(c) Code Link: t.ly/vooZs → (or follow link in 7(a))

(d) → github.com/Ibtida01/TF_Codes_1_2/tree/main

8(a), (b), (c), (d) → [Same]

2012-13

```
import java.util.ArrayList;
1(a) class Animal{
    private String name;
    private int age;
    public String getName() { return name;}
    public int getAge() {return age;}
    Animal(String name, age){
        this.name=name;
        this.age=age;
    }
}
class AnimalList{
    ArrayList<Animal> al=new ArrayList<>();
    void addAnimal(Animal animal)
    {   al.add(~~ani~~ animal);
    }
    void removeAnimal(~~Animal animal~~ String name)
    {   for(Animal an: al){ if(an.getName().equals(~~animal.getName~~ name)
            al.remove(~~animal~~ an);
            return;}
        System.out.println("Not found");}
```

```
public void setName
        (String name)
{
  this->name=name;
}
public void
public class Q1a{
public static void main(String [] args){
    Animal a1=new Animal("Lion", 12);
    Animal a2=new Animal("Tiger", 5);
    Animal a3 =new Animal ( "Rabbit", 3);
    AnimalList alist=new AnimalList();
    alist.addAnimal(a1);
    alist.addAnimal(a2);
    alist.addAnimal(a3);
   for(Animal an:alist.al)
    {
     System.out.println(an.getName());
    }
    alist.removeAnimal("Lion");
 }
}
```

1ⓑ → Out of Syllabus

```
1ⓒ class Movie{
private String name;
private ~~String[]~~ ArrayList<String> directors =new
                                    ArrayList<>();
   Movie(String name, String... directors)
   {
    this.name=name;
    ~~for~~
for(String director:directors)
 {
   this.directors.add(director);
 }
}
public ~~gf~~ String getName() {
   return name;
 }
public ArrayList<String> getDirectors(){
      return directors;
  }
}
class MovieClub{
  ArrayList <Movie> movies =new ArrayList<>();
  int movieCount;
  MovieClub(~~mo~~ int movieCount, Movie... movies)
  { this.movieCount=movieCount;
   for(Movie m: movies)
   { ~~m~~ this.movies.add(m);
   }
 }
```

```
void addMovie(Movie m)
{ for(Movie mv: movies)
  { if(mv.equals(m))
    { System.out.println("Already exists");
       return;
    }
  }
  movies.add(m);
}
}
```

2(a) Buffered Reader maintains a buffer ~~fr~~ which makes it efficient.

Details (As far as I remember): When a file is to be read, JVM (or Java Compiler/ RE can't remember) makes a request to the OS to read a portion of the file; if the reading is done Byte by Byte, only a single Byte is read, but the rest is discarded, so a lot of system loss is faced. To avoid this, Buffered Reader maintains a buffer where ~~a~~ the rest portion of the chunk is cached and the single intended Byte is read. Until the whole buffer isn't fully read, ~~or~~ JVM (or Java Compiler or JRE ~~mo~~ doesn't make any request to (can't specify) the OS, resulting in efficient operations, lesser requirement of time & resources, better performance, lesser ~~&~~ interruption in the natural workflow of the OS.

```
~~#~~necessary imports
public class PFSolve {
~~2(b)~~ public static void main(String [] args)
{ { ArrayList<String> textLines = new ArrayList
                                      <>();
  try {
  BufferedReader br=new BufferedReader(new
                          FileReader("a.txt"));
  String Line="";
  while((line=br.readLine())!=null)
  {
    textLines.add(line);
  }
  ~~f~~ br.close();
  }
  catch(Exception e) {
    * System.out.println(e);
  }
try {
BufferedWriter bw=new BufferedWriter(new FileWriter
                                      ("a.txt"));
for (String l : textLines)
 {
    bw.write(~~t~~l);
    bw.newLine();
  }
 bw.append("The Line to be appended");
 bw.newLine();
 } catch (Exception e) {
   System.out.println(e);
   }
}
}
```

2(b) → Slide

2(c) 2 uses:

(i) Passing necessary parameters to the super-class's constructor in the subclass's constructor.

