Abstraction in OOPS | Selenium

Table of content

Abstraction in Java/selenium

Okay Guys, Forgive me for the information I am about to talk because I am not able to accept the descriptions present on the Internet about the **Abstraction**.

Internet Says (almost all websites) : Hiding internal details and showing functionality is known as abstraction.; I do not agree with this because this is what Encapsulation. isn't it ?

I could be wrong in my description, but I will continue with my perception if you are not comfortable, please skip this topic.

Abstraction is nothing but the process of giving an set of guidelines to the user, who are going to implement these guidelines based on their needs.

For Example : I hope you are aware of the headphone jack in the phones and radios, or the different chargers for different mobiles, which were available till 2008 (approximately, I do not know the exact year). Previously we had different phones with different chargers and different headphones jacks.

After 2008 most of the companies came to an agreement that everyone should manufacture the Phones based on few specifications, which included headphones, charger ports(except Apple).

So they have concluded with 3.5mm jack and standard charger port, now you can interchangeably use the chargers and the headphones.

Like you can use Samsung headphones with onePlus mobile.

Similar to the above scenarios when an architect design a framework, or a library or software, he will give specifications/guidelines.

If somebody wants to take advantage of the framework/library, they have to implement those specifications or guidelines.

Java provides **Interface and Abstract class** to achieve this abstraction.

Interface in Java :

An interface defines a set of methods as a contract or as guidelines. The class that is implementing this interface must implement these functionalities in the concrete class.

An interface allows you to **guarantee that certain methods exist and return the required types**. When the compiler knows this, it can use that assumption to work with unknown classes as if they had certain known behavior.

The interfaces add value when you are building a larger system or a library.

Using references to interfaces instead of classes helps in minimizing future changes as the user of an interface reference doesn't have to worry about the changes in the underlying concrete implementation.

Whenever there is an interface, there is Inheritance, as the class which changes/ overrides the interface method must be a subclass of Interface.

Webdriver interface provides few standard abstract methods to handle the webpages, So when some browsers want to access the Selenium method, then they have to implement the Webdriver interface and provide implementation to the methods present in the Webdriver interface.

You must implement all the abstract methods present in the interface, but the behavior of the methods could be in your own way.

```
public interface WebDriver {
   // public- access modifier
   // void- return type
   // get - method name
   public void get(String url);
   public String getTitle();
}
```

Below Class implements the WebDriver interface.

```
public class FireFoxDriver implements WebDriver{
   @Override
   public void get(String url) {
      System.out.println("Open webpage");
      // actual code differs, in get method
   }
   @Override
   public String getTitle() {

      System.out.println("Webpage title");
```

```
        // actual code differs, in getTite method
        return "title";
    }
}
```

If you do not implement the methods present in the WebDriver class, then Compiler shows an error.

Types of Methods in Interface from Java9

YES, we can have private methods in the interface in **java 9; private** methods could be static or non-static. In both cases, the private method is not inherited by sub-interfaces or implementations.

They are mainly there to improve code re-usability within the same interface only – thus improving encapsulation.

Using private methods in interfaces have four rules :

Private interface methods cannot be abstract.

A private method can be used only inside the interface.

A private static method can be used inside other static and non-static interface methods.

Private non-static methods cannot be used inside private static methods.

Interfaces can have below items in the interface in Java9 (aka JDK 1.9) :

Constant variables

Abstract methods

Default methods

Static methods

Private methods

Private Static methods

```
package frames;
interface ISam {
    public static final String name = "Java9";
    public void testAbstract();
    default void testDefault(){
        // call non-static method from another non-static method.
        testPrivateNonStatic();
        // call static method from non-static method
        testPrivateStatic();

        System.out.println("Default method");
    }

    public static void testStatic(){
        System.out.println("static method");
    }
    private void testPrivateNonStatic(){
        System.out.println("private non-static method");
    }
    private static void testPrivateStatic(){
        System.out.println("private static method");
    }
}
public class Sample implements ISam{
    @Override
    public void testAbstract() {
        System.out.println("Overridden 'testAbstract' method");
    }
    public static void main(String[] args) {
        System.out.println("constant variable value : "+ name);
        // access static method
        ISam.testStatic();
        // create object for Sample class
        Sample sam = new Sample();
        // we can access default method using object
        sam.testDefault();
        // call abstract method (implementation in Sample class)
        sam.testAbstract();
    }
}
```

