BSCCS2005: Practice Assignment with Solutions
Week 8

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

```
public class ComplexNum implements Cloneable{
    private double r, i;
    public ComplexNum(double r, double i) {
        this.r = r;
        this.i = i;
    }
    public void setRe(double r) {
        this.r = r;
    }
    public void setIm(double i) {
        this.i = i;
    }
    public String toString() {
        return "(" + r + " + " + i + "i)";
    }
    public Object clone() throws CloneNotSupportedException{
        return super.clone();
    }
}

public class FClass{
    public static void main(String[] args) {
        try {
            ComplexNum c1 = new ComplexNum(10.0, 20.0);
            ComplexNum c2 = c1;
            ComplexNum c3 = (ComplexNum)c1.clone();
            c1.setRe(100.0);
            c1.setIm(200.0);
            System.out.println(c1 + " , " + c2 + ", " + c3);
        }
        catch(CloneNotSupportedException e) {
            System.out.println("clone() not supported");
        }
    }
}
```

What will the output be?

- ◯ (100.0 + 200.0i) , (100.0 + 200.0i), (100.0 + 200.0i)
- √ (100.0 + 200.0i) , (100.0 + 200.0i), (10.0 + 20.0i)
- ◯ (100.0 + 200.0i) , (10.0 + 20.0i), (10.0 + 20.0i)
- ◯ clone() not supported

Solution: Since, `c1` and `c2` refers to the same object, any change to `c1` would be reflected on `c2`. However, `c3` creates a separate copy of the `c1` object. Thus, the changes in `c1` are not reflected on `c3`.

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

```
public class Product{
    private String prodname;
    private double prodprice;
    public Product(String prodname, double prodprice) {
        this.prodname = prodname;
        this.prodprice = prodprice;
    }
    public void updateProduct(String prodname, double prodprice) {
        this.prodname = prodname;
        this.prodprice = prodprice;
    }
    public String toString() {
        return prodname + " : " + prodprice;
    }
}

public class Order implements Cloneable{
    private int orderid;
    private Product prod;
    public Order(int orderid, Product prod) {
        this.orderid = orderid;
        this.prod = prod;
    }
    public Order clone() throws CloneNotSupportedException{
        return (Order)super.clone();
    }
    public void updateOrder(int orderid, String prodname, double prodprice) {
        this.orderid = orderid;
        prod.updateProduct(prodname, prodprice);
    }
    public String toString() {
        return orderid + " : " + prod;
    }
}

public class FClass{
    public static void main(String[] args) {
        try {
            Order od1 = new Order(1001, new Product("Pen", 15.0));
            Order od2 = od1.clone();
            od1.updateOrder(1010, "Pencil", 20.0);
            System.out.print(od1 + ", " + od2);
```

```
        }
        catch(CloneNotSupportedException e) {
            System.out.println("clone() not supported");
        }
    }
}
```

What will the output be?

- √ `1010 : Pencil : 20.0, 1001 : Pencil : 20.0`
- ◯ `1010 : Pencil : 20.0, 1001 : Pen : 15.0`
- ◯ `1010 : Pencil : 20.0, 1010 : Pencil : 20.0`
- ◯ `clone() not supported`

Solution: Since the instance variable `prod` in `Order` class is a reference type, The bitwise copy made by `od2.clone()` copies the reference. Thus, although `od1` and `od2` refers to different object, both the object hold a reference `prod` referring to the same memory holding `Product` object. Thus, update on `od1.orderid` would not be reflected on `o2.orderid`; however, update on `od1.prod` would be reflected on `o2.prod`.

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

```
public class Address implements Cloneable{
    private int houseno;
    private String city;
    public Address(int houseno, String city) {
        this.houseno = houseno;
        this.city = city;
    }
    public void updateAddress(int houseno, String city) {
        this.houseno = houseno;
        this.city = city;
    }
    public String toString() {
        return houseno + " : " + city;
    }
    public Address clone() throws CloneNotSupportedException{
        return (Address)super.clone();
    }
}

public class Person implements Cloneable{
    private String name;
    private Address addr;
    public Person(String name, Address addr){
        this.name = name;
        this.addr = addr;
    }
    public Person clone() throws CloneNotSupportedException{
        Person newPer = (Person)super.clone();
        newPer.addr = addr.clone();
        return newPer;
    }
    public void updatePerson(String name, int houseno, String city) {
        this.name = name;
        addr.updateAddress(houseno, city);
    }
    public String toString() {
        return name + " : " + addr;
    }
}

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

