M. Sc. ST 1st SEMESTER

UNIT – V

Generics:

Introduction:

- Since the original 1.0 release in 1995, many new features have been added to Java. The one that has had the most profound impact is generics.
- Introduced by JDK 5, generics changed Java in two important ways.
  - First, it added a new syntactical element to the language.
  - Second, it caused changes to many of the classes and methods in the core API.
- Because generics represented such a large change to the language, some programmers were unwilling to adopt its use.
- However, with the release of JDK 6, generics can no longer be ignored. Simply put, if you will be programming in Java SE 6, you will be using generics.
- Through 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.
- Many algorithms are logically the same no matter what type of data they are being applied to.
  - For example, the mechanism that supports a stack is the same whether that stack is storing items of type Integer, String, Object, or Thread.
- With generics, you can define an algorithm once, independently of any specific type of data, and then apply that algorithm to a wide variety of data types without any additional effort.
- Perhaps the one feature of Java that has been most significantly affected by generics is the Collections Framework.
- A collection is a group of objects. The Collections Framework defines several classes, such as lists and maps, that manage collections.
- The collection classes have always been able to work with any type of object.
- The benefit that generics add is that the collection classes can now be used with complete type safety.
- Thus, in addition to providing a powerful, new language element, generics also enabled an existing feature to be substantially improved.

What Are Generics?

- At its core, the term generics means parameterized types. Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.
- Using generics, it is possible to create a single class, for example, 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.
- It is important to understand that Java has always given you the ability to create generalized classes, interfaces, and methods by operating through references of type Object.
- As Object is the superclass of all other classes, the problem was that they could not do so with type safety.
- Generics add the type safety that was lacking. They also streamline the process, because it is no longer necessary to explicitly employ casts.
- With generics, all casts are automatic and implicit. Thus, generics expand your ability to reuse code and let you do so safely and easily.

A Simple Generics Example:

- Let's begin with a simple example of a generic class.
- The following program defines two classes. The first is the generic class Gen, and the second is GenDemo, which uses Gen.

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

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

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
```

- As the comments in the program state, the assignment

iOb = new Gen<Integer>(88);

- makes use of autoboxing to encapsulate the value 88, which is an int, into an Integer. This works because Gen<Integer> creates a constructor that takes an Integer argument
- Because an Integer is expected, Java will automatically box 88 inside one. Of course, the assignment could also have been written explicitly, like this:

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

- However, there would be no benefit to using this version. Next, the program obtains the value of ob by use of the following line:

```
int v = iOb.getob();
```

```
class Student<T>{
   T age;
   Student(T age){
      this.age = age;
   }
   public void display() {
      System.out.println("Value: "+this.age);
   }
}
public class GenericsExample {
   public static void main(String args[]) {
      Student<Float> std1 = new Student<Float>(25.5f);
      std1.display();
      Student<String> std2 = new Student<String>("25");
      std2.display();
      Student<Integer> std3 = new Student<Integer>(25);
      std3.display();
   }
}
```

```
Output      Value: 25.5      Value: 25      Value: 25
```

Generics Work 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.
  - For e.g., with Gen, it is possible to pass any class type to T, but you cannot pass a primitive type to a type parameter.
- Therefore, the following declaration is illegal:
  - Gen<int> strOb = new Gen<int>(53); // Error, can't use primitive type
- Of course, not being able to specify a primitive type is not a serious restriction because you can use the type wrappers (as the preceding example did) to encapsulate a primitive type..
- Further, Java's autoboxing and auto-unboxing mechanism makes the use of the type wrapper transparent.

Generic Types Differ Based on Their Type Arguments:

- 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, assuming the program just shown, 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.

A Generic Class with Two Type Parameters:

- You can declare more than one type parameter in a generic type. To specify two or more type parameters, 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();
  }
}
```

The 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:

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*);

Bounded Types:

