

Unit I

Syllabus: Generics: Introduction to Generics, simple Generics examples, Generic Types, Generic methods, Bounded Type Parameters and Wild cards, Inheritance & Sub Types, Generic super class and sub class, Type Inference, Restrictions on Generics.

Introduction to Generics:

Definition

"Generics allow the reusability of code, where one single method can be used for different data-types of variables or objects."

- The idea is to allow different types like Integer, String, ... etc and user-defined types to be a parameter to methods, classes, and interfaces.
- For example, classes like HashSet, ArrayList, HashMap, etc use generics very well. We can use them for any type.

Why Use Generics?

In a nutshell, generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar *formal parameters* used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

Code that uses generics has many benefits over non-generic code:

1.Stronger type checks at compile time.

- A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing errors at compile-time is easier than fixing errors at runtime, which can be difficult to find.

Example:

//Using ArrayList without Generics

```
List list = new ArrayList();  
list.add(10);  
list.add("10");  
//Using ArrayList  
//With Generics, it is required to specify the type of object we need to store.
```

```
List<Integer> list = new ArrayList<Integer>();  
list.add(10);  
list.add("10");// compile-time error
```

2.Elimination of Typecasts.

- The following code snippet without generics requires casting:

Example: List without Generics

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0); //while retrieving the data  
//specifying the type of data to be retrieved  
//explicitly using typecasting
```

When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
String s = list.get(0); // no typecasting needed
```

3.Enabling programmers to implement generic algorithms.

By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Non-generic Programming

The following example illustrates three non-generic (type-sensitive) functions for finding maximum out of 3 inputs:

Example:

```

class Main {
    public static void main(String[] args) {
        System.out.printf("Max of %d, %d and %d is %d\n\n", 3, 4, 5,
            MaximumTest.maximum(3, 4, 5));

        System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",
            6.6, 8.8, 7.7,MaximumTest.maximum(6.6, 8.8, 7.7));

        System.out.printf("Max of %s, %s and %s is %s\n", "pear", "apple",
            "orange",MaximumTest.maximum("pear", "apple", "orange"));
    }
}

class MaximumTest {
    // determines the largest of three Comparable objects
    public static int maximum(int x, int y, int z) {
        int max = x; // assume x is initially the largest
        if (y > max) {
            max = y; // y is the largest so far
        }
        if (z > max) {
            max = z; // z is the largest now
        }
        return max; // returns the largest object
    }

    public static double maximum(double x, double y, double z) {
        double max = x; // assume x is initially the largest

        if (y > max) {
            max = y; // y is the largest so far
        }
    }
}

```

```

    if (z > max) {
        max = z; // z is the largest now
    }
    return max; // returns the largest object
}

public static String maximum(String x, String y, String z) {
    String max = x; // assume x is initially the largest

    if (y.compareTo(max) > 0) {
        max = y; // y is the largest so far
    }

    if (z.compareTo(max) > 0) {
        max = z; // z is the largest now
    }

    return max; // returns the largest object
}
}

```

Three methods that do exactly the same thing, but cannot be defined as a single method because they use different data types. (int, double & String)

Generic methods

To use generic methods, we use the following syntax.

Syntax

```
<T> returntype nameOfGenericMethod(T element)
```

```
{
```

```
//Body
```

```
}
```

T is our generic data type's name, and when the method is to be called, it would be the same as if T was a typedef for your datatype.

The following example now illustrates how the maximum method would be written using a template:

```
class Main {
```

```
    public static void main(String[] args) {
```

```
        System.out.printf("Max of %d, %d and %d is %d\n\n", 3, 4, 5,
```

```
            MaximumTest.maximum(3, 4, 5));
```

```
        System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n", 6.6, 8.8, 7.7,MaximumTest.maximum(6.6, 8.8, 7.7));
```

```
        System.out.printf("Max of %s, %s and %s is %s\n", "pear", "apple", "orange",MaximumTest.maximum("pear", "apple", "orange"));
```

```
    }
```

```
}
```

```
class MaximumTest {
```

```
    // determines the largest of three Comparable objectspublic static < T extends Comparable < T >> T maximum(T x, T y, T z) {
```

```
        T max = x; // assume x is initially the largest
```

```
        if (y.compareTo(max) > 0) {
```

```
            max = y; // y is the largest so far
```

```
        }
```

```
        if (z.compareTo(max) > 0) {
```

```
            max = z; // z is the largest now
```

```

    }
    return max; // returns the largest object
}
}

```

Generic methods with multiple type parameters

In the above code, all of the arguments to `maximum()` must be the same type. Optionally, a template can have more type options, and the syntax is pretty simple. For a template with three types, called T1, T2 and T3, we have:

```

class Generics {
    public static < T1, T2, T3 > void temp(T1 x, T2 y, T3 z) {
        System.out.println("This is x =" + x);
        System.out.println("This is y =" + y);
        System.out.println("This is z =" + z);
    }
    public static void main(String args[]) {
        temp(1, 2, 3);
    }
}

```

Generic Class

Generic objects & type members

As another powerful feature of Java, you can also make *Generic classes*, which are classes that can have members of the **generic** type.

Example

```

class Test<T>
{
    T obj;    // An object of type T is declared
    Test(T obj) // parameterized constructor
    {
        this.obj = obj;
    }
    public T getObject() // get method
    {
        return this.obj;
    }
}

```

Explanation

- We have declared a class `Test` that can keep the `obj` of any type.
- The constructor `Test(T obj)` assigns the value passed as a parameter to data member `obj` of type `T`.
- The get method `T getObject()` returns the `obj` of type `T`.

Syntax to instantiate object

To create objects of the generic class, we use the following syntax.

```

// To create an instance of generic class
Test <DataType> obj = new Test <DataType>()

```

Have a look at the detailed implementation of Generic Class and its methods.

```

// We use <> to specify Parameter type
class Test < T > {
    T obj;
    Test(T obj) {
        this.obj = obj;
    }
    public T getObject() {
        return this.obj;
    }
}

class Main {
    public static void main(String[] args) {

```

```

// Test for Integer type
Test < Integer > obj1 = new Test < Integer > (5);
System.out.println(obj1.getObject());

// Test for double type
Test < Double > obj2 = new Test < Double > (15.777755);
System.out.println(obj2.getObject());

// Test for String type
Test < String > obj3 = new Test < String > ("Yayy! That's my
first Generic Class.");
System.out.println(obj3.getObject());
}
}

```

As you can see we create 3 different Integer, Double and String type variables for three different generic class objects.

Generics Types in Java

- A *generic type* is a generic class or interface or a method that is parameterized over types.

Generic Method:

Generic Java method takes a parameter and returns some value after performing a task. It is exactly like a normal function, however, a generic method has type parameters that are cited by actual type. This allows the generic method to be used in a more general way. The compiler takes care of the type of safety which enables programmers to code easily since they do not have to perform long, individual type castings.

```

// Java program to show working of user defined Generic
methods

class Test {
    // A Generic method example

```



```

static <T> void genericDisplay(T element)
{
    System.out.println(element.getClass().getName()
        + " = " + element);
}

// Driver method
public static void main(String[] args)
{
    // Calling generic method with Integer argument
    genericDisplay(11);

    // Calling generic method with String argument
    genericDisplay("GeeksForGeeks");

    // Calling generic method with double argument
    genericDisplay(1.0);
}
}

```

Output

```

java.lang.Integer = 11
java.lang.String = GeeksForGeeks
java.lang.Double = 1.0

```

Generics Work Only with Reference Types:

When we declare an instance of a generic type, the type argument passed to the type parameter must be a reference type. We cannot use primitive data types like **int**, **char**.

```
Test<int> obj = new Test<int>(20);
```

The above line results in a compile-time error that can be resolved using type wrappers to encapsulate a primitive type.

But primitive type arrays can be passed to the type parameter because arrays are reference types.

```
ArrayList<int[]> a = new ArrayList<>();
```

int ✓
int[] ✓

Generic Types Differ Based on Their Type Arguments:

Consider the following Java code.

```
// Java program to show working of user-defined Generic classes

// We use < > to specify Parameter type
class Test<T> {
    // An object of type T is declared
    T obj;
    Test(T obj) { this.obj = obj; } // constructor
    public T getObject() { return this.obj; }
}

// Driver class to test above
class Main {
    public static void main(String[] args)
    {
        // instance of Integer type
        Test<Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());

        // instance of String type
        Test<String> sObj = new Test<String>("GeeksForGeeks");
        System.out.println(sObj.getObject());
        iObj = sObj; // This results an error
    }
}
```

Output:

error:

incompatible types:

Test cannot be converted to Test

Even though iObj and sObj are of type Test, they are the references to different types because their type parameters differ. **Generics add type safety through this and prevent errors.**

Rules to declare Generic Methods in Java

There are a few rules that we have to adhere to when we need to declare generic methods in Java. Let us look at some of them.



1. You should explicitly specify type parameters before the actual name of the return type of the method. The type parameter is delimited by angle brackets. Example: <T>
2. You should also include the type parameters in the declaration of the program and separate them with commas. A type parameter specifies a generic type name in the method.
3. All the type parameters are only useful when they have to declare reference types. Remember that the type parameters cannot denote primitive data types such as int, float, String etc.
4. The type parameters come in handy when the programmer has to specify the return type of the variables. Note that the parameters are also generic. The type parameters are also useful for denoting the type of the variables in the function parameters. These are actual type parameters.

Java program to understand how we can declare generic methods in Java:

```
package com.dataflair.javagenerics;
public class GenericMethod {
    public <T> void methodgen(T var1)
    {
        System.out.println("The value passed is of type "+var1.getClass().getSimpleName());
    }
    public static void main(String[] args) {
        GenericMethod ob = new GenericMethod();
        ob.<String>methodgen("DataFlair is the best");
        //Sometimes we can omit the explicit mention of the type in <>
        and the compiler can automatically identify the type.
        ob.methodgen("Learning Java at DataFlair");
        ob.methodgen(154);
    }
}
```

Output:

```
The value passed is of type String
The value passed is of type String
The value passed is of type Integer
```

Generic Classes:


A generic class is implemented exactly like a non-generic class. The only difference is that it contains a type parameter section. There can be more than one type of parameter, separated by a comma. The classes, which accept one or more parameters, are known as parameterized classes or parameterized types.

Generic Class

Like C++, we use <> to specify parameter types in generic class creation. To create objects of a generic class, we use the following syntax.

```
// To create an instance of generic class
```

```
BaseType <Type> obj = new BaseType <Type>()
```

 **Note:** In Parameter type we can not use primitives like 'int', 'char' or 'double'.

Example: Java program to show working of user defined Generic classes

```
// We use < > to specify Parameter type
```

```
class Test<T> {  
    // An object of type T is declared  
    T obj;  
    Test(T obj)  
    {  
        this.obj = obj;  
    } // constructor  
    public T getObject()  
    {  
        return this.obj;  
    }  
}
```

```
// Driver class to test above
```

```
class Main {  
    public static void main(String[] args)  
    {
```

```

// instance of Integer type
Test<Integer> iObj = new Test<Integer>(15);
System.out.println(iObj.getObject());

// instance of String type
Test<String> sObj = new Test<String>("Hi,Hello World");
System.out.println(sObj.getObject());
}
}

```

Output:

15

Hi, Hello World

We can also pass multiple Type parameters in Generic classes.

```

// Java program to show multiple type parameters in Java Generics

// We use < > to specify Parameter type
class Test<T, U>
{
    T obj1; // An object of type T
    U obj2; // An object of type U

    // constructor
    Test(T obj1, U obj2)
    {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }

    // To print objects of T and U
    public void print()
    {
        System.out.println(obj1);
        System.out.println(obj2);
    }
}

```

```

}

// Driver class to test above
class Main
{
    public static void main (String[] args)
    {
        Test <String, Integer> obj =new Test<String, Integer>("GfG",
15);

        obj.print();
    }
}

```

Output

GfG

15

Generics with Interfaces:

Generic Interfaces in Java are the interfaces that deal with abstract data types. Interface help in the independent manipulation of java collections from representation details. They are used to achieving **multiple inheritance** in java forming hierarchies. They differ from the java class. These include all abstract methods only, have static and final variables only. The only reference can be created to interface, not objects, Unlike class, these don't contain any constructors, instance variables. This involves the "implements" keyword. These are similar to generic classes.

The benefits of Generic Interface are as follows:



1. This is implemented for different data types.
2. It allows putting constraints i.e. bounds on data types for which interface is implemented.

Syntax:

interface interface-Name < type-parameter-list >

{

//set of methods and variables

}

Ex:

```
interface Sample<T>
{
    T getType();
}
```

Implementation of an Interface into a Class

class class-name <type-parameter-list> implements interface-name <type-arguments-list>

```
{
    //body
}
```

Example:

```
interface Pair<K,V>
{
    public K getKey();
    public V getValue();
}
class OrderedPair<K,V> implements Pair<K,V>
{
    K key;
    V value;
    OrderedPair(K key,V value)
    {
        this.key=key;
        this.value=value;
    }
    public K getKey()
    {
```

```

        return key;
    }
    public V getValue()
    {
        return value;
    }
}
class TestDemo
{
    public static void main(String args[])
    {
        OrderedPair<String,Integer> pair1=new
OrderedPair<String,Integer>("Mango",90);
        OrderedPair<String,String> pair2=new
OrderedPair<String,String>("Hello","World");
        System.out.println("*****Pair1 values*****");
        System.out.println(pair1.getKey());
        System.out.println(pair1.getValue());
        System.out.println("*****Pair2 values*****");
        System.out.println(pair2.getKey());
        System.out.println(pair2.getValue());
    }
}

```

Output:


```
Microsoft Windows [Version 10.0.19045.2006]
(c) Microsoft Corporation. All rights reserved.

E:\Generics>javac TestDemo.java

E:\Generics>java TestDemo
****Pair1 values****
Mango
90
****Pair2 values****
Hello
World

E:\Generics>
```

Bounded Types with Generics in Java

There may be times when you want to restrict the types that can be used as type arguments in a parameterized type.

For example, a method that operates on numbers might only want to accept instances of Numbers or their subclasses. This is what bounded type parameters are for.

- Sometimes we don't want the whole class to be parameterized. In that case, we can create a Java generics method. Since the constructor is a special kind of method, we can use generics type in constructors too.
- Suppose we want to restrict the type of objects that can be used in the parameterized type. For example, in a method that compares two objects and we want to make sure that the accepted objects are Comparables.
- The invocation of these methods is similar to the unbounded method except that if we will try to use any class that is not Comparable, it will throw compile time error.

How to Declare a Bounded Type Parameter in Java?

1. List the type parameter's name,
2. Along with the extends keyword



3. Followed by its upper bound.

Syntax

<T extends **superClassName**>

Note that, in this context, extends is used in a general sense to mean either “extends” (as in classes). Also, This specifies that T can only be replaced by superClassName or subclasses of superClassName. Thus, a superclass defines an inclusive, upper limit.

```
class Sample <T extends Number>
{
    T data;
    Sample(T data){
        this.data = data;
    }
    public void display() {
        System.out.println("Data value is: "+this.data);
    }
}

public class BoundsExample {
    public static void main(String args[]) {
        Sample<Integer> obj1 = new Sample<Integer>(20);
        obj1.display();
        Sample<Double> obj2 = new Sample<Double>(20.22d);
        obj2.display();
        Sample<Float> obj3 = new Sample<Float>(125.332f);
        obj3.display();
    }
}
```

Output

```
Sample<String> strobj=new Sample<>("Hello");
```

```
Data value is: 20
```

Data value is: 20.22

Data value is: 125.332

Now, if you pass other types as parameters to this class (say, String for example) a compile time error will be generated.

Example

```
public class BoundsExample {
    public static void main(String args[]) {
        Sample<Integer> obj1 = new Sample<Integer>(20);
        obj1.display();
        Sample<Double> obj2 = new Sample<Double>(20.22d);
        obj2.display();
        Sample<String> obj3 = new Sample<String>("Krishna");
        obj3.display();
    }
}
```

Compile time error

BoundsExample.java:16: error: type argument String is not within bounds of type-variable T

```
    Sample<String> obj3 = new Sample<String>("Krishna");
        ^
```

where T is a type-variable:

T extends Number declared in class Sample

BoundsExample.java:16: error: type argument String is not within bounds of type-variable T

```
                Sample<String>        obj3        =        new
Sample<String>("Krishna");                ^
```

where T is a type-variable:

T extends Number declared in class Sample

2 errors

Defining bounded-types for methods

Just like with classes to define bounded-type parameters for generic methods, **specify them after the extends keyword**. If you pass a type which is not sub class of the specified bounded-type an error will be generated.

Example

In the following example we are setting the `Collection<Integer>` type as upper bound to the typed-parameter i.e. this method accepts all the collection objects (of Integer type).

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Iterator;
public class GenericMethod {
    public static <T extends Collection<Integer>> void
sampleMethod(T ele){
    Iterator<Integer> it = ele.iterator();
    while (it.hasNext()) {
        System.out.println(it.next());
    }
}
public static void main(String args[]) {
    ArrayList<Integer> list = new ArrayList<Integer>();
    list.add(24);
    list.add(56);
    list.add(89);
    list.add(75);
    list.add(36);
    sampleMethod(list);
}
}
```

Output

```
24
56
89
75
36
```

Now, if you pass types other than collection as typed-parameter to this method, it generates a compile time error.

Example

```
public class GenericMethod {
    public static <T extends Collection<Integer>> void
sampleMethod(T ele){
    Iterator<Integer> it = ele.iterator();
    while (it.hasNext()) {
        System.out.println(it.next());
    }
}
public static void main(String args[]) {
    ArrayList<Integer> list = new ArrayList<Integer>();
    list.add(24);
    list.add(56);
    list.add(89);
    list.add(75);
    list.add(36);
    sampleMethod(list);
    Integer [] intArray = {24, 56, 89, 75, 36};
    sampleMethod(intArray);
}
}
```

Compile time error

GenericMethod.java:23: error: method sampleMethod in class GenericMethod cannot be applied to given types;

```
sampleMethod(intArray);
```

^

required: T

found: Integer[]

reason: inferred type does not conform to upper bound(s)

inferred: Integer[]

upper bound(s): Collection<Integer>

where T is a type-variable:

T extends Collection<Integer> declared in method
<T>sampleMethod(T)

1 error

USER Defined

Let's take an example of how to implement bounded types (extend superclass) with generics.

Example 1:

```
// This class only accepts type parameters as any class  
// which extends class A or class A itself.  
// Passing any other type will cause compiler time error
```

```
class Bound<T extends A>  
{  
  
    private T objRef;  
  
    public Bound(T obj)  
    {  
        this.objRef = obj;  
    }  
  
    public void doRunTest(){  
        this.objRef.displayClass();  
    }  
}
```

```
class A
{
    public void displayClass()
    {
        System.out.println("Inside super class A");
    }
}
```

```
class B extends A
{
    public void displayClass()
    {
        System.out.println("Inside sub class B");
    }
}
```

```
class C extends A
{
    public void displayClass()
    {
        System.out.println("Inside sub class C");
    }
}
```

```
public class BoundedClass
{
    public static void main(String a[])
    {
```

```
// Creating object of sub class C and passing it to Bound as a
type parameter.
```

```
    Bound<C> bec = new Bound<C>(new C());
    bec.doRunTest();
```

```
//Creating object of sub class B and passing it to Bound as a type
//parameter.
```

```
    Bound<B> beb = new Bound<B>(new B());
    beb.doRunTest();
```

```
    // similarly passing super class A
    Bound<A> bea = new Bound<A>(new A());
    bea.doRunTest();
```

```
    }  
}
```

Output

Inside sub class C

Inside sub class B

Inside super class A

Example2: Now, we are restricted to only type A and its subclasses, So it will throw an error for any other type of subclasses.

```
// This class only accepts type parameters as any class  
// which extends class A or class A itself.  
// Passing any other type will cause compiler time error
```

```
class Bound<T extends A>  
{  
    private T objRef;  
  
    public Bound(T obj){  
        this.objRef = obj;  
    }  
  
    public void doRunTest(){  
        this.objRef.displayClass();  
    }  
}
```

```
class A  
{  
    public void displayClass()  
    {  
        System.out.println("Inside super class A");  
    }  
}
```

```
class B extends A  
{  
    public void displayClass()  
    {
```



```

        System.out.println("Inside sub class B");
    }
}

```

class C extends A

```

{
    public void displayClass()
    {
        System.out.println("Inside sub class C");
    }
}

```

public class BoundedClass

```

{
    public static void main(String a[])
    {
        // Creating object of sub class C and
        // passing it to Bound as a type parameter.
        Bound<C> bec = new Bound<C>(new C());
        bec.doRunTest();

        // Creating object of sub class B and
        // passing it to Bound as a type parameter.
        Bound<B> beB = new Bound<B>(new B());
        beB.doRunTest();

        // similarly passing super class A
        Bound<A> beA = new Bound<A>(new A());
        beA.doRunTest();

        Bound<String> beS = new Bound<String>(new String());
        beS.doRunTest();
    }
}

```

Output :

error: type argument String is not within bounds of type-variable T

Typeparameters with Multiple Bounds

Bounded type parameters can be used with methods as well as classes and interfaces.

Java Generics supports multiple bounds also, i.e., In this case, A can be an interface or class. If A is class, then B and C should be interfaces.

Note: We can't have more than one class in multiple bounds.

Syntax:

<T extends superClassName & Interface >

Example3:

```
class Bound<T extends A & B>
{
    private T objRef;

    public Bound(T obj){
        this.objRef = obj;
    }

    public void doRunTest(){
        this.objRef.displayClass();
    }
}

interface B
{
    public void displayClass();
}

class A implements B
{
    public void displayClass()
    {
        System.out.println("Inside super class A");
    }
}

public class BoundedClass
{
    public static void main(String a[])
    {
        //Creating object of sub class A and
```

```

        //passing it to Bound as a type parameter.
        Bound<A> bea = new Bound<A>(new A());
        bea.doRunTest();
    }
}

```

Output

Inside super class A

Example2

```


public class GenericsTester {
    public static void main(String[] args) {
        System.out.printf("Max of %d, %d and %d is %d\n\n",
            3, 4, 5, maximum( 3, 4, 5 ));

        System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",
            6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));
    }

    public static <T extends Number
        & Comparable<T>> T maximum(T x, T y, T z) {
        T max = x;
        if(y.compareTo(max) > 0) {
            max = y;
        }


        if(z.compareTo(max) > 0) {
            max = z;
        }
        return max;
    }
}

```



```
// Compiler throws error in case of below declaration
/* public static <T extends Comparable<T>
   & Number> T maximum1(T x, T y, T z) {
   T max = x;
   if(y.compareTo(max) > 0) {
       max = y;
   }

   if(z.compareTo(max) > 0) {
       max = z;
   }
   return max;
}*/
}
```



Output

Max of 3, 4 and 5 is 5

Max of 6.6, 8.8 and 7.7 is 8.8

Wildcard Arguments In Java

- **Wildcard arguments** means unknown type arguments. They just act as placeholder for real arguments to be passed while calling method.
- In Java Programming, we can represent the wildcard arguments using the symbol “question mark (?)”.
- The wildcard can be used as the type of a parameter, field, or local variable and sometimes as a return type also.
- One important thing is that the types which are used to declare wildcard arguments must be generic types.
- In Java Programming we can use the Wildcard arguments in three ways.

1. Wildcard Arguments With An Unknown Type(or) unbounded wildcard parameters
2. Wildcard Arguments with An Upper Bound
3. Wildcard Arguments with Lower Bound

1)Wildcard Arguments With An Unknown Type :

→ In Java Programming, we can declare the wildcard parameters as unbounded as follows:

Syntax:

GenericType<?>

Note:The arguments which are declared as above can hold any type of objects.

Example: Collection<?> or ArrayList<?> can hold any type of objects like String, Integer, Double etc.

Example 1:

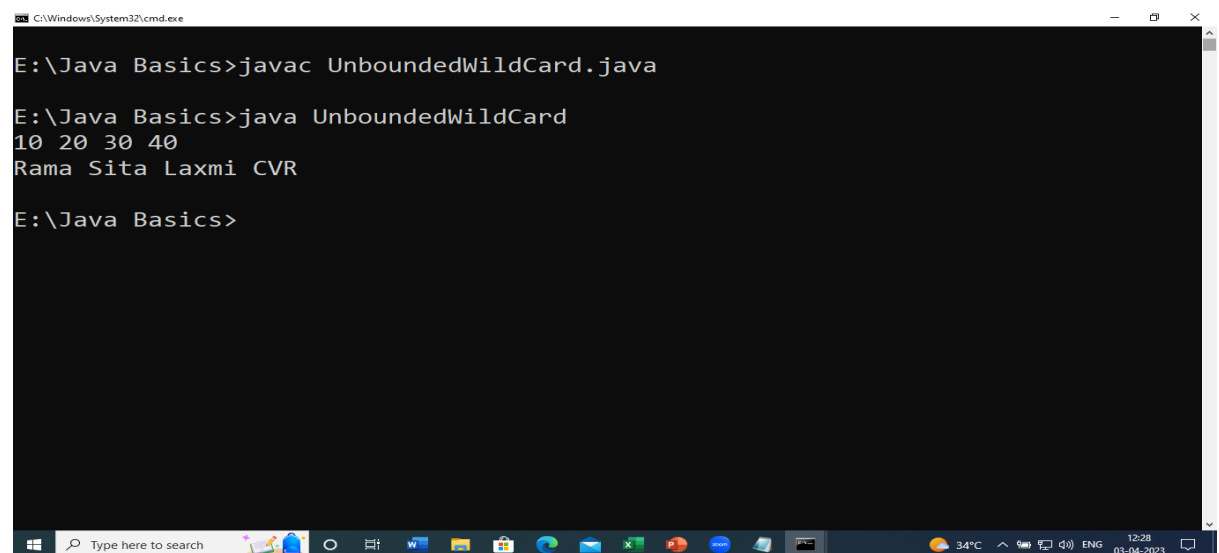
```
import java.util.*;

class UnboundedWildCard
{
    public static void display(List<?> list)
    {
        for(Object obj:list)
        {
            System.out.print(obj+" ");
        }
        System.out.println();
    }

    public static void main(String... A)
    {
        ArrayList<Integer> ilist=new ArrayList<Integer>();
```

