ACHARYA INSTITUTE OF TECHNOLOGY

DR. SARVEPALLI RADHAKRISHNAN ROAD,

BENGALURU-560107

Department of Computer Science and Engineering

VI SEMESTER

ADVANCED JAVA PROGRAMMING

[21CS642]

Module – 2 : Generics

Prepared by:

Mrs. Nagamma Aravalli

Assistant Professor,

Dept, of Computer Science & Engineering,AIT

Module-2: Generics

❖ Introduction:

1. Generics introduced in JDK 5 and added new syntax to the java programming language.
2. With the use of generics, it is possible to create classes,interfaces and methods that will work in a type-safe manner with various kinds of data.
3. Generics allow algorithms to be defined once and applied to various data types without additional coding.
4. The benefits that generics added to Collections is that Collection classes can be used with complete type safety.

❖ What Are Generics?

1. Generics refer to parameterized types.
2. They are important because they enable the creation of classes, interfaces, and methods where the data type they operate on is specified as a parameter. This allows a single class, for example, to work automatically with different data types.
3. A class, interface, or method that operates on parameterized types is called generic, such as a generic class or a generic method.
4. The primary goals of Generics are to provide stronger type checks at compile time and eliminate the need for casting.
5. With generic all casts are automatic and implicit.

❖ A Simple Generic Example

Program 01:

The following program defines two classes. The first is the generic class **Gen**, and the second is **GenDemo**, which uses **Gen**.

```java
//A simple generic class
class Gen<T>{

     T ob;
     Gen(T o){
         ob = o;
     }

     T getob() {
         return ob;
     }

     void showType(){
         System.out.println("Type of T is: "+ob.getClass().getName());
     }
}
//Demonstrate the generic class
public class GenDemo {
    public static void main(String[] args) {
        Gen<Integer> iob;
        iob = new Gen<Integer>(88);

        int v = iob.getob();
        System.out.println("value: "+v);

        iob.showType();
        System.out.println();
        Gen<String> str;
        str = new Gen<String>("This is string param");

        String s = str.getob();
        System.out.println("value: "+s);

        str.showType();
    }

}
```

Output:

```
value: 88
Type of T is: java.lang.Integer

value: This is string param
Type of T is: java.lang.String
```

Program Explanation:

Generic class Gen:

1. Generic Class Declaration:
    - `class Gen<T>` declares a generic class with a type parameter `T`.Type parameters are declared within angle brackets.
    - `T ob` is a variable of type `T`, which means its type will be specified later when an object of `Gen` is created.
2. Constructor:
    - The constructor `Gen(T o)` initializes the object `ob` with a value of type `T` passed as an argument.
3. Getter Method:
    - The type parameter T can also be used to specify the return type of a method.
    - `T getob()` returns the object `ob` of type `T`.
4. Type Display Method:
    - `void showType()` displays the type of `T` using `ob.getClass().getName()`.
    - The `getClass()` method is available in all objects because it comes from the `Object` class. When you use `getClass()`, it gives you a `Class` object for that object's class. You can then call `getName()` on this `Class` object to get the class name as a string.

Demonstration in GenDemo class:

1. Creating a `Gen` Object for `Integer`:
    - A reference `iOb` for a `Gen` object that holds `Integer` as type argument is declared.
    - A `Gen<Integer>` object is instantiated with `88`. Autoboxing converts the `int` 88 to an `Integer` object.
2. Showing Type and Value:
    - `iOb.showType()` prints the type of `T`, which is `java.lang.Integer`.
    - `int v = iOb.getob();` retrieves the value without needing a cast and prints it.
3. Creating a `Gen` Object for `String`:
    - A `Gen<String>` object `strOb` is instantiated with the string `"Generics Test"`.
4. Showing Type and Value:
    - `strOb.showType()` prints the type of `T`, which is `java.lang.String`.
    - `String str = strOb.getob();` retrieves the string without needing a cast and prints it.

- The Java compiler does not create multiple versions of the generic class Gen.
- Instead, it removes all generic type information.
- It adds the necessary type casts to make the code work as if specific versions of Gen were created.
- Only one version of Gen actually exists in your program.
- This process is called **type erasure**.

Generics Work Only with Reference Types

- At the time of declaring an instance of generic type, the argument passed to the type parameter must be a reference type not primitive type.
  Ex: `Gen<int> intOb = new Gen<int>(53); // Error, can't use primitive type`

Generic types differ based on their type arguments.

- A reference of one specific version of a generic type is not type compatible with another version of the same generic type.
- From **Program 01** the following line of code is in error and will not compile:
  Ex: `iOb = strOb; // Wrong!`

How Generics Improve Type Safety?

- Using generics ensures type safety and eliminates the need for explicit type casts.
- Without generics, we can use `Object` to store any type, but this requires manual type casting and can lead to runtime errors.
- Generics catch type mismatch errors at compile time, preventing potential runtime exceptions.

Program 02:

To understand the benefits of generics, first consider the following program that creates a non-generic equivalent of **Gen**:

```java
class NonGen{
    Object ob; // ob is of type Object
    // Constructor
    NonGen(Object o) {
        ob = o;
    }
    // Return type Object
    Object getob() {
        return ob;
    }

    // Show type of ob
    void showType() {
        System.out.println("Type of ob is " + ob.getClass().getName());
    }
}

class NonGenDemo {
    public static void main(String args[]) {
        NonGen iOb;
        // Store an Integer in NonGen object
        iOb = new NonGen(88);
        iOb.showType();
        // Retrieve value with explicit cast
        int v = (Integer) iOb.getob();
        System.out.println("value: " + v);
        System.out.println();
        // Store a String in another NonGen object
        NonGen strOb = new NonGen("Non-Generics Test");
        strOb.showType();
        // Retrieve value with explicit cast
        String str = (String) strOb.getob();
        System.out.println("value: " + str);
        // This compiles but causes runtime error
        iOb = strOb;
        v = (Integer) iOb.getob(); // Runtime error!
    }
}
```

Output:

```
Type of ob is java.lang.Integer
value: 88

Type of ob is java.lang.String
value: Non-Generics Test
Exception in thread "main" java.lang.ClassCastException: class java.lang.String cannot be cast to class java.lang.Integer
```

Key points:

1. **Using `Object` Type:**
    i. `NonGen` class uses `Object` instead of a generic type.

   ii. This allows it to store any type of object, but requires explicit type casting.

2. **Type Safety Issues:**
   i. Without generics, the compiler cannot enforce type safety.
   ii. Explicit casts are necessary, e.g., `(Integer) iOb.getob()`
3. **Runtime Errors:**
   i. Assigning one `NonGen` object to another (`iOb = strOb;`) is allowed, but conceptually wrong.
   ii. Casting a `String` to `Integer` leads to a runtime exception.
4. **Generics Advantage:**
   i. Generics catch type mismatch errors at compile time, preventing runtime exceptions.
   ii. They ensure type-safe code by converting runtime errors into compile-time errors, enhancing code reliability and safety.

By using generics, we avoid these pitfalls, making our code safer and reducing the risk of runtime errors.

❖ A Generic Class with Two Type Parameters

We can declare more than one type parameter in a generic type. To specify two or more type parameters, simply use a comma-separated list.

