Reflection Example—The Java Reflection API

- For every loaded class, the Java Runtime Environment (JRE) maintains an associated Class object
  - The Class object “reflects” the class it represents
  - Can use the Class object to discover information about a loaded class
    - name
    - modifiers (public, abstract, final)
    - superclasses
    - implemented interfaces
    - fields
    - methods
    - constructors
  - Can instantiate classes and invoke their methods via Class object

How the Java Reflection API works:

- Accessing the Class object for a loaded class:

  To get the Class object for an object mystery:
  
  ```java
  Class c = mystery.getClass();
  ```

  Or, using the class name:
  
  ```java
  Class c = Class.forName("mysteryClass");
  ```

  Can also get the superclass of MysteryClass:
  
  ```java
  Class s = c.getSuperclass();
  ```
Java Reflection--Continued

Introspecting (examining) a class via its `Class` object:

Getting the class name:
```java
Class c = mysteryObject.getClass();
String s = c.getName();
```

Discovering the interfaces implemented by a class:
```java
Class[] interfaces = c.getInterfaces();
```

Discovering the fields of a class:
```java
Field[] fields = c.getFields();
```

Discovering the methods of a class:
```java
Method[] methods = c.getMethods();
```

Example Code:
```java
static void showMethods(Object o) {
    Class c = o.getClass();
    Method[] theMethods = c.getMethods();
    for (int i = 0; i < theMethods.length; i++) {
        String methodString = theMethods[i].getName();
        System.out.println("Name: " + methodString);
        String returnString = theMethods[i].getReturnType().getName();
        System.out.println(" Return Type: " + returnString);
        Class[] parameterTypes = theMethods[i].getParameterTypes();
        System.out.println(" Parameter Types: ");
        for (int k = 0; k < parameterTypes.length; k++) {
            String parameterString = parameterTypes[k].getName();
            System.out.print(" " + parameterString);
        }
        System.out.println();
    }
}
```
Example--Continued

Output for a call: of the form:
Polygon P = new Polygon();
showMethods(p);

Name: equals
  Return Type: boolean
  Parameter Types: java.lang.Object
Name: getClass
  Return Type: java.lang.Class
  Parameter Types:
Name: intersects
  Return Type: boolean
  Parameter Types: double double double double . . .

Additional Features of Java Reflection

• Can obtain constructors for a class
• Can instantiate objects and invoke methods via information obtained from the reflection API.
A Java Reflection Example

Illustrates Four Issues:

1) Runtime type Information (RTTI)
2) Introspection
3) Invoking Method Objects
4) Dynamic Instantiation

Java RTTI Example
HornNote and ViolinNote are subclasses of Note that override the inherited play() method:

```java
class HornNote extends Note {
    public void play() {
        System.out.println("Playing a horn note");
    }
}
class ViolinNote extends Note {
    public void play() {
        System.out.println("Playing a violin note");
    }
}
```

Now consider the following test code:

```java
Note note;
note = new HornNote();
Class c = note.getClass();
System.out.println("class of note = \" + c.getName());
note = new ViolinNote();
c = note.getClass();
System.out.println("now class of note = \" + c.getName());
```

The output produced would be:

```
class of note = HornNote
now class of note = ViolinNote
```
JAVA RTTI Example--Continued

We could also reassign c to reference any super class of ViolinNote:

```java
c = c.getSuperclass();
System.out.println("base class of note = " + c.getName());
c = c.getSuperclass();
System.out.println("base of base class of note = " + c.getName());
```

Here is the output produced:

```
base class of note = Note
base of base class of note = java.lang.Object
```

Introspection Example

We can also find out about the methods and fields of a class. Assume that c still references an object of the ViolinNote class. Then the following loop prints out the names of all of the ViolinClass methods:

```java
Method methods[] = c.getMethods();
for(int i = 0; i < methods.length; i++)
    System.out.println(methods[i].getName());
```

