Core Java

Reflection API

Lesson Objectives

- In this lesson, you will learn:
  - What is Reflection?
  - Uses of Reflection
  - Drawbacks of Reflection
  - Using Reflection accessing and manipulating the following:
    - Classes
    - Fields
    - Methods
    - Constructors

10.1 : What is Reflection?

Overview of Reflection

- Reflection is the ability of a class or object to examine itself.

- The **Java Reflection API** is a framework that we can use for **introspection** of objects at run time.
- Reflection can be used to examine or modify the runtime behavior of objects running in the Java Virtual Machine.
- Reflection API comes under **java.lang.reflect** package.

Overview of Reflection (Contd…)

- Using reflection you can:
  - Determine the class of an object
  - Find out all information about a class – its access modifiers, superclass, fields, constructors, and methods
  - Find out what is there in an interface

Overview of Reflection (Contd…)

- Using reflection you can (contd.):
  - Even if you do not know the names of things when you write the program, you can:
    - Create an instance of a class.
    - Get and set instance variables.
    - Invoke a method on an object.
    - Create and manipulate arrays.

What is the use of Reflection?

- **Reflection** is used to analyze the capabilities of classes at runtime.
- In “normal” programs you do not need reflection.
- You do need reflection if you are working with programs that process programs.
  - Typical examples:
    - A class browser
    - A debugger
    - A GUI builder
    - An IDE, such as BlueJ or Forté

What are the Drawbacks of Reflection?

- Following are the drawbacks of Reflection:
  - Performance Overhead
  - Security restrictions
  - Exposure of Internals

The Class class

- The entry point for all reflection operations is **java.lang.Class**.
- For every type of object, the Java Virtual Machine instantiates an immutable instance of **java.lang.Class**.
- It provides methods to examine the runtime properties of the object including its members and type information.
- Class also provides the ability to create new classes and objects.

Various methods used to get the Class Object

- Let us discuss some methods used to retrieve class object:
  - Using Object.getClass() method
    - String s=new String("Hello");
    - Class c=s.getClass();
  - Using .class syntax – It is used if the type is available but there is no instance available.
    - String s;
    - Class c = String.class; // correct
  - Using Class.forName() – It is used if fully qualified name is available.
    - Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Various methods used to get the Class Object

- Using TYPE Field for Primitive Type Wrappers:
  - Class c = Double.TYPE;

Demo : Retrieving Class and Method Name

Demo:

SampleName.java

Process of Examining Class Modifiers & Types

- A class may be declared with one or more modifiers which affect its runtime behavior:
  - Access modifiers: public, protected, and private
  - Modifier requiring override: abstract
  - Modifier restricting to one instance: static
  - Modifier prohibiting value modification: final
  - Modifier forcing strict floating point behavior: strictfp
  - Annotations

Demo : Examining Class Modifiers & Types

Demo:

SampleModifier.java

Process of Examining Class Modifiers & Types

- A Class object has an instance method:
  - int getModifiers()
- Modifier class has got abstract methods which can be used to examine the returned value, namely:
  - isAbstract(int), isFinal(int), isInterface(int), isPrivate(int), isProtected(int), isPublic(int), isStatic(int mod), isStrict(int mod), isSynchronized(int mod), isTransient(int mod) , isVolatile(int mod)
- Usage:

```
if (Modifier.isPublic(m))
    System.out.println("public");
```

Class Methods for locating Members

Member	Class API	List of members?	Inherited members?	Private members?
Field	getDeclaredField()	N	N	Y
	getField()	N	Y	N
	getDeclaredFields()	Y	N	Y
	getFields()	Y	Y	N
Method	getDeclaredMethod()	N	N	Y
	getMethod()	N	Y	N
	getDeclaredMethods()	Y	N	Y
	getMethods()	Y	Y	N
Constructor	getDeclaredConstructor()	N	N/A	Y
	getConstructor()	N	N/A	N
	getDeclaredConstructors()	Y	N/A	Y
	getConstructors()	Y	N/A	N

Demo : Retrieving methodinfo and constructorinfo

Demo:

SampleMethod.java

ConstructorInfo.java

Member Interface

- Reflection defines an interface **java.lang.reflect.Member** which is implemented by the following:
  - java.lang.reflect.Field
  - java.lang.reflect.Method
  - java.lang.reflect.Constructor

Concept of Fields

- Fields have a type and value.
- The **java.lang.reflect.Field** class provides methods for accessing type information and setting and getting values of a field on a given object.
- Using these methods one can do the following:
  - Obtain the Field Types
  - Retrieve and Parse Field Modifiers
  - Get and Set Field Values

Fields Types

- A field may be either of **primitive** or **reference** type.
- There are eight primitive types:
  - boolean, byte, short, int, long, char, float, and double
- A reference type is anything that is a direct or indirect subclass of java.lang.Object
  - It includes interfaces, arrays, and enumerated types.