Program 03:

This example demonstrates the flexibility and type safety provided by generics when dealing with multiple data types in a single class.

```java
class TwoGen<T, V> {
    T ob1;
    V ob2;
    // Constructor accepting two parameters of types T and V
    TwoGen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }
    // Method to show the types of T and V
    void showTypes() {
        System.out.println("Type of T is " + ob1.getClass().getName());
        System.out.println("Type of V is " + ob2.getClass().getName());
    }
    // Getter for T type object
    T getob1() {
        return ob1;
    }
    // Getter for V type object
    V getob2() {
        return ob2;
    }
}

class SimpGen {
    public static void main(String args[]) {
        // Creating an instance of TwoGen with Integer and String types
        TwoGen<Integer, String> tgObj = new TwoGen<Integer, String>(88, "Generics");
        // Display the types of the objects
        tgObj.showTypes();
        // Retrieve and display the values
        int v = tgObj.getob1();
        System.out.println("value: " + v);
        String str = tgObj.getob2();
        System.out.println("value: " + str);
    }
}
```

Output:

```
Type of T is java.lang.Integer
Type of V is java.lang.String
value: 88
value: Generics
```

Key Points:

1. **Declaration with Two Type Parameters:**
    - `class TwoGen<T, V>` specifies two type parameters, T and V.
    - When creating an instance, you provide two type arguments.
2. **Creating an Instance:**

```
TwoGen<Integer, String> tgObj = new TwoGen<Integer,
String>(88, "Generics");
```

- Integer is used for T and String for V.
- The constructor initializes ob1 with an Integer and ob2 with a String.

3. **Showing Types and Values:**
   - `tgObj.showTypes()` prints the types of ob1 and ob2.
   - `tgObj.getob1()` and `tgObj.getob2()` retrieve the values without casting.
4. **Flexibility with Type Parameters:**

```
TwoGen<String, String> x = new TwoGen<String,
String>("A","B");
```

- Using the same type for both parameters is valid, though typically one parameter would suffice if the types are always the same.

Summary:

- The TwoGen class uses two type parameters, T and V, to handle objects of different types.
- When creating an instance of TwoGen, you need to specify two type arguments.
- This setup ensures type safety and allows handling multiple data types in one class.

❖ The General Form of a Generic Class

- Following is the format for declaring a generic class.

Syntax:

```
class class-name<type-param-list> { // ...
```

- Following is the format for declaring a reference to a generic class and creating an instance of it.

Syntax:

```
class-name<type-arg-list> var-name = new
class-name<type-arg-list>(cons-arg-list);
```

❖ Bounded Types

Introduction:

- In Java, type parameters in generics can usually be any class type.
- Sometimes, we need to restrict (or "bound") the types that can be used with a type parameter.
- Bounded types allow us to specify constraints, such as requiring the type to be a subclass of a particular class or implementing a certain interface.
- For example, if we want a generic class to handle only numerical types for calculating averages, we can restrict the type parameter to numerical types like `Integer`, `Float`, and `Double`.

Definition:

- We can bound the type parameter for a particular range by using **extends** keywords such as types are called bounded types.

Key Features:

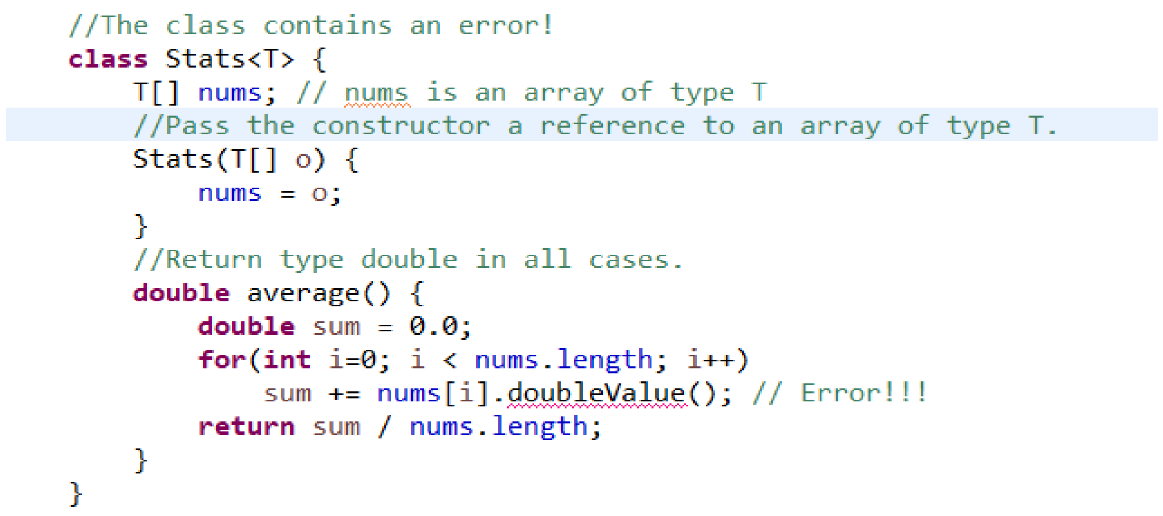
- Bounded types help ensure type safety and compatibility with specific operations.
- When specifying a type parameter, we can create an upper bound that declares the superclass from which all type arguments must be derived.
  General form : `<T extends superclass>`
- This specifies that `T` can only be replaced by superclass, or subclasses of superclass. Thus, superclass defines an inclusive, upper limit.
- We can use an interface type as a bound for a generic type parameter. Multiple interfaces can be specified as bounds.
- A bound can include both a class type and one or more interfaces. The class type must be specified first.
- Use the `&` operator to connect multiple bounds.

  Example: `class Gen<T extends MyClass & MyInterface> { // ... }`
  Here, `T` must be a subclass of `MyClass` and implement `MyInterface`.

The need of Bounded Types with an example.

The intention of the below program is to create a generic class `Stats<T>` that can compute the average of an array of numbers of any given type (e.g., integers, floats, doubles). However, the program contains an error.

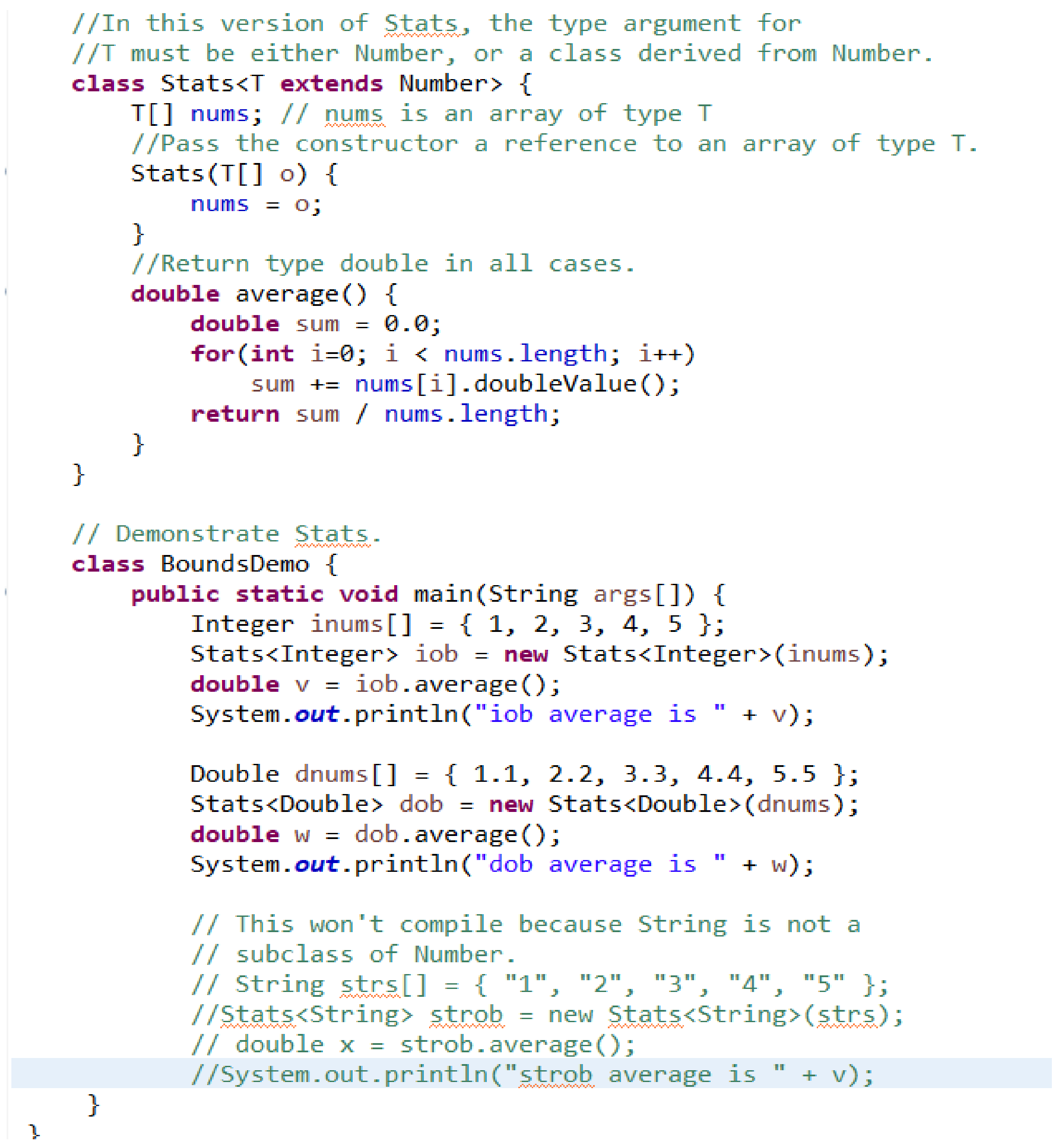
Code snippet:



```java
//The class contains an error!
class Stats<T> {
    T[] nums; // nums is an array of type T
    //Pass the constructor a reference to an array of type T.
    Stats(T[] o) {
        nums = o;
    }
    //Return type double in all cases.
    double average() {
        double sum = 0.0;
        for(int i=0; i < nums.length; i++)
            sum += nums[i].doubleValue(); // Error!!!
        return sum / nums.length;
    }
}
```

