DATE:22/11/24

NAME: Venkatesh Kumar B

JAVA- TASKS

1.**DICTIONARY:**

- Dictionary was part of the original Java 1.0 API, but it was less efficient and had certain design flaws.
- Java 1.2 introduced the Map interface, which is a more modern, flexible, and widely-used way to represent key-value pairs.
- Java 1.2 introduced the Map interface, which is a more modern, flexible, and widely-used way to represent key-value pairs.

HashMap:

A very common implementation, which offers fast lookups and is not thread-safe.

Tree Map:

A Map implementation that maintains the order of keys based on their natural ordering or a specified comparator.

LinkedHashMap:

An implementation that maintains the insertion order of keys

CODE:

```
import java.util.HashMap;

import java.util.Map;


public class SimpleMapExample {
    public static void main(String[] args) {
        // Create a Map using HashMap
        Map<String, String> map = new HashMap<>();


        map.put("name", "John");
        map.put("city", "New York");
        map.put("country", "USA");


        System.out.println("Name: " + map.get("name"));
        System.out.println("City: " + map.get("city"));
        System.out.println("Country: " + map.get("country"));


        if (map.containsKey("city")) {
            System.out.println("City is present in the map.");
        }

        System.out.println("\nMap contents:");
        for (Map.Entry<String, String> entry : map.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }
    }
}
```

15 Interface Name and Explanation:

Comparable:

- Used to compare objects of a class.
- Method: int **compareTo(T o);**

Serializable:

- Marks classes whose objects can be serialized (converted into a byte stream).
- No methods, serves as a marker interface.

Cloneable:

- Used to indicate that a class allows a "clone" of its objects
- No methods, serves as a marker interface.

Iterable:

- Defines a collection that can be iterated over.
- Method: Iterator<T> iterator();

Collection:

- The root interface of the Java Collections Framework, extending Iterable
- Methods: boolean add(E e);, boolean remove(Object o);

List:

- A subinterface of Collection, represents an ordered collection (list).
- It extends Collection and includes classes like ArrayList and LinkedList.

Observer:

- The Observer interface is part of the **Observer Design Pattern**. It is implemented by classes that need to be notified of changes in another object (the subject).

AutoCloseable:

- The AutoCloseable interface is used by classes whose objects should be automatically closed when no longer needed (typically in the try-with-resources statement).
- It requires the close() method to be implemented.

Set:

- The Set interface represents a collection that does not allow duplicate elements. It is part of the **Java Collections Framework** and is implemented by classes such as HashSet, LinkedHashSet, and TreeSet.

Function:

- The Function interface is a part of the Java 8 **functional programming** additions. It represents a function that accepts one argument and produces a result

Consumer:

- The Consumer interface represents a function that accepts a single input argument and returns no result. It's used for operations like processing or modifying the provided argument without producing a return value. Its primary method is accept().

Map:

- The Map interface represents a collection of key-value pairs. It does not extend the Collection interface but is still part of the Collections Framework. Common implementations include HashMap, TreeMap, and LinkedHashMap.

Runnable:

- The Runnable interface represents a task that can be executed concurrently in a thread. It contains a single method run(), which contains the code to be executed when the thread starts.

Cloneable:

- The Cloneable interface is used to indicate that a class can create a field-for-field copy of its objects. Classes that implement this interface must override the clone() method from the Object class to support object cloning

INTERFACE AND CLASS:

FEATURES	INTERFACE	CLASS
Purpose	Defines a contract for implementing classes.	Defines behavior and state of objects.
Instantiation	A class can be instantiated to create objects (unless it's abstract)	An interface cannot be instantiated directly.
Constructor	A class can have constructors to initialize objects	An interface cannot have constructors.
Methods	A class can have both instance methods (with a body) and abstract methods (if it's abstract).	An interface contains abstract methods (no implementation) until Java 8 (where default and static methods are allowed).
Access Modifiers for Methods	Methods in a class can have any access modifier (private, protected, public).	All methods in an interface are implicitly public
Inheritance		An interface can extend multiple interfaces (multiple inheritance).

	A class can inherit from one superclass (single inheritance).	

