OBJECT ORIENTED PROGRAMMING

NOTE:

MAKAUT course structure and syllabus of 5th semester has been changed from 2020. The syllabus of this subject is restructured & reorganized with selected topics from previous **OBJECT ORIENTED PROGRAMMING [CS 504D]**. Few new topics are introduced in present curriculum. Taking special care of this matter we are providing chapterwise relevant MAKAUT university solutions and some model questions & answers for newly introduced topics, so that students can get an idea about university questions patterns.

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

Multiple Choice Type Questions

1. Which one of the following statements is wrong? [WBUT 2012, 2015]

a) A base class reference can refer to an object of a derived class
b) The dynamic method dispatch is not carried out at the run time
c) The super( ) construct refers to the base class constructor
d) The super.base-class-method-name ( ) format can be used only within a derived class

Answer: (b)

2. Out of the following which one is not correctly matched? [WBUT 2012, 2015]

a) Int – 24 bits **b) Short – 16 bits** **c) Double – 64 bits** **d) Byte – 8 bits**

Answer: (a)

Short Answer Type Questions

1. What do you mean by object-oriented programming? [WBUT 2004, 2007]
How is it different from conventional procedural/structural programming? [WBUT 2004, 2005, 2007]

OR,

List out the differences between Procedure Oriented Programming and Object Oriented programming. [WBUT 2017]

Answer:

A type of programming in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure. In this way, the data structure becomes an *object* that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can *inherit* characteristics from other objects.

To perform object-oriented programming, one needs an *object-oriented programming language (OOPL)*. Java, C++ and Smalltalk are three of the more popular languages, and there are also object-oriented versions of Pascal.

PROCEDURAL LANGUAGE	OBJECT ORIENTED LANGUAGE
i) More concerned with the processing of procedures and functions.	i) Concerned to develop an object or application based on real time.
ii) It is not applicable to procedural language.	ii) More emphasis is given on data rather than procedures, while the programs are divided into Objects and the data is encapsulated (Hidden) from the external environment, providing more security to data.

PROCEDURAL LANGUAGE	OBJECT ORIENTED LANGUAGE
iii) Here it is possible to expose Data and/or variables to the external entities.	iii) Here it is strictly restricted.
iv) There is no communication in procedural language rather its simply a passing values to the Arguments to the Functions and / or procedures.	iv) The Objects communicate with each other via Functions.
v) It follows Top Down Approach to Program Execution.	v) It follows Bottom Up Approach of Program Execution.
vi) This type of programming uses traditional way of calling functions and returning values.	vi) Object oriented concepts includes Inheritance, Encapsulation and Data Abstraction, Late Binding, Polymorphism, Multithreading, and Message Passing
vii) **Examples:** C, VB, Perl, Basic, FORTRAN.	vii) **Examples:** JAVA, VB.NET, C#.NET

2. What is the difference between Java and C++ in respect of language functions? [WBUT 2014]

Answer:

Everything is an object in Java (Single root hierarchy as everything gets derived from java.lang.Object).

Java does not have all the complicated aspects of C++ (For ex: Pointers, templates, unions, operator overloading, structures etc..).

There are no destructors in Java. (automatic garbage collection).

Java does not support conditional compile (#ifdef/#ifndef type).

Thread support is built into java but not in C++.

Java does not support default arguments. There's no scope resolution operator :: in Java. Java uses the dot for everything, but can get away with it since you can define elements only within a class. Even the method definitions must always occur within a class, so there is no need for scope resolution there either.

There's no "goto" statement in Java.

Java doesn't provide multiple inheritance (MI), at least not in the same sense that C++ does. Exception handling in Java is different because there are no destructors. Java has method overloading, but no operator overloading. The String class does use the + and += operators to concatenate strings and String expressions use automatic type conversion, but that's a special built-in case.

Java is interpreted for the most part and hence platform independent.

3. What is an Abstract Data Type? How to implement an ADT? [MODEL QUESTION]

Answer:

An Abstract Data Type (ADT) is the specification of a data type within some programming language, independent of an implementation. The interface for the ADT is defined in terms of a type and a set of operations on that type. The behaviour of each operation is determined by its inputs and outputs. An ADT does not specify how the data type is implemented. A data structure is the implementation for an ADT. In an object-

oriented language like Java, an ADT and its implementation together make up a class. Each operation associated with the ADT is implemented by a member, function or method. The variables that define the space required by a data item are referred to as data members. An object is an instance of a class, that is, something that is created and takes up storage during the execution of a computer program.

The operations of an abstract data type are classified as follows:

1. **Creators** create new objects of the type. A creator may take an object as an argument, but not an object of the type being constructed.
2. **Producers** create new objects from old objects of the type. The concat method of String, for example, is a producer. It takes two strings and produces a new one representing their concatenation.
3. **Observers** take objects of the abstract type and return objects of a different type. The size method of List, for example, returns an int.
4. **Mutators** change objects. The add method of List, for example, mutates a list by adding an element to the end.

Implementation of ADT:
There can be different ways to implement an ADT. For example, the List ADT can be implemented using arrays, or singly linked list or doubly linked list. Similarly, stack ADT and Queue ADT can be implemented using arrays or linked lists.

In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored. The program allocates memory for the *data* and *address* is passed to the stack ADT. The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack. The stack head structure also contains a pointer to *top* and *count* of number of entries currently in stack.

Pseudocode:
This is a simple way of implementing the Stack ADT using an array. Here, elements are added from left to right and a variable keeps track of the index of the top element.

```
public class Stack {
    private static final int CAPACITY=10;
    private int capacity;
    private int top=-1;
    private Object [] obj;
    public Stack(int cap){
        capacity=cap;
        obj=new Object[capacity];
    }
    public Stack(){
        this(CAPACITY);
    }
    public int size(){
        return top+1;
```

```
}
public Object top(){
    if(isEmpty())
        return "Stack is empty.";
    return obj[top];
}
public boolean isEmpty(){
    return top<0;
}
public boolean isFull(){
    return size()>=capacity;
}
public void push(Object o){
    if(size()>=capacity){
        System.out.println("Stack is full.");
        return;
    }
    obj[++top]=o;
}
public Object pop(){
    if(isEmpty()) return "Stack is empty.";

    Object temp;
    temp=obj[top];
    obj[top--]=null;
    return temp;
}
```

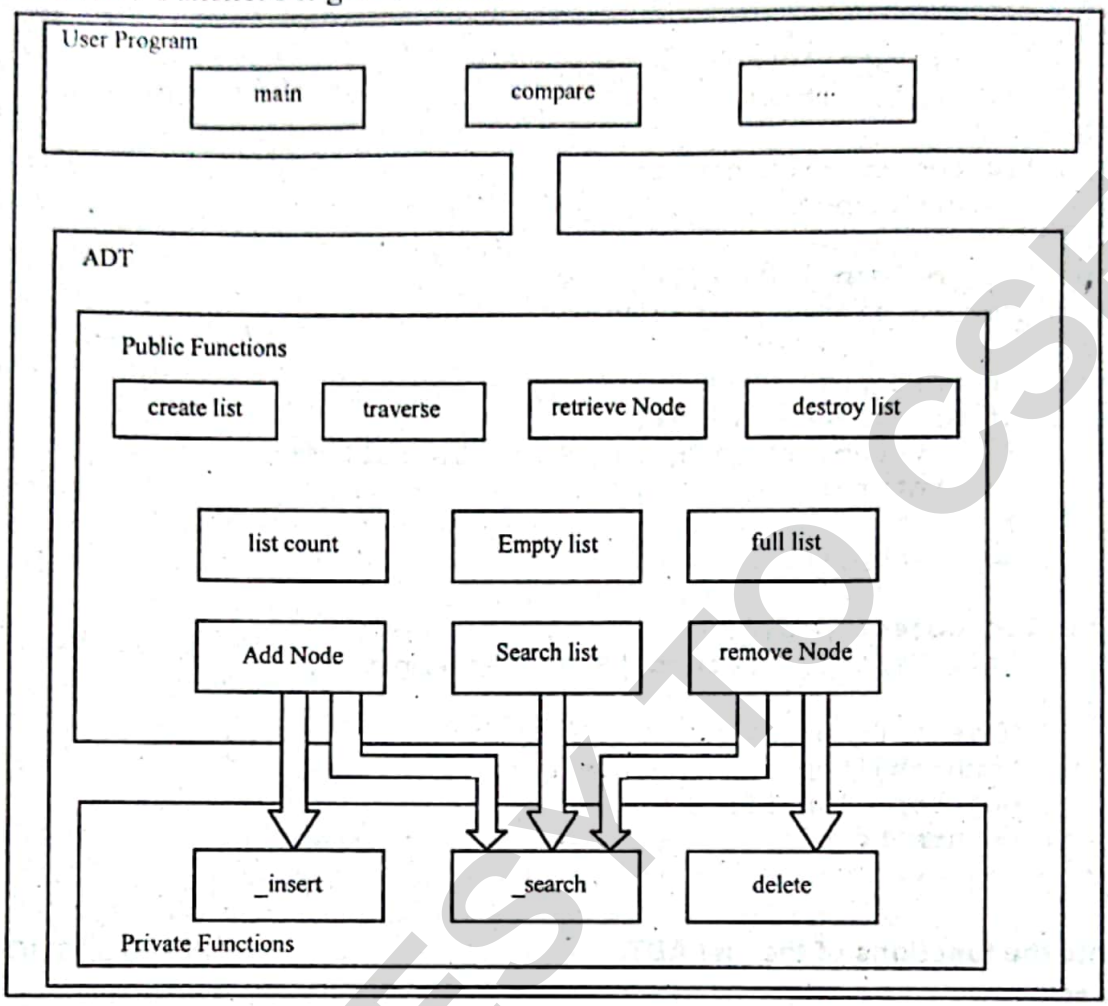
4. Write the functions of the List ADT. **[MODEL QUESTION]**

Answer:

A list contains elements of the same type arranged in sequential order and following operations can be performed on the list.

- get() – Return an element from the list at any given position.
- insert() – Insert an element at any position of the list.
- remove() – Remove the first occurrence of any element from a non-empty list.
- removeAt() – Remove the element at a specified location from a non-empty list.
- replace() – Replace an element at any position by another element.
- size() – Return the number of elements in the list.
- isEmpty() – Return true if the list is empty, otherwise return false.
- isFull() – Return true if the list is full, otherwise return false.

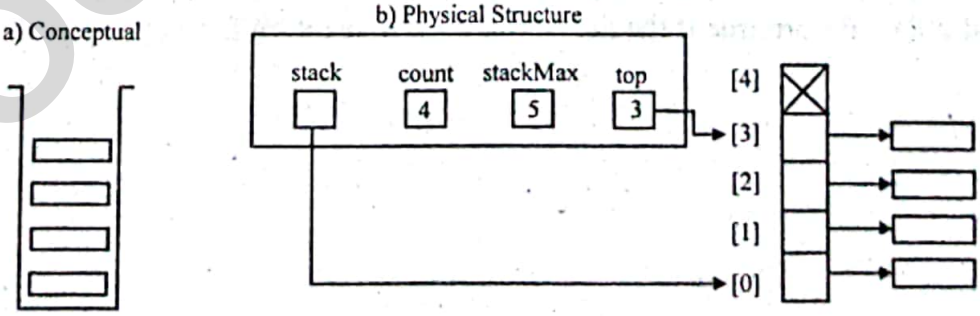
The List ADT Functions is given below:



5. What is Stack ADT? **[MODEL QUESTION]**

Answer:

In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored. The program allocates memory for the *data* and *address* is passed to the stack ADT. The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack. The stack head structure also contains a pointer to *top* and *count* of number of entries currently in stack.



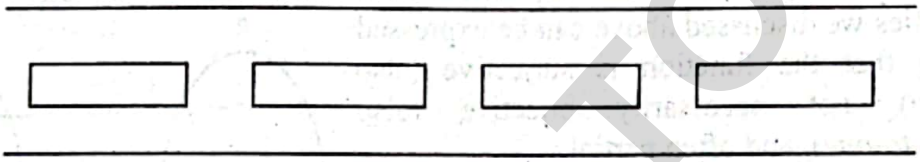
A Stack contains elements of the same type arranged in sequential order. All operations take place at a single end that is top of the stack and following operations can be performed:

- push() – Insert an element at one end of the stack called top.
- pop() – Remove and return the element at the top of the stack, if it is not empty.
- peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
- size() – Return the number of elements in the stack.
- isEmpty() – Return true if the stack is empty, otherwise return false.
- isFull() – Return true if the stack is full, otherwise return.

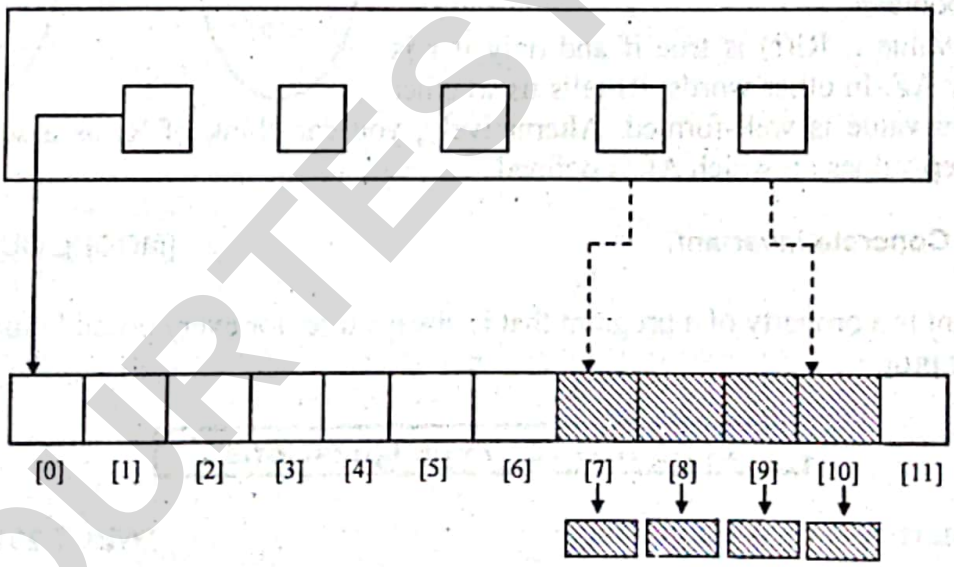
6. What is Queue ADT? Explain its operations. **[MODEL QUESTION]**

Answer:

The queue abstract data type (ADT) follows the basic design of the stack abstract data type. Each node contains a void pointer to the data and the link pointer to the next element in the queue. The program's responsibility is to allocate memory for storing the data.



a) Conceptual



b) Physical Structure

A Queue contains elements of the same type arranged in sequential order. Operations take place at both ends, insertion is done at the end and deletion is done at the front. Following operations can be performed:

- enqueue() – Insert an element at the end of the queue.

- dequeue() – Remove and return the first element of the queue, if the queue is not empty.
- peek() – Return the element of the queue without removing it, if the queue is not empty.
- size() – Return the number of elements in the queue.
- isEmpty() – Return true if the queue is empty, otherwise return false.
- isFull() – Return true if the queue is full, otherwise return false.

7. Define: Abstraction function. **[MODEL QUESTION]**

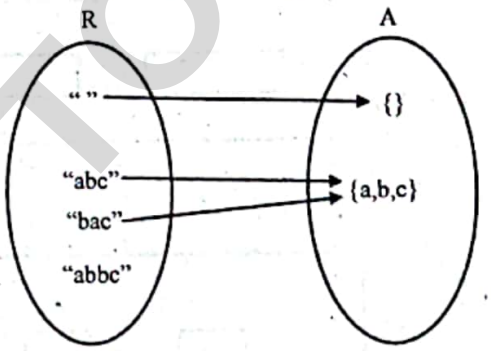
Answer:

An abstraction function maps a state of the concrete machine to a state of the abstract machine. It explains how to interpret each state of the concrete machine as a state of the abstract machine. It solves the problem of the concrete and abstract machines having different sets of states.

1. An *abstraction function* that maps rep values to the abstract values they represent:

AF : R → A

The arcs in the diagram show the abstraction function. In the terminology of functions, the properties we discussed above can be expressed by saying that the function is surjective (also called *onto*), not necessarily bijective (also called *one-to-one*), and often partial.



2. A *rep invariant* that maps rep values to booleans:

RI : R → boolean

For a rep value r, RI(r) is true if and only if r is mapped by AF. In other words, RI tells us whether a given rep value is well-formed. Alternatively, you can think of RI as a set: it's the subset of rep values on which AF is defined.

8. Define: Concrete invariant. **[MODEL QUESTION]**

Answer:

An invariant is a property of a program that is always true, for every possible runtime state of the program.

Long Answer Type Questions

1. Write short note on Abstraction. **[WBUT 2014, 2016]**

Answer:

Abstraction is "To represent the essential feature without representing the back ground details."

Abstraction lets us focus on what the object does instead of how it does it.

Abstraction provides us a generalized view of our classes or object by providing relevant information.