- The program tries to compute the average of an array of numbers.
- The `average` method iterates through the `nums` array and attempts to call `doubleValue()` on each element to add it to the `sum`.
- The error occurs because the method `doubleValue()` is not defined for the type `T`. The method `doubleValue()` is specific to the `Number` class, but `T` could be any type, not necessarily a subclass of `Number`.
- To fix the error, we need to bound the type parameter `T` to ensure it is a subclass of `Number`

Error fixed program : Program 04



```
    //In this version of Stats, the type argument for
    //T must be either Number, or a class derived from Number.
    class Stats<T extends Number> {
        T[] nums; // nums is an array of type T
        //Pass the constructor a reference to an array of type T.
        Stats(T[] o) {
            nums = o;
        }
        //Return type double in all cases.
        double average() {
            double sum = 0.0;
            for(int i=0; i < nums.length; i++)
                sum += nums[i].doubleValue();
            return sum / nums.length;
        }
    }

    // Demonstrate Stats.
    class BoundsDemo {
        public static void main(String args[]) {
            Integer inums[] = { 1, 2, 3, 4, 5 };
            Stats<Integer> iob = new Stats<Integer>(inums);
            double v = iob.average();
            System.out.println("iob average is " + v);

            Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
            Stats<Double> dob = new Stats<Double>(dnums);
            double w = dob.average();
            System.out.println("dob average is " + w);

            // This won't compile because String is not a
            // subclass of Number.
            // String strs[] = { "1", "2", "3", "4", "5" };
            //Stats<String> strob = new Stats<String>(strs);
            // double x = strob.average();
            //System.out.println("strob average is " + v);
     }
}
```

Output:

```
iob average is 3.0
dob average is 3.3
```

- The type parameter `T` is now bounded by `Number`, ensuring `T` has a `doubleValue()` method.

- This change ensures that the `average` method can safely call `doubleValue()` on elements of the `nums` array.
- The corrected class can now compute the average of an array of any numeric type, such as `Integer`, `Float`, or `Double`.

With these changes, the program successfully computes the average of an array of numbers while maintaining type safety and eliminating the error.

❖ Using Wildcard Arguments

- The **question mark (?)** is known as the **wildcard** in generic programming. It represents an unknown type.
- It can be used in place of the type parameter in a generic class or method

The need for Wildcard Arguments in Java Generics with Example

- The need for wildcard arguments in Java generics arises from the limitations of strict type safety, which can sometimes make it difficult for operations that are valid and necessary across different data types.
- For instance, consider the scenario where we want to compare the averages of arrays held within two different instances of a generic `Stats` class - one containing `Double` values and the other `Integer` values using the method sameAvg( ) and we can call this method as shown in below code snippet.

```
Integer inums[] = { 1, 2, 3, 4, 5 };
Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
Stats<Integer> iob = new Stats<Integer>(inums);
Stats<Double> dob = new Stats<Double>(dnums);
if(iob.sameAvg(dob))
  System.out.println("Averages are the same.");
else
  System.out.println("Averages differ.");

// This won't work!
// Determine if two averages are the same.
boolean sameAvg(Stats<T> ob) {
    if(average() == ob.average())
     return true;
 return false;
}
```

- Initially attempting to create a method like `sameAvg(Stats<T> ob)` proves restrictive because it only allows comparisons between `Stats` objects of the exact same type parameter (`T`).

- This approach fails when trying to compare averages of `Stats<Integer>` with `Stats<Double>`, for example.
- To compare averages of `Stats` objects with different numeric types, a **wildcard argument** is used.

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

- This method can now accept any `Stats` object, regardless of its type parameter, enabling a broader and more flexible comparison.

The following program demonstrates wildcard argument:

Program 05:

```
//Use a wildcard.
 class Stats<T extends Number> {
        T[] nums;
        Stats(T[] o) {
              nums = o;
        }
        //Return type double in all cases.
        double average() {
              double sum = 0.0;
              for(int i=0; i < nums.length; i++)
                    sum += nums[i].doubleValue();
              return sum / nums.length;
        }
// Determine if two averages are the same.Notice the use of the wildcard.
        boolean sameAvg(Stats<?> ob) {
              if(average() == ob.average())
                    return true;
              return false;
        }
}
// Demonstrate wildcard.
class WildcardDemo {
        public static void main(String args[]) {
              Integer inums[] = { 1, 2, 3, 4, 5 };
              Stats<Integer> iob = new Stats<Integer>(inums);
              double v = iob.average();
              System.out.println("iob average is " + v);
```

```
                Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
                Stats<Double> dob = new Stats<Double>(dnums);
                double w = dob.average();
                System.out.println("dob average is " + w);

                Float fnums[] = { 1.0F, 2.0F, 3.0F, 4.0F, 5.0F };
                Stats<Float> fob = new Stats<Float>(fnums);
                double x = fob.average();
                System.out.println("fob average is " + x);

                // See which arrays have same average.
                System.out.print("Averages of iob and dob ");
                if(iob.sameAvg(dob))
                        System.out.println("are the same.");
                else
                        System.out.println("differ.");
                System.out.print("Averages of iob and fob ");
                if(iob.sameAvg(fob))
                        System.out.println("are the same.");
                else
                        System.out.println("differ.");
        }
}
```

Output:

```
iob average is 3.0
dob average is 3.3
fob average is 3.0
Averages of iob and dob differ.
Averages of iob and fob are the same.
```

Note: The wildcard does not change what types of `Stats` objects you can create. This is controlled by the `extends` clause in the `Stats` declaration. The wildcard just allows any valid `Stats` object to be used.

Bounded Wildcards

In Java generics, wildcard arguments can be bounded similarly to how type parameters are bounded. Bounded wildcards are particularly useful when working with a class hierarchy. To understand their importance, let's go through an example with a class hierarchy that encapsulates coordinates.

