutate(String s) {
    Field value = String.class.getDeclaredField("value");
    value.setAccessible(true);
    value.set(s, "Goodbye").toCharArray());
}

Usage:

String s = "Hello!";
mute(s); // s = "Goodbye!"
Reflection in Java – what is missing?

• Reflection is introspection only (unlike squeak and C#)
  – Can’t add / modify fields (structure), methods (behavior)

• Implementation is not available
  – Program logic is not reflected

• Major performance impact
  – Much slower then doing the same operations directly...

• Complex code
Java Serialization

• The process of converting objects into a linear stream of bytes.
  – Depends on reflection
  – Serialization includes object state as well as information about the object’s type and the member types.
  – JVM independent

```java
public class PersistentTime implements Serializable {
    private Date time;

    public PersistentTime() {
        time = Calendar.getInstance().getTime();
    }

    public Date getTime() { return time; }
}
```
• Serializing PersistenceTime object:

```java
PersistentTime time = new PersistentTime();

FileOutputStream fos =
    new FileOutputStream("time.ser");
ObjectOutputStream out =
    new ObjectOutputStream(fos);
out.writeObject(time);
out.close();
```

• `writeObject` does the actual serialization