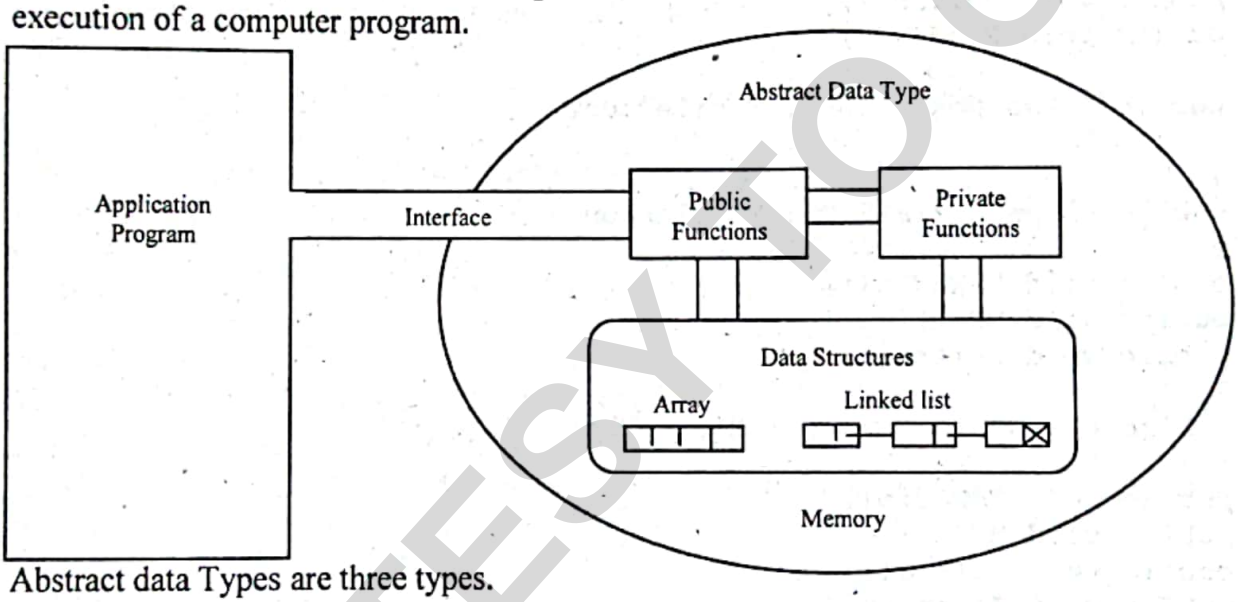
Abstraction is the process of hiding the working style of an object, and showing the information of an object in understandable manner.
Real world Example of Abstraction:
Suppose we have an object Mobile Phone.
Suppose we have 3 mobile phones as following:
Nokia 1400 (Features:- Calling, SMS)
Nokia 2700 (Features:- Calling, SMS, FM Radio, MP3, Camera)
Black Berry (Features:-Calling, SMS, FM Radio, MP3, Camera, Video Recording, Reading E-mails)
Abstract information (Necessary and Common Information) for the object "Mobile Phone" is make a call to any number and can send SMS."
So that, for mobile phone object we will have abstract class like following:

```
abstract class MobilePhone
{
public void Calling();
public void SendSMS();
}
public class Nokia1400 : MobilePhone
{
}
public class Nokia2700 : MobilePhone
{
public void FMRadio();
public void MP3();
public void Camera();
}
public class BlackBerry : MobilePhone
{
public void FMRadio();
public void MP3();
public void Camera();
public void Recording();
public void ReadAndSendEmails();
}
```

Abstraction means putting all the variables and methods in a class which are necessary.
For example: Abstract class and abstract method.
Abstraction is the common thing.
example:
If somebody in our collage tell us to fill application form, we will fill our details like name, address, data of birth, which semester, percentage you have got etc.
If some doctor gives us an application to fill the details, we will fill the details like name, address, date of birth, blood group, height and weight.
So, in the above example what is the common thing?
Age, name, address so you can create the class which consist of common thing that is called abstract class.
That class is not complete and it can inherit by other class.

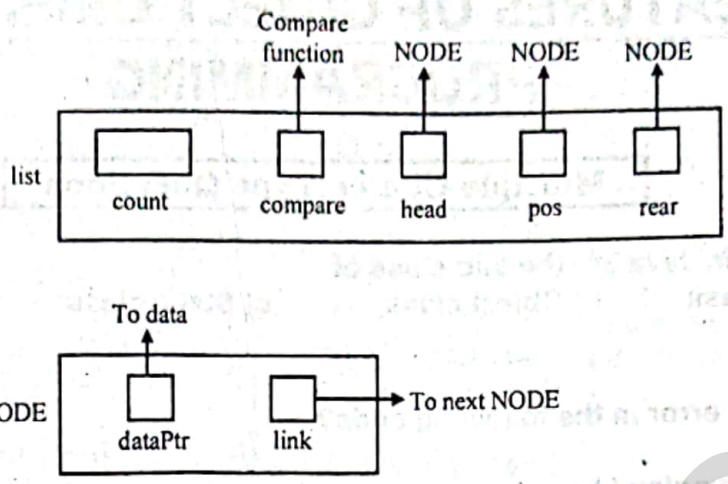
2. Write short note on Abstract data types and their specifications.
[MODEL QUESTION]

Answer:
An Abstract Data Type (ADT) is the specification of a data type within some programming language, independent of an implementation. The interface for the ADT is defined in terms of a type and a set of operations on that type. The behaviour of each operation is determined by its inputs and outputs. An ADT does not specify how the data type is implemented. These implementation details are hidden from the user of the ADT and protected from outside access, a concept referred to as Encapsulation.
A data structure is the implementation for an ADT. In an object-oriented language like Java, an ADT and its implementation together make up a class. Each operation associated with the ADT is implemented by a member, function or method. The variables that define the space required by a data item are referred to as data members. An object is an instance of a class, that is, something that is created and takes up storage during the execution of a computer program.



Abstract data Types are three types.
1. List ADT
2. Stack ADT
3. Queue ADT.

1. List ADT:
The data is generally stored in key sequence in a list which has a head structure consisting of *count*, *pointers* and *address of compare function* needed to compare the data in the list. The data node contains the *pointer* to a data structure and a *self-referential pointer* which points to the next node in the list.



2. Stack ADT:

In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored. The program allocates memory for the *data* and *address* is passed to the stack ADT. The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack. The stack head structure also contains a pointer to *top* and *count* of number of entries currently in stack.

3. Queue ADT

The queue abstract data type (ADT) follows the basic design of the stack abstract data type. Each node contains a void pointer to the *data* and the *link pointer* to the next element in the queue. The program's responsibility is to allocate memory for storing the data.

FEATURES OF OBJECT ORIENTED PROGRAMMING

Multiple Choice Type Questions

1. All classes in Java are the sub-class of [WBUT 2007, 2018]
   a) Final class b) Object class c) Static class d) Super class

Answer: (b)

2. What is the error in the following code? [WBUT 2007, 2009]

```
class Test {
abstract void display ( ) ;
}
```

   a) No error
   b) Method display ( ) should be declared as static
   c) Test class should be declared as abstract
   d) Test class should be declared as public

Answer: (c)

3. In which class is the wait ( ) method defined? [WBUT 2007, 2018]
   a) Applet b) Runnable c) Thread d) Object

Answer: (d)

4. Which statement about static inner classes is true? [WBUT 2008]
   a) Static inner classes may access any of the enclosing classes' members
   b) Static inner classes may have only static methods
   c) Static inner classes may not be instantiated outside of the enclosing class
   d) Static inner classes do not have a reference to the enclosing class

Answer: (a)

5. Select the most appropriate answer. [WBUT 2010]

```
public void Base( ) {
        System.out.println("Base");
}
public class In extends Base {
        public static void main(String argv[ ]) {
                IN i = new In( );
    .}
}
```

   a) Compile time error Base is a keyword
   b) Compilation and no output at run time
   c) Output of Base
   d) Run time error Base has no valid constructor.

Answer: (b)

6. Under what situations do you obtain a default constructor? **[WBUT 2010]**

a) When you define any class
b) When the class has no other constructors
c) When you define at least one constructor
d) None of these

Answer: (a)

7. What is the correct ordering for the import, class and package declarations when found in a single file? **[WBUT 2010, 2012, 2015]**

a) package, import, class
b) class, import, package
c) import, package, class
d) package, class, import

Answer: (a)

8. What will be the result of compiling the following code? **[WBUT 2010]**

```
public class Test {
    public static void main (String args[]) {
    int age;
    age = age + 1;
    System.out.println("The age is" + age);
}
}
```

a) Complies and runs with no output
b) Complies and runs printing out The age is 1
c) Complies but generates a run time error
d) Does not compile
e) Compiles but generates a compile time error

Answer: (d), variable age is not initialized.

9. What is the legal range of a byte integral type? **[WBUT 2010]**

a) 0 – 65, 535 b) (–128) – 127 c) (–32, 768) –32, 767 d) (–256) – 255

Answer: (b)

10. Which of the following returns true? **[WBUT 2010]**

a) `"john" == "john"`
b) `"john".equals("john")`
c) `"john" = "john"`
d) `"john".equal(new Button ("john"))`

Answer: (a) and (b)

11. Which of the following do not lead to runtime error? **[WBUT 2010]**

a) "john" + "was" + "here"
b) "john" + 3
c) 3 + 5
d) 5 + 5.5

Answer: none of the above

12. Which code declares class A to belong to the mypackage. financial package? **[WBUT 2011]**

a) package mypackage. : financial package
b) import mypackage.;
c) package mypackage. financial. A;
d) import mypackage. financial. *;
e) package mypackage. financial.

Answer: (c)

13. A package is a collection of [WBUT 2011]
a) Classes b) Interfaces c) Editing tools d) both (a) and (b)
Answer: (d)

14. Which of the following statements is valid array declaration? [WBUT 2011]
a) int number ( ) b) float average [ ] c) int marks d) count int [ ]
Answer: (b)

15. The use of protected keyword to a member in a class will restrict its visibility as
a) Visible only in the class and its subclass in the same package
b) Visible only inside the same package
c) Visible in all classes in the same package and subclasses in other package
d) None of these [WBUT 2011]
Answer: (a)

16. Which of the following is a Wrapper class? [WBUT 2011]
a) Byte b) Random c) Vector d) String
Answer: (a)

17. A sub-class having more than one super class is called [WBUT 2012]
a) category b) Classification c) Combination d) Partial participation
Answer: (a)

18. Which one of the following statements is not correct? [WBUT 2012]
a) An interface can inherit another interface
b) The package name and subdirectory name need not be identical
c) Only the classes declared as public in a package are accessible outside that package
d) The import java.awt.*; directive will not import classes in java.awt.event package
Answer: (d)

19. If a data-item is declared as a protected access specifier then it can be accessed [WBUT 2012]
a) anywhere in the program b) by the base and derived classes
c) only by base classes d) only by derived classes
Answer: (b)

20. What is the output of this code fragment? [WBUT 2012, 2015]
1. int x = 3; int y = 10;
2. System.out.println (y% x);
a) 0 b) 1 c) 2 d) 3
Answer: (b)

21. Which three form part of correct array declarations? [WBUT 2013]
1. public int a [ ]
2. static int [ ] a
3. public [ ] int a
4. private int a [3]
5. private int [3] a [ ]
6. public final int [ ] a

a) 1, 3, 4 b) 2, 4, 5 c) 1, 2, 6 d) 2, 5, 6

Answer: (c)

22. Which cause a compiler error? [WBUT 2013]
a) int [ ] scores = {3, 5, 7};
b) int [ ] [ ] scores = {2, 7, 6}, {9, 3, 45};
c) String cats [ ] = {"Fluffy", "Spot", "Zeus"};
d) boolean results [ ] = new Boolean [ ] {true, false, true};
e) Integer results [ ] = {new Integer (3), new Integer (5), new Integer (8)};

Answer: (b)

23. Which two cause a compiler error? [WBUT 2013]
1. float [ ] f = new float (3);
2. float f2[ ] = new float [ ];
3. float [ ]f1 = new float [3];
4. float f3 [ ] = new float [3];
5. float f5 [ ] = {1.0f, 2.0f, 2.0f};

a) 2, 4 b) 3, 5 c) 4, 5 d) 1, 2

Answer: (d)

24. public class Test { }
What is the prototype of the default constructor? [WBUT 2013]
a) Test ( ) b) Test (void) c) public Test ( ) d) public Test (void)

Answer: (c)

25. What is the most restrictive access modifier that will allow members of one class to have access to members of another class in the same package?
a) public b) abstract c) protected [WBUT 2013]
d) synchronized e) default access

Answer: (e)

26. You want a class to have access to members of another class in the same package. Which is the most restrictive access that accomplishes this objective?
a) public b) private [WBUT 2013]
c) protected d) default access

Answer: (d)

27. Byte code of java is [WBUT 2014]
a) platform dependent b) platform independent
c) no specific rule d) depend upon OS

Answer: (b)

28. Java virtual machine is [WBUT 2014, 2015]
a) platform dependent totally b) independent
c) depends on machine architecture only d) depends on OS only
Answer: (c)

29. Java is robust because [WBUT 2014]
a) it is object oriented b) garbage collection is present
c) platform independent d) exception handling
Answer: (d)

30. Constructor can be overloaded [WBUT 2014, 2015]
a) never b) always c) partially d) either (b) or (c)
Answer: (b)

31. Which of the following cannot be used for a variable name is Java? [WBUT 2016]
a) Identifier b) Keyword
c) Identifier & Keyword d) None of these
Answer: (b)

32. What is the range of the char type? [WBUT 2016]
a) 0 to 2^{16} b) 0 to 2^{15} c) 0 to $2^{16} - 1$ d) 0 to $2^{15} - 1$
Answer: (a)

33. The import statement is always [WBUT 2016]
a) the first non-comment statement in a java program file
b) the default non-comment statement in java program file
c) a non-comment statement and can be defined anywhere in the program
d) none of these
Answer: (a)

34. Which of the following values can a Boolean variable contain? [WBUT 2016]
a) True & False b) 0 & 1
c) Any integer value d) True
Answer: (a)

35. What is the output of this program? [WBUT 2016]

```
class area {
public static void main (String args[])
{
        double r, pi, a;
        r=9.8;
        pi=3.14;
        a=pi*r*r;
        System.out.printin(a);
    }
}
```

a) 301.5656 b) 301 c) 301.56 d) 301.56560000
Answer: (a)

36. How many public class can be allowed in Java? [WBUT 2017]
a) One b) Two c) Many d) None of these
Answer: (a)

37. Whether Java need compiler or interpreter [WBUT 2017]
a) compiler b) interpreter
c) both (a) and (b) d) none of these
Answer: (c)

38. Does any Java program contains more than one main method? [WBUT 2017]
a) Yes b) No
c) Sometimes it is possible d) In different package
Answer: (a)

39. AWT package is uses for [WBUT 2017]
a) Component and graphics b) Component
c) Graphics d) None of these
Answer: (a)

40. What is bytecode in context of Java? [WBUT 2018]
a) The type of code generated by a Java compiler
b) The type of code generated by Java Virtual Machine
c) It is another name for a Java Source file
d) It is the code written within the instance methods of a class
Answer: (b)

41. Which of the following statements regarding static methods are correct? [WBUT 2018]
a) Static methods are difficult to maintain, because you cannot change their implementation
b) Static methods can be called using an object reference to an object of the class in which this method is defined
c) Static methods are always public, because they are defined at class-level
d) Static methods do not have direct access to non-static methods which are defined inside the same class
Answer: (b)

42. Given the flowing piece of code [WBUT 2018]

```
public class C{
public abstract double calc_sa( );
}
```

Which of the following statements is true?
a) The keywords public and abstract cannot be used together
b) The method calc_sal() in class C must have a body
c) Must add a return statement in method calc_sal()
d) Class C must be defined abstract
Answer: (d)

43. A subclass is placed in a different package than the super class. In order to allow the subclass access a method defined in the super class, identify the correct access specifiers(s) **[WBUT 2018]**

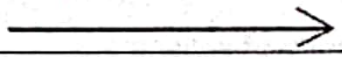
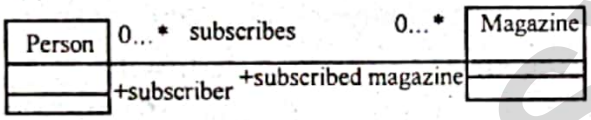
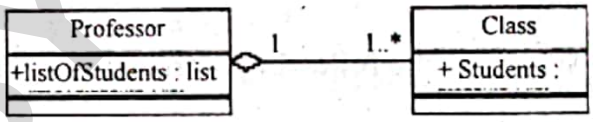
a) protected b) public c) private d) default

Answer: (c)

Short Answer Type Questions

1. Differentiate between association and aggregation. **[WBUT 2006, 2007, 2018]**

Answer:

ASSOCIATION	AGGREGATION
i) association is a relationship between two objects.	i) Aggregation is a special case of association.
ii) association defines the multiplicity between objects. for example one-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects	ii) A directional association between objects. When an object 'has-a' another object, then you have got an aggregation between them. Direction between them specified which object contains the other object. Aggregation is also called a "Has-a" relationship.
iii) Denoted by, 	iii) Denoted by, 
iv)  Class diagram example of association between two class	iv)  Class diagram showing Aggregation between two classes

2. Explain various access modifiers (public, protected, private, default) in a class definition. **[WBUT 2006]**

OR,

What is the difference between default access specifier and public access specifier?

Explain the differences between the 'private' and 'protected' access specifier in java. **[WBUT 2013]**

OR,

Differentiate between protected and friendly access specifier. **[WBUT 2018]**

Answer:

One of the techniques in object-oriented programming is *encapsulation*. It concerns the hiding of data in a class and making this class available only through methods. In this way the chance of making accidental mistakes in changing values is minimized. Java allows you to control access to classes, methods, and fields via so-called *access specifiers*.

