Module-2

What Are Generics?

Generics in Java allow you to create classes, interfaces, and methods that operate on various types while providing compile-time type safety. They enable you to define a class, interface, or method with type parameters, which can then be used with different types in a type-safe manner.

Key Concepts

1. **Parameterized Types**: Generics allow you to specify types as parameters. This means you can create a class or method that works with any data type, but still maintains type safety.
2. **Type Safety**: Before generics, you could create generalized classes or methods using Object references, but this approach lacked type safety. Generics provide compile-time type checks and eliminate the need for explicit type casting.
3. **Code Reusability**: Generics help in writing reusable and maintainable code. You can write a class or method that works with different types of objects without duplicating code for each type

A Simple Generics Example

Generics in Java allow you to define classes, interfaces, and methods with type parameters, enabling them to work with different types while maintaining type safety. Here's a simple example of a generic class and how it's used.

Example Code

Generic Class (Gen<T>)

```
// A simple generic class.
// Here, T is a type parameter that
// will be replaced by a real type
// when an object of type Gen is created.
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() {
        return ob;
    }

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

Demo Class (GenDemo)

```
// Demonstrate the generic class.
class GenDemo {
    public static void main(String args[]) {
        // Create a Gen reference for Integers.
        Gen<Integer> iOb;

        // Create a Gen<Integer> object and assign its
        // reference to iOb. Notice the use of autoboxing
        // to encapsulate the value 88 within an Integer object.
        iOb = new Gen<Integer>(88);

        // Show the type of data used by iOb.
        iOb.showType();

        // Get the value in iOb. Notice that
        // no cast is needed.
        int v = iOb.getob();
        System.out.println("value: " + v);
```

```
        System.out.println();

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

        // Show the type of data used by strOb.
        strOb.showType();

        // Get the value of strOb. Again, notice
        // that no cast is needed.
        String str = strOb.getob();
        System.out.println("value: " + str);
    }
}
```

Explanation

1. **Generic Class Definition**:
    - class Gen<T>: Defines a generic class where T is a placeholder for a specific type.
    - T ob;: Declares a member variable ob of type T.
    - Gen(T o): Constructor that initializes ob with a value of type T.
    - T getob(): Method that returns ob, which is of type T.
    - void showType(): Method that displays the runtime type of ob.
2. **Using Generics**:
    - Gen<Integer> iOb;: Creates a Gen object with Integer as the type parameter.
    - iOb = new Gen<Integer>(88);: Initializes iOb with an Integer value of 88.
    - iOb.showType();: Displays the type of T (in this case, Integer).
    - int v = iOb.getob();: Retrieves the value of ob, which is 88, and auto-unboxes it into an int.
    - Gen<String> strOb = new Gen<String>("Generics Test");: Creates a Gen object with String as the type parameter and initializes it with "Generics Test".
    - strOb.showType();: Displays the type of T (in this case, String).

- String str = strOb.getob();: Retrieves the value of ob, which is "Generics Test", and assigns it to str.

Output

The program produces the following output:

Type of T is java.lang.Integer

value: 88

Type of T is java.lang.String

value: Generics Test

Key Points

- **Type Parameter (T)**: Acts as a placeholder for the actual type that will be used when an instance of the generic class is created.
- **Type Safety**: Generics provide compile-time type safety. You can't assign an object of one type to a generic class of another type without a compile-time error.
- **Autoboxing and Unboxing**: Java automatically converts between primitive types and their corresponding wrapper classes (e.g., int to Integer and vice versa).

Generics Work Only with Reference Types

Java generics only work with reference types, not primitive types. This is because generics are implemented using type erasure, which involves replacing generic types with their bounds or Object if no bound is specified. Primitive types like int or char are not reference types and thus cannot be used as type arguments in generics.

Example of Illegal Use:

Gen<int> intOb = new Gen<int>(53); // Error, can't use primitive type

To handle primitives, Java provides wrapper classes like Integer, Character, etc. These classes are reference types and can be used with generics.

Example of Legal Use:

Gen<Integer> intOb = new Gen<Integer>(53); // Correct, using Integer wrapper class

Generic Types Differ Based on Their Type Arguments

Generic types are differentiated based on their type arguments. This means that even if two generic types are structurally similar, they are considered different if their type parameters differ.

Example of Type Safety:

Gen<Integer> iOb = new Gen<Integer>(88);

Gen<String> strOb = new Gen<String>("Hello");