Demo : Fields Types

Demo:

FieldType.java

Field Modifiers

- There are several modifiers that may be part of a field declaration:
  - Access modifiers: public, protected, and private
  - Field-specific modifiers governing runtime behavior: transient and volatile
  - Modifier restricting to one instance: static
  - Modifier prohibiting value modification: final
  - Annotations

Field Modifiers

- The method **Field.getModifiers()** can be used to return the integer representing the set of declared modifiers for the field.

Demo : Field Modifiers

Demo:

fieldmodifier.java

Steps for Getting & Setting the Field Values

- Using reflection one can set the values of fields in that class.
- This is typically done only in special circumstances when setting the values in the usual way is not possible.
- Since such access usually violates the design intentions of the class, it should be used with the utmost discretion.

Concept of Methods

- The **java.lang.reflect.Method** class provides APIs to access information about the following:
  - Method’s modifiers
  - Return type
  - Parameters
  - Annotations and
  - Thrown exceptions
- It can also be used to invoke methods.

Method Type Information

- Consider the following code:

```
public Method[] getMethods()
        throws SecurityException
```

  - It returns an array of Method objects.
  - The methods are returned in no particular order.

```
public Method getMethod(String name,
                 Class[] parameterTypes)
throws NoSuchMethodException, SecurityException
```

Method Type Information

- getDeclaringClass() :
  - It returns the Class object representing the class or interface.
- getName() :
  - It returns the name of the method represented by this Method.
- getModifiers() :
  - It returns the Java language modifiers for the method.
- getParameterTypes() :
  - It returns an array of Class objects that represent the formal parameter types, in declaration order, of the method represented by this Method object.

Demo : Method Type Information

Demo:

MethodInfo.java

Types of Retrieving& Parsing Method Modifiers

- There are several modifiers that may be part of a method declaration:
  - Access modifiers: public, protected, and private
  - Modifier restricting to one instance: static
  - Modifier prohibiting value modification: final
  - Modifier requiring override: abstract
  - Modifier preventing reentrancy: synchronized
  - Modifier indicating implementation in another programming language: native

Types of Retrieving& Parsing Method Modifiers

- Modifier forcing strict floating point behavior: strictfp
- Annotations

Process of Invoking a Method

- Let us see an example on invoking a method:

```
Class aClass = anObject.getClass();
Class[] paramTypes = new Class[1];
paramTypes[0] = String.class;
Method m = null;
try {
    m = aClass.getMethod("confirmMsg", paramTypes);
}catch (NoSuchMethodException nsme)
{nsme.printStackTrace();}
Object[] params = new Object[1];
params[0] = "This is a test";
```

contd.

Process of Invoking a Method

```
try {
    String result = (String)m.invoke(anObject, params);
    System.out.println(result);
}catch (IllegalAccessException iae) {iae.printStackTrace();}
catch (InvocationTargetException ite) {ite.printStackTrace();}
```

Concept of Constructor

- A **Constructor** is used in the creation of an object.
- Typically it performs operations required to initialize the class before methods are invoked or fields are accessed.
- Constructors are never inherited.
- The operations that can be done on Constructors are:
  - Finding Constructors
  - Retrieving and Parsing Constructor Modifiers
  - Creating New Class Instances

Process of Retrieving& Parsing a Constructor

- Let us see an example on retrieving and parsing a constructor:

```
class SampleConstructor {
public static void main(String[] args) {
Rectangle r = new Rectangle();
showConstructors(r);
}
static void showConstructors(Object o) {
 Class c = o.getClass();
 Constructor[] theConstructors = c.getConstructors();
 for (int i = 0; i < theConstructors.length;i++)      {
```

contd.

Process of Retrieving& Parsing a Constructor

```
Class cx[] = theConstructors[i].getParameterTypes();
System.out.print(theConstructors[i].getName()+"( ");
if (cx.length > 0) {
    for (int j = 0; j < cx.length; j++) {
        System.out.print(cx[j].getName());
        if (j < (cx.length - 1)) System.out.print(", ");  }  }
System.out.print(") ");
System.out.println("{ ... }");  } } }
```

Methods for Creating new Instances

- Two reflective methods are used for creating instances of classes:
  - Class.newInstance()
    - It can only invoke the zero-argument constructor.
    - It requires that the constructor be visible.
  - Constructor.newInstance()
    - It may invoke any constructor, regardless of the number of parameters.
    - It may invoke private constructors under certain circumstances.

Demo : Methods for Creating new Instances

Demo:

NewInstance.java