BSCCS2005: Programes discussed in the live session
Weeks {1-6}

1. Abstract classes. [Anand:Week 4]

```
A. Before collecting classes together

class Swiggy{
    public void swiggyOrder() {
        System.out.println("you ordered from swiggy");
    }
}
class Zomato{
    public void zomatoOrder() {
        System.out.println("you ordered from zoamto");
    }
}
class FoodPanda{
    public void foodPandaOrder() {
        System.out.println("you ordered from foodpanda");
    }
}
class Person{
    public void foodOrder(Swiggy swiggy) {
        swiggy.swiggyOrder();
    }
}
public class Abstract1 {
    public static void main(String[] args) {
        Person person=new Person();
        person.foodOrder(new Swiggy());
    }
}

B. After collecting classes together, Here we are forcing every class should
extend the class FoodOrder

class FoodOrder{
    public void order() {
        System.out.println("Here you can order food");
    }
}
class Swiggy extends FoodOrder{
    public void swiggyOrder() {
        System.out.println("you ordered from swiggy");
    }
}
class Zomato extends FoodOrder{
    public void zomatoOrder() {
        System.out.println("you ordered from zoamto");
    }
}
class FoodPanda extends FoodOrder{
    public void foodPandaOrder() {
        System.out.println("you ordered from foodpanda");
    }
}
class Person{
    public void foodOrder(FoodOrder order) {
        if(order instanceof FoodPanda)
            new FoodPanda().foodPandaOrder();
        if(order instanceof Swiggy)
            new Swiggy().swiggyOrder();
    }
}
public class Abstract2 {
    public static void main(String[] args) {
        Person person=new Person();
        person.foodOrder(new Swiggy());
    }
}
```

C. Now force every class to override order(), to do that change public void order(); to public abstract void order();

```
abstract class FoodOrder{
    public abstract void order();
}
class Swiggy extends FoodOrder{
    public void order() {
        System.out.println("you ordered from swiggy");
    }
}
class Zomato extends FoodOrder{
    public void order() {
        System.out.println("you ordered from zoamto");
    }
}
```

```
class FoodPanda extends FoodOrder{
    public void order() {
        System.out.println("you ordered from foodpanda");
    }
}
class Person{
    public void foodOrder(FoodOrder order) {
        order.order();
    }
}
public class Abstract3 {
    public static void main(String[] args) {
        Person person=new Person();
        person.foodOrder(new Swiggy());
    }
}
```

2. Call Backs [Anand:Week 4]

```
import java.util.Date;
interface Notification{
    void notification();
}
class WorkingDay implements Notification{
    public void notification() {
        System.out.println("Today is working day please stay on your work......");
    }
}
class Weekend implements Notification{
    public void notification() {
        System.out.println("Today is weekend enjoy the day........");
    }
}
class Timer{
    Notification notification;
    public void start(Date date) {
        if(date.getDay()==6 || date.getDay()==0){
            notification=new Weekend();
            notification.notification();
        }
        else{
            notification=new WorkingDay();
            notification.notification();
        }
    }
}
public class User {
    public static void main(String[] args) {
        Timer timer=new Timer();
        timer.start(new Date());
    }
}
```

3. Private classes. [Anand:Week 4]

A. Here BookMyshow and Payment both are defined seperately, BookMyshow depends on the Payment, Here Payment should only available to BookMyshow class because it has sensitive information

```
class BookMyshow{
    String userEmail;
    String movieName;
    double amount;
    public void bookTciket() {
        Payment payment=new Payment();
        if(payment.makePayment()) {
            System.out.println("Tickets booked");
        }
    }
}
class Payment{
    int cardno;
    int cvv;
    public boolean makePayment() {
        return true;
    }
}
public class UserTicket {
    public static void main(String[] args) {
        BookMyshow myshow=new BookMyshow();
        myshow.bookTciket();
    }
}
```

B. Now Payment is a private member of the BookMyshow class, now no other class can access this Payment class.

```
class BookMyshow{
    String userEmail;
    String movieName;
    double amount;
    public void bookTciket() {
        Payment payment=new Payment();
        if(payment.makePayment()) {
            System.out.println("Tickets booked");
        }
    }
    private class Payment{
        int cardno;
        int cvv;
        public boolean makePayment() {
            return true;
        }
    }
}
public class UserTicket {
    public static void main(String[] args) {
        BookMyshow myshow=new BookMyshow();
        myshow.bookTciket();
    }
}
```

4. Interfaces [Anand:Week 4]

```
A. By default in an Interface every variable is public static final
By default in an Interface every method is public abstract


interface A{
    int a=0;
    void show();
}
interface B{
    public static final int x=20;
    public abstract void display();
}
public class InterfaceDemo1 {
    public static void main(String[] args) {
    }
}
```