```
// This is illegal and will not compile:
// iOb = strOb; // Error: incompatible types
```

This strict type checking ensures that operations involving generics are type-safe, reducing runtime errors.

How Generics Improve Type Safety

Generics improve type safety by allowing the compiler to enforce type constraints, thus reducing the need for explicit casting and catching type errors at compile time rather than at runtime.

Non-Generic Equivalent Example

Here is a non-generic class that mimics the functionality of a generic class but lacks type safety:

```
class NonGen {
   Object ob; // ob can hold any type of object
   NonGen(Object o) {
      ob = o;
   }
   Object getob() {
      return ob;
   }

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

class NonGenDemo {
   public static void main(String args[]) {
      NonGen iOb = new NonGen(88); // Stores an Integer
      iOb.showType(); // Shows Integer type

      int v = (Integer) iOb.getob(); // Cast required to retrieve Integer value
```

```
        System.out.println("value: " + v);

        NonGen strOb = new NonGen("Non-Generics Test"); // Stores a String
        strOb.showType(); // Shows String type

        String str = (String) strOb.getob(); // Cast required to retrieve String value
        // This line is allowed but incorrect, leading to runtime errors
        iOb = strOb;
        v = (Integer) iOb.getob(); // Causes a runtime ClassCastException
    }
}
```

Issues in Non-Generic Code:

1. **Explicit Casting:** You must cast the objects to the appropriate type, which can lead to runtime errors if the cast is incorrect.
2. **Type Mismatch Errors:** The code compiles even if there are type mismatches. Runtime exceptions occur if incorrect casts are made.

Generic Equivalent Example:

```
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());
    }
}
```

```
class GenDemo {
   public static void main(String args[]) {
      Gen<Integer> iOb = new Gen<Integer>(88);
      iOb.showType(); // Shows Integer type

      int v = iOb.getob(); // No cast required
      System.out.println("value: " + v);

      Gen<String> strOb = new Gen<String>("Generics Test");
      strOb.showType(); // Shows String type

      String str = strOb.getob(); // No cast required

      // Compile-time error if you try to assign a Gen<String> to a Gen<Integer>
      // iOb = strOb; // Error: incompatible types
   }
}
```

Benefits of Generics:

1. **Type Safety:** Errors are caught at compile time, reducing the risk of runtime exceptions.
2. **Elimination of Casts:** The need for explicit casts is eliminated, making the code cleaner and less error-prone.

Declaration of a Generic Class with Two Type Parameters

The TwoGen class you've provided is a great example of how to declare a generic class with two type parameters:

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

   // Constructor to initialize both type parameters
```

```
    TwoGen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }

    // Method to show 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());
    }

    // Methods to return the values of T and V
    T getob1() {
        return ob1;
    }

    V getob2() {
        return ob2;
    }
}
```

Using the Generic Class with Two Type Parameters

To use TwoGen, you need to specify both type parameters when creating an instance. Here's an example demonstrating how to instantiate and use the TwoGen class:

```
class SimpGen {
    public static void main(String args[]) {
        // Create an instance of TwoGen with Integer and String type parameters
        TwoGen<Integer, String> tgObj = new TwoGen<Integer, String>(88, "Generics");

        // Show the types of T and V
        tgObj.showTypes();
```

```
        // Obtain and show values
        int v = tgObj.getob1();
        System.out.println("value: " + v);
        String str = tgObj.getob2();
        System.out.println("value: " + str);
    }
}
```

Output

The output of the SimpGen class will be:

Type of T is java.lang.Integer

Type of V is java.lang.String

value: 88

value: Generics

Key Points:

1. **Multiple Type Parameters:** The TwoGen class uses two type parameters, T and V, which can be any reference types. This allows the class to handle different types of objects in a type-safe manner.
2. **Constructor and Methods:** The constructor initializes both type parameters, and the methods showTypes, getob1, and getob2 work with the type parameters to provide functionality.
3. **Same Type Arguments:** It's possible for both type parameters to be of the same type. For example:

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

Here, both T and V are String. However, if both parameters are always of the same type, using a single type parameter might be more appropriate.

Benefits of Multiple Type Parameters:

- **Flexibility:** Allows the creation of more complex and versatile generic classes.
- **Type Safety:** Ensures that the types used with the generic class are consistent and prevents type mismatches.
- **Reusability:** Promotes code reuse by allowing the same class to work with different types without sacrificing type safety.

Creating an Instance

To create an instance of a generic class, you specify the type arguments in angle brackets and provide constructor arguments as needed:

ClassName<TypeArgList> varName = new ClassName<TypeArgList>(constructorArgs);

- **TypeArgList**: The type arguments for the type parameters.
- **varName**: The variable name for the instance.
- **constructorArgs**: Arguments passed to the constructor.

Bounded Types

Sometimes you want to restrict the types that can be used as type parameters. Bounded types help achieve this by specifying constraints on the type parameters.

Upper Bound

You can specify an upper bound for a type parameter to restrict it to a certain class or its subclasses. This is done using the extends keyword:

```
class ClassName<T extends Superclass> {
   // Class body
}
```

Here, T can only be a subclass of Superclass or Superclass itself. This ensures that the type parameter T will have all the methods and properties of Superclass.

Example with Numeric Types

Consider the Stats class, which calculates the average of an array of numbers. The original version of the class fails because T is not constrained, so the compiler does not recognize doubleValue() method from Number:

```
class Stats<T> {
   T[] nums;

   Stats(T[] o) {
      nums = o;
   }

   double average() {
      double sum = 0.0;
      for(int i = 0; i < nums.length; i++)
```

```
        sum += nums[i].doubleValue(); // Error!
    return sum / nums.length;
  }
}
```

To resolve this, you can bound T by Number, which is the superclass of all numeric wrapper classes:

```
class Stats<T extends Number> {
  T[] nums;

  Stats(T[] o) {
    nums = o;
  }

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

Example Usage

Here's how you can use the bounded Stats class:

```
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);

        // The following line would cause a compile-time error
        // because String is not a subclass of Number.
        // String[] strs = { "1", "2", "3", "4", "5" };
        // Stats<String> strob = new Stats<String>(strs);
    }
}
```

Multiple Bounds

You can also specify multiple bounds for a type parameter. The syntax involves combining class and interface bounds:

```
class Gen<T extends MyClass & MyInterface> {
    // Class body
}
```

Here, T must be a subclass of MyClass and must implement MyInterface.

- **MyClass**: A class type that T must extend.
- **MyInterface**: An interface type that T must implement.

Summary

- **Generic Class Syntax:** class ClassName<TypeParamList> { ... }
- **Instance Creation:** ClassName<TypeArgList> varName = new ClassName<TypeArgList>(constructorArgs);
- **Bounded Types:** Restrict type parameters with extends to ensure they meet certain constraints.
- **Multiple Bounds:** Use extends with multiple types to combine class and interface bounds.

Using Wildcard Arguments in Generics

Wildcard arguments in Java generics provide a way to handle situations where you want to work with types that are unknown or variable. The wildcard is denoted by ? and allows for more flexible and generic code. Here's how you can use wildcards effectively, especially when dealing with generic classes.

Problem Statement

You have a Stats class that calculates the average of an array of numbers. You want to add a method sameAvg( ) that compares the average of the current Stats object with another Stats object, regardless of the specific numeric type they use (e.g., Integer, Double, Float).

The challenge is that the sameAvg( ) method must work with any Stats object, irrespective of the specific numeric type it holds.

Initial Attempt and Issues

A straightforward approach might look like this:

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

However, this approach only works if ob is of the same type as the invoking Stats object. If the invoking object is Stats<Integer>, then ob must also be Stats<Integer>. This is too restrictive and does not accommodate comparisons between different numeric types.

Solution: Using Wildcards

To create a more flexible sameAvg( ) method, you can use a wildcard argument. The wildcard ? represents an unknown type and allows for greater flexibility in handling different types:

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

Here's what's happening:

- **Stats<?>**: This represents a Stats object with an unknown type. It can be any type that extends Number.
- **average()**: This method computes the average of the numbers in the array and is compatible with the Stats<?> parameter because it uses the wildcard.

Example Code

Here's a full example that demonstrates the use of wildcards with the Stats class:

```
class Stats<T extends Number> {
   T[] nums; // array of Number or subclass

   // Pass the constructor a reference to an array of type Number or subclass.
   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;
   }
}

public class WildcardDemo {
   public static void main(String[] args) {
      Integer[] inums = {1, 2, 3, 4, 5};
      Stats<Integer> iob = new Stats<>(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<>(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<>(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.

Upper Bounded Wildcards (<? extends T>)

An upper bounded wildcard allows you to specify that a type parameter must be a specific type or a subtype of that type. This is useful when you want to work with a range of types that share a common superclass.

Lower Bounded Wildcards (<? super T>)

Lower bounded wildcards allow you to specify that a type parameter must be a specific type or a superclass of that type. This is useful when you want to work with a type that can accept a range of types that share a common superclass.

For example:

```
void processCoords(Coords<? super ThreeD> c) {
   // This can accept Coords<ThreeD> or Coords<FourD>
}
```

In this case, processCoords can handle Coords<ThreeD> and Coords<FourD>, but it could not handle Coords<TwoD>.

```
// 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.
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<>(td);
        System.out.println("Contents of tdlocs.");
        showXY(tdlocs); // OK, is a TwoD
        // The following lines will result in compile-time errors:
        // showXYZ(tdlocs); // Error, not a ThreeD
        // showAll(tdlocs); // Error, not a FourD
        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<>(fd);
        System.out.println("Contents of fdlocs.");
        showXY(fdlocs); // OK, is a FourD
        showXYZ(fdlocs); // OK, is a FourD
        showAll(fdlocs); // OK, is a FourD
    }
```

```
}
```

Upper Bound Wildcard (<? extends T>): Useful for reading values of a specific type or its subtypes.

Lower Bound Wildcard (<? super T>): Useful for writing values to a collection, ensuring that you can add instances of a specific type or its subtypes.

Generic Method Definition:

```
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;
}
```

- <T extends Comparable<T>, V extends T>: This specifies two type parameters:
  - T extends Comparable<T>, meaning T must be a type that can be compared.
  - V extends T, meaning V must be T or a subclass of T.
- T x: The object to search for in the array.
- V[] y: The array to search through.

2. **Method Usage in main()**:

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

   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");
```

```
}
```

- Calls isIn() with Integer and String arrays.
- Demonstrates type safety: trying to call isIn() with incompatible types (like String and Integer) will result in a compile-time error.

Generic Constructor Example

In the provided code, the class GenCons has a generic constructor, which allows it to handle various numeric types. Here's a detailed explanation:

Code

java

Copy code

```
// Use a generic constructor.
class GenCons {
   private double val;

   // Generic constructor
   <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);      // Integer argument
      GenCons test2 = new GenCons(123.5F);  // Float argument

      test.showval();
      test2.showval();
```

```
    }
}
```

Generic Constructor:

```
<T extends Number> GenCons(T arg) {
    val = arg.doubleValue();
}
```

- **Type Parameter**: <T extends Number> indicates that the constructor is generic and can accept any type that extends Number.
- **Constructor Logic**: val = arg.doubleValue(); converts the given argument to a double and assigns it to the instance variable val.

2. **Class GenCons**:
   - **Field**: private double val; stores the converted value.
   - **Method showval()**: Prints the value of val.
3. **Class GenConsDemo**:
   - **Object Creation**:

```
GenCons test = new GenCons(100);      // Integer argument
GenCons test2 = new GenCons(123.5F);  // Float argument
```

test is created with an Integer value 100.

test2 is created with a Float value 123.5F.

Method Calls:

```
test.showval();
test2.showval();
```

These calls print the values stored in val for each GenCons object.

Output

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

This output shows that the constructor correctly handles different numeric types and converts them to double.

Generic Interface Declaration:

```
interface MinMax<T extends Comparable<T>> {
    T min();
```

```
    T max();
}
```

- **T extends Comparable<T>**: This means that the type parameter T must implement the Comparable<T> interface, which is used for comparing objects of type T.
- **Methods**: The MinMax interface declares two methods, min() and max(), which are expected to return the minimum and maximum values of the collection.

2. **Implementation of the Generic Interface**:

```
class MyClass<T extends Comparable<T>> implements MinMax<T> {
    T[] vals;
    MyClass(T[] o) {
        vals = o;
    }

    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;
    }

    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;
    }
}
```

- **MyClass<T extends Comparable<T>>**: This class implements the MinMax<T> interface and provides concrete implementations for the min() and max() methods.

- **Constructor**: MyClass(T[] o) initializes the vals array with the given array of type T.
- **min() and max() Methods**: These methods iterate through the array to find and return the minimum and maximum values, respectively.

3. **Using the Generic Interface and Implementation**:

```
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());
   }
}
```

- **Creating Instances**: MyClass<Integer> and MyClass<Character> instances are created for arrays of Integer and Character, respectively.
- **Method Calls**: Calls to max() and min() methods demonstrate the functionality of finding the maximum and minimum values in the arrays.

Output

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

Key Points

- **Generic Interface**: The MinMax interface uses generics to define a contract for finding minimum and maximum values in a collection.

- **Bounds on Generics**: The type parameter T is constrained by Comparable<T>, ensuring that the type used with the interface supports comparison.
- **Implementing Generics**: MyClass provides implementations for the generic methods of MinMax and uses the type parameter T to work with different types.
- **Type Safety**: By using generics, the code ensures type safety, as only types that implement Comparable can be used with MinMax

Raw Types

Definition and Use:

- **Raw Type**: When a generic class is used without specifying type arguments, it is referred to as a raw type. For example, Gen raw = new Gen(new Double(98.6)); creates a raw type Gen object.
- **Compatibility**: Raw types are compatible with legacy code that does not use generics. This compatibility allows older code to interact with newer generic code without modification.

1. **Creation of Raw Type**:

Gen raw = new Gen(new Double(98.6));

   - No type parameter is specified, so Gen is treated as Gen<Object>. This makes the type of ob unknown, leading to potential type safety issues.

2. **Casting Issues**:

double d = (Double) raw.getob();

   - This cast works because raw was originally created with a Double, but if the type changes (as shown later in the code), this can lead to ClassCastException at runtime.

3. **Assignment Issues**:

strOb = raw; // OK, but potentially wrong

   - raw can be assigned to strOb without compile-time errors, but this is unsafe. If raw actually contains a Double, then trying to get it as a String results in runtime errors.

raw = iOb; // OK, but potentially wrong

   - Similarly, assigning iOb to raw is allowed, but it's unsafe because the type information is lost. Accessing it as a Double later will cause runtime errors.

Type Safety and Warnings

- **Unchecked Warnings**:
   - Raw types generate unchecked warnings when used in a way that could lead to type safety issues. For example:

Gen raw = new Gen(new Double(98.6)); // Unchecked warning

strOb = raw; // Unchecked warning

- **Why No Warning Here?**:

raw = iOb; // OK, but potentially wrong

  - No warning is issued for this line because the assignment does not introduce any new type safety issues beyond those already present.

Example Code:

```
// Demonstrate a raw type.
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() {
      return ob;
   }
}
// Demonstrate raw type.
class RawDemo {
   public static void main(String args[]) {
      // Create a Gen object for Integers.
      Gen<Integer> iOb = new Gen<>(88);
      // Create a Gen object for Strings.
      Gen<String> strOb = new Gen<>("Generics Test");

      // Create a raw-type Gen object and give it a Double value.
      Gen raw = new Gen(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
    }
}
```

Generic Class Hierarchies

Generic Superclass with a Generic Subclass

In a class hierarchy involving generics, a generic class can act as a superclass, and its type parameters must be passed to subclasses.

Example: Generic Superclass with Generic Subclass

```
// 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);
   }
}
```

- **Explanation**:
    - Gen<T> is a generic class with a type parameter T.
    - Gen2<T> extends Gen<T> and must also use the type parameter T to match the superclass.

When you create an instance of Gen2, the type parameter T specified in Gen2 is passed to Gen, ensuring type consistency.

Gen2<Integer> num = new Gen2<>(100);

In this case, Integer is passed as the type parameter T, so ob in Gen will be of type Integer.

Example: Generic Subclass Adding More Type Parameters

```
// A subclass can add its own type parameters.
class Gen<T> {
   T ob;
   Gen(T o) {
      ob = o;
   }
   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[]) {
        Gen2<String, Integer> x = new Gen2<>("Value is: ", 99);
        System.out.print(x.getob());
        System.out.println(x.getob2());
    }
}
```

- **Explanation**:
    - Gen2<T, V> extends Gen<T> and introduces an additional type parameter V.
    - Gen2 can use both type parameters T and V in its own fields and methods.

Non-Generic Superclass with a Generic Subclass

A non-generic class can be the superclass of a generic subclass.

Example: Non-Generic Superclass with Generic Subclass

```
// 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;
    }
    T getob() {
        return ob;
    }
}

// Create a Gen object.
class HierDemo2 {
    public static void main(String args[]) {
        Gen<String> w = new Gen<>("Hello", 47);
        System.out.print(w.getob() + " ");
        System.out.println(w.getnum());
    }
}
```

- **Explanation**:
    - NonGen is a non-generic class with an integer field and method.
    - Gen<T> extends NonGen and adds its own type parameter T.
    - Gen can use both its generic type T and the non-generic members of NonGen.

Using instanceof with Generics

The instanceof operator works with generic types, but with some limitations due to type erasure, which removes generic type information at runtime. Let's break down how instanceof operates within a generic hierarchy using the provided example:

Example: Generic Class Hierarchy

```
class Gen<T> {
   T ob;
   Gen(T o) {
      ob = o;
   }
   T getob() {
      return ob;
   }
}

class Gen2<T> extends Gen<T> {
   Gen2(T o) {
      super(o);
   }
}

class HierDemo3 {
   public static void main(String args[]) {
      // Create a Gen object for Integers.
      Gen<Integer> iOb = new Gen<Integer>(88);
      // Create a Gen2 object for Integers.
      Gen2<Integer> iOb2 = new Gen2<Integer>(99);
      // Create a Gen2 object for Strings.
      Gen2<String> strOb2 = new Gen2<String>("Generics Test");

      // Check instances using instanceof
      if (iOb2 instanceof Gen2<?>)
         System.out.println("iOb2 is instance of Gen2");
      if (iOb2 instanceof Gen<?>)
```