```
ilist.add(10);
ilist.add(20);
ilist.add(30);
ilist.add(40);
display(ilist);
ArrayList<String> slist=new ArrayList<String>();
slist.add("Rama");
slist.add("Sita");
slist.add("Laxmi");
slist.add("CVR");
display(slist);
}
}
```

Output:



```
C:\Windows\System32\cmd.exe

E:\Java Basics>javac UnboundedWildCard.java

E:\Java Basics>java UnboundedWildCard
10 20 30 40
Rama Sita Laxmi CVR

E:\Java Basics>
```

Example2:

```
class Stats<T extends Number>
{
```

```
T[] nums;

Stats(T nums[])
{
    this.nums=nums;
}

double average()
{
    double sum=0.0;
    for(int i=0;i<nums.length;i++)
    {
        sum+=nums[i].doubleValue();
    }
    return (sum/nums.length);
}

boolean Sumavg(Stats<?> ob)
{
    if(average()==ob.average())
    {
        return true;
    }
    return false;
}

public static void main(String args[])
```

```

{
    Integer inums[]={1,2,3,4,5};

    Stats<Integer> iobj=new Stats<Integer>(inums);

    double d1=iobj.average();

    Double dnums[]={1.0,2.0,3.0,4.0,5.0};

    Stats<Double> dobj=new Stats<Double>(dnums);

    double d2=dobj.average();

    if(iobj.Sumavg(dobj))

        System.out.println("Same");

    else

        System.out.println("Not Same");

}
}

```

Output:

```

C:\Windows\System32\cmd.exe

E:\Java Basics>javac UnboundedWildcard.java

E:\Java Basics>java UnboundedWildcard
10 20 30 40
Rama Sita Laxmi CVR

E:\Java Basics>javac Stats.java

E:\Java Basics>java Stats
Same

E:\Java Basics>

E:\Java Basics>

```


2)Wildcard Arguments With An Upper Bound :

In the above example, if We want the display() method to work with only numbers, then you can specify an upper bound for wildcard argument. To specify an upper bound for wildcards, use this

Syntax:

GenericType<? extends SuperClass>

Ex: List<? Extends Number>

This specifies that a wildcard argument can contain 'SuperClass' type or its sub classes. Remember that extends clause is an inclusive bound. i.e 'SuperClass' also lies in the bound.

The above processElements() method can be modified to process only numbers like below,

Example:

```
public class GenericsInJava
{
    static void processElements(ArrayList<? extends Number> a)
    {
        for (Object element : a)
        {
            System.out.println(element);
        }
    }

    public static void main(String[] args)
    {
        //ArrayList Containing Integers

        ArrayList<Integer> a1 = new ArrayList<>();

        a1.add(10);

        a1.add(20);
```

```

a1.add(30);

processElements(a1);

//Arraylist containing Doubles

ArrayList<Double> a2 = new ArrayList<>();

a2.add(21.35);

a2.add(56.47);

a2.add(78.12);

processElements(a2);

//Arraylist containing Strings

ArrayList<String> a3 = new ArrayList<>();

a3.add("One");

a3.add("Two");

a3.add("Three");

//This will not work

processElements(a3);    //Compile time error
}
}

```

3.Wildcard Arguments With Lower Bound :

We can also specify a lower bound for wildcard argument using “**super** clause(or) super keyword”.

Syntax:

GenericType<? super SubClassname>

Ex:

(1) List<? Super Integer>

Note: This means that a wildcard argument can contain the actual values of type 'SubClass' type or its super classes.

```
public class GenericsInJava
{
    static void processElements(ArrayList<? super Integer> a)
    {
        for (Object element : a)
        {
            System.out.println(element);
        }
    }

    public static void main(String[] args)
    {
        //ArrayList Containing Integers

        ArrayList<Integer> a1 = new ArrayList<>();

        a1.add(10);

        a1.add(20);

        a1.add(30);

        processElements(a1);

        //Arraylist containing Doubles

        ArrayList<Double> a2 = new ArrayList<>();

        a2.add(21.35);

        a2.add(56.47);
```

```

a2.add(78.12);
//This will not work

processElements(a2);    //Compile time error
}
}

```



Note : 'super' clause is used to specify the lower bound for only wildcard arguments. It does not work with bounded types.

Generics with Inheritance & Subtypes

Inheritance with Generics

To implement inheritance with generic classes we need to follow the certain rules. They are

1.A generic class can extend a non-generic class.i.e

```

class NonGenericClass
{
    //Non Generic Class body
}

class GenericClass<T> extends NonGenericClass
{
    //Generic class extending non-generic class
}

```

2.Generic class can also extend another generic class. When generic class extends another generic class, sub class should have at least same type and same number of type parameters and at most can have any number and any type of parameters.i.e

```

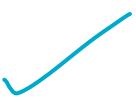

class GenericSuperClass<T>
{
    //Generic super class with one type parameter
}

class GenericSubClass1<T> extends GenericSuperClass<T>
{
    //sub class with same type parameter
}

class GenericSubClass2<T, V> extends
GenericSuperClass<T>
{
    //sub class with two type parameters
}

class GenericSubClass3<T1, T2> extends
GenericSuperClass<T>
{
    //Compile time error, sub class having different type of
    parameters
}

```

3. When generic class extends another generic class, the type parameters are passed from sub class to super class same as in the case of **constructor chaining where super class constructor is called by sub class constructor by passing required arguments. For example, in the below program 'T' in 'GenericSuperClass' will be replaced by String.**

```

class GenericSuperClass<T>
{
    T t;
    public GenericSuperClass(T t)
    {
        this.t = t;
    }
}

class GenericSubClass<T> extends
GenericSuperClass<T>
{

```

```

        public GenericSubClass(T t)
        {
            super(t);
        }
    }

    public class GenericsInJava
    {
        public static void main(String[] args)
        {
            GenericSubClass<String> gen = new
            GenericSubClass<String>("I am string");

            System.out.println(gen.t);    //Output : I am
            string
        }
    }

```

4. A generic class can extend only one generic class and one or more generic interfaces. Then its type parameters should be union of type parameters of generic class and generic interface(s).

```

class GenericSuperClass<T1>
{
    //Generic class with one type parameter
}

interface GenericInterface1<T1, T2>
{
    //Generic interface with two type parameters
}

interface GenericInterface2<T2, T3>
{
    //Generic interface with two type parameters
}

```

```
class GenericClass<T1,T2, T3> extends
GenericSuperClass<T1> implements GenericInterface1<T1,
T2>, GenericInterface2<T2, T3>
{
    //Class having parameters of both the interfaces and
    super class
}
```

5. Non-generic class can't extend generic class except of those generic classes which have already pre defined types as their type parameters. i.e

```
class GenericSuperClass<T>
{
    //Generic class with one type parameter
}
```

```
class NonGenericClass extends GenericSuperClass<T>
{
    //Compile time error, non-generic class can't extend
    generic class
}
```

```
class A
{
    //Pre defined class
}
```

```
class GenericSuperClass1<A>
{
    //Generic class with pre defined type 'A' as type
    parameter
}
```

```
class NonGenericClass1 extends GenericSuperClass1<A>
{
    //No compile time error, It is legal
}
```

6. Non-generic class can extend generic class by removing the type parameters. i.e as a raw type. But, it gives a warning.i.e

```

class GenericClass<T>
{
    T t;

    public GenericClass(T t)
    {
        this.t = t;
    }
}

class NonGenericClass extends GenericClass //Warning
{
    public NonGenericClass(String s)
    {
        super(s); //Warning
    }
}

public class GenericsInJava
{
    public static void main(String[] args)
    {
        NonGenericClass nonGen = new NonGenericClass("I
am String");

        System.out.println(nonGen.t); //Output : I am
String
    }
}

```

7.While extending a generic class having bounded type parameter, type parameter must be replaced by either upper bound or it's sub classes.i.e

```

class GenericSuperClass<T extends Number>
{
    //Generic super class with bounded type parameter
}

class GenericSubClass1 extends
GenericSuperClass<Number>
{

```



```

    //type parameter replaced by upper bound
}

class GenericSubClass2 extends
GenericSuperClass<Integer>
{
    //type parameter replaced by sub class of upper bound
}

class GenericSubClass3 extends GenericSuperClass<T
extends Number>
{
    //Compile time error
}

```

8. Generic methods of super class can be overridden in the sub class like normal methods.i.e

```

class GenericClass
{
    <T> void genericMethod(T t)
    {
        System.out.println(1);
    }
}

class NonGenericClass extends GenericClass
{
    @Override
    <T> void genericMethod(T t)
    {
        System.out.println(2);
    }
}

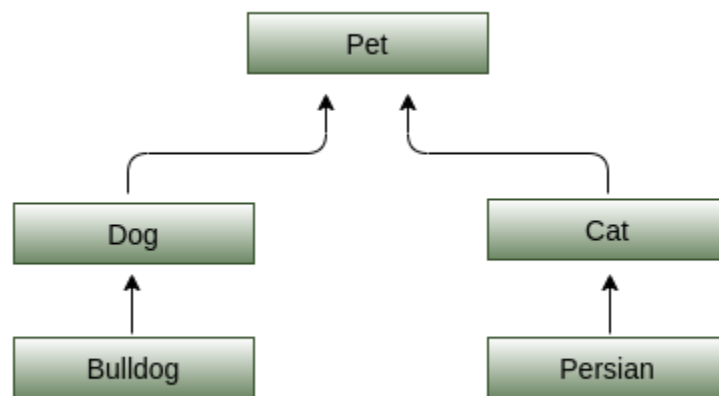
public class GenericsInJava
{
    public static void main(String[] args)
    {
        new GenericClass().genericMethod("I am String");
    }
}
//Output : 1

```

```
        new NonGenericClass().genericMethod("I am String");  
    //Output : 2  
    }  
}
```

Generic Inheritance and Subtypes

To explain Generic inheritance, subtypes and wildcards, we use following class hierarchy:



The Pet, Bulldog and Bulldog classes are as below. Cat and Persian classes are similar.

```
public class Pet { }
```

```
public class Dog extends Pet { }
```

```
public class Bulldog extends Dog { }
```

Example:

// A subclass can add its own type parameters.

```
class MyClass<T> {  
    T ob;                // declare an object of type T  
    public MyClass(T o) {  
        ob = o;  
    }  
    public T getob1() {  
        return ob;  
    }  
}
```

// A subclass that defines a second type parameter, called V.

```
class MySubclass<T, V> extends MyClass<T> {  
    V ob2;  
    public MySubclass(T o, V o2) {  
        super(o);  
        ob2 = o2;  
    }  
    public V getob2() {  
        return ob2;  
    }  
}  
  
public class Main {  
    public static void main(String args[]) {  
        MySubclass<String, Integer> x =
```

```

        new MySubclass<String, Integer>("Value is: ", 99);
        System.out.print(x.getob1());
        System.out.println(x.getob2());
    }
}

```

Java Generics Example Class Hierarchy

The Pet, Bulldog and Bulldog classes are as below. Cat and Persian classes are similar.

```

public class Pet { }

public class Dog extends Pet { }

public class Bulldog extends Dog { }

```

Create an instance of Box<Pet> and put Pet, Dog and Bulldog, and it accepts Pet and all its subtypes. i.e

//Java Program to describe representation of subtypes with
//generics

```

    public class Box<T> {
        private T pet;
        public void put(final T pet) {
            this.pet = pet;
        }
    }
    Public class BoxDemo
    {
        Public static void main(String args[])
        {
            Box<Pet> petBox = new Box<>();
            petBox.put(new Bulldog());
            petBox.put(new Dog());
            petBox.put(new Pet());
            Box<Dog> dogBox = new Box<>();
            dogBox.put(new Bulldog());
            dogBox.put(new Dog());
            dogBox.put(new Pet());           // Error
        }
    }

```

```
}  
}
```



Note: Substitution Principle allows us to add subtypes to a list. We can add Bulldog, Dog and Pet to Box<Pet> as all are subtypes of Pet. However, the Box<Dog> accepts only the Dog and its subtype Bulldog but not the supertype Pet.

Substitution Principle: A variable of a given type may be assigned a value of any subtype of that type, and a method with a parameter of a given type may be invoked with an argument of any subtype of that type.

As expected, inheritance in put(T t) is similar to plain types such as put(Pet pet). Now, let's see whether same holds true with generic subtypes.

Generic Subtype and Type Compatibility

As we do with plain types, we can subtype a generic class or interface by extending or implementing it. The Collection classes from Java library is a good example of this.

Java Generics Collection

The ArrayList<E> implements List<E> which in turn extends Collection<E>. Similarly, the HashSet<E> implements Set<E> which extends Collection<E>.

We can assign an instance Bulldog or Dog to Pet variable as they are regular types i.e. non generic types.