- In the preceding examples, the type parameters could be replaced by any class type. This is fine for many purposes, but sometimes it is useful to limit the types that can be passed to a type parameter.
  - For example, assume that you want to create a generic class that contains a method that returns the average of an array of numbers.
  - Furthermore, you want to use the class to obtain the average of an array of any type of number, including integers, floats, and doubles.
  - Thus, you want to specify the type of the numbers generically, using a type parameter. To create such a class, you might try something like this……

```
// Stats attempts (unsuccessfully) to
// create a generic class that can compute
// the average of an array of numbers of
// any given type.
//
// 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;
  }
}
```

- In Stats, the average( ) method attempts to obtain the double version of each number in the nums array by calling doubleValue( ).
- Because all numeric classes, such as Integer and Double, are subclasses of Number, and Number defines the doubleValue( ) method, this method is available to all numeric wrapper classes.
- The trouble is that the compiler has no way to know that you are intending to create Stats objects using only numeric types.
- Thus, when you try to compile Stats, an error is reported that indicates that the doubleValue( ) method is unknown.
- To solve this problem, you need some way to tell the compiler that you intend to pass only numeric types to T.
- Furthermore, you need some way to ensure that only numeric types are actually passed. To handle such situations, Java provides bounded types.
- When specifying a type parameter, you can create an upper bound that declares the superclass from which all type arguments must be derived.
- This is accomplished through the use of an extends clause when specifying the type parameter, as shown here:

<T extends superclass>

- This specifies that T can only be replaced by superclass, or subclasses of superclass. Thus, superclass defines an inclusive, upper limit.
- You can use an upper bound to fix the Stats class shown earlier by specifying Number as an upper bound, as shown here.

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

```
// 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);

  }
}
```

The output is shown here:

```
Average is 3.0
Average is 3.3
```

Type Parameters in Java Generics:

- The type parameters naming conventions are important to learn generics thoroughly.
- The common type parameters are as follows:
  - T – Type, E – Element, K – Key, N – Number and V – Value.

Wildcard Arguments:

- Instead of the typed parameter in generics (T) you can also use "?", representing an unknown type. You can use **a wild card as a**
  - **Type of parameter, Field and Local field.**
- The only restriction on wilds cards is that you cannot it as a type argument of a generic method while invoking it.
- Java provides 3 types of wild cards namely upper-bounded, lower-bounded, un-bounded.

Upper-bounded wildcards:

- These are similar to the bounded type in generics. Using this you can enable the usage of all the subtypes of a particular class as a typed parameter.
- For example, if want to accept a Collection object as a parameter of a method with the typed parameter as a sub class of the Number class,

you just need to declare a wild card with the Number class as upper bound.

- To create/declare an upper-bounded wildcard, you just need to specify the extends keyword after the "?" followed by the class name.
- Following Java example demonstrates the creation of the upper-bounded wildcard.
- If you pass a collection object other than type that is subclass of Number as a parameter to the sampleMethod() of the above program a compile time error will be generated..

```
import java.util.List;
import java.util.HashSet;
public class UpperBoundExample {
   public static void sampleMethod(Collection<? extends Number> col){
      for (Number num: col) {
         System.out.print(num+" ");
      }
      System.out.println("");
   }
   public static void main(String args[]) {
      ArrayList<Integer> col1 = new ArrayList<Integer>();
      col1.add(24);  col1.add(56);  col1.add(89);
      col1.add(75);  col1.add(36);
      sampleMethod(col1);

      List<Float> col2 = Arrays.asList(22.1f, 3.32f, 51.4f, 82.7f, 95.4f, 625.f);
      sampleMethod(col2);
      HashSet<Double> col3 = new HashSet<Double>();
      col3.add(25.225d);
      col3.add(554.32d);
      col3.add(2254.22d);
      col3.add(445.21d);
      sampleMethod(col3);
   }
}
```

Lower-Bounded wildcards:

- Upper-bounded wildcard enables the usage of all the subtypes of a particular class as a typed parameter.
- Similarly, if we use the lower-bounded wildcards you can restrict the type of the "?" to a particular type or a super type of it.
- For example, if want to accept a Collection object as a parameter of a method with the typed parameter as a super class of the Integer class, you just need to declare a wildcard with the Integer class as lower bound.
- To create/declare a lower-bounded wildcard, you just need to specify the super keyword after the "?" followed by the class name.

