5. Integrated Generic data structure in java

A data structure is an organized collection of data that not only stores information but also facilitates operations for accessing and manipulating the stored data.

In object-oriented programming, everything must be depicted as objects (except java primitive type), including data structures, which are objects designed to store other objects. Defining a data structure essentially involves defining a class. The class representing a data structure should use the data fields for storing information and provide methods to facilitate operations such as searching, insertion, and deletion.

5.1 Collection

Java's generic data structures can be categorized into two classes: ***collections*** and ***maps***. A collection represents a group of objects. A map is a key-value association between objects in one set with objects in another set.

Collections and maps are represented by the interfaces Collection<T> and Map<T,S>. With, "T" and "S" stand for a given object type.

5.1.1 Interfaces and implementation

As seen in the previous chapter, an interface is a set of abstract methods that must be implemented by any class intending to implement this interface. Additionally, the interface can be contains some predefined methods and variables.

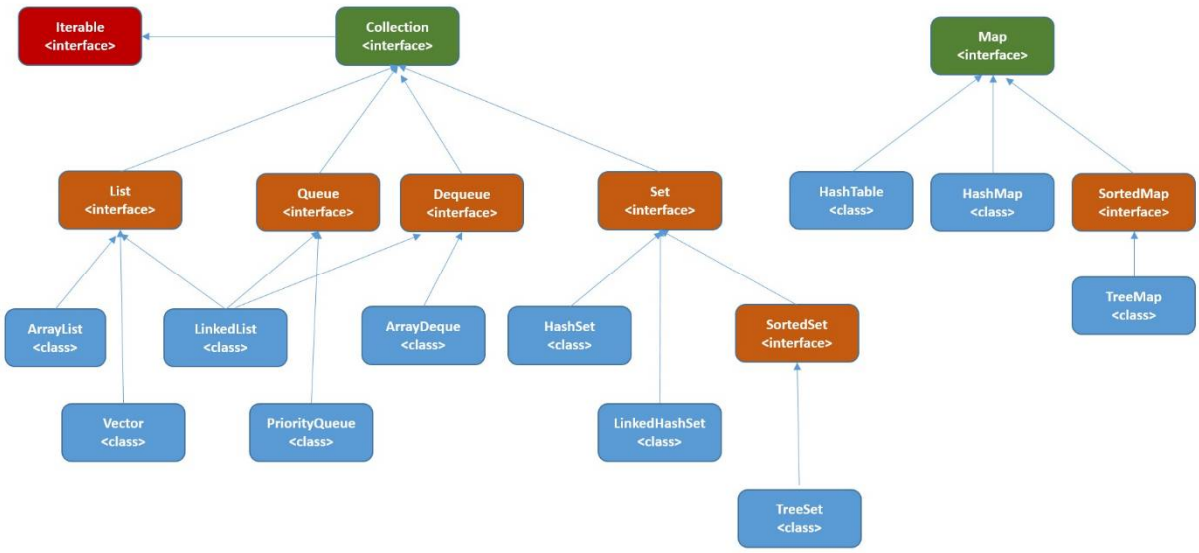

Figure 7 data collection

In Java, generic data structures like **map** and **collection** are represented as interfaces in a package called **util**. So, to use them, you must import this package in your worksheet (import all components) by writing ***import java.util.****. You can specify what you would like to import.

If you want to import one thing. Example **import java.util.map** for using only the map interface.

As we see in above figure, this family of related interfaces and classes is called ***Java* collections *framework***

5.1.2. Generic types in Java

Defining classes that utilize the generic type construct introduced in Java 5.0 requires adopting a new syntax for referencing the class name. These classes and interfaces, which include those in the collections framework, utilize angle brackets that enclose one or more variables (separated by commas) to denote unspecified type names. For instance, you might employ <*E*> or <*K,V*> to signify unspecified type names. Consequently, the names of classes or interfaces implemented with generic types are expressed using the syntax ClassName<*E*>.

5.1.2.1. Set interface

In java, the Set interface is a part of the Java Collections Framework, which defined in the java.util package. It provides a way to store and manipulate non-duplicate elements with any guarantee of the element order. Java provides three implementations of the Set interface in the java.util package: ***HashSet, LinkedHashSet*** and ***TreeSet.***

The Common Methods provided in set interface are:

boolean add(E e): add method perform the addition of the specified element to the set.

boolean remove(Object o): Removes the specified element from the set.

boolean contains(Object o): is boolean method that returns true if the set contains the element.

int size(): it determine the number of elements in the set.

boolean isEmpty(): Returns true if the set is empty.

void clear(): perform a deletion of all elements.