Consider the following classes that represent two-dimensional, three-dimensional, and four-dimensional coordinates:

```
// Two-dimensional coordinates.
class TwoD {
    int x, y;
    TwoD(int a, int b) {
        x = a;
        y = b;
    }
}

// Three-dimensional coordinates.
class ThreeD extends TwoD {
    int z;
    ThreeD(int a, int b, int c) {
        super(a, b);
        z = c;
    }
}

// Four-dimensional coordinates.
class FourD extends ThreeD {
    int t;
    FourD(int a, int b, int c, int d) {
        super(a, b, c);
        t = d;
    }
}
```

- **TwoD**: Represents a point in 2D space with coordinates (x, y).
- **ThreeD**: Extends TwoD to include a third dimension (z).
- **FourD**: Extends ThreeD to include a fourth dimension (t).

Next, we create a generic class `Coords` to store an array of these coordinate objects:

```
// This class holds an array of coordinate objects.
class Coords<T extends TwoD> {
    T[] coords;
    Coords(T[] o) { coords = o; }
}
```

- `Coords<T extends TwoD>`: The type parameter `T` is bounded by `TwoD`, meaning `T` can be `TwoD` or any subclass of `TwoD`. This ensures that any array stored in a `Coords` object will contain objects of type `TwoD` or its subclasses.

We want to write methods to display coordinates. Here's a method to display the X and Y coordinates of any `Coords` object:

```
static void showXY(Coords<?> c) {
    System.out.println("X Y Coordinates:");
    for (int i = 0; i < c.coords.length; i++)
        System.out.println(c.coords[i].x + " " + c.coords[i].y);
    System.out.println();
}
```

- `Coords<?>`: The wildcard `?` matches any type that `Coords` can hold. Since `Coords` is bounded by `TwoD`, this means it can match `Coords<TwoD>`, `Coords<ThreeD>`, `Coords<FourD>`, etc.

This method can display X and Y coordinates for any `Coords` object, regardless of whether it holds `TwoD`, `ThreeD`, or `FourD` objects.

Bounded Wildcard to Display X, Y, and Z Coordinates

Now, suppose we want to write a method that displays the X, Y, and Z coordinates for `Coords<ThreeD>` and `Coords<FourD>` objects. We need to ensure that this method is not used with `Coords<TwoD>` objects, which don't have a Z coordinate. Here's how we can do it using a bounded wildcard:

```
static void showXYZ(Coords<? extends ThreeD> c) {
    System.out.println("X Y Z Coordinates:");
    for (int i = 0; i < c.coords.length; i++)
        System.out.println(c.coords[i].x + " " + c.coords[i].y + " "
+ c.coords[i].z);
    System.out.println();
}
```

- Coords<? extends ThreeD>: The wildcard ? is bounded by ThreeD. This means it can match any Coords object where the type is ThreeD or a subclass of ThreeD (like FourD).

This method ensures that only objects with at least three dimensions (X, Y, Z) are passed to it, thus preventing runtime errors when trying to access the Z coordinate.

Program that demonstrates the actions of a bounded wildcard argument

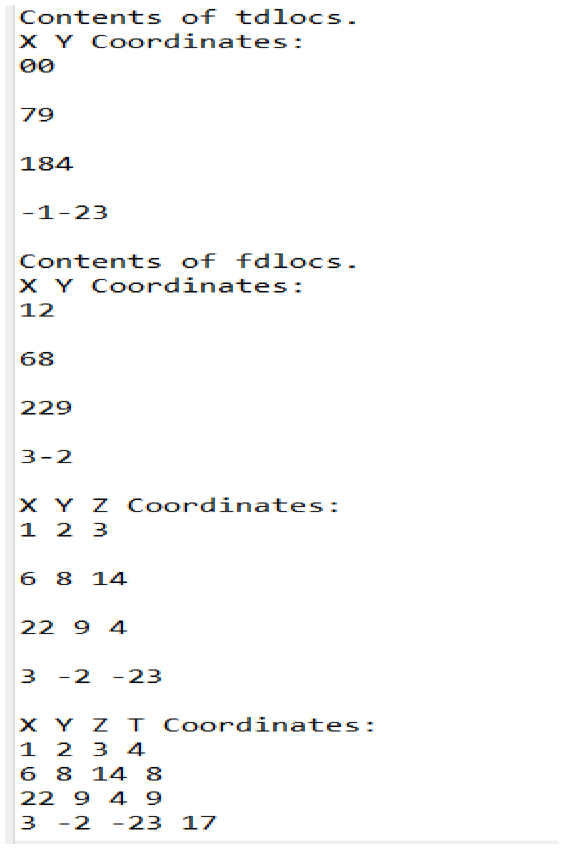
Program 06:

```
//Bounded Wildcard arguments.
//Two-dimensional coordinates.
class TwoD{
      int x,y;
      TwoD(int a, int b){
            x=a;
            y=b;
      }
}
//Three-dimensional coordinates.
class ThreeD extends TwoD{
      int z;
      ThreeD(int a, int b, int c){
            super(a,b);
            z=c;
      }
}
//Four-dimensional coordinates.
class FourD extends ThreeD{
      int t;
      FourD(int a, int b, int c, int d){
            super(a,b,c);
            t=d;
      }
}
//This class holds an array of coordinate objects.
class Coords<T extends TwoD>{
      T[] coords;
      Coords(T[] o){
            coords = o;
      }
}
```

```
//Demonstrate a bounded wildcard.
public class BoundedWildcard {
  static void showXY(Coords<?> c) {
      System.out.println("X Y Coordinates:");
      for(int i=0;i<c.coords.length;i++) {
            System.out.println(c.coords[i].x + ""+c.coords[i].y);
                  System.out.println();
            }
      }

      static void showXYZ(Coords<? extends ThreeD> c) {
            System.out.println("X Y Z Coordinates:");
            for(int i=0;i<c.coords.length;i++) {
                  System.out.println(c.coords[i].x+" "+
                                          c.coords[i].y+" "+
                                              c.coords[i].z);
                  System.out.println();

            }
      }

      static void showAll(Coords<? extends FourD> c) {
            System.out.println("X Y Z T Coordinates:");
            for(int i=0; i < c.coords.length; i++)
            System.out.println(c.coords[i].x + " " +
            c.coords[i].y + " " +
            c.coords[i].z + " " +
            c.coords[i].t);
            System.out.println();
            }

      public static void main(String args[]) {
            TwoD td[] = {
            new TwoD(0, 0),
            new TwoD(7, 9),
            new TwoD(18, 4),
            new TwoD(-1, -23)
            };
            Coords<TwoD> tdlocs = new Coords<TwoD>(td);
            System.out.println("Contents of tdlocs.");
            showXY(tdlocs); // OK, is a TwoD
            // showXYZ(tdlocs); // Error, not a ThreeD
            // showAll(tdlocs); // Error, not a FourD
```

```
                // Now, create some FourD objects.
                FourD fd[] = {
                new FourD(1, 2, 3, 4),
                new FourD(6, 8, 14, 8),
                new FourD(22, 9, 4, 9),
                new FourD(3, -2, -23, 17)
                };
                Coords<FourD> fdlocs = new Coords<FourD>(fd);
                System.out.println("Contents of fdlocs.");
                // These are all OK.
                showXY(fdlocs);
                showXYZ(fdlocs);
                showAll(fdlocs);
                }
        }
```

Output:



```
Contents of tdlocs.
X Y Coordinates:
00

79

184

-1-23

Contents of fdlocs.
X Y Coordinates:
12

68

229

3-2

X Y Z Coordinates:
1 2 3

6 8 14

22 9 4

3 -2 -23

X Y Z T Coordinates:
1 2 3 4
6 8 14 8
22 9 4 9
3 -2 -23 17
```

In Java generics, wildcards can be bounded to specify constraints on the type parameters that can be used. These constraints can either set an upper bound or a lower bound.

Upper Bound Wildcard

To establish an upper bound for a wildcard, use the following expression:

```
<? extends superclass>
```

This specifies that the wildcard can match any type that is a subclass of superclass or the superclass itself.

Lower Bound Wildcard

To establish a lower bound for a wildcard, use the following expression:

```
<? super subclass>
```

This specifies that the wildcard can match any type that is a superclass of `subclass` or the `subclass` itself.

