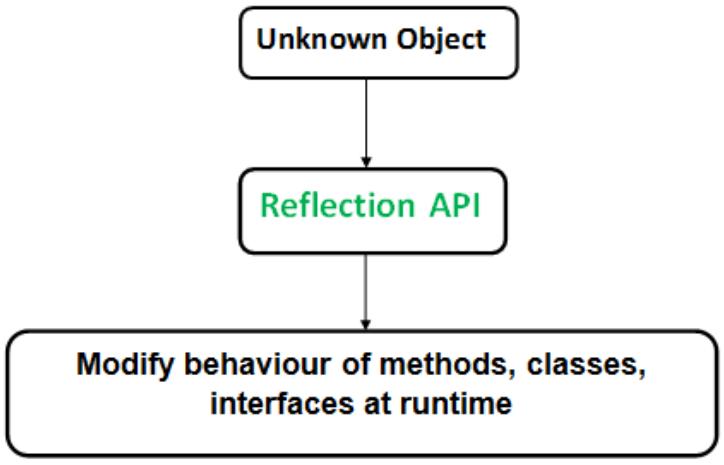
Reflection in Java

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 can be used to get information about –

1. **Class** The getClass() method is used to get the name of the class to which an object belongs.
2. **Constructors** The getConstructors() method is used to get the public constructors of the class to which an object belongs.
3. **Methods** The getMethods() method is used to get the public methods of the class to which an objects belongs.

```
// A simple Java program to demonstrate the use of reflection
import java.lang.reflect.Method;
import java.lang.reflect.Field;
import java.lang.reflect.Constructor;

// class whose object is to be created
class Test
{
    // creating a private field
    private String s;

    // creating a public constructor
    public Test()  {  s = "GeeksforGeeks"; }

    // Creating a public method with no arguments
    public void method1()  {
        System.out.println("The string is " + s);
    }

    // Creating a public method with int as argument
    public void method2(int n)  {
        System.out.println("The number is " + n);
    }

    // creating a private method
    private void method3() {
        System.out.println("Private method invoked");
    }
}

class Demo
{
    public static void main(String args[]) throws Exception
    {
        // Creating object whose property is to be checked
        Test obj = new Test();

        // Creating class object from the object using
        // getclass method
        Class cls = obj.getClass();
        System.out.println("The name of class is " +
                           cls.getName());
```

```
        // Getting the constructor of the class through the
        // object of the class
        Constructor constructor = cls.getConstructor();
        System.out.println("The name of constructor is " +
                           constructor.getName());

        System.out.println("The public methods of class are : ");

        // Getting methods of the class through the object
        // of the class by using getMethods
        Method[] methods = cls.getMethods();

        // Printing method names
        for (Method method:methods)
            System.out.println(method.getName());

        // creates object of desired method by providing the
        // method name and parameter class as arguments to
        // the getDeclaredMethod
        Method methodcall1 = cls.getDeclaredMethod("method2",
                                                   int.class);

        // invokes the method at runtime
        methodcall1.invoke(obj, 19);

        // creates object of the desired field by providing
        // the name of field as argument to the
        // getDeclaredField method
        Field field = cls.getDeclaredField("s");

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

        // takes object and the new value to be assigned
        // to the field as arguments
        field.set(obj, "JAVA");

        // Creates object of desired method by providing the
        // method name as argument to the getDeclaredMethod
        Method methodcall2 = cls.getDeclaredMethod("method1");

        // invokes the method at runtime
        methodcall2.invoke(obj);

        // Creates object of the desired method by providing
        // the name of method as argument to the
        // getDeclaredMethod method
        Method methodcall3 = cls.getDeclaredMethod("method3");

        // allows the object to access the method irrespective
        // of the access specifier used with the method
        methodcall3.setAccessible(true);

        // invokes the method at runtime
        methodcall3.invoke(obj);
    }
}
```

Output :


```
The name of class is Test
The name of constructor is Test
The public methods of class are :
method2
method1
wait
wait
wait
equals
toString
```

```
hashCode
getClass
notify
notifyAll
The number is 19
The string is JAVA
Private method invoked
```

Important observations :

1. We can invoke an method through reflection if we know its name and parameter types. We use below two methods for this purpose
   getDeclaredMethod() : To create an object of method to be invoked. The syntax for this method is

   ```
   Class.getDeclaredMethod(name, parametertype)
   name- the name of method whose object is to be created
   parametertype - parameter is an array of Class objects
   ```

   invoke() : To invoke a method of the class at runtime we use following method–

   ```
   Method.invoke(Object, parameter)
   If the method of the class doesn’t accepts any
   parameter then null is passed as argument.
   ```

2. Through reflection we can **access the private variables and methods** of a class with the help of its class object and invoke the method by using the object as discussed above. We use below two methods for this purpose.
   Class.getDeclaredField(FieldName) : Used to get the private field. Returns an object of type Field for specified field name.
   Field.setAccessible(true) : Allows to access the field irrespective of the access modifier used with the field.

Advantages of Using Reflection:

- **Extensibility Features:** An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
- **Debugging and testing tools**: Debuggers use the property of reflection to examine private members on classes.

Drawbacks:

- **Performance Overhead:** Reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.
- **Exposure of Internals:** Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

Reference:
https://docs.oracle.com/javase/tutorial/reflect/index.html

This article is contributed by **Akash Ojha**.If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Recommended Posts:

Java.util.BitSet class methods in Java with Examples | Set 2
Shadowing of static functions in Java
How does default virtual behavior differ in C++ and Java ?
How are Java objects stored in memory?
How are parameters passed in Java?
Are static local variables allowed in Java?
final variables in Java