

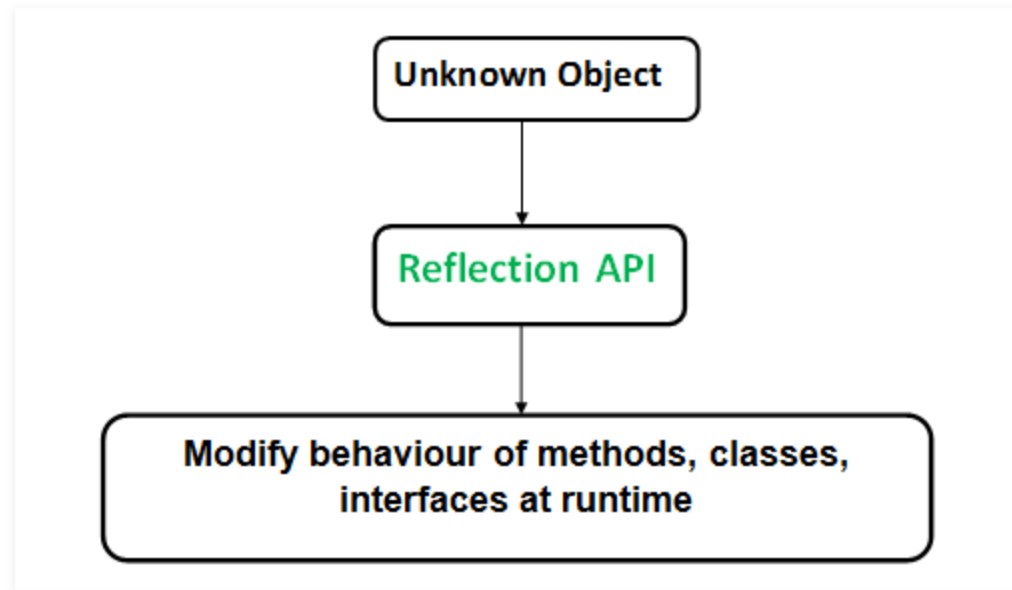
Reflection in JAVA

12.11.2020

Reflection

- ▶ Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.
- ▶ The required classes for reflection are provided under *java.lang.reflect* package.
- ▶ Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.
- ▶ Through reflection we can invoke methods at runtime irrespective of the access specifier used with them.

Reflection



Reflection

- ▶ Reflection can be used to get information about -
- ▶ **Class** The `getClass()` method is used to get the name of the class to which an object belongs.
- ▶ **Constructors** The `getConstructors()` method is used to get the public constructors of the class to which an object belongs.
- ▶ **Methods** The `getMethods()` method is used to get the public methods of the class to which an object belongs.

Reflection of Java Classes

- ▶ In order to reflect a Java class, we first need to create an object of *Class*.
- ▶ And, using the object we can call various methods to get information about methods, fields, and constructors present in a class.
- ▶ There exists three ways to create objects of Class:
 - ▶ **Using forName() method**
 - ▶ forName() method takes the name of the class to be reflected as its argument.

```
class Dog {...}  
  
// create object of Class  
// to reflect the Dog class  
Class a = Class.forName("Dog");
```

Reflection of Java Classes

► Using getClass() method

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

// create an object of Class
// to reflect Dog
Class b = d1.getClass();
```

- Here, we are using the object of the *Dog* class to create an object of *Class*.

► Using .class extension

```
// create an object of Class
// to reflect the Dog class
Class c = Dog.class;
```

- Now that we know how we can create objects of the *Class*. We can use this object to get information about the corresponding class at runtime.

Reflection of Java Classes

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

class Animal {
}

// put this class in different Dog.java file
public class Dog extends Animal {
    public void display() {
        System.out.println("I am a dog.");
    }
}

// put this in Main.java file
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 name of the class
            String name = obj.getName();
            System.out.println("Name: " + name);

            // get the access modifier of the class
            int modifier = obj.getModifiers();

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

            // get the superclass of Dog
            Class superClass = obj.getSuperclass();
            System.out.println("Superclass: " + superClass.getName());
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
Name: Dog
Modifier: public
Superclass: Animal
```

In the above example, we have created a superclass: `Animal` and a subclass: `Dog`. Here, we are trying to inspect the class `Dog`.

Reflection of Java Classes

Notice the statement,

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

Here, we are creating an object `obj` of `Class` using the `getClass()` method. Using the object, we are calling different methods of `Class`.

- **`obj.getName()`** - returns the name of the class
- **`obj.getModifiers()`** - returns the access modifier of the class
- **`obj.getSuperclass()`** - returns the super class of the class

To learn more about `Class`, visit [Java Class \(official Java documentation\)](#).

Note: We are using the `Modifier` class to convert the integer access modifier to a string.