Java offers four access specifiers, listed below in decreasing accessibility:

- public
- protected
- default (no specifier)
- private

We look at these access specifiers in more detail.

public

`public` classes, methods, and fields can be accessed from everywhere. The only constraint is that a file with Java source code can only contain one `public` class whose name must also match with the filename. If it exists, this `public` class represents the application or the applet, in which case the `public` keyword is necessary to enable your Web browser or appletviewer to show the applet. You use `public` classes, methods, or fields only if you explicitly want to offer access to these entities and if this access cannot do any harm. An example of a square determined by the position of its upper-left corner and its size:

```
public class Square {   // public class
  public x, y, size;    // public instance variables
}
```

protected

`protected` methods and fields can only be accessed within the same class to which the methods and fields belong, within its subclasses, and within classes of the same package, but not from anywhere else. You use the `protected` access level when it is appropriate for a class's subclasses to have access to the method or field, but not for unrelated classes.

default (no specifier)

If you do not set access to specific level, then such a class, method, or field will be accessible from inside the same package to which the class, method, or field belongs, but not from outside this package. This access-level is convenient if you are creating packages. For example, a `geometry` package that contains `Square` and `Tiling` classes, may be easier and cleaner to implement if the coordinates of the upper-left corner of a `Square` are directly available to the `Tiling` class but not outside the `geometry` package.

private

`private` methods and fields can only be accessed within the same class to which the methods and fields belong. `private` methods and fields are not visible within subclasses and are not inherited by subclasses. So, the `private` access specifier is opposite to the `public` access specifier. It is mostly used for encapsulation: data are hidden within the class and accessor methods are provided. An example, in which the position of the upper-left corner of a square can be set or obtained by accessor methods, but individual coordinates are not accessible to the user.

```
public class Square {    // public class
private double x, y   // private (encapsulated) instance
variables
public setCorner(int x, int y) {  // setting values of private
fields
this.x = x;
this.y = y;
}
public getCorner() {  // setting values of private fields
return Point(x, y);
}
}
```

Summary of Access Specifiers

The following table summarizes the access level permitted by each specifier.

Situation	public	protected	default	private
Accessible to class from same package?	yes	yes	yes	no
Accessible to class from different package?	yes	no, *unless it is a subclass*	no	no

Note the difference between the default access which is in fact more restricted than the `protected` access. Without access specifier (the default choice), methods and variables are accessible only within the class that defines them and within classes that are part of the same package. They are not visible to subclasses unless these are in the same package. `protected` methods and variables are visible to subclasses regardless of which package they are in.

3. Differentiate between composition and aggregation with suitable example.
[WBUT 2007, 2011]

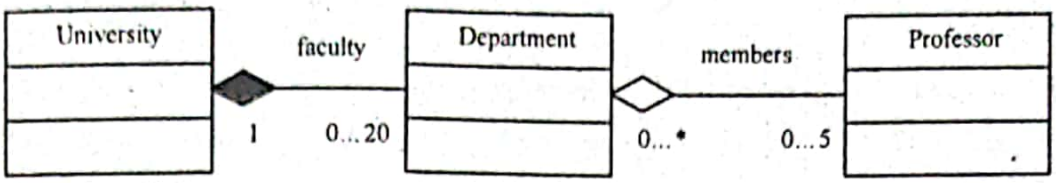
Answer:

Aggregation differs from ordinary composition in that it does not imply ownership. In composition, when the owning object is destroyed, so are the contained objects. In aggregation, this is not necessarily true. For example, a university owns various departments (e.g., chemistry), and each department has a number of professors. If the university closes, the departments will no longer exist, but the professors in those departments will continue to exist. Therefore, a University can be seen as a composition of departments, whereas departments have an aggregation of professors. In addition, a Professor could work in more than one department, but a department could not be part of more than one university.

In aggregation, the object may only contain a reference or pointer to the object (and not have lifetime responsibility for it):

Sometimes aggregation is referred to as composition when the distinction between ordinary composition and aggregation is unimportant.

UML Class diagram:



4. State the Difference between the following:
a) Object and object reference **[WBUT 2009, 2010]**
b) Static and final keyword **[WBUT 2009, 2010, 2018]**

Answer:

a) Employee s = new Employee ("Joe",20);
Here 's' is not an object, it's a variable which contains a reference to an object. Objects don't have names, just types and locations in memory (and, of course, fields and methods). The above statement can be read as: Create a new Employee object in memory, initializing it with the data sent as arguments to a constructor, and when created, assign a reference to that object to the Employee variable 's'. 's' is a reference or object type variable which may reference a Employee object or an object of any subclass of Employee.

b) A **final class** cannot be extended. This is done for reasons of security and efficiency. A **final method** cannot be overridden by subclasses.
A **final variable** can only be initialized once, either via an initializer or an assignment statement.
The "final" keyword is useful when applied to variables whose values will not change during the lifetime of the program. If you've got constant values in your program which will not change throughout, it is useful to declare them as final.
The keyword ``static" when applied to a variable OR a method means that the variable or method can be accessed without creating an instance of the class.
For examples of static variables see the Color class in the Java API. It uses static variables such as BLACK, BLUE, PINK etc.

They can be referenced like:
Code:

```
useColor(Color.BLACK);
```

instead of
Code:

```
Color myColorBlack = new Color();
useColor(myColorBlack.BLACK);
```

static can be used for methods for much the same effect.
When "final" is applied to a class, the principle effect is that the class cannot be inherited from. For example, the following would throw an error:
Code:

```
public final class Dog {
```

```
public Dog() {
}
}
public class JackRussell extends Dog {
public JackRussell() {
}
}
```

[WBUT 2009, 2010]

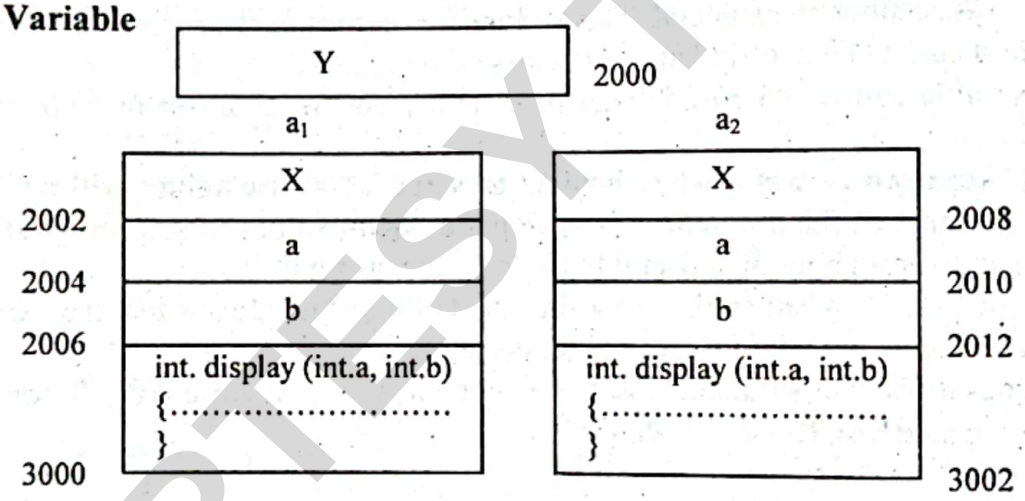
```
5. class A {
  private int x;
  static int y;
int display (int a, int b)
 {
   System.out.prinln("X=" + x + "Y=" + y);
}
```

What will be the memory allocation for attributes and method on objects a1, a2 of type A? Only picture of a1 and a2 along with their member and their memory size required.

Answer:

The memory allocation for static variable 'y'.

Static Variable



6. Differentiate between Early binding and late binding. [WBUT 2010, 2018]

OR,

What is the difference between static binding and dynamic binding? [WBUT 2011]

Answer:

Late Binding:

- At run time, when it is known what class objects are under consideration, the appropriate version of function is invoked. Since the function is linked with a particular class much later after the compilation, this process is known as ***late binding.*** It is also known as ***dynamic binding*** because the selection of the appropriate function is done at run time dynamically. Dynamic binding requires use of pointers to objects.

- Late binding is implemented when it is not known which function will be called, though early binding is faster than late binding.

Early Binding:

- The overloaded member functions are selected for invoking by matching arguments. The compiler knows this information at the compile time and, therefore, compiler is able to select appropriate function for a particular call at compile time itself. This is called ***early binding*** or ***static binding*** or ***static linking***. This is also known as ***compile time polymorphism***
- Early binding determines execution path at compilation and late binding allows for dynamic execution at runtime.
  for example: In a native Win32 code environment (i.e., non .NET), late binding could refer to the use of a DLL library vs. the use of a static library - all the references in a static library can be determined at compile time, but the references in a DLL (dynamic link library) are not determined later until run time.

7. Write a Java program to implement the following:
Create a class named as FIRST and keep it in a package named MY_FIRST_PACKAGE
Create another class named as SECOND, this class should be able to access every method and variables declared within the FIRST class. [WBUT 2011]

Answer:

Suppose we have a file called First.java, and we want to put this file in a package **my_first_package.** First thing we have to do is to specify the keyword **package** with the name of the package we want to use (**my_first_package** in our case) on top of our source file, before the code that defines the real classes in the package, as shown in our First class below:

```
package my_first_package;
public class First{
public static void main(String[] args) {
void Show(){
System.out.println("My first class");
    }
  }
}
```

One thing you must do after creating a package for the class is to create nested subdirectories to represent package hierarchy of the class. In our case, we have the my_first_package package, which requires only one directory. So, we create a directory my_first_package and put our First.java into it.

Right now we have First class inside my_first_package package. Next, we have to introduce the my_first_package package into our **CLASSPATH.**

Now, to create another class named as SECOND, which class be able to access every method and variables declared within the FIRST class, can be written as:

```
package my_first_package;
```

```
public class Second {
public static void main(String[] args) {
First f = new First();
f.show();
System.out.println("My Second class");
 }
}
```

8. What are the main characteristics of OOP Language? Explain each. [WBUT 2012]

Answer:

Encapsulation:

In programming, it is the process of combining elements to create a new entity. For example, a procedure is a type of encapsulation because it combines a series of computer instructions. Likewise, a complex data type, such as a record or class, relies on encapsulation. Object-oriented programming languages rely heavily on encapsulation to create high-level objects. Encapsulation is closely related to abstraction and information hiding.

Customer, waiter and kitchen are three shielded objects in the 'cup of coffee' example. Customer and kitchen do not know each other. The waiter is the intermediary between those two. Objects can't see each other in an Object Oriented world. The 'hatch' enables them to communicate and exchange coffee and money.

Encapsulation keeps computer systems flexible. The business process can change easily. The customer does not care about the coffee brew process. Even the waiter does not care. This allows the kitchen to be reconstructed, is only the 'hatch' remains the same. It is even possible to change the entire business process. Suppose the waiter will brew coffee himself. The customer won't notice any difference.

Encapsulation enables OO experts to build flexible systems. Systems that can extend as your business extends. Every module of the system can change independantly, no impact to the other modules.

Polymorphism:

In object-oriented programming, polymorphism (from the Greek meaning "having multiple forms") is the characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow an entity such as a variable, a function, or an object to have more than one form. There are several different kinds of polymorphism.

1) A variable with a given name may be allowed to have different forms and the program can determine which form of the variable to use at the time of execution. For example, a variable named USERID may be capable of being either an integer (whole number) or a string of characters (perhaps because the programmer wants to allow a user to enter a user ID as either an employee number - an integer - or with a name - a string of characters). By giving the program a way to distinguish which form is being handled in each case, either kind can be recognized and handled.

2) A named function can also vary depending on the parameters it is given. For example, if given a variable that is an integer, the function chosen would be to seek a match against a list of employee numbers; if the variable were a string, it would seek a match against a list of names. In either case, both functions would be known in the program by the same name. This type of polymorphism is sometimes known as overloading.

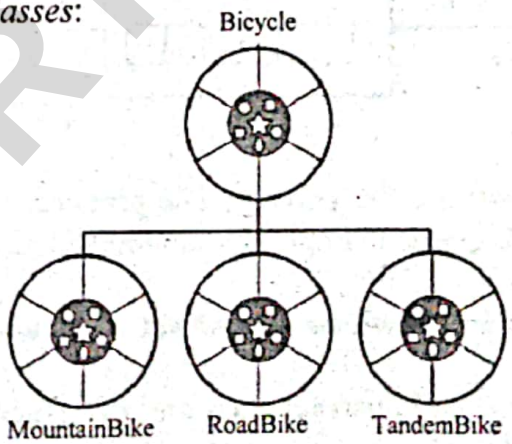
You drive an automobile, which has properties like wheel size, engine size, gas tank size, and other properties. The automobile you drive is a concrete implementation of the *automobile interface*. It has additional features, like sliding doors, logo type, cd changer slot count, moon roof lever location, or other various properties that are specific to the make/model of the car. In addition, automobile may have certain behaviors like open/close door, open trunk, turn wheel, and other behaviors that would be specific to an automobile.

In OO programming, using the automobile example, Automobile would be the base class, and each automobile manufacturer would have its own implementation. For instance, Honda has V-Tec technology, which is in its own implementation. Volvo uses diesel engines, which is the TDI technology. More importantly, you may add an added level of detail between automobile and the make/model implementation, such as Car, Truck, or Suv super types, to provide more relevant information.

Inheritance:

Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes. In this example, Bicycle now becomes the *superclass* of MountainBike, RoadBike, and TandemBike. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of *subclasses*:



A hierarchy of bicycle classes.

The syntax for creating a subclass is simple. At the beginning of your class declaration, use the extends keyword, followed by the name of the class to inherit from:

```
class MountainBike extends Bicycle {
    // new fields and methods defining a mountain bike would go
here
}
```

This gives MountainBike all the same fields and methods as Bicycle, yet allows its code to focus exclusively on the features that make it unique. This makes code for your subclasses easy to read. However, you must take care to properly document the state and behavior that each superclass defines, since that code will not appear in the source file of each subclass.

9. What do you meant by 'Dynamic Method Dispatch'? [WBUT 2013]

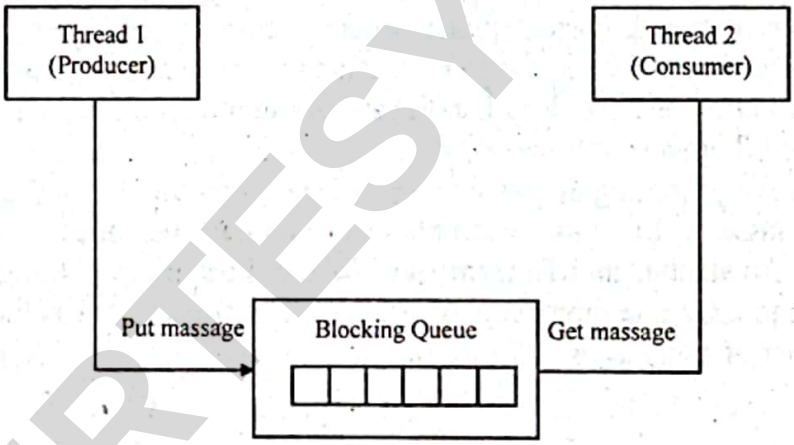
Answer:

Refer to Question No. 6(a) of Long Answer Type Questions.

10. What is message passing? [WBUT 2018]

Answer:

Massage passing is a form of communication used in object-oriented programming as well as parallel programming : Massage passing in Java is like sending an object i.e. massages from one thread to another thread. It is used when threads do not have shared memory and are unable to share variables to communicate.



As a example,

The producer and Consumer are the Threads. The producer will produce and consumer will consume only. We use Queue to implement communication between thread.

11. Write a Java program to show use of abstract class and Interface. [WBUT 2018]

Answer:

A Java Program to show use of abstract class and Interface:

```
interface  MyInterface
{
public void method1 ();
public void method2 ();
```

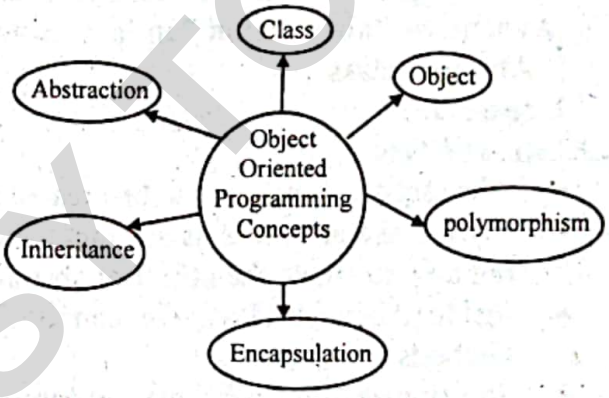
```
}
class Demo implements  MyInterface
{
public void method1 ();
  {
   System.out.println ("implementation of method1");
  }
public void method2 ();
  {
   System.out.println ("implementation of method2");
   }
public static void main (String args[ ])
  {
  MyInterface obj = newDemo ();
  Obj.method1();
  }
}
```

12. Write down the features of Object oriented Programming. [MODEL QUESTION]

Answer:

Features of OOPs:

1. Classes
2. Objects
3. Data Abstraction
4. Encapsulation
5. Inheritance
6. Polymorphism



1. Class: Class represents a real world entity which acts as a blueprint for all the objects. We can create as many objects as we need using Class.

Example:

We create a class for " Student " entity as below

```
Student.java
Class Student{
String id;
int age;
String course;
void enroll(){
System.out.println("Student enrolled");
}
}
```

Above definition of class contains 3 fields id,age and course and also it contains behavior or a method called " enroll ".

2. Objects: Object Oriented Programming system(OOPS) is designed based on the concept of **"Object".** It contains both **variables** (used for holding the data) and **methods** (used for defining the behaviors). We can create any number of objects using this class and all those objects will get the same fields and behavior.

Student s1 = new Student();

Now we have created 3 objects **s1,s2** and **s3** for the same class **" Student ".** We can create as many objects as required in the same way.

We can set the value for each field of an object as below,

```
s1.id=123;
s2.age=18;
s3.course="computers";
```

3. Data Abstraction:

Abstraction is a process where you show only "relevant" data and "hide" unnecessary details of an object from the user.

