Day-16 JA-111 Batch

Topics

- **The java.util.Collections class**
- **The java.util.Arrays class**
- **Contra-variance and Co-variance in Collection**
- **Writing own Generics**

The java.util.Collections Interface

- This class consists exclusively of static methods that operate on or return collections.
- The methods of this class all throw a NullPointerException if the collections or class objects provided to them are null.
- This class has several methods that can be applied on collection and map objects.

Constant	Description
static List EMPTY_LIST	The empty list (immutable).
static Map EMPTY_MAP	The empty map (immutable).
static SetEMPTY_SET	The empty set (immutable).

Method	Description
static <T> boolean addAll(Collection<? super T> c, T... elements)	Adds all of the specified elements to the specified collection.
static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)	Searches the specified list for the specified object using the binary search algorithm.
static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)	Searches the specified list for the specified object using the binary search algorithm.
static <T> void copy(List<? super T> dest, List<? extends T> src)	Copies all of the elements from one list into another.
static <T> void fill(List<? super T> list, T obj)	Replaces all of the elements of the specified list with the specified element.
static int frequency(Collection<?> c, Object o)	Returns the number of elements in the specified collection equal to the specified object.
static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp)	Returns the maximum element of the given collection, according to the order induced by the specified comparator.
static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)	Returns the maximum element of the given collection, according to the natural ordering of its elements.

static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp)	Returns the minimum element of the given collection, according to the order induced by the specified comparator.
static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)	Returns the minimum element of the given collection, according to the natural ordering of its elements.
static <T> boolean replaceAll(List<T> list, T oldVal, T newVal)	Replaces all occurrences of one specified value in a list with another.
static void reverse(List<?> list)	Reverses the order of the elements in the specified list.
static <T> Comparator<T> reverseOrder(Comparator<T> cmp)	Returns a comparator that imposes the reverse ordering of the specified comparator.
static void rotate(List<?> list, int distance)	Rotates the elements in the specified list by the specified distance.
static void shuffle(List<?> list)	Randomly permutes the specified list using a default source of randomness.
static <T extends Comparable<? super T>> void sort(List<T> list)	Sorts the elements of list as determined by comp.
static <T> void sort(List<T> list, Comparator<? super T> c)	Sorts the specified list into ascending order, according to the natural ordering of its elements.
static void swap(List<?> list, int i, int j)	Swaps the elements at the specified positions in the specified list.

An Example

```
Mobile m1 = new Mobile("Z31", 35000, LocalDate.parse("2021-01-01"));
Mobile m2 = new Mobile("N35", 26000, LocalDate.parse("2019-01-01"));
Mobile m3 = new Mobile("A35", 27500, LocalDate.parse("2022-01-01"));

List<Mobile> mobileList = new LinkedList<>();
mobileList.add(m1);
mobileList.add(m2);
mobileList.add(m3);

Mobile m4 = new Mobile("J37", 24000, LocalDate.parse("2020-01-01"));
Mobile m5 = new Mobile("S63", 58000, LocalDate.parse("2021-10-01"));

Collections.addAll(mobileList, m4, m5);
for(Mobile currentMobile: mobileList)
  System.out.println(currentMobile);
```

Output

```
Mobile [ModelName=Z31, price=35000 MFG. Date 2021-01-01
Mobile [ModelName=N35, price=26000 MFG. Date 2019-01-01]
Mobile [ModelName=A35, price=27500 MFG. Date 2022-01-01]
Mobile [ModelName=J37, price=24000 MFG. Date 2020-01-01]
Mobile [ModelName=S63, price=58000 MFG. Date 2021-10-01]
```

```
Collections.sort(mobileList);
for(Mobile currentMobile: mobileList)
  System.out.println(currentMobile);
```

Output
```
Mobile [ModelName=J37, price=24000 MFG. Date 2020-01-01]
Mobile [ModelName=N35, price=26000 MFG. Date 2019-01-01]
Mobile [ModelName=A35, price=27500 MFG. Date 2022-01-01]
Mobile [ModelName=Z31, price=35000 MFG. Date 2021-01-01]
Mobile [ModelName=S63, price=58000 MFG. Date 2021-10-01]
```

```
System.out.println("The " + m2 + " is at index " +
Collections.binarySearch(mobileList, m2));
```

Output
```
The Mobile [ModelName=N35, price=26000 MFG. Date 2019-01-01] is at index
1
```

```
System.out.println("The minimum element according to natural ordering is
" + Collections.min(mobileList));
System.out.println("The maximum element according to natural ordering is
" + Collections.min(mobileList));
```

Output
```
The minimum element according to natural ordering is Mobile
[ModelName=J37, price=24000 MFG. Date 2020-01-01]
The maximum element according to natural ordering is Mobile
[ModelName=J37, price=24000 MFG. Date 2020-01-01]
```

```
System.out.println("The minimum element according to Name is " +
Collections.min(mobileList, new
MobileComparatorByNameDateOrderAmount()));
System.out.println("The maximum element according to Name is " +
Collections.max(mobileList, new
MobileComparatorByNameDateOrderAmount()));
```

Output
```
The minimum element according to Name is Mobile [ModelName=A35,
price=27500 MFG. Date 2022-01-01]
The maximum element according to Name is Mobile [ModelName=Z31,
price=35000 MFG. Date 2021-01-01]
```

```
//call the reverse order method with custom comparator
Collections.sort(mobileList, new
MobileComparatorByDateNameOrderAmount());
for(Mobile currentMobile: mobileList)
  System.out.println(currentMobile);
```

Output

```
Mobile [ModelName=N35, price=26000 MFG. Date 2019-01-01]
Mobile [ModelName=J37, price=24000 MFG. Date 2020-01-01]
Mobile [ModelName=Z31, price=35000 MFG. Date 2021-01-01]
Mobile [ModelName=S63, price=58000 MFG. Date 2021-10-01]
Mobile [ModelName=A35, price=27500 MFG. Date 2022-01-01]
```

```
//swap the element of index 2 with index 4
Collections.swap(mobileList, 2, 4);
//print elements of list
for(Mobile currentMobile: mobileList)
  System.out.println(currentMobile);
```

Output

```
Mobile [ModelName=N35, price=26000 MFG. Date 2019-01-01]
Mobile [ModelName=J37, price=24000 MFG. Date 2020-01-01]
Mobile [ModelName=A35, price=27500 MFG. Date 2022-01-01]
Mobile [ModelName=S63, price=58000 MFG. Date 2021-10-01]
Mobile [ModelName=Z31, price=35000 MFG. Date 2021-01-01]
```

The java.util.Arrays Interface

- ➢ This class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists.
- ➢ The methods in this class all throw a NullPointerException, if the specified array reference is null, except where noted.

Method	**Description**
static <T> List<T> asList(T... a)	Returns a fixed-size list backed by the specified array.
static <T> int binarySearch(T[] a, T key, Comparator<? super T> c)	Searches the specified array for the specified object using the binary search algorithm.
static <T> int binarySearch(T[] a, int fromIndex, int toIndex, T key, Comparator<? super T> c)	Searches a range of the specified array for the specified object using the binary search algorithm.
static <T> T[] copyOf(T[] original, int newLength)	Copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length.
static <T> T[] copyOfRange(T[] original, int from, int to)	Copies the specified range of the specified array into a new array.

static boolean equals(Object[] a, Object[] a2)	Returns true if the two specified arrays of Objects are equal to one another.
static void fill(Object[] a, Object val)	Assigns the specified Object reference to each element of the specified array of Objects.
static <T> void sort(T[] a, Comparator<? super T> c)	Sorts the specified array of objects according to the order induced by the specified comparator.
static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)	Sorts the specified range of the specified array of objects according to the order induced by the specified comparator.

An Example

```
int[] arr = {10,2,5,12,15,16,7,6};
Arrays.sort(arr);
System.out.println(Arrays.toString(arr));

List<String> list= Arrays.asList("one","two","three","four","five");
System.out.println(list);

output:
[2, 5, 6, 7, 10, 12, 15, 16]
[one, two, three, four, five]
```

Covariance and contravariance

Arrays are said to be covariant which basically means that, given the subtyping rules of Java, an array of type T[] may contain elements of type T or any subtype of T. For instance:

```
Number[] numbers = newNumber[3];
numbers[0] = newInteger(10);
numbers[1] = newDouble(3.14);
numbers[2] = newByte(0);
```

