

# 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<>();
```

## 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

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

**Example1:**

```
// 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:**

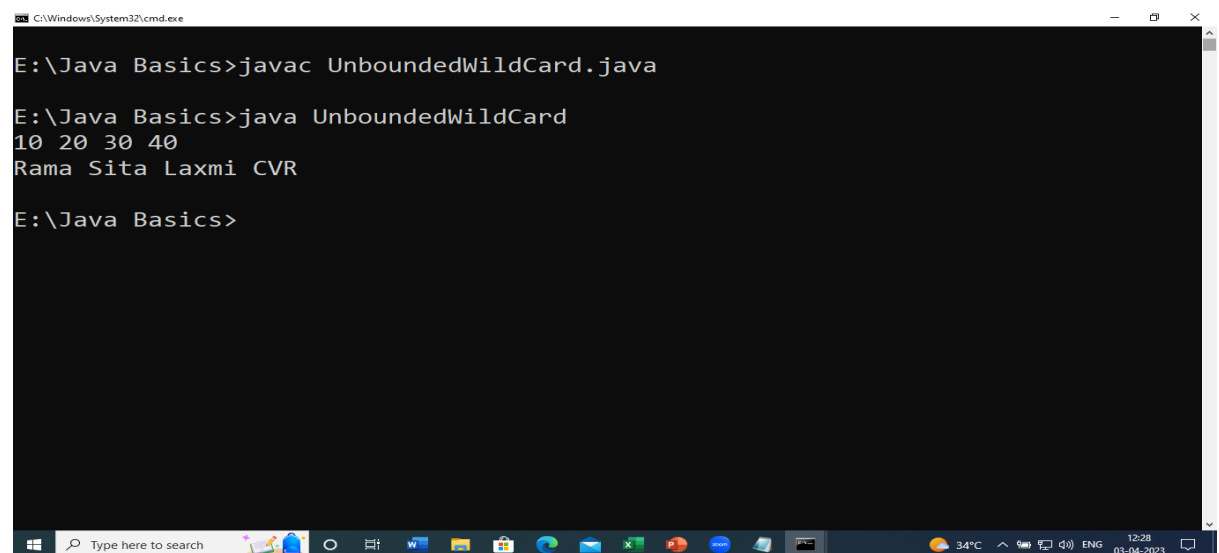
```
import java.util.*;

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 it's 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**

As you already know, it is possible to assign an object of one type to an object of another type provided that the types are compatible. For example, you can assign an Integer to an Object, since Object is one of Integer's supertypes:

```

Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger; // OK

```

In object-oriented terminology, this is called an "is a" relationship. Since an Integer *is a* kind of Object, the assignment is allowed. But Integer is also a kind of Number, so the following code is valid as well:

```

public void someMethod(Number n) { /* ... */ }

someMethod(new Integer(10)); // OK
someMethod(new Double(10.1)); // OK

```

**Note:** According to Liskov substitution principle, whenever we are expecting an object of sometype then we can always provide an object of subtype of that type.

The same is also true with generics. You can perform a generic type invocation, passing Number as its type argument, and any subsequent invocation of add will be allowed if the argument is compatible with Number:

```
Box<Number> box = new Box<Number>();  
box.add(new Integer(10)); // OK  
box.add(new Double(10.1)); // OK
```

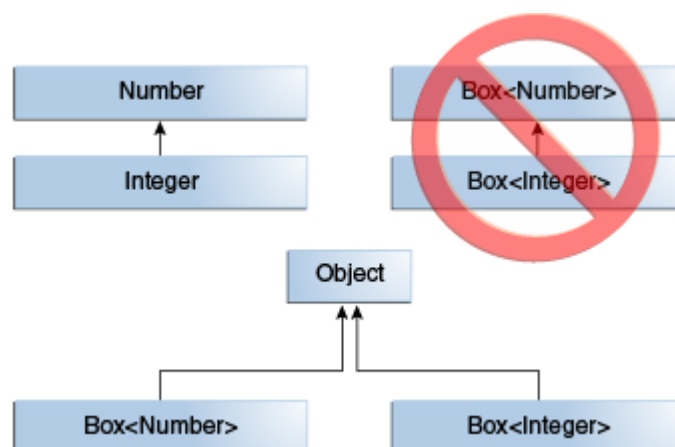
## **Subtypes**

Now consider the following method:

```
public void boxTest(Box<Number> n) { /* ... */ }
```

What type of argument does it accept? By looking at its signature, you can see that it accepts a single argument whose type is Box<Number>. But what does that mean? Are you allowed to pass in Box<Integer> or Box<Double>, as you might expect? The answer is "no", because Box<Integer> and Box<Double> are not subtypes of Box<Number>.

This is a common misunderstanding when it comes to programming with generics, but it is an important concept to learn.



Box<Integer> is not a subtype of Box<Number> even though Integer is a subtype of Number.

**Note:** Given two concrete types A and B (for example, Number and Integer), MyClass<A> has no relationship to MyClass<B>, regardless of whether or not A and B are related. The common parent of MyClass<A> and MyClass<B> is Object.

**Ex:**

Box.java

```
public class Box<T>{  
    T size;  
  
    public Box(T size)  
    {  
        this.size=size;  
    }  
  
    public String toString()  
    {  
        return "Box value="+size;  
    }  
}
```

**Main.java**

```
public class Main {  
  
    public static void main(String[] args) {  
        someMethod(new Box<Integer>(10));  
        someMethod(new Box<Double>(20.5));  
    }  
}
```

```

public static void someMethod(Box<?> o)
{
    System.out.println(o);
}
}

```

**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.

## **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 introduce in Java 7. Here, you can left 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 of 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 left 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<String>();  
        genericClass2.setName("peter");  
        System.out.println(genericClass2.getName());  
  
        // New improved type inference  
        System.out.println(genericClass2.genericMethod(new GenericClass<>()));  
    }  
}
```

## **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 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`.

### **5) 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
```

```

        // ...
    }
}

```

## **6)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.