❖ Creating a Generic Method

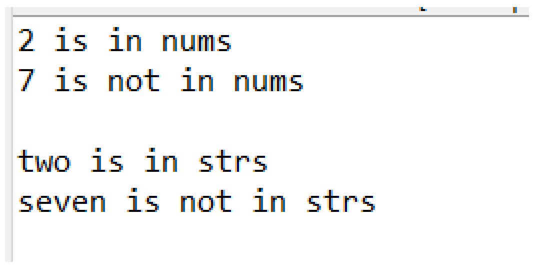
Generic methods allow you to define methods that operate on different types of data while maintaining type safety. These methods can be part of a generic class or standalone within a non-generic class. They can take one or more type parameters, making them versatile and reusable across different types.

Program 07: The program demonstrates a static generic method `isIn( )` that checks if an object is a member of an array.

```
//Demonstrate a simple generic method.
 class GenMethDemo {
   //Determine if an object is in an array.
   static <T extends Comparable<T>, V extends T> boolean isIn(T x, V[] y) {
             for(int i=0; i < y.length; i++)
                   if(x.equals(y[i])) return true;
             return false;
       }

       public static void main(String args[]) {
             //Use isIn() on Integers.
             Integer nums[] = { 1, 2, 3, 4, 5 };
```

```
            if(isIn(2, nums))
                    System.out.println("2 is in nums");
            if(!isIn(7, nums))
                    System.out.println("7 is not in nums");
            System.out.println();
            //Use isIn() on Strings.
            String strs[] = { "one", "two", "three", "four", "five" };
            if(isIn("two", strs))
                    System.out.println("two is in strs");
            if(!isIn("seven", strs))
                    System.out.println("seven is not in strs");
            //Oops! Won't compile! Types must be compatible.
            //if(isIn("two", nums))
                    //System.out.println("two is in strs");
  }
}
```

Output:



```
2 is in nums
7 is not in nums

two is in strs
seven is not in strs
```

1. **Generic Method Declaration**:
    - `<T extends Comparable<T>, V extends T>`: Declares two type parameters:
        - `T`, which extends `Comparable<T>`, ensuring `T` can be compared.
        - `V`, which extends `T`, ensuring `V` is compatible with `T` or its subclass.
    - `boolean isIn(T x, V[] y)`: The method returns a boolean indicating if `x` is in array `y`.
    - It checks if an object (`x`) is present in an array (`y`).
2. **Comparable Interface**
    - **Upper Bound**: By specifying `T extends Comparable<T>`, it ensures that `isIn` can only be used with objects that implement `Comparable`, thus can be compared.
3. **Static Method**

- **Static Declaration**: The `isIn` method is static, which means it can be called without creating an instance of the enclosing class (`GenMethDemo`). Generic methods can be either static or non-static.

4. **Method Call and Type Inference**
   - **Type Inference:** When calling `isIn`, you don't need to explicitly specify the type arguments. Java infers them based on the method arguments.

     Example:

```
Integer nums[] = {1, 2, 3, 4, 5};
if(isIn(2, nums)) // Integer inferred for T and V
    System.out.println("2 is in nums");
if(!isIn(7, nums)) // Integer inferred for T and V
    System.out.println("7 is not in nums");

String strs[] = {"one", "two", "three", "four", "five"};
if(isIn("two", strs)) // String inferred for T and V
    System.out.println("two is in strs");
if(!isIn("seven", strs)) // String inferred for T and V
    System.out.println("seven is not in strs");
```

5. **Explicit Type Specification**
   - **Explicit Types**: Though usually unnecessary, you can explicitly specify the type parameters:

```
GenMethDemo.<Integer, Integer>isIn(2, nums)
```

6. **Type Safety**
   - **Type Safety Enforcement**: The method ensures type safety through its bounds. For instance, the commented-out code would cause a compile-time error:

```
// if(isIn("two", nums))
// System.out.println("two is in nums");
```

This fails because T would be String, making V need to be String or a subclass. However, nums is an Integer[], which is incompatible with String.

General Syntax for Generic Methods

```
<type-param-list> ret-type meth-name(param-list) { // method body }
```

In all cases, *type-param-list* is a comma-separated list of type parameters.

Example:

```
static <T extends Comparable<T>, V extends T> boolean isIn(T x, V[]
y)
```

- type-param-list: <T extends Comparable<T>, V extends T>
- ret-type: boolean
- meth-name: isIn
- param-list: (T x, V[] y)

Conclusion:

The `isIn` method effectively demonstrates how to use type parameters in a method to ensure type safety and flexibility. By using bounded type parameters (`T` extends `Comparable<T>` and `V` extends `T`), it ensures that only compatible types are used, preventing runtime errors and enforcing compile-time checks. This illustrates the power and importance of generic methods in Java.

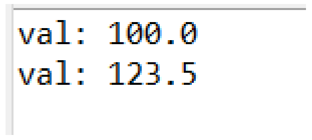
Generic Constructors

It is possible to have a generic constructor in a non-generic class. The following program demonstrates this concept:

```
//Use a generic constructor.
class GenCons {
  private double val;
    <T extends Number> GenCons(T arg) {
    val = arg.doubleValue();
   }
  void showval() {
    System.out.println("val: " + val);
   }
}
class GenConsDemo {
public static void main(String args[]) {
    GenCons test = new GenCons(100);
    GenCons test2 = new GenCons(123.5F);
    test.showval();
```

```
            test2.showval();
        }
      }
```

Output:



```
val: 100.0
val: 123.5
```

Because GenCons( ) specifies a parameter of a generic type, which must be a subclass of Number, GenCons( ) can be called with any numeric type, including Integer, Float, or Double. Therefore, even though GenCons is not a generic class, its constructor is generic

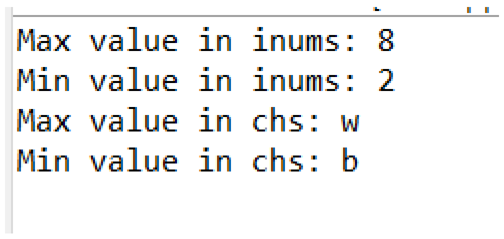
❖ Generic Interfaces

In Java, just like generic classes and methods, you can also create generic interfaces. This allows for a high degree of flexibility and reusability of code by allowing the interface to work with different types of data. Below is an example that illustrates how to define and implement a generic interface.

1. **Generic Interface Example**:

```
//Min/Max interface.
interface MinMax<T extends Comparable<T>> {
T min();
T max();
}
//Implement MinMax
class MyClass<T extends Comparable<T>> implements MinMax<T> {
T[] vals;
MyClass(T[] o) { vals = o; }
// Return the minimum value in vals.
public T min() {
    T v = vals[0];
    for (int i = 1; i < vals.length; i++)
        if (vals[i].compareTo(v) < 0) v = vals[i];
    return v;
}
// Return the maximum value in vals.
public T max() {
    T v = vals[0];
    for (int i = 1; i < vals.length; i++)
        if (vals[i].compareTo(v) > 0) v = vals[i];
    return v;
}
}
class GenIFDemo {
public static void main(String args[]) {
    Integer inums[] = {3, 6, 2, 8, 6};
    Character chs[] = {'b', 'r', 'p', 'w'};
    MyClass<Integer> iob = new MyClass<>(inums);
    MyClass<Character> cob = new MyClass<>(chs);
    System.out.println("Max value in inums: " + iob.max());
    System.out.println("Min value in inums: " + iob.min());
    System.out.println("Max value in chs: " + cob.max());
    System.out.println("Min value in chs: " + cob.min());
 }
}
```

Output:



```
Max value in inums: 8
Min value in inums: 2
Max value in chs: w
Min value in chs: b
```

