Polymorphism in OOPS | Selenium

Table of content

Polymorphism in Java/selenium

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

We can create functions or reference variables that behave differently in a different programmatic context.

Polymorphism is one of the primary building blocks of object-oriented programming along with inheritance, abstraction, and encapsulation.

There are two kinds of polymorphism in Java :

Compile Time polymorphism / Static polymorphism

Run Time Polymorphism / Dynamic Polymorphism

Method Overloading :

Method Overloading is a feature that allows a class to have more than one method having the same name, with a different type of parameters or with a different number of parameters or both.

Things to remember on method Overloading

To call an overloaded method in Java, it is must to use the type and/or a number of arguments to determine which version of the overloaded method to actually call.

Overloaded methods may have different return types; the return type alone is insufficient to distinguish two versions of a method.

When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

It allows the user to achieve compile-time polymorphism.

An overloaded method can throw different exceptions.

It can have different access modifiers.

```
class OverloadExample
{
   void demo (int a)
   {
      System.out.println ("a: " + a);
   }
   void demo (int a, int b)
   {
      System.out.println ("a and b: " + a + "," + b);
   }
   double demo(double a) {
      System.out.println("double a: " + a);
      return a*a;
   }
}
class OverloadingMainClass
{
   public static void main (String args [])
   {
      // create object for OverloadExample
      OverloadExample Obj = new OverloadExample();
      Obj.demo(10);
      Obj.demo(10, 20);
      double result = Obj.demo(5.5);
      System.out.println("Output : " + result);
   }
}
```

Exception in selenium

Rules for Method Overloading

Overloading can take place in the same class only; it would be considered as a different method if the overloading is happening in its sub-class.

A Constructor in Java can be overloaded the same as Method

Overloaded methods must have a different argument list. The parameters may differ in their type or number, or both.

They may have the same or different return types

It is also known as compile-time polymorphism

Xpath in Selenium

Method Overloading in Selenium :

Selenium uses method overloading to accept different parameters, without disturbing the other methods.

I hope you are familiar with Frames. How do you **switch a frame in selenium ?**

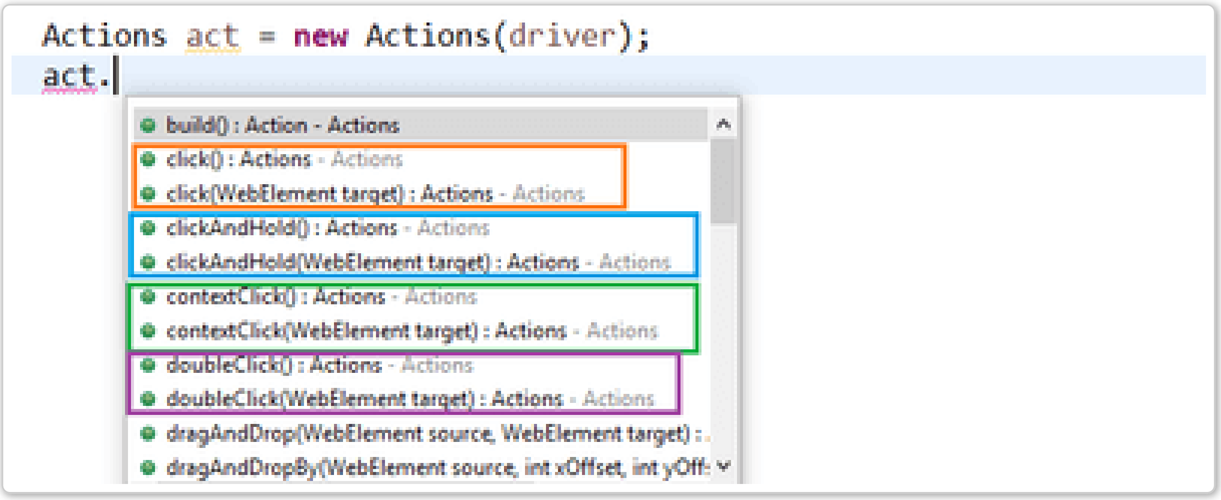
You will use `driver.switchTo().frame(String id or Name), frame(Webelement element` , Did you notice both the **names of the methods are the same but they accept different types of parameters**.

The first `frame(String id or Name)` accepts a String parameter, but the second frame(WebElement element) accepts the Webelement as a parameter.

When there is more than one method with the same name, and those methods accept either different types of parameters or different number parameter, then it is called as Method Overloading.

