ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.write(

2016-2017 Raihan

Apologies for the trash hand writing

1

a) 'public static void' The following is the signature of the Java main method. 'public static void main(String[] args) {...}'

public is used to make it accessible by all. JVM as an outside entity tries to access the main so it should be public

static is used so that the method is available without creating any object as JVM wants to access main without creating an object.

void means main doesn't return anything.

String args[] represents the array of String Objects containing the Command Line Arguments

1c

Threads can be created in two ways:

One way is extending Threads and another way is by implementing the runnable interface.

Extending Threads:

```
class MyTread extends Thread {
    MyThread (String name) {
        super(name);
        start();
    }
    @Override
    public void run () {
        // Thread logic
    }
}
```

Implementing Runnable:

```
class My Thread implements
  Runnable {
  Thread t;
  My Thread () {
    t = new Thread (this)
    t. start();
  }
  public void run () {
    // thread logic
  }
}
```

Which is better

Implementing Runnable is better and here are my logic,

~~1) By extending the Thread class we are restricting our class to one thread~~

1) Implementing Runnable Interface allows your class to be used by multiple ~~thread~~ threads., while extending thread restricts my class to a single thread only

2) Runnable is more flexible because it allows me to extend more class or implement more interface as needed, however, if I already extend a the threads class I cannot extend any more class, thus it is less flexible.

3) Extending thread is a bit heavier interms of resource usage, Implementing Runnable is comparatively more efficient in terms of resource usage

4) Implementing Runnable promotes better code organisation and flexibility. Thus it is overally better than Extending Threads.

2.

a) The two uses of super keyword in Java are:

i) Calling a Super Class Constructor

ii) accessing superclass members which are hidden or overridden.

An abstract class in Java is a class that ~~cannot~~ cannot be instantiated directly. Contains an abstract method ~~abstract method~~ abstract method f(); No instance can be created of the abstract class The subclass must implement the abstract method. otherwise it will be an abstract class too.

Abstract class are defined using the keyword 'abstract'. Abstract class can have concrete ~~not~~ / final methods or not

Example:

```
abstract class AbstractClass {
    abstract void f1();
    void f2() {
        // logic
    }
    final void f3() {
    } // end
}
```

```
// Restrictions on the derived class
class niceClass extends AbstractClass {
// It must implement f1();
// or else it will be an abstract class
void f1() {
    // code
}
}
```

2//

b) In the given code multiple inheritance issue arises due to ambiguity. Here, C is inheriting a default method named as f() which is defined in both the interface A and B. Thus, ~~C becomes confused~~ compiler becomes confused as which should C actually inherit. Thus, C must explicitly implement the ambiguous method in order to remove ambiguity.

or the ~~default~~ default void f() can be removed from either of the two interfaces. or B can ~~inherit~~ extend interface A and C can only implement B.

Possible fixes for the given scenario ~~way 1:~~

method 1

```
class C implement A, B {
    public void f() {
        // code
    }
}
```

method (2)

```
interface A {
    default void f() {
        System.out.println("A's f");
    }
}
interface B extends A {
    default void f() {
        System.out.println("B's f
    }
}
```

2// c)

```
class c2 extends c1 implements i2 {
    void public void f2() { },
        public void f3() { }
        public void f4() { }
        void f5() { };
}
```

2// d) Autoboxing: the process by which a primitive type is automatically encapsulated into its equivalent type wrapper whenever an object of that type is needed.

Auto-Unboxing: the process by which the value of a boxed object is automatically extracted from a type wrapper whenever its value is needed.

In Performance-critical code we should not use Autoboxing and unboxing, ~~During~~, as those require more ~~necessary~~ memory.

3/d) T extends class X, implements Y and Z

T implements class X, Y, Z.

3/ d) Here, T refers to a bounded class type, which either implement at least two of X, Y or Z or it inherits from at most one of X, or Y or Z.

In Between X, Y and Z at least two must be an interface and at most one can be a class type

Thus X can be a class or an interface

4// b)

ArrayList	Vector
i) not synchronized	i) Synchronized
ii) increases 50% in size if items exceeds capacity	ii) increases 100% in size if items exceeds capacity

HashTable	HashMap
i) synchronized	i) Not synchronized
ii) doesn't store null in key or value	ii) Can store null in one key and many values
iii) comparatively slower	iii) comparatively faster.

During equality testing we must avoid autoboxing. because we aren't concerned with the equality of any ~~integer~~ reference rather we are concerned with the value.

We should also not use Autoboxing if there is a collections which utilizes ~~primitive~~ or stores datatypes such as int[], double[].

Sometimes autoboxing or unboxing may also lead to null pointer exception.