Output

```
constant variable value : Java9
static method
private non-static method
private static method
Default method
Overridden 'testAbstract' method
```

Abstract Class :

Note : With the latest updates of Interface, Abstract class may go obsolete in 3-4 years as Interface now started to accept the method with implementations.

Abstract method : method with no implementation

Concrete Method : method with implementation

We can override both abstract and concrete methods in subclasses, but we cannot create an object to an abstract class, but we can create an object to the subclass which overrides the abstract class completely.

Abstract class must have an **abstract** keyword in method declaration so abstract methods.

```
public abstract class WebDriver {
   // public- access modifier
   // void- return type
   // get - method name
   public abstract void get(String url);
   public String getTitle() {
      System.out.println("Webpage title");
      // actual code differs, in getTite method
      return "title";
   }
}
```

Below Class implements the WebDriver interface.

```
public class ABSFireFoxDriver extends WebDriver{
   @Override
   public void get(String url) {
      System.out.println("Open webpage");
      // actual code differs, in get method
   }
}
```

Handling Custom Dropdowns in selenium

Multi-level Abstraction in Java :

It is not mandatory to override all the abstract methods present in the abstract class in a single sub-class; if we do not provide all the implementation in a subclass, then we have to mark it as an abstract class only.

We can create an object to a subclass that does not have any abstract methods in it. So we can implement the abstraction methods in multi-level.

Multilevel Abstraction is nothing but the implementing a interface or Abstract class methods in different sub classes

For Example an Interface 'A' has 4 abstract methods called testa(), testb(), testc(), testd().

Now we can implement multilevel abstraction like class SubB implements interface A and provides implementation only to testa(), another class SubC extends SubB class and provides implementation to testb(), another class SubD extends SubC

class and provides implementation to testc() and testd().

Now SubB, SubC classes are providing implementation to only a few methods but not for all so they must be marked as abstract classes. But class SubD provides implementation to all the methods so we can mark it as a normal class and we can create an object to it.

