Assingment16

Part A:

1) Difference Between Shallow Copy and Deep Copy in Collections

Shallow Copy:
A shallow copy creates a new collection object but does not create new copies of the objects inside it.
Instead, it copies only the references of the objects. Both collections point to the same objects in memory.

Example:
```
ArrayList<Student> list1 = new ArrayList<>();
list1.add(new Student("Nitin", 22));
ArrayList<Student> list2 = new ArrayList<>(list1);
```

In this case, list1 and list2 are different lists, but they contain references to the same Student object.
If you modify the object using list2, the change will reflect in list1 as well.

Deep Copy:
A deep copy creates a new collection and also creates new copies of all objects inside it.
Each object is independently recreated.

Example:
```
ArrayList<Student> list2 = new ArrayList<>();
for(Student s : list1){
   list2.add(new Student(s.getName(), s.getAge()));
}
```

Now, modifying objects in list2 will not affect list1.

Key Difference:
Shallow copy copies structure only, while deep copy copies structure and objects both.

----

2) How Do You Sort a Collection?

Sorting in Java can be done in multiple ways.

A) Using Comparable (Natural Ordering)
If a class implements Comparable, it defines its natural sorting order.

```
class Student implements Comparable<Student> {
  int age;
  public int compareTo(Student s) {
    return this.age - s.age;
  }
}
Collections.sort(list);
```

B) Using Comparator (Custom Sorting)
Used when we want custom sorting logic or multiple sorting conditions.

```
Collections.sort(list, (a, b) -> a.getName().compareTo(b.getName()));
```

C) Using List.sort() (Java 8+)
```
list.sort((a, b) -> a - b);
```

D) Using Stream API
```
list.stream().sorted().forEach(System.out::println);
```

----

3) How Do You Convert an Array to a List?

Using Arrays.asList():
```
String[] arr = {"A", "B", "C"};
List<String> list = Arrays.asList(arr);
```

Important:
- The returned list is fixed-size.
- You cannot add or remove elements.
- It is backed by the original array.

To create a modifiable list:
```
List<String> list = new ArrayList<>(Arrays.asList(arr));
```

Special Case (Primitive Arrays):
```
int[] arr = {1,2,3};
```

```
Arrays.asList(arr);
```

Use Integer[] instead of int[].

----

4) How Do You Convert a List to an Array?

Using toArray():

```
List<String> list = new ArrayList<>();
list.add("Java");
String[] arr = list.toArray(new String[0]);
```

The parameter new String[0] allows Java to create an array of the correct size.

For primitive array conversion:

```
List<Integer> list = Arrays.asList(1,2,3);
int[] arr = list.stream().mapToInt(i -> i).toArray();
```

----

5) What Are Lambda Expressions?

Lambda expressions were introduced in Java 8.
They are anonymous functions used to implement functional interfaces.

Functional Interface:
An interface that contains only one abstract method.
Examples: Runnable, Comparator, Predicate, Function.

Syntax:

```
(parameters) -> expression
or
(parameters) -> { block of code }
```

Before Java 8:

```
Comparator<Integer> c = new Comparator<Integer>() {
   public int compare(Integer a, Integer b) {
      return a - b;
   }
};
```

Using Lambda:

```
Comparator<Integer> c = (a, b) -> a - b;
```

Advantages of Lambda:
- Reduces boilerplate code
- Improves readability
- Enables functional programming
- Works with Stream API

Part B:

1) Program to Find Common Keys Between Two HashMaps

```
import java.util.*;

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

    HashMap<Integer, String> map1 = new HashMap<>();
    map1.put(1, "Java");
    map1.put(2, "Python");
    map1.put(3, "C++");

    HashMap<Integer, String> map2 = new HashMap<>();
    map2.put(2, "Python");
    map2.put(3, "C++");
    map2.put(4, "JavaScript");

    Set<Integer> commonKeys = new HashSet<>(map1.keySet());
    commonKeys.retainAll(map2.keySet());

    System.out.println("Common Keys: " + commonKeys);
  }
}
```

----

2) Program to Remove a Key from HashMap

```
import java.util.*;

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

    HashMap<Integer, String> map = new HashMap<>();
    map.put(1, "Java");
    map.put(2, "Python");
    map.put(3, "C++");

    map.remove(2);

    System.out.println("After removing key 2: " + map);
  }
}
```

----

3) Program to Check if a Key Exists in HashMap

```
import java.util.*;

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

    HashMap<Integer, String> map = new HashMap<>();
    map.put(1, "Java");
    map.put(2, "Python");

    if (map.containsKey(1)) {
      System.out.println("Key 1 exists");
    } else {
      System.out.println("Key 1 does not exist");
    }
  }
}
```

----

4) Program to Check if a Value Exists in HashMap

```
import java.util.*;
```

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

        HashMap<Integer, String> map = new HashMap<>();
        map.put(1, "Java");
        map.put(2, "Python");

        if (map.containsValue("Java")) {
            System.out.println("Value exists");
        } else {
            System.out.println("Value does not exist");
        }
    }
}
```

----

5) Program to Demonstrate Inheritance

```
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}

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

        Dog d = new Dog();
        d.sound();
        d.bark();
    }
}
```