- If you pass a collection object other of type other than Integer and its super type as a parameter gives a compile time error..

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.List;
import java.util.Iterator;
public class LowerBoundExample {
   public static void sampleMethod(Collection<? super Integer> col){
      Iterator it = col.iterator();
         while (it.hasNext()) {
         System.out.print(it.next()+" ");
      }
   System.out.println("");
   }
   public static void main(String args[]) {
      ArrayList<Integer> col1 = new ArrayList<Integer>();
      col1.add(24);
      col1.add(56);
      col1.add(89);
      col1.add(75);
      col1.add(36);
      sampleMethod(col1);
      List<Object> col2 = Arrays.asList(22.1f, 3.32f, 51.4f, 82.7f, 95.4f, 625.f);
      sampleMethod(col2);
   }
}
```

Unbounded wildcards:

- An unbounded wildcard is the one which enables the usage of all the subtypes of an unknown type i.e. any type (Object) is accepted as typed-parameter.
- For example, if want to accept an ArrayList of object type as a parameter, you just need to declare an unbounded wildcard.
- To create/declare a Unbounded wildcard, you just need to specify the wild card character “?” as a typed parameter within angle brackets.

```
import java.util.List;
import java.util.Arrays;
public class UnboundedExample {
   public static void sampleMethod(List<?> col){
      for (Object ele : col) {
         System.out.print(ele+" ");
      }
      System.out.println("");
   }
```

```
    public static void main(String args[]) {
        ArrayList<Integer> col1 = new ArrayList<Integer>();
        col1.add(24);   col1.add(56);   col1.add(89);
        col1.add(75);   col1.add(36);

        sampleMethod(col1);

        ArrayList<Double> col2 = new ArrayList<Double>();
        col2.add(24.12d);
        col2.add(56.25d);
        col2.add(89.36d);
        col2.add(75.98d);
        col2.add(36.47d);
        sampleMethod(col2);
    }
}
```

Generic Methods:

- We can write a single generic method declaration that can be called with arguments of different types.
- Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.
- Following are the rules to define Generic Methods……
  1. All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
  2. Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name..
  3. The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
  4. A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Generic Constructors:

- Generic Constructors are similar to methods and just like generic methods we can also have generic constructors in Java though the class is non-generic.
- Since the method does not have return type for generic constructors the type parameter should be placed after the public keyword and before its (class) name.

```
public class GenericMethodTest {
   // generic method printArray
   public static < E > void printArray( E[] inputArray ) {
      // Display array elements
      for(E element : inputArray) {
         System.out.printf("%s ", element);
      }
      System.out.println();
   }

   public static void main(String args[]) {
      // Create arrays of Integer, Double and Character
      Integer[] intArray = { 1, 2, 3, 4, 5 };
      Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
      Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

      System.out.println("Array integerArray contains:");
      printArray(intArray);   // pass an Integer array

      System.out.println("\nArray doubleArray contains:");
      printArray(doubleArray);   // pass a Double array

      System.out.println("\nArray characterArray contains:");
      printArray(charArray);   // pass a Character array
   }
}
```

- Once you define a generic constructor you can invoke it by passing any type (object) as parameter.

```
class Employee{
   String data;
   public <T> Employee(T data){
      this.data = data.toString();
   }
   public void dsplay() {
      System.out.println("value: "+this.data);
   }
}
public class GenericConstructor {
   public static void main(String args[]) {
      Employee emp1 = new Employee("Raju");
      emp1.dsplay();
      Employee emp2 = new Employee(12548);
      emp2.dsplay();
   }
}
```

Output

```
value: Raju
value: 12548
```

```
// 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
```

NOTE:

Java Comparable interface

Java Comparable interface is used to order the objects of the user-defined class. This interface is found in java.lang package and contains only one method named compareTo(Object). It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only. For example, it may be rollno, name, age or anything else.

compareTo(Object obj) method

public int compareTo(Object obj): It is used to compare the current object with the specified object. It returns

- positive integer, if the current object is greater than the specified object.
- negative integer, if the current object is less than the specified object.
- zero, if the current object is equal to the specified object.

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

Generic Interfaces:

- In addition to generic classes and methods, you can also have generic interfaces. Generic interfaces are specified just like generic classes. 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.
- 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..

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, you must specify the type arguments, as shown here:

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

- The generic interface offers two benefits……
  - First, it can be implemented for different types of data.
  - Second, it allows you to put constraints (that is, bounds) on the types of data for which the interface can be implemented.
- In the MinMax example, only types that implement the Comparable interface can be passed to T.

```
// A generic interface example.

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

// Now, 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<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
```

Generic Class Hierarchies:

- Generic classes can be part of a class hierarchy in just the same way as a non-generic class. Thus, a generic class can act as a superclass or be a subclass.
- The key difference between generic and non-generic hierarchies is that in a generic hierarchy, any type arguments needed by a generic superclass must be passed up the hierarchy by all subclasses.

- This is similar to the way that constructor arguments must be passed up a hierarchy.

Using a Generic Superclass:

- Here is 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);
  }
}
```

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

// 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());
  }
}
```

Notice the declaration of this version of **Gen2**, which is shown here:

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

Here, **T** is the type passed to **Gen**, and **V** is the type that is specific to **Gen2**. **V** is used to declare an object called **ob2**, and as a return type for the method **getob2( )**. In **main( )**, a **Gen2** object is created in which type parameter **T** is **String**, and type parameter **V** is **Integer**. The program displays the following, expected, result:

A Generic Subclass:

- It is perfectly acceptable for a non-generic class to be the superclass of a generic subclass. For example, consider this program..

```
// 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; // declare an object of type T

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

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 is 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:

- A method in a generic class can be overridden just like any other method. 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.
class OverrideDemo {
  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");

    System.out.println(iOb.getob());
    System.out.println(iOb2.getob());
    System.out.println(strOb2.getob());
  }
}
```

The output is shown here:

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

Example of Bounded Generic class:

Define a Bounded Generic class GenArray{T [] } with a parameterized constructor and the following methods. Use the same in main method to illustrate use of Generics with Integer, Float and Double class objects.

- A method to display all the elements.
- A method to find the maximum element.
- A method to sort all the elements.

```
import java.lang.Number;
class GeneralArray <T extends Number>{
     T []vals;
     GeneralArray(T []tmp){
          vals=tmp;
     }
     void display(){
          for(int i=0;i<vals.length;i++)
               System.out.println(vals[i]);
     }
```

```
      double findMaxEle(){
            double max=vals[0].doubleValue();
            for(int i=1;i<vals.length;i++){
                  if(vals[i].doubleValue()> max)
                        max=vals[i].doubleValue();
            }
            return max;
      }
      void sortValues(){
            T temp;
            for(int i=0;i<vals.length;i++){
                  for(int j=0;j<vals.length-i-1;j++){
                  if(vals[j].doubleValue()>vals[j+1].doubleValue()){
                              temp=vals[j];
                              vals[j]=vals[j+1];
                              vals[j+1]=temp;
                        }
                  } // inner for loop
            } // outer for loop
      }     // end of method
} // end of class
public class GenericEgRecord {
    public static void main(String[] args) {
      Integer intObj[]={10,20,15,18,30,9};
         GeneralArray<Integer> iar=new GeneralArray<Integer>(intObj);
         System.out.println("Maximum integer is "+iar.findMaxEle());
         System.out.println("Before sorting:");
         iar.display();
         iar.sortValues();
         System.out.println("After sorting:");
         iar.display();

         Float floatObj[]={1.2f,2.3f,1.5f,1.8f,3.0f,9.1f};
         GeneralArray<Float> far = new GeneralArray<Float>(floatObj);
         System.out.println("Maximum integer is "+far.findMaxEle());
```