B. A class can inplement any number of interfaces.
But implemented class should override all abstract methods from the interfaces.

```
interface A{
    int a=20;
    void show();
    }
interface B{
    public static final int x=20;
    public abstract void display();
}
class Impl implements A,B{
    public void display() {
    }
    public void show() {
    }
}
public class InterfaceDemo1 {
    public static void main(String[] args) {
    }
}
```

C. From Java 1.8 version onwards you can also write default and static methods in an Interfaces.
If interfaces has same default/static method then its implemented class should rpovide fresh implementation in it.

```
interface A{
    default void display() {
        System.out.println("This is A  display");
    }
}
interface B{
    default void display() {
        System.out.println("This is B  display");
    }
}
class Impl implements A,B{
    public void display() {
        System.out.println("This is Impl class display");
        A.super.display();
        B.super.display();
    }
}
public class InterfaceDemo1 {
    public static void main(String[] args) {
        Impl obj=new Impl();
        obj.display();
    }
}
```

5. **Week 4: Nested Class , Callbacks & Classical Producer Consumer Problem.**

Points to note:

1. All orders can go through only the Bakery ie cupcakebakery can not participate in an order on its own
2. The bakery can not directly interfere into the work of cupcakebakery on its own because *outer class does not have access to inner class methods or variables and hence need a dedicated inner class object to call the inner class methods every time.*
3. All the objects of CupcakeBakery that we create in this question are actually part of the one single Bakery object which was defined in '`main`' method i.e. `Bakery b`.
   One outer class object can have multiple inner class objects mapped within it. But an inner class object is always associated with a particular outer class instance.
4. Bakery can only have the count of orders it gave to each cupcakebakery but not the details/specifics of each order for each cupcakebakery. Also cupcakebakery can not bypass the parent bakery to bag an order on its own.
5. This is a HAS-A relationship with Composition because CucakeBakery cannot exist without a Bakery. Composition and Aggregation types of HAS-A relation is discussed the corresponding live session.

Points to ponder:
Now you all have learnt about Lambdas, so here is a thinking exercise for you all:

In this code we were passing the object so as to make a bijective mapping between the producer and consumer objects i.e. if a Producer tells a consumer to consume then after the consumer finishes it must callback that specific caller producer not some random producer.

1. Using lambdas you can ignore this notion of passing objects to make a callback. Can you ponder how ?
2. We have yet not covered threads, but once we do that, come back to this question and think what happens if we have multiple producer and consumers in a concurrent environment. It should work in a way such that if a particular consumer is served by a particular producer then no other producer thread should serve that specific consumer, same should be true the other way round also. Can we manipulate/restructure (but maintaining the object oriented constraints discussed under Points to note section) this code to implement something like that? If so, how?

```
import java.util.Scanner;
interface Bakeable{
    default void orderCupcakes() {
        System.out.println("Cupcakes are not available");
    }
}
class Consumer{
    private Bakery order;
    public Consumer(Bakery curr_order){
        order = curr_order;
        // initializing the identity of the producer/bakery
        // object which is associated with this specific consumer
    }
    public void eatCupcakes() {
        for(int i=0;i<order.cakes.length;i++)
            order.cakes[i] = "Eaten up!!";
        for(int i=0;i<order.cakes.length;i++)
            System.out.print(order.cakes[i]+" ");
        System.out.println();
        //Finished eating all cakes
        order.Order();
    }
}
class Bakery{
    Scanner s = new Scanner(System.in);
    private Consumer c;
    public String[] cakes;

    private class CupcakeBakery implements Bakeable{
        private int order_quantity;

        public void orderCupcakes() {
            //Taking input the number of cupcakes for the new order;
            order_quantity = s.nextInt();
            cakes = new String[order_quantity];

            //Preparing the cakes
            for(int  i=0;i<order_quantity;i++) {
                cakes[i] = "Cake" + (i+1);
            }
            for(int i=0;i<order_quantity;i++)
                System.out.print(cakes[i]+" ");
            System.out.println();
            //Dispatching the order to a specific consumer
            c = new Consumer(Bakery.this);
            c.eatCupcakes();
        }
    }
    public void Order() {
        //checking whether any new order is available...
        //console input should be Y
        char order = s.next().charAt(0);

        if(order == 'Y') {
        (new CupcakeBakery()).orderCupcakes();
        }
    }
}

public class BakeryMain {
    public static void main(String args[]) {
        Bakery b = new Bakery();
        b.Order();
    }
}
```