For example, when you login to your bank account online, you enter your user_id and password and press login, what happens when you press login, how the input data sent to server, how it gets verified is all abstracted away from the you.

We can achieve " **abstraction** " in Java using **2 ways**

1. **Abstract class**
2. **Interface**

1. Abstract Class

- Abstract class in Java can be created using " abstract " keyword.
- If we make any class as abstract then it can't be instantiated which means we are not able to create the object of abstract class.
- Inside Abstract class, we can declare abstract methods as well as concrete methods.
- So using abstract class, we can achieve 0 to 100 % abstraction.

Example:

```
Abstract class Phone{
void receiveCall();
Abstract void sendMessage();
}
```

Anyone who needs to access this functionality has to call the method using the Phone object pointing to its subclass.

2. Interface

- Interface is used to achieve pure or complete abstraction.
- We will have all the methods declared inside Interface as abstract only.
- So, we call interface as 100% abstraction.

Example:

We can define interface for Car functionality abstraction as below

```
Interface Car{
public void changeGear( int gearNumber);
public void applyBrakes();
}
```

Now these functionalities like changing a gear and applying brake are abstracted using this interface.

4. Encapsulation:

- Encapsulation is the process of binding object state (fields) and behaviors (methods) together in a single entity called "Class".
- Since it wraps both fields and methods in a class, it will be secured from the outside access.
- We can restrict the access to the members of a class using access modifiers such as private,protected and public keywords.
- When we create a class in Java, it means we are doing encapsulation.
- Encapsulation helps us to achieve the re-usability of code without compromising the security.

Example:

```
class EmployeeCount
{
private int numOfEmployees = 0;
public void setNoOfEmployees (int count)
{
numOfEmployees = count;
}
public double getNoOfEmployees ()
{
return numOfEmployees;
}
}
public class EncapsulationExample
{
public static void main(String args[])
{
EmployeeCount obj = new EmployeeCount ();
obj.setNoOfEmployees(5613);
System.out.println("No Of Employees:
"+(int)obj.getNoOfEmployees());
}
}
```

5. Inheritance:

- One class inherits or acquires the properties of another class.
- Inheritance provides the idea of reusability of code and each sub class defines only those features that are unique to it, rest of the features can be inherited from the parent class.
  1. Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.
  2. It allows us to reuse of code, it improves reusability in your java application.
  3. The parent class is called the **base class** or **super class**. The child class that extends the base class is called the derived class or **sub class** or **child class**.

To inherit a class we use extends keyword. Here class A is child class and class B is parent class.

```
class A extends B
{
}
```

6. Polymorphism:

- It is the concept where an object behaves differently in different situations.
- Since the object takes multiple forms, it is called Polymorphism.
- In java, we can achieve it using method overloading and method overriding.
- There are 2 types of Polymorphism available in Java,

Method overloading

In this case, which method to call will be decided at the compile time itself based on number or type of the parameters. Static/Compile Time polymorphism is an example for method overloading.

Method overriding

In this case, which method to call will be decided at the run time time based on what object is actually pointed by the reference variable.

Long Answer Type Questions

1. Distinguish between the following terms:

i) Data abstraction and data encapsulation **[WBUT 2004, 2007, 2011]**

ii) Dynamic binding and message passing. **[WBUT 2004, 2008]**

Answer:

i)

DATA ENCAPSULATION	DATA ABSTRACTION
i) Encapsulation is hiding the implementation details which may or may not be for generic or specialized behavior(s).	i) Abstraction is providing a generalization (say, over a set of behaviors).
ii) Encapsulation means hiding the internal details or mechanics of how an object does something.	ii) Abstraction lets you focus on what the object does instead of how it does it
iii) Encapsulation require modularity. It requires you to create objects that has the data and the methods to process the data. In this case you can view it as a module.	iii) Abstraction provides you a generalized view of your classes.
iv) encapsulation is wrapping up of a data into single unit **Example :** A real world example, consider u have setup a big building(say a company), the details regarding materials used to built (glass, bricks), type of work, manager of the company, number of floors, design of the building, cost of the building etc. can be classified as ABSTRACTION.	iv) abstraction is hiding unnecessary background details and representing only important and essential detail **Example:** Whereas, type of glass or bricks(grey one or red one) used, who all work for which all departments n how they work, cost of each and every element in the building etc. comes under Data ENCAPSULATION.

ii)

DYNAMIC BINDING	MESSAGE PASSING
i) Dynamic binding is binding a call to a particular method at run-time. It is also referred to as late binding. ii) Dynamic binding is the process of resolving the function to be associated with the respective functions calls during their runtime rather than compile time. **Example:** Suppose all life-forms are mortal. In object oriented programming, we can say that the Person and Plant classes must implement the Mortal interface, which contains the method die(). Persons and Plants die in different ways; for example, Plants do not have hearts that stop. **Dynamic binding** is the practice of figuring out which method to invoke at runtime. For example, if we write `void kill(Mortal&m){ m.die(); }` it's not clear whether m is a Person or a Plant, and thus whether Plant.die() or Person.die() should be invoked on the object. With dynamic binding, the m object is examined at runtime, and the method corresponding to its actual class is invoked.	i) Message passing is a form of communication where objects (instances) exchange messages. ii) Every data in an object in oops that is capable of processing request known as message .All object can communicate with each other by sending message to each other. Message passing enables extreme late binding in systems. **Example:** For example, the object called Breeder may tell the Lassie object to sit by passing a "sit" message which invokes Lassie's "sit" method. The syntax varies between languages, for example: [Lassie sit] in Objective-C. In Java, code-level message passing corresponds to "method calling". Some dynamic languages use double-dispatch or multi-dispatch to find and pass messages.

2. a) Explain role name in an association with example. **[WBUT 2004]**

OR,

Discuss association. **[WBUT 2013]**

b) What is aggregation? How aggregation is different from association and generalization? **[WBUT 2004]**

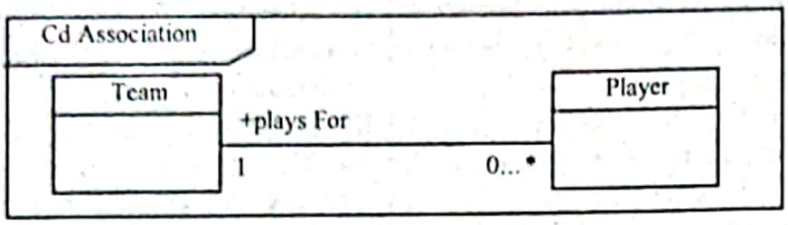
OR,

Discuss aggregation. **[WBUT 2013]**

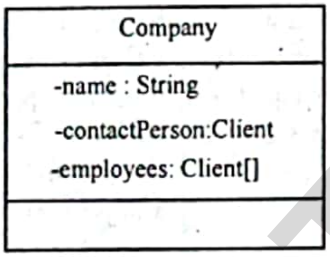
Answer:

a) An association implies two model elements have a relationship - usually implemented as an instance variable in one class. This connector may include named roles at each end, cardinality, direction and constraints. Association is the general relationship type between elements. For more than two elements, a diamond representation toolbox element can be used as well. When code is generated for class diagrams, named association ends become

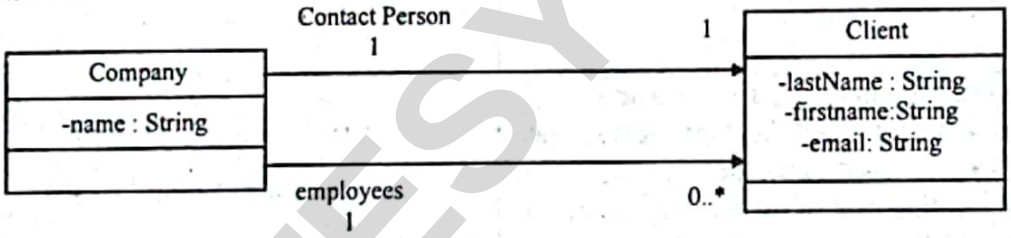
instance variables in the target class. So, for the example below, "playsFor" will become an instance variable in the "Player" class.



An association end role is a specialization of an association end, used to describe an association end's behavior in a particular context.
In the UML metamodelAssociationEndRole is a sub-class of AssociationEnd.
Two or more association end roles are associated with each association role.
Classes can also contain references to each other. The *Company* class has two attributes that reference the Client class.



Although this is perfectly correct, it is sometimes more expressive to show the attributes as associations.



The above two associations have the same meaning as the attributes in the old version of the *Contact* class.
The first association (the top one) represents the old *contactPerson* attribute. There is one contact person in a single *Company*. The multiplicity of the association is one to one meaning that for every *Company*there is one and only one *contactPerson* and for each *contactPerson* there is one *Company*. In the bottom association there are zero or many employees for each company. Multiplicities can be anything you specify. Some examples are shown:

0	Zero
1	One
1..*	one or many
1..2, 10..*	one, two or ten and above but **not** three through nine

The arrows at the end of the associations represent their navigability. In the above examples, the *Company* references *Clients*, but the *Client* class does not have any knowledge of the *Company*. You can set the navigability on either, neither or both ends of your associations. If there is no navigability shown then the navigability is unspecified.

b) Difference between Association and Aggregation:

Refer to Question No. 1 of Short Answer Type Questions.

Difference between Aggregation and Generalization:

AGGREGATION	GENERALIZATION
i) A directional association between objects. When an object 'has-a' another object, then you have got an aggregation between them.	i) Generalization uses a "is-a" relationship from a specialization to the generalization class. Common structure and behavior are used from the specialization to the generalized class. At a very broader level we can understand this as inheritance.
ii) Denotes by, 	ii) Denotes by, 
iii) It means for example one College is build up of Departments and again departments contains classes right here school is aggregation of departments and again department is aggregation of classes, here you can perfect aggregation of things if you delete the main object called school all associated departments and classes will get delete.	iii) Consider there exists a class named Person. A student is a person. A faculty is a person. Therefore here the relationship between student and person, similarly faculty and person is generalization.

3. What are the differences between 'abstract class' and 'interface'?

[WBUT 2005, 2007, 2009, 2010, 2011, 2013, 2018]

Answer:

	Abstract class	Interface
1.	An abstract class can contain implementation of some methods apart from the abstract methods. This makes abstract class to make a read only class.	An interface can have only abstract methods.
2.	A class that extends or inherits an abstract class must reside in the same class hierarchy where the parent abstract class belongs.	A class that implements or inherits an interface need not be in the same hierarchy in which the interface belongs.
3.	A new abstract class cannot be fitted easily into a class hierarchy. For example if two classes inherits or extends the same new abstract class, then the abstract class needs to be placed higher up in the hierarchy above these two classes. This can disturb the class hierarchy, as this will force all the descendants to extend the new abstract class. This may involve some extra implementation overhead for these descendants.	Multiple classes can implement a new interface, as it is not a mandate that these classes will be in the same hierarchy in which the interface belongs.

	Abstract class	Interface
4.	A class can at most extend or inherit one single parent. This is known as single directional inheritance.	A class can implement multiple interfaces and at the same time can extend a class for any concrete implementation. This mixed type of inheritance helps us to design more flexible data structures. This is known as multi-directional inheritance.

4. What do you mean by parameter passing? What is call by value and call by reference? Write down two programs to define call by value and call by reference. **[WBUT 2014]**

OR,

a) What do you mean by parameter passing?
b) What is the difference between call by value or pass by value and call by reference or pass by reference? Explain. **[WBUT 2015]**

Answer:

1st Part:

Parameter passing is the mechanism used to pass parameters to a procedure (subroutine) or function. The most common methods are to pass the value of the actual parameter (*call by value*), or to pass the address of the memory location where the actual parameter is stored (*call by reference*). The latter method allows the procedure to change the value of the parameter, whereas the former method guarantees that the procedure will not change the value of the parameter. Other more complicated parameter-passing methods have been devised, notably *call by name* in Algol 60, where the actual parameter is re-evaluated each time it is required during execution of the procedure.

2nd Part:

In programming, there are two ways to pass arguments to a method, call-by-value and call-by-reference:

- When we have a call-by-value parameter, a copy of the argument is stored into the memory location allocated for the formal parameter. In this case, any changes made to the formal parameter inside the method will not affect the value of the argument back in the calling method.
- When a parameter is call-by-reference, the memory address of the argument is passed to the method, making the formal parameter an alias for the argument. This means that changes made to the formal parameter inside the method will be reflected in the value of the argument when control is returned to the calling function.

3rd Part:

There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

Example of call by value in java

```
class Operation{
int data=50;
void change(int data){
data=data+100;//changes will be in the local variable only
}
public static void main(String args[]){
Operation op=new Operation();
System.out.println("before change "+op.data);
op.change(500);
System.out.println("after change "+op.data);
}
}
```

Output:

```
before change 50
 after change 50
```

Another Example of call by value in java

In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed. In this example we are passing object as a value. Let's take a simple example:

```
class Operation2{
int data=50;
void change(Operation2 op){
op.data=op.data+100;//changes will be in the instance variable
}
public static void main(String args[]){
Operation2 op=new Operation2();
System.out.println("before change "+op.data);
op.change(op);//passing object
System.out.println("after change "+op.data);
}
}
```

Output:

```
before change 50
 after change 150
```

5. What are the different characteristics of abstract keyword? Explain abstract class through a program. **[WBUT 2015]**

Answer:

1st Part:

The "abstract" keyword can be used on classes and methods. A class declared with the "abstract" keyword cannot be instantiated, and that is the only thing the "abstract" keyword does. Example of declaring a abstract class:

```
abstract CalendarSystem (String name);
```

When a class is declared abstract, then, its methods may also be declared abstract.

When a method is declared abstract, the method can not have a definition. This is the only effect the abstract keyword has on method. Here's a example of a abstract method:

```
abstract int get_person_id (String name);
```

Abstract classes and abstract methods are like skeletons. It defines a structure, without any implementation.

Only abstract classes can have abstract methods. Abstract class does not necessarily require its methods to be all abstract.

Classes declared with the abstract keyword are solely for the purpose of extension (inheritance) by other classes. The characteristics of abstract class are given below:

1) Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
2) Abstract class can have normal functions and variables along with a pure virtual function.
3) Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
4) Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

2nd Part:

The purpose of an abstract class is to specify the default functionality of an object and let its sub-classes to explicitly implement that functionality. Thus, it stands as an abstraction layer that must be extended and implemented by the corresponding sub-classes.

A sample example of using an abstract class is the following. We declare an abstract class, called Instrument:

Instrument.java:

```
abstract class Instrument {
 protected String name;
abstract public void play();
}
```

As we can observe, an Instrument object contains a field name and a method called play, that must be implemented by a sub-class.

Next, we define a sub-class called StringedInstrument that extends the Instrument class and adds an extra field called numberOfStrings:

```
StringedInstrument.java:
abstract class StringedInstrument extends Instrument {
     protected int numberOfStrings;
}
```

Finally, we add two more classes that implement the functionality of a StringedInstrument, calledElectricGuitar and ElectricBassGuitar accordingly.

The definition of these newly added classes is shown below:

```
ElectricGuitar.java:
public class ElectricGuitar extends StringedInstrument {
public ElectricGuitar() {
super();
this.name = "Guitar";
```

```
this.numberOfStrings = 6;
}
public ElectricGuitar(int numberOfStrings) {
super();
this.name = "Guitar";
this.numberOfStrings = numberOfStrings;
}
@Override
public void play() {
System.out.println("An electric " + numberOfStrings + "-string "
+ name
+ " is rocking!");
}}
```

ElectricBassGuitar.java:

```
public class ElectricBassGuitar extends StringedInstrument {
public ElectricBassGuitar() {
super();
this.name = "Bass Guitar";
this.numberOfStrings = 4;
}
public ElectricBassGuitar(int numberOfStrings) {
super();
this.name = "Bass Guitar";
this.numberOfStrings = numberOfStrings;
}
@Override
public void play() {
System.out.println("An electric " + numberOfStrings + "-string "
+ name
+ " is rocking!");    }}
```

Finally, we create a new class called Execution that contains a single main method:

Execution.java:

```
import main.java.music.ElectricBassGuitar;
import main.java.music.ElectricGuitar;
public class Execution {
public static void main(String[] args) {
ElectricGuitar guitar = new ElectricGuitar();
ElectricBassGuitar bassGuitar = new ElectricBassGuitar();
guitar.play();
bassGuitar.play();
guitar = new ElectricGuitar(7);
bassGuitar = new ElectricBassGuitar(5);
guitar.play();
bassGuitar.play();
}}
```

In this example, we create two different instances of an ElectricGuitar and an ElectricBassGuitar classes and we call their play methods. A sample execution of the aforementioned main method is shown below:

```
An electric 6-string Guitar is rocking!
An electric 4-string Bass Guitar is rocking!
An electric 7-string Guitar is rocking!
An electric 5-string Bass Guitar is rocking!
```

6. Write short notes on the following:

a) Dynamic method dispatch [WBUT 2008, 2017]
b) Encapsulation [WBUT 2014]
c) Link and Association. [WBUT 2015]
d) Co-Varient Return [WBUT 2015]
e) Polymorphism [WBUT 2016]
f) Abstract Class [WBUT 2017]
g) Dynamic Binding [WBUT 2017, 2018]
h) Method overriding with example [WBUT 2017]
i) Properties of OOP [WBUT 2018]

Answer:

a) Dynamic method dispatch:

Dynamic method dispatch is a concept which comes during overriding a method. This is also known as runtime polymorphism. The actual flavor of polymorphism comes with overriding. Only methods can be overridden. Usually the child class overrides methods of a parent class. By inheritance methods of a parent class is always available in the child class. The child class just changes the implementation details.

