BSCCS2005: Graded Assignment with Solutions
Week 6

1. Consider the code segment given below. [MCQ:2 points]

```
__________________________________________________;    //LINE 1
names.put(40, "four");
names.put(10, "one");
names.put(30, "three");
names.put(20, "two");
System.out.println(names);
```

Identify the output for the following given cases:

1. `LINE 1` is `Map<Integer, String> names = new LinkedHashMap<Integer, String>();`
2. `LINE 1` is `Map<Integer, String> names = new TreeMap<Integer, String>();`
3. `LINE 1` is `Map<Integer, String> names = new HashMap<Integer, String>();`

- ◯ In case 1: prints all four key-value pairs; however, the order cannot be determined
  In case 2: prints {`10=one, 20=two, 30=three, 40=four`}
  In case 3: prints {`10=one, 20=two, 30=three, 40=four`}
- ◯ In case 1: prints {`10=one, 20=two, 30=three, 40=four`}
  In case 2: prints {`10=one, 20=two, 30=three, 40=four`}
  In case 3: prints {`10=one, 20=two, 30=three, 40=four`}
- ◯ In case 1: prints {`40=four, 10=one, 30=three, 20=two`}
  In case 2: prints {`10=one, 20=two, 30=three, 40=four`}
  In case 3: prints {`40=four, 10=one, 30=three, 20=two`}
- ✓ In case 1: prints {`40=four, 10=one, 30=three, 20=two`}
  In case 2: prints {`10=one, 20=two, 30=three, 40=four`}
  In case 3: prints all four key-value pairs; however, the order cannot be determined

Solution:

- For `HashMap`, the ordering of the keys can not be determined.
- For `TreeMap`, keys are ordered by their insertion order.
- For `LinkedHashMap`, keys are in sorted order.

2. Consider the code given below. [MCQ:2 points]

```
import java.util.*;

public class FClass{
    public static void main(String[] args){
        Map<Character, Integer> frequencyTab
                                = new LinkedHashMap<Character, Integer>();
        String str = "incomprehensibilities";
        for(int i = 0; i < str.length(); i++) {
            Character c = str.charAt(i);
            ------------------------------------------
        }
        for(Map.Entry<Character, Integer> e: frequencyTab.entrySet()) {
            System.out.print("[" + e.getKey() + ", " + e.getValue() + "] ");
        }
    }
}
```

Identify the appropriate option to fill in the blank at `LINE 1`, such that the output is:
`[i, 5] [n, 2] [c, 1] [o, 1] [m, 1] [p, 1] [r, 1] [e, 3] [h, 1] [s, 2] [b, 1] [l, 1] [t, 1]`

- ◯ `frequencyTab.put(c, frequencyTab.get(c) + 1);`
- ◯ `frequencyTab.putIfAbsent(c, frequencyTab.get(c) + 1);`
- ✓ `frequencyTab.put(c, frequencyTab.getOrDefault(c, 0) + 1);`
- ◯ `frequencyTab.putIfAbsent(c, 1);`

Solution: Option-3 is the correct answer for the following reason:
For each character from the `String str`, if character is not present in the the `Map frequencyTab`, add the character as key and `0` as value; else for key equals to the character, increase the value by one.

3. Consider the code given below. [MSQ:2 points]

```
import java.util.*;

public class FClass{
    public static void main(String[] args) {
        Map<Integer, Character> entries = new TreeMap<Integer, Character>();
        entries.put(30, 'b');
        entries.put(40, 'a');
        entries.put(20, 'd');
        entries.put(10, 'c');

        ---------------------------------------    //LINE 1
        for(Character c : values)
            System.out.print(c + " ");
    }
}
```

Identify the appropriate option(s) to fill in the blank at `LINE 1`, such that the output of the above code is

```
a b c d
```

- ◯ `Collection<Character> values = entries.values();`
- ✓ `TreeSet<Character> values = new TreeSet<Character>(entries.values());`
- ✓ `PriorityQueue<Character> values = new PriorityQueue<Character>(entries.values());`
- ◯ `LinkedList<Character> values = new LinkedList<Character>(entries.values());`

Solution: In `TreeMap`, the iterator will visit the key-value pairs in sorted order of the keys. However, if the values are added to a `TreeSet` or a `PriorityQueue`, the iterator will visit the elements in sorted order.

