REFLECTION API

Done by
Syed Shahinsha
Riya Jain
Rima Ray

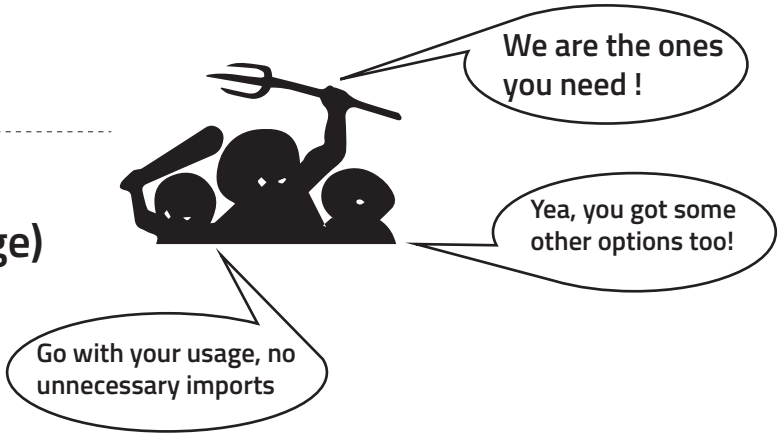
So What is Reflection API?

In java, we can get the metadata of methods, constructors, access modifiers ... of a class by using the reflection API.

Reflection is the process of analysing the capabilities of a particular class at runtime.

In order to use reflection API, we need something right? And that's

java.lang.reflect.*;



(or import as per our usage)
java.lang.Class;
java.lang.reflect.Field;
java.lang.reflect.Method;
java.lang.reflect.Constructors;
java.lang.reflect.modifiers;

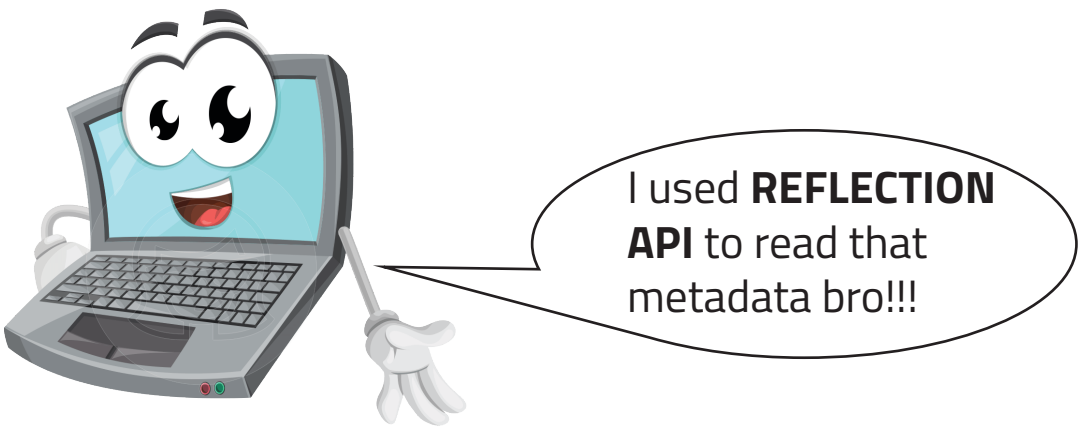
Still Confused?? Let's understand with an example how JVM works with classes

```
private class A
{
    private class B{
    }

    public static void main(String [] args){

    }

}
```

Here, the compiler will throw an error. In the first line, we gave outer class A, a private access specifier.

In Java, outer classes should not use private/ protected. They should use public/default.

But wait!! How did the compiler read that **private type** of class A??



Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.

Reflection API is used more in product development (such as JVM architecture, testing tools, debuggers, Compiler design) than in project development.

And Why is that??? **So much stuff alert!!!!!!!**

1. IDEs, Debuggers, testing tools can heavily make use of reflection in order to provide solutions for auto completion features, dynamic typing, hierarchy structures, etc.

   For example, **IDEs like Eclipse** or PHP Storm provide a mechanism to retrieve dynamically the arguments expected for a given method or a list of public methods starting by "get" for a given instance. All these are done using reflection.

2. **Dependency injection** frameworks **use reflection to inject beans** and properties at runtime and initialize all the context of an application.

3. External tools that make **use of the code dynamically** may use reflection as well.

Reflection can be used to get information about –

- Class - by using getClass()
- Constructors -by using getConstructors()
- Methods - by using getMethods() or grtDeclaredMethods()
- Modifiers
- Fields (variables) - by using getFields() or getDeclaredFields()

Reflection API is used to read the meta data of a class

These are some of the methods that we are going to use in REFLECTION API

Method	Description
1) public String getName()	returns the class name
2) public static Class forName(String className)throws ClassNotFoundException	loads the class and returns the reference of Class class.
3) public Object newInstance()throws InstantiationException,IllegalAccessException	creates new instance.
4) public boolean isInterface()	checks if it is interface.
5) public boolean isArray()	checks if it is array.
6) public boolean isPrimitive()	checks if it is primitive.
7) public Class getSuperclass()	returns the superclass class reference.
8) public Field[] getDeclaredFields()throws SecurityException	returns the total number of fields of this class.
9) public Method[] getDeclaredMethods()throws SecurityException	returns the total number of methods of this class.
10) public Constructor[] getDeclaredConstructors()throws SecurityException	returns the total number of constructors of this class.
11) public Method getDeclaredMethod(String name,Class[] parameterTypes)throws NoSuchMethodException,SecurityException	returns the method class instance.

And yea, we know this will be your reaction after seeing this table!

We'll go on with the concepts..

Reflection API for class

To get the metadata of a class, we need to first create the obj of the class.

There are three ways to create an object,

- Class c= Class.forName ("**Emloyee**");
- Employee emp=new **Employee()**;
  Class c= emp.getClass();
- Class c= **Employee**.class injecting into c using class file

To get the **name of the class - getName();**

To get the **interfaces of the class - getInterfaces();**

To get the **superclass of the class - getSuperclass();**

Note that the superclass that we retrieve is also an object of another class

To get the **access modifiers data of the class,** we use some different approach

int i=**getModifiers()**; --->stores the value of modifier
Modifier.toString(i); --->converts the value to modifier

Let us consider an example,

Employee.java -------

```
public abstract class Emloyee implements java.io.Serializable{

}
```

Test.java -------

```
import java.lang.reflect.*;
public class Test{

    public static void main(String args[]) throws Exception{
        Class c= Class.forName("Employee");
        System.out.println("Name :" + c.getName());
        System.out.println("Superclass :" +c.getSuperclass()
                                                .getName());
        System.out.println("Interfaces :");
        Class[] c1=c.getInterfaces();
        for(Class c1:c)
        {
            System.out.print(c1.getName()+" ");
        }
        int i=c.getModifiers();
        System.out.println("Modifiers :"+Modifier.toString(i));
    }
}
```

Output:

Name : Employee
Superclass :java.lang.Object
Interfaces :java.io.Serializable
Modifiers :public abstract

Reflection API for Field

To get the metadata of the fields (variables), we use

- **getFields()** -->to get only the public variables from same class and superclass
- **getDeclaredFields()** -->to get all variables of current class
- **get(Field f)** -->to get the value of the variable
- **getType()** -->to get the data type class of the variable

The other methods such as getName(), getModifiers() are similar to the Class

Let us consider an example,

Employee.java -------

```
public  class Emloyee {
        public int eno=111;
        private String ename="Syed";
        public String address="Mumbai";
}
```

Test.java -------

```
import java.lang.reflect.*;
public class Test{
        public static void main(String args[]) throws Exception{
                Class c= Class.forName("Employee");
                Fields[] field=c.getDeclaredFields();
                for(Field f:field){
                        System.out.println("Name :" + f.getName());
                        System.out.println("Datatype :" +f.getType()
                                                        .getName());
                        int i=c.getModifiers();
                        System.out.println("Modifiers :"+Modifier.toString(i));
                        System.out.println("Value :"+f.get(f);
                }
        }
}
```

Output:

Name :eno
Datatype :int
Modifiers :public
Value :111

Name :ename
Datatype : java.lang.String
Modifiers :private
Value : Syed

Name :address
Datatype : java.lang.String
Modifiers :public
Value : Mumbai

Reflection API for Method

To get the metadata of the methods, we use

- **getMethods()** -->to get only the public methods from same class and superclass
- **getDeclaredMethods()** -->to get all methods of current class
- **getParameterType()** -->to get the parameter type of the method
- **getReturnType()** -->to get the return data type of the method
- **getExceptionTypes()** -->to get the exception of method

The other methods such as getName(), getModifiers() are similar to the Class

Let us consider an example,

Employee.java -------

```
public  class Emloyee {
        public int eno=111;
        private String ename="Syed";
        public String address="Mumbai";

        public int add(int empno) throws SQLException{
        }
        public String search(int empno) throws SQLException{

        }
}
```

Test.java -------

```
import java.lang.reflect.*;
public class Test{
    public static void main(String args[]) throws Exception{
        Class c= Class.forName("Employee");
        Methods[ ] method=c.getDeclaredMethods();
        for(Method m:method){
            System.out.println("Name :" + m.getName());
            System.out.println("Datatype :" +m.getReturnType()
                                                .getName());
            int i=c.getModifiers();
            System.out.println("Modifiers :"+Modifier.toString(i));
            System.out.println("Parameter Type :");
            Class[] c1=m.getParameterTypes();
            for(Class :c1){
                System.out.print(p.getName()+" ");
            }
        } }
```

