Q1) What do you mean by JVM, JRE and JDK?

A1) JVM (Java Virtual Machine)
JVM is an abstract computing machine that provides a runtime environment for executing Java bytecode. It is responsible for executing compiled Java programs.

- JRE (Java Runtime Environment)
JRE is a set of tools & libraries that allows Java applications to be executed. It includes JVM, necessary libraries, and other components required to run Java applications.

- JDK (Java Development Kit) JDK is a software development environment for building, debugging and monitoring Java applications. It includes JRE, development tools and libraries.

Concept :-

- JVM :- It provides a virtualized execution environment for JAVA programs. It abstracts hardware and operating system details, allowing Java applications to be

platform - independent.

- JRE :- It is the environment required to run compiled Java programs. It includes the JVM & other components that provides the necessary runtime support.

- JDK :- It is a comprehensive development package that includes tools for writting compiling & debugging Java programs. It also includes JRE for running Java application

Example :-

JVM : When you run a Java program using the 'java' command, it's the JVM that interprets the bytecode & executes the program.

JRE :- When you install Java on your system, you're actually installing the JRE. This allows you to run Java applications.

JDK : If you're a developer, you use the JDK to write, compile, and run Java programs. It includes tools like 'javac' (java compiler) & 'java' (Java runtime).

→ Real-life example for correlation :-

- Imagine a car (Java Program) : The JVM is like the engine of the car, responsible for executing instructions (or driving the car). The JRE is like the car itself, providing all the necessary components for the engine to work (e.g. fuel tank, exhaust system). The JDK is like a car manufacturing plant, providing all the tools needed to build & maintain cars.

→ Advantages :-

- JVM :- Allows Java programs to be platform-independent. Provides memory management & garbage collection.

- JRE :- Enables users to run Java applications without needing to install development tools.

- JDK :- Offers a complete set of tools for Java development including compilers, debuggers & profilers.

Limitations :-

- JVM :- Performance overhead due to bytecode interpretation. Limited access to system resources compared to native application.

- JRE :- Only provides runtime support ; does not include development tools.

- JDK :- Takes up more disk space due to inclusion of development tools.

Q2) Is JRE platform dependent or independent ?

→ JRE Platform Dependency :-

Defination :-

- JRE : JRE is platform - independent. It provides the necessary runtime environment for Java appln to execute.

Concept :-

- JRE :- It's designed to be platform - independent, meaning you can install and run the same JRE on different operating systems as long as they support Java.

Example :-

JRE :- If you have a JRE installed on a Windows system, you can run Java appln's on it. The same JRE can be installed on a

Mac or a Linux system to run the same Java application.

→ Real-life example correlation :-

- Electricity :- Consider electricity as JRE. It can power different type of appliances regardless of there model. Similarly, JRE can run on various platform.

→ Advantage :-

→ Platform independency

→ compatibility

→ Limitations :-

Depending on JVM :- While JRE itself is platform independent, it relies on a platform specific JVM to execute bytecode.

Q3. Ultimate Base class in Java class?

Ultimate Base class in Java class Hierarchy :-

Defination :-

Root class :- Java.lang.object.

Every class in Java implicitly extend this class.

Concept :- It is at the top of class hierarchy & provides common methods that all java object inherits.

→ Real life :- Living organism :-
Every living org. has some common characteristics, ability to grow, reproduce & respond to stimuli.

Methods of java.lang.object

5 Final

```
public final native Class<?> getClass()
public final native void notify()
public final native void notifyAll()
public final void wait() throws InterruptedException
public final native void wait(long timeout) throws InterruptedException
public final void wait(long timeout, int nanos) throws InterruptedException
```

6 Non Final

```
public boolean equals(Object obj)
public int hashCode()
public String toString()
protected Object clone() throws CloneNotSupportedException
```

Adv :-
- Uniformity
- Method overridding
- No Customization of fundamental method

Q4) Reference types in Java :-

Reference types in Java :-

→ Data types that store references to objects. They do not directly contains the data but instead point to the location of data.

Concept :- They allow us to work with complex data structures & objects in java. They include classes, interfaces, arrays enumerations & annotations.

Ex: Myclass obj = new Myclass();
↳ reference variable

→ Real life example :-
Library card : doesn't contains actual books but only holds reference to books.