```
Pet pet = new Pet();
```

```
pet = new Dog();
```

```
pet = new Bulldog();
```

Whether same holds good with the generic types, i.e can we assign a `List<E>` to `Collection<E>`? Answer is as long as type arguments are same, the inheritance is allowed.

Java Generics Collection Subtype

```
ArrayList<Dog> dogs = new ArrayList<>();
```

```
List<Dog> dogList = dogs;
```

```
HashSet<Dog> dogHashSet = new HashSet<>();
```

```
Set<Dog> dogSet = dogHashSet;
```

```
Collection<Dog> dogCollection = dogs;
```

```
dogCollection = dogList;
```

```
dogCollection = dogHashSet;
```

```
dogCollection = dogSet;
```

In above example the type argument `Dog` is used in all the parameterized types and we can assign generic subtypes (`ArrayList` or `List`) to its supertype (`Collection`). Same way, we can assign `HashSet<Dog>` or `Set<Dog>` to `Collection<Dog>`.

The Substitution Principle does not work with the parameterized types with different type arguments. When type arguments are different, the rules are quite strict. Forget subtypes even the parameterized types of same type with different type arguments are not compatible with each other. Consider the following

```
List<Dog> dogs = new ArrayList<>();
```

```
List<Bulldog> bulldogs = dogs; // not allowed, error
```

```
List<Pet> pets = dogs; // not allowed, error
```

When we try to assign List<Dog> to List<Pet> compiler throws error

Type mismatch: cannot convert from List<Dog> to List<Pet>

Let's consider another example to understand its significance.

```
List<Bulldog> bulldogs = new ArrayList<>();
```

```
List<Dog> dogs = new ArrayList<>();
```

```
dogs = bulldogs; // error
```

```
bulldogs = dogs; // error
```

The List<Bulldog> is not assignable to List<Dog>: reason begin, the original intent with List<Bulldog> is that

it should hold only the Bulldog and nothing else. However, when it is assigned to List<Dog> JVM has to allow it

to hold both Bulldog and Dog and this messes the list. Hence, they are not compatible. Similarly, List<Dog> is not

assignable to List<Bulldog>: the List<Dog> may contain Dog and its subtype Bulldog and we can't assign it

List<Bulldog> as the later can hold only the list with Bulldog but not the list with Dog and Bulldog.

We can use another way to remember this important aspect of generics. The `List<Dog>`, `ArrayList<Dog>` and

`Collection<Dog>` can hold `Dog` and its subtype `Bulldog`. As all three can hold similar items we can assign

`ArrayList<Dog>` to `List<Dog>`, and also `ArrayList<Dog>` or `List<Dog>` to `Collection<Dog>`. However, the `List<Pet>`

can hold `Pet`, `Dog` and `Bulldog` and `List<Dog>` can hold `Dog` and `Bulldog`. As they can hold different types they are

not similar and we can assign one to another. Same reasoning applies to the `List<Pet>`, `ArrayList<Dog>` or any other

combination.

Following diagram summarizes these two rules. In the LHS hierarchy, the all type arguments are `Dog` and inheritance works as expected. In the RHS hierarchy, all type arguments are `Bulldog` and again, inheritance works as expected. But, the left side is not compatible with right side as type arguments are different.

Java Generics Collection Subtype

Same is true while calling the method that has generic types as parameters. In the following example, we can't pass `List<Bulldog>` to the method as its parameter is `List<Dog>`. Again, `List<Bulldog>` is not compatible with `List<Dog>` even though `Bulldog` is subtype of `Dog`.

```
public void bark(List<Dog> s) {  
    ...  
}
```

```
List<Dog> dogs = new ArrayList<>();
```

```
List<Bulldog> bulldog = new ArrayList<>();
```



```
bark(dog);
```

```
bark(bulldog);          // Error
```

This is a one of the important aspects while working with generics subtypes. Let's go through one more example to drive home the point.

```
List<Object> objects = new ArrayList<>();
```

```
List<String> strings = new ArrayList<>();
```

```
objects.add(new Object());          // allowed
```

```
objects.add(new String("hello"));   // allowed
```

```
strings = objects;                  // error
```

String is subclass of Object and we can add an object or a string to List<Object>; however, there is no relationship whatsoever between the two lists - List<Object> and List<String>. That is to say, the List<Object> can hold objects of type Object as well as instances of any Java type as every class is subclass of Object, but we can't assign list of another type to it.

Summary

List<Pet> can hold a Pet, Dog or Bulldog, but we can't assign List<Bulldog> or List<Dog> to List<Pet>.

when type arguments are same: we can assign generic subtypes to supertype variable.

For example, ArrayList<Dog> is assignable to List<Dog> or Collection<Dog>. Similarly, we can assign HashSet<Dog> or Set<Dog> to Collection<Dog>.

when type arguments are different: generics of same type are not compatible with each other.

The `List<Bulldog>` is not subtype of `List<Dog>`, even though `Bulldog` is subtype of `Dog`. There is no relationship between these lists and `List<Bulldog>` is not assignable to `List<Dog>`.

when type arguments are different: inheritance ceases to exist.

The `ArrayList<Bulldog>` is not subtype of `List<Dog>`, even though `Bulldog` is subtype of `Dog` and `ArrayList` is subtype of `List`. The `ArrayList<Bulldog>` is not assignable to `List<Dog>` or `Collection<Dog>`. Same is true for any other combination.

We can overcome these restrictions with wildcards.

Before getting into generic subtype, let's see how inheritance works with generic types. I know this is trivial for those who have already worked with generic types; nevertheless, let's go through it to avoid confusion that kicks-in when we deal with generic subtypes.

Create an instance of `Box<Pet>` and put `Pet`, `Dog` and `BullDog`, and it accepts `Pet` and all its subtypes.

```
public class Box<T> {  
    private T pet;  
  
    public void put(final T pet) {  
        this.pet = pet;  
    }  
}  
  
Box<Pet> petBox = new Box<>();  
petBox.put(new Bulldog());  
petBox.put(new Dog());  
petBox.put(new Pet());  
  
Box<Dog> dogBox = new Box<>();  
dogBox.put(new Bulldog());  
dogBox.put(new Dog());  
dogBox.put(new Pet());           // Error
```

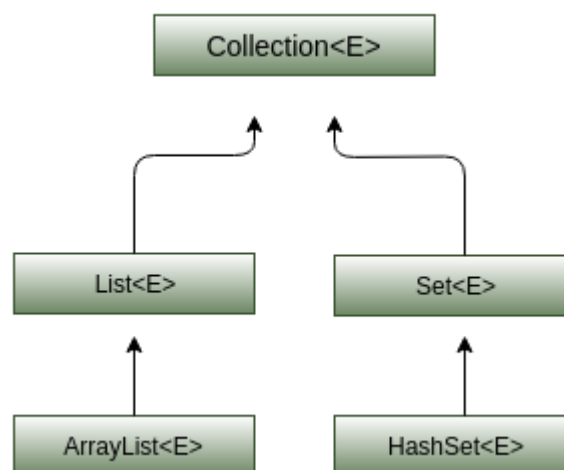
Substitution Principle allows us to add subtypes to a list. We can add Bulldog, Dog and Pet to Box<Pet> as all are subtype of Pet. To Box<Dog> accepts only the Dog and its subtype Bulldog but not the supertype Pet.

Substitution Principle: A variable of a given type may be assigned a value of any subtype of that type, and a method with a parameter of a given type may be invoked with an argument of any subtype of that type.

As expected, inheritance in put(T t) is similar to plain types such as put(Pet pet). Now, let's see whether same holds true with generic subtypes.

Generic Subtype and Type Compatibility

As we do with plain types, we can subtype a generic class or interface by extending or implementing it. The Collection classes from Java library is a good example of this.

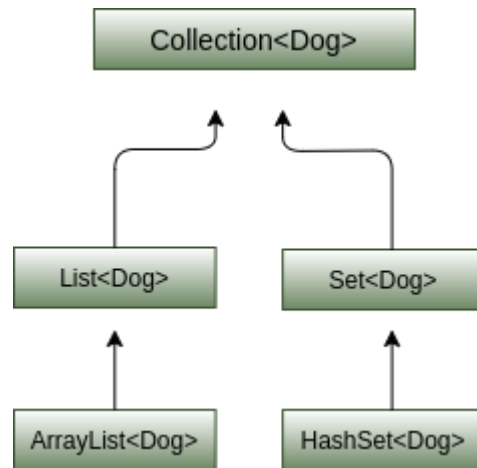


- The ArrayList<E> implements List<E> which extends Collection<E>. Similarly, the HashSet<E> implements Set<E> which extends Collection<E>.
- In plain types we can assign an instance Bulldog or Dog to Pet variable.

```
Pet pet = new Pet();  
pet = new Dog();  
pet = new Bulldog();
```

Whether same holds good with the generic types, i.e can we assign a List<E> to Collection<E>.

As long as type arguments are same, the inheritance is allowed.



```
ArrayList<Dog> dogs = new ArrayList<>();  
List<Dog> dogList = dogs;
```

```
HashSet<Dog> dogHashSet = new HashSet<>();  
Set<Dog> dogSet = dogHashSet;
```

```
Collection<Dog> dogCollection = dogs;  
dogCollection = dogList;  
dogCollection = dogHashSet;  
dogCollection = dogSet;
```

In all parameterized types, the type argument is Dog and we can assign generic subtypes (ArrayList or List) to its supertype (Collection). Same way, we can assign HashSet<Dog> or Set<Dog> to Collection<Dog>.

When type arguments are different, the rules are quite strict. Forget subtypes even same parameterized types with different type arguments are not compatible with each other. Consider the following

```
List<Dog> dogs = new List<>();  
List<Bulldog> bulldogs = dogs; // not allowed, error  
List<Pet> pets = dogs; // not allowed, error
```

When we try to assign `List<Dog>` to `List<Pet>` compiler throws error

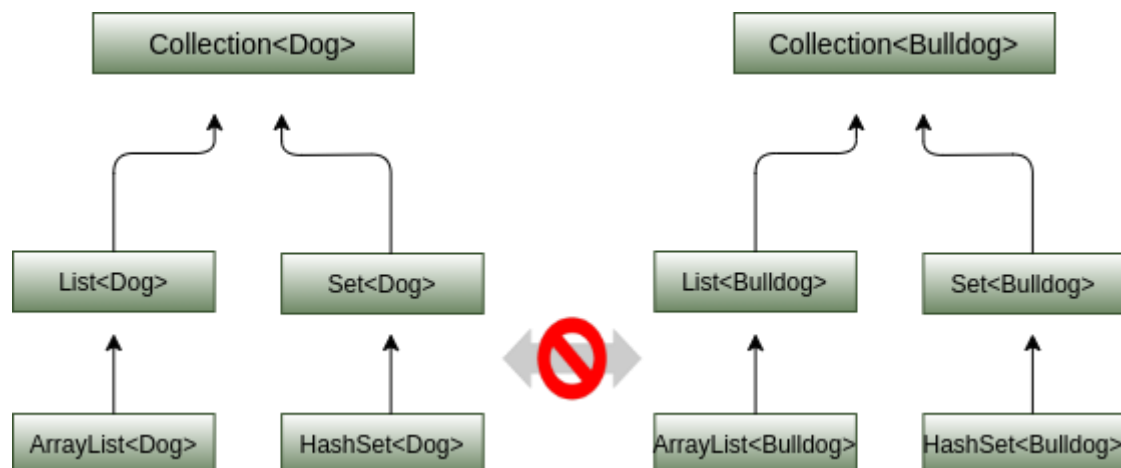
- Type mismatch: cannot convert from `List<Dog>` to `List<Pet>`

Let's consider another example to understand its significance.

```
List<Bulldog> bulldogs = new List<>();  
List<Dog> dogs = bulldogs; // not allowed, error
```

Why `List<Bulldog>` is not assignable to `List<Dog>`: the original intent with `List<Bulldog>` is that it should hold only the `Bulldog` and nothing else. However, when it is assigned to `List<Dog>`, JVM has to allow it to hold both `Bulldog` and `Dog` objects and this messes the list. Hence, they are not compatible.

Following diagram summarizes these two rules. In the LHS hierarchy, the parameter types are all `Dog` and inheritance works as expected. In the RHS hierarchy, the parameter types are all `Bulldog` and again, inheritance works as expected. But, the left side is not compatible with right side as parameter types are different.



Same is true while calling the methods that has generic types as parameters. In the following example, we can't pass `List<Bulldog>` to the method as its parameter is `List<Dog>`. Again, `List<Bulldog>` is not compatible with `List<Dog>` even though `Bulldog` is subtype of `Dog`.

```

public void bark(List<Dog> s) {
    ...
}

List<Dog> dogs = new ArrayList<>();
List<Bulldog> bulldog = new ArrayList<>();
bark(dog);
bark(bulldog);           // Error

```

This is a one of the important aspects while working with generics subtypes. Let's go through one more example to drive home the point.

```

List<Object> objects = new ArrayList<>();
List<String> strings = new ArrayList<>();

objects.add(new Object());           // allowed
objects.add(new String("hello"));    // allowed

strings = objects;                   // error

```

String is subclass of Object, and we can add an object or a string to List<Object>; however, there is no relationship whatsoever between the two lists - List<Object> and List<String>. That is to say, the List<Object> can hold objects of type Object as well as instances of any Java type as every class is subclass of Object, but we can't assign list of another type to it.

Type Inference in Java

Type inference is a feature of Java which provides ability to compiler to look at each method invocation and corresponding declaration to determine the type of arguments.


Java provides improved version of type inference in Java 8. the following example explains, how we can use type inference in our code:

Here, we are creating arraylist by mentioning integer type explicitly at both side. The following approach is used earlier versions of Java.



1. `List<Integer> list = new ArrayList<Integer>();`

In the following declaration, we are mentioning type of arraylist at one side. This approach was introduced in Java 7. Here, you can leave second side as blank diamond and compiler will infer type of it by type of reference variable.



1. `List<Integer> list2 = new ArrayList<>();`

Improved Type Inference

In Java 8, you can call specialized method without explicitly mentioning type of arguments.



1. `showList(new ArrayList<>());`

Java Type Inference Example

We can use type inference with generic classes and methods.

```
import java.util.ArrayList;
import java.util.List;
public class TypeInferenceExample {
    public static void showList(List<Integer>list){
        if(!list.isEmpty()){
            list.forEach(System.out::println);
        }else System.out.println("list is empty");
    }
    public static void main(String[] args) {
        // An old approach(prior to Java 7) to create a list
        List<Integer> list1 = new ArrayList<Integer>();
        list1.add(11);
        showList(list1);
        // Java 7
        List<Integer> list2 = new ArrayList<>(); // You can leave it
        blank, compiler can infer type
        list2.add(12);
        showList(list2);
    }
}
```

```
// Compiler infers type of ArrayList, in Java 8
    showList(new ArrayList<>());
}
}
```

Output:

```
11
12
list is empty
```

We can also create your own custom generic class and methods. In the following example, we are creating our own generic class and method.

Java Type Inference Example 2

```
class GenericClass<X> {
    X name;
    public void setName(X name){
        this.name = name;
    }
    public X getName(){
        return name;
    }
    public String genericMethod(GenericClass<String> x){
        x.setName("John");
        return x.name;
    }
}

public class TypeInferenceExample {
    public static void main(String[] args) {
        GenericClass<String> genericClass = new GenericClass<String>();
        genericClass.setName("Peter");
        System.out.println(genericClass.getName());
    }
}
```



```

        GenericClass<String> genericClass2 = new GenericClass<
>();
        genericClass2.setName("peter");
        System.out.println(genericClass2.getName());

        // New improved type inference
        System.out.println(genericClass2.genericMethod(new Ge
nericClass<>()));
    }
}

```

Output:

```

Peter
peter
John

```

Restriction on Generics

1. Cannot Instantiate Generic Types with Primitive Types

Consider the following parameterized type:

```

class Pair<K, V> {
    private K key;
    private V value;
    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }
    // ...
}

```

When creating a Pair object, you cannot substitute a primitive type for the type parameter K or V:

```
Pair<int, char> p = new Pair<>(8, 'a'); // compile-time error
```

- We can substitute only non-primitive types for the type parameters K and V:

```
Pair<Integer, Character> p = new Pair<>(8, 'a');
```

- Note that the Java compiler autoboxes 8 to Integer.valueOf(8) and 'a' to Character('a').
- Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));

2) Cannot Create Instances of Type Parameters

We cannot create an instance of a type parameter. For example, the following code causes a compile-time error:

Example:

```
public static <E> void append(List<E> list) {  
    E elem = new E(); // compile-time error  
    list.add(elem);  
}
```

As a workaround, you can create an object of a type parameter through reflection:

```
public static <E> void append(List<E> list, Class<E> cls) throws  
Exception {  
    E elem = cls.newInstance(); // OK  
    list.add(elem);  
}
```

We can invoke the append() method as follows:

```
List<String> ls = new ArrayList<>();  
  
append(ls, String.class);
```

3)Cannot Declare Static Fields Whose Types are Type Parameters

A class's static field is a class-level variable shared by all non-static objects of the class. Hence, static fields of type parameters are not allowed. Consider the following class:

```
public class MobileDevice<T> {  
  
    private static T os;  
  
    // ...  
  
}
```

If static fields of type parameters were allowed, then the following code would be confused:

```
MobileDevice<Smartphone> phone = new MobileDevice<>();  
  
MobileDevice<Pager> pager = new MobileDevice<>();  
  
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

Because the static field `os` is shared by `phone`, `pager`, and `pc`, what is the actual type of `os`? It cannot be `Smartphone`, `Pager`, and `TabletPC` at the same time. You cannot, therefore, create static fields of type parameters.

4)Cannot Use Casts or instanceof with Parameterized Types

Because the Java compiler erases all type parameters in generic code, you cannot verify which parameterized type for a generic type is being used at runtime:

Example:

```
public static <E> void rtti(List<E> list) {
```

```

    if (list instanceof ArrayList<Integer>) { // compile-time error
        // ...
    }
}

```

The set of parameterized types passed to the `rtti()` method is:

```

S      =      {      ArrayList<Integer>,      ArrayList<String>
LinkedList<Character>, ... }

```

The runtime does not keep track of type parameters, so it cannot tell the difference between an `ArrayList<Integer>` and an `ArrayList<String>`. The most we can do is to use an unbounded wildcard to verify that the list is an `ArrayList`:

Example:

```

public static void rtti(List<?> list) {

    if (list instanceof ArrayList<?>) { // OK; instanceof requires a
reifiable type

        // ...

    }

}

```

Typically, we cannot cast to a parameterized type unless it is parameterized by unbounded wildcards.

For example:

```

List<Integer> li = new ArrayList<>();

List<Number> ln = (List<Number>) li; // compile-time error

```

However, in some cases the compiler knows that a type parameter is always valid and allows the cast.

For example:

```
List<String> l1 = ...;
```

```
ArrayList<String> l2 = (ArrayList<String>)l1; // OK
```

5)Cannot Create Arrays of Parameterized Types

We cannot create arrays of parameterized types. For example, the following code does not compile:

```
List<Integer>[] arrayOfLists = new List<Integer>[2]; // compile-time error
```

The following code illustrates what happens when different types are inserted into an array:

```
Object[] strings = new String[2];
```

```
strings[0] = "hi"; // OK
```

```
strings[1] = 100; // An ArrayStoreException is thrown.
```

If you try the same thing with a generic list, there would be a problem:

```
Object[] stringLists = new List<String>[2]; // compiler error, but pretend it's allowed
```

```
stringLists[0] = new ArrayList<String>(); // OK
```

```
stringLists[1] = new ArrayList<Integer>(); // An  
ArrayStoreException should be thrown,
```

```
// but the runtime can't detect it.
```

If arrays of parameterized lists were allowed, the previous code would fail to throw the desired `ArrayStoreException`.

6)Cannot Create, Catch, or Throw Objects of Parameterized Types

- A generic class cannot extend the Throwable class directly or indirectly. For example, the following classes will not compile:
- `// Extends Throwable indirectly`
- `class MathException<T> extends Exception { /* ... */ } // compile-time error`
- `// Extends Throwable directly`
- `class QueueFullException<T> extends Throwable { /* ... */ } // compile-time error`
- A method cannot catch an instance of a type parameter:

```
public static <T extends Exception, J> void execute(List<J> jobs)
{
    try {
        for (J job : jobs)
            // ...
    } catch (T e) { // compile-time error
        // ...
    }
}
```

We can, however, use a type parameter in a throws clause:

```
class Parser<T extends Exception> {
    public void parse(File file) throws T { // OK
        // ...
    }
}
```

7) Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

A class cannot have two overloaded methods that will have the same signature after type erasure.

```
public class Example {  
    public void print(Set<String> strSet) {}  
    public void print(Set<Integer> intSet) {}  
}
```



The overloads would all share the same classfile representation and will generate a compile-time error.

Type Erasure in Java

Type erasure can be explained as the process of enforcing type constraints only at compile time and discarding the element type information at runtime.

For example:

```
public static <E> boolean containsElement(E [] elements, E  
element){  
    for (E e : elements){  
        if(e.equals(element)){  
            return true;  
        }  
    }  
    return false;  
}
```

The compiler replaces the unbound type E with an actual type of Object:

```
public static boolean containsElement(Object [] elements, Object  
element){
```

```

    for (Object e : elements){
        if(e.equals(element)){
            return true;
        }
    }

    return false;
}

```

Therefore the compiler ensures type safety of our code and prevents runtime errors.

Types of Type Erasure

Type erasure can occur at class (or variable) and method levels.

1. Class Type Erasure

At the class level, the compiler discards the type parameters on the class and replaces them with its **first bound**, or **Object** if the type parameter is unbound.

Let's implement a Stack using an array:

```

public class Stack<E> {
    private E[] stackContent;

    public Stack(int capacity) {
        this.stackContent = (E[]) new Object[capacity];
    }

    public void push(E data) {
        // ..
    }
}

```



```

    public E pop() {
        // ..
    }
}

```

Upon compilation, the compiler replaces the unbound type parameter E with **Object**:

```

public class Stack {
    private Object[] stackContent;

    public Stack(int capacity) {
        this.stackContent = (Object[]) new Object[capacity];
    }

    public void push(Object data) {
        // ..
    }

    public Object pop() {
        // ..
    }
}

```

In a case where the type parameter E is bound:

```

public class BoundStack<E extends Comparable<E>> {
    private E[] stackContent;

    public BoundStack(int capacity) {

```

```
    this.stackContent = (E[]) new Object[capacity];  
}
```

```
public void push(E data) {  
    // ..  
}
```

```
public E pop() {  
    // ..  
}  
}
```

The compiler will replace the bound type parameter E with the first bound class, `Comparable` in this case:

```
public class BoundStack {  
    private Comparable [] stackContent;  
  
    public BoundStack(int capacity) {  
        this.stackContent = (Comparable[]) new Object[capacity];  
    }  
  
    public void push(Comparable data) {  
        // ..  
    }  
}
```

```

    }

    public Comparable pop() {

        // ..

    }

}

```

2. Method Type Erasure

For method-level type erasure, the method's type parameter is not stored but rather converted to its parent type `Object` if it's unbound or it's first bound class when it's bound.

Let's consider a method to display the contents of any given array:

```

public static <E> void printArray(E[] array) {

    for (E element : array) {

        System.out.printf("%s ", element);

    }

}

```

Upon compilation, the compiler replaces the type parameter `E` with `Object`:

```

public static void printArray(Object[] array) {

    for (Object element : array) {

        System.out.printf("%s ", element);

    }

}

```

For a bound method type parameter:

```

public static <E extends Comparable<E>> void printArray(E[]
array) {

    for (E element : array) {

        System.out.printf("%s ", element);

    }

}

```

We'll have the type parameter E erased and replaced with Comparable:

```

public static void printArray(Comparable[] array) {

    for (Comparable element : array) {

        System.out.printf("%s ", element);

    }

}

```

Java Generic Class Ambiguity Errors

- Ambiguity errors occur when erasure causes two seemingly distinct generic declarations to resolve to the same erased type.
- Here is an example that involves method overloading:
- It shows an ambiguity caused by erasure on overloaded methods.

```

class MyGenClass<T, V> {

    T ob1;

    V ob2;

    // These two overloaded methods are ambiguous

    // and will not compile.

    void set(T o) {

```

```

        ob1 = o;
    }

    void set(V o) {

        ob2 = o;
    }
}

```

- Notice that MyGenClass declares two generic types: T and V.
- Inside MyGenClass, an attempt is made to overload set() based on parameters of type T and V.
- This looks reasonable because T and V appear to be different types.
- As MyGenClass is written, there is no requirement that T and V actually be different types.
- For example, it is perfectly correct to create a MyGenClass object as shown here:
- `MyGenClass<String, String> obj = new MyGenClass<String, String>()`
- In this case, both T and V will be replaced by String.
- This makes both versions of set() identical, which is, of course, an error.
- Furthermore, the type erasure of set() reduces both versions to the following:
- `void set(Object o) { // ...`
- Thus, the overloading of set() as attempted in MyGenClass is ambiguous.
- For example, if you know that V will always be some type of Number, you might try to fix MyGenClass by rewriting its declaration as shown here:
- `class MyGenClass<T, V extends Number> { // almost OK!`
- This change causes MyGenClass to compile, and you can even instantiate objects like the one shown here:
- `MyGenClass<String, Number> x = new MyGenClass<String, Number>(); // No ambiguity error`
- This works because Java can accurately determine which method to call.
- However, ambiguity returns when you try this line:

- `MyGenClass<Number, Number> x = new MyGenClass<Number, Number>();` //ambiguity error
- For the preceding example, we should use two separate method names.

Java Lambda Expressions

Syllabus:

Functional Programming: Functional Interfaces – Function, BiFunction, Predicate, and Supplier, Lambda Expression Fundamentals, Block Lambda Expressions, Passing Lambda Expressions as Arguments, Lambda Expressions and Exceptions, Variable Capture, Method References.

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

Functional Interface

- Lambda expression provides implementation of *functional interface*.
- An interface which has only one abstract method is called “functional interface”.
- Java provides an annotation `@FunctionalInterface`, which is used to declare an interface as functional interface.

Why use Lambda Expression

- To provide the implementation of Functional interface.
- Less coding.

Java Lambda Expression Syntax

(argument-list) -> {body}

Java lambda expression is consisted of three components.

Argument-list: It can be empty or non-empty as well(Optional).

Arrow-token: It is used to link arguments-list and body of expression.

Body: It contains expressions and statements for lambda expression.

Lambda Expression Parameters

There are three Lambda Expression Parameters are mentioned below

1. Zero Parameter
2. Single Parameter
3. Multiple Parameters

1.No Parameter Syntax(Zero Parameters)

```
() -> {  
    //Body of no parameter lambda  
};
```

2.One Parameter Syntax(One Parameters)

```
(p1) -> {  
    //Body of single parameter lambda  
};
```

3.Two Parameter Syntax(Multiple Parameters)

```
(p1,p2) -> {  
    //Body of multiple parameter lambda  
};
```

Let's see a scenario where we are not implementing Java lambda expression.
Here, we are implementing an interface without using lambda expression.

Without Lambda Expression

```
interface Drawable{  
    public void draw();
```



```

}
public class WithoutLambdaExpressionExample {
    public static void main(String[] args) {
        int width=10;

        //without lambda, Drawable implementation using anonymous class
        Drawable d=new Drawable(){
            public void draw(){System.out.println("Drawing "+width);}
        };
        d.draw();
    }
}

```

Output:

```
Drawing 10
```

Java Lambda Expression Example

Now, we are going to implement the above example with the help of Java lambda expression.

```

@FunctionalInterface //It is optional
interface Drawable{
    public void draw();
}

public class LambdaExpressionExample2 {
    public static void main(String[] args) {
        int width=10;

        //with lambda
        Drawable d2=()->{
            System.out.println("Drawing "+width);
        };
        d2.draw();
    }
}

```

```
}  
}
```

Output:

Drawing 10

A lambda expression can have zero or any number of arguments. Let's see the examples:

Java Lambda Expression Example: No Parameter

```
interface Sayable{  
    public String say();  
}  
public class LambdaExpressionExample3{  
    public static void main(String[] args) {  
        Sayable s=()->{  
            return "I have nothing to say.";  
        };  
        System.out.println(s.say());  
    }  
}
```

Output:

I have nothing to say.

Java Lambda Expression Example: Single Parameter

```
interface Sayable{  
    public String say(String name);  
}  
  
public class LambdaExpressionExample4{  
    public static void main(String[] args) {
```

```

// Lambda expression with single parameter.
Sayable s1=(name)->{
    return "Hello, "+name;
};
System.out.println(s1.say("Sonoo"));

// You can omit function parentheses
Sayable s2= name ->{
    return "Hello, "+name;
};
System.out.println(s2.say("Sonoo"));
}
}

```

Output:

```

Hello, Sonoo
Hello, Sonoo

```

Java Lambda Expression Example: Multiple Parameters

```

interface Addable{
    int add(int a,int b);
}

public class LambdaExpressionExample5{
    public static void main(String[] args) {

        // Multiple parameters in lambda expression
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));

        // Multiple parameters with data type in lambda expression
        Addable ad2=(int a,int b)->(a+b);
        System.out.println(ad2.add(100,200));
    }
}

```

```
}
```

Output:

```
30  
300
```

Java Lambda Expression Example: with or without return keyword

In Java lambda expression, if there is only one statement, you may or may not use return keyword. You must use return keyword when lambda expression contains multiple statements.

```
interface Addable{  
    int add(int a,int b);  
}
```

```
public class LambdaExpressionExample6 {  
    public static void main(String[] args) {  
  
        // Lambda expression without return keyword.  
        Addable ad1=(a,b)->(a+b);  
        System.out.println(ad1.add(10,20));  
  
        // Lambda expression with return keyword.  
        Addable ad2=(int a,int b)->{  
            return (a+b);  
        };  
        System.out.println(ad2.add(100,200));  
    }  
}
```

Output:

```
30  
300
```

Java Lambda Expression Example: Foreach Loop

```
import java.util.*;

public class LambdaExpressionExample7{
    public static void main(String[] args) {

        List<String> list=new ArrayList<String>();
        list.add("ankit");
        list.add("mayank");
        list.add("irfan");
        list.add("jai");

        list.forEach( (n)->System.out.println(n) );
    }
}
```

Output:

```
ankit
mayank
irfan
jai
```

Java Lambda Expression Example: Multiple Statements

```
@FunctionalInterface
interface Sayable{
    String say(String message);
}

public class LambdaExpressionExample8{
    public static void main(String[] args) {

        // You can pass multiple statements in lambda expression
        Sayable person = (message)-> {
            String str1 = "I would like to say, ";
            String str2 = str1 + message;
        }
    }
}
```

```

        return str2;
    };
    System.out.println(person.say("time is precious."));
}
}

```

Output:

```
I would like to say, time is precious.
```

Java Lambda Expression Example: Creating Thread

You can use lambda expression to run thread. In the following example, we are implementing run method by using lambda expression.

```

public class LambdaExpressionExample9{
    public static void main(String[] args) {

        //Thread Example without lambda
        Runnable r1=new Runnable(){
            public void run(){
                System.out.println("Thread1 is running...");
            }
        };
        Thread t1=new Thread(r1);
        t1.start();
        //Thread Example with lambda
        Runnable r2=()->{
            System.out.println("Thread2 is running...");
        };
        Thread t2=new Thread(r2);
        t2.start();
    }
}

```

Output:

```
Thread1 is running...
Thread2 is running...
```

Java lambda expression can be used in the collection framework. It provides efficient and concise way to iterate, filter and fetch data. Following are some lambda and collection examples provided.

Java Lambda Expression Example: Comparator

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
class Product{
int id;
String name;
float price;
public Product(int id, String name, float price) {
    super();
    this.id = id;
    this.name = name;
    this.price = price;
}
}
public class LambdaExpressionExample10{
public static void main(String[] args) {
    List<Product> list=new ArrayList<Product>();

    //Adding Products
    list.add(new Product(1,"HP Laptop",25000f));
    list.add(new Product(3,"Keyboard",300f));
    list.add(new Product(2,"Dell Mouse",150f));

    System.out.println("Sorting on the basis of name...");

    // implementing lambda expression
```

```

        Collections.sort(list,(p1,p2)->{
            return p1.name.compareTo(p2.name);
        });
        for(Product p:list){
            System.out.println(p.id+" "+p.name+" "+p.price);
        }
    }
}

```

Output:

```

Sorting on the basis of name...
2 Dell Mouse 150.0
1 HP Laptop 25000.0
3 Keyboard 300.0

```

Java Lambda Expression Example: Filter Collection Data

```

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        super();
        this.id = id;
        this.name = name;
        this.price = price;
    }
}

public class LambdaExpressionExample11{
    public static void main(String[] args) {
        List<Product> list=new ArrayList<Product>();
        list.add(new Product(1,"Samsung A5",17000f));
        list.add(new Product(3,"Iphone 6S",65000f));
    }
}

```



```

list.add(new Product(2,"Sony Xperia",25000f));
list.add(new Product(4,"Nokia Lumia",15000f));
list.add(new Product(5,"Redmi4 ",26000f));
list.add(new Product(6,"Lenevo Vibe",19000f));

// using lambda to filter data
Stream<Product> filtered_data = list.stream().filter(p -
> p.price > 20000);

// using lambda to iterate through collection
filtered_data.forEach(
    product -
    > System.out.println(product.name+": "+product.price)
);
}
}

```

Output:

```

Iphone 6S: 65000.0
Sony Xperia: 25000.0
Redmi4 : 26000.0

```

Lambda Expression as arguments

A **lambda expression** passed in a method that has an argument of type of **functional interface**.

If we need to pass a lambda expression as an argument, the type of parameter receiving the lambda expression argument must be of a functional interface type.

Example 1: Define lambda expressions as method parameters

```
import java.util.ArrayList;
```

```

class Main {
    public static void main(String[] args) {
        // create an ArrayList
        ArrayList<String> languages = new ArrayList<>();

        // add elements to the ArrayList
    }
}

```

```

languages.add("java");
languages.add("swift");
languages.add("python");
System.out.println("ArrayList: " + languages);

// pass lambda expression as parameter to replaceAll() method
languages.replaceAll(e -> e.toUpperCase());
System.out.println("Updated ArrayList: " + languages);
}
}

```

Example2:

In the below example, the lambda expression can be passed in a method which argument's type is "**TestInterface**".

```

interface TestInterface {
    boolean test(int a);
}
class Test {
    // lambda expression can be passed as first argument in the check() method
    static boolean check(TestInterface ti, int b) {
        return ti.test(b);
    }
}
public class LambdaExpressionPassMethodTest {
    public static void main(String arg[]) {
        boolean result = Test.check((x) -> (x%2) == 0, 10);
        System.out.println("The result is: "+ result);
    }
}

```

Output

The result is: true

Example3:

```

interface Test1 {
    void print();
}

class CVR {
    // functional interface parameter is passed
    static void fun(Test1 t) { t.print(); }
    public static void main(String[] args)

```

```

{
    // lambda expression is passed
    // without parameter to functional interface t
    fun(() -> System.out.println("Hello"));
}
}

```

Lambda Expressions and Exceptions

A **lambda expression body** can't throw any exceptions that haven't specified in a **functional interface**.

If the **lambda expression** can **throw** an exception then the **"throws"** clause of a **functional interface** must declare the same exception or one of its supertype.

Example

```

interface Student {
    void studentData(String name) throws Exception;
}
public class LambdaExceptionTest {
    public static void main(String[] args) {
        // lambda expression
        Student student = name -> {
            System.out.println("The Student name is: " + name);
            throw new Exception();
        };
        try {
            student.studentData("Adithya");
        } catch (Exception e) {
        }
    }
}

```

Output

The Student name is: Adithya

Variable Capture in Lambda Expression

- Variable defined by the enclosing scope of a lambda expression are accessible within the lambda expression.

- For example, a lambda expression can use an instance or static variable defined by its enclosing class.
- A lambda expression also has access to (both explicitly and implicitly), which refers to the invoking instance of the lambda expression's enclosing class. Thus, a lambda expression can obtain or set the value of an intrinsic or static variable and call a method defined by its enclosing class.
- When a lambda expression uses an assigned local variable from its enclosing space there's a crucial restriction. **A lambda expression may only use a local variable whose value doesn't change. That restriction is referred to as "variable capture" which is described as; lambda expression capture values, not variables.**
- The local variables that a lambda expression may use are referred to as "effectively final variables".
- An effectively final variable is one whose value doesn't change after it's first assigned. There is no need to explicitly declare such a variable as final, although doing so would not be an error.

- **Example1:**

// Java Program Illustrating Difference between Effectively final and Mutable Local Variables

```
import java.io.*;
interface MyFunction {

    // Method inside the interface
    int func(int n);
}

class Main {

    public static void main(String[] args)
    {

        // Custom local variable that can be captured
        int number = 10;

        MyFunction myLambda = (n) ->
        {

            // This use of number is OK It does not modify num
            int value = number + n;
```

```

    // However, the following is illegal because it attempts to modify the
    //value of number

    // number++;
    return value;
};

// The following line would also cause an error,
// because it would remove the effectively final
// status from num.
    number = 9; //error

    System.out.println("GFG!");
}
}

```

Example 2: Java Program Illustrating Difference between Effectively final and Mutable Local Variables

```

// Importing input output classes
import java.io.*;

// Interface
interface MyInterface {

    // Method inside the interface
    void myFunction();
}

// Main class
class Main {

    // Custom initialization
    int data = 170;

    // Main driver method
    public static void main(String[] args)
    {

        // Creating object of this class inside the main() method
        GFG gfg = new GFG();
    }
}

```

```

// Creating object of interface
// inside the main() method
MyInterface intFace = () ->
{
    System.out.println("Data : " + gfg.data);
    gfg.data += 500;

    System.out.println("Data : " + gfg.data);
};

intFace.myFunction();
gfg.data += 200;

System.out.println("Data : " + gfg.data);
}
}

```

Output

```

Data : 170
Data : 670
Data : 870

```

Note: It is important to emphasize that a lambda expression can use and modify an instance variable from its invoking class. It just can't use a local variable of its enclosing scope unless that variable is effectively final.

Method References

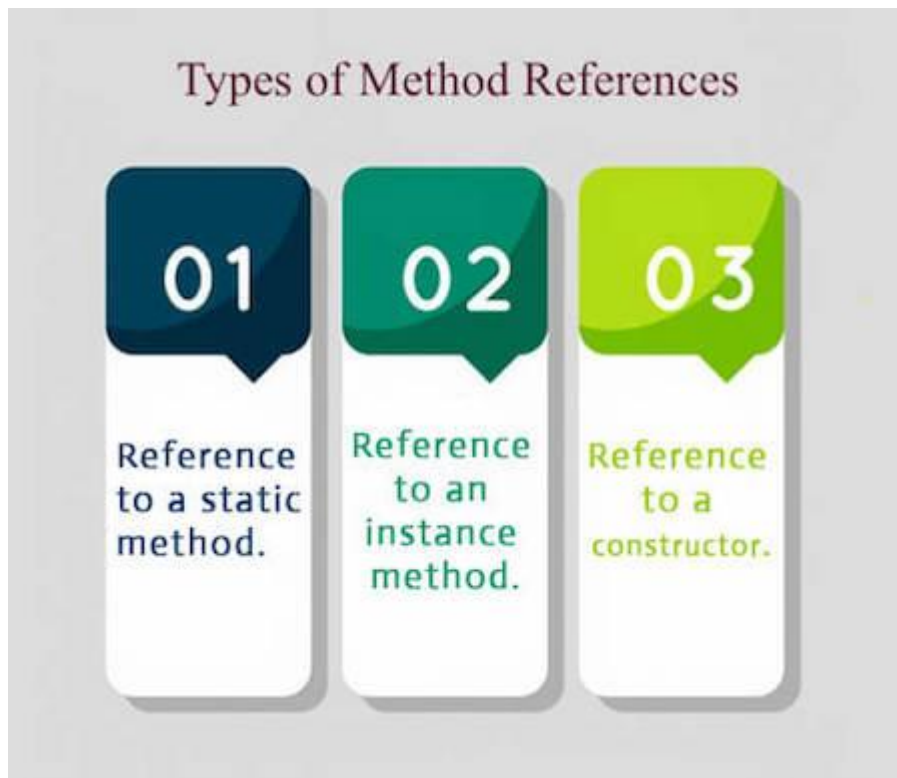
- Java provides a new feature called method reference in Java 8.
- Method reference is used to refer method of functional interface.
- It is compact and easy form of lambda expression.
- Each time when we are using lambda expression to just referring a method, we can replace our lambda expression with method reference.

Types of Method References

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.

3. Reference to a constructor.



1.Reference to a Static Method

You can refer to static method defined in the class. Following is the syntax and example which describe the process of referring static method in Java.

Syntax

ContainingClass::staticMethodName

Example 1

In the following example, we have defined a functional interface and referring a static method to it's functional method say().

```
interface Sayable{
    void say();
}
public class MethodReference {
    public static void saySomething(){
        System.out.println("Hello, this is static method.");
    }
}
```

```

    }
    public static void main(String[] args) {
        // Referring static method
        Sayable sayable = MethodReference::saySomething;
        // Calling interface method
        sayable.say();
    }
}

```

Output:

```

Hello, this is static method.

```

Example 2

In the following example, we are using predefined functional interface Runnable to refer static method.

```

public class MethodReference2 {
    public static void ThreadStatus(){
        System.out.println("Thread is running...");
    }
    public static void main(String[] args) {
        Thread t2=new Thread(MethodReference2::ThreadStatus);
        t2.start();
    }
}

```

Output:

```

Thread is running...

```

Example 3

You can also use predefined functional interface to refer methods. In the following example, we are using BiFunction interface and using it's apply() method.


```

import java.util.function.BiFunction;
class Arithmetic {
public static int add(int a, int b){
return a+b;
}
}

public class MethodReference3 {
public static void main(String[] args) {
BiFunction<Integer, Integer, Integer>adder = Arithmetic::add;
int result = adder.apply(10, 20);
System.out.println(result);
}
}

```

Output:

```
30
```

Example 4

You can also override static methods by referring methods. In the following example, we have defined and overloaded three add methods.

```

import java.util.function.BiFunction;
class Arithmetic {
public static int add(int a, int b){
return a+b;
}

public static float add(int a, float b){
return a+b;
}

public static float add(float a, float b){
return a+b;
}
}

public class MethodReference4 {
public static void main(String[] args) {

```

```

BiFunction<Integer, Integer, Integer>adder1 = Arithmetic::add;
BiFunction<Integer, Float, Float>adder2 = Arithmetic::add;
BiFunction<Float, Float, Float>adder3 = Arithmetic::add;
int result1 = adder1.apply(10, 20);
float result2 = adder2.apply(10, 20.0f);
float result3 = adder3.apply(10.0f, 20.0f);
System.out.println(result1);
System.out.println(result2);
System.out.println(result3);
}
}

```

Output:

```

30
30.0
30.0

```

2) Reference to an Instance Method

like static methods, you can refer instance methods also. In the following example, we are describing the process of referring the instance method.

Syntax

containingObject::instanceMethodName

Example 1

In the following example, we are referring non-static methods. You can refer methods by class object and anonymous object.

```

interface Sayable{
    void say();
}
public class InstanceMethodReference {
    public void saySomething(){
        System.out.println("Hello, this is non-static method.");
    }
}

```

```

    }
    public static void main(String[] args) {
InstanceMethodReference methodReference = new InstanceMethodReferenc
e(); // Creating object
    // Referring non-static method using reference
    Sayable sayable = methodReference::saySomething;
    // Calling interface method
    sayable.say();
    // Referring non-static method using anonymous object
    Sayable sayable2 = new InstanceMethodReference()::saySomething;
// You can use anonymous object also
    // Calling interface method
    sayable2.say();
    }
}

```

Output:

```

Hello, this is non-static method.
Hello, this is non-static method.

```

Example 2

In the following example, we are referring instance (non-static) method. Runnable interface contains only one abstract method. So, we can use it as functional interface.

```

public class InstanceMethodReference2 {
    public void printnMsg(){
        System.out.println("Hello, this is instance method");
    }
    public static void main(String[] args) {
        Thread t2=new Thread(new InstanceMethodReference2()::printnMsg);
        t2.start();
    }
}

```

Output:

```
Hello, this is instance method
```

Example 3

In the following example, we are using BiFunction interface. It is a predefined interface and contains a functional method apply(). Here, we are referring add method to apply method.

```
import java.util.function.BiFunction;
class Arithmetic{
public int add(int a, int b){
return a+b;
}
}
public class InstanceMethodReference3 {
public static void main(String[] args) {
BiFunction<Integer, Integer, Integer>adder = new Arithmetic()::add;
int result = adder.apply(10, 20);
System.out.println(result);
}
}
```

Output:

```
30
```

3) Reference to a Constructor

You can refer a constructor by using the new keyword. Here, we are referring constructor with the help of functional interface.

Syntax

ClassName::new

Example

```

interface Messageable{
    Message getMessage(String msg);
}
class Message{
    Message(String msg){
        System.out.print(msg);
    }
}
public class ConstructorReference {
    public static void main(String[] args) {
        Messageable hello = Message::new;
        hello.getMessage("Hello");
    }
}

```

Output:

```
Hello
```

Functional Interfaces – Function, BiFunction, Predicate, and Supplier(SAM Interfaces)

Functional programming is a paradigm that allows programming using expressions i.e. declaring functions, passing functions as arguments, and using functions as statements. In Java 8's several functional interfaces are introduced by Java.

Consumer

A Consumer is an in-build functional interface in the java.util.function package. we use consumers when we need to consume objects, the consumer takes an input value and returns nothing. The consumer interface has two methods.

```
void accept(T value);
default Consumer<T> andThen(Consumer<? super T> after);
```

Here, we printing the cities by creating a consumer. passing city in consumer as argument and printing.

```
@Test
public void printCities() {

    List<String> cities = new ArrayList<>();
    cities.add("Delhi");
    cities.add("Mumbai");
    cities.add("Goa");
    cities.add("Pune");

    Consumer<String> printConsumer= city-> System.out.println(city);
    cities.forEach(printConsumer);
}
```

Predicate

A Predicate is a functional interface, which accepts an argument and returns a boolean. Usually, it is used to apply in a filter for a collection of objects.

```
boolean test(T value);
default Predicate<T> and(Predicate<? super T> other);
default Predicate<T> negate();
default Predicate<T> or(Predicate<? super T> other);
static <T> Predicate<T> isEqual(Object targetRef);
static <T> Predicate<T> not(Predicate<? super T> target);
```

Here, we filter cities according to city name by a predicate and print it on the console.

```
@Test
public void filterCities() {

    List<String> cities = new ArrayList<>();
    cities.add("Delhi");
    cities.add("Mumbai");
    cities.add("Goa");
    cities.add("Pune");

    Predicate<String> filterCity = city -> city.equals("Mumbai");
```

```
cities.stream().filter(filterCity).forEach(System.out::println);
}
```

Function

A Function is another in-build functional interface in java.util.function package, the function takes an input value and returns a value. The function interface has four methods, mostly function used in map feature of stream APIs.

```
R apply(T var1);
default <V> Function<V, R> compose(Function<V, T> before);
default <V> Function<T, V> andThen(Function<R, V> after);
static <T> Function<T, T> identity();
```

We take an example, here we passing city's as argument and returning the first character of cities.

```
@Test
public void mapCities() {

    List<String> cities = new ArrayList<>();
    cities.add("Delhi");
    cities.add("Mumbai");
    cities.add("Goa");
    cities.add("Pune");

    Function<String, Character> getFirstCharFunction = city -> {
        return city.charAt(0);
    };
    cities.stream().map(getFirstCharFunction)
        .forEach(System.out::println);
}
```

Supplier

The Supplier Interface is a part of the java.util.function package. It represents a function that does not take in any argument but produces a value of type T. It contains only one method.

```
T get();
```

We created a city's supplier for producing cities list and printing it.

```

@Test
public void supplyCities() {

    Supplier<String[]> citySupplier = () -> {
        return new String[]{"Mumbai", "Delhi", "Goa", "Pune"};
    };
    Arrays.asList(citySupplier.get()).forEach(System.out::println);
}

```

Java Functional Interfaces with Examples

October 10, 2021 by [Onur Baskirt](#)

In this article, we will learn **Java Functional Interfaces** which are coming by default in Java. These interfaces are; **Supplier**, **Consumer**, **Predicate**, **Function**, **Runnable**, and **Callable**. First of all, I highly suggest you use Java 8 and higher versions of Java to work with these interfaces. Currently, the latest LTS version is Java 17 and I will do these examples with Java 17 JDK. You can download the [Java 17 JDK here](#) and for installation please [read here](#).

Java Functional Interfaces

Let's start with the Supplier interface.

Supplier Interface

The **Supplier Interface** does not allow input, it returns a value based on a defined type. It is like a function without parameters, with a return type. Now, let's see this behavior with examples.

```

public class SupplierInterface {

    //Supplier function declarations.

    Supplier<String> textSupplier = () -> "Hello SW Test Academy!";

    Supplier<Integer> numberSupplier = () -> 1234;

    Supplier<Double> randomSupplier = () -> Math.random();
}

```



```

    Supplier<Double> randomSupplierMR = Math::random; //With
Method Reference (MR)

@Test

public void supplierTest() {

    //Calling Supplier functions.

