Java Reflection: Dynamic Proxy.

Design Patterns defines a proxy as an object that provides a surrogate or placeholder for another object to control access to it.

- local representation of remote objects;
- delay of expensive operations;
- access protection for secure objects.
**Java’s Dynamic Proxy.**

**What Is a Dynamic Proxy?**

A dynamic proxy class is a class that implements a list of interfaces specified at run-time such that a method invocation through one of the interfaces on an instance of the class will be encoded and dispatched to another object through a uniform interface.

Thus a dynamic proxy class can be used to create a type-safe proxy object for a list of interfaces without requiring pre-generation of the proxy class, such as with compile-time tools.

Dynamic proxy classes are useful to an application or library that needs to provide type-safe reflective dispatch of invocations on objects that present interface APIs.

**Java’s Dynamic Proxy.**

**java.lang.reflect.InvocationHandler.**

Each proxy instance has an associated invocation handler.

When a method is invoked on a proxy instance, the method invocation is encoded and dispatched to the `invoke()` method of its invocation handler:

```
Object invoke(Object proxy, Method m, Object[] args)
```

**Java’s Dynamic Proxy.**

**java.lang.reflect.Proxy.**

Proxy provides static methods for creating dynamic proxy classes and instances, and it is also the superclass of all dynamic proxy classes created by those methods.

- `Proxy(InvocationHandler ih)`
- `static Class getProxyClass(ClassLoader cl, Class<?>... interfaces)`
  The class is dynamically built using the given class loader and the given array of interfaces.
- `static boolean isProxyClass(Class<?> myProxyClass)`
- `static Object newProxyInstance(ClassLoader cl, Class<?>[] interfaces, InvocationHandler ih)`

Basically, `Proxy` permits of implementing the meta-object approach in Java.

**Java’s Dynamic Proxy.**

**How Java Creates a Proxy.**
Java’s Dynamic Proxy.
Example: to Trace the Method Calls.

```java
class TraceHandler implements InvocationHandler {
    private Object baseObject;
    private Object result = null;
    private TraceHandler(baseObject) { baseObject = baseObject; }
    private Object invoke(Object proxy, Method m, Object[] args) {
        try {
            System.out.println("before " + m.getName());
            result = m.invoke(baseObject, args);
            System.out.println("after " + m.getName());
            return result;
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

```java
IPoint p = new Point(10, 20);
InvocationHandler th = new TraceHandler(p);
IPoint th_p = (IPoint) Proxy.newProxyInstance(
    P.class.getClassLoader(), P.class.getInterfaces(), th);
p.getX(); /* standard call */ th_p.getX(); /* traced call */
```

Java’s Dynamic Proxy.
Example: Invariant Checking and Proxy Chaining.

```java
class InvariantHandler implements InvocationHandler {
    private Object target; private Method method; 
    public InvariantHandler(Object target) {
        this.target = target;
        try {
            method = target.getClass().getMethod("invariant", new Class[]{ });
            if (!method.getReturnType().equals(boolean.class)) method = null;
        } catch (NoSuchMethodException ex) { method = null; }
    }
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable{
        this.invokeInvariant(method);
        Object retvalue = method.invoke(target, args);
        this.invokeInvariant(method);
        return retvalue;
    }
    private void invokeInvariant(Method method) {
        if ((invariant == null) || (method.equals(invariant)) return;
        try {
            Boolean passed = (Boolean) invariant.invoke(target, new Object[] {});
            if (!passed.booleanValue()) throw new RuntimeException();
        } catch (Exception e) { /* handling goes here */ }
    }
}
```

```java
// create an invariant handler and a proxy for a point
InvariantHandler ih = new InvariantHandler(new Point(0, 7));
Point invariantCheckedPoint = (Point) Proxy.newProxyInstance(
    Point.class.getClassLoader(), new Class[]{ IPoint.class },
    ih);

// create a trace handler and a proxy for an InvariantCheckedPoint
TraceHandler th = new TraceHandler(invariantCheckedPoint);
Point tracedInvariantCheckedPoint = (Point) Proxy.newProxyInstance(
    Point.class.getClassLoader(), new Class[]{ IPoint.class },
    th);
```
Java Reflection: Class Loading.

Class Loading

Defined by the abstract class `ClassLoader` and its subclasses such as `NetworkClassLoader`.

A class loader is an object that is responsible for loading classes.
- Given the name of a class, it should attempt to locate or generate data (array of bytes) that constitutes a definition for the class.

Class loaders define namespaces.
- Every Class object contains a `Class#getClassLoader()` reference to the `ClassLoader` that defined it.
- `<Class A loaded by CL1> = <Class A loaded by CL2>` if and only if `CL1 = CL2`.

Java Class Loader.

Slide 13 of 26

Walter Cazzola

Dynamic Proxy and Class Loading

Slide 14 of 26

Walter Cazzola

Java Class Loader.

About the Java Class Loader.

Class Loaders and Namespaces.

A class loader is an object that is responsible for loading classes.
- Given the name of a class, it should attempt to locate or generate data (array of bytes) that constitutes a definition for the class.

Class loaders define namespaces.
- Every Class object contains a `Class#getClassLoader()` reference to the `ClassLoader` that defined it.
- `<Class A loaded by CL1> = <Class A loaded by CL2>` if and only if `CL1 = CL2`.

Java Class Loader.

Example of a Class Loaded by Two Different Class Loaders.