(ii) To access hidden members of superclass

~~2(d) public class TfSolve{~~

~~public void~~

```
Q(d) class CustomException extends Exception{
    String message = "Value is greater than 100";
    CustomException(~~String~~){
        //Nothing
    }
  @Override
  public String toString()
    {
       return message;
    }
}
 void f(int value) ~~throws~~ CustomException
 {try{
    if(value > 100)
    {
       throw new CustomException();
    }
 }catch(CustomException e)
   {
     System.out.pointln(e);
   }
 finally{
   throw new CustomException();
 }
}
```

```
public class TFSolve{
  psvm(String [] args)
  {
    int v;
    v= 9;
    try{
      f(v);
      System.out.println("Reached here");
      ~~ff~~ v= 400;
      f(v);
      System.out.println("Reached after v=400");
    } catch (Exception e)
    {
      System.out.println(e);
    }
  }
}
```

3ⓐ Interface itfc{

```
    void f();
  }
  class C1 implements itfc{
    void f(){
      // Body
    }
  }
  public class TF{
```

```
psvm(string[] args){
  Itfe obj=new c1();
  itfe.f1();
 }
}
```

3(b) Properties of static variables:

(i) Belong to the class and not to a specific object.

(ii) Shared by all the objects

(iii) Only one copy of the static variable is created, ~~no~~ no matter how many objects are created.

(iv) ~~A~~ Since they are shared, changing their values takes effect for all objects.

(v) Can be accessed without even creating an object.

Properties of static methods:

(i) [Same]

(ii) [(v)]

(iii) Can only access static variables, & NOT the non-static ones.

(iv) Can ~~st~~ only call static methods, & NOT the non-static ones.

Reasons of main() method being static:

(i) Convention

(ii) ∵ main is the entry point, if it wasn't static we would need a object of the public class ~~hat~~

holding it, but for that we need to start the program, i.e. call main, which again ~~needs~~ would needa an object (if it was not static), the situation would ~~be~~come to a race condition. So the convention is to make the main static

3(c)
```
class Student{
  String id;
  String name;
  String course;
public int unitCmCount;
~~Public~~ Student(String id, String name, String course,
                                     int unitCmCount)
   {
     this.id = id;
     this.name=name;
     this.course=course;
  this.unitCmCount= unitCmCount;

   }
public void inc() {
       ++unitCmCount;
  }
}
 class AnotherClass //I have no idea of the Question
{ //so this kind of naming
}
```

```
Student student;
AnotherClass(Student student, int DesiredCount)
{
  this->student=student;
  ~~DesiredCount =~~ int currentState=this.~~##~~student.
                                        unitCmCount;
  int diff = DesiredCount-currentState;
  for(int i=1; i<=diff; i++)
  {  this.student.inc();

  }

}
}
```

4ⓐ(i) Abstract Class	Interface
(i) Has (Might have) ~~predefined~~ defined methods as well as abstract methods	(i) ONLY has abstract methods, ~~a~~ but after Java 7, from Java 8, default methods were introduced.
(ii) Abstract class might be fully defined (all abstract methods got body) by extender subclass or partially too.	(ii) A class implementing an interface must define all of its abstract methods, otherwise ~~they~~ the Class will become abstract.

(iii) Abstract class might have final, non-final, static, non static variables	(iii) Interfaces on'y have static & final variables.
(iv) Abstract class (or any class in Java) does n't Support multiple Inheritance	(iv) Interfaces support the concept of multiple inheritance
(v) Abstract class can Implement an interface (completely) but still have abstract methods of its own & stay abstract	(v) An interface can't Implement a class.

4 (a) (ii)

4(b)

```
public int CountKeysOcc(String[] SearchList, String
                                                SearchKey)
{
 int Count = 0;
 for(String s : SearchList)
 {
   if(s.equals(SearchKey))
        ++Count;
 }
 return Count;
}.
```

4(c)

```
Integer[][] a = new Integer[4][];
  a[0] = new Integer[3];
  a[1] = new Integer[1];
  a[2] = new Integer[10];
  a[3] = new Integer[5];
```

4(d) OutOfSyllabusException in syllabus. CSE 107. Java

B | 5(a) Kindly read slide please

problem: the ~~overloaded~~ overloading of << ~~function~~ operator must be done in a friend function of the concerning class.

```
class circle{
 ~~in~~ double r;
   ~~do~~ int x, y;
 circle(double r, int x, int y){
   this->r = r;
   this->x = x;
   this->y = y;
 }
 friend istream& operator>>(istream& in, circle
                                          c);
 friend ostream& operator<<(ostream& os, circle
                                          c);
};
istream& operator>>(istream& in, circle c){
   in>> c.r >> c.x >> c.y;
   return in;
}
ostream& operator<<(ostream& out, circle c){
```

