Reflection

Java **Reflection** provides ability to inspect and modify the runtime behavior of application. Reflection in Java is one of the advance topic of core java. Using java reflection we can inspect a class, interface, enum, get their structure, methods and fields information at runtime even though class is not accessible at compile time. We can also use reflection to instantiate an object, invoke its methods, and change field values.

Reflection in Java is a very powerful concept and it's of little use in normal programming but it's the backbone for most of the Java, debugger tool, J2EE frameworks like spring, Hibernate and Java editors like eclipse, net beans etc. All spring technology runs on reflection, to understand this I need to study **Spring dependency**.

The list is endless and they all use java reflection because all these frameworks have no knowledge and access of user defined classes, interfaces, their methods etc.

#. We should not use reflection in normal programming where we already have access to the classes and interfaces because of following drawbacks.

- **Poor Performance –**
  Since java reflection resolve the types dynamically, it involves processing like scanning the classpath to find the class to load, causing slow performance.
- **Security Restrictions –**
  Reflection requires runtime permissions that might not be available for system running under security manager. This can cause our application to fail at runtime because of security manager.
- **Security Issues –**
  Using reflection we can access part of code that we are not supposed to access, for example we can access private fields of a class and change its value. This can be a serious security threat and cause our application to behave abnormally.
- **High Maintenance –**
  Reflection code is hard to understand and debug, also any issues with the code can't be found at compile time because the classes might not be available, making it less flexible and hard to maintain.

#.Reflection is used to find out the details of any class at runt time whenever we are getting some clue about that class.

Rule: one "**Class**" class object is treated as a reflected image of some other class.

Rule: Whenever any class is loaded into a memory the one "**Class**" class object for that loaded class will be created by the JVM. It means every object will have a mirror image, from there we can get detail of original class.

java.lang.Class is the entry point for all the reflection operations. For every type of object, JVM implicitly instantiates an immutable instance of java.lang.Class that **provides** methods to examine the runtime properties of the object and create new objects, invoke its method and get/set object fields.

#. If we get some clues at run time then via reflection we can get class details. There are **three** types of clue we usually get.

 # Hint 1:. We are getting a class name at run time in the form of string and most of the time we get this hint.

If we are getting class name as string then first we have to load the class and get the mirror object that is "Class" class Object.

"Class" class has a static method **forName(String s)** that accepts class name as string and returns the mirror object. It throws **ClassNotFoundException**. It is a checked exception.

Class.forName() method takes a class name as argument in the form of string with the complete package. Return type of this method is "Class" class object.

 # IQ: When a ClassNotFoundException occurs?
Whenever we are loading a class at runtime by using a **forName()** method of a "Class" class, if that class is not found in a "**class path variable**" then JVM throws ClassNotFoundException.

 # IQ: When ClassCastException occurs?
Whenever and Up-Casting and Down-Casting fails then JVM throws ClassCastException.

IQ: When Instantiation Exception occurs?

newInstance() method of "Class" class always use the default Constructor for creating the object of any class. If default Constructor is not found in that class then it throws InstantiationException.

 # Hint 2: We are getting an object of some class at run time into the reference value of parent class.

 # Hint 3: We know the class name at compile time and we want to find out the details of that class at run time.

Program with All Hints:

```
/* Reflection : Reflection is used to find out the detail of any class
at runtime whenever we are getting a some clue about
    that class at run-time. */

import java.lang.reflect.*;
import java.awt.*;
class ClassInformation{

/*  Hint 2: We are getting an object of some class at run time into
the reference value of parent class. */
public static void printName(Object o){
    /* returns the object id of "Class" class object */
    Class c = o.getClass();
    System.out.println(c.getName());
}

/* Hint 1: We are getting a class name at runtime in the form of
string:
In this case we have to first load the class, once we get the object
we can find out all details of the class. */

public static void printName(String s){
    try{
        /* load the class and it will return "Class" object that
represents  "MyClass" class. */
        Class c = Class.forName(s);

        /* method of "Class" that returns name of given class. */
        System.out.println(c.getName());

        /* Down-casting from "Object" class object to StringBuffer.
Also creating object without new keyword */
        StringBuffer sb = (StringBuffer) c.newInstance();
```

```
        /* confirming that object is ready and can call it's methods.
*/
        System.out.println(sb.length());
        System.out.println(sb.capacity());
    }catch(ClassNotFoundException e){
        /* Class.forName() throws ClassNotFoundException */
        System.out.println("CNF: " + e);
    }
    catch(Exception e){
        System.out.println("Exception: " + e);
    }
}

/* We are getting a class name at runtime in the form of string:
In this case we have to first load the class, once we get the object
we can find out all details of the class.
To create the object it requires to downcast in the same class which
we gathering information.      */
public static void printName1(String s){
    try{
        Class c = Class.forName(s);
        System.out.println(c.getName());

        /* If default constructor not found in class then throws
InstantiationException */
        Temp sb = (Temp) c.newInstance();
        sb.show();
    }catch(Exception e){
        System.out.println(e);
    }
}

public static void main(String...a){
    Button b = new Button("BB");
    printName(b);

    printName("java.lang.StringBuffer");
    //printName("java.awt.Frame");

    printName1("Temp");

    /* Hint 3: We know the class name at compile time and we want to
find out the detail of that class at run time.*/
    Class c = java.lang.Thread.class;
    System.out.println(c.getName());
}
}

class Temp{

Temp(){
```

```
    System.out.println("Default Constructor");
}

void show(){
    System.out.println("Creation of object via reflection");
}
}
```

 # Get Super Class
getSuperclass() method on a Class object returns the super class of the class. If this Class represents either the Object class, an interface, a primitive type, or void, then null is returned. If this object represents an array class then the Class object representing the "**Object**" class is returned.