```
        System.out.println("Before sorting:");
        far.display();
        far.sortValues();
        System.out.println("After sorting:");
        far.display();
    }
}
```

Collection Framework:

- A collection sometimes called a container is simply an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve, manipulate, and communicate aggregate data.
- Typically, they represent data items that form a natural group, such as
  - a poker hand (a collection of cards),
  - a mail folder (a collection of letters), or
  - a telephone directory (a mapping of names to phone numbers).

What is a Collections Framework?

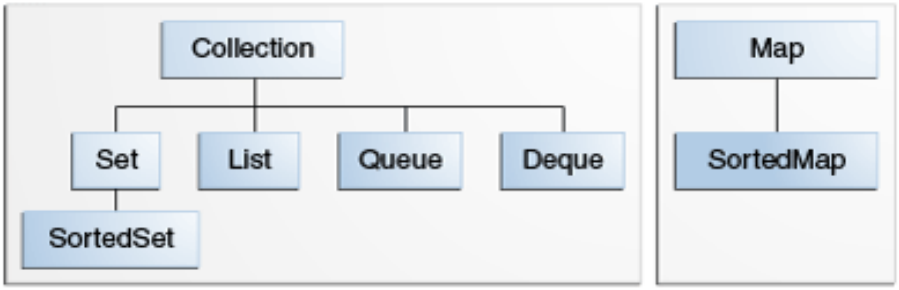
- A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following……
  - **Interfaces:** These are abstract data types that represent collections.
    - Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
  - **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
  - **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.
    - The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.
    - In essence, algorithms are reusable functionality.

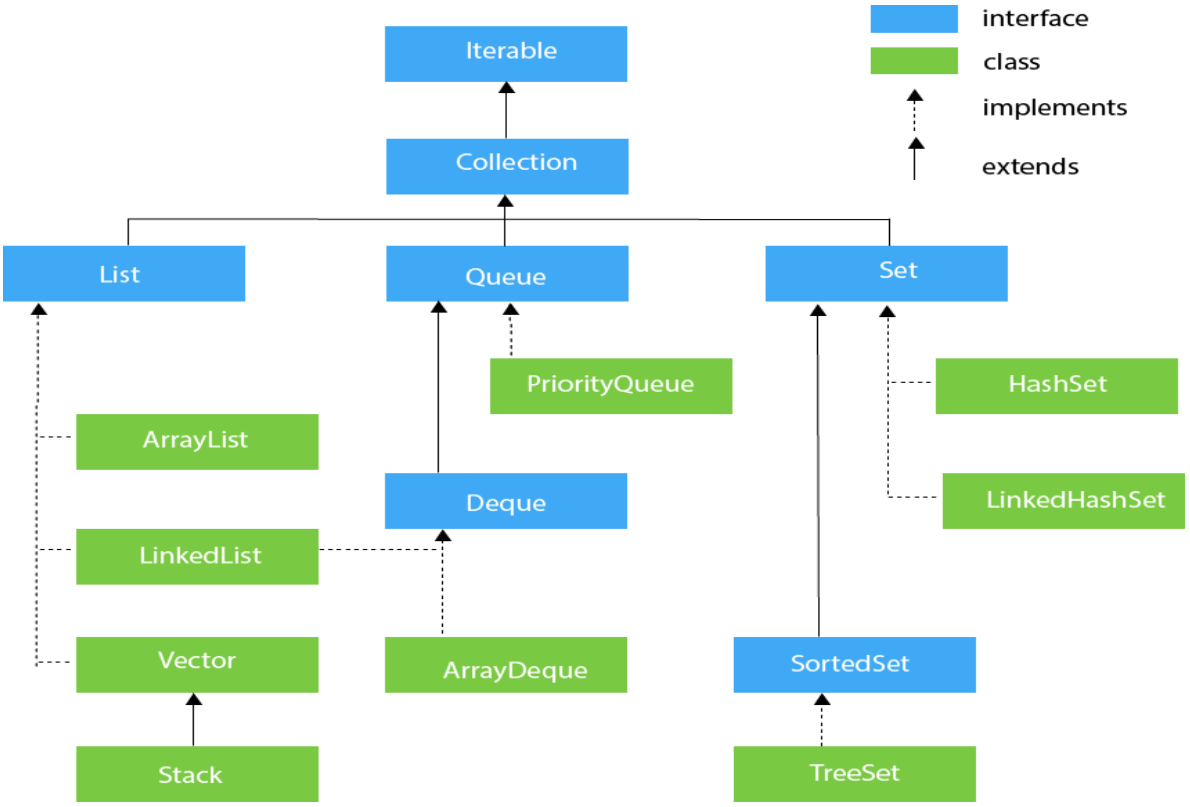
Benefits of the Java Collections Framework:

- Reduces programming effort:
  - By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work.
  - By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.
- Increases program speed and quality:
  - This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms.
  - The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations.
- Allows interoperability among unrelated APIs:
  - The collection interfaces are the language by which APIs pass collections back and forth.
  - If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.
- Reduces effort to learn and to use new APIs:
  - Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections.
  - There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them.
  - With the advent of standard collection interfaces, the problem went away.
- Reduces effort to design new APIs:
  - This is the flip side of the previous advantage.
  - Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces..
- Fosters software reuse:
  - New data structures that conform to the standard collection interfaces are by nature reusable.
  - The same goes for new algorithms that operate on objects that implement these interfaces.

Collection Interfaces:

- The core collection interfaces encapsulate different types of collections, which are shown in the figure below…….





- These interfaces allow collections to be manipulated independently of the details of their representation.
- Core collection interfaces are the foundation of the Java Collections Framework. As you can see in the above figure, the core collection interfaces form a hierarchy.
- Note that all the core collection interfaces are generic. For example, this is the declaration of the Collection interface……

  public interface Collection<E>...