2. **Generic Interface Syntax**:
    - Generic interfaces are declared similarly to generic classes.
    - `interface MinMax<T extends Comparable<T>>` defines a generic interface `MinMax` with a type parameter `T` bounded by `Comparable<T>`.
3. **Implementation of Generic Interface**:
    - `class MyClass<T extends Comparable<T>> implements MinMax<T>` defines `MyClass` as generic with `T` bounded by `Comparable<T>,` and `implements MinMax<T>.`
    - The type parameter `T` must match the type parameter of the interface it implements.
4. **Type Constraints**:
    - The implementing class (`MyClass`) must specify the same bounds as the interface.
    - **Incorrect**: `class MyClass<T extends Comparable<T>> implements MinMax<T extends Comparable<T>>`

   - **Correct**: `class MyClass<T extends Comparable<T>> implements MinMax<T>`
5. **Type Safety and Flexibility**:
   - Generic interfaces allow the implementation for various types while ensuring type safety.
   - Constraints (bounds) ensure that only types that meet certain criteria (like implementing `Comparable`) can be used.
6. **Using the Generic Class**:
   - Instances of `MyClass` can be created with different types like `Integer` and `Character`.
   - Example usage in `main`: `MyClass<Integer> iob = new MyClass<>(inums);`
7. **Generalized Syntax**:
   - For a generic interface: `interface interface-name<type-param-list> { ... }`
   - For a class implementing a generic interface: `class class-name<type-param-list> implements interface-name<type-arg-list> { ... }`

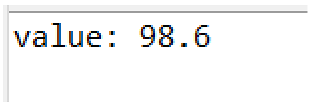
By understanding these points and the provided example, you can see how generic interfaces and their implementations provide powerful tools for creating flexible and type-safe code in Java.

❖ Raw Types and Legacy Code

- **Raw Types-Definition**: A raw type is a generic class or interface used without specifying any type parameters.
- **Legacy code -Definition**:legacy code refers to code that was written before generics were introduced in JDK 5. This code does not use generics and relies on raw types, meaning that it does not specify type parameters.
- Java allows raw types to maintain compatibility between pre-generics and generics code.
- Raw types enable old code to function with new generic code, but at the cost of type safety.
- Using raw types bypasses the type-checking mechanism of generics, leading to potential runtime errors.
- **Example**:Casting raw type to a specific type can cause runtime errors if the underlying type doesn't match.

```
package org.generic;
class Gen3<T> {
    T ob; // declare an object of type T

    Gen3(T o) {
        ob = o;
    }

    T getob() {
        return ob;
    }
}
public class RawDemo {
    public static void main(String[] args) {
       // Create a Gen object for Integers.
        Gen3<Integer> iOb = new Gen3<Integer>(88);

        // Create a Gen object for Strings.
        Gen3<String> strOb = new Gen3<String>("Generics Test");

        // Create a raw-type Gen3 object(legacy style) and give it a
Double value.
        Gen3 raw = new Gen3(new Double(98.6));

        // Cast here is necessary because type is unknown.
        double d = (Double) raw.getob();
        System.out.println("value: " + d);

        // The use of a raw type can lead to run-time exceptions.
Here are some examples.
        // The following cast causes a run-time error!
        // int i = (Integer) raw.getob(); // run-time error

        // This assignment overrides type safety.
        strOb = raw; // OK, but potentially wrong
        // String str = strOb.getob(); // run-time error

        // This assignment also overrides type safety.
        raw = iOb; // OK, but potentially wrong
        // d = (Double) raw.getob(); // run-time error
    }
}
```

Output:



```
value: 98.6
```

Key points from above example:

- Raw type is created as follows:

```
Gen3 raw = new Gen3(new Double(98.6));
```

- No type argument is specified, making `T` effectively `Object`.
- Assigning and casting raw types can cause runtime errors.

```
int i = (Integer) raw.getob(); // Runtime error
```

- Java compiler (`javac`) issues unchecked warnings when raw types are used in ways that could jeopardize type safety.
- Example warnings:

```
Gen3  raw = new Gen3(new Double(98.6)); // Warning

strOb = raw; // Warning
```

- Assignments between generic and raw types are syntactically correct but can lead to runtime issues.

```
raw = iOb; // Potentially wrong

strOb = raw; // Potentially wrong
```

Note: You should only use raw types when you need to mix old, pre-generics code with new, generic code. Raw types are meant to help transition from older code and should not be used in new code.

❖ Generic Class Hierarchies

- Generic classes can be part of a class hierarchy just like non-generic classes.
- A generic class can function as either a superclass or a subclass.
- In a generic hierarchy, type arguments needed by a generic superclass must be passed up the hierarchy by all subclasses.
- This requirement is similar to the way constructor arguments are passed up a class hierarchy.

Using a Generic Superclass

A simple example of a hierarchy that uses a generic superclass:

```
//A simple generic class hierarchy.
      class Gen<T> {
            T ob;
            Gen(T o) {
                  ob = o;
            }
            //Return ob.
            T getob() {
                  return ob;
            }
      }
      //A subclass of Gen.
      class Gen2<T> extends Gen<T> {
            Gen2(T o) {
                  super(o);
            }
      }
```

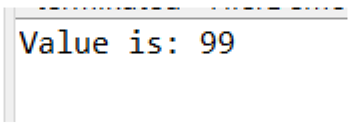
- In this hierarchy, Gen2 extends the generic class Gen
- Gen2 is declare as following line:

```
class Gen2<T> extends Gen<T> {
```

- The type parameter T is specified by Gen2 and is also passed to Gen in the **extends** clause.
- When a generic subclass is declared, it must pass its type parameter to the generic superclass.
- **Example**: `Gen2<Integer> num = new Gen2<Integer>(100);` passes `Integer` as the type parameter to Gen, making ob inside Gen of type `Integer`.
- Even if a subclass does not need to be generic for its own purposes, it must still declare and pass the type parameters required by the superclass.
- A subclass can add its own type parameters in addition to those of the superclass.
- Example with additional type parameter:

```
//A subclass can add its own type parameters.

            T ob;

            class Gen<T> {      Gen(T o) {
                  ob = o;
            }
            // Return ob.
            T getob() {
                  return ob;
            }
      }
      // A subclass of Gen that defines a second
      // type parameter, called V.
      class Gen2<T, V> extends Gen<T> {
            V ob2;
            Gen2(T o, V o2) {
                  super(o);
                  ob2 = o2;
            }
            V getob2() {
                  return ob2;
            }
      }
      // Create an object of type Gen2.
      class HierDemo {
            public static void main(String args[]) {
                  // Create a Gen2 object for String and Integer.
                  Gen2<String, Integer> x =
                  new Gen2<String, Integer>("Value is: ", 99);
                  System.out.print(x.getob());
                  System.out.println(x.getob2());
            }
      }
```

Output:



```
Value is: 99
```

- `Gen2<String, Integer> x = new Gen2<String, Integer>("Value is: ", 99);`
- Here, `T` is `String` and `V` is `Integer`.
- `Gen2` uses `T` for the superclass `Gen` and adds its own type parameter `V`.
- The example creates a `Gen2` object with `T` as `String` and `V` as `Integer`.
- The program outputs: `Value is: 99`.

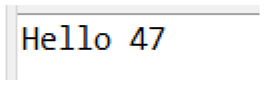
This demonstrates how a subclass in a generic hierarchy can handle and pass type parameters up the hierarchy, and how it can introduce additional type parameters if needed.