For example let us consider the following class definition

Parent.java

```
public class Parent
{
public void display()
{
System.out.println("In parent");
}
}
```

Child.java

```
public class Child extends Parent
{
public void display()
{
System.out.println("In child");
}
}
```

MainInherit.java

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

```
Parent p = new Child(); // dynamic method dispatch
Child c = (Child)p;
c.display();
}
}
```

Explanation

`Parent p = new Child();`

During compile time, p is the object reference of class Parent. But during execution of the program, it binds to an object of sub-class Child.

`Child c = (Child)p;`

Finally,

- `c.display();` executes and an output is given like this

In Child

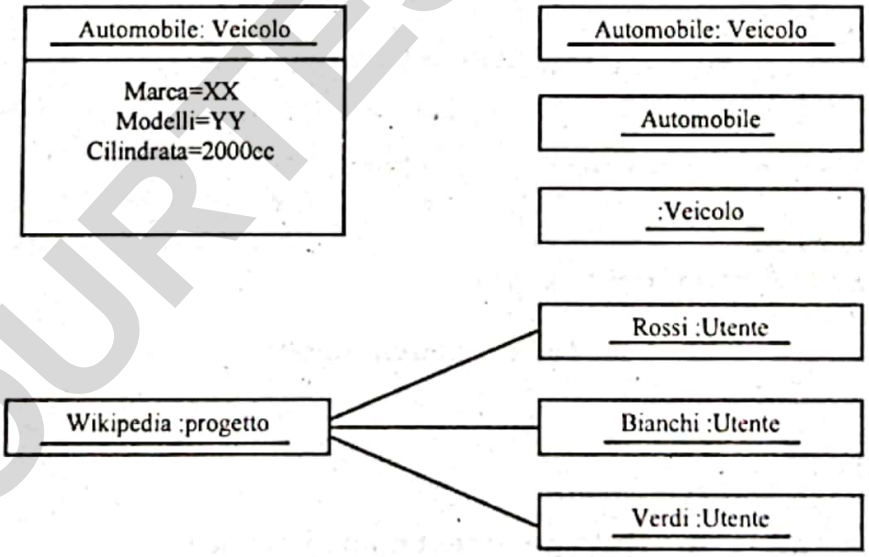
The overridden version of display( ) is called. This is known as **runtime polymorphism** as the call to the method is decided during the execution time of the program as the dynamic linking is done only at that time.

b) Encapsulation: ***Refer to Question No. 8 of Short Answer Type Questions.***

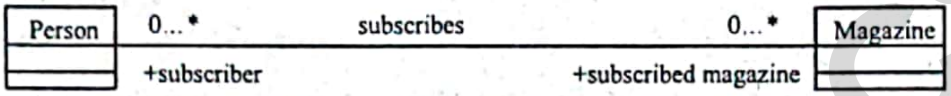
c) Link and Association:

Link: A *Link* is the basic relationship among objects. It is represented as a line connecting two or more object boxes. It can be shown on an object diagram or class diagram. A link is an instance of an association. In other words, it creates a relationship between two classes.

Object Diagram



Association: An *Association* represents a family of links. Binary associations (with two ends) are normally represented as a line, with each end connected to a class box. Higher order associations can be drawn with more than two ends. In such cases, the ends are connected to a central diamond.
An association can be named, and the ends of an association can be adorned with role names, ownership indicators, multiplicity, visibility, and other properties. There are five different types of association. Bi-directional and uni-directional associations are the most common ones. For instance, a flight class is associated with a plane class bi-directionally. Associations can only be shown on class diagrams. Association represents the static relationship shared among the objects of two classes. Example: "department offers courses", is an association relation.



LINK	ASSOCIATION
i) link is related to objects	i) association is related to classes

d) Co-Varient Return:
A covariant return type is a method return type that, in the superclass's method declaration, is the supertype of the return in the subclass's overriding method declaration. Lising 4-21 provides a demonstration of this language feature.
A Demonstration of Covariant Return Types

```
Class SuperReturnType
{
      @Override
      Public String toString()
      {
            return "superclass return type";
      }
}
class SubReturnType extends SuperReturnType
{
      @Override
      Public String toString()
      {
          return "subclass return type";
      }
}
class Superclass
{
          SuperReturnType createReturnType()
          {
          Return new SuperReturnType();
      }
}
```

```
Class Sublcass extends Superclass
{
      @override
      SubReturnType createReturnType()
      {
            Return new SubReturnType();
      }
}
public class CovarDemo
{
Public static void main(String[] args)
{
   SuperReturnType = new Superclass().createReturnType();
   System.out.println(suprt); // Output: Superclass return type
   SubReturn subrt = new Subclass().createReturnType();
   System.out.printin(subrt); // Output: subclass return type
    }
}
```

SuperReturnType and Superclass superclasses and SubReturnType and Subclass subclasses; each of Superclass and Subclass declares a createReturnType() method. Superclass's method has its return type set to SuperRteturnType, whereas Subclass's overriding method has its return type set to SubReturnType, a subclass of SuperReturnType.

Covariant return types minimize upcasting and downcasting. For example, Subclass's createReturnType() method doesn't need to upcast its SubReturnType instance to its SubReturnType return type. Futhermore, this instance doesn't need to be downcast to SubReturnType when assigning to variable subrt.

e) Polymorphism:

Another fundamental object-oriented mechanism in Java is polymorphism. Java allows polymorphism on methods. Polymorphism, meaning one object and many shapes, is a simple concept that allows a method to have multiple implementations. This is also known as *method overload*. Following illustrates the idea of polymorphism:

// Polymorphism concept //

```
class point {
   int x, y;
   Point (int x, int y) {      // It is a constructor
      this.x = y;
      this.y = y;
   }
/*M1*/float distance (int x, int y) { // One definition of
distance
      int dx = this.x - x;
      int dy = this.y - Y;
      return float Math.sqrt (dx * dx + dy * dy);
   }
```

```
/*M2*/float distance (Point p) { //Overload definition of
distance return distance (p.x, p.y);}

Class Point 3D extends point {
   int z;
Point 3D (int x, int y, int z) { // Constructor of Point 3D
   super (x, y);
   this.z = z;
}
/*M3*/ float distance (int x, int y, int z) {//Another definition
of distance
   int dx = this.x - x;
   int dy = this.y - y;
   int dz = this,z - z;
   return (float) Math.sqrt (dx * dx + dy * dy + dz * dz);
}
/*M4*/ float distance (Point 3D pt) {
      return distance (pt.x, pt.y, pt.z);
   }
}
class PointDistance {
   public static void main (String args[]) {
      Point p1 = new Point (10, 5);          //2-D point
      Point p3 = new Point3D (5, 10, 5)      //3-D point
      Point p2 = new Point (4, 1);           //another 2-D point
      Point p4 = new Point3D (2, 3, 4);      //another 3-D point
      float d0 = p1.distance (0, 0);         //M1 will be referred
      float d1 = p1.distance (p2);           //M2 will be referred
      System.out.println ("Distance from p1 to Origin =" + d0);
      System.out.println ("Distance from p2 to p1 =" + d1);
      d0 = p3.distance (0, 0, 0);            //M4 will be referred
      d1 = p4.distance (p3);                 //M4 will be referred
      System.out.println ("Distance from p3 to origin =" + d0);
      System.out println ("Distance from p3 to p4 =" + d1);
   }
}
```

Output:

Distance from P1 to Origin	11.180341
Distance from P2 to P1	7.211102
Distance from P3 to Origin	12.247449
Distance from P3 to P4	7.6811457

In the above example, we have seen how the same method can be implemented in different ways. The concept of this type of method overloading is the same as in C++. However, C++ allows operator overloading, Java does not.

f) Abstract Class:

Refer to Question no. 5(2nd Part) of Long Answer Type Questions.

g) Dynamic binding:

i) Dynamic binding also called dynamic dispatch is the process of linking procedure call to a specific sequence of code (method) at run-time. It means that the code to be executed for a specific procedure call is not known until run-time. Dynamic binding is also known as late binding or run-time binding. Dynamic binding requires use of pointers to objects.

ii) Late binding is implemented when it is not known which function will be called, though early binding is faster than late binding. Method overriding is termed as dynamic binding.

h) Method overriding in java:

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**. In other words, if subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

1) Method overriding is used to provide specific implementation of a method that is already provided by its super class.

2) Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

Example:

```
// method overriding in java
// Base Class
class Parent
{
    void show() { System.out.println("Parent's show()"); }
}
// Inherited class
class Child extends Parent
{
    // This method overrides show() of Parent
    @Override
    void show() { System.out.println("Child's show()"); }
}
// Driver class
class Main
{
    public static void main(String[] args)
    {
```

```
        // If a Parent type reference refers
        // to a Parent object, then Parent's
        // show is called
        Parent obj1 = new Parent();
        obj1.show();
        // If a Parent type reference refers
        // to a Child object Child's show()
        // is called. This is called RUN TIME
        // POLYMORPHISM.
        Parent obj2 = new Child();
        obj2.show();
    }
}
```

Output:

Parent's show()
Child's show()

i) Properties of OOP: ***Refer to Question No. 8 of Short Answer Type Questions.***

INHERITANCE IN OO DESIGN

Multiple Choice Type Questions

1. Exception is defined in .....package. [WBUT 2007, 2009, 2018]
   a) java.util b) java.lang c) java.awt d) java.io

Answer: (b)

2. Consider the class [WBUT 2011]

```
1. Public class Over {
2. public int test (int a, int b)
3. }
4. // add here
5. }
```

Which of the following overloaded methods would be legal is added at line 4?
   a) Public float test (float a, float b) { }
   b) Public int test (float a, float b) { }
   c) Public int test (int x, int y) { }
   d) Public float test (int a, int b) { }

Answer: (a) & (b)

3. Which three are valid method signatures in an interface? [WBUT 2013]
   1. private int getArea();
   2. public float get Vol(float x);
   3. public void main (String [] args);
   4. public static void main (String [] args);
   5. boolean setFlag (Boolean [] test);

   a) 1 and 2 b) 2, 3 and 5 c) 3, 4 and 5 d) 2 and 4

Answer: (b)

4. class A [WBUT 2013]

```
{
  protected int method 1 (int a, int b)
  {
    Return 0;
  }
}
```

Which is valid in a class that extends *class A*?
   a) public int method 1 (int a, int b) {regurn0;)
   b) private int method 1 (int a, int b) {return 0;}
   c) public short method 1 (int a, int b) {return 0;)
   d) static protected int method 1 (int a, int b) {return 0;)

Answer: (a)

5. Which is a valid declaration within an interface? [WBUT 2013]
   a) pubic static short stop = 23;
   b) protected short stop = 23;
   c) transient short stop = 23;
   d) final void madness (short stop);

Answer: (a)

6. What is the narrowest valid return Type for method A in line 3? [WBUT 2013]

```
public class ReturnIt
{
  return Type method A (byte x, double y)/*Line 3*/
  {
     Return (long)x/y*2;
  }
}
```

   a) int
   b) byte
   c) long
   d) double

Answer: (d)

7. Runtime binding occurs [WBUT 2014]
   a) when method overloaded
   b) interface only
   c) class and interface
   d) only subpackage

Answer: (c)

8. Abstract class is used for [WBUT 2014]
   a) inheritance only
   b) instantiation only
   c) both (a) and (b)
   d) useless

Answer: (a)

9. What is an example of polymorphism? [WBUT 2016]
   a) Inner class
   b) Anonymous classes
   c) Method overloading
   d) Method overriding

Answer: (c)

10. The relation between classes can be represented by [WBUT 2016]
   a) polymorphism
   b) method
   c) message
   d) inheritance

Answer: (d)

11. Method overloading occurs only when [WBUT 2016]
   a) the names and the type signature of two methods are not identical
   b) the names and the type signature of two methods are identical
   c) the names and the return types of two methods are identical
   d) only the names are identical

Answer: (b)

12. Final is useful [WBUT 2017]
   a) to protect a method from overloading
   b) to protect a class from inherit
   c) to protect a interface from implementing
   d) none of these

Answer: (a) and (b)

13. Dynamic method dispatcher is useful for **[WBUT 2017]**

a) to resolve method override
b) to resolve multilevel inheritance annomaly
c) to resolve multiple inheritance annomaly
d) none of these

Answer: (a)

14. An abstract class can contain no method at all **[WBUT 2017]**

a) True b) False c) Variable d) None of these

Answer: (b)

15. The parent class of all the exceptions in Java is **[WBUT 2018]**

a) Throwable b) Throw c) Exception d) Throws

Answer: (c)

16. String univ = new String ("WBUT"); **[WBUT 2018]**
System.out.printlb(univ.length());
What is printed?

a) 6 b) 4 c) 8 d) WBUT

Answer: (b)

Short Answer Type Questions

1. What are the benefits of using design patterns? **[MODEL QUESTION]**

Answer:

Some of the benefits of using design patterns are:

1. Design Patterns are already defined and provides industry standard approach to solve a recurring problem, so it saves time if we sensibly use the design pattern. There are many java design patterns that we can use in our java based projects.
2. Using design patterns promotes reusability that leads to more robust and highly maintainable code. It helps in reducing total cost of ownership (TCO) of the software product.
3. Since design patterns are already defined, it makes our code easy to understand and debug. It leads to faster development and new members of team understand it easily.

2. What is iterator pattern in java? How it is implemented? **[MODEL QUESTION]**

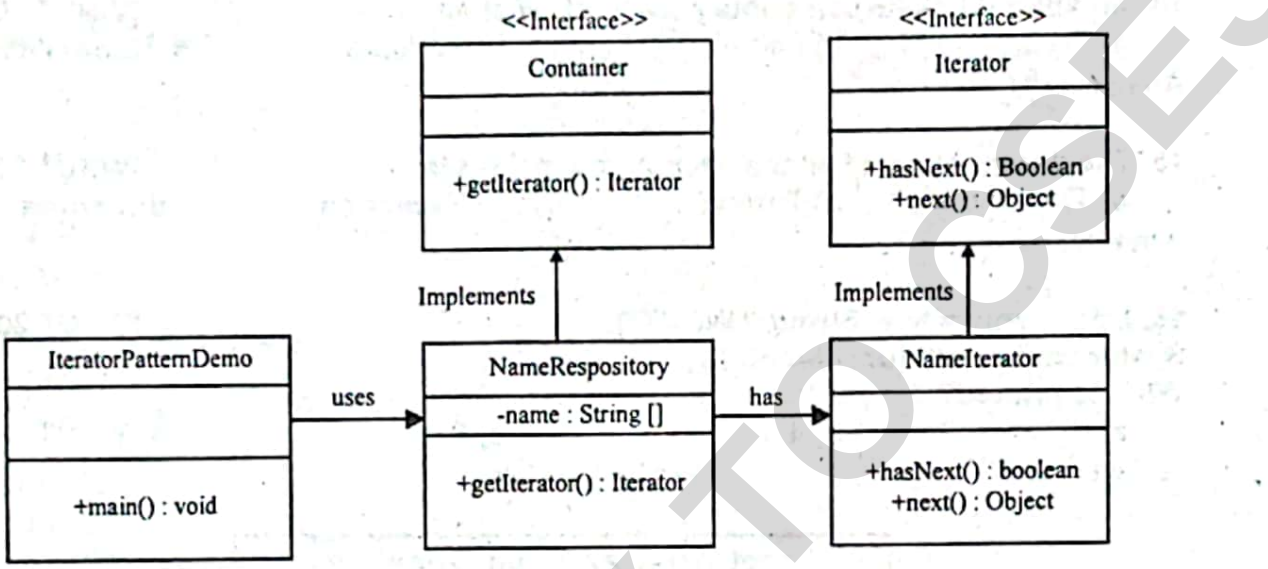
Answer:

Iterator pattern is very commonly used design pattern in Java environment. This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation. Iterator pattern falls under behavioral pattern category.

Implementation

Here create a Iterator interface which narrates navigation method and a Container interface which returns the iterator. Concrete classes implementing the Container interface will be responsible to implement Iterator interface and use it.

IteratorPatternDemo, our demo class will use NamesRepository, a concrete class implementation to print a Names stored as a collection in NamesRepository.



Step 1:

Create interfaces.

Iterator.java

```
public interface Iterator {
   public boolean hasNext();
   public Object next();
}
```

Container.java

```
public interface Container {
   public Iterator getIterator();
}
```

Step 2:

Create concrete class implementing the Container interface. This class has inner class NameIterator implementing the Iterator interface.

NameRepository.java

```
public class NameRepository implements Container {
   public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};
   @Override
   public Iterator getIterator() {
      return new NameIterator();
   }
```

```
   private class NameIterator implements Iterator {
      int index;

      @Override
      public boolean hasNext() {

         if(index < names.length){
            return true;
         }
         return false;
      }
      @Override
      public Object next() {

         if(this.hasNext()){
            return names[index++];
         }
         return null;
      }
   }
}
```

Step 3:
Use the NameRepository to get iterator and print names.

IteratorPatternDemo.java

```
public class IteratorPatternDemo {
   public static void main(String[] args) {
      NameRepository namesRepository = new NameRepository();

      for(Iterator iter = namesRepository.getIterator();
iter.hasNext();){
         String name = (String)iter.next();
         System.out.println("Name : " + name);
      }
   }
}
```

Step 4:
Verify the output.

```
Name : Robert
Name : John
Name : Julie
Name : Lora
```

3. What is design pattern in java? What are the categories in which the design patterns can be divided? Explain in each. **[MODEL QUESTION]**

Answer:

1st Part:

The design pattern is one of the most common repeatable solutions that are given to many different software designs. Design patterns always systematically names, motivate and also explains the general designs that address the design pattern. A design pattern is a well-described solution to a common software problem.