```java
public class ConstructOnce {
    static private boolean runOnce = false;
    public ConstructOnce() {
        if (runOnce) throw new IllegalStateException("run twice");
        runOnce = true;
    }
}

public class SimpleClassLoaderTest {
    public static void main(String[] args) {
        SimpleClassLoader CL1 = new SimpleClassLoader("testclasses");
        Class c1 = CL1.loadClass("ConstructOnce");
        SimpleClassLoader CL2 = new SimpleClassLoader("testclasses");
        Class c2 = CL2.loadClass("ConstructOnce");
        Object x = c1.newInstance();
        try {
            Object y = c1.newInstance();
            throw new RuntimeException("Test Fails!!!");
        } catch (IllegalStateException e) {
            Object z = c2.newInstance();
        }
    }
}
```
**Java Class Loader.**

Why Subclass the Class Loader?

Applications implement subclasses of `ClassLoader` in order
- to extend the manner in which the Java virtual machine dynamically loads classes;
- to obtain different protected name spaces; and
- to transform bytecode!

Some applications related to the class loaders are:
- Security
  - examine classes for a proper digital signature
  - disallow the loading of particular packages (for example, `java.io`)
- Encryption
  - decrypt classes on the fly, so that your class files on disk are not readable by someone with a decompiler
- Archiving
  - distribute your code in a special format or with special compression
- Testing
  - keep test cases separated by using different loaders

---

```java
public class ClassLoader {
    protected ClassLoader() {...}
    protected ClassLoader(ClassLoader parent) {...}
    public final ClassLoader getParent() {...}
    public static ClassLoader getSystemClassLoader() {...}
    protected Class<?> findClass(String name) throws ClassNotFoundException {...}
    public Class<?> loadClass(String name) throws ClassNotFoundException {...}
    protected Class<?> defineClass(String name, byte[] b, int off, int len) throws ClassFormatError {...}
    protected Class<?> findLoadedClass(String name) {...}
    protected Class<?> findSystemClass(String name) throws ClassNotFoundException {...}
    ...
}
```

---

**Java Class Loader.**

Another Reason to Create Your Own Class Loader.

**Problem:**
Write a method that outputs the inheritance hierarchy of an application.

**Subproblem:**
How do you discover what classes belong to your application?

**Solution:**
Impossible — The subproblem is extrinsic to Java meta-object protocol.

So what can we do?

*Create our own class loader.*

The class loader object knows what classes have been loaded.
Java Class Loader.
Class Loading: How It Works

loadClass() is where the loading process starts.
- it invokes the findLoadedClass() to check if the class has been already loaded;
- if yes it returns the found class otherwise it delegates the work to its parent: Delegation Model.
- if everything fails, it invokes the findClass() to find the class, read the bytecode and create the class object (defineClass()).

Something like:

```java
public Class loadClass(String name) throws ClassNotFoundException {
    Class c = findLoadedClass(name);
    if (c != null) return c;
    try {
        return parent.loadClass(name);
    } catch (ClassNotFoundException e) {
        return findClass(name);
    }
}
```

Walter Cazzola

Java Class Loader.
Class Loading: How It Works (Cont’d)

How to subclass the class ClassLoader
- creates both a no-parameter constructor and one that supports the delegation model;
- override findClass();
- absolutely do not override loadClass() because its original implementation supports the delegation model.

Suppose you would like to dynamically add to the directories on the classpath so that new files can be loaded.

```java
public class SimpleClassLoader extends ClassLoader {
    String[] directories;
    public SimpleClassLoader(String path) { directories = path.split( ";" ); }
    public SimpleClassLoader(String path, ClassLoader parent) {
        super(parent);
        directories = path.split(";");
    }
    public synchronized Class findClass(String name) throws ClassNotFoundException {
        for (int i = 0; i < directories.length; i++) {
            byte[] buf = getClassData( directories[i], name );
            if (buf != null) return defineClass(name,buf,0,buf.length);
        }
        throw new ClassNotFoundException();
    }
    protected byte[] getClassData( String directory, String fileName ) { ... }
}
```

Instead of generating Java work with bytecodes
```
defineClass(String name, byte[] b, int off, int len)
```
The subarray from byte[off] to byte[off+len-1] contains the bytecodes for the class.

You can:
- Create the bytecodes for a new class;
- Modify the bytecodes of an existent class.

This is beyond the scope of this lesson, but we will see it again in the follow (see Javassist and BCEL).
Three essential facts about class loaders

1. at run-time, a class object is identified by the combination of the class name and the class loader object that created the class object

   \texttt{ClassLoader.defineClass()}

   creates class objects.

   The loader object that executed \texttt{defineClass()} for a class object is its defining loader;

2. all other classes referenced by a class are loaded by its defining loader;

3. class loaders delegate to other class loaders; especially the system class loader, which interprets the class path.

### Referents

- SUN Microsystems.  
  \textit{Java™ Core Reflection API and Specification}.  

- Ira R. Forman and Nate B. Forman.  
  \textit{Java Reflection in Action}.  

- Sheng Liang and Gilad Bracha.  
  \textit{Dynamic Class Loading in the Java Virtual Machine}.  

- Li Gong.  
  \textit{Secure Java Class Loading}.  