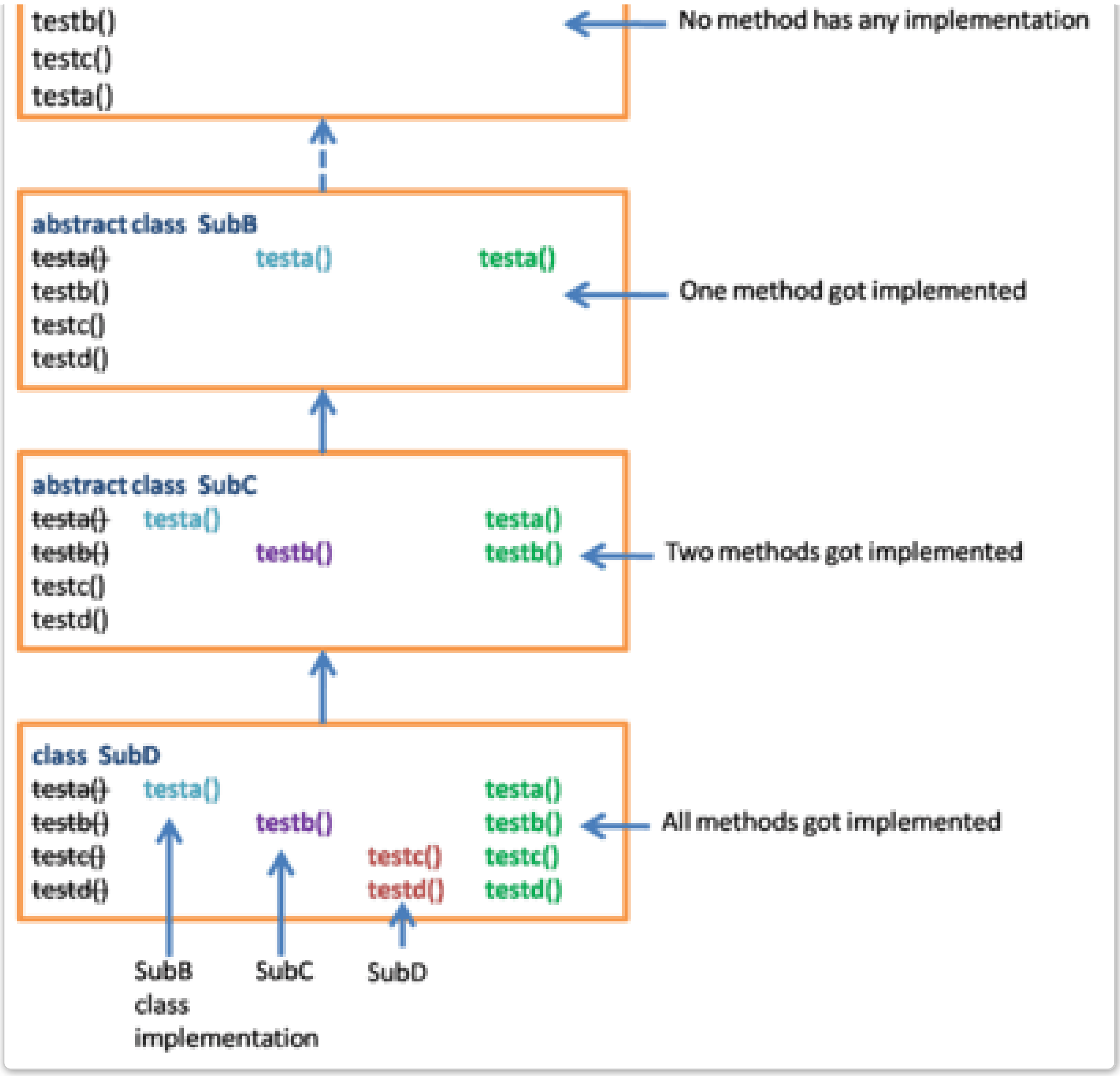
If you have a query like SubD provides implementation to only testc(), testd(), then you might not notice the relation between the SubB, SubC, SubD.

When a parent class provides implementation to some abstract class, then it is not mandatory for a subclass to provide implementation to the method, which has an implementation in the parent class.

If you ask cannot I override it in a subclass ?, yes you can override, but it is optional only.

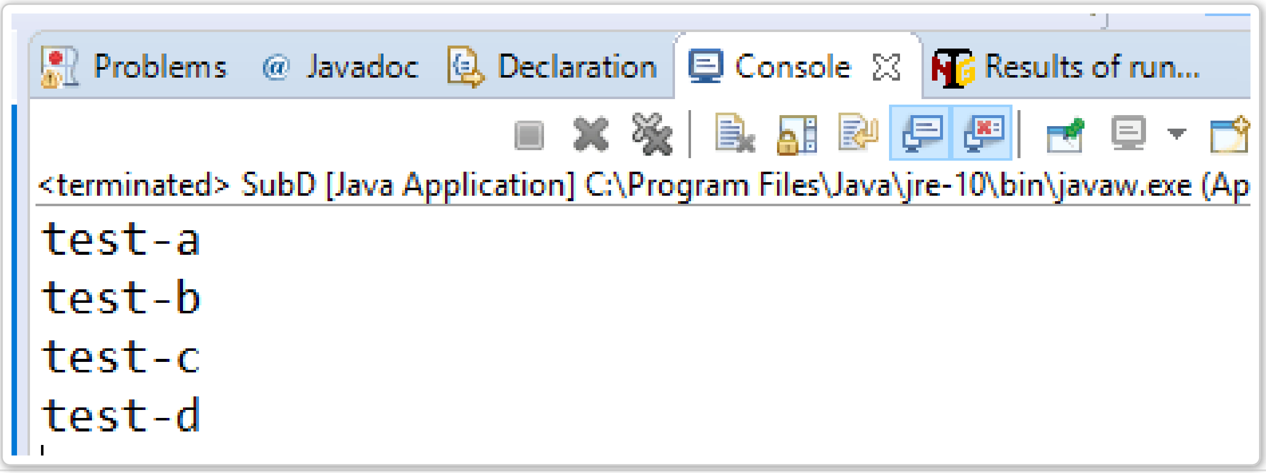
Below images show how the multi-level abstraction works:

Interface A
testa()


testb()
testc()
testa()
No method has any implementation
abstract class SubB
~~testa()~~ testa() testa()
testb()
testc()
testd()
One method got implemented
abstract class SubC
~~testa()~~ testa() testa()
~~testb()~~ testb() testb()
testc()
testd()
Two methods got implemented
class SubD
~~testa()~~ testa() testa()
~~testb()~~ testb() testb()
~~testc()~~ testc() testc()
~~testd()~~ testd() testd()
All methods got implemented
SubB class implementation
SubC
SubD

Program for multi-level abstraction

```
package test;
interface A{
   public void testa();
   public void testb();
   public void testc();
   public void testd();
}
abstract class SubB implements A {
   public void testa() {
      System.out.println("test-a");
   }
}
abstract class SubC extends SubB{
   public void testb() {
      System.out.println("test-b");
   }
}
public class SubD extends SubC{
   public void testc() {
        System.out.println("test-c");
    }
    public void testd() {
        System.out.println("test-d");
    }
    public static void main(String[] args) {
        // create object to class SubD
        SubD subdObj = new SubD();
        subdObj.testa();
        subdObj.testb();
        subdObj.testc();
        subdObj.testd();
    }
}
```



```
<terminated> SubD [Java Application] C:\Program Files\Java\jre-10\bin\javaw.exe (Ap
test-a
test-b
test-c
test-d
```

Methods in Interface

YES, we can have private methods in the interface in **java 9; private** methods could be static or non-static. In both cases, the private method is not inherited by sub-interfaces or implementations.

They are mainly there to improve code re-usability within the same interface only – thus improving encapsulation.

Using private methods in interfaces have four rules : 1. Private interface methods cannot be abstract.

2. A private method can be used only inside the interface.

3. A private static method can be used inside other static and non-static interface methods.

4. Private non-static methods cannot be used inside private static methods. Interfaces can have below items in the interface in Java9 (aka JDK 1.9) : 1. Constant variables

2. Abstract methods

3. Default methods

4. Static methods

5. Private methods

6. Private Static methods

```
package frames;

interface ISam {
   public static final String name = "Java9";
   public void testAbstract();
   default void testDefault(){
      // call non-static method from another non-static method.
      testPrivateNonStatic();
      // call static method from non-static method
      testPrivateStatic();

      System.out.println("Default method");
   }

   public static void testStatic(){
      System.out.println("static method");
   }
   private void testPrivateNonStatic(){
      System.out.println("private non-static method");
   }
   private static void testPrivateStatic(){
      System.out.println("private static method");
   }
}

public class Sample implements ISam{
   @Override
   public void testAbstract() {
      System.out.println("Overridden 'testAbstract' method");
   }
   public static void main(String[] args) {
      System.out.println("constant variable value : "+ name);
      // access static method
      ISam.testStatic();
      // create object for Sample class
      Sample sam = new Sample();
      // we can access default method using object
      sam.testDefault();
      // call abstract method (implementation in Sample class)
      sam.testAbstract();


   }
}
```

Output

```
constant variable value : Java9
static method
private non-static method
private static method
Default method
Overridden 'testAbstract' method
```

Recommended Readings

- **Encapsulation in OOPS | Selenium**
- **Wrapper Classes in Java**
- **Collections in Selenium**
- **Arraylist in Selenium**
- **Inheritance in OOPS | Selenium**
- **Classes & Constructors in OOPS | Selenium**
- **Benefits of Java**
- **Sprint**