Here is the output produced:

```
main hashCode wait wait wait getClass equals toString
notify notifyAll play
```

Note: we could also find out parameter lists, exception lists, return types, etc.
**Introspection example continued**

To print the names of the ViolinNote fields as well as their current values in the particular ViolinNote object referenced by note:

```java
Field fields[] = c.getFields();
try {
    for(int i = 0; i < fields.length; i++) {
        System.out.print(fields[i].getName() + " = ");
        System.out.println(fields[i].getInt(note));
    }
} catch(Exception e) {
    // handle e
}
```

Here is the output produced:

- frequency = 60
- duration = 300

Non-public fields aren't printed.

---

**Example--Invoking Method Objects**

We can ask a Method object to invoke the method it represents. (Of course we must provide it with the implicit and explicit arguments.)

For example, let's create a generic Note object, then call its play() method using reflection:

```java
note = new Note();
c = note.getClass();
Method meth = c.getMethod("play", null);
meth.invoke(note, null);
```

Here is the output produced:

Playing a generic note
Invoking Method Objects--Continued

We repeat the experiment using a HornNote:

```java
note = new HornNote();
c = note.getClass();
meth = c.getMethod("play", null);
meth.invoke(note, null);
```

Here is the output produced:

Playing a horn note

Notice that the HornNote play() method was invoked instead of the Note play() method.

JAVA Dynamic Instantiation
Example
Consider a universal instrument that can imitate all other types of instruments. This is done with a play() method that expects as its input only the name of the type of note to play:

```java
class UniversalInstrument {
    public void play(String noteType) {
        try {
            Class c = Class.forName(noteType);  // find & load a class
            Note note = (Note) c.newInstance();
            note.play();
        } catch (Exception e) {
            // handle e here
        }
    }
}
```
Dynamic Instantiation Example--
continued

After creating a universal instrument, our test driver calls the play() method twice. The first time the string "ViolinNote" is the argument. The second time the string "HornNote" is the argument:

```java
UniversalInstrument inst = new UniversalInstrument();
String noteType;
noteType = "ViolinNote";
inst.play(noteType);
noteType = "HornNote";
inst.play(noteType);
```

Here is the output produced:

- Playing a violin note
- Playing a horn note

Dynamic Instantiation Example--
Continued

Of course if we wanted to create and play a HornNote followed by a ViolinNote, why not simply do it directly:

```java
note = new HornNote();
note.play();
note = new ViolinNote();
note.play();
```

To see why, suppose instead of hardwiring the "ViolinNote" and "HornNote" strings into our test program, we allow the user to specify the strings:

```java
System.out.print("enter a type of note: ");
noteType = MyTools.stdin.readLine();
inst.play(noteType);
```

We don't know what the user will enter, so we don't know what type of notes to make.
Dynamic Class Loading

55:182/22c:182
Software Engineering Languages and Tools

public class MyClassLoader extends ClassLoader {
    public Class loadClass(String name) throws ClassNotFoundException {
        try {
            String url = "file:C:/data/projects/dcl_example/classes/" + name;
            URL myUrl = new URL(url);
            URLConnection connection = myUrl.openConnection();
            InputStream input = connection.getInputStream();
            ByteArrayOutputStream buffer = new ByteArrayOutputStream();

            int data = input.read();
            while(data != -1) {
                buffer.write(data);
            }
            input.close();
            byte[] classData = buffer.toByteArray();
            return defineClass("MyNewClass", classData, 0, classData.length);
        }
    }

    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}

Class Loader Example
Example: Using the class loader

```java
public static void main(String[] args) throws ClassNotFoundException,
    IllegalAccessException,
    InstantiationException {

    MyClassLoader classLoader = new MyClassLoader();
    Class myNewClass = classLoader.loadClass("MyNewClass");
    AnInterface object1 = (AnInterface) myNewClass.newInstance();
    ...
}

The body of the class to be loaded:

```public class MyClass implements AnInterface {
    //... body of class ... implement interface methods
}```