    System.out.println(textSupplier.get());

    System.out.println(numberSupplier.get());

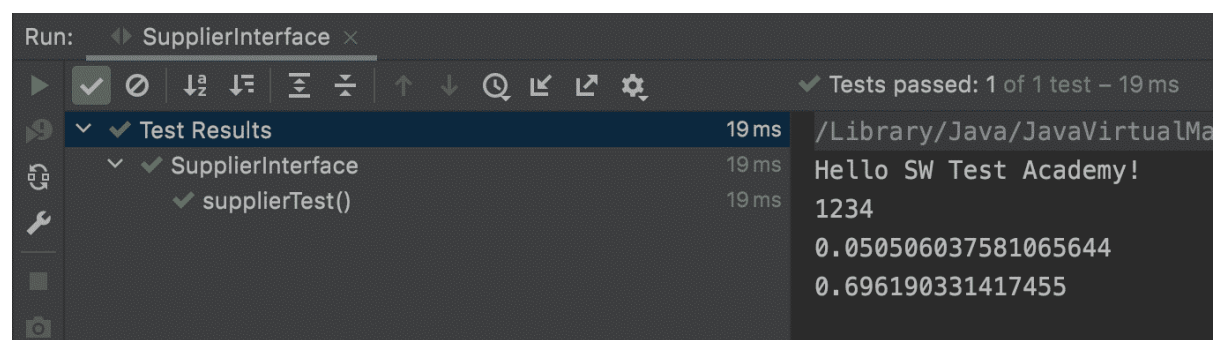
    System.out.println(randomSupplier.get());

    System.out.println(randomSupplierMR.get());

}
}

```

Output



Consumer Interface

The **Consumer Interface** takes an input, it does not return a value. It is like a function with a parameter, without a return type. **BiConsumer Interface** takes two inputs and does not return anything. That's why it is called "Bi" Consumer. If we chain multiple consumers with `andThen` method, first the first consumer will be executed and the second one executed. It is working like left to right flow.

Now, it is an examples time for each case. :)

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
```

```
public class ConsumerInterface {

    //Consumer function declarations.

    Consumer<String> upperCaseConsumer = (text) ->
System.out.println(text.toUpperCase());

    Consumer<String> lowerCaseConsumer = (text) ->
System.out.println(text.toLowerCase());

    Consumer<Double> logOfTenConsumer = (number) ->
System.out.println(Math.log10(number));

    //BiConsumer takes two parameters and does not return
anything!

    BiConsumer<Integer, Integer> powConsumer = (base, power) ->
System.out.println(Math.pow(base, power));

    @BeforeEach

    public void setup(TestInfo testInfo) {

        System.out.println("Test name: " +
testInfo.getDisplayName());

    }

    @AfterEach

    public void tearDown(){

        System.out.println();

    }

    @Order(1)

    @Test

    public void consumerTest() {

        //Calling Consumer functions.

        upperCaseConsumer.accept("Hello SW Test Academy!");

        lowerCaseConsumer.accept("Hello SW Test Academy!");

    }

}
```

```

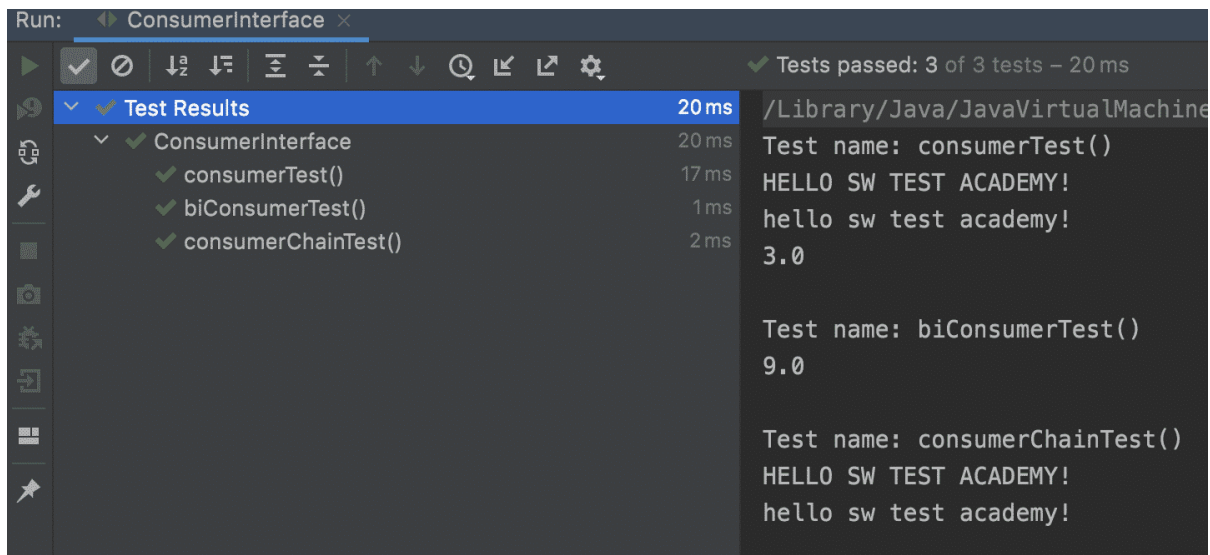
        logOfTenConsumer.accept(1000.00);
    }

    @Order(2)
    @Test
    public void biConsumerTest() {
        //Calling BiConsumer function.
        powConsumer.accept(3,2);
    }

    @Order(3)
    @Test
    public void consumerChainTest() {
        //Consumer chaining with andThen method.
        upperCaseConsumer
            .andThen(lowerCaseConsumer)
            .accept("Hello SW Test Academy!");
    }
}

```

Output



Function Interface

The **Function Interface** takes an input, it returns a defined type. It is like a function with a parameter, with a return type. The **first declaration is input, the second is the return type**. The **BiFunction Interface** takes two inputs rather than one input. That's the only difference between Function and BiFunction interfaces. Also, **UnaryOperator** interface takes and returns the same type.

If we chain the functions with **andThen method**, the execution order will be like the **left to right flow**. First, the first function will be run, then the others will be run. If we want to **run these functions right to left** we can use **compose method** rather than the andThen method.

Now, examples time to see all of these behaviors in action!

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)

public class FunctionInterface {

    //FunctionInterface function declarations. (Input Type,
    Return Type)

    Function<String, String> toUpperCase = (text) ->
text.toUpperCase();

    Function<String, String> toLowerCase = (text) ->
text.toLowerCase();
```

```

    Function<Integer, Double> log10      = (number) ->
Math.log10(number);

    //Method Reference Declarations (Input Type, Return Type)

    Function<String, String>  toUpperCaseMR =
String::toUpperCase;

    Function<String, String>  toLowerCaseMR =
String::toLowerCase;

    Function<Integer, Double> log10MR      = Math::log10;

    //BiFunction Example (Input Type, Input Type, Return Type)

    BiFunction<Integer, Integer, Integer> powerOf = (base, power)
-> (int) Math.pow(base, power);

    //UnaryOperator Example (Input and Return type are same.)

    UnaryOperator<String> appendText = (text) -> "I am appending:
" + text;

    @BeforeEach

    public void setup(TestInfo testInfo) {

        System.out.println("Test name: " +
testInfo.getDisplayName());

    }

    @AfterEach

    public void tearDown(){

        System.out.println();

    }

    @Order(1)

    @Test

    public void functionTest() {

        //Calling functions.

```

```

        String upperCaseResult = toUpperCase.apply("hello sw test
academy!");

        Double log10Result = log10.apply(10000);

        System.out.println(upperCaseResult);

        System.out.println(log10Result);

    }

    @Order(2)

    @Test

    public void functionChainWithAndThen() {

        //Function chaining. First do the first function then do
the second one.

        String chainResult1 =
toUpperCase.andThen(toLowerCase).apply("heLLo sW teSt ACadEmy!");

        String chainResult2 =
toLowerCase.andThen(toUpperCase).apply("heLLo sW teSt ACadEmy!");

        System.out.println(chainResult1);

        System.out.println(chainResult2);

    }

    @Order(3)

    @Test

    public void functionChainWithCompose() {

        //Function chaining. First do the second function then do
the first one. Vice versa of andThen.

        String chainResult1 =
toUpperCase.compose(toLowerCase).apply("heLLo sW teSt ACadEmy!");

        String chainResult2 =
toLowerCase.compose(toUpperCase).apply("heLLo sW teSt ACadEmy!");

        System.out.println(chainResult1);

```

```

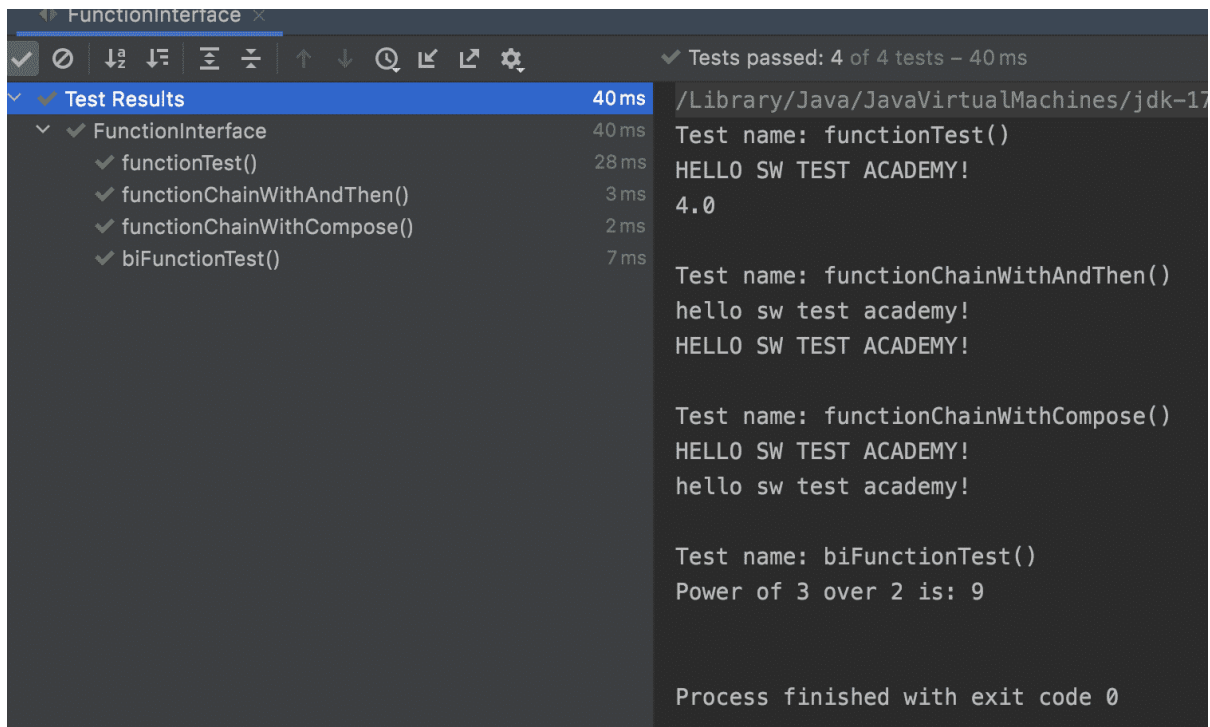
        System.out.println(chainResult2);
    }

    @Order(4)
    @Test
    public void biFunctionTest() {
        //Calling functions.
        int result = powerOf.apply(3, 2);
        System.out.println("Power of 3 over 2 is: " + result);
    }

    @Order(5)
    @Test
    public void unaryOperatorTest(){
        //Calling UnaryOperator
        System.out.println(appendText.apply("Hello SW Test
Academy!"));
    }
}

```

Output



Predicate Interface

The predicate takes an input, it returns a boolean value as true or false. It is like a function with a parameter, with the boolean return type. **BiPredicate Interface** takes two inputs and returns a boolean value.

Negate method does the inversion for the value.

And method is working as logical AND operation.

Or method is working as a logical OR operation.

Let's see all of these with examples.

```

@TestMethodOrder(MethodOrderer.OrderAnnotation.class)

public class PredicateInterface {

    //Predicate function declaration.

    String sampleText = "Hello SW Test Academy";

```



```

    Predicate<String> containsPredicate = (text) ->
sampleText.contains(text);

    //BiPredicate function declaration.

    BiPredicate<String, String> containsBiPredicate    = (text,
pattern) -> text.contains(pattern);

    BiPredicate<String, String> containsBiPredicateMR =
String::contains; //Method reference version.

    @BeforeEach

    public void setup(TestInfo testInfo) {

        System.out.println("Test name: " +
testInfo.getDisplayName());

    }

    @AfterEach

    public void tearDown(){

        System.out.println();

    }

    @Order(1)

    @Test

    public void predicateTest() {

        //Calling Predicate functions.

        boolean result = containsPredicate.test("SW");

        boolean resultOfNegate =
containsPredicate.negate().test("SW"); //negate is inverse
operation like "does not contain".

        boolean andResult =
containsBiPredicate.and(containsBiPredicate.negate()).test("SW",
"SW"); //Logical AND operation.

```

```

        boolean orResult =
containsBiPredicate.or(containsBiPredicate.negate()).test("SW",
"SW"); //Logical OR operation.

        System.out.println(result);

        System.out.println(resultOfNegate);

        System.out.println(andResult);

        System.out.println(orResult);

    }

    @Order(2)

    @Test

    public void predicateListTest() {

        List<Predicate<String>> predicateList = new
ArrayList<>();

        predicateList.add(containsPredicate);

        predicateList.add(containsPredicate.negate());

        predicateList

            .forEach(predicate ->
System.out.println(predicate.test("SW")));

    }

    @Order(3)

    @Test

    public void biPredicateTest() {

        //Calling BiPredicate functions.

        boolean result = containsBiPredicate.test("Hello SW Test
Academy", "SW");

        System.out.println(result);

    }

```

```
}
```

Output

Run: PredicateInterface x

✓ Tests passed: 3 of 3 tests – 20 ms

Test Results	20 ms	/Library/Java/JavaVirtualMachines/jdk-
✓ PredicateInterface	20 ms	Test name: predicateTest()
✓ predicateTest()	17 ms	true
✓ predicateListTest()	2 ms	false
✓ biPredicateTest()	1 ms	false
		true
		Test name: predicateListTest()
		true
		false
		Test name: biPredicateTest()
		true