5.1.2.1.1. *HashSet*

A ***HashSet*** implements the Set interface, like a hash table assuming a good hash function. It does not allow duplicate elements, and it does not guarantee the order of elements.

```
import java.util.HashSet;
import java.util.Set;
public class Example {
public static void main(String[] args) {
Set<String> cars = new HashSet<>();
cars.add("Renault");
cars.add("BMW");
cars.add("peugeot");
cars.add("BMW"); // Duplicate, will not be added
```

```
cars.add("Mazda");
// Displaying elements (order may vary)
System.out.println(cars);
// Removing an element
cars.remove(5);
```

The result of this example is **`[Renault, Mazda, BMW, peugeot]`**

5.1.2.1.2. *LinkedHashSet:*

linkedHashSet is a class in Java that extends HashSet and implements the Set interface. It is part of the java.util package and a member of the Java Collections Framework. LinkedHashSet is intended to preserve the order of insertion, which means that it guarantee the order in which elements were added to the set.

Let us consider the previous example:

```
import java.util.HashSet;
import java.util.Set;
public class Example {
public static void main(String[] args) {
Set<String> cars = new LinkedHashSet<>();
cars.add("Renault");
cars.add("BMW");
cars.add("peugeot");
cars.add("BMW"); // Duplicate, will not be added
cars.add("Mazda");
// Displaying elements (order may vary)
System.out.println(cars);
```

The output of this example is **`[Renault, BMW, Peugeot, Mazda]`** which illustrates that the order in which elements were inserted into the LinkedHashSet is maintained.

5.1.2.3. List

A list is a common data structure for storing data in sequential order—for example, a list of available books, a list of students, list of cities …etc. Java provides several operations on lists based on their index positions such as insertion, deletion and retrieval.

To implement a list, you can use one of two methods: dynamic or static. Static method is to use an array to store the elements with fixed size. Dynamic approach is to use a linked structure, which means that each node is dynamically created to store an element. All the nodes are related to form a list.

There are several implementation of list interface such as ArrayList, and LinkedList. We will use ArrayList for example.

5.1.2.3.1 ArrayList

The ArrayList<***E***> class, which is the most popular generic type of the Java collections framework, has a generic type implementation in Java 5.0. The ArrayList class covers the methods for accessing, retrieving and storing objects by using their index position within the

ArrayList form. The ***E*** refers to the name of an interface or class. This type must be specified before calling the constructor.

Example

```
import java.util.ArrayList;
import java.util.List;
public class ArrayListDemo {
public static void main(String args[]) {
// declaration of arraylist of Strings
List<String> stringlist = new ArrayList<>();
stringlist.add("Un");
stringlist.add("deux");
String str = stringlist.get(0);
System.out.println("The 0th element is " + str);
// print the ArrayLists without a loop!
System.out.println(stringlist);}}
```

5.1.2.4. Map

Interface map is presented in ***java.util.Map*** package, which includes the common methods (get and put) for working with maps as in other programming languages. The main purpose of this structure is to define the key-value relationship. The key must be unique, and it is used to recover the corresponding value.

The map interface, is parameterized by two objects T and S Map<T,S>. The first parameter, T, indicates the objects type that is considered as a key in the map; the second parameter, S, specifies the objects type that is considered as a value in the map.

Java offers some implementations of the Map interface, with different characteristics. Some common implementations are:

HashMap: Similar to a ***HashSet***, this implementation does not ensure any specific order. It is designed for fast access and insertion of key-value pairs.

TreeMap: keeps its keys in a sorted order. it provides special methods that make it easy to navigate through the keys.

LinkedHashMap: keeps keys in the order of putting them in, which allows a fast access and known iteration order.

Consequently, java provides several methods that working with the map such as:

- ***values():*** Returns all Collection values in the map
- ***get(key):*** Returns the value associated with the specified key.
- ***put(key, value)***: sets the specified value with the specified key.
- ***remove(key):*** Removes the key and its associated value from the map.

Example

```
import java.util.*;
public class NumberCityMapExample {
    public static void main(String[] args) {
```

```
        Map<Integer, String> numberCityMap = new HashMap<>();
        numberCityMap.put(1, "Adrar");
        numberCityMap.put(28, "Msila");
        numberCityMap.put(19, " setif");
        numberCityMap.put(5, " batna");
        String cityForNumber2 = numberCityMap.get(5);
        System.out.println("City for number 5: " + cityForNumber2);
        // result = City for number 5: batna
        System.out.println("All willaya are:");
        for (Map.Entry<Integer, String> entry : numberCityMap.entrySet()) {
            int number = entry.getKey();
            String city = entry.getValue();
            System.out.println("Number " + number + ": " + city);
        }}}// iterating over key-value pairs
```

5.1.2.5. Queue

In Java, the abstraction of FIFO (First-In-First-Out) structure is represented as staructure that called **queue.** The queue is a data structure that follows this principle, which mean that the first element stored in the queue will be the first one to be deleted.

Here are some Basic Methods provided by Java Queue interface:

add(element) or ***offer(element)***: Adds an element to the end of the queue.

remove() or ***poll()***: Removes and returns the element from the front of the queue.

element() or ***peek()***: Retrieves, but does not remove, the element at the front of the queue.

Java presents the Queue interface in the java.util package, and provides several classes implementing this interface, such as LinkedList and PriorityQueue.

LinkedList: a LinkedList is a class that implements the **List** interface (as seen in the previous section) and ***Deque*** interface. It is a linked list data structure, also known as a chaining list. This means that each element in the list contains two references: one to the previous element and another to the next element.

