## 1. Introduction to generics:

Generic was first introduced in JDK 5. Generics in simple terms refer to the term general. By using generics, it is possible to create a single class that automatically works with different types of data. A class, interface, or method that operates on a parameterized type is called *generic,* as in *generic class* or *generic method.* Parameterized types are important because they enable us to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.

It is important to understand that Java has always given us the ability to create generalized classes, interfaces, and methods by operating through references of type **Object**. Because **Object** is the superclass of all other classes, an **Object** reference can refer to any type object. Thus, in pre-generics code, generalized classes, interfaces, and methods used **Object** references to operate on various types of objects. Object type can be used easily in the programming because it is no longer necessary to explicitly use casts to translate between **Object** and the type of data that is actually being operated upon. With generics, all casts are automatic and implicit. Thus, generics expand our ability to reuse code and let us use code safely and easily.

**Generics programs have following properties;**

- **Generics works only with objects:** When declaring an instance of a generic type, the type argument passed to the type parameter must be a class type. You cannot use a primitive type, such as int or char.
- **Generic type differs based on their type argument:** A key point to understand about generic types is that a reference of one specific version of a generic type is not type compatible with another version of the same generic type. For example, the following line of code is in error and will not compile:

  iOb = strOb; // Wrong!

  Even though both **iOb** and **strOb** are of type **Gen<T>**, they are references to different types because their type parameters differ. This is part of the way that generics add type safety and prevent errors.

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

- **Stronger type checks at compile time:** A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.
- **Elimination of casts:**
  The following code snippet without generics requires casting:

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

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

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

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

## 2. Generics class with parameters

The following program defines two classes. The first is the generic class **Gen**, and the second is **GenDemo**, which uses **Gen**. It has only one parameter T. To run the below program just save it as **Gen.java** not as **Gen<T>.java** .

**// 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());
	}
}
```

**// 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.

		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);
	}
}
```

The output produced by the program is shown here:

**Type of T is java.lang.Integer**

**value: 88**
**Type of T is java.lang.String**
**value: Generics Test**

**In another example,**

```
public class Gen<T> {
	T t;
	public Gen(T t) {
		this.t=t;
	}
	T getT() {
		return t;
	}
	void setT(T t) {
		this.t=t;
	}
	public void displayT() {
		System.out.println(t.getClass().getName());
	}
	public static void main(String[] args) {
		Gen<Integer> a = new Gen<>(8);
		a.displayT();
		System.out.println(a.getT());
	}
}
```

In above example it is shown that how a generic program works with a single parameter T. To specify two or more type parameters, we simply use a comma-separated list. For example, the following **TwoGen** class is a variation of the **Gen** class that has two type parameters:

```
// A simple generic class with two type parameters: T and V.
class TwoGen<T, V> {
	T ob1;
	V ob2;
	// Pass the constructor a reference to an object of type T and an object
of type V.
	TwoGen(T o1, V o2) {
		ob1 = o1;
		ob2 = o2;
	}
	// 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());
	}
	T getOb1() {
		return ob1;
	}
	V getOb2() {
		return ob2;
```

```
	}
}
```

**// Demonstrate TwoGen.**

```
class SimpGen {
	public static void main(String args[]) {
		TwoGen<Integer, String> tgObj = new TwoGen<Integer, String>(88,
"Generics");

		// Show the types.
		tgObj.showTypes();
		// Obtain and show values.
		int v = tgObj.getOb1();
		System.out.println("value: " + v);
		String str = tgObj.getOb2();
		System.out.println("value: " + str);
	}
}
```

The output from this program is shown here:

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

```
public class PhoneBook<K,V> {
	K k;
	V v;

	public PhoneBook(K k, V v) {
		this.k=k;
		this.v=v;
	}

	public K getK() {
		return k;
	}
	public void setK(K k) {
		this.k = k;
	}
	public V getV() {
		return v;
	}
	public void setV(V v) {
		this.v = v;
	}
	public void displayPhoneBook() {
		System.out.println(k.getClass().getName());
```

```
            System.out.println(v.getClass().getName());
      }
      public static void main(String[] args) {
            PhoneBook<String, Long> sunil = new PhoneBook<>("Sunil Bist",9848333333L);
            sunil.displayPhoneBook();
            System.out.println(sunil.getK());
            System.out.println(sunil.getV());
      }
}
```

## 3. General form of a generic class

The generics syntax shown in the preceding examples can be generalized. Here is the syntax for declaring a generic class:

```
public class ClassName<Parameter-Lists>{
      //properties
      //constructors
      //methods
}
```

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

}

Here is the syntax for declaring a reference to a generic class:

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

## 4. Creating a generic method, constructors, interfaces

### 4.1. Creating generic method:

Methods inside a generic class can make use of a class' type parameter and are, therefore, automatically generic relative to the type parameter. However, it is possible to declare a generic method that uses one or more type parameters of its own. Furthermore, it is possible to create a generic method that is enclosed within a non-generic class.

The following program declares a non-generic class called **GenMethDemo** and a static generic method within that class called **isIn( )**. The **isIn( )** method determines if an object is a member of an array. It can be used with any type of object and array as long as the array contains objects that are compatible with the type of the object being sought.