There are few more places in selenium where method Overloading is implemented, Those are :

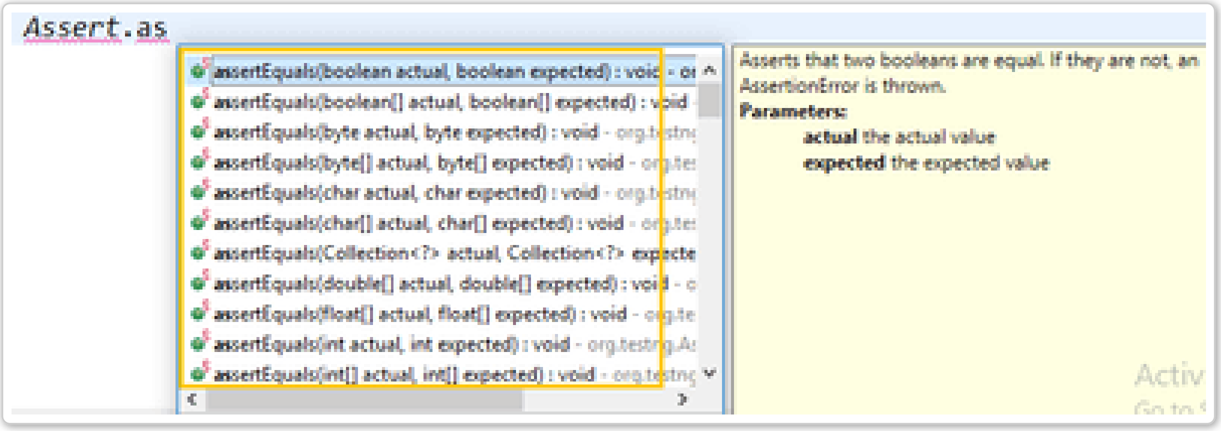
1. Methods present in Actions class, Method accepts webelement, and another method accepts no parameter, few methods accepts co-ordinates with webelement



```
Actions act = new Actions(driver);
act.
```

```
build() : Action - Actions
click() : Actions - Actions
click(WebElement target) : Actions - Actions
clickAndHold() : Actions - Actions
clickAndHold(WebElement target) : Actions - Actions
contextClick() : Actions - Actions
contextClick(WebElement target) : Actions - Actions
doubleClick() : Actions - Actions
doubleClick(WebElement target) : Actions - Actions
dragAndDrop(WebElement source, WebElement target) : .
dragAndDropBy(WebElement source, int xOffset, int yOff:
```

2. Almost all the methods present in the **Assert class in TestNG**.



```
Assert.as
```

```
assertEquals(boolean actual, boolean expected) : void - o
assertEquals(boolean[] actual, boolean[] expected) : void
assertEquals(byte actual, byte expected) : void - org.testng
assertEquals(byte[] actual, byte[] expected) : void - org.te
assertEquals(char actual, char expected) : void - org.testng
assertEquals(char[] actual, char[] expected) : void - org.te
assertEquals(Collection<?> actual, Collection<?> expecte
assertEquals(double[] actual, double[] expected) : void - o
assertEquals(float[] actual, float[] expected) : void - org.te
assertEquals(int actual, int expected) : void - org.testng.A
assertEquals(int[] actual, int[] expected) : void - org.testng
```

Asserts that two booleans are equal. If they are not, an AssertionError is thrown.
Parameters:
actual the actual value
expected the expected value

Fluent wait in selenium

Type Casting in Method Overloading :

Changing the type of a value from one primitive type to another primitive type is called as Types casting. Mostly typecasting occurs with the types whose parent is Number class.