4. Consider the code given below. [MCQ:2 points]

```
import java.util.*;
public class FClass {
    public static void main(String args[]){
        ArrayList<String> sList = new ArrayList<String>();
        sList.add("A");
        sList.add("B");
        ListIterator<String> iter = sList.listIterator();
        if(iter.hasNext()){
            iter.next();
            iter.add("C");
        }
        if(iter.hasPrevious()){
            iter.previous();
            iter.add("D");
        }
        sList.add("E");
        System.out.println(sList);
    }
}
```

What will the output be?

- ◯ `[A, B, C, D, E]`
- ◯ `[A, B, D, C, E]`
- ✓ `[A, D, C, B, E]`
- ◯ `[A, D, B, C, E]`

Solution: The statements `sList.add("A"); sList.add("B");` make the list as `[A, B]`.
The iterator `iter` start from the begining of the list. The statement `iter.next();` skip the first element. The statement `iter.add("C");` makes the list as `[A, C, B]`.
The iterator `iter` positioned after `C` in the list. The statement `iter.previous();` skip the previous element, i.e. `C`. The statement `iter.add("C");` makes the list as `[A, D, C, B]`.
The statement `sList.add("E");`, add `E` to the end of the list, which makes the list as `[A, D, C, B, E]`.

5. Consider the code given below. [MCQ:2 points]

```
import java.util.*;
public class Employee implements Comparable{
    private String name;
    private double salary;
    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }
    public String toString() {
        return "[" + name + " : " + salary + "]";
    }
    public int compareTo(Object e) {
        Employee d = (Employee)e;
        if(salary == d.salary)
            return name.compareTo(d.name);
        else {
            if (d.salary > salary)
                return 1;
            else if(d.salary < salary)
                return -1;
            else
                return 0;
        }
    }
}
public class FClass{
    public static void main(String[] args) {
        TreeSet<Employee> empList = new TreeSet<Employee>();
        empList.add(new Employee("raj", 30000.00));
        empList.add(new Employee("akash", 60000.00));
        empList.add(new Employee("biraj", 60000.00));
        empList.add(new Employee("vinay", 40000.00));
        for(Employee e : empList)
            System.out.println(e);
    }
}
```

What will the output be?

◯
```
[raj : 30000.0]
[vinay : 40000.0]
[akash : 60000.0]
[biraj : 60000.0]
```

- ◯
```
[akash : 60000.0]
[biraj : 60000.0]
[raj : 30000.0]
[vinay : 40000.0]
```
- √
```
[akash : 60000.0]
[biraj : 60000.0]
[vinay : 40000.0]
[raj : 30000.0]
```
- ◯
```
[raj : 30000.0]
[vinay : 40000.0]
[biraj : 60000.0]
[akash : 60000.0]
```

Solution: As per the `compareTo()` method implementation in `class Employee`, the `Employee` objects are first sorted by the `salary` in descending order. When the there is a equality on the `salary`, those `Employee` objects are sorted by the `name` in the ascending order.

6. Consider the code given below. `HashMap m` maps each element of `ArrayList list` to its frequency of occurrence in `list`. [MSQ:2 points]

```
import java.util.*;
public class Example {
  public static void main(String[] args) {
    List<Integer> list = new ArrayList<Integer>();
    list.add(1);
    list.add(1);
    list.add(2);
    list.add(3);
    HashMap<Integer, Integer> m = new HashMap<Integer, Integer>();
    -----------SEGMENT 1-------------------------
    for (HashMap.Entry<Integer, Integer> entry : m.entrySet()){
            System.out.println("Key = " + entry.getKey() +
                                   ", Value = " + entry.getValue());
    }
  }
}
```

Identify the appropriate option to fill in the blank at `SEGMENT 1`, such that the output is:

```
Key = 1, Value = 2
Key = 2, Value = 1
Key = 3, Value = 1
```

○
```
for(Integer i : list){
      m.merge(i, 0, m.get(i)+1);
}
```

○
```
for(Integer i : list){
      m.put(i, m.get(i)+1);
}
```

✓
```
for(Integer i : list){
      m.put(i, m.getOrDefault(i,0)+1);
}
```

✓
```
for(Integer i : list){
      if(m.containsKey(i)){
        m.put(i, m.get(i)+1);
      }
      else{
        m.put(i, 1);
      }
}
```

Solution: For each element from the `ArrayList list`, if element is not present as a key in the `HashMap m`, add the element as key and 0 as its value; else for key equals to the element, increase the value by one.