5(b) Output :

```
Constructing 1
Received 1
Destructing 1
```

5(c)

```
class A{
   int x;
   ~~void f~~ public:
    void f_A(){
    }
 };
 class B{
   int y, z;
 Public
   double height;
   void f2(){
     //Body
   }
};
```

```
class C:public A, public B
{ int t;
  public:
  void hello(){
    //Body
  }
};
```

Generic Class:

6(a) ~~generic~~ Generic class means the classes that can have generic types of member variables and have generic methods that can work on variety of types having single implementation, the class becomes type specific once declared.

```
template <class t>
 class ARR{
  t arr[100];
  int sz;
 public:
   void bubbleSort(){
    for (int i=0; i<sz; i++)
    { for(int j=0; j<sz-1; j++)
      {  if (arr[j] > arr[j+1])
            swap(arr[j], arr[j+1]);
      }

    }
 }
   void compact(int* ~~zero th~~ ~~frost~~ array, int n, int start, int End)
  { int* temp = new int[End - start + 1];
    for(int i = start; i<= End; i++)
       temp[i-start] = array[i];
    int i;
    for(~~int~~ i=End; i<n; i++) //n <= 100 assumed
    {
       array[i-(End-start)] = array[i];

    }int ct=0;
    for(int j = i; j<=(i+End-start); j++)
       array[j] = temp[ct++]; }
  }
```

6(b) Fnc overloading is the process of writing newer function with the same name & return type but with different number or types of arguments or both or having rearrangement of different arguments of DIFFERENT types, an overloaded function's body might or might not be different than the original fnc.

(Dummy Overloading)

Overloaded fnc → might have different def^n than the original one (or might not), radically different behaviour might appear.

Generic fnc → Def^n is not going to be change, the fundamental operations stay the same.

6(c) By making that fnc as static, we can call it ~~them~~ by its ~~keep~~ ~~their~~ class names, eg.

```
class hello{
  int x=5;
  public:
   static int show(){
     cout<<"hello"<<endl;
     return x;
   }
};
```

```
int main(){
 hello.show();
};
```

7ⓐ Copy constructor is a special type of constructor that copies all the attributes of an obj. to another obj. and instantiates that new object.

3 cases:

(i) When an obj. of a class is declared with the reference of another preexisting object.

```
className obj2;
className obj1=obj2;//copy constructor is called
```

(ii) When an obj. is passed to a fnc. as an argument

```
fnc(obj1);//copy constructor is called
```

(iii) When the called fnc returns and inside the caller function, receiver object gets the obj (returned)

```
obj3=fnc(obj1); // fnc signature: ClassName fnc(ClassName obj1);
```

7ⓑ
```
long long Compute_Volume(int h,int w, int l)
{
    return h*w*l;

}
long long New_Compute_Volume(int h, int w=1, int l=1)
{ return h*w*l;

}
```

7ⓒ
```
template<class className>
class NoOfInstancesTracker{
    ~~pr~~ int count=0;
public:
   NoOfInstancesTracker()
   {
      ++count;
   }
   ~NoOfIntancesTracker(){
      --count;
   }
   NoOfIntancesTracker(const ClassName& obj){
      ++count;}
      int getCount(){ return count;}
};
//for any class X,
class X: private NoOfInstancesTracker< X >
{
   //Body
};
//and so on...

8(b) (a) void print(array &ob) ——→ MAJOR Problem
   { array x = obj;
     int i;
     for(~~int~~ i = 0; i <10; i++) cout<<x.get(i);
                          ↓
                         sz হবে
   }
```

when the fnc. will exit & return to its caller, the array object x will be destroyed. But x has the address i.e. reference of object ob. So after the fnc. return, the num object will not be existent. &num will be invalid memory location, so will cause segmentation fault & double freeing up.

Fix: writing a copy constructor to stop address Copying

```
array (array & ob){
    int ~~new~~ sz = ob.size;
this->size = sz;
 try {
      p = new int [size];
  } catch(bad_alloc xa) {
     cout<<"Allocation failure\n";
     exit (EXIT_FAILURE);
   }
for(int i=0; i<size; i++)
   this->p[i] = ob.p[i];
}
```

8(b) Slide + YES: github.com/Ibtida01/~~TF_1_2_C~~ TF_Codes_1_2/tree/main

8(c) Kindly read the answer from slides, thanks

8(d) class derived : ~~private~~ base{ → MUST be public