2nd Part:

The following are the categories in which the design pattern can be divided and they are:

- Creational pattern
- Behavioral patterns
- Functional patterns
- Concurrency pattern

A creational design pattern is a type of design pattern that mainly deals with the object of the creational mechanism. It also helps to create an object in a manner that is suitable for different situations.

A behavioral design pattern is the type of pattern that identifies the common communication between the objects and also understands these types of patterns. By this method, these types of design pattern increase the flexibility to carry the communication.

The following are examples of the behavioral design pattern:

- Blackboard design pattern
- Chain of Responsibility pattern
- Command pattern
- Interpreter pattern
- Iterator pattern
- Mediator pattern
- Memento pattern
- Null object pattern
- Observe pattern
- Visitor pattern

The concurrency patterns are the types of design patterns that mainly deals with the multi-thread programming paradigm.

The following are some of the concurrency design patterns:

- Active object
- Baulking pattern
- Barrier
- Double checked locking

- Guarded suspension
- Monitor object
- Read-write lock pattern

4. What are the most important software design patterns? [MODEL QUESTION]

Answer:

The following are some of the most important software design patterns:

- Singleton
- Factory method
- Strategy
- Observer
- Builder
- Adapter
- State

The Singleton design patterns are the types of patterns that mainly help to restrict the instantiation of the classes to one. This Singleton design pattern is very helpful when exactly one object is needed to control all the above systems.

The factory pattern will always work around the super factories that generally helps to create other types of factories. This factory design pattern falls under the creational pattern. This is because it helps to create another type of object.

Builder patterns help to create more of the complex patterns by the use of simple types of objects. Using steps by step process the Builder patterns create more complex patterns. These types of patterns also fall under the creational patterns.

5. What are the differences between design patterns and the frameworks? [MODEL QUESTION]

Answer:

The following are some of the differences between the design pattern and the framework:

Design Patterns	Framework
1. A design pattern is the type of pattern that mainly deals with the object-oriented software.	1. The framework is mainly made up of the group of the concrete classes that can be directly implemented on the platform.
2. The design patterns always represent the solutions to the problems that mainly arises when developing software within a particular context.	2. Frameworks are mainly concerned with the specific application domains.
3. The types of design pattern are not written in the programming language.	3. Frameworks are mainly written with the programming language.

6. What is proxy in design pattern? What are the types of proxy pattern? What are the benefits of proxy in design pattern? **[MODEL QUESTION]**

Answer:

1st Part:

Proxy generally means the in place of representing or on the behalf. Proxies are also known as the surrogate, handles and wrappers. The Proxy design is closely related to the structural design pattern.

2nd Part:

The following are some of the types of the proxy design you will get in the design pattern they are:

- Remote Proxy
- Virtual proxy
- Protection Proxy
- Smart proxy

3rd Part:

The following are some of the benefits of the Proxy in the design Pattern and they are:

- One of the best advantages of the Proxy design pattern is security. The proxy patterns increase security by installing the local code proxy.
- The proxy pattern avoids all the duplications of the objects that may be of the highest size and acquire huge memory.
- The proxy patterns also increase the performance of the application.

Long Answer Type Questions

1. a) How Java implements multiple inheritance? Justify your answer with a small example. **[WBUT 2005]**

OR,

i) Multiple inheritance can be performed in Java. Explain how?
ii) Write a Java program in support of your views. **[WBUT 2011]**

OR,

Explain whether java supports multiple inheritance or not. **[WBUT 2013]**

b) What is method overloading? **[WBUT 2005, 2008]**

OR,

Explain Function overloading with an example. **[WBUT 2013]**

How is it different from method over-riding? **[WBUT 2005, 2006]**

OR,

Differentiate between Method overloading and Method overriding. **[WBUT 2009, 2010, 2011, 2013, 2015, 2018]**

c) Write how to prevent method over-riding. **[WBUT 2005, 2008]**

Answer:

a) When Sun was designing Java, it omitted multiple inheritance - or more precisely multiple implementation inheritance - on purpose. Yet multiple inheritance can be useful, particularly when the potential ancestors of a class have orthogonal concerns. This article

presents a utility class that not only allows multiple inheritance to be simulated, but also has other far-reaching applications.
Have you ever found yourself wanting to write something similar to:

```
public class Employee extends Person, Employment {
// detail omitted
}
```

Here, Person is a concrete class that represents a person, while Employment is another concrete class that represents the details of a person who is employed. If you could only put them together, you would have everything necessary to define and implement an Employee class. Except in Java - you can't. Inheriting implementation from more than one superclass - multiple implementation inheritance - is not a feature of the language. Java allows a class to have a single superclass and no more.
On the other hand, a class can implement multiple interfaces. In other words, Java supports multiple interface inheritance. Suppose the PersonLike interface is:

```
public interface PersonLike {
String getName();
int getAge();
}
and the EmployeeLike interface is:
public interface EmployeeLike {
float getSalary();
java.util.Date getHireDate();
}
```

If Person implements the Person-Like interface, and Employment implements an EmployeeLike interface, it's perfectly acceptable to write:

```
public class Employee implements PersonLike, EmployeeLike {
// detail omitted
}
Here there is no explicit superclass. Since we are allowed to
specify at most one superclass, we could also write:
public class Employee extends Person implements PersonLike,
EmployeeLike {
// detail omitted

}
We would need to write the implementation of EmployeeLike, but
the implementation of PersonLike is taken care of through the
Person superclass. Alternatively we might write:
public class Employee extends Employment implements PersonLike,
EmployeeLike{
// detail omitted
}
```

This is the opposite situation: the EmployeeLike interface is taken care of through the Employment superclass, but we do need to write an implementation for PersonLike.

Java does not support multiple implementation inheritance, but does support multiple interface inheritance. When you read or overhear someone remark that Java does not support multiple inheritance, what is actually meant is that it does not support multiple implementation inheritance.

b) 1st Part:

This type of polymorphism is achieved during writing of the program. Programmers are aware of the polymorphic nature they want to impose. Methods as well as constructors can be made polymorphic within the class body. Each method/constructor have the same name but with different arguments. This phenomenon is known as overloading. Parameterized constructors are said to be an overloaded version of the default constructor.

The rules of overloading are:

- The parameters of the methods/constructors should vary in their numbers and data-type.
- The return type of the methods/constructors is not considered while overloading.

For example let us consider the following class definition

```
public class Overload
{
public void sum(int x, int y){ }
}
```

The valid overloading methods in the same class are:

1. **Change in data type of the arguments**
   ```
   public void sum(float x, float y){ }
   public void sum(float x, int y){ }
   ```
2. **Change in the number of arguments**
   ```
   public void sum(int x, int y, int z){ }
   ```
3. **Return type is not considered. Hence the following line is not a valid overloading.**
   ```
   //public int sum(int x, int y){ }
   ```

Although we are using the same name in the overloaded versions but actually each method is different and is unique on it's own. That is why, often overloading is not considered as polymorphism. The return type has nothing to do with the implementation details of a method, hence it is not considered during overloading.

2nd Part:

OVERLOADING	OVERRIDING

i) Method overloading is defining several methods in the same class, that accept different numbers and types of parameters. In this case, the actual method called is decided at compile-time, based on the number and types of arguments. For instance, the method System.out.println() is overloaded, so that you can pass ints as well as Strings, and it will call a different version of the method.
 	
ii) Overloading is an example of compile time polymorphism.
 	
Example:
`class b{`
`public void add(int i int j){`
`int n i+j;`
`System.out.println(n);`
`}`
`public void add(int i int j int k){`
`int n i+j+k;`
`System.out.println(n);`
`}`
`public void add(int i int j int k int l){`
`int n i+j+k+l;`
`System.out.println(n);`
`}`
`public static void main(String []aa){`
`b b1 new b();`
`b1.add(1 2 5);`
`}`
`}`
output: 8
	i) Method overriding is when a child class redefines the same method as a parent class, with the same parameters. For example, the standard Java class java.util.LinkedHashSet extends java.util.HashSet. The method add() is overridden in LinkedHashSet. If you have a variable that is of type HashSet, and you call its add() method, it will call the appropriate implementation of add(), based on whether it is a HashSet or a LinkedHashSet. This is called polymorphism.
	ii) Overriding is an example of run time polymorphism. The JVM does not know which version of method would be called until the type of reference will be passed to the reference variable. It is also called Dynamic Method Dispatch.
	Example:
	`class a{`
	`public void display(){`
	`System.out.println( it is a firest class method );`
	`}`
	`}`
	`class overiding extends a{`
	
	`public void display(){'`
	`System.out.println( it is a Second class method );`
	`}`
	`public static void main(String []anil){`
	`overiding obj new overiding();`
	
	`obj.display();`
	
	`}`
	`}`
	output: it is a Second class method

c) We can prevent overriding in Java by declaring the method as final.

2. Compare between method-overriding and method-overloading in JAVA. **[WBUT 2006, 2009]**

Answer:

Refer to Question No. 1(b) (2nd Part) of Long Answer Type Questions.

3. a) What is an Interface? Implement Interface in java with a simple code. **[WBUT 2007, 2009]**

b) Explain inheritance with its types. **[WBUT 2007, 2009, 2010]**

OR,

How many types of Inheritance in java? **[WBUT 2015]**

Answer:

a) An interface in Java supports multiple inheritance. Unlike classes, an interface does not provide any concrete implementation. Rather the implementation is provided by a class, which implements the interface.

To declare an interface we use the keyword interface. To have a class implementing an interface the keyword implements is used.

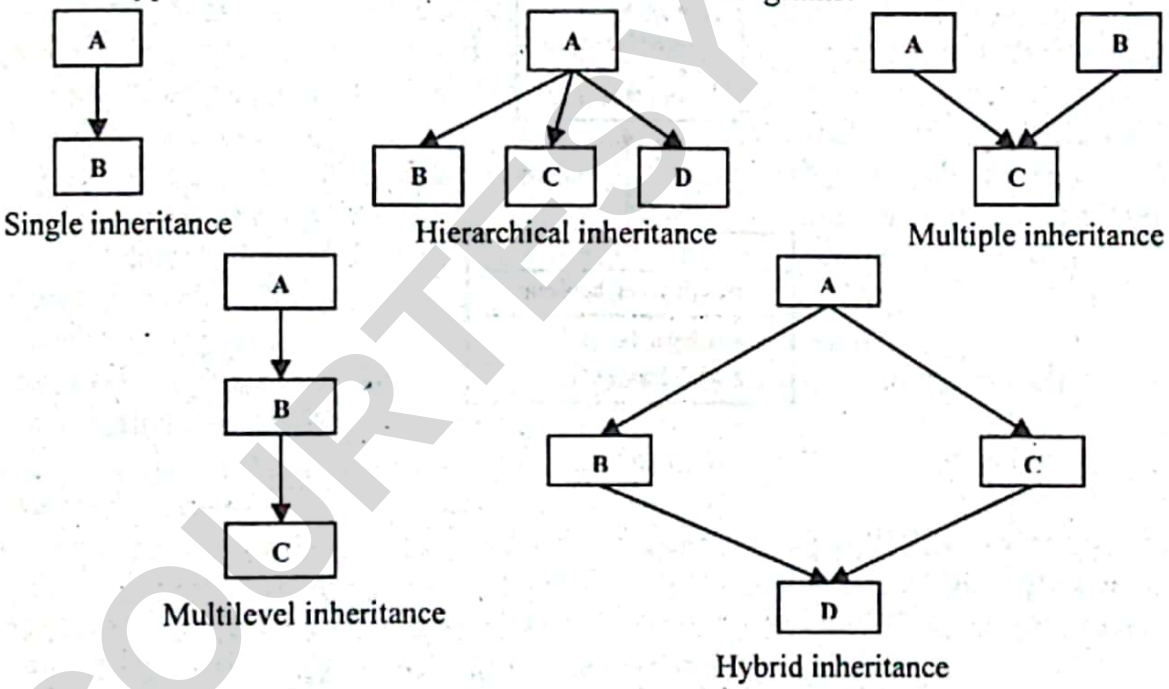
```
package  matrix.ad.edu;
interface interface1
{
void display1();
}
interface interface2 extends interface1
{
void display2();
}
class Extra
{
public void display3()
{
System.out.println("Display of the extra class");
}
}
public class MixedDataType extends Extra implements interface2
{
public void display2()
{
System.out.println("interface2 is implemented");
}
public void display1()
{
System.out.println("interface1 also needs to be implemented as
it" +"is extended byinterface2 ");
}
public static void main(String [] args)
{
MixedDataType mxdtyp = new MixedDataType();
mxdtyp.display1();
```

```
mxdtyp.display2();
mxdtyp.display3();
}
}
```

b) In object-oriented programming (OOP), inheritance is a way to compartmentalize and reuse code by creating collections of attributes and behaviors called objects which can be based on previously created objects.

The derived class inherits some or all of the traits or properties from the base class. A class can also inherit properties from more than one class or from more than one level. A derived class with only one base class is called *single inheritance* and one with several base classes is called *multiple inheritances*. *Multiple inheritances* allow us to combine the features of several existing classes as a starting point for defining new classes. Now, in case of *multilevel inheritance*, a class is derived from another derived class. Suppose a class is derived from one class, which in turn serves as a base class for another derived class. So, here the features inheriting continue level by level. On the other hand, the traits of one class may be inherited by more than one class. This process is known as *hierarchical inheritance*. There could be situations where we need to apply two or more types of inheritance to design a program. This type of inheritance is called *hybrid inheritance*.

Different types of inheritance are shown below with diagrams:



4. What is inheritance? How many types of inheritance java supports are there? Discuss it. Given a method that does not declare any exception, can I override that method in a subclass to throw an exception? [WBUT 2007, 2017]

Answer:

1st Part:

The term inheritance in object oriented programming implies a parent-child relationship between two or more than two classes. In Java, inheritance is achieved by using the **extends** keyword.

Illustration

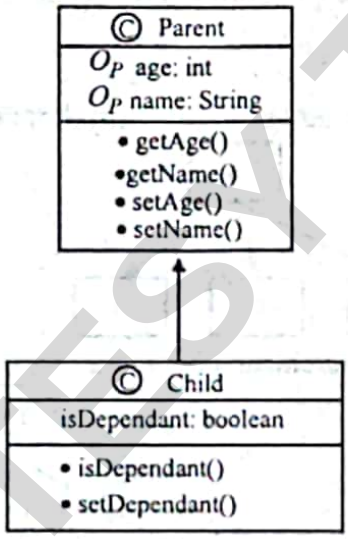
The diagram shown below illustrates a parent-child relationship between two classes.

The class Parent has two private instance variables and a set of public get set methods, which provides a way to access the variables outside the class.

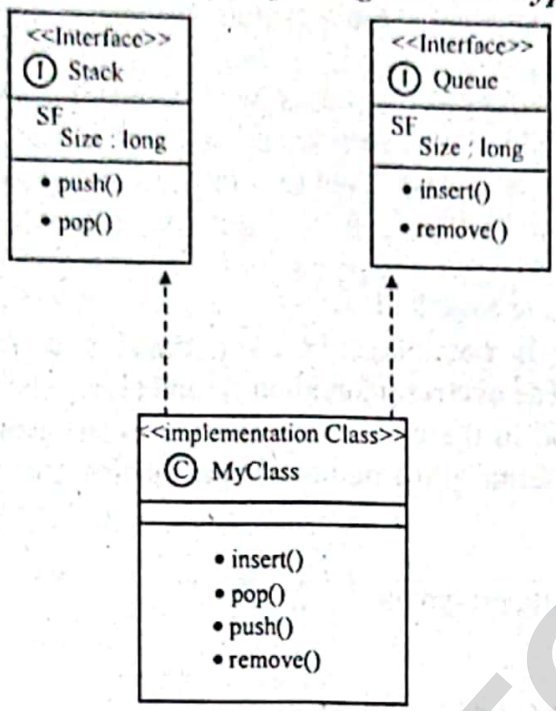
The class child inherits the properties of the parent except the private instance variables. That is, the child class has an automatic access to the get set methods of the parent class. Apart from this the child class can also define it's own attributes. Here the child class defines a boolean variable and the corresponding methods to access the variable.

This inheritance relationship is often referred to as

"There **IS-A** relationship between child and parent"



2nd Part: ***Refer to Question No. 3(b) of Long Answer Type Questions.***



3rd Part:

When you extend a class, the subclass inherits the methods of the super class. You can replace (override) these inherited methods by declaring them again in the subclass. The method in the subclass is called the *overriding* method and the method in the super class is called the *overridden* method. Java has a specific rule for declaration of these overriding methods. ***You cannot declare an overriding method to throw checked exceptions, which are not declared in the method in the super class.*** The overriding method can do any of the following things when an overridden method is throwing a checked exception:

The overriding method may decide *not to throw any exception.*

The overriding method may throw exception that is a *subclass* of the exception thrown by the overridden method.

The overriding method *cannot* throw exception that is a *super class* of the exception thrown by the overridden method.

These rules are simple to understand with an example. Figure 5.5 illustrates these rules. It shows a Base class with aMethod() which declares to throw an IOException. The Derived1, Derived2, IllegalOne, and IllegalTwo classes are all subclasses of the Base class. Each of these classes overrides aMethod() of Base class.

5. How two methods have same signature? Briefly describe the method overriding with example. Why Java does not support multiple inheritance? What does the super keyword do? **[WBUT 2008]**

Answer:
In Java two methods can have the same signature if there is a parent-child relationship between two classes.
The actual flavor of polymorphism comes with overriding. Only methods can be overridden. Usually the child class overrides methods of a parent class. By inheritance methods of a parent class is always available in the child class. The child class just changes the implementation details.

