

Object – Oriented Design with UML and Java

Part XIV: Java Reflection

Reflection

- Also known as introspection.
- Enables a running program to ask a class for details about its
 - Attributes / Methods / Constructors / Annotations (JDK 1.5)
- Enables a running program to load a new class, create new instances of that class, and invoke methods on the fly, without knowing anything about the class at compile-time.
- Is essential to the operation of many advanced Java features
 - Javadoc
 - Enterprise Java Beans (EJB)
 - JUnit framework
 - Configurable application frameworks with “pluggable” components
 - Aspect Oriented Programming (AOP) using annotations
- Java’s reflection is similar to RTTI (Run-Time Type Identification) in C++, only better.

Reflection (cont.)

How can a program find out about a class?

- Three ways to get a class object:

```
Class c1 = Class.forName( "myPackage.MyClass" );  
Class c2 = someObject.getClass();  
Class c3 = myPackage.MyClass.class;
```

Then...

- Ask the Class object for its Methods, Fields, and Constructors
- Ask the Fields about their types
- Ask the Methods about their names, parameters, exceptions, and return types
- Ask any of the above about their annotations

Reflection (cont.)

- Instances of the class `Class` represent classes and interfaces in such a way that they can be manipulated at run time. The Java Virtual Machine automatically constructs `Class` objects as classes are dynamically *loaded* (as needed).
- Java's class loader (`java.lang.ClassLoader`) eliminates the need for a link step (as required in C++). This is a powerful feature.

```
String myClassName = "myPackage.MyClass";
try {
    Class class = Class.forName( myClassName );
    Object o = class.newInstance();
    MyClass mc = (MyClass) o;
    . . .
}
catch( Exception ex ) . . .
```

java.lang.Object

- The class **Object** is the root class for all other classes. If a class specifies no superclass, it extends **Object** by default.
- Therefore, every class inherits these methods from *class Object*:

```
public final Class getClass();
public String      toString();
public boolean     equals( Object o );
public int         hashCode();
protected Object  clone() throws CloneNotSupportedException;
protected void    finalize();
// Thread related stuff:
public final void  wait() throws InterruptedException;
public final void  notify();
public final void  notifyAll();
```

java.lang.Object (cont.)

```
public String      toString(); // comes in handy
public boolean     equals( Object o ); // OVERRIDE ME
public int         hashCode(); // OVERRIDE ME
protected Object  clone() // don't use
protected void     finalize(); // don't use
```

- The methods `hashCode()` and `equals()` go together. The designers of Java anticipated the ubiquitous use of `Hashtable` and `HashMap`. These data structures rely on value-based equality for the key (as opposed to memory-address equality). Thus it is essential to override `equals()` for any class that might be used as a hashing key. The `hashCode()` method is purely for performance.
- Refer to Chapter VII – Collection Classes

java.lang.Class

What can a program do once it knows about a class?

- Perform on-the-fly maintenance without bringing down the system – It is possible to load new classes into a running system and use them. Of course this must be a well-designed feature of the system.
- Perform framework functions using new classes (e.g., GUI builders and JUnit) – Especially since the advent of annotations in JDK 1.5.
- Generate adapter and proxy classes (e.g., Enterprise Java Beans).
- Other clever stuff.
- There is one instance of class **Class** per class per **ClassLoader** .

java.lang.Class (cont.)

Here are some of the **public** methods from *class Class* (pre-JDK 5):

- Note: *no constructor*.
- Many of these methods might throw an Exception

```
static Class   forName( String className ) // Variation of Singleton
String        getName () ;
boolean       isInterface () ;
Object        newInstance () ;
Class         getSuperclass () ;
Class[]       getInterfaces () ;
ClassLoader  getClassLoader () ;
Constructor[] getDeclaredConstructors () ;
Method[]     getDeclaredMethods () ;
Method       getDeclaredMethod( String name, Class[] paramTypes ) ;
Field[]      getFields () ;
```

java.lang.reflect.Method

```
public final class Method . . . // Pre JDK 1.5
{
    public boolean equals( Object o );
    public Class    getDeclaringClass();
    public Class[]  getExceptionTypes();
    public int      getModifiers();
    public String   getName();
    public Class[]  getParameterTypes();
    public Class    getReturnType();
    public Object   invoke( Object o, Object[] args ) throws . . .;
    . . .
}
```

// To see if a method is static, use:

```
java.lang.reflect.Modifier.isStatic( m.getModifiers() );
```

Invoke a Method

- Given information (as *metadata*) about a class & a method (that takes no arguments and returns nothing) create an instance of the class and invoke the method.

```
try {
    String className = getClassNameFromMetaDataFile();
    String methodName = getMethodNameFromMetaDataFile();
    Class c = Class.forName( className ); // Load the class
    Object o = c.newInstance(); // Instantiate an object
    Method m = c.getDeclaredMethod( methodName, new Class[0] );

    m.invoke( o, new Object[0] ); // Invoke the method
}
catch( Throwable t ) { ...; }
```

Invoke a Method (cont.)