```
        try {
            Person per1 = new Person("binit", new Address(100, "Delhi"));
            Person per2 = per1.clone();
            per1.updatePerson("rajiv", 200, "Kolkata");
            System.out.print(per1 + ", " + per2);
        }
        catch(CloneNotSupportedException e) {
            System.out.println("clone() not supported");
        }
    }
}
```

What will the output be?

- ◯ `rajiv : 200 : Kolkata, binit : 200 : Kolkata`
- ◯ `rajiv : 200 : Kolkata, rajiv : 200 : Kolkata`
- √ `rajiv : 200 : Kolkata, binit : 100 : Delhi`
- ◯ `clone() not supported`

Solution: The `clone()` method of `Person` performs a deep copy, i.e. it clones the `Address` type instance variable `addr` also. Thus, any update on `per1` would not be reflected on `per2`.

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

```
public class FClass{
    public static void main(String[] args) {
        ________________________    //LINE 1
        for(var i : iArr)
            System.out.print(i + " ");
    }
}
```

At `LINE 1`, identify the appropriate definition of the array `iArr`, such that the output is:
`10 20 30 40 50`

- √ `int[] iArr = new int[] {10, 20, 30, 40, 50};`
- √ `var iArr = new int[] {10, 20, 30, 40, 50};`
- ◯ `var[] iArr = new int[] {10, 20, 30, 40, 50};`
- ◯ `var iArr = {10, 20, 30, 40, 50};`

Solution: Among the given options, option-3 and option-4 are wrong syntax using `var`.

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

```
import java.util.*;
public class Person{
    private String name;
    private int age;
    public Person(String n, int a) {
        name = n;
        age = a;
    }
    public int getAge(){
        return age;
    }
    public void print() {
        System.out.println(name + " : " + age);
    }
}

public class FClass{
    public static void main(String[] args) {
        var list = new ArrayList<Person>();
        list.add(new Person("Robin", 33));
        list.add(new Person("Indra", 76));
        list.add(new Person("Smita", 35));
        list.add(new Person("Rikki", 26));
        Collections.sort(list, ________________________);
        for(var l: list)
            l.print();
    }
}
```

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

```
Indra : 76
Smita : 35
Robin : 33
Rikki : 26
```

◯ `(a, b) -> a.getAge() - b.getAge()`

√ `(a, b) -> b.getAge() - a.getAge()`

◯ `(Person a, Person b) -> a.getAge() - b.getAge()`

√ `(a, b) -> { return b.getAge() - a.getAge(); }`

Solution: Option 1 – lamda expression `(a, b) -> a.getAge() - b.getAge()` and option 3 – lamda expression `(Person a, Person b) -> a.getAge() - b.getAge()` are syntactically correct. However, they sort the `Person` objects in ascending order of `age`.
Option 2 – lamda expression `(a, b) -> b.getAge() - a.getAge()` and option 4 – lamda expression `(a, b) -> { return b.getAge() - a.getAge(); }` are syntactically correct and they sort the `Person` objects in descending order of `age`.

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

```
import java.util.*;
public interface Operatable<T extends Number>{
    public T operate(T a);
}

public class FClass{
    ------------------------------------------------ {    //LINE 1
        for(int i = 0; i < x.length; i++)
            x[i] = ob.operate(x[i]);
    }
    public static void main(String[] args) {
        Integer[] iArr = new Integer[]{1, 2, 3, 4, 5};
        map(iArr, i -> i * i);
        for(int i: iArr)
            System.out.print(i + " ");
    }
}
```

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

```
1 4 9 16 25
```

- ◯ `public static <T> void map(T[] x, Operatable<T> ob)`
- √ `public static <T extends Number> void map(T[] x, Operatable<T> ob)`
- ◯ `public static void map(Integer[] x, Operatable<?> ob)`
- ◯ `public static <T extends Number> void map(Operatable<T> ob, T x[])`

Solution: Since `operate` method in `Operatable` applicable to any subtype of `Number`, the `for_each` function must define a generic parameter type that extends `Number`. As per the call made from `main` – `map(i_Arr, i -> i * i)`, the first argument of `for_each` must be a generic array, and second argument must be an object of `Operatable` type.

7. The `merge` method of `Map` has three arguments - key, value and reference to a function accepting two arguments - and merges the old value with the new value for a given key. Consider the code given below. [MSQ:2 points]

```
import java.util.*;
public class FClass{
    public static void main(String[] args){
        Map<String, Double> prices = new LinkedHashMap<String, Double>();
        prices.put("Pen", 30.0);
        prices.put("Pencil", 10.0);
        prices.put("Notebook", 40.0);
        prices.put("Paper", 5.0);

        for(Map.Entry<String, Double> e : prices.entrySet())
            prices.merge(_____________________________________);

        System.out.println(prices);
    }
}
```

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

```
{Pen=33.0, Pencil=11.0, Notebook=44.0, Paper=5.5}
```

- √ `e.getKey(), 0.1, (x, y) -> x + x * y`
- ◯ `e.getKey(), 0.1, (y, x) -> x + x * y`
- ◯ `e.getKey(), 0.1, (x, y) -> y + x * y`
- √ `e.getKey(), 0.1, (y, x) -> y + x * y`

Solution: The computation of new value (`new_value`) of for each (key, value) pair would be calculated as:
`new_value = (old_value, 0.1) -> old_value + old_value * 0.1`

8. Which of the following statements can find out the number of integers between 0 to 50 that are divisible by 3? [MCQ:2 points]

◯
```
long c = Stream.iterate(0, n -> n + 3)
              .limit(50).count();
```

√
```
long c = Stream.iterate(0, n -> n + 3)
              .takeWhile(n -> n <= 50).count();
```

√
```
long c = Stream.iterate(0, n -> n <= 50, n -> n + 1)
              .filter(n -> n % 3 == 0).count();
```

◯
```
long c = Stream.iterate(0, n -> n + 3)
              .dropWhile(n -> n < 50).count();
```

Solution: The question asked is to determine the number of elements that are in between 0 and 50 and divisible by 3.
Option-1 – generate total 50 elements from 0 which are divisible by 3.
Option-2 and -3 options are correct.
Option-4 – starts generating the numbers divisible by 3 from 50.

9. Consider the Java code given below that adds a list of students and their marks to a TreeMap.

[MCQ:2pts]

```
import java.util.stream.Stream;
import java.util.*;

public class TreeStream{
    public static void main(String []args){
        Map<Integer,String> student_map = new TreeMap<Integer,String>();

        student_map.put(80,"Arya");
        student_map.put(62,"Diya");
        student_map.put(71,"Fiona");
        student_map.put(79,"Mason");
        student_map.put(90,"Maria");

        Stream<Map.Entry<Integer,String>> scores
                            = student_map.entrySet().stream().limit(3);
        System.out.println(Arrays.toString(scores.toArray()));
    }
}
```

What will the output be?

- √ `[62=Diya, 71=Fiona, 79=Mason]`
- ◯ `[80=Arya, 62=Diya, 71=Fiona]`
- ◯ `[90=Maria, 80=Arya, 79=Mason]`
- ◯ `[71=Fiona, 79=Mason, 90=Maria]`

Solution: The list is sorted inside the TreeMap in the increasing order of the keys, which are the scores here. `limit(3)` extracts only the first three, which will be the list of students with the least scores.

10. From among the options, choose the code segment that gives the same output as is given by the Java code inside the `CODE BLOCK`.

[MSQ:2pts]

```
import java.util.stream.Stream;
import java.util.*;
import java.util.stream.Collectors;

public class FClass {
    public static void main(String[] args) {
      //CODE BLOCK begins here
      Stream<Integer> integers = Stream.iterate(0, i -> i < 50, i -> i+1);
      integers.map(i -> i % 7 == 0).forEach(System.out::println);
      //CODE BLOCK ends here
    }
}
```

◯
```
Stream<Integer> integers = Stream.iterate(0, i -> i < 50, i -> i+1);
integers = integers.filter(i -> i % 7 == 0);
integers.forEach(System.out::println);
```

√
```
Stream.iterate(0, i -> i < 50, i -> i+1)
              .map(i -> i % 7 == 0)
              .forEach(System.out::println);
```

◯
```
Stream<Integer> integers
              = Stream.iterate(0, i -> i < 50, i -> i+1)
              .map(i -> i % 7 == 0)
              .forEach(System.out::println);
```

√
```
Stream<Integer> integers = Stream.iterate(0, i -> i < 50, i -> i+1);
List<Boolean> newList = integers.map(i -> i % 7 == 0)
                                .collect(Collectors.toList());
for (int i = 0; i < newList.size();i++){
    System.out.println(newList.get(i));
}
```

Solution: The `CODE BLOCK` maps each number to whether it is divisible by 7 or not. It prints the boolean value based on the result for each element. In Option 1, it is filtering out the ones that are divisible by 7, and displays the actual values that are divisible by 7.
Option 2 also prints the boolean value based on whether each element is divisible by 7 or not.
Option 3 will give an error because forEach(System.out::println) returns void, and

hence it cannot be assigned to a Stream object. Option 4 assigns the boolean values to a List using method `collect(Collectors.toList())`, and prints the values by iterating through the list.