PriorityQueue: A ***PriorityQueue*** is a data structure in Java that organizes elements based on their priority. Elements with higher priority are dequeued before those with lower priority. It is typically implemented as a binary heap, ensuring efficient insertion and removal of the highest-priority element.

Example

```
import java.util.LinkedList;
import java.util.Queue;
public class SimpleQueueExample {
    public static void main(String[] args) {
        // Creating a Queue using LinkedList
        Queue<String> myQueue = new LinkedList<>();
        // Adding elements to the queue using add
        myQueue.add("Element1");
        myQueue.add("Element2");
        myQueue.add("Element3");

        // Removing elements from the front of the queue using remove
```

```
        while (!myQueue.isEmpty()) {
          String removedElement = myQueue.remove();
          System.out.println("Removed from the front: " + removedElement);}}}
```

5.2. Collection Algorithms

The Java Collections Framework provides certain algorithms as static methods that can be applied to collections and map objects. These collection algorithms are proposed in a class called ***Collections*** within the ***java.util*** package. All these algorithms are better to use them to enhance the coding process.

5.2.1. Fill

The Collections.fill() method is particularly useful when you want to initialize or reset the values of a collection, which it doesn't return any value (void). It is used to replace all elements of a specified collection with a given object value.

Example

Let us consider a vector, which is a collection type that can be used in java

```
import java.util.Collections;
import java.util.Vector;
public class VectorInitializationExample {
    public static void main(String[] args) {
        // Creating a Vector
        Vector<integer> myVector = new Vector<>();
        // Specifying the size of the vector
        int vectorSize = 5;
        // Initializing the Vector with a default value
        String defaultValue = 0;
        // Filling the Vector with the default value
        Collections.fill(myVector, defaultValue);}}
```

5.2.3. Sort

sort(List<T> list): Sorts the specified list into ascending order.

Example:

```
List<Integer> nbers = Arrays.asList(5, 7, 4, 5, 9, 6, 2, 0, 7);
Collections.sort(nbers);
System.out.println("Sorted List: " + nbers);
```

5.2.4. Shuffle:

shuffle(List<?> list): Randomly permutes elements of a list

Example:

```
List<String> Dayname = Arrays.asList("Saturday", "Monday", "Friday",);
 Collections.shuffle(Dayname);
System.out.println("Shuffled List: " + Dayname);
```

5.2.5. Reverse Order:

reverse(List<?> list): Reverses the order of the elements in the specified list.

Example:

```
List<String> names = Arrays.asList("name1", " name2", " name3");
Collections.reverse(names);
System.out.println("Reversed List: " + names);
```

5.2.6. Binary Search:

binarySearch(List<? extends Comparable<? super T>> list, T key): Searches for the specified element in the specified sorted list using a binary search algorithm.

Example:

```
List<Integer> nbers = Arrays.asList(5, 7, 4, 5, 9, 6, 2, 0, 7);
Int key = 2;
Collections.sort(nbers); // sorting the collection
int index = Collections.binarySearch(nbers, key);
System.out.println("Sorted List: " + nbers);
```

5.3. Collections' iterating

In Java, iterating a collection (such as a set, list, or map) can be assured by using numerous methods. The wish of operations and the type of collection are two factors that can be influencing on a selected method. The common used techniques for iterating a collection are:

5.3.1. For-each

The ***for-each*** loop is an easy and elegant way for iterating a collections. It is applicable to classes that implement the ***Iterable*** interface

Example

```
import java.util.ArrayList;
import java.util.List;
public class ForEachLoopExample {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Orange");
        // Using for-each loop to iterate over the elements in the list
        for (String fruit : fruits) {
            System.out.println(fruit); } } }
```

5.3.2. Iterator

The ***Iterator interface*** allow us a sequential iteration of elements in a collection. You can use the ***iterator()*** constructor to obtain an Iterator instance and iterate through the collection.

Example

```
List<String> list = Arrays.asList("hello", "world", "!");
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    String elm = iterator.next();}
```

5.3.3. Massive operations in Java

Massive or Large-scale operations refer to the management and processing of massive quantities of data within Java programs. When performing with Large-scale operations or datasets, some considerations and strategies must be adhered in order to guarantee efficiency performance. Some practices are:

Concurrency and Parallelism: the parallelism and concurrency programming paradigms are often used to accelerate the processing of vast amounts of data. Java provides structures such as ***threads*** and classes of ***java.util.concurrent*** package to work with parallel execution.

Efficient Algorithms: In vast amounts of data, some algorithms are not efficient, especially when working to sorting or selecting data. The developers should be chosen the algorithms based on specific requirements. Some algorithms are more efficient for large datasets that provided by ***java.util.Collections*** package like ***sort*** that use a confirmed efficiency algorithms.

Memory supervision: manage usage memory is essential when dealing with vast amounts of data. For example, the creation of unnecessary temporary objects can reduce pressure on the garbage collector, which leading to improve the performance.