Output

```
Y
6
Cake1 Cake2 Cake3 Cake4 Cake5 Cake6
Eaten up!! Eaten up!! Eaten up!! Eaten up!! Eaten up!! Eaten up!!
Y
3
Cake1 Cake2 Cake3
Eaten up!! Eaten up!! Eaten up!!
Y
2
Cake1 Cake2
Eaten up!! Eaten up!!
N
```

6. **Week 6: Generics, Wildcard & recalling Stacks and Queues**

   Points to note:

   1. Both the stack and the queue can store any type of data as long as they are numeric type i.e. subtypes of `Number` class. So our first requirement is to make the data structures store any kind of numeric value, this we can achieve using *generics*: `<T extends Number>`.
   2. Now, we want to compare two data structures of different types - stack and queue on the basis of some common property (here, sum of stored items). In such scenarios wildcards can be helpful. This has been discussed in the livesession too,
      Whenever you want to compare between two entities (class objects) of different types which may again store generic type of data you may think of using wildcards. Observe the `pop()` method inside `isEqualSum` method is working on both Integer and Double type data stored in different data structures which we are comparing.

   Points to ponder:

   1. In this example we were trying to check the property of equality of sum of elements in a stack and a queue. Both of them are related by a common parent (interface `DataStructure`). But everytime we want to use dynamic polymorphism (like `pop`, `getSize` methods in this case) we have to have a parent-child relationship (*IS-A*) but what happens if two entities/classes which we want to compare on some specific property are totally unrelated but that property is meaningful for both?
      Example: Say you want to check eligibility of a person to vote and eligibility of zone to be declared as containment zone. Both are checking eligibility but the eligibility check would be totally different. You cannot just make a superclass for both these classes (Person, Zone) and use dynamic dispatch by overriding `eligibility` method, reason being every inheritance must satisfy IS-A relation. So there should be some logical similarity between the two child classes.

      Here is where you can take help of lambdas/functional interfaces. Think, how?
      This is also discussed in live session.

```
import java.util.*;
interface DataStructure<T extends Number>{
    int getSize();
    void display();
    T pop();
}

class Stack <S extends Number> implements DataStructure{
    S[] arr_stack;
    int size;
    int top;
    Stack(int capacity, S[] arr){
        size = capacity;
        top = -1;
        arr_stack = arr;
    }
    public void push(S element) {
        if(top < (size-1)) {
            top = top + 1;
            arr_stack[top] = element;
        }
        else
            System.out.println("Overflow");
    }
    public void display() {
        int iterator = top;
        while( iterator >= 0) {
            System.out.print(arr_stack[iterator]+" ");
            iterator--;
        }
        System.out.println();
    }
    public int getSize() {
        return size;
    }
    public S pop() {
        int temp;
        if(top > -1) {
            temp = top;
            top--;
            return arr_stack[temp];
        }
        return null;
    }
```

```
}
class Queue <S extends Number> implements DataStructure{
    S[] arr_queue;
    int size;
    int front;
    int rear;
    Queue(int capacity,S[] arr){
        size = capacity;
        arr_queue = arr;
        front = -1;
        rear = -1;
    }
    public int getSize() {
        return size;
    }
    public void push(S element) {
        if(front == -1 && rear == -1) {
            front++;
            rear++;
            arr_queue[rear] = element;
        }
        else if(rear < (size-1)) {
            rear = rear + 1;
            arr_queue[rear] = element;
        }
        else
            System.out.println("Overflow");
    }
    public void display() {
        int iterator = front;
        while( iterator < size) {
            System.out.print(arr_queue[iterator]+" ");
            iterator++;
        }
        System.out.println();
    }
    public S pop() {
        int temp;
        if(front < size) {
            temp = front;
            front++;
            return arr_queue[temp];
        }
        return null;
    }

}

class DataStructureSum<T extends DataStructure>{
    T ds;
    DataStructureSum(T obj){
        ds = obj;
    }
    // We specifically use wildcard when we have
    //to compare two dissimilar types which are
    public void isEqualSum(DataStructureSum<?> obj2) {
        Double sum1=0.0;
        Double sum2=0.0;

        for(int i = 0;i< this.ds.getSize();i++) {
            sum1 += (this.ds.pop()).doubleValue();
        }
        for(int i = 0;i< obj2.ds.getSize();i++) {
            sum2 += (obj2.ds.pop()).doubleValue();
        }
        if(sum1.equals(sum2)) {
            System.out.println("Equal");
        }
        else {
            System.out.println("Not Equal");
        }
    }
}
public class GenericDataStructure{
    public static void main(String args[]) {
        Scanner reader = new Scanner(System.in);
        // Input for the size of Integer array
        int size1 = reader.nextInt();
        // Input for the size of Double array
        int size2 = reader.nextInt();

        Integer[] arr1 = new Integer[size1];
        Double[] arr2 = new Double[size2];
        Stack<Integer> s = new Stack<Integer>(size1,arr1);
        Queue<Double> q = new Queue<Double>(size2,arr2);

        //Inserting elements in the data structures
        for(int i=1;i<=size1;i++) {
            s.push(i*10);
        }
        for(int i=1;i<=size2;i++) {
            q.push(i*10.0);
        }
        //Printing the elements in proper order:
        //The last element inserted into stack
        // should be printed first and first
        // element should be printed at last

        s.display();

        //The first element inserted into queue
        //should be printed first and last element
        // should be printed at last

        q.display();

        DataStructureSum<Stack> set1 = new DataStructureSum<Stack>(s);
        DataStructureSum<Queue> set2 = new DataStructureSum<Queue>(q);


        // This method should print "Equal" if the
        // arithmetic sum of all elements in the two
        // DataStructures is same otherwise it should
        // print "Not Equal"
        set1.isEqualSum(set2);

    }
}
```

Input 1:

```
5
5
```

Output:

```
50 40 30 20 10
10.0 20.0 30.0 40.0 50.0
Equal
```

Input 2:

```
7
4
```

Output:

```
70 60 50 40 30 20 10
10.0 20.0 30.0 40.0
Not Equal
```

7. **Week 8: Lambda & Functional interface**

   Points to note:

   1. Lambdas are a way to represent a function in the form of a reference variable of a functional interface, which can be passed as parameters just like objects.
   2. This coded example tries to bring out the basic syntax of lambdas and also tries to communicate the idea that when totally different entities have to be operated upon on the basis of some common feature (eligibility in this case) we may use lambdas. *Discussed in detail in the live session.*
   3. Another advantage that lambda provides us is implementing abstraction of complex code. In this example we can see that the method `do_for_each` takes in an arraylist and a lambda which provides the logic for testing eligibility of the elements stored in the arraylist. Now assuming the definition of `do_for_each` is pre-written, only thing we have to do is define the logic for eligibility check and pass it as lambda. A coder does not have to worry everytime about how the method `do_for_each` is traversing the list and calling the lambda on each object.
      This is precisely what you have been doing while overriding `compareTo` method with just the logic to sort the TreeSet elements and some internal method which is pre-written is doing the dirty work of calling compareTo on each pair of object inside the tree set and then re-positioning them appropriately so as to make it sorted.

      This is also how the `Collections.sort(), forEach()` etc. methods work.

```
import java.util.*;

interface eligible<T>{
    boolean checkEligibility(T obj);
}
class Voter{
    int age;
    Voter(int x){
        age =x;
    }
}
class ContainmentZone{
    String colour;
    ContainmentZone(String s){
        colour = s;
    }
}

class MutualFund{
    double returns;
    MutualFund(double r ){
        returns = r;
    }
}

class Account{
    int balance;
    Account(int bal){
        balance = bal;
    }
}

public class sample2 {
    public static void do_for_each(eligible fn, ArrayList l) {
        int i=0;
        while(!l.isEmpty()) {
            // do some complex work using this same named method
            if(!fn.checkEligibility(l.get(i))) {
                l.remove(i);
            }
            i++;
        }
        for(int j=0;j<l.size();j++) {
            System.out.println(l.get(j));
        }
    }

    public static void main(String args[]) {
        Voter v1 = new Voter(10);
        ContainmentZone c1 = new ContainmentZone("red");
        MutualFund m1 = new MutualFund(5.6);

        eligible<Voter> l1 =  x -> {
            if(x.age >= 18)
                return true;
            else
                return false;
            };

        eligible<ContainmentZone> l2 = x -> {
            if(x.colour.equals("red"))
                return true;
            else
                return false;
            };

        // We used lambdas to check a common feature
        // ie. eligibility check for totally unrelated
        // classes with separate implementation
        System.out.println(l1.checkEligibility(v1));
        System.out.println(l2.checkEligibility(c1));

        // Comparable interface is inside java.util
        // and hence it is imported implicitly
        // we do not have to explicitly define Comparable
        // interface to use compareTo

        // Recall that this is the same method which TreeSet
        // Collection was using to sort its elements, but there
        // we were implementing the interface and overriding compareTo

        Comparable<Account> com1 =  x -> {
            if(x.balance > 5000)
                return 1;
            else
                return -1;
        };

        Account a1 = new Account(200000);
        System.out.println(com1.compareTo(a1));

        ArrayList<Voter> voterlist = new ArrayList<>();
        ArrayList<ContainmentZone> zonelist = new ArrayList<>();
        do_for_each(l1,voterlist);
        do_for_each(l2,zonelist);
    }
}
```