 # Get Type Parameters
getTypeParameters() returns the array of TypeVariable if there are any Type parameters associated with the class. The type parameters are returned in the same order as declared.

 # Get Implemented Interfaces
getGenericInterfaces() method returns the array of interfaces implemented by the class with generic type information. We can also use **getInterfaces()** to get the class representation of all the implemented interfaces.

 # Get All Public Methods
getMethods() method returns the array of **public** methods of the Class including public methods of its **SuperClasses** and **SuperInterfaces**.

 # Get All Public Constructors
getConstructors() method returns the list of **public** constructors of the class reference of object.

 # Get All Public Fields
getFields() method returns the array of public fields of the class including public fields of its super classes and super interfaces.

 # Get All Annotations
getAnnotations() method returns all the annotations for the element, we can use it with class, fields and methods also. Note that only annotations available with reflection are with retention policy of RUNTIME, check out Java Annotations Tutorial.

 # Getting Class Modifiers
getModifiers() method returns the "**int**" representation of the class modifiers, we can use java.lang.reflect.Modifier.toString() method to get it in the string format as used in source code.

```
/* It print modifiers name as string */
System.out.println(Modifier.toString(concreteClass.getModifiers(
)));
System.out.println(Modifier.toString(Class.forName("com.journald
ev.reflection.BaseInterface").getModifiers()));
```

 # Getting Package Name
getPackage() method returns the package for this class. The class loader of this class is used to find the package. We can invoke getName() method of Package to get the name of the package.

 # Get Declared Classes
getDeclaredClasses() method returns an array of Class objects reflecting all the classes and interfaces declared as members of the class represented by this Class object. The returned array doesn't include classes declared in inherited classes and interfaces.

 # Get Declaring Class
getDeclaringClass() method returns the Class object representing the class in which it was declared.

 # Java Reflection for Fields
Reflection API provides several methods to analyze Class fields and modify their values at runtime, in this section we will look into some of the commonly used reflection functions for methods.

 # Get Public Field
In last section, we saw how to get the list of all the public fields of a class. Reflection API also provides method to get specific public field of a class through **getField()** method. This method look for the field in the specified class reference and then in the **super interfaces** and then in the **super classes**.

```
Field field =
Class.forName("com.journaldev.reflection.ConcreteClass").getFiel
d("interfaceInt");
```

Above call will return the field from Child class if not then check in Parent class. If there is no field found then it throws **NoSuchFieldException**.

 # Field Declaring Class
We can use getDeclaringClass() of field object to get the class declaring the field.

```
try {
    Field field =
Class.forName("com.journaldev.reflection.ConcreteClass").getField("interfaceI
nt");
    Class fieldClass = field.getDeclaringClass();

    //prints com.journaldev.reflection.BaseInterface
    System.out.println(fieldClass.getCanonicalName());
} catch (NoSuchFieldException | SecurityException e) {
    e.printStackTrace();
}
```

 # Get Field Type
getType() method returns the Class object for the declared field type, if field is primitive type, it returns the wrapper class object.

 # Get/Set Public Field Value:
We can get and set the value of a field in an Object using reflection.

Get/Set Public Field Value
We can get and set the value of a field in an Object using reflection.

get() method return Object, so if field is primitive type, it returns the corresponsing Wrapper Class. If the field is static, we can pass Object as null in get() method.

There are several set*() methods to set Object to the field or set different types of primitive types to the field. We can get the type of field and then invoke correct function to set the field value correctly. If the field is final, the set() methods throw **java.lang.IllegalAccessException.**

 # Get/Set Private Field Value
We know that private fields and methods can't be accessible outside of the class but using reflection we can get/set the private field value by turning off the java access check for field modifiers.

Java Reflection for Methods
Using reflection we can get information about a method and we can invoke it also. In this section, we will learn different ways to get a method, invoke a method and accessing private methods.

 # Get Public Method
We can use getMethod() to get a public method of class, we need to pass the method name and parameter types of the method. If the method is not found in the class, reflection API looks for the method in superclass.

 # Invoking Public Method
We can use invoke() method of Method object to invoke a method. If the method is static, we can pass NULL as object argument.

 # Invoking Private Methods
We can use getDeclaredMethod() to get the private method and then turn off the access check to invoke it, below example shows how we can invoke method3() of BaseClass that is static and have no parameters.

 # Java Reflection for Constructors
Reflection API provides methods to get the constructors of a class to analyze and we can create new instances of class by invoking the constructor.

 # Get Public Constructor
We can use getConstructor() method on the class representation of object to get specific public constructor. We also get the array of parameter types for the constructor.

 # Instantiate Object using Constructor
We can use newInstance() method on the constructor object to instantiate a new instance of the class. Since we use reflection when we don't have the class's information at compile time, we can assign it to Object and then further use reflection to access its fields and invoke its methods.

 # Reflection for Annotations
Annotations was introduced in Java 1.5 to provide **metadata** information of the class, methods or fields and now it's heavily used in frameworks like Spring and Hibernate. Reflection API was also extended to provide support to analyze the annotations at runtime.

Using reflection API we can analyze annotations whose retention policy is Runtime. I would suggest you to check out Java Annotations Tutorial.