→ Adv :-
- Memory efficiency
- Dynamic M/M mgmt

→ Limitations :-
- Indirect access
- Potential Null reference

Q5) Narrowing & Widening

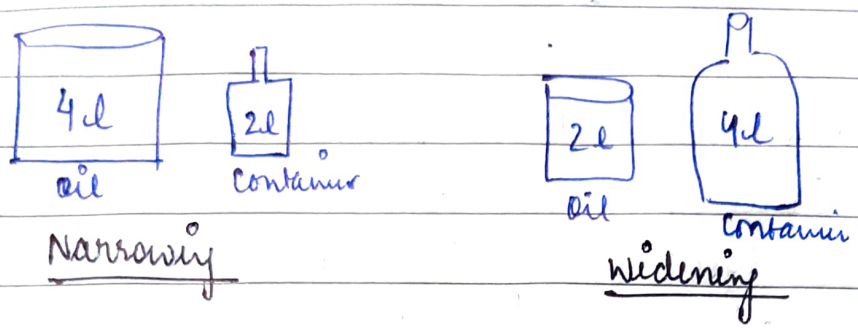
Defination :-
Narrowing :- Type casting :- Converting a data type of higher size to lower size.

Widening :- (Automatic type conversion) : lower size data type to higher level.

Concept :- Narrowing :- loss of data / precision as destination type might not be able to represent values of source type.

Widening :- Safe, No loss of information

Real life :- Narrowing :- Trying to keep 4l oil in a 2l container.
widening :- Keep 2l oil in 4l container



Adv :- Narrowing : Allows you to explicitly specify type conv.
widening : automatic

Limitation N : Data loss / possible runtime error.
W : No data loss / Potential loss of precision

Q6) Printing "Hello CDAC" W/O semicolon :

```
public class Main {
    public static void main (String [] args) {
        if (System.out.printf("Hello CDAC") != null) {
    }
}
```

Printf returns 'PrintStream' object, which is not null. ∴ cond" = True.

→ Adv :- Compact Code.
→ Limitation :- Unconventional & Confusing.

Q7) Java appⁿ w/o main funcⁿ ?

→ Yes, you can create Java applⁿ w/o a 'main' funcⁿ using a static block.

```
public class Main {
    static {
        System.out.println("Hello CDAC");
        System.exit(0);
    }
}
```

Explain :- Above, static block is used.
It will be executed when class is loaded, w/o need of main method.

Real life ex :- Automated greetig msg :-
consider a system that displays greetig msg when it starts. w/o any i/p from user.

Adv :- Automatic initialization

Limitation :- Limited usefulness.

Q8) Calling Main method in static block :-

Calling main method in a static block is possible but it's not recommended.
It can lead to issues like infinite

recursion & is bad practice.

```
public class Main {
    static {
        main (New String [] {});
    }
public static void main (String[] args) {
    Sopln ("Hello COAC");
}
```

Explain :- In this example, main method is called from static block. This can lead to infinite loop of method calls (undesirable)

Adv :- NONE → Not recommended.

Limitation :- infinite recursion :- Can lead to infinite loop of method calls.

Q9) System.out.println explain?

System :- class in Java's core library.

out :- static member of the 'System' class of type 'PrintStream'

println :- method of 'PrintStream' class is used to print a line of text.

Concept :- print a line of text to the standard output in Java.

Adv :- Convenience, Flexibility.

Limitations :- Limited to text output.

Q10) Passing object by Reference :-

In Java, all objects are passed by reference. When an object is passed to a method, the reference to the object is passed, not the copy of object itself.

```
class Person {
    String name;
}

public class Main {
    psvm modifyName ( Person p ) {
        p.name = "Anuja";
    }
    psvm ( String[] args ) {
        Person person = new Person();
        person.name = "Anuja";
        modifyName ( person );
        sopln ( person.name );        // "Anuja"
    }
}
```

Object →
Name = "Anuja" →
Method →

Explain :- a 'Person' object is created with name "Anuja". The modify name method takes 'Person' object as a parameter & changes its 'name' attribute to "Anuja".

Adv :- • Memory ↓ • Consisting with OOP

Limitation :- Potential for unintended side effect change in one part of code can lead to unexpected change in other.

Q 11) Constructor chaining :-

Def :- Process of calling one constructor with another constructor in same class or in parent class.

Concept :-
We use 'this()' or 'super()'

Example :-

Parent class

```
class Parent {
    int x;
    Parent(int x) {                    // Parameterized cstor
        this.x = x;
    }
}
```

Child class

```
class child extends Parent {
    int y;
    child(int x, int y) {
        super(x);
        this.y = y;
    }
}
```

Explain :- In this example, the 'child' class constructor calls the constructor of parent class 'Parent' using 'super(x)'.

Real life ex :- As constructor in a class calls constructor of other class to co-ordinate in order to initialize object correctly. Same a contractor to build a building co-ordinates with the team to construct house properly.

Adv :- • Code Reusability • Ensure proper initialization.

Limitation :- Confusion.

Q12) Rules to overload method in subclass :

In Java, when overloading methods in a subclass, you must follows these rules :-

(1) Parameter list :- The parameter list of the overloaded method must differ from the original method.