HashMap:

put(key, value): Adds a key-value pair to the map.

get(key): Retrieves the value associated with the given key.

containsKey(key): Checks if the map contains a specific key.

remove(key): Removes the entry with the specified key.

keySet(): Returns a set view of the keys contained in the map.

values(): Returns a collection view of the values contained in the map.

Code :

```
import java.util.HashMap;


public class HashMapExample {
  public static void main(String[] args) {


    HashMap<Integer, String> map = new HashMap<>();




    map.put(1, "Car");
    map.put(2, "Bus");
    map.put(3, "Flight");
```

```
        map.put(4, "Train");

        System.out.println("Value for key 2: " + map.get(2)); // Output: Bus

        if (map.containsKey(3)) {
            System.out.println("Key 3 is present in the map.");
        }

        if (map.containsValue("Flight")) {
            System.out.println("Flight is present in the map.");
        }

        map.remove(4); // Removes the entry with key 4


        System.out.println("\nIterating through the map:");
        for (Integer key : map.keySet()) {
            System.out.println("Key: " + key + ", Value: " + map.get(key));
        }

        System.out.println("\nSize of map: " + map.size()); // Output: 3
    }
}
```

HashSet :

In Java, a **HashSet** is part of the java.util package and implements the Set interface. It is a collection that does not allow duplicate elements and does not maintain any order of the elements.

add(element): Adds an element to the set. Returns true if the element was added successfully (i.e., it was not already in the set).

remove(element): Removes the specified element from the set

contains(element): Checks if the set contains the specified element.

size(): Returns the number of elements in the set.

clear(): Removes all elements from the set.

isEmpty(): Checks if the set is empty.

Code:

```
import java.util.HashSet;

public class HashSetExample {
  public static void main(String[] args) {
    HashSet<String> set = new HashSet<>();

    set.add("Car");
    set.add("Bus");
    set.add("flight");
    set.add("Car"); // Duplicate value, won't be added
    set.add("Train");

    System.out.println("Set contains 'Bus': " + set.contains("Bus"));
    set.remove("true");


    System.out.println("\nElements in the HashSet:");
    for (String vechicles : set) {
      System.out.println(vechicles);
    }



    System.out.println("\nSize of HashSet: " + set.size());


    System.out.println("Is HashSet empty? " + set.isEmpty());


    set.clear();
    System.out.println("After clearing, is HashSet empty? " + set.isEmpty());
  }
}
```

HashList:

A **HashMap** stores key-value pairs where each key is unique.

An **ArrayList** is a resizable array that allows you to store elements in an ordered list.

If you meant to combine **HashMap** and **ArrayList**, this can be done when you want to store a collection of items in a HashMap and associate those items with a list.

Code:

```
import java.util.ArrayList;
               import java.util.HashMap;
```

```
public class HashListExample {
  public static void main(String[] args) {
    HashMap<Integer, ArrayList<String>> hashList = new HashMap<>();

    ArrayList<String> list1 = new ArrayList<>();
    list1.add("Apple");
    list1.add("Banana");

    ArrayList<String> list2 = new ArrayList<>();
    list2.add("Carrot");
    list2.add("Cucumber");


    hashList.put(1, list1);
    hashList.put(2, list2);


    System.out.println("HashMap contents:");
    for (Integer key : hashList.keySet()) {
      System.out.println("Key: " + key + " -> " + hashList.get(key));
    }


    System.out.println("\nArrayList for key 1: " + hashList.get(1)); // Output: [Apple, Banana]
    hashList.get(1).add("Orange");



    System.out.println("\nUpdated ArrayList for key 1: " + hashList.get(1)); // Output: [Apple, Banana, Orange]
  }
}
```

TreeMap:

In Java, a **TreeMap** is part of the **java.util** package and implements the Map interface.

It is a collection that stores key-value pairs, just like **HashMap,** but with the added feature that it **maintains the keys in a sorted order** (based on the natural ordering of the keys or a custom comparator).

Code:

```
import java.util.TreeMap;


public class TreeMapExample {
  public static void main(String[] args) {

    TreeMap<Integer, String> map = new TreeMap<>();

    // Adding key-value pairs to the TreeMap
    map.put(3, "Apple");
    map.put(1, "Banana");
}


        map.put(4, "Cherry");
        map.put(2, "Date");

        System.out.println("TreeMap: " + map);


        System.out.println("Value for key 3: " + map.get(3));


        System.out.println("Contains key 2: " + map.containsKey(2));


        map.remove(4);


        System.out.println("\nIterating over the TreeMap:");
        for (Integer key : map.keySet()) {
            System.out.println("Key: " + key + ", Value: " + map.get(key));
        }


        System.out.println("\nFirst key: " + map.firstKey());
        System.out.println("Last key: " + map.lastKey());
    }
}
```

Features	Class	Collections
Definition	A class is a blueprint or template for creating objects that define fields (attributes) and methods (functions).	The Collections framework is a group of classes and interfaces in the java.util package designed for storing, managing, and manipulating data in groups (collections) of objects.
Purpose	Used to define the structure and behavior of an object.	Provides various data structures and algorithms to manage groups of objects efficiently.
Instantiation	A class can be instantiated to create objects (unless it is abstract).	Collections themselves (like List, Set, Map) are interfaces, but concrete classes such as **ArrayList, HashSet, HashMap**, etc., can be instantiated.
Role in Java	A class is a core concept in object-oriented programming, used to create custom data types.	Collections are part of the Java API that provides standard implementations for storing and managing collections of objects.
Examples	**class Person, class Car, class Employee**	**List, Set, Map, ArrayList, HashSet, HashMap, TreeMap, etc.**

Throw and Throws:

In Java, the **throw** keyword is used to explicitly throw an exception from a method or a block of code.

When you use `throw`, you are creating an instance of an exception class (either built-in or user-defined) and throwing it to indicate an exceptional condition that needs to be handled.

Syntax:

throw new ExceptionType("Message");

Code:

```
public class ThrowExample {

    public static void setAge(int age) {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative!");
        } else {
            System.out.println("Age is set to: " + age);
        }
    }

    public static void main(String[] args) {
        try {

            setAge(-5);
        } catch (IllegalArgumentException e) {
```

```
      System.out.println("Exception caught: " + e.getMessage());
    }

    setAge(25);
  }
}
```

Throws:

The throws keyword is used to declare that a method might throw one or more exceptions. It allows a method to signal that it might throw certain exceptions during its execution, and the calling method must either handle those exceptions using a try-catch block or declare them further in its own throws clause.

Syntax:

```
public void methodName() throws ExceptionType1, ExceptionType2 {
  // Code that might throw exceptions
}
```

Code:

```
import java.io.*;

        public class ThrowsExample {


          public static void readFile(String fileName) throws IOException {
```

```
        FileReader file = new FileReader(fileName);
        BufferedReader fileInput = new BufferedReader(file);


        String line = fileInput.readLine();
        System.out.println(line);

        fileInput.close();
    }

    public static void main(String[] args) {
        try {
            readFile("nonexistentfile.txt");
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }
}
```

Collections:

The term **Collection** refers to the root interface of the Java Collections Framework, which provides a standard way to store and manipulate groups of objects. It is part of the java.util package and is the parent interface for more specific collection interfaces such as List, Set, and Queue.

To illustrate how the Collection interface works, let's create a simple program using ArrayList, which implements the List interface, a subtype of Collection. We will use basic operations like add(), remove(), and contains().

Feature	Collection Interface	Collections Class
Type	Interface	Utility class (final class with static methods)
Definition	The root interface of the Java Collections Framework. It is used to represent a group of objects.	A utility class that provides static methods to operate on or return collections.
Purpose	To represent a group of objects (elements), such as List, Set, or Queue.	To provide utility methods for performing operations on collections (e.g., sorting, searching).
Instantiation	Cannot be instantiated directly. Used as a type for collections like List, Set, etc.	Cannot be instantiated because it is a utility class with static methods.
Key Methods	add(), remove(), size(), contains(), clear(), etc.	sort(), reverse(), shuffle(), max(), min(), etc.
Common Implementations	List, Set, Queue	Not a collection, but provides operations on collections (e.g., ArrayList, HashSet)
Inheritance	A parent interface of List, Set, and Queue interfaces.	No inheritance, it is a standalone utility class.

Vector:

Vector is a class that implements the **List** interface and provides a growable array of objects.

It is part of the **java.util** package and is similar to an ArrayList, but with a few key differences

Vector is synchronized, meaning it is thread-safe, which can be useful in multi-threaded environments.

Vector can grow dynamically as elements are added to it.