- Example: use reflection to invoke class Foo's method: `bar(FooBar fb) : Barf`
- To invoke a static method, use the class object as the invocation target.

```
try {
    Class    fooClass = Class.forName( "myPackage.Foo" );
    Class    fooBarClass = myPackage.FooBar.class;
    Object    targetObj = fooClass.newInstance();
    Object    fooBarArg = new FooBar( 46 );
    Class[]   barParamTypes = new Class[] { fooBarClass };
    Method    bar = fooClass.getDeclaredMethod( "bar", barParamTypes );
    Object[]  barArgs = new Object[] { fooBarArg };
    Barf      barReturnObj = (Barf) bar.invoke( targetObj, barArgs );
}
catch( Throwable t ) { ...; }
```

Inspect a class at run time

- Get self-descriptive “introspective” information from any object

```
class Reflect {
    public static void main( String[] argv ) {
        String foo = "foo"; // new String( "foo" );
        Object o = foo;
        Class c = o.getClass();
        System.out.println( "The class " + c.getName() + " has " +
            c.getDeclaredConstructors().length + " constructors." );
    }
}
```

- The program outputs:

```
The class java.lang.String has 13 constructors.
```

Reflection & Encapsulation

- A reflection-based “object parser” must obey the attribute visibility rules of Java. Therefore, it can never look at the private attributes of other classes. It can look at protected attributes of a class only if it is in the same package. Any attempts to circumvent this restriction will result in an `IllegalAccessException`.
- There is an advanced work around to this, however, using `java.lang.reflect.AccessibleObject`, the superclass to `Field`, `Method`, and `Constructor`. This class has a `setAccessible(boolean)` method, where a value of `true` indicates that the reflected object should suppress Java language access checking when it is referenced via the reflection API.

Static Polymorphism

// Here is a work around to the lack of static polymorphism in Java.

```
import java.lang.reflect.Method;

public class StaticPoly {
    public static void main( String[] args ) {
        StaticPoly staticPoly = new StaticPoly();
        StaticPoly subStaticPoly = new SubStaticPoly();
        System.out.println( "A=" + staticPoly.foo() );
        System.out.println( "B=" + subStaticPoly.foo() );
        System.out.println( "C=" + invokeFoo( staticPoly ) );
        System.out.println( "D=" + invokeFoo( subStaticPoly ) );
    }

    public static int foo() {
        return 1;
    }
}
```

Static Polymorphism (cont.)

```
public static int invokeFoo( Object o ) {
    try {
        Class c = o.getClass();
        Method m = c.getDeclaredMethod( "foo",
                                         new Class[] { } );
        Object returnVal = m.invoke( o, new Object[] { } );
        Integer returnInt = (Integer) returnVal;
        return returnInt; // uses "auto-boxing"
    }
    catch( Throwable t ) {
        t.printStackTrace();
    }
    return -1;
} }
```

Static Polymorphism (cont.)

```
class SubStaticPoly extends StaticPoly {  
    public static int foo() {  
        return 2;  
    }  
}
```

// The program outputs:

A=1

B=1

C=1

D=2

Other useful reflection operations

instanceof

- A Java *operator* that tests if an object is an instance of a class. The class must be specified at compile time.

```
if( o instanceof java.lang.reflect.Method ) ...
```

isAssignableFrom(Class c)

- From class Class. Determine if a class is a subclass of c or implements interface c.

isInstance(Object obj)

- From class Class. Determine if an object is an instance of a class.

JUnit – Testing Framework

- JUnit is a popular unit *testing* framework, available from:
<http://www.junit.org/>
 - Before JDK 1.5 came along with annotations, JUnit Test classes needed to extend **TestCase**, and test methods relied on the following naming conventions:
 - `void setUp()` // sets up data for tests
 - `void testFoo()` // testXXX methods run tests
 - `void tearDown()` // restores environment
- Tests call: `assert(<boolean expression>);`

JUnit (cont.)

- With JDK 1.5 annotations, this got a lot easier!

```
@Test // Annotate the method as a test method
```

```
public void aMethodWithoutTestInItsName() {  
    assertEquals(-1, myFunction( 2, 3 ));  
}
```

```
@BeforeClass // Invoke once BEFORE all test methods
```

```
public void noNamingConvention() {  
    doMeFirstToSetUpTheTestSuite();  
}
```

```
@After // Invoke AFTER every test method
```

```
public void annotationsAreGood() {  
    doMeToCleanUpAfterEachTest();  
}
```

JUnit (cont.)

More JUnit annotation examples...

- The JUnit framework will use reflection to find these test methods and invoke them, auto-magically.
- Learn JUnit. Use JUnit. Test your code!

```
@Test(expected = ArithmeticException.class)
public void divideByZero() {
    int infinity = 1.0 / 0.0;
}

@Test(timeout = 1000) // wait one second
public void infiniteLoop() {
    while (true) {}
}
```