```
        System.out.println("iOb2 is instance of Gen");
    System.out.println();

    if (strOb2 instanceof Gen2<?>)
        System.out.println("strOb2 is instance of Gen2");
    if (strOb2 instanceof Gen<?>)
        System.out.println("strOb2 is instance of Gen");
    System.out.println();

    if (iOb instanceof Gen2<?>)
        System.out.println("iOb is instance of Gen2");
    if (iOb instanceof Gen<?>)
        System.out.println("iOb is instance of Gen");
  }
}
```

Output

iOb2 is instance of Gen2

iOb2 is instance of Gen

strOb2 is instance of Gen2

strOb2 is instance of Gen

iOb is instance of Gen

Explanation

1. **Type Checking with Wildcards (?)**:
    - iOb2 instanceof Gen2<?> returns true because iOb2 is indeed an instance of some form of Gen2 (in this case, Gen2<Integer>). The wildcard ? signifies that iOb2 could be an instance of any Gen2 with any type parameter.
    - iOb2 instanceof Gen<?> returns true because Gen2 is a subclass of Gen, so any instance of Gen2 is also an instance of Gen.
2. **Type Checking for Different Types**:
    - strOb2 instanceof Gen2<?> and strOb2 instanceof Gen<?> both return true because strOb2 is an instance of Gen2 and thus also of Gen.

- iOb instanceof Gen2<?> returns false because iOb is an instance of Gen<Integer> and not Gen2. Therefore, iOb cannot be an instance of any Gen2.

3. **Compile-Time Errors**:
   - if(iOb2 instanceof Gen2<Integer>) cannot be compiled because generic type information is erased at runtime. Java's type system cannot retain specific generic type parameters like Integer after compilation.

Casting

Casting in generic classes is subject to the same type compatibility rules:

```
// Legal
Gen<Integer> g = (Gen<Integer>) iOb2;

// Illegal
Gen<Long> g2 = (Gen<Long>) iOb2;
```

- The cast (Gen<Integer>) iOb2 is legal because iOb2 is indeed an instance of Gen<Integer>.
- The cast (Gen<Long>) iOb2 is illegal because iOb2 is not an instance of Gen<Long>; it's specifically a Gen<Integer>.

Summary

- The instanceof operator can be used with generics, but you can only check for the existence of generic types with wildcards (?), not specific generic types.
- Generic type information is erased at runtime, so checking against specific generic parameters is not possible.
- Casting between generic types requires that the cast be consistent with the actual type of the object.

Type Inference with Generics

Since JDK 7, Java has supported a shorthand syntax for creating instances of generic types, known as the **diamond operator** (<>). This allows the compiler to infer the type arguments based on the context, reducing verbosity and making code easier to read.

Example: Diamond Operator

Consider the generic class:

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

```
    MyClass(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }
    // Additional methods
}
```

Before JDK 7, creating an instance of MyClass required specifying type arguments twice:

MyClass<Integer, String> mcOb = new MyClass<Integer, String>(98, "A String");

With JDK 7 and later, you can use the diamond operator to simplify this:

MyClass<Integer, String> mcOb = new MyClass<>(98, "A String");

Here, <> indicates that the compiler should infer the type arguments from the variable declaration MyClass<Integer, String>.

General Form

The general syntax for using the diamond operator with generic classes is:

java

Copy code

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

- class-name<type-arg-list>: Declares the generic class with type arguments.
- new class-name<>(cons-arg-list): Creates a new instance using the diamond operator, letting the compiler infer the type arguments.

Example with Method Calls

Type inference is also useful when passing generic arguments to methods:

```
boolean isSame(MyClass<T, V> o) {
    if (ob1 == o.ob1 && ob2 == o.ob2) return true;
    else return false;
}

// Usage
if (mcOb.isSame(new MyClass<>(1, "test"))) {
    System.out.println("Same");
```