Auto boxing

```
Integer a = 100;
```

Auto-unboxing

```
int b = a;
// a was an Integer obj
```

Hash Table	HashMap
i) Synchronized	i) Not-Synchronized
ii) Thread-safe	ii) Not threadsafe
iii) Slower comparatively	iii) Faster comparatively
~~iv~~	

// The answers to the rest of the part of question 4 is given through code

Section-B

5// a) Constructors and destructor are special member function of a class. Constructors are automatically called when an object is created. Sometimes it is necessary for fixing certain values or proving certain constraints while creating an object and a constructor provides an elegant way to do so. We can also initialize certain value without even passing the parameter using a default constructor or constructor in general.

Destructors are automatically called when an object is destroyed. It is necessary inorder to de-allocate dynamic memory locations in order to stop memory leak.

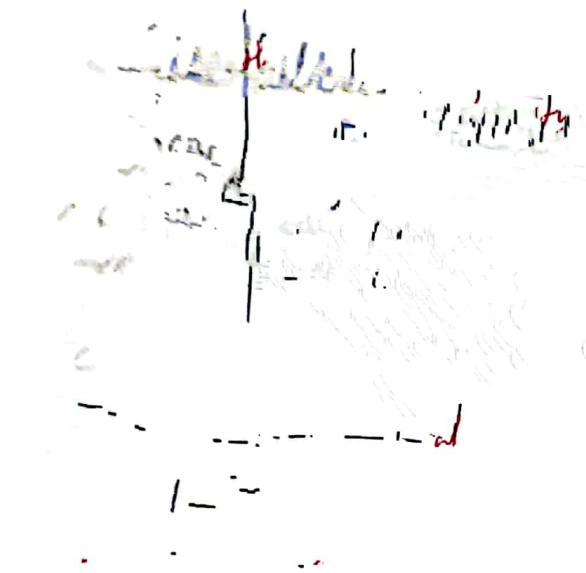
Three special properties of Constructor are:

i) Automatically called when an object is created

ii) They are public and we don't need to define any return type

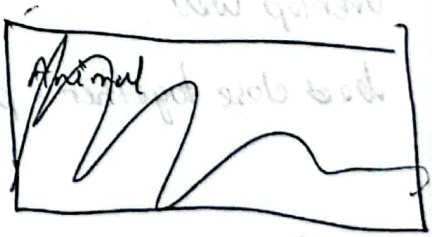
iii) They have the same name as the class.

5) b) If a copy constructor is not defined then cpp performs a bitwise copy of an object if we initialize another object with it. Hence, when we pass Animal obj to a function by value then it is seen that a copy will be made of the object that is passed as parameter. And if ~~Then~~ it happens to return the same obj. then another bitwise copy will be made. But when the function has ended then destructor for the Animal obj that is passed as parameter is for



```
//required copy constructor
Animal (const Animal & animal) {
    this -> age = new int;
    ~~this -> age =~~
    *age = ~~0~~ * animal.age;
}
```

function will be called. So, the memory address of age will be freed.. So, ~~when ever~~ whenever someone tries to access the age again then they ~~not~~ will face memory errors as the memory was already freed.



The way around the problems are:

i) defining a copy constructor to avoid bitwise copy

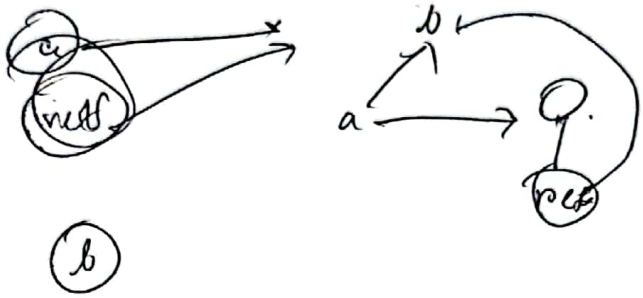
ii) Not using pointers in the Animal class, instead ~~using~~ use value int age.

iii) Passing ~~value~~ by ~~refer~~ reference or by pointer

iv) Passing the values only necessary for ~~over~~ some specific function and not the entire object.

8 // ~~a) If class B inherits from class A and A~~

a) The program would run without any issue if even if we not implement the virtual functions of A or B inside B. Virtual function are typically used for runtime polymorphism in cpp. But the ~~from~~ situation changes if A had some pure virtual function. A class having a some pure virtual function is an abstract function in cpp. No instances of the an abstract class can be not be created but we can create pointers of a abstract class and point it to any of its derived class which is not abstract. If B doesn't implement all the ~~abstract~~ pure virtual method of A, then B itself becomes an abstract class. A compiler will throw an error if we try to instantiate an object A or an object of B.



(b)

8//

b) 20 20 20 10

reason: When ref's value is changed a's value will also be changed as ref is the reference of the memory address from where a gets its value