Generics in Java

Generics means parameterized types. The idea is to allow a type (like Integer, String, etc., or user-defined types) to be a parameter to methods, classes, and interfaces.
For example, classes like HashSet, ArrayList, HashMap, etc., use generics very well.

Why Generics?

The **Object** is the superclass of all other classes, and Object reference can refer to any object. These features lack type safety. Generics add that type of safety feature.

Syntax of Generics in Java

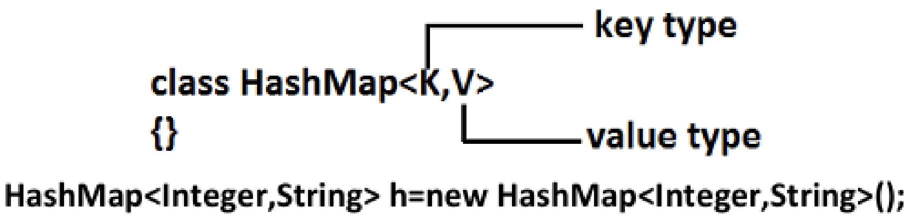
Generics use **type parameters**, usually denoted by single uppercase letters:

- T: Type
- E: Element
- K: Key
- V: Value

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

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

Example:



```
class HashMap<K,V>
{}
HashMap<Integer,String> h=new HashMap<Integer,String>();
```

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

1) **Type-safety**: We can hold only a single type of objects in generics. It doesn?t allow to store other objects.

Without Generics, we can store any type of objects.

```
List list = new ArrayList();
list.add(10);
list.add("10");
```

With Generics, it is required to specify the type of object we need to store.

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

2) **Type casting is not required**: There is no need to typecast the object.

Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

3) **Compile-Time Checking:** It is checked at compile time so problems will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

Syntax to use generic collection

```
ClassOrInterface<Type>
```

Example to use Generics in java

```
ArrayList<String>
```

Generic class

A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.

Let's see a simple example to create and use the generic class.

Creating a generic class:

```
class MyGen<T>{
T obj;
void add(T obj){this.obj=obj;}
T get(){return obj;}
}
```

The T type indicates that it can refer to any type (like String, Integer, and Employee). The type you specify for the class will be used to store and retrieve the data.

Using generic class:

Let's see the code to use the generic class.

```
        package com.generics;

class MyGen<T> {
        T obj;
        void add(T obj) {
                this.obj = obj;
        }
        T get() {
                return obj;
        }
}
class TestGenerics3 {
        public static void main(String args[]) {
                MyGen<Integer> m = new MyGen<Integer>();
                m.add(2);
//m.add("vivek");//Compile time error
                System.out.println(m.get());
        }
}
```

Output

```
2
```

Ex2:

```
package com.generics;
class UDGenerics<T> {
        T obj;
        UDGenerics(T obj) {
                this.obj = obj;
        }
        public void show() {
                System.out.println("The type of object is :" + obj.getClass().getName());
        }
        public T getObject() {
                return obj;
        }
}
class GenericsDemo {
        public static void main(String[] args) {
                UDGenerics<Integer> g1 = new UDGenerics<Integer>(10);
                g1.show();
                System.out.println(g1.getObject());
                UDGenerics<String> g2 = new UDGenerics<String>("bhaskar");
                g2.show();
                System.out.println(g2.getObject());
                UDGenerics<Double> g3 = new UDGenerics<Double>(10.5);
                g3.show();
                System.out.println(g3.getObject());
        }
}
```

Output

```
The type of object is :java.lang.Integer
10
The type of object is :java.lang.String
bhaskar
The type of object is :java.lang.Double
10.5
```

Generic Method

A method can be generic by specifying the type parameter before the return type.