But not only that, the subtyping rules of Java also state that an array S[] is a subtype of the array T[] if S is a subtype of T, therefore, something like this is also valid:

```
Integer[] myInts = {1,2,3,4};
Number[] myNumber = myInts;
```

so far all is okay.... but now look at the statement

```
myNumber[0] = 3.14; //attempt of heap pollution so run time error
```

This last line would compile just fine, but if we run this code, we would get an **`ArrayStoreException`** because we're trying to put a double into an integer array. The fact that we are accessing the array through a Number reference is irrelevant here, what matters is that the myNumber is an array of integers. This means that we can fool the compiler, but we cannot fool the run-time type system. And this is so because arrays are what we call a ***reifiable type***. This means that at run-time Java knows that this array was actually instantiated as an array of integers which simply happens to be accessed through a reference of type Number[].

Now, the problem with generic types in Java is that the type information for type parameters is discarded by the compiler after the compilation of code is done; therefore this type information is not available at run time. This process is called **type erasure**. The important point here is that since at run-time there is no type information, there is no way to ensure that we are not committing heap pollution.

```
List<Integer> myInts = new ArrayList<Integer>();
myInts.add(1);
myInts.add(2);
List<Number> myNums = myInts; //compiler error
myNums.add(3.14); //heap polution
```

Generic types are ***non-reifiable***, since at run time we cannot determine the true nature of the generic type. So compiler does not allow assignment. The problem is that now we cannot consider a list of integers to be subtype of a list of numbers, as we saw above, that would be considered unsafe for the type system and compiler rejects it immediately. Evidently, this is affecting the power of polymorphism and it needs to be fixed. The solution consists in learning how to use two powerful features of Java generics known as covariance and contravariance.

The ? (question mark) symbol represents the wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number, e.g., Integer, Float, and double. Now we can call the method of Number class through any child class object.

We can use a wildcard as a type of a parameter, field, return type, or local variable. However, it is not allowed to use a wildcard as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

Covariance

For this case, instead of using a type T as the type argument of a given generic type, we use a wildcard declared as ***? extends T***, where T is a known base type. With covariance we can read items from a structure, but we cannot write anything into it. All these are valid covariant declarations.

```
List<? extends Number> myNums = null;
myNums = new ArrayList<Integer>();
myNums = new ArrayList<Float>();
myNums = new ArrayList<Double>();
Number n = myNums.get(0);     //okay
```

but for the statement

```
myNumst.add(45L); //compiler error
```

This would not be allowed because the compiler cannot determine the actual type of the object in the generic structure. It can be anything that extends Number (like Integer, Double, Long), but the compiler cannot be sure what, and therefore any attempt to write a generic value is considered an unsafe operation and it is immediately rejected by the compiler. So **we can read, but not write.**

Contravariance

For contravariance we use a different wildcard called ***? super T***, where T is our base type. With contravariance we can do the opposite. **We can write, but we cannot read** anything out of it. In this case, the actual nature of the object is List of Object, and through contravariance, we can put a Number in it, basically because a Number has Object as its common ancestor. As such, all numbers are also objects, and therefore this is valid.

However, we cannot safely read anything from this contravariant structure assuming that we will get a number.

```
ArrayList<Number> arr = new ArrayList<>();
arr.add(10);
arr.add(50);
arr.add(30);
List<? super Integer> myNums = arr;
myNums.add(40);
```

but for the statement

```
Number num = myNums.get(0);   //Compile Time Error
```

As we can see, if the compiler allowed us to write this line, we would get a ClassCastException at run time. So, once again, the compiler does not run the risk of allowing this unsafe operation and rejects it immediately.

```
An Example
public static void copy(List<? extends Number> source, List<? super
Number> destiny) {
  for(Number number : source) {
    destiny.add(number);
  }
}
```

Type erasure means that generic type information is not available to the JVM at runtime, only compile time. The reasoning behind major implementation choice is simple – preserving backward compatibility with older versions of Java. When a generic code is compiled into bytecode, it will be as if the generic type never existed. This means that the compilation will:

(i) Replace generic types with objects
(ii) Replaces] bounded types with the first bound class
(iii) Insert the equivalent of casts when retrieving generic objects.

What are bounded parameters?
When we use bounded parameters, we are restricting the types that can be used as generic type arguments. As an example, let's say we want to force our generic type always to be a subclass of animal:

```
public abstract class Cage<T extends Animal> {
   abstract void addAnimal(T animal)
}
```

By using extends, we are forcing T to be a subclass of animal. We could then have a cage of cats:

```
Cage<Cat> catCage;
```

and

```
Cage<Object> objectCage; // Compilation error
```

Generic types in java

Consider the following example

```
package com.masai.sprint3;
```

```
class A{
  static Integer getMaximum(Integer a, Integer b){
    if(a.compareTo(b) > 0){
      return a;
    }
    return b;
  }

  static Double getMaximum(Double a, Double b){
    if(a.compareTo(b) > 0){
      return a;
    }
    return b;
  }

  static String getMaximum(String a, String b){
    if(a.compareTo(b) > 0){
      return a;
    }
    return b;
  }

  public static void main(String args[]){
    System.out.println(A.getMaximum(10, 20));          //30
    System.out.println(A.getMaximum(3.5, 2.5));          //3.5
    System.out.println(A.getMaximum("INDIA", "india"));  //india
  }
}
```

Output

```
20
3.5
india
```

In the above example it is easy to observe that for different data types we are writing different versions despite that the algorithm is same and such kind of codes are actually make you code WET (Write Everything Twice) Such codes are very difficult to maintains and such practice is also discouraged. To overcome such problems it is always good to use generic type instead of the specific type. i.e. generic type are working for all the data types.

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name. The most commonly used type parameter names are:

E - Element (used extensively by the Java Collections Framework)
K - Key

N - Number
T - Type
V - Value
S,U,V etc. - 2nd, 3rd, 4th types

Note: Generics never works for primtive data type it works only for objects yet autoboxing and unboxing plays its role.

```
package com.masai.sprint3;
class A{
  static <E,F> void fun(E a, F b) {
    System.out.println("Class of of object a is " +
a.getClass().getName());
    System.out.println("Class of of object b is " +
b.getClass().getName());
    System.out.println();
  }

  public static void main(String args[]){
    A.fun(10, 30.5);
    A.fun("INDIA", new Exception());
  }
}
```

Output

```
Class of of object a is java.lang.Integer
Class of of object b is java.lang.Double

Class of of object a is java.lang.String
Class of of object b is java.lang.Exception
```

An Example of Generic class

```
package com.masai.sprint3;
import java.util.ArrayList;
import java.util.List;

public class Queue<E>{
  private int front;
  private int rear;
  List<E> list;

  Queue(){
    front = rear = -1;
    list = new ArrayList<>();
  }

  E front(){
    if (front == -1)
      return null;
    return list.get(front);
  }

  E rear(){
    if (rear == -1)
      return null;
    return list.get(rear);
  }

  void enqueue(E X){
    if (this.empty()) {
      front = 0;
      rear = 0;
      list.add(X);
    }else {
      front++;
      if (list.size() > front) {
        list.set(front, X);
      }else{
        list.add(X);
      }
    }
  }

  void dequeue(){
    if (this.empty())
      System.out.println("Queue is already empty");
    else if (front == rear) {
      front = rear = -1;
    }else {
      rear++;
    }
  }

  boolean empty(){
    if (front == -1 && rear == -1)
      return true;
    return false;
  }

  public String toString(){
    if(this.empty())
      return "";

    String text = "";
    for (int i = rear; i < front; i++) {
      text += String.valueOf(list.get(i)) + "->";
    }
    text += String.valueOf(list.get(front));
    return text;
  }

  public static void main(String args[]){
    Queue<Integer> q1 = new Queue<>();
    q1.enqueue(5);
    q1.enqueue(10);
    q1.enqueue(20);
    System.out.println("q1 after enqueue of 3 elements:\n" + q1);
    q1.dequeue();
    System.out.println("q1 after dequeue :\n" + q1);

    Queue<String> q2 = new Queue<>();
    q2.enqueue("ABC");
    q2.enqueue("XYZ");
    q2.enqueue("PQR");
    System.out.println("\nq2 after enqueue of 3 elements:\n" + q2);
    System.out.println("q2 front = " + q2.front() + ", q2 rear = " +
q2.rear());
  }
}
```

Output

```
q1 after enqueue of 3 elements:
5->10->20
q1 after dequeue :
10->20

q2 after enqueue of 3 elements:
ABC->XYZ->PQR
q2 front = PQR, q2 rear = ABC
```