  - The <E> syntax tells you that the interface is generic. When you declare a Collection instance you can and should specify the type of object contained in the collection.
  - Specifying the type allows the compiler to verify (at compile-time) that the type of object you put into the collection is correct, thus reducing errors at runtime.
- To keep the number of core collection interfaces manageable, the Java platform doesn't provide separate interfaces for each variant of each collection type.

- The following list describes the core collection interfaces:

1. **Collection:**
    - It is the root of the collection hierarchy. A collection represents a group of objects known as its elements.
    - The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired.
    - Some types of collections allow duplicate elements, and others do not, similar to ordered and unordered..
    - The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List.
2. **Set:**
    - A collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets.
3. **List:**
    - Its an ordered collection (sometimes called a sequence). Lists can contain duplicate elements.
    - The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).
4. **Queue:**
    - A collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.
    - Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner.
    - Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering.
    - Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll.
    - In a FIFO queue, all new elements are inserted at the tail of the queue.
5. **Deque:**
    - A collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Deque provides additional insertion, extraction, and inspection operations.
    - Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out). In a deque all new elements can be inserted, retrieved and removed at both ends.
6. **Map:**
    - An object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value.
    - If you've used Hashtable, you're already familiar with the basics of Map.

- The last two core collection interfaces are merely sorted versions of Set and Map.
- **SortedSet:**
  - A Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering.
  - Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.
- **SortedMap:**
  - A Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet.
  - Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

The Collection Interface:

- A Collection represents a group of objects known as its elements. The Collection interface is used to pass around collections of objects where maximum generality is desired.
- For example, by convention all general-purpose collection implementations have a constructor that takes a Collection argument.
- This constructor, known as a conversion constructor, initializes the new collection to contain all of the elements in the specified collection.
- Suppose, for example, that you have a Collection<String> c, which may be a List, a Set, or another kind of Collection..
- This expression creates a new ArrayList (an implementation of the List interface), initially containing all the elements in c.