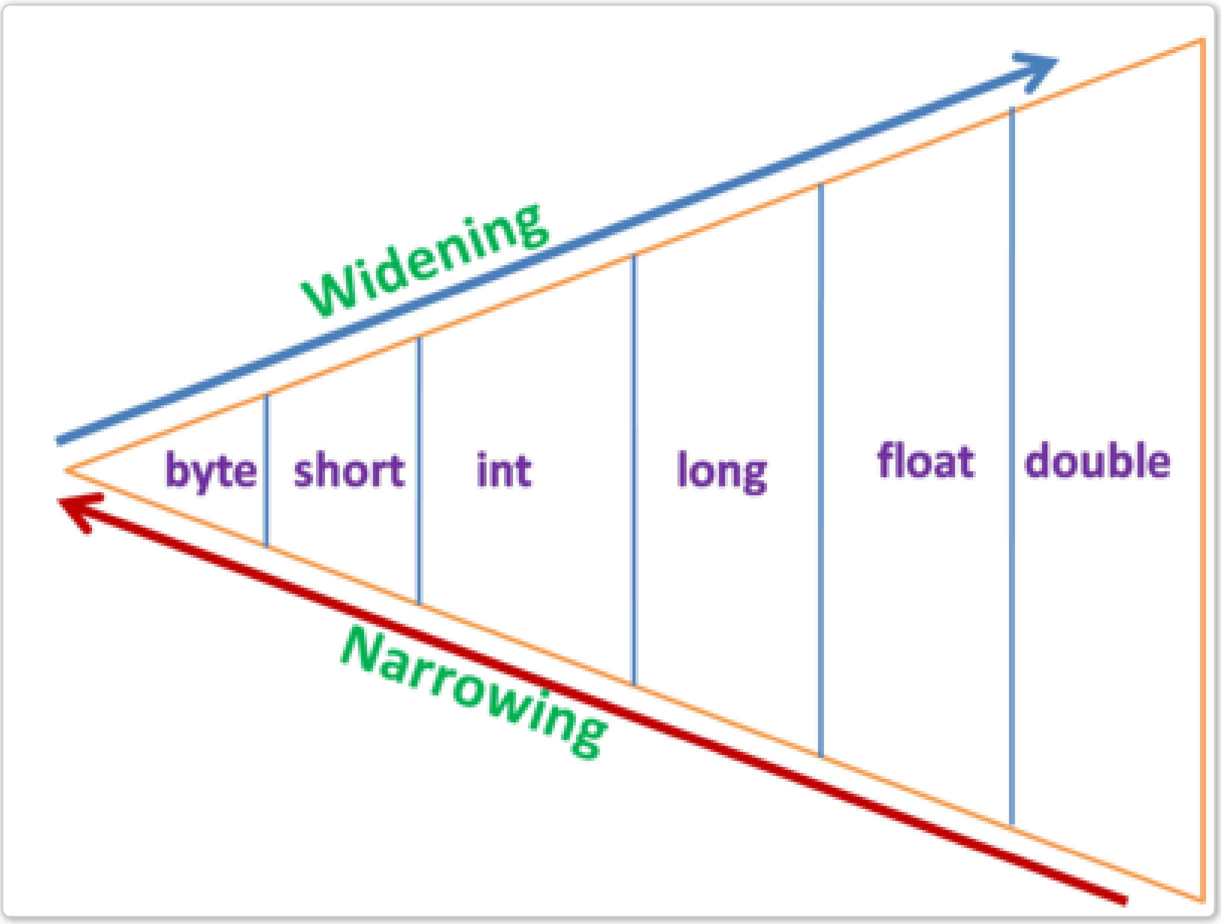
When a data type of **smaller size is promoted to the data type of bigger size than this is called Widening.**

If the Bigger size is converted to a lower size, then it is called Narrowing. There is no auto-narrowing, which means the compiler cannot narrow values unless the user specifies so.

If we do not specify the type conversion explicitly, then the compiler itself will perform the widening operation, and this is called Auto-Widening.

For example, the byte data type can be promoted to short; a short data type can be promoted to int, long, double, etc.

Type Casting can occur towards bigger size, or to lower size as well.



Example for widening and Auto-widening, and Narrowing

```
int i = 10;
// widening
float f = (float) i;
// auto widening
float fAuto = i;
// narrarowing


double doub =  10.12;
int inarrow = (int) doub;
```

Not only in normal scenarios the typecasting occurs but in case of method overloading also the typecasting occurs, but mostly Auto-widening and also called as TypePromotion

Typecasting will not occur in method overloading if there is a method with an exact matching signature.

More than one method with the same name and argument list cannot be given in a class even though their return type is different; **Method return type doesn't matter in case of overloading.**

```
package cherchertech;
public class TypeCastingExample {
   // method with byte inputs
   public void add(byte b1, byte b2)
   {
      System.out.println("BYTE addition result is : " + (b1+b2));
   }
   // method with int inputs
   public void add(float f1, float f2)
   {
      System.out.println("FLOAT addition result is : " + (f1+f2));
   }

   public static void main(String[] args) {
      // we will call the add method with int values
      //there is no method which accepts int values
      TypeCastingExample te = new TypeCastingExample();
      int a = 10, b = 10;
      te.add(a, b);
   }
}
```

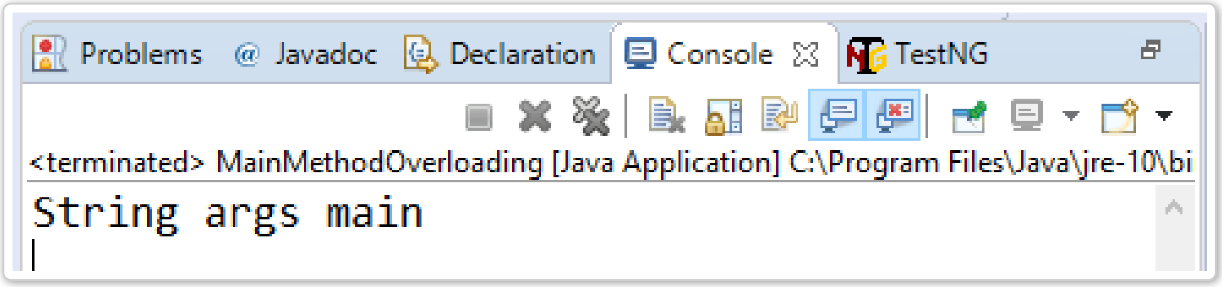
Can we overload the java main() method?