A Generic Subclass

It is acceptable for a non-generic class to be the superclass of a generic subclass.
Example,

```
//A non-generic class.
  class NonGen {
        int num;
        NonGen(int i) {
              num = i;
        }
  int getnum() {
        return num;
        }
  }
  //A generic subclass.
  class Gen<T> extends NonGen {
        T ob;
        Gen(T o, int i) {
        super(i);
              ob = o;
        }
        //Return ob.
        T getob() {
              return ob;
        }
  }
  //Create a Gen object.
  class HierDemo2 {
        public static void main(String args[]) {
              //Create a Gen object for String.
              Gen<String> w = new Gen<String>("Hello", 47);
              System.out.print(w.getob() + " ");
              System.out.println(w.getnum());
        }
  }
```

Output:



```
Hello 47
```

- A non-generic class NonGen and a generic subclass Gen<T>.
- The class NonGen is non-generic and has an integer field num.
- It includes a constructor that initializes num and a method getnum() that returns num.
- The class Gen<T> is generic with a type parameter T.
- Gen<T> extends NonGen, inheriting its properties and methods.

- `Gen<T>` has a field `ob` of type `T`.
- The constructor of `Gen<T>` takes a parameter of type `T` and an integer, passing the integer to the superclass constructor.
- `Gen<T>` includes a method `getob()` that returns `ob`.
- In the `main` method, a `Gen<String>` object is created with the string `"Hello"` and the integer `47`.
- The program prints: `Hello 47`.
- The generic subclass `Gen<T>` inherits the non-generic superclass `NonGen` normally.
- The type parameter `T` of `Gen` is not needed or used by `NonGen`.

This example demonstrates how a non-generic class can serve as a superclass for a generic subclass, with the generic subclass handling its own type parameter independently of the non-generic superclass.

Run-Time Type Comparisons Within a Generic Hierarchy

- The `instanceof` operator checks if an object is an instance of a specific class or can be cast to a specified type.
- It returns `true` if the object is of the specified type or a subclass of the specified type.
- The `instanceof` operator can be applied to objects of generic classes.
- The example consists of a generic superclass `Gen<T>` and a generic subclass `Gen2<T>`.
- Three objects are created:

  `iOb`: an instance of `Gen<Integer>`.

  `iOb2`: an instance of `Gen2<Integer>`.

  `strOb2`: an instance of `Gen2<String>`.

- the program checks the types of the created objects using `instanceof`.
- `iOb2` is checked against `Gen2<?>` and `Gen<?>`:
  - Both checks succeed because `iOb2` is an instance of `Gen2` and `Gen`.
- `strOb2` is checked against `Gen2<?>` and `Gen<?>`:
  - Both checks succeed because `strOb2` is an instance of `Gen2` and `Gen`.
- `iOb` is checked against `Gen2<?>` and `Gen<?>`:
  - The check against `Gen2<?>` fails because `iOb` is not an instance of `Gen2`.
  - The check against `Gen<?>` succeeds because `iOb` is an instance of `Gen`.

- The program includes commented-out lines that attempt to check if `iOb2` is an instance of `Gen2<Integer>`:
- These lines cannot be compiled because generic type information does not exist at runtime.
- The `instanceof` operator cannot determine specific generic types at runtime.

This example highlights how `instanceof` works with generic classes and the limitations regarding runtime type information for specific generic types.

Casting

- We can cast one instance of a generic class into another only if the two are otherwise compatible and their type arguments are the same.
- For example, assuming the foregoing program, this cast is legal:
  `(Gen<Integer>) iOb2` // legal
- because iOb2 includes an instance of `Gen<Integer>.`
- But, this cast: `(Gen<Long>) iOb2` // illegal
  is not legal because iOb2 is not an instance of `Gen<Long>.`

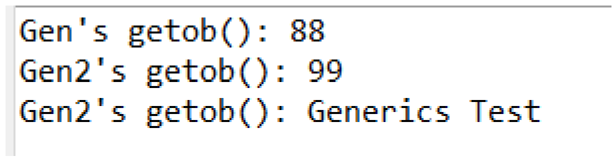
Overriding Methods in a Generic Class

In Java, methods in a generic class can be overridden just like methods in non-generic classes. For example,
consider this program in which the method **getob( )** is overridden:

```
//Overriding a generic method in a generic class.
      class Gen<T> {
       T ob; // declare an object of type T

       // Pass the constructor a reference to an object of type T.
       Gen(T o) {
           ob = o;
       }

       // Return ob.
       T getob() {
           System.out.print("Gen's getob(): ");
           return ob;
       }
      }

      //A subclass of Gen that overrides getob().
      class Gen2<T> extends Gen<T> {
```

```
 Gen2(T o) {
     super(o);
 }

 // Override getob().
 T getob() {
     System.out.print("Gen2's getob(): ");
     return ob;
 }
}

//Demonstrate generic method override.
public class OverrideDemo {
 public static void main(String[] args) {
     // Create a Gen object for Integers.
     Gen<Integer> iOb = new Gen<>(88);

     // Create a Gen2 object for Integers.
     Gen2<Integer> iOb2 = new Gen2<>(99);

     // Create a Gen2 object for Strings.
     Gen2<String> strOb2 = new Gen2<>("Generics Test");

     // Display the values.
     System.out.println(iOb.getob());
     System.out.println(iOb2.getob());
     System.out.println(strOb2.getob());
 }
}
```

Output:



```
Gen's getob(): 88
Gen2's getob(): 99
Gen2's getob(): Generics Test
```

Explanation:

Superclass Gen:

- The class Gen<T> is a generic class with a type parameter T.
- It has a constructor that initializes an object of type T.

- The `getob()` method returns the value of `ob` and prints a message indicating that it is `Gen`'s version of `getob()`.

Subclass `Gen2`:

- The class `Gen2<T>` extends `Gen<T>`, inheriting its properties and methods.
- The `getob()` method is overridden in `Gen2` to return the value of `ob` and print a message indicating that it is `Gen2`'s version of `getob()`.

Demonstration in `OverrideDemo`:

- An object `iOb` of type `Gen<Integer>` is created, and the `getob()` method is called. It calls the `Gen`'s version of `getob()`.
- An object `iOb2` of type `Gen2<Integer>` is created, and the `getob()` method is called. It calls the overridden version in `Gen2`.
- An object `strOb2` of type `Gen2<String>` is created, and the `getob()` method is called. It calls the overridden version in `Gen2`.

The output confirms that the overridden version of `getob()` is called for objects of type `Gen2`, while the superclass version is called for objects of type `Gen`. This demonstrates how method overriding works with generic classes in Java.

❖ Erasure

- Generics in Java had to be designed to be compatible with existing non-generic code.
- This ensures that older code does not break with the introduction of generics.
- Java uses a process called **erasure** to implement generics.
- During compilation, all generic type information is removed (erased).
- **Erasure Process**:
  - **Type Parameters Replacement**: Generic type parameters are replaced with their bound types. If no specific bound is provided, they are replaced with `Object`.
  - **Type Compatibility**: The compiler inserts appropriate casts to ensure type compatibility according to the specified type arguments.
- **Compile-Time Mechanism**:
  - The enforcement of type checks and compatibility happens at compile time.
  - At runtime, no generic type information is present; generics are purely a source-code mechanism.

Bridge Methods

- Bridge methods are automatically generated by the Java compiler.
- They handle situations where type erasure of an overriding method in a subclass

doesn't match the erasure of the method in the superclass.
- Type erasure removes generic type information during compilation.
- This can cause method signatures to differ between superclass and subclass after erasure.
- Bridge methods ensure compatibility between subclass and superclass methods.
- They allow the subclass to correctly override methods from the superclass after type erasure.
- Bridge methods only occur at the bytecode level, are not seen by we, and are not available for our use.