7. In a temple, there are two types of tickets for devotees waiting for Darshan - free tickets and tickets costing ₹100. The devotees carrying ₹100 tickets are given preference over devotees carrying free tickets. A batch of 5 devotees are let into a special queue, from which they are let inside the temple based on the ticket that they carry. After this, another batch is let in and so on. Consider the Java code that models the sequencing process in the special queue, and answer the question that follows.

```
import java.util.*;

class Devotee implements Comparable{
    String name;
    int ticket_type;

    Devotee(String p_name, int p_type){
       name = p_name;
       ticket_type = p_type;
    }
    public int compareTo(Object a) {
        Devotee d = (Devotee)a;
        if(ticket_type < d.ticket_type)
            return 1;
        else if (ticket_type > d.ticket_type)
            return -1;
        else return 0;
    }
}
public class SpecialQueue{
    public static void main(String[] args){
        Devotee[] dev_arr = new Devotee[]{new Devotee("Pavya",0),
                            new Devotee("Arya",100),new Devotee("Sana",0),
                            new Devotee("Meenu",0),new Devotee("Naina",100)};

        PriorityQueue<Devotee> specialQ = new PriorityQueue<Devotee>();
        ________________________ //Hidden Lines
    }
}
```

What should the Java code in the given blanks at `Hidden Lines` be, so that the code will print the names and ticket types of all devotees who carry ₹100 tickets before those with free tickets?

```
◯ for (int i = 0; i<5; i++){
      specialQ.add(dev_arr[i]);
      Devotee d = specialQ.poll();
```

```
        System.out.println(d.name + " "+ d.ticket_type);
    }
```

○
```
    for (int i = 0; i<5; i++){
        specialQ.add(dev_arr[i]);
    }
    for (int i = 0; i<5; i++){
        Devotee d = specialQ.poll();
        System.out.println(specialQ.poll().name + " "+
                    specialQ.poll().ticket_type);
    }
```

○
```
    for (int i = 0; i<5; i++){
        specialQ.add(dev_arr[i]);
        Devotee d = specialQ.poll();
        System.out.println(specialQ.poll().name + " "+
                    specialQ.poll().ticket_type);
    }
```

✓
```
    for (int i = 0; i<5; i++){
        specialQ.add(dev_arr[i]);
    }
    for (int i = 0; i<5; i++){
        Devotee d = specialQ.poll();
        System.out.println(d.name + " "+ d.ticket_type);
    }
```

Solution:
In Option 1, adding and polling takes place in the same loop. The poll method returns what is currently the max among the ones in the heap, and hence in this case may not yield the right answer.
In Option 2, specialQ.poll() is invoked twice in the sentence. This will generate a NullPointerException because every time the poll() method is invoked, it returns and removes the current element.
Option 3 has both the issues as in Option 1 and Option 2.
In Option 4, the current 5 devotees are added to the priority queue. Using another for loop, the devotee objects are pulled out based on the priority of the ticket type.

8. Consider the following code. [MCQ:2points]

```
import java.util.*;
public interface Account{
    default void showBalance() {
        System.out.println("Abstract Account");
    }
}
public class SavingsAccount implements Account{
    double balance;
    public SavingsAccount(double amt){
        balance = amt;
    }
    public void showBalance() {
        System.out.println("SavingsAccount balance: " + balance);
    }
}
public class CurrentAccount implements Account{
    double balance;
    public CurrentAccount(double amt){
        balance = amt;
    }
    public void showBalance() {
        System.out.println("CurrentAccount balance: " + balance);
    }
}
public class Test2{
    public static void main(String args[]) {
        ArrayList<SavingsAccount> acc1 = new ArrayList<SavingsAccount>();
        acc1.add(new SavingsAccount(10000.0));
        acc1.add(new SavingsAccount(20000.5));

        ____________________________ // LINE 1
        acc2.add(0,new CurrentAccount(50000.0));
        ____________________________ // LINE 2

        for(Account t : acc2) {
            t.showBalance();
        }
    }
}
```

If the code given above produces the output:

```
CurrentAccount balance:  50000.0
SavingsAccount balance:  20000.5
```

What should be the correct choice for LINE 1 and LINE 2?

- [✓] LINE 1:
  `ArrayList<Account> acc2 = new ArrayList<Account>(acc1);`
  LINE 2:
  `acc2.remove(1);`
- ◯ LINE 1:
  `ArrayList<CurrentAccount> acc2 = new ArrayList<CurrentAccount>(acc1);`
  LINE 2:
  `acc2.remove(1);`
- ◯ LINE 1:
  `ArrayList<SavingsAccount> acc2 = new ArrayList<SavingsAccount>(acc1);`
  LINE 2:
  `acc2.remove(2);`
- ◯ LINE 1:
  `ArrayList<Account> acc2 = new ArrayList<Account>(acc1);`
  LINE 2:
  `acc2.remove(2);`

Solution:
Interface Account is implemented by both the classes SavingsAccount and CurrentAccount. In order to call the respective version of the overridden method `showBalance()` the ArrayList should store `Account` type reference.

the statement `acc2.add(0,new CurrentAccount(50000.0));` adds the new object of CurrentAccount at position 0 in the acc2 List, so inorder to remove the object of SavingsAccount corresponding to balance 10000.0 we should use `acc2.remove(1)`.

9. Match the following regarding a LinkedList object O.

[MCQ : 2 points]

A.`getFirst()`	I. If the list is empty, it returns null, else it returns the first element of O.
B.`peekFirst()`	II. If the list is empty, it returns null, else it returns the last element of O.
C.`removeLast()`	III. If the list is empty, it throws `NoSuchElementException`, else it returns the first element of O.
D.`peekLast()`	IV. If the list is empty, it throws `NoSuchElementException`, else it returns the last element of O.

◯ A-II,B-I,C-IV,D-III.

✓ A-III,B-II,C-IV,D-II.

◯ A-III,B-IV,C-I,D-II.

◯ A-III,B-I,C-II,D-IV.

Solution: `getFirst()` and `peekFirst()` are both used to get the first element of a `LinkedList` object. The difference is that if the object is empty, `getFirst()` throws an exception, whereas `peekFirst()` returns null.
`getLast()` and `peekLast()` are both used to get the last element of a `LinkedList` object. The difference is that if the object is empty, `getLast()` throws an exception, whereas `peekLast()` returns null.

10. Consider the Java program given below, and choose the correct options for Line 1 and 2.

[MCQ : 2 points]

```
import java.util.*;
public class MapEx{
    public static void main(String[] args) {
        HashMap<String,String> map1=new HashMap<String,String>();
        map1.put("India","Delhi");
        map1.put("Srilanka","Colombo");
        map1.put("Australia","Sydney");
        //Line 1
        //Line 2
        Iterator<String> it1=keys.iterator();
        Iterator<String> it2=values.iterator();
        System.out.println("Keys are:");
        while(it1.hasNext())
           System.out.println(it1.next());
        System.out.println("values are:");
        while(it2.hasNext())
           System.out.println(it2.next());
    }
}
```

- [✓ Line 1: `Set<String> keys=map1.keySet();`
  Line 2: `Collection<String> values=map1.values();`]
- ◯ Line 1: `Set<String> keys=map1.keySet();`
  Line 2: `Set<String> values=map1.values();`
- [✓ Line 1: `Collection<String> keys=map1.keySet();`
  Line 2: `Collection<String> values=map1.values();`]
- ◯ Not possible to get keys and values from map object separately.

Solution: `keySet()` method return `Set` as return type, you can assign it to either `Set` or `Collection` interface, because `Set` extended from `Collection` interface. `values()` method return `Collection` as return type, you can assign to `Collection` object directly.