```
}
```

In this example, new MyClass<>(1, "test") uses the diamond operator, and the type arguments for MyClass are inferred from the method parameter.

Erasure

Java generics use **type erasure** to maintain compatibility with older versions of Java. This means that generic type information is removed during compilation. Instead of using the generic type parameters, the compiler replaces them with their bound types (e.g., Object if no specific bound is set) and inserts appropriate casts.

How Erasure Works

1. **Type Parameters Replaced**: Generic type parameters are replaced by their bounds or Object if no bounds are specified.
2. **Casting**: The compiler inserts casts to ensure type safety.
3. **Compatibility**: This process ensures that generic code is compatible with older, non-generic code.

Because of type erasure, generic type parameters do not exist at runtime, which affects operations like type checks.

Bridge Methods

To handle situations where type erasure results in a mismatch between overridden methods in subclasses, the Java compiler generates **bridge methods**. These are synthetic methods that bridge the gap between the type-erased method signatures of the superclass and the specific signatures in the subclass.

Example: Bridge Method

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

   Gen(T o) {
      ob = o;
   }

   T getob() {
      return ob;
   }
}
```

```
class Gen2 extends Gen<String> {
   Gen2(String o) {
      super(o);
   }

   @Override
   String getob() {
      System.out.print("You called String getob(): ");
      return ob;
   }
}

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

In this example, Gen2 extends Gen<String> and overrides getob() to return a String. Due to type erasure, the JVM expects getob() to return Object. To ensure compatibility, the compiler generates a bridge method:

- java.lang.String getob(); (specific to Gen2)
- java.lang.Object getob(); (bridge method to handle type erasure)

This bridge method allows polymorphic calls to work correctly with the type-erased superclass methods.

Ambiguity Errors in Generics

When working with generics in Java, ambiguity errors can arise due to type erasure. These errors occur when different generic declarations resolve to the same erased type, leading to conflicts in method resolution. Here's a deeper look into this issue:

Example of Ambiguity with Method Overloading

Consider the following class with generic type parameters:

```
class MyGenClass<T, V> {
```

```
    T ob1;
    V ob2;

    // Overloaded methods
    void set(T o) {
        ob1 = o;
    }

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

In this example, MyGenClass has two overloaded set() methods, one taking a parameter of type T and the other of type V. However, ambiguity issues can arise due to type erasure:

1. **Type Erasure Conflict:**
    - Java's type erasure process converts all generic type parameters to their bound type or Object if no bound is specified. For both set(T o) and set(V o), the erased signature is void set(Object o). This makes it impossible to distinguish between the two methods at runtime, causing ambiguity.
2. **Instance Creation Example:**

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

In this case, both T and V are String, so both set() methods effectively become void set(String o), further exacerbating the ambiguity.

3. **Attempt to Restrict Type Parameters:**

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

Even with a restriction like V extends Number, ambiguity can still occur if both type parameters end up being the same bound type.

```
MyGenClass<Number, Number> x = new MyGenClass<Number, Number>();
// Ambiguous method call
```

A practical solution is to use different method names rather than overloading them. Overloading with generics often requires careful consideration and can sometimes indicate a design issue.

Generic Restrictions

There are several important restrictions when using generics in Java:

1. **Cannot Instantiate Type Parameters:** You cannot create an instance of a type parameter directly:

```
class Gen<T> {
   T ob;
   Gen() {
      ob = new T(); // Illegal
   }
}
```

Since T is a placeholder, the compiler cannot know what specific object to create.

2. **Static Members Restrictions:** You cannot declare static members using type parameters of the enclosing class:

```
class Wrong<T> {
   static T ob; // Illegal
   static T getob() { // Illegal
      return ob;
   }
}
```

However, static generic methods with their own type parameters are allowed.

3. **Generic Arrays:**

You cannot create an array of a generic type parameter:

```
class Gen<T extends Number> {
   T vals[];
   Gen(T o, T[] nums) {
      ob = o;
      // vals = new T[10]; // Illegal
      vals = nums; // Legal
   }
```

}

You can declare an array of references to a wildcard generic type:

Gen<?> gens[] = new Gen<?>[10]; // Legal

You cannot create an array of a specific generic type:

// Gen<Integer> gens[] = new Gen<Integer>[10]; // Illegal

4. **Generic Exception Restriction:**

You cannot create generic exceptions:

class MyException<T> extends Exception { // Illegal

}

This restriction is because exceptions are handled differently in Java and cannot be parameterized.