(2) Return Type :- The return type can be the same or a subclass of the original method's return type.

(3) Access modifier :- It can have the same or wider access modifier (e.g., if the original method is 'protected', the overload method can be 'protected' or 'public', but not 'private'.)

(4) Exception :- The overloaded method can declare exceptions from the same hierarchy or subclasses of those declared by the original method.

Q13) Difference b/w 'finalize' & 'dispose' :-

'Finalize' :

- Definition :- 'Finalize' is a method in the 'Object' class. It's called by the garbage collector before an object is a garbage collected.

- Concept : It provides an opportunity for an object to clean up resources (eg. close file) before it is removed from memory.

- Advantage : Can be used for resource clean up, but it's not recommended due to its unpredictable nature.

- Limitations :- Not guaranteed to be called, and it's generally recommended to use explicit cleanup mechanisms (like 'try-with-resources' or manual resource management).

→ 'Dispose' :-

- Definition :- 'dispose' is a method used in GUI toolkits like Swing & AWT. It's used to release resources held by a graphical component.
- Concept :- It's used to release resources like window handles, fonts & other system resources.
- Advantage :- Ensures that system resource are promptly released, improving appl^n performance.
- Limitations Specific to GUI components & not applicable in non GUI context.

Q14) Difference btw" :-

① 'final' :-

Def :- 'final' is keyword in java. It can be applied to classes, methods & variables.

Concept :- when applied to class, it means the class cannot be subclassed. when applied to a variable, it means that variable cannot be reassigned.

Adv :- Provides a way to create constants, prevent method overriding, & Enforces immutability.

Limitations :- once 'final' is applied, it cannot be changed.

② 'finally'

Def :- 'finally is a block in java that follows a try-catch block. It's used to ensure that a sec" of code is always executed, regardless of wheather an exception is thrown.

• Concept :- It's used for clean up activities like closing files or releasing resources.

Adv :- Ensures that critical cleanup code is executed, even if an exception occurs.

Limitation :- Cannot be used w/o a preceding try - catch block.

(3) 'finalize' :-

Def :- 'finalize' is a method in the 'object' class. It's called by the garbage collector before an object is garbage collected.

Concept :- It provides an opportunity for an object to clean up resources before it is removed from memory.

Adv :- Can be used for resource cleanup, but its' not predictable so not recommended.

Limitations :- Not guaranteed to be called, and it's generally recommended.

Q15) Difference b/w checked & unchecked Exceptions

Def :- checked exceptions are exceptions that are checked at compile time.

Concept :- They represent cond^n that a well-

written application should anticipated and recover them.

→ Ex :- 'IOException', 'SQLException' etc

→ Real-life Example for Correlation :-
A checked exception is like a predictable delay in a project schedule that can be planned for and managed.

→ Unchecked Exceptions :

Def :- Unchecked exceptions are exceptions that are not checked at compile-time.

Concept :- They represent cond^n that reflects errors in the program's logic and are typically fatal.

Ex :- 'NullPointerException' 'ArrayIndexOutOfBoundException'

Q16) Exception chaining :-

Def :- Exception chaining is the process of associating one exception with another. This allows you to capture information about the root cause of an exception.

Concept :-

It helps maintain a chain of exception, providing a detailed history of what went wrong.

Ex :-

```
try {
    // Some code that may throw exception.
} catch (Exception e) {
throw new CustomException ("Additional information", e);
}
```

Explanation :-

In this example, a new 'CustomException' is thrown with the original exception 'e' as its cause.

→ Real life :-

Consider a relay race where the baton is passed from one runner to another. If baton is dropped, it's important to know which runner dropped it & why. Exception chaining helps trace the original cause of an exception.

Q17) Difference b/w 'throw' & 'throws'

'throw' :-

Def :- 'throw' is a keyword in Java used to explicitly throws an exception.

Concept :- It is used to indicate that a specific exception has occurred & provides a custom message.

Ex :- 'throw new CustomException("An error occured");

Real time :- It's deliberately causing an issue in production line to highlight a specific problem.

'throws'

Def :- 'throws' is a keyword used in method signatures to indicate that the method may throw a specific type of exception.

Concept :- It's a declaration to inform the caller of the method that it might encounter a particular exception.

Ex :- 'public void myMethod () throws CustomException { ... }'

Real-life Example :- It's like a warning sign on a road indicating that a certain cond^n (eg. slippery road) might cause an accident.

Q18) When 'finally' Block Doesn't Execute :-

The 'finally' block may not execute in the following situations.

(1) If the JVM exists during the execution of the 'try' or 'catch' block.
(eg. due to 'System.exit()')

(2) If there is an infinity loop in the 'try' or 'catch' block.

(3) If the thread executing the 'try' or 'catch' block is interrupted or killed.

