# ICE-10

1. 1) class B extends A { // constructor

B(Arraylist a)

{

   Super(a);

}

}

1. a. addData(T data) , deleteData(T data), public queryStats(T parameters),

b. Class B does not have control over Types

3) Tightly coupled with parent, decreased cohesion ,overhead of parent has to be carried.

1) Private A mine;

D(Arraylist arraylist)

{

 mine =new A();

 mine.setData(arraylist);

}

2)a.  addData(T data) , deleteData(T data), public queryStats(T parameters),

   b. Class D has control over what types to read from user. However using the same type as A will reduce complexity and increase maintainability of software for the future.

addData(String s), deleteData(String s), query(String s)

1. more stable ,less prone to bugs, easily maintainable, less coupling

Class A {

Collection<String> data;

Public A(Arraylist<String> a) {

this. Data=a;

}

protected String public generateStats();

}

## IIa. a) Early Binding.

b)Yes, As its not a abstract class and has public contructor.

c) virtual keyword to the methods.

d)GChild(int x): Child(x) { }

GChild(int x, float y): Child(x,y)

e) int main() {

int age=5;

float data = 5f;

Parent \*parentPtr;

Child childPtr(age);

Child childPtr1(age,data);

Child2 child2Ptr();

Child2 child2Ptr1(age);

GChild gchildPtr();

GChild gchildPtr1(age);

GChild gchildPtr2(age,data);

parentPtr = &childPtr;

parentPtr->myFunction();

Parent \*collection[8];

collection[0] = &childPtr;

collection[1] = &childPtr1;

collection[2] = &child2Ptr;

collection[3] = &child2Ptr1;

collection[4] = &child2Ptr1;

collection[5] = &gchildPtr;

collection[6] = &gchildPtr1;

collection[7] = &gchildPtr2;

for(int i =0;i<9;i++) {

collection[i]->myFunction();

}

Parent p1 = new Child();

Parent p4 = new Child2();

Parent p5 = new Child2(age);

Parent p6 = new GChild();

Parent p7 = new GChild(age);

Parent p8 = new GChild(age,data);

Child c1 = new GChild();

Child c2 = new GChild(age);

Child c3 = new GChild(age,data);

}

IIb. a) abstract class Parent {

int old;

public Parent() {

old = 0;

}

public Parent(int old) {

this.old = old;

}

void myFunction();

}

class Child extends Parent {

float data;

public Child() {

super();

data = 0;

}

public Child(int old) {

super(old);

data = 0;

}

public Child(int old, float data) {

super(old);

this.data = data;

}

void myFunction() {

System.out.println("Child myFunction\n");

}

}

class Child2 extends Parent {

public Child() {

super();

}

public Child(int old) {

super(old);

}

void myFunction() {

System.out.println("Child2 myFunction\n");

}

}

class GChild extends Child {

public GChild() {

super();

}

public GChild(int age) {

super(age);

}

public GChild(int age, float data) {

super(age,data);

}

void myFunction() {

System.out.println("GChild myFunction\n");

}

}

b)By default java supports runtime polymorphism

c)subclass implementation can change anytime and it creates dependencies. dynamic binding can create software dependencies and maintainability issues

d)functions cannot vary at runtime ,no runtime flexibility,

III) interface Shape {

double volume();

double surfaceArea();

}

abstract class Polygon {

protected int sides;

}

class Sphere implements Shape {

protected double radius;

/\* The constant PI \*/

private static final double PI;

Sphere(double radius) {

}

/\* Calculate and return values\*/

double volume();

double surfaceArea();

}

class Cube extends Polygon {

Cube(int sides) {

}

/\* Calculate and returns \*/

double volume(){ }

/\* Calculate and returns area\*/

double surfaceArea(){ }

}

class Cylinder implements Shape {

private double radius;

private double height;

/\* The constant PI \*/

private static final double PI;

Cylinder(double radius,double height) {

}

/\* Calculate and return values\*/

double volume();

double surfaceArea();

}

class Cone implements Shape {

private double radius;

private double height;

private double length;

/\* The constant PI \*/

protected static final double PI;

Cone(double radius,double height,double length) {

}

/\* Calculate and return values\*/

double volume();

double surfaceArea();

}

IV)

1. void changeMe(string &s) {

s = "change me"; }

void swapThemByRef(int& x, int& y) {

int temp = x;

x = y;

y = temp;

}

1. a:10 ,b:5,s:change me

3)

A *deep copy* copies all fields, *and* makes copies of dynamically allocated memory pointed to by the fields. To make a deep copy, you must write a copy constructor and overload the assignment operator.

V)**Composition**

Lifetime: Lifetime association

Cardinality: variable

Association: Stable

Ownership: Transferable

**Containment**

Lifetime: no

Cardinality: variable

Association: Temporary

Ownership: No

**Inheritance**

Lifetime: 1 to 1,variable by design for composion

Cardinality: fixed 1:1

Association: permanent

Ownership: Impied child owns parents obj and may not be released

**VI) a) IS –a OCP – parent is open for extension through subclass**

**Least Knowledge – child has implied information about parent**

**HAS-a OCP – No ocp,Closed for modification and closed for extension**

**Least knowledge - both class doesn’t assume about the other**

**Holds-a OCP – No ocp,Closed for modification and closed for extension**

**Least knowledge - both class doesn’t assume about the other**

**b)OCP : must be aware of parent class implementation and design**

**c) If its open for extension then the class should be aware of some parameters this conflicts with the concept of least knowledge**

**VII)**

**1)** They normally return true or false, depending on the result of the test that their name suggests. Programmer expects bool value true of false based on the comparison of the object.

**2)when we want to specify certain parameters or fields are necessary for comparison. Programmers expects the object to be equal.**

**3)Equals method violate the rule of**

**“Postconditions cannot be weakened in a subtype.”**