You can have n - number of main methods in a class by method overloading. But JVM calls **main(String[] args) method** which receives string array as argument only.

```
package newtest;
public class MainMethodOverloading {
   public static void main(String[] args) {
      System.out.println("String args main");
   }
   public static void main(int i) {
      System.out.println("int main");
   }
   public static void main(int i1, int i2) {
      System.out.println("two int main");
   }
   public static void main(boolean b) {
      System.out.println("boolean main");


   }
}
```

the output of main method overloading :



```
<terminated> MainMethodOverloading [Java Application] C:\Program Files\Java\jre-10\bi
String args main
```

Reading QR Code from the website using Webdriver

Method Overriding :

The subclass has the same method as of Superclass. In such cases, Subclass overrides the parent class method without even touching the source code of the Superclass. This feature is known as method overriding.

The Subclass will provide the implementation as per its wish; Parent class will not have any control over how to implement the method.

Method Overriding is also known as Runtime polymorphism.

```
public class BaseClass
{
   public void methodToOverride() //Super class method
   {
       System.out.println ("I'm the method of BaseClass");
   }
}
public class DerivedClass extends BaseClass
{
   public void methodToOverride() //Sub Class method
   {
       System.out.println ("I'm the method of DerivedClass");
   }
}

public class TestMethod
{
    public static void main (String args []) {
      // BaseClass reference and object
      BaseClass obj1 = new BaseClass();
      // BaseClass reference but DerivedClass object
      BaseClass obj2 = new DerivedClass();
      // Calls the method from BaseClass class
      obj1.methodToOverride();
```

```
        //Calls the method from DerivedClass class
        obj2.methodToOverride();
    }
}
```

Rules for Method Overriding :

Applies only to inherited methods
Object type (NOT reference variable type) determines which overridden method will be used at runtime
The overriding method can have different return types.
The overriding method must not have a more restrictive access modifier
Abstract methods must be overridden
Static and final methods cannot be overridden
Constructors cannot be overridden

Javascript Executor

Dynamic method dispatch in overriding ::

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime.

When an overridden method is called by a reference, **java determines which version of that method to execute based on the type of object it refers to**.

In simple words, you have a grandfather and brother and sister. All of your grandchildren have inherited your grandfather's behaviors, along with your own behaviors.

Now can I create a method which accepts your Grandfather behaviors, the same method can accept you (any of your sibling) because you are his grandchildren.

If I pass your grandfather as a parameter, then method behaves based on your grandfather's characters, but if I pass you as a parameter, then you will exhibit your + your grandfather's characteristics.

So this method is not locked with your grandfather or with your sibling, but it behaves based on the parameter you pass. So the coupling between the method and the caller happens only during the runtime, this is called dynamic dispatching.

If you have heard of **Multiple personality disorder** in the real world.

In the below example the compiler cannot decide the coupling as the parameter is resolved at run time, so dynamic dispatching or runtime coupling occurs.

Sorry for the below example, just yesterday I watched the 'Avengers : Infinity War 1'.

```
package newtest;
class SuperHero {
    public void protection(GrandFather_Avenger hero) {
        hero.tool();
    }
}
class GrandFather_Avenger {
    public void tool() {
        System.out.println("All Tools of Avengers");
    }
}
class Child_Hulk extends GrandFather_Avenger{
    public void tool() {
        System.out.println("Size is tool for Hulk");
    }
}
class Child_Strange extends GrandFather_Avenger{
    public void tool() {
        System.out.println("Time is tools for Dr.Strange");
    }
}
class Child_TomCruise extends GrandFather_Avenger{
    public void tool() {
        System.out.println("Trust me !, I'm also super Hero, Didn't you watch 'MI rogue nation'");
    }
}
public class TestDynamicDispatching {
    public static void main(String[] args) {
        SuperHero sh = new SuperHero();
        // Hulk characteristics
        sh.protection(new Child_Hulk());
        // Dr.Strange characteristics
        sh.protection(new Child_Strange());
        // Tom Cruise characteristics
        sh.protection(new Child_TomCruise());
        // Avenger characteristics
        sh.protection(new GrandFather_Avenger());
    }
}
```

iFrames in Selenium

Recommended Readings

Abstraction in OOPS | Selenium

Encapsulation in OOPS | Selenium

Wrapper Classes in Java

Collections in Selenium

Classes & Constructors in OOPS | Selenium

Benefits of Java

Sprint

Core Java Interview Questions Set 1