```
List<String> list = new ArrayList<String>(c);
```

- Or – if you are using JDK 7 or later – you can use the diamond operator:

```
List<String> list = new ArrayList<>(c);
```

- Because all collections implement Collection, familiarity with its methods is necessary for a clear understanding of the framework.
- Several of these methods can throw an UnsupportedOperationException, Some of them are as shown in following diagram.
- Java collections only store Objects, not primitive types; however, we can store the wrapper classes. There are three ways to traverse collections:
  - With the for-each construct, By using Iterators and
  - Using aggregate operations.

Sr.No.	Method & Description
1	**boolean add(Object obj)** Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates.
2	**boolean addAll(Collection c)** Adds all the elements of c to the invoking collection. Returns true if the operation succeeds (i.e., the elements were added). Otherwise, returns false.
3	**void clear( )** Removes all elements from the invoking collection.
4	**boolean contains(Object obj)** Returns true if obj is an element of the invoking collection. Otherwise, returns false.
5	**boolean containsAll(Collection c)** Returns true if the invoking collection contains all elements of **c**. Otherwise, returns false.

Sr.No.	Method & Description
6	**boolean equals(Object obj)** Returns true if the invoking collection and obj are equal. Otherwise, returns false.
7	**int hashCode( )** Returns the hash code for the invoking collection.
8	**boolean isEmpty( )** Returns true if the invoking collection is empty. Otherwise, returns false.
9	**Iterator iterator( )** Returns an iterator for the invoking collection.
10	**boolean remove(Object obj)** Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false.
11	**boolean removeAll(Collection c)** Removes all elements of c from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.

for-each Construct:

- The for-each construct allows you to concisely traverse a collection or array using a for loop.
- The following code uses the for-each construct to print out each element of a collection on a separate line.

```
for (Object o : collection)
        System.out.println(o);
```

Iterators:

- An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired.
- You get an Iterator for a collection by calling its iterator method. The following is the Iterator interface.

```
public interface Iterator<E> {
        boolean hasNext();
        E next();
        void remove(); //optional
}
```

  - The hasNext method returns true if the iteration has more elements, and the next method returns the next element in the iteration.
  - The remove method removes the last element that was returned by next from the underlying Collection.
  - The remove method may be called only once per call to next and throws an exception if this rule is violated.
- Note that Iterator.remove is the only safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.
- Use Iterator instead of the for-each construct when you need to……
  - Remove the current element. The for-each construct hides the iterator, so you cannot call remove
  - Iterate over multiple collections in parallel.

```
import java.util.*;
class TestJavaCollection1{
public static void main(String args[]){
        ArrayList<String> list=new ArrayList<String>();//Creating arraylist
        list.add("Ravi");//Adding object in arraylist
        list.add("Vijay");
        list.add("Ravi");
        list.add("Ajay");
        //Traversing list through Iterator
        Iterator itr=list.iterator();
        while(itr.hasNext()){
                System.out.println(itr.next());
        }
        }
}
```

List Interface:

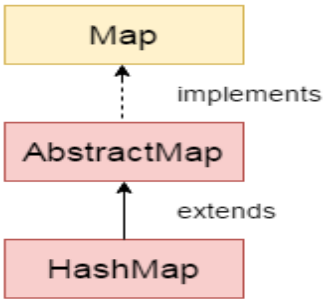
- List interface is the child interface of Collection interface.
- It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.
- List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack. To instantiate the List interface, we must use…