Consider the following code which creates a scenario requiring a bridge method:

```
//Generic class
      class Gen<T> {
       T ob;

       Gen(T o) {
           ob = o;
       }

       T getob() {
           return ob;
       }
      }

      //Subclass with a specific type
      class Gen2 extends Gen<String> {
       Gen2(String o) {
           super(o);
       }

       // Override with a specific return type
       String getob() {
           System.out.print("You called String getob(): ");
           return ob;
       }
      }

      //Demonstration class
      class BridgeDemo {
       public static void main(String args[]) {
           Gen2 strOb2 = new Gen2("Generics Test");
           System.out.println(strOb2.getob());
       }
      }
```

Output:

```
You called String getob(): Generics Test
```

Explanation:

- **Generic Class Gen**: Defines a generic type T and a method getob() that returns T.
- **Subclass Gen2**: Extends Gen with a specific type String and Overrides getob() to return String.
- After type erasure, the superclass Gen's method signature becomes Object getob().
- The subclass Gen2's method signature remains String getob().
- To resolve this discrepancy, the compiler generates a bridge method in Gen2.
- This bridge method matches the erased method signature: Object getob().
- The bridge method calls the subclass's overridden method.
- Using javap, we can see both the original and the bridge method in the compiled class file:

```
class Gen2 extends Gen<java.lang.String> {
    Gen2(java.lang.String);
    java.lang.String getob();
    java.lang.Object getob(); // bridge method
}
```

- The overridden method in Gen2 is called, displaying the custom message and returning the string.

Bridge methods are a mechanism to maintain method overriding compatibility across generic class hierarchies when type erasure causes method signatures to differ. They are automatically generated by the compiler and ensure that method calls work as expected in both generic and non-generic contexts.

❖ Ambiguity Errors

- **Ambiguity Errors:**
  - These errors occur when type erasure causes distinct generic declarations to resolve to the same erased type, leading to conflicts.
- **Example of Ambiguity with Method Overloading**:

```
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;
   }
}
```

- **Problem Explanation:**
  - MyGenClass declares two generic types: T and V.
  - The class attempts to overload the set method with parameters of type T and V.
  - If T and V are the same type (e.g., String), the overloaded methods become identical, leading to a compilation error.
  - Type erasure reduces both methods to void set(Object o), making them ambiguous.
- **Instantiation Example:**
  - This instantiation would cause ambiguity:

```
MyGenClass<String, String> obj = new MyGenClass<String, String>();
```

- **Resolving Ambiguity with Bounded Types**:
  - Adding bounds to V might seem to solve the issue:

```
class MyGenClass<T, V extends Number> {
    // ...
}
```

- **Successful Instantiation:**

```
MyGenClass<String, Number> x = new MyGenClass<String, Number>();
```

- **Ambiguous Instantiation:**

```
MyGenClass<Number, Number> x = new MyGenClass<Number, Number>();
```

  - In this case, since both T and V are Number, the call to set becomes ambiguous again.

- **Solution to Ambiguity:**
  - Instead of overloading methods, use distinct method names to avoid ambiguity.
- **Example Solution:**

```
class MyGenClass<T, V> {
    T ob1;
    V ob2;

    void setT(T o) {
        ob1 = o;
    }

    void setV(V o) {
        ob2 = o;
    }
}
```

❖ Some Generic Restrictions

- There are a few restrictions that you need to keep in mind when using generics.
- They involve creating objects of a type parameter, static members, exceptions, and arrays

1. **Type Parameters Can't Be Instantiated**

   It is not possible to create an instance of a type parameter. **Ex**:

```
// Can't create an instance of T.
class Gen<T> {
 T ob;
 Gen() {
  ob = new T(); // Illegal!!!
 }
}
```

   Here, it is illegal to attempt to create an instance of T.Because the compiler does not know what type of object to create

2. **Restrictions on Static Members**
   - Static members in a generic class cannot use the type parameters declared by the enclosing class.

- Attempting to declare static members that use the type parameter T declared by the class will result in a compile-time error.
- Ex:

```
class Wrong<T> {
    // Illegal: static variable cannot use T
    static T ob;

    // Illegal: static method cannot use T
    static T getob() {
        return ob;
    }
}
```

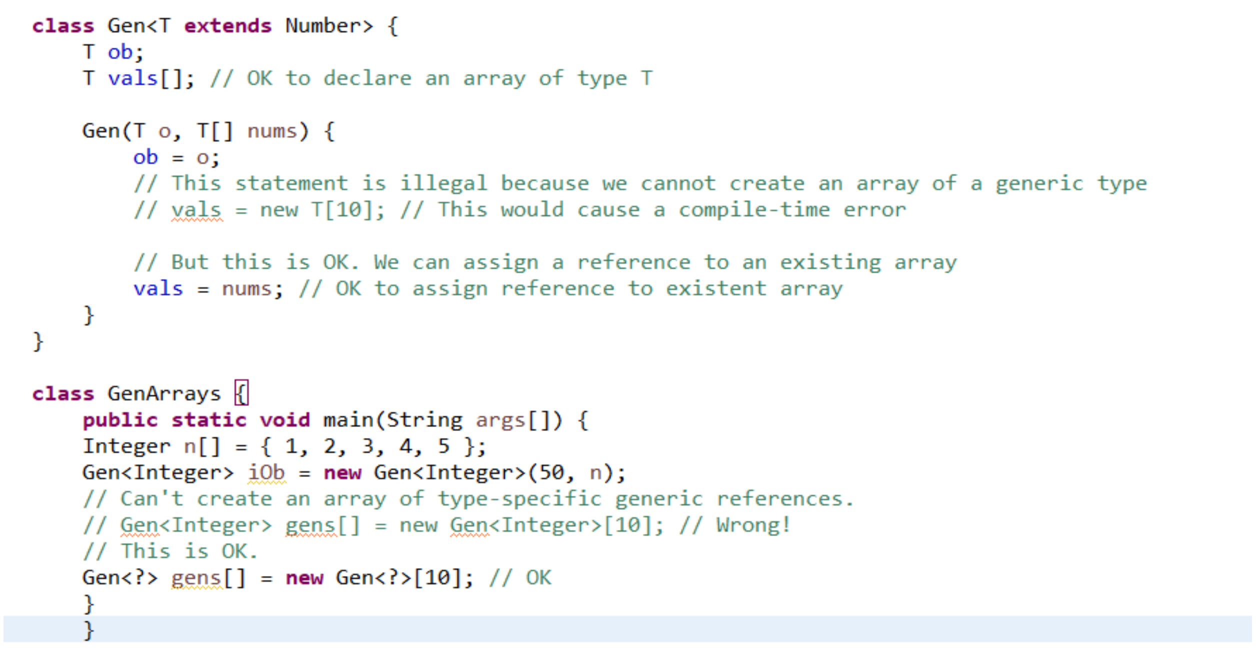
- We can declare static generic methods, which define their own type parameters.

3. Generic Array Restrictions

a. Cannot Instantiate an Array of a Type Parameter

b. Cannot Create an Array of Type-Specific Generic References

The following short program shows both situations:



```
class Gen<T extends Number> {
    T ob;
    T vals[]; // OK to declare an array of type T

    Gen(T o, T[] nums) {
        ob = o;
        // This statement is illegal because we cannot create an array of a generic type
        // vals = new T[10]; // This would cause a compile-time error

        // But this is OK. We can assign a reference to an existing array
        vals = nums; // OK to assign reference to existent array
    }
}

class GenArrays {
    public static void main(String args[]) {
    Integer n[] = { 1, 2, 3, 4, 5 };
    Gen<Integer> iOb = new Gen<Integer>(50, n);
    // Can't create an array of type-specific generic references.
    // Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!
    // This is OK.
    Gen<?> gens[] = new Gen<?>[10]; // OK
    }
    }
```

From above example:

Situation a:

- T vals[]; is a legal declaration of an array of type T.
- vals = new T[10]; would cause a compile-time error because there is no way for the compiler to know what type of array to actually create.
- Assigning an existing array nums to vals is allowed because nums has a concrete type known at runtime.

Situation b:

- Gen<Integer> gens[] = new Gen<Integer>[10]; would cause a compile-time error because you cannot create an array of Gen<Integer>.
- Gen<?> gens[] = new Gen<?>[10]; is allowed because Gen<?> denotes an array of Gen instances of any type.

4. Generic Exception Restriction

A generic class cannot extend Throwable. This means that you cannot create generic exception classes.