**// Demonstrate a simple generic method.**

```
public class GenMethDemo {
      // Determine if an object is in an array.
      public static <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! no same type! Types must be compatible.
        if(isIn("two", nums))
            System.out.println("two is in strs");
    }
}
```

The output from the program is shown here:

**2 is in nums**
**7 is not in nums**
**two is in strs**
**seven is not in strs**

### 4.2. Creating constructors:

It is possible for constructors to be generic, even if their class is not. For example, consider the following short program:

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

**The output is shown here:**
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.

### 4.3. Creating interfaces:

In addition to generic classes and methods, we can also have generic interfaces. Generic interfaces are specified just like generic classes. The generic interface offers two benefits. First, it can be implemented for different types of data. Second, it allows us to put constraints (that is, bounds) on the types of data for which the interface can be implemented.

Here is the generalized syntax for a generic interface:

**interface *interface-name*<*type-param-list*> { // ...**

Here, *type-param-list* is a comma-separated list of type parameters. When a generic interface is implemented, we must specify the type arguments, as shown below:

**class *class-name*<*type-param-list*> implements *interface-name*<*type-arg-list*> {**

Here is an example. It creates an interface called **MinMax** that declares the methods **min( )** and **max( )**, which are expected to return the minimum and maximum value of some set of objects.

```
// A generic interface example.
// A Min/Max interface.
public interface MinMax<T extends Comparable<T>> {
      T min();
      T max();
}

// Now, implement MinMax
public 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;
      }
}

public 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<Integer>(inums);
            MyClass<Character> cob = new MyClass<Character>(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());
      }
}
```

**The output is shown here:**

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

In above example **MinMax** is declared like this:

```
interface MinMax<T extends Comparable<T>> {

}
```

In general, a generic interface is declared in the same way as is a generic class. In this case, the type parameter is **T**, and its upper bound is **Comparable**, which is an interface defined by **java.lang**. A class that implements **Comparable** defines objects that can be ordered. Thus, requiring an upper bound of **Comparable** ensures that **MinMax** can be used only with objects that are capable of being compared. **Comparable** is also generic. It takes a type parameter that specifies the type of the objects being compared. **MinMax** is implemented by **MyClass**.

The declaration of **MyClass** is as shown below:

```
class MyClass<T extends Comparable<T>> implements MinMax<T> {

}
```

**MinMax** requires a type that implements **Comparable**, so the implementing class (**MyClass** in this case) must specify the same bound. Furthermore, once this bound has been established, there is no need to specify it again in the **implements** clause.

**class MyClass<T extends Comparable<T>> implements MinMax<T extends Comparable<T>> {**
**}**

Once the type parameter has been established, it is simply passed to the interface without further modification. In general, if a class implements a generic interface, then that class must also be generic.

## 5. Polymorphism in generics

To understand generic programming, we need to understand about the base type and generic type. We can make polymorphic references in base type while it is not supported in generic type:
The general syntax is:
List<String> strings = new ArrayList<String>(); // This is valid
List<Object> objects = new ArrayList<String>(); // This is not valid

- List is the base type.
- String is the generic type
- ArrayList is the base type.

There is a simple rule. **The type of declaration must match the type of object we are creating.** We can make polymorphic references for the base type NOT for the generic type. Hence the first statement is valid while second is not.

ArrayList<String> list = new ArrayList<String>(); // Valid. Same base type, same generic type
List<String> list1 = new ArrayList<String>(); // Valid. Polymorphic base type but same generic type
ArrayList<Object> list2 = new ArrayList<String>(); // Invalid. Same base type but different generic type.
List<Object> list3 = new ArrayList<String>(); // Invalid. Polymorphic base type different generic type

**Example:**

- Bike extends Vehicle
- Car extends Vehicle
- Every class has service() method

**Vehicle class**

```
public class Vehicle {
	public void service() {
		System.out.println("Generic vehicle servicing");
	}
}
```

**Bike Class**

```
public class Bike extends Vehicle {
	public void service(){
		System.out.println("Bike specific servicing");
	}
}
```

**Car class**

```
public class Car extends Vehicle {
	public void service() {
		System.out.println("Car specific servicing");

	}
}
```

We have a Mechanic class who can do the servicing of all the vehicles. Here, the concept of polymorphism is well explained in the below example.

```
import java.util.*;
public class Mechanic {
	public void serviceVehicles(List<Vehicle> vehicles){
		for(Vehicle vehicle: vehicles){
			vehicle.service();
		}
	}
	public static void main(String[] args) {
		List<Vehicle> vehicles = new ArrayList<Vehicle>();
		vehicles.add(new Vehicle());
		vehicles.add(new Vehicle());

		List<Bike> bikes = new ArrayList<Bike>();
		bikes.add(new Bike());
		bikes.add(new Bike());

		List<Car> cars = new ArrayList<Car>();
		cars.add(new Car());
		cars.add(new Car());

		Mechanic mechanic = new Mechanic();
		// This works fine.
		mechanic.serviceVehicles(vehicles);

		// Compiler error.
		mechanic.serviceVehicles(bikes);
		//Compiler error.
		mechanic.serviceVehicles(cars);
	}
}
```

Here, mechanic.serviceVehicles(vehicles) works well because we are passing ArrayList<Vehicle> to the method that takes List<Vehicle>.

The mechanic.serviceVehicles(bikes) and mechanic.serviceVehicles(cars) does not compile because we are passing ArrayList<Bike> and ArrayList<Car> to a method that takes List<Vehicle> .