// A Generic Method

```
package com.generics;
public class Main {
        public static <T> void printArray(T[] array) {
                for (T element : array) {
                        System.out.print(element + " ");
                }
                System.out.println();
        }
        public static void main(String[] args) {
                Integer[] intArray = { 1, 2, 3, 4 };
                String[] strArray = { "A", "B", "C" };
                printArray(intArray); // Prints: 1 2 3 4
                printArray(strArray); // Prints: A B C
        }
}
```

Output

```
1 2 3 4
```

A B C

Bounded Type Parameters:

A bounded type parameter has a constraint, limiting the types that can be used for the generic type. The constraint ensures that the type argument must either be a specific class or implement a particular interface, or extend a certain class.

There are two types of bounds:

1. **Upper Bound**: Restricts the type to be a subtype of a given class or interface. This is done using the extends keyword in Java (or the in keyword in Kotlin).
    - `<? extends T>` allows a method to accept arguments of a type that is either T or a subclass of T
    - Example: <T extends Number> means T can be Number or any subclass of Number (like Integer, Double, etc.).

```
public <T extends Number> void printNumbers(T num) {
   System.out.println(num);
}
```

```
package com.generics;
class Box<T> {
        private T value;
        public void setValue(T value) {
                this.value = value;
        }
        public T getValue() {
                return value;
        }
        public void printValue() {
                System.out.println(value);
        }
}
public class UpperBounded {
        public static void printBox(Box<? extends Number> box) {
                // Box can hold a type of Number or its subclasses (Integer, Double, etc.)
                System.out.println(box.getValue());
        }
        public static void main(String[] args) {
                Box<Integer> intBox = new Box<>();
                intBox.setValue(10);
                printBox(intBox); // Allowed: Integer is a subclass of Number
                Box<Double> doubleBox = new Box<>();
                doubleBox.setValue(5.5);
                printBox(doubleBox); // Allowed: Double is a subclass of Number
        }
}
```

Output

```
10
5.5
```

2. **Lower Bound**: Restricts the type to be a supertype of a given class. This is usually used in wildcard type parameters like in Java's generics.
    - Example: <? super Integer> allows Integer or any superclass of Integer.

```
public void addNumbers(List<? super Integer> list) {
  list.add(1); // Can add Integer or any of its supertypes
}
```

```
package com.generics;
class Box<T> {
    private T value;
    public void setValue(T value) {
        this.value = value;
    }
    public T getValue() {
        return value;
    }
}
public class LowerBounded {
    public static void addIntegerToBox(Box<? super Integer> box) {
        // Box can hold Integer or its supertypes (e.g., Number, Object)
        box.setValue(10);
    }
    public static void main(String[] args) {
        Box<Number> numBox = new Box<>();
        addIntegerToBox(numBox); // Allowed: Box<Number> can accept Integer
        System.out.println("numBox value: " + numBox.getValue()); // Prints: 10
        Box<Object> objBox = new Box<>();
        addIntegerToBox(objBox); // Allowed: Box<Object> can accept Integer
        System.out.println("objBox value: " + objBox.getValue()); // Prints: 10
    }
}
```

Output

```
numBox value: 10
```

```
objBox value: 10
```

Unbounded Type Parameters:

An unbounded type parameter means there are no restrictions on the type that can be used for the generic type. It allows any type to be passed to the generic class, interface, or method.

Example: <T> can be any type, without any restrictions.

```
public <T> void printAnything(T obj) {
   System.out.println(obj);

        }
```

Ex1:

```
class Box<T> {
        private T value;
        public void setValue(T value) {
                this.value = value;
        }
        public T getValue() {
                return value;
        }
}
public class Main {
        public static void main(String[] args) {
                Box<Integer> intBox = new Box<>();
                intBox.setValue(10);
                System.out.println(intBox.getValue());
                Box<String> strBox = new Box<>();
                strBox.setValue("Hello");
                System.out.println(strBox.getValue());
        }
}
```

```
10
Hello
```

why generic use wrapper class as parameter not others

1. Generics Cannot Use Primitive Types

Generics in Java are designed to work with objects (reference types), not primitive types. This is due to the way Java's type system works.