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

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();
}
}
```

```
Method Name: display
Modifier: public
Return Types: void
```

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

Reflection of Java Methods

In the above example, we are trying to get information about the methods present in the `Dog` class. As mentioned earlier, we have first created an object `obj` of `Class` using the `getClass()` method.

Notice the expression,

```
Method[] methods = obj.getDeclaredMethod();
```

Here, the `getDeclaredMethod()` returns all the methods present inside the class.

Also, we have created an object `m` of the `Method` class. Here,

- **`m.getName()`** - returns the name of a method
- **`m.getModifiers()`** - returns the access modifier of methods in integer form
- **`m.getReturnType()`** - returns the return type of methods

The `Method` class also provides various other methods that can be used to inspect methods at run time. To learn more, visit [the Java Method class \(official Java documentation\)](#).

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();
    }
}
```

Value: labrador
Modifier: public

Reflection of Java Fields

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

Reflection of Java Fields

Similarly, we can also access and modify private fields as well. However, the reflection of private field is little bit different than the public field. For example,

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

class Dog {
    private String color;
}

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 the private field color
            Field field1 = obj.getDeclaredField("color");

            // allow modification of the private field
            field1.setAccessible(true);

            // set the value of color
            field1.set(d1, "brown");
```

```
            // get the value of field color
            String colorValue = (String) field1.get(d1);
            System.out.println("Value: " + colorValue);

            // get the access modifier of color
            int mod2 = field1.getModifiers();

            // convert the access modifier to string
            String modifier2 = Modifier.toString(mod2);
            System.out.println("Modifier: " + modifier2);
        }

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

```
Value: brown
Modifier: private
```

Reflection of Java Fields

In the above example, we have created a class named `Dog`. The class contains a private field named `color`. Notice the statement.

```
Field field1 = obj.getDeclaredField("color");  
  
field1.setAccessible(true);
```

Here, we are accessing `color` and assigning it to the object `field1` of the `Field` class. We then used `field1` to modify the accessibility of `color` and allows us to make changes to it.

We then used `field1` to perform various operations on the private field `color`.

To learn more about the different methods of `Field`, visit [Java Field Class \(official Java documentation\)](#).

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();
}
}
```

Constructor Name: Dog
Modifier: public
Parameters: 0

Constructor Name: Dog
Modifier: private
Parameters: 1

Reflection of Java Constructor

In the above example, we have created a class named `Dog`. The class includes two constructors.

We are using reflection to find the information about the constructors of the class. Notice the statement,

```
Constructor[] constructors = obj.getDeclaredConstructor();
```

Here, the we are accessing all the constructors present in `Dog` and assigning them to an array `constructors` of the `Constructor` type.

We then used object `c` to get different informations about the constructor.

- **c.getName()** - returns the name of the constructor
- **c.getModifiers()** - returns the access modifiers of the constructor in integer form
- **c.getParameterCount()** - returns the number of parameters present in each constructor

To learn about more methods of the `Constructor` class, visit [Constructor class](#)

Assignment 1

- ▶ Create class Test with one argument String s;
- ▶ Create one constructor without parameters which will set the value of s to “Programming languages”
- ▶ Create public void method1() which will print the value of s
- ▶ Create public void method2(int n) which will print the value of n
- ▶ Create private void method3() which will print “Private method is invoked”

Assignment 1

- ▶ Create public class Main with public static void main() method
- ▶ Create an object of the Test class
 - ▶ Print the class name

```
Class cls = test.getClass();  
System.out.println("The name of class is " +  
    cls.getName());
```

- ▶ Print the constructor name

```
Constructor constructor = cls.getConstructor();  
System.out.println("The name of constructor is " +  
    constructor.getName());
```

- ▶ Print all methods names

```
Method[] methods = cls.getMethods();
```

Assignment 1

Output should look like this:

```
"C:\Program Files\Java\jdk1.8.0_231\bin\java.exe" ...  
The name of class is Test  
The name of constructor is Test  
The public methods of class are :  
Method name: method1  
Method name: method2  
Method name: wait  
Method name: wait  
Method name: wait  
Method name: equals  
Method name: toString  
Method name: hashCode  
Method name: getClass  
Method name: notify  
Method name: notifyAll  
The name of class is Test
```

Why method3 is not printed? Where all these additional methods come from?

Assignment 1

Methods `wait`, `equals`, `toString`... are methods defined in the `Object` class from which every Java class has implicit inheritance from.

`Method3` is not printed because it has private access modifier.

The reflection API has a way to access private methods and fields using `setAccessible(true)`;

Assignment 2

Extend the code from assignment 1

Invoke the method1 by providing the parameter name

```
System.out.println("Printing the result of method 1:");  
Method methodCall1 = cls.getDeclaredMethod( name: "method1");  
methodCall1.invoke(test);
```

Invoke the method1 by providing the parameter name and parameter class

```
Method methodCall2 = cls.getDeclaredMethod( name: "method2",int.class);  
methodCall2.invoke(test, ...args: 15);
```

Assignment 2

Try to print the value of the field s:

```
Field field = cls.getDeclaredField( name: "s");  
System.out.println(field.get(test));
```

Run the program.

This will result with `java.lang.IllegalAccessException: Class Main can not access a member of class Test with modifiers "private" because the attribute has private access`

Assignment 2

Enable access to the field s before printing it

```
// allows the object to access the field irrespective
// of the access specifier used with the field
field.setAccessible(true);

//Now the field is accessible and the value will be printed
System.out.println(field.get(test));
```

Now the value will be printed

Change the value of s

```
System.out.println("Print the changed value of s");
field.set(test,"JAVA");
methodCall1.invoke(test);
```

Now do the same for the method3. Enable the access and invoke it!

Assignment 2

Output:

```
Printing the result of method 1:  
The value of the string is Programming languages|  
Printing the result of method 2:  
The number is 15  
Printing the value of s  
Programming languages  
Print the changed value of s  
The value of the string is JAVA  
Printing the result of method 3:  
Private method invoked
```