```
List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();
```

```
import java.util.*;
public class TestJavaCollection4{
public static void main(String args[]){
        Stack<String> stack = new Stack<String>();
        stack.push("Ayush");
        stack.push("Garvit");
        stack.push("Amit");
        stack.push("Ashish");
        stack.push("Garima");
        stack.pop();
        Iterator<String> itr=stack.iterator();
        while(itr.hasNext()){
                System.out.println(itr.next());
        }
}
}
```

Java HashMap class:

- Java HashMap class implements the Map interface which allows us to store key and value pair, where keys should be unique.
- If you try to insert the duplicate key, it will replace the element of the corresponding key.
- It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.
- HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key.
- Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.



- Points to remember:
  - Java HashMap contains values based on the key.
  - Java HashMap contains only unique keys.
  - Java HashMap may have one null key and multiple null values.
  - Java HashMap is non synchronized.
  - Java HashMap maintains no order.
  - The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.
- **Declaration of HashMap class:**

  public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable
- HashMap class is available in java.util.HashMap class.
  - K: It is the type of keys maintained by this map.
  - V: It is the type of mapped values.
- The following are the available Constructors of Java HashMap class.

Constructor	Description
HashMap()	It is used to construct a default HashMap.
HashMap(Map<? extends K,? extends V> m)	It is used to initialize the hash map by using the elements of the given Map object m.
HashMap(int capacity)	It is used to initializes the capacity of the hash map to the given integer value, capacity.
HashMap(int capacity, float loadFactor)	It is used to initialize both the capacity and load factor of the hash map by using its arguments.

- The load factor is the measure that decides when to increase the capacity of the Map. The default load factor is 75% of the capacity.
- The threshold of a HashMap is approximately the product of current capacity and load factor.
- You cannot store duplicate keys in HashMap. However, if you try to store duplicate key with another value, it will replace the value.

Sr.No.	Method & Description
1	**void clear()** Removes all mappings from this map.
2	**boolean containsKey(Object key)** Returns true if this map contains a mapping for the specified key.
3	**boolean containsValue(Object value)** Returns true if this map maps one or more keys to the specified value.
4	**Object get(Object key)** Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key.
5	**boolean isEmpty()** Returns true if this map contains no key-value mappings.
6	**Object put(Object key, Object value)** Associates the specified value with the specified key in this map.
7	**putAll(Map m)** Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map.
8	**Object remove(Object key)** Removes the mapping for this key from this map if present.
9	**int size()** Returns the number of key-value mappings in this map.
10	**Set entrySet()** Returns a collection view of the mappings contained in this map.
11	**Set keySet()** Returns a set view of the keys contained in this map.

Example:

```
import java.util.*;
public class HashMapDemo {

   public static void main(String args[]) {

      // Create a hash map
      HashMap hm = new HashMap();

      // Put elements to the map
      hm.put("Zara", new Double(3434.34));
      hm.put("Mahnaz", new Double(123.22));
      hm.put("Ayan", new Double(1378.00));
      hm.put("Daisy", new Double(99.22));
      hm.put("Qadir", new Double(-19.08));

      // Get a set of the entries
      Set set = hm.entrySet();
```

```
      // Get an iterator
      Iterator i = set.iterator();

      // Display elements
      while(i.hasNext()) {
         Map.Entry me = (Map.Entry)i.next();
         System.out.print(me.getKey() + ": ");
         System.out.println(me.getValue());
      }
      System.out.println();

      // Deposit 1000 into Zara's account
      double balance = ((Double)hm.get("Zara")).doubleValue();
      hm.put("Zara", new Double(balance + 1000));
      System.out.println("Zara's new balance: " + hm.get("Zara"));
   }
}
```

****

SAMPLE QUESTIONS:

1. Define Generics. Give its general form and an example.
2. Write java program to illustrate two type parameters in a Generic class.
3. Explain the importance of Bounded type Generic classes with an example.
4. Explain the role of wildcard arguments with an example.
5. Write a java program to illustrate the use of Generic with Integer and String objects.
6. Define a Generic method. Explain the rules to be considered while defining a Generic method.
7. Write a java program to illustrate Bounded types while defining a Generic class.
8. Explain briefly the Generic Constructors and Interfaces with its syntax and an example
9. Write a java program to define a Generic method which will display the contents of an array, use the same to display Integer and Character array.
10. Define Collections. Explain the components of Collection framework.
11. Discuss the benefits of Collection framework.
12. Explain briefly the available different core collection interfaces.
13. Explain any five methods available in Collection interface with its general syntax.

14. Explain how the for-each and iterator constructs can be used to traverse the collections with its general syntax.
15. Write a java program to accept ‘N’ strings from the user and add the same to an ArrayList. Now display only the strings which contains more than three vowels in it by traversing the ArrayList.
16. Define an Iterator. Explain the role of Iterator with an example.
17. Define Java Hashmap class. Explain any five methods with its syntax.

****