The basic rules for overriding a method are:

- A method overriding is possible only when there is a parent-child relationship between two classes. The overriding method should always be in the child class.
- The overriding method in the child class and the overridden method of the parent class must be same in terms of the number of parameters, the data type and the return type as well.

Let us consider the following programs

```
class Parent
{
public String format(String inputStr)
{

if("".equals(inputStr) || inputStr == null)
{
return " Cannot format Blank/Null String ";
}
return ("#"+inputStr+"#");
}
}
class Child extends Parent
{
/* performs a different operation by overriding the parent method */
public String format(String inputStr)
{
if("".equals(inputStr) || inputStr == null)
{
return " Cannot format Blank/Null String ";
}
return ("<"+inputStr.toUpperCase()+">");
}
}
```

Multiple inheritance often makes a class hierarchy very complex. If the number of classes increase in the system then the relationship becomes very difficult to maintain. As there will be lots of child classes inheriting different properties of their parent classes. One addition of an extra class in the hierarchy can cause severe distortion to the entire

hierarchy itself. If not properly handled the relationship between the classes can give erroneous results. That's why Java does not support multiple inheritance in class.
Java absolutely support multiple inheritance in terms of Interface. We can extend one class only to avoid ambiguity problem. In interface we have to define the functions. So we don't get any ambiguity. In c++ it is big problem with multiple inheritance but in JAVA this thing is improved by introducing Interfaces.
The super key word is used to refer the parent constructor or method in parent child relationship. In some cases the child class will refer to the parent constructor or method rather than overriding these methods explicitly in the class body. The only restriction super has is, that it should be first statement to be called in the child class method or constructor.

6. Correct the following code for overloading method: **[WBUT 2009, 2010]**

```
public class Figure
{
Public String draw (String s)
{
Return "Figure Drawn"
}
public void draw (string s) { }
public void draw (double f) { }
```

Answer:

```
public class Figure (
public String draw (String s) {
return "Figure Drawn";
}

public void draw(String s, double f) { }
// corrected overloading
public void draw (double f) { }
}
```

7. a) What are the features of the design pattern system? **[MODEL QUESTION]**

Answer:

The following are some of the features of the design patterns system:

- The design pattern provides the names to describe all the patterns.
- One of the best features is that it provides all the solutions to solve all the software design related problems.
- The Design patterns provide the dates and the author information.
- The design patterns provide the sample codes that are related to the problems.
- The design patterns provide the features of the references and the keywords that are mainly used for the searching.
- The design pattern provides the context for the solution.
- The design pattern provides the rationale behind the solution.

b) What are the Pros and cons of the design pattern system? [MODEL QUESTION]

Answer:

The following are some of the pros of the design patterns:

- The design patterns always provide the developer with a selection of the tried and tested solutions to work with.
- The design patterns are the language neutral and they can be applied to any of the languages that mainly supports the subject-oriented patterns.
- The design pattern provides the aid communication by the fact they are well documented and can also be researched if it is not the case.
- The design patterns always have the proven track record as they are always used and also help to reduce the technical risk of the project.
- The design patterns are really one of the most flexible patterns.
- They can be used practically and in any type of the domain name.
- The design patterns are really easy to adapt the predictable changes in the business needs.
- The design patterns are really very much easy for the unit testing and also validate individual components.
- The design pattern can provide the organization with a structure when the types of business requirements become very complicated.

The following are some of the cons of the design pattern:

- The design pattern will not lead to the reusing of direct coding.
- The design patterns are really very much simple
- The design pattern is really very much hard to understand for the beginners of software engineering.
- The design patterns can overload the memory and the processing, so this is really not appropriate for any such application such as the low-level system programming or for the certain embedded systems.

IMPLEMENTATION OF OO LANGUAGE FEATURES

Short Answer Type Questions

1. Explain "public static void main(String args [ ])" in brief.
[WBUT 2007, 2009, 2010, 2011]

Answer:

1. public - declares that the main method is publicly accessible to other classes
2. static - declares that the main method can be invoked without creating an instance of the class. The main( ) is loaded in static memory by java runtime loader at the time of class loading. And main() starts execution automatically.
3. void - declares that the main method does not return any value.
4. main - defines the name of the method
5. String[] args - defines a parameter to the main method which will contain any command line options passed by the user when invoking the program.

A small example:

```
class best
{
public static void [b]main(String[] s)
{
System.out.println("good");
}
}
```

2. What is JVM? **[WBUT 2007, 2009, 2010]**

OR,

Explain function of a JVM in brief. **[WBUT 2011]**

OR,

What do you mean by JVM? **[WBUT 2013]**

What do you mean by Java is a platform dependent language?
[WBUT 2007, 2009, 2010]

Answer:

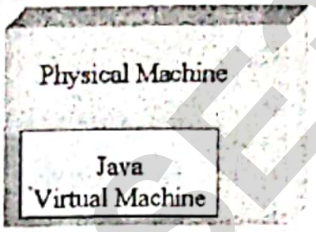
JAVA is not platform dependent language.
The 2nd Part question statement is wrong.

At the heart of the Java platform lies the Java Virtual Machine, or JVM. Most programming languages compile source code directly into machine code, suitable for execution on a particular microprocessor architecture. The difference with Java is that it uses bytecode - a special type of machine code.

Java bytecode executes on a special type of microprocessor. Strangely enough, there wasn't a hardware implementation of this microprocessor available when Java was first released. Instead, the processor architecture is emulated by what is known as a "*virtual*

machine". This virtual machine is an emulation of a real Java processor - a machine within a machine (Figure One). The only difference is that the virtual machine isn't running on a CPU - it is being emulated on the CPU of the host machine.

JVM emulation run on a physical CPU

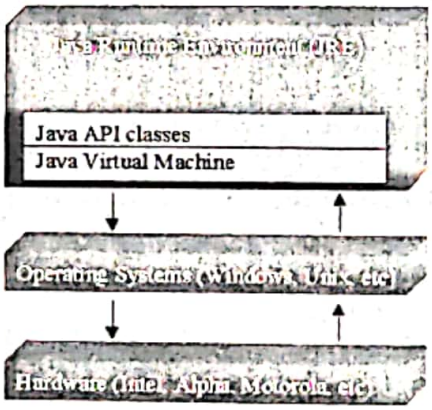


The Java Virtual Machine is responsible for interpreting Java bytecode, and translating this into actions or operating system calls. For example, a request to establish a socket connection to a remote machine will involve an operating system call. Different operating systems handle sockets in different ways - but the programmer doesn't need to worry about such details. It is the responsibility of the JVM to handle these translations, so that the operating system and CPU architecture on which Java software is running is completely irrelevant to the developer.

JVM handles translations

The Java Virtual Machine forms part of a large system, the Java Runtime Environment (JRE). Each operating system and CPU architecture requires a different JRE. The JRE comprises a set of base classes, which are an implementation of the base Java API, as well as a JVM. The portability of Java comes from implementations on a variety of CPUs and architectures. Without an available JRE for a given environment, it is impossible to run Java software.

The Java Virtual Machine provides a platform-independent way of executing code, by abstracting the differences between operating systems and CPU architectures. Java Runtime Environments are available for a wide variety of hardware and software combinations, making Java a very portable language. Programmers can concentrate on writing software, without having to be concerned with how or where it will run. The idea of virtual machines is nothing new, but Java is the most widely used virtual machine used today. Thanks to the JVM, the dream of Write Once-Run Anywhere (WORA) software has become a reality.



When Java Code is compiled a byte code is generated which is independent of the system. This byte code is fed to the JVM (Java Virtual Machine) which resides in the system. Since every system has its own JVM, it doesn't matter where you compile the source code. The byte code generated by the compiler can be interpreted by any JVM of any machine. Hence it is called Platform independent Language.

Java's byte codes are designed to be read and interpreted in exactly same manner on any computer hardware or operating system that supports Java Runtime Environment.

3. Write down a program to implement Command Line arguments.

[WBUT 2011, 2012]

Answer:

```
class CmndLineArguments {
public static void main(String[] args) {
int length = args.length;
if (length <= 0) {
System.out.println("You need to enter some arguments.");
}
for (int i = 0; i < length; i++) {
System.out.println(args[i]);
}
}
}
```

4. Discuss the garbage collection procedure in java. **[WBUT 2013, 2015]**

Answer:

i) Garbage collection is a mechanism provided by Java Virtual Machine to reclaim heap space from objects which are eligible for Garbage collection.

ii) Garbage collection relieves java programmer from memory management which is essential part of C++ programming and gives more time to focus on business logic.

iii) Garbage Collection in Java is carried by a daemon thread called Garbage Collector.

iv) Before removing an object from memory Garbage collection thread invokes finalize () method of that object and gives an opportunity to perform any sort of cleanup required.

v) We as Java programmer cannot force Garbage collection in Java; it will only trigger if JVM thinks it needs a garbage collection based on Java heap size.

5. What is JVM? Explain the process of compilation and interpretation in Java. **[WBUT 2014]**

Answer:

1st Part: ***Refer to Question No. 2 of Short Answer Type Questions.***

2nd Part:

A java code is compiled into a byte code which is platform independent. This byte code is then interpreted by the java interpreter to get the desired output of the java program. All java interpreters can interpret a class file generated by a Java compiler under a particular operating system. A java code compiled with windows based Java compiler can run in a Linux operating system environment without re-compilation. Hence Java is a compiled as well as an interpreted language.

So, Java is a compiled programming language, but rather than compile straight to executable machine code, it compiles to an intermediate binary form called JVM byte code. The byte code is then compiled and/or interpreted to run the program.

6. What is Model View Controller? Explain the advantages of MVC pattern. **[MODEL QUESTION]**

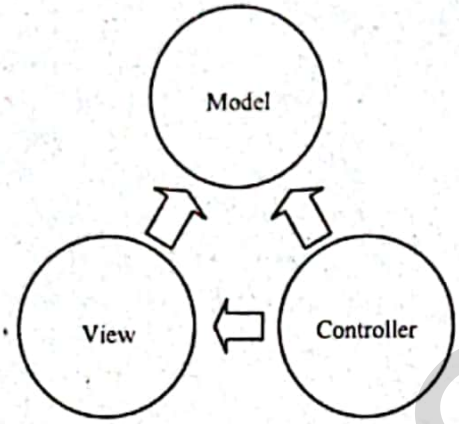
Answer:

Model–view–controller (MVC) is a software architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts, so as to separate internal representation of information from the way that information is presented to or accepted from the user.

MVC is a framework for building web applications using an MVC (Model View Controller) design:

- The Model represents the application core (for instance a list of database records).
- The View displays the data (the database records).
- The Controller handles the input (to the database records).

The MVC model also provides full control over HTML, CSS, and JavaScript.



The MVC model defines web applications with 3 logic layers,

- The business layer (Model logic)
- The display layer (View logic)
- The input control (Controller logic)

The Model is the part of the application that handles the logic for the application data. Often model objects retrieve data (and store data) from a database.

The View is the part of the application that handles the display of the data. Most often the views are created from the model data.

The Controller is the part of the application that handles user interaction. Typically controllers read data from a view, control user input, and send input data to the model.

The MVC separation helps you manage complex applications because you can focus on one aspect a time. For example, you can focus on the view without depending on the business logic. It also makes it easier to test an application.

The MVC separation also simplifies group development. Different developers can work on the view, the controller logic, and the business logic in parallel.

The advantages of Model View controller patterns are;

- Multiple developers can work simultaneously on the model, controller and views.
- MVC enables logical grouping of related actions on a controller together. The views for a specific model are also grouped together.
- Models can have multiple views.

7. Explain MVC application life cycle. **[MODEL QUESTION]**

Answer:

Any web application has two main execution steps, first understanding the request and depending on the type of the request sending out an appropriate response. MVC

application life cycle is not different it has two main phases, first creating the request object and second sending our response to the browser.

Creating the request object:
The request object creation has four major steps. The following is a detailed explanation of the same.

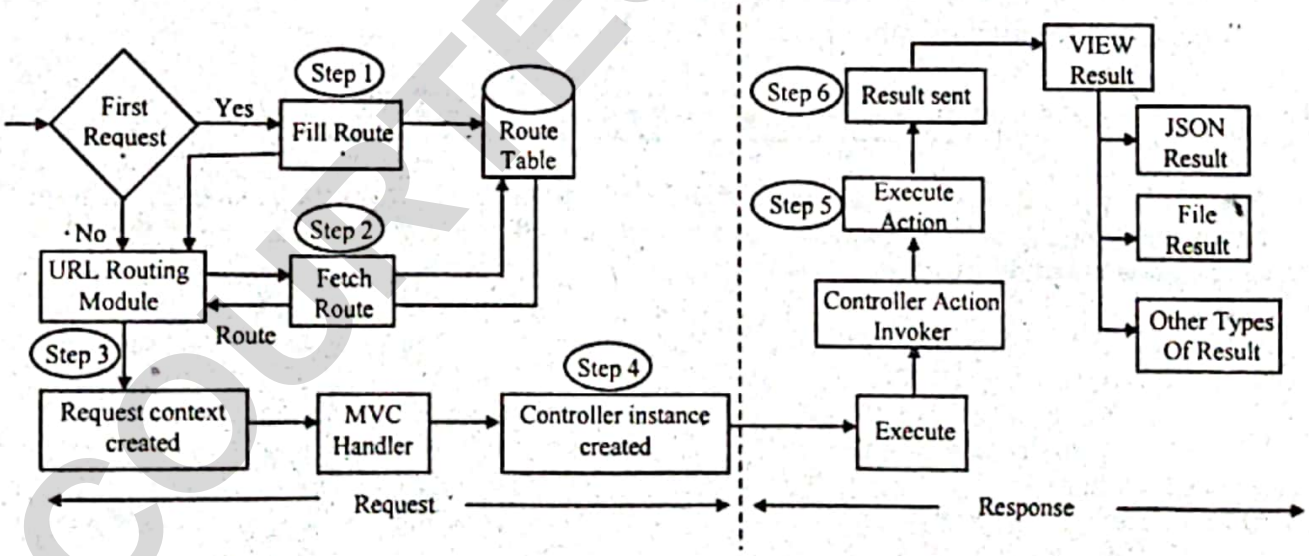
Step 1 - Fill route
MVC requests are mapped to route tables which in turn specify which controller and action to be invoked. So if the request is the first request the first thing is to fill the rout table with routes collection. This filling of the route table happens the global.asax file.

Step 2 - Fetch route
Depending on the URL sent "UrlRoutingModule" searches the route table to create "RouteData" object which has the details of which controller and action to invoke.

Step 3 - Request context created
The "RouteData" object is used to create the "RequestContext" object.

Step 4 - Controller instance created
This request object is sent to "MvcHandler" instance to create the controller class instance. Once the controller class object is created it calls the "Execute" method of the controller class.

Creating a Response object
This phase has two steps executing the action and finally sending the response as a result to the view.



8. What are the filters in MVC? **[MODEL QUESTION]**

Answer:

In MVC, controllers define action methods and these action methods generally have a one-to-one relationship with UI controls such as clicking a button or a link, etc.

Types of Filters

ASP.NET MVC framework supports the following action filters,

- **Action Filters:** Action filters are used to implement logic that gets executed before and after a controller action executes. We will look at Action Filters in detail in this chapter.
- **Authorization Filters:** Authorization filters are used to implement authentication and authorization for controller actions.
- **Result Filters:** Result filters contain logic that is executed before and after a view result is executed. For example, you might want to modify a view result right before the view is rendered to the browser.
- **Exception Filters:** Exception filters are the last type of filter to run. You can use an exception filter to handle errors raised by either your controller actions or controller action results. You can also use exception filters to log errors.

Action filters are one of the most commonly used filters to perform additional data processing, or manipulating the return values or canceling the execution of an action or modifying the view structure at run time.

9. Explain the disadvantages of MVC pattern. **[MODEL QUESTION]**

Answer:

- The framework navigation can be complex because it introduces new layers of abstraction and requires users to adapt to the decomposition criteria of MVC.
- Knowledge on multiple technologies becomes the norm. Developers using MVC need to be skilled in multiple technologies.

Long Answer Type Questions

1. Write a program to implement a dynamic stack. Each stack is constructed with an initial length. If this length is exceeded, i.e., if more room is needed then the size of the stack is doubled. **[WBUT 2007]**

Answer:

```
import java.util.Iterator;
import java.util.NoSuchElementException;
public class DoublingStack<Item> implements Iterable<Item> {
private Item[] a;
private int N = 0;
public DoublingStack() {
a = (Item[]) new Object[2];
}
public boolean isEmpty() { return N == 0; }
public int size()        { return N;      }
private void resize(int capacity) {
```

```
assert(capacity >= N);
Item[] temp = (Item[]) new Object[capacity];
for (int i = 0; i < N; i++)
temp[i] = a[i];
a = temp;
}
public void push(Item item) {
if (N == a.length) resize(2*a.length);
a[N++] = item;
}
public Item pop() {
if (isEmpty()) { throw new RuntimeException("Stack underflow
error"); }
Item item = a[N-1];
a[N-1] = null;
N--;
// shrink size of array if necessary
if (N > 0 && N == a.length/4) resize(a.length/2);
return item;
}
public String toString() {
String s = "[ ";
for (int i = 0; i < N; i++)
s += a[i] + " ";
s += "]";
return s;
}
public Iterator<Item> iterator()  { return new
ArrayStackIterator();  }
// an iterator, doesn't implement remove() since it's optional
private class ArrayStackIterator implements Iterator<Item> {
private int i = N;
public boolean hasNext()  { return i > 0;
}
public void remove()      { throw new
UnsupportedOperationException();  }
public Item next() {
if (!hasNext()) throw new NoSuchElementException();
return a[--i];
}
}
public static void main(String[] args) {
DoublingStack<String> stack = new DoublingStack<String>();
stack.push("Hello");
stack.push("World");
stack.push("how");
stack.push("are");
stack.push("you");
for (String s : stack)
System.out.println(s);
System.out.println();
```

```
while (!stack.isEmpty())
System.out.println(stack.pop());
}
}
```

2. How command line argument is written? Explain through a program. How an array is declared in Java? Write a program to ascending sort an array and display it. **[WBUT 2014]**

Answer:

1st Part: ***Refer to Question No. 3 of Short Answer Type Questions.***

2nd Part:

This is how an array in java can be declared:

ArrayDataType[] ArrayName;

OR

ArrayDataType ArrayName[];

Where ArrayDataType defines the data type of array element like int, double etc.