Output:

Name :add
Datatype :int
Modifiers :public
Parameter Type :int

Name :search
Datatype :java.lang.String
Modifiers :public
Parameter Type :int

Reflection API for Constructor

To get the metadata of the constructors , we use

- **getConstructors()** -->to get only the public constructor of same class
- **getDeclaredConstructors()** -->to get all the constructors of same class
- **getParameterTypes()** -->to get the parameter datatype of the constructor

The other methods such as getName(), getModifiers() are similar to the Methods[ ]

Let us consider an example,

Employee.java -------

```
public  class Emloyee {
        public int eno=111;
        private String ename="Syed";
        public String address="Mumbai";

        public Employee(int empno) {
        }
        public Employee(int empno, String ename, String address ){

        }
}
```

Test.java -------

```
import java.lang.reflect.*;
public class Test{
    public static void main(String args[]) throws Exception{
        Class c= Class.forName("Employee");
        Constructors[] constructor=c.getDeclaredConstructors();
        for(Constructor c1:constructor){
            System.out.println("Name :" + c1.getName());
            int i=c.getModifiers();
            System.out.println("Modifiers :"+Modifier.toString(i));
            System.out.println("Parameter Type :");
            Class[] c2=m.getParameterTypes();
            for(Class c2:c1){
                System.out.print(c2.getName()+" ");
            }
        } }
```

Output:

Name : Employee
Modifiers :public
Parameter Type :int

Name : Employee
Modifiers :public
Parameter Type :int java.lang.String

You're almost done!! Just some more!!

In some cases, we can actually change or modify the behavior of the methods, classes, interfaces at runtime.

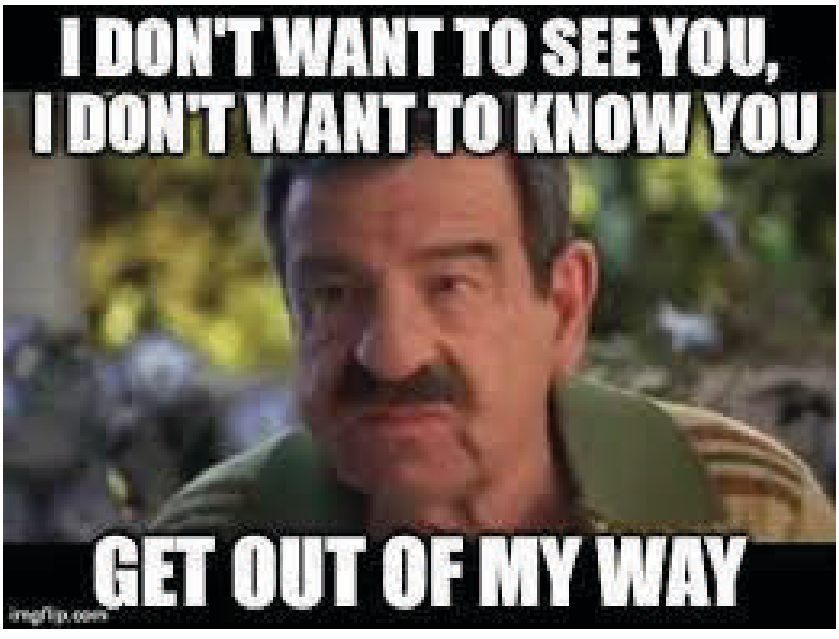
Lets go with an example

```
import java.lang.reflect.Constructor;
 class Test
{
   private String s;
   public Test()  {  s = "Java"; }
   public void method1()  {
      System.out.println("The string is " + s);
   }
}
```

```
class Demo
{
   public static void main(String args[]) throws Exception
   {
      Class cls= Class.forName("Test");

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

      /* allows the object to access the field irrespective
         of the access specifier used with the field */
      field.setAccessible(true);
      field.set(obj, "Hogaya Re Baba");
      }
}
```

Output:

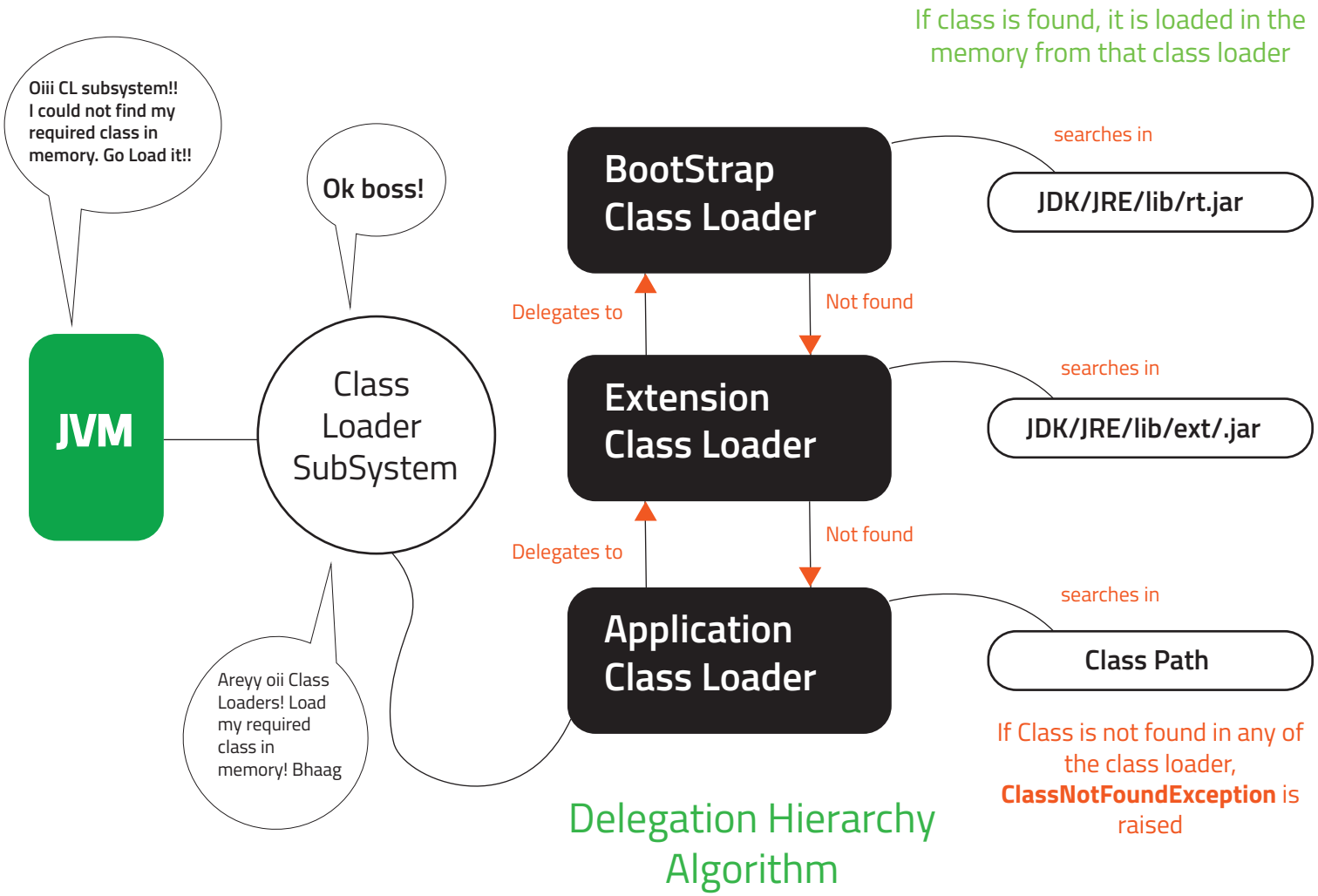
The string is Hogaya Re Baba



That's all for now!!!

Now to the next topic, how Class Loader works

Working of Class Loader



1 At compilation, all the required class files are loaded in the memory.

2 If the JVM cannot find its required class file, it requests the class loader sub system to load the class.

3 ClassLoader SubSystem has three Class Loaders :

- **Bootstrap or Primordial ClassLoader:** This classloader is responsible for loading the internal core java classes present in the rt.jar and other classes present in the java.lang.* package

- **Extension ClassLoader:** This classloader is the child class of Primordial classloader and is responsible for loading the classes from the extension classpath (i.e. jdk\jre\lib\ext).

- **Application or System ClassLoader:** This classloader is the child class of Extension classloader and is responsible for loading the classes from the system classpath. It internally uses the 'CLASS PATH' environment variable and is written in Java language.

4 The Class Loaders load the required class in the memory.

5 If the required class is not found, it raises **ClassNotFoundException**

A Simple Example

```
public void printClassLoaders() throws ClassNotFoundException {

   System.out.println("Classloader of this class:"
      + PrintClassLoader.class.getClassLoader());

   System.out.println("Classloader of Logging:"
      + Logging.class.getClassLoader());

   System.out.println("Classloader of ArrayList:"
      + ArrayList.class.getClassLoader());
}
```

Output:

```
Class loader of this class:sun.misc.Launcher$AppClassLoader@18b4aac2
Class loader of Logging:sun.misc.Launcher$ExtClassLoader@3caeaf62
Class loader of ArrayList:null
```

We can also create custom class loaders by extending the ClassLoader and over riding the required method in the class Loader or invoking any particular function as per our requirement.

But for now, we can rest with the concept... And there are many programs on net!
Self Study bro!!!