Q19) Up Casting :-

Def :- Up casting is the process of casting a reference variable to a superclass type.

Concept :-

It allows you to treat an object of a subclass as an object of its superclass.

Ex :-

```
class Animal {
    void sound () {
        System.out.println ("Generic Animal Sound");
    }
}
class Dog extends Animal {
    void sound () {
        System.out.println ("Bark");
    }
}
public class main {
    public static void main (String[] args) {
        Animal myPet = new Dog ();
        myPet.sound ();
    }
}
```

Explain :- In above example 'myPet' is of type 'Animal', but it refers to an object of type 'Dog'. The 'sound ()' method of 'Dog' is called.

Real life :- Consider a scenario where a pet store owner refers to all pets as "Animals" rather than specifying the specific breed. This allows them to generalize their interactions with all pets.

S 20) Dynamic Method Dispatch :-

Def :- Dynamic method dispatch is a mechanism in Java that allows a subclass to provide a specific implementation of a method defined in its superclass.

Concept :- It enables runtime polymorphism, where the method that gets called is determined by the type of actual object at runtime.

Ex :-

```
class Animal {
    void sound () {
        Sopln ("Generic Animal sound");
    }
}

class Dog extends Animals {
    void sound () {
        Sopln ("Bark");
    }
}

public class main {
    public static void main (String [] args) {
        Animal myPet = new Dog (); // upcasting
        myPet.sound ();          // O/P : "Bark"
    }
}
```

Explain :-

In this example, 'myPet.sound()' calls the 'sound()' method of actual object type

('Dog'), not reference type ('Animal').

Real life :- Imagine a car rental company. Regardless of the make or model, customers expect the "start engine" button to start the car. The specific implementation depends on the actual car they rent.

Q 21) Final method :-

Def :- A final method is a method that cannot be overridden by subclass.

Concept :- It provides a way to prevent subclasses from changing the behaviour of a method.

Ex :-
```
class Parent {
    final void display () {
        Sopln ("Display from Parent");
    }}

class child extends Parent {
    // Error : Cannot override the final method
    // void display () {
    // Sopln ("Display from child");
    // }
}
```

Explain :-

In this example, the 'display()' method in the 'Parent' class is marked as 'final', so it cannot be overridden in the 'child' class.

Real-life :- Consider a company policy that states a certain procedure must be followed w/o exceptions. This is like marking a method as 'final' to ensure it cannot be overridden.

Q22) Fragile Base class Problem & overcoming it :-

This is a situation where changes to a base class can break derived classes.

→ How to overcome it :-

(1) Use Abstraction :- Hide implementation details from derived classes.

(2) Avoid over-reliance on inheritance :- Prefer composition over inheritance when possible.

(3) Document Contracts clearly document the expected behaviour of base classes and methods.

Q23) Why Java doesn't support Multiple Implementation Inheritance :-

Definition :-

Multiple implementation inheritance is the ability

to inherit from more than one class.

→ Why Java doesn't support it :-

(1) Diamond Problem :- It can lead to the diamond problem, where a class inherits two methods with the same signature from different classes, creating ambiguity.

(2) Complexity :- It increases the complexity of method dispatch & introduces potential conflict.

§ 24 Marker Interface :-

Def :- A marker interface is an empty interface (interface with no methods). That serves to provide metadata about the class implementing it.

Concept :- It's used to indicate to the compiler or runtime environment that certain objects should be treated in a special way.

```
# interface Serializable {
  }
class MyClass implements Serializable {
  }
```

Explanation :-

In this example, 'Serializable' is a marker interface. By implementing it, 'My class' indicates that objects of this class can be serialized.

Real life :- Consider a shipping company that marks packages with a special sticker to indicate they need special handling. The sticker itself doesn't do anything but provides important information.

Adv :- Metadata identification :- Provides a way to add metadata to classes.

Indicates intent :- clearly signals that a class has a certain capability or should be treated in a specific way.

Limitation :- No Additional Behaviour :- Unlike regular interfaces, marker interfaces don't define any methods, so they don't add behaviour to a class.

Q25) Significance of Marker Interface :-

Def :- A marker interface is significant because it provides a way to add metadata to classes w/o adding any additional methods.

Concept :- It's a means of signaling to the compiler

or runtime environment that a class should be treated differently.

```
interface Cacheable {
}
class Product implements Cacheable {
}
```

Exp :- In this example 'Cacheable' indicates that objects of the 'Product' class can be cached.

Real-life :- Consider a library where certain books are marked with a "New Arrival" sticker. The sticker doesn't change the content of the book, but it provides information about its status.

Adv :- Metadata Identification
Code flexibility

Limitation :- No Additional Behaviour : Marker interfaces don't define any method, so they don't add behaviours to a class.