ArrayName is the name of array.

We can also create/ Instantiate an array by using new keyword as follows:

In java, initialize an array can be done by using new keyword as well:

int arrayName = new int[10];

As we can see array initialization is done, where [10] specifies that array length is ten or array can contain ten elements.

Assigning values to arrays

This is how you can assign value to arrays:

arrayName[0] = 10;

Alternatively you can also assign values as follows:

int[]ArrList = {1, 2, 3, 4,5};

The above array is five elements length.

Example of int array java

In this example we will declare, initialize and access array items. The for loop is used ("foreach") to display array items. The array type is int. See example by clicking the link below:

```
public class array_ex {
public static void main(String []args)
{
int arrex[] = {10,20,30};                    //Declaring and
initializing an array of three elements
for (int i=0;i&lt;arrex.length;i++){
System.out.println(arrex[i]);
} } }
```

The output will be:

```
10
20
30
```

3rd Part:

Java program for bubble sort in ascending order

```
public class BubbleSortAscendingOrderDemo
{
public static void main(String a[])
{
//Numbers which need to be sorted
int numbers[] = {23,5,23,1,7,12,3,34,0};

//Displaying the numbers before sorting
System.out.print("Before sorting, numbers are ");
for(int i = 0; i < numbers.length; i++)
{
System.out.print(numbers[i]+" ");
}
System.out.println();
//Sorting in ascending order using bubble sort
bubbleSortInAscendingOrder(numbers);
//Displaying the numbers after sorting
System.out.print("Before sorting, numbers are ");
for(int i = 0; i < numbers.length; i++)
{
System.out.print(numbers[i]+" ");
}
}
//This method sorts the input array in asecnding order
public static void bubbleSortInAscendingOrder( int numbers[])
{
int temp;
for(int i = 0; i < numbers.length; i++)
{
for(int j = 1; j < (numbers.length -i); j++)
{
//if numbers[j-1] > numbers[j], swap the elements
if(numbers[j-1] > numbers[j])
{
temp = numbers[j-1];
numbers[j-1]=numbers[j];
numbers[j]=temp;
}}}}}
```

Output

```
Before sorting, numbers are 23,5,23,1,7,12,3,34,0
Before sorting, numbers are 0,1,3,5,7,12,23,23,34
```

3. Write short notes on the following:

a) Templates **[WBUT 2008]**

b) Virtual Method Table **[WBUT 2014]**

c) Command Line Argument **[WBUT 2015, 2016]**

Answer:

a) Templates:

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

b) Virtual Table:

Every class that uses virtual functions (or is derived from a class that uses virtual functions) is given it's own virtual table as a secret data member.
This table is set up by the compiler at compile time.
A virtual table contains one entry as a function pointer for each virtual function that can be called by objects of the class.
Virtual table stores NULL pointer to pure virtual functions in ABC.
Virtual Table is created even for classes that have virtual base classes. In this case, the vtable has pointer to the shared instance of the base class along with the pointers to the classe's virtual functions if any.

_vptr:

This vtable pointer or _vptr, is a hidden pointer added by the Compiler to the base class.
And this pointer is pointing to the virtual table of that particular class.
This _vptr is inherited to all the derived classes.
Each object of a class with virtual functions transparently stores this _vptr.
Call to a virtual function by an object is resolved by following this hidden _vptr.
Here is a simple example with a vtable representation.
Here we have 3 classes Base, D1 and D2. Where D1 and D2 are derived from class Base.

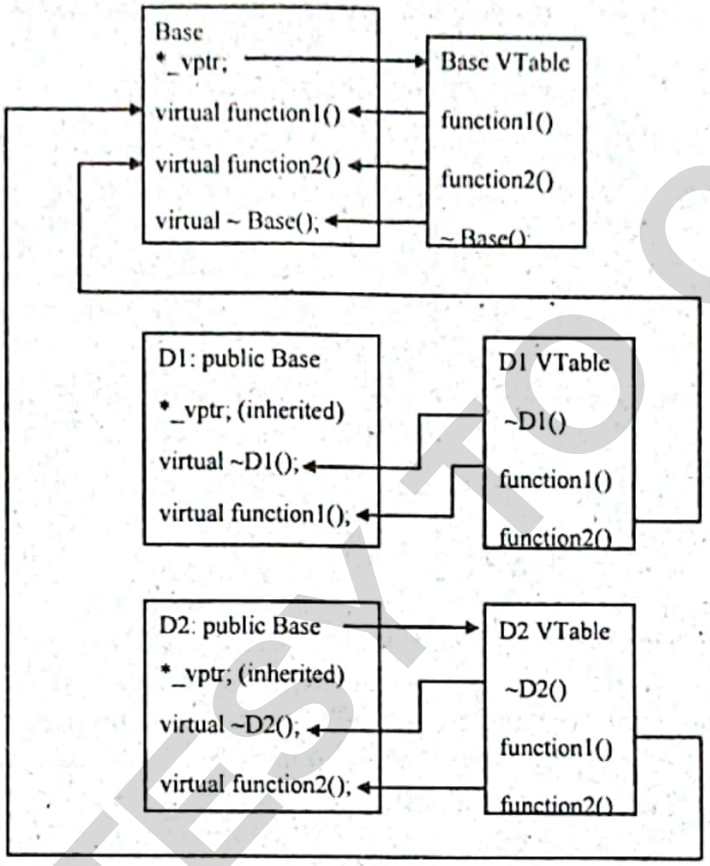
```
#include<iostream.h>
class Base
{
 public:
virtual void function1() {cout<<"Base :: function1()\n";};
virtual void function2() {cout<<"Base :: function2()\n";};
virtual ~Base(){};
};
class D1: public Base
{
public:
~D1(){};
virtual void function1() { cout <<"D1 :: function1()\n";};
};
class D2: public Base
{
public:
~D2(){};
virtual void function2() { cout<< "D2 :: function2\n";};
};
int main()
{
D1 *d = new D1;;
Base *b = d;
b->function1();
```

```
b->function2();
delete (b);
return (0);
}
```

Output:

```
D1 :: function1()
Base :: function2()
```

Here is a pictorial representation of Virtual Table and _vptr for the above code:



Explanation:

Here in function main b pointer gets assigned to D1's _vptr and now starts pointing to D1's vtable. Then calling to a function1(), makes it's _vptr straightway calls D1's vtablefunction1() and so in turn calls D1's method i.e. function1() as D1 has it's ownfunction1() defined it's class.

Where as pointer b calling to a function2(), makes it's _vptr points to D1's vatble which in-turn pointing to Base class's vtable function2 () as shown in the diagram (as D1 class does not have it's own definition or function2()).

So, now calling delete on pointer b follows the _vptr - which is pointing to D1's vtable calls it's own class's destructor i.e. D1 class's destructor and then calls the destructor of Base class - this as part of when derived object gets deleted it turn deletes it's embedded base object. That's why we must always make Base class's destructor as virtual if it has any virtual functions in it.

c) Command Line Argument:

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to main(). A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy– they are stored as strings array passed to the args parameter of main() . The first command-line argument is stored at args[0], the second at args[1], and so on. For example, the following program displays all of the command-line arguments that it is called with:

```
// Display all command-line arguments.
class CommandLine {
public static void main(String args[])  {
for(int i=0; i<args.length; i++)
System.out.println("args[" + I + "): " +
args[i]);
}
}
```

4. Explain the major concept of java memory management. [MODEL QUESTION]

Answer:

The major concepts in Java Memory Management :

- JVM Memory Structure
- Working of Garbage Collector

Java Memory Structure:

JVM defines various run time data area which are used during execution of a program. Some of the areas are created by the JVM whereas some are created by the threads that are used in a program. However, the memory area created by JVM is destroyed only when the JVM exits. The data areas of thread are created during instantiation and destroyed when the thread exits.

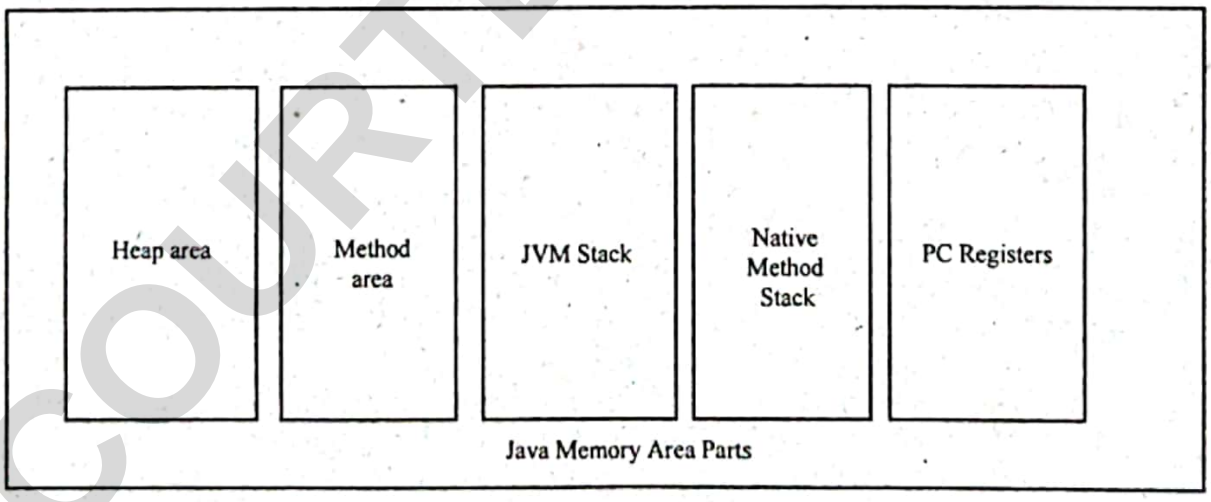


Fig: JVM Memory Area

The parts of memory area in detailed:

Heap:

- It is a shared runtime data area and stores the actual object in a memory. It is instantiated during the virtual machine startup.
- This memory is allocated for all class instances and array. Heap can be of fixed or dynamic size depending upon the system's configuration.
- JVM provides the user control to initialize or vary the size of heap as per the requirement. When a new keyword is used, object is assigned a space in heap, but the reference of the same exists onto the stack.
- There exists one and only one heap for a running JVM process.

Scanner sc = new Scanner(System.in);

The above statement creates the object of Scanner class which gets allocated to heap whereas the reference 'sc' gets pushed to the stack.

Method Area:

- It is a logical part of the heap area and is created on virtual machine startup.
- This memory is allocated for class structures, method data and constructor field data, and also for interfaces or special method used in class. Heap can be of fixed or dynamic size depending upon the system's configuration.
- Can be of a fixed size or expanded as required by the computation. Needs not to be contiguous.

JVM Stacks:

- A stack is created at the same time when a thread is created and is used to store data and partial results which will be needed while returning value for method and performing dynamic linking.
- Stacks can either be of fixed or dynamic size. The size of a stack can be chosen independently when it is created.
- The memory for stack needs not to be contiguous.

Native method Stacks:

Also called as C stacks, native method stacks are not written in Java language. This memory is allocated for each thread when its created. And it can be of fixed or dynamic nature.

Program counter (PC) registers:

Each JVM thread which carries out the task of a specific method has a program counter register associated with it. The non native method has a PC which stores the address of the available JVM instruction whereas in a native method, the value of program counter is undefined. PC register is capable of storing the return address or a native pointer on some specific platform.

5. Explain the working principle of garbage collector. [MODEL QUESTION]

Answer:

Working of a Garbage Collector:

- JVM triggers this process and as per the JVM garbage collection process is done or else withheld. It reduces the burden of programmer by automatically performing the allocation or deallocation of memory.
- Garbage collection process causes the rest of the processes or threads to be paused and thus is costly in nature. This problem is unacceptable for the client but can be eliminated by applying several garbage collector based algorithms. This process of applying algorithm is often termed as **Garbage Collector tuning** and is important for improving the performance of a program.
- Another solution is the generational garbage collectors that adds an age field to the objects that are assigned a memory. As more and more objects are created, the list of garbage grows thereby increasing the garbage collection time. On the basis of how many clock cycles the objects have survived, objects are grouped and are allocated an 'age' accordingly. This way the garbage collection work gets distributed.
- In the current scenario, all garbage collectors are generational, and hence, optimal.

GENERIC TYPES AND COLLECTIONS GUIS

Short Answer Type Questions

1. What is java collections framework? List out some benefits of collections framework. **[MODEL QUESTION]**

Answer:

Collections are used in every programming language and initial java release contained few classes for collections: **Vector, Stack, Hash table, Array.** But Java 1.2 came up with Collections Framework that group all the collections interfaces, implementations and algorithms.

Java Collections have come through a long way with the usage of Generics and Concurrent Collection classes for thread-safe operations. It also includes blocking interfaces and their implementations in java concurrent package.

Some of the benefits of collections framework are:

1. Reduced development effort by using core collection classes rather than implementing our own collection classes.
2. Code quality is enhanced with the use of well tested collections framework classes.
3. Reduced effort for code maintenance by using collection classes shipped with JDK.
4. Reusability and Interoperability.

2. What are the basic interfaces of Java collections framework? **[MODEL QUESTION]**

Answer:

Collection is the root of the collection hierarchy. A collection represents a group of objects known as its elements. The Java platform doesn't provide any direct implementations of this interface.

Set is a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the deck of cards.

List is an ordered collection and can contain duplicate elements. You can access any element from its index. The list is more like an array with dynamic length.

A Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value.

Some other interfaces are Queue, Dequeue, Iterator, SortedSet, SortedMap and ListIterator.

3. Why Map interface doesn't extend collection interfaces? [MODEL QUESTION]

Answer:

Although Map interface and its implementations are part of the Collections Framework, Map is not collections and collections are not Map. Hence it doesn't make sense for Map to extend collections or vice- versa.

If Map extends Collection interface, then the map contains key-value pairs and it provides methods to retrieve the list of Keys or values as Collection but it doesn't fit into the "group of elements" paradigm.

4. What is Iterator? Differentiate between Iterator and enumeration. [MODEL QUESTION]

Answer:

1st Part:

The Iterator interface provides methods to iterate over any Collection. We can get iterator instance from a Collection using *iterator()* method. Iterator takes the place of Enumeration in the Java Collections Framework. Iterators allow the caller to remove elements from the underlying collection during the iteration. Java Collection iterator provides a generic way for traversal through the elements of a collection and implements Iterator Design Pattern.

2nd Part:

Enumeration is twice as fast as Iterator and uses very little memory. Enumeration is very basic and fits basic needs. But the Iterator is much safer as compared to Enumeration because it always denies other threads to modify the collection object which is being iterated by it.

Iterator takes the place of Enumeration in the Java Collections Framework. Iterators allow the caller to remove elements from the underlying collection that is not possible with Enumeration. Iterator method names have been improved to make its functionality clear.

5. What is Collections class? What is blocking queue? [MODEL QUESTION]

Answer:

1st Part:

java.util.Collections is a utility class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

This class contains methods for collection framework algorithms, such as binary search, sorting, shuffling, reverse, etc.

2nd Part:

java.util.concurrent.BlockingQueue is a Queue that supports operations that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element.

BlockingQueue interface is part of the java collections framework and it's primarily used for implementing the producer-consumer problem. We don't need to worry about waiting for the space to be available for producer or object to be available for consumers in BlockingQueue as it's handled by implementation classes of BlockingQueue.

Java provides several BlockingQueue implementations such as ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue, etc.

6. Write short notes on: java generics. **[MODEL QUESTION]**

Answer:

- Java Generics, introduced in Java 5, provide stronger type safety.
- Generics allow *types* to be passed as *parameters* to class, interface, and method declarations. For example:

```
List<Employee> emps = new ArrayList<Employee>();
```

- The **<Employee>** in this example is a *type parameter*.
  - With the type parameter, the *compiler* ensures that we use the collection with objects of a compatible type only.
  - Another benefit is that we won't need to cast the objects we get from the collection:

```
Employee e = emps.get(0);
```

  - Object type errors are now detected at compile time, rather than throwing casting exceptions at runtime.

Long Answer Type Questions

1. What are the similarities and differences between ArrayList and Vector? **[MODEL QUESTION]**

Answer:

The similarities between ArrayList and Vector are:

1. Both are index based and backed up by an array internally.
2. Both maintains the order of insertion and we can get the elements in the order of insertion.
3. The iterator implementations of ArrayList and Vector both are fail-fast by design.
4. ArrayList and Vector both allows null values and random access to element using index number.

The differences between ArrayList and Vector are:

1. Vector is synchronized whereas ArrayList is not synchronized. However if you are looking for modification of list while iterating, you should use CopyOnWriteArrayList.
2. ArrayList is faster than Vector because it doesn't have any overhead because of synchronization.
3. ArrayList is more versatile because we can get synchronized list or read-only list from it easily using Collections utility class.