Object Oriented Programming with Java
Practical-6

Nested class and Reflection API

Reflection API

Java Reflection is a *process of examining or modifying the run time behavior of a class at run time*.

The **java.lang.Class** class provides many methods that can be used to get metadata, examine and change the run time behavior of a class.

The java.lang and java.lang.reflect packages provide classes for java reflection.

java.lang.Class class

The java.lang.Class class performs mainly two tasks:

- provides methods to get the metadata of a class at run time.
- provides methods to examine and change the run time behavior of a class.

ow to get the object of Class class?

There are 3 ways to get the instance of Class class. They are as follows:

- forName() method of Class class
- getClass() method of Object class
- the .class syntax

1) forName() method of Class class

- is used to load the class dynamically.
- returns the instance of Class class.
- It should be used if you know the fully qualified name of class.This cannot be used for primitive types.

Let's see the simple example of forName() method.

FileName: Test.java

```
class Simple{}

public class Test{
 public static void main(String args[]) throws Exception {
  Class c=Class.forName("Simple");
  System.out.println(c.getName());
 }
```

}

Output:

```
Simple
```

2) getClass() method of Object class

It returns the instance of Class class. It should be used if you know the type. Moreover, it can be used with primitives.

FileName: Test.java

```
class Simple{}

class Test{
  void printName(Object obj){
  Class c=obj.getClass();
  System.out.println(c.getName());
  }
  public static void main(String args[]){
   Simple s=new Simple();

   Test t=new Test();
   t.printName(s);
 }
}
```

Output:

```
Simple
```

3) The .class syntax

If a type is available, but there is no instance, then it is possible to obtain a Class by appending ".class" to the name of the type. It can be used for primitive data types also.

FileName: Test.java

```
class Test{
  public static void main(String args[]){
   Class c = boolean.class;
   System.out.println(c.getName());

   Class c2 = Test.class;
   System.out.println(c2.getName());
 }
}
```

Output:

```
boolean
Test
```

Determining the class object

The following methods of Class class are used to determine the class object:

1) public boolean isInterface(): determines if the specified Class object represents an interface type.

2) public boolean isArray(): determines if this Class object represents an array class.

3) public boolean isPrimitive(): determines if the specified Class object represents a primitive type.

Let's see the simple example of reflection API to determine the object type.

FileName: Test.java

```
class Simple{}
interface My{}

class Test{
 public static void main(String args[]){
  try{
   Class c=Class.forName("Simple");
   System.out.println(c.isInterface());

   Class c2=Class.forName("My");
   System.out.println(c2.isInterface());

  }catch(Exception e){System.out.println(e);}

 }
}
```

Output:

```
 false
 true
```

Reflecting Fields, Methods, and Constructors

The package `java.lang.reflect` provides classes that can be used for manipulating class members. For example,

- **Method class** - provides information about methods in a class
- **Field class** - provides information about fields in a class
- **Constructor class** - provides information about constructors in a class

1. Reflection of Java Methods

The `Method` class provides various methods that can be used to get information about the methods present in a class. For example,

```
import java.lang.Class;
import java.lang.reflect.*;

class Dog {

  // methods of the class
  public void display() {
    System.out.println("I am a dog.");
  }

  private void makeSound() {
    System.out.println("Bark Bark");
  }
}

class Main {
  public static void main(String[] args) {
    try {

      // create an object of Dog
      Dog d1 = new Dog();

      // create an object of Class
      // using getClass()
      Class obj = d1.getClass();

      // using object of Class to
      // get all the declared methods of Dog
      Method[] methods = obj.getDeclaredMethods();
```

```
      // create an object of the Method class
      for (Method m : methods) {

        // get names of methods
        System.out.println("Method Name: " + m.getName());

        // get the access modifier of methods
        int modifier = m.getModifiers();
        System.out.println("Modifier: " + Modifier.toString(modifier));

        // get the return types of method
        System.out.println("Return Types: " + m.getReturnType());
        System.out.println(" ");
      }
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

O/P

```
Method Name: display

Modifier: public
Return Types: void

Method Name: makeSound
Modifier: private
Return Types: void
```

2. Reflection of Java Fields

Like methods, we can also inspect and modify different fields of a class using the methods of the `Field` class. For example,

```
import java.lang.Class;

import java.lang.reflect.*;


class Dog {

  public String type;

}


class Main {

  public static void main(String[] args) {

    try {

      // create an object of Dog

      Dog d1 = new Dog();


      // create an object of Class

      // using getClass()

      Class obj = d1.getClass();


      // access and set the type field

      Field field1 = obj.getField("type");

      field1.set(d1, "labrador");


      // get the value of the field type

      String typeValue = (String) field1.get(d1);

      System.out.println("Value: " + typeValue);


      // get the access modifier of the field type
```

```
      int mod = field1.getModifiers();


      // convert the modifier to String form
      String modifier1 = Modifier.toString(mod);
      System.out.println("Modifier: " + modifier1);
      System.out.println(" ");
    }


    catch (Exception e) {
      e.printStackTrace();
    }
  }
}


O/P

Value: labrador

Modifier: public
```

In the above example, we have created a class named *Dog*. It includes a public field named *type*. Notice the statement,

```
Field field1 = obj.getField("type");
```

Here, we are accessing the public field of the *Dog* class and assigning it to the object *field1* of the *Field* class.

We then used various methods of the `Field` class:

- **field1.set()** - sets the value of the field
- **field1.get()** - returns the value of field
- **field1.getModifiers()** - returns the value of the field in integer form

3. Reflection of Java Constructor

We can also inspect different constructors of a class using various methods provided by the Constructor class. For example,

```
import java.lang.Class;

import java.lang.reflect.*;



class Dog {



  // public constructor without parameter

  public Dog() {



  }



  // private constructor with a single parameter

  private Dog(int age) {



  }



}



class Main {

  public static void main(String[] args) {

    try {

      // create an object of Dog

      Dog d1 = new Dog();



      // create an object of Class

      // using getClass()

      Class obj = d1.getClass();
```

```
      // get all constructors of Dog
      Constructor[] constructors = obj.getDeclaredConstructors();

      for (Constructor c : constructors) {

        // get the name of constructors
        System.out.println("Constructor Name: " + c.getName());

        // get the access modifier of constructors
        // convert it into string form
        int modifier = c.getModifiers();
        String mod = Modifier.toString(modifier);
        System.out.println("Modifier: " + mod);

        // get the number of parameters in constructors
        System.out.println("Parameters: " + c.getParameterCount());
        System.out.println("");
      }
    }

    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

O/P

```
Constructor Name: Dog
Modifier: public
```

```
Parameters: 0

Constructor Name: Dog
Modifier: private
Parameters: 1
```