- Primitive Types (like int, double, char, etc.) are not objects and don't inherit from Object. Since generics rely on type parameters that must be objects (which all classes in Java inherit from Object), primitive types cannot be used directly as type arguments in generics.
- Wrapper Classes (like Integer, Double, Character, etc.) are objects and are used to "wrap" their respective primitive types. For example, int is wrapped by Integer, and char is wrapped by Character. These wrapper classes are designed to allow the primitive types to be treated as objects and thus work with generics.

Example:

```
// Invalid, primitive types cannot be used as generic parameters
public <T> void processData(T data) {
    // ...
}
// processData(10);  // Compilation error, primitive 'int' cannot be used as type parameter

// Valid, using wrapper class 'Integer' instead of primitive 'int'
public <T> void processData(T data) {
    // ...
}
processData(10);  // 'Integer' is automatically used here
```

2. **Autoboxing and Unboxing**

Java provides a feature called autoboxing and unboxing, which automatically converts between primitive types and their corresponding wrapper classes.

- Autoboxing: This is the automatic conversion of a primitive type to its corresponding wrapper class (e.g., int to Integer).
- Unboxing: The reverse process, where an object of a wrapper class is converted to its corresponding primitive type (e.g., Integer to int).

This allows you to use wrapper classes in generics, but still work with primitive values under the hood. Java automatically converts between the wrapper classes and primitives as needed, so the behavior is seamless.

Example:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(10);  // Autoboxing: int is converted to Integer
int value = numbers.get(0);  // Unboxing: Integer is converted to int
```

3. Object-Oriented Nature of Java

Java is an object-oriented programming (OOP) language, and generics are part of the language's OOP framework. In OOP, data structures are expected to hold references to objects (which inherit from Object), and this includes when you use generics.

- Generics are designed to work with reference types (which are objects), not primitives. Primitive types are simple data types and are not treated as objects in Java. By using wrapper classes, you allow primitive values to be treated as objects, ensuring compatibility with Java's object-oriented system.

4. Flexibility with Generics

Using wrapper classes in generics allows for additional flexibility and operations that are not possible with primitive types:

- Wrapper classes allow you to take advantage of methods that are built into them (e.g., Integer.parseInt(), Double.isNaN(), etc.).
- You can use null values with wrapper classes, which is impossible with primitives (since primitive types always have a default value, such as 0 for int or false for boolean).

For example:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(null);  // Valid, null can be assigned to Integer
Integer num = numbers.get(0);  // num will be null
```

With primitives, you can't store null, which can be important for some use cases (e.g., representing "no value").

5. Compatibility with Collections Framework

The Java Collections Framework (like List, Set, Map) is designed to work with objects. This means that generics are built around object types, and the collections expect reference types (like Integer, String, etc.) rather than primitive types.

- If you try to use primitives directly with generics in collections, you'll run into issues. For example, a List<int> is not allowed, but you can use List<Integer>.

```
// Invalid: Collections only work with objects, not primitives
List<int> numbers = new ArrayList<>();

// Valid: Using Integer wrapper class
List<Integer> numbers = new ArrayList<>();
```

6. **Consistency and Utility Methods**

Wrapper classes provide useful utility methods for converting, comparing, and manipulating values, which is not available for primitive types. For example:

- The Integer class provides methods like Integer.valueOf(), Integer.parseInt(), and Integer.compare().
- The Double class provides methods like Double.isNaN() and Double.isInfinite().

These methods allow more sophisticated operations than are available for primitives, making wrapper classes the preferred choice in generics.

Summary:

- Generics work only with reference types, not primitive types.
- Wrapper classes allow primitive types to be used as objects, making them compatible with generics.
- Autoboxing and unboxing in Java make the transition between primitives and wrapper classes seamless.
- Wrapper classes provide utility methods and allow null values, which primitives cannot.
- The Collections Framework expects objects, so wrapper classes are necessary for working with collections in generics.

Which of the following declarations are allowed?
1) ArrayList l1=new ArrayList();//(valid)
2) ArrayList l2=new ArrayList();//(valid)
3) ArrayList l3=new ArrayList();//(valid)
4) ArrayList l4=new ArrayList();//(valid)
5) ArrayList l5=new ArrayList();(invalid)