Question bank Answers JAVA

Unit 3

1. Explain java packages

Ans.

One of the main features of OOP is its ability to reuse the code already created.

• One way is to inheritance – limited to reusing the classes within the program.

• What if we need to use classes from other programs?

• This can be accomplished in java by using “Packages”

• Packages are similar to “class libraries” in other languages.

Packages are java’s way of grouping a variety of classes and/ or interfaces together.

• The grouping is usually done according to functionality.

• Packages act as containers for classes.

• Examples of packages: lang, awt, util, applet, javax, swing, net, io, sql etc.

Advantages of packages -

1. The classes contained in the package can be easily reused.

2. Two classes in two different packages can have the same name.

3. Package provide a way to hide classes thus preventing other programs or packages from accessing classes that are for internal use only.

4. Provide a way of separating “design” from “coding”.

There are two types of packages in Java:

• Built-in packages :

• They are also called as Java API Packages.

• Java API provides a large number of classes grouped into different packages according to functionality.

Java API Packages

• Java.lang: lang stands for language.

• This package has primary classes and interfaces essential for developing a basic Java program.

• It consists of wrapper classes , String, StringBuffer, StringBuilder classes to handle strings, thread class.

• Java.util: util stands for utility.

• This package contains useful classes and interfaces like Stack, LinkedList, Hashtable, Vector, Arrays, Date and Time classes.

• **Java.io: io stands for input and output.**

• This package contains streams. A stream represents flow of data from one place to another place. Streams are useful to store data in the form of files and also to perform input-output related tasks

Java.awt: awt stands for abstract window toolkit.

• This helps to develop GUI(Graphical user Interfaces). It consists of , classes which are useful to provide action for components like push buttons, radio buttons, menus etc.

• **Java.net: net stands for network.**

• Client-Server programming can be done by using this package. Classes related to obtaining authentication for network, client and server to establish communication between them are also available in java.net package.

• **Java.applet:**

• applets are programs which come from a server into a client and get executed on the client machine on a network. Applet class of this package is useful to create and use applets.

•**User-defined packages –**

The users of the Java language can also create their own packages.

• They are called user-defined packages.

• User-defined packages can also be imported into other classes and used exactly in the same way as the Built-in packages

Creating Packages

• package packagename; //to create a package

• package packagename.subpackagename;

• //to create a sub package within a package. e.g.: package pack;

• The first statement in the program must be package statement while creating a package.

• While creating a package except instance variables, declare all the members and the class itself as public then only the public members are available outside the package to other programs.

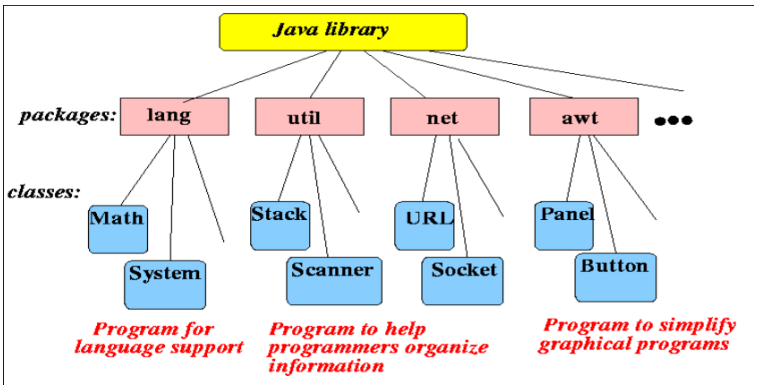
A program to create a package pack with Addition class.

```
package pack;

public class Addition

{

private double d1,d2;

public Addition(double a, double b) {

d1 = a;

d2 = b; }
```

public void sum()

{ System.out.println ("Sum is : " + (d1+d2)); }

}

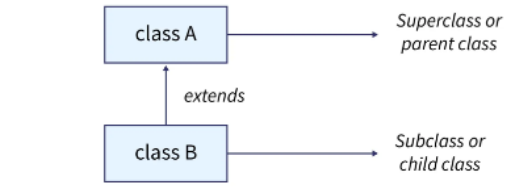


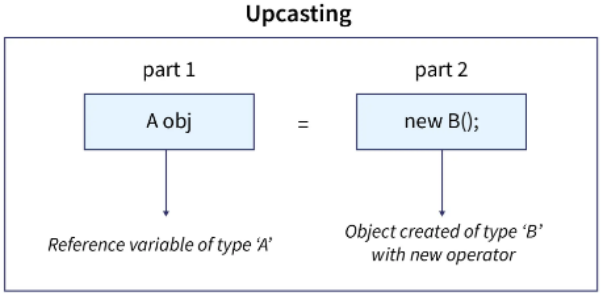
2. Explain Dynamic method dispatch while overriding methods.

Ans. Dynamic Method Dispatch is another name for Runtime polymorphism in Java which originates with the concept of method overriding. **In this case, the call to an overridden method will be resolved at the time of code execution (runtime) rather than the compile time.**

The basis of dynamic method dispatch in Java starts with Inheritance where two or more classes exhibit a parent-child relationship. Now, there might be several versions of the same method in the parent as well as child classes or you can also call them superclass and subclasses.

It seems puzzling as to which version of the same method will be called. However, Java Virtual Machine (JVM) easily perceives the same during runtime based on the type of object being referred to.





Method Overriding

When you can find the same method with the same signature (**method name, parameters, and return type**) in inherited classes, it leads to **method overriding**. The implementation of the class, i.e., the body of the class might differ in the overridden methods.

Therefore, the inherited classes having the same method with varied implementations make up an example of runtime polymorphism.

Advantages

- Runtime polymorphism in Java allows the **superclass** to define as well as share its own method and also allows the sub-classes to define their own implementation.
- The subclasses have the privilege to use the same method as their parent or define their specific implementation for the same method wherever necessary. Therefore, in one way, it supports code reusability when using the same method and implementation.
- It allows method overriding which is the basis for runtime polymorphism in Java

Method overriding is one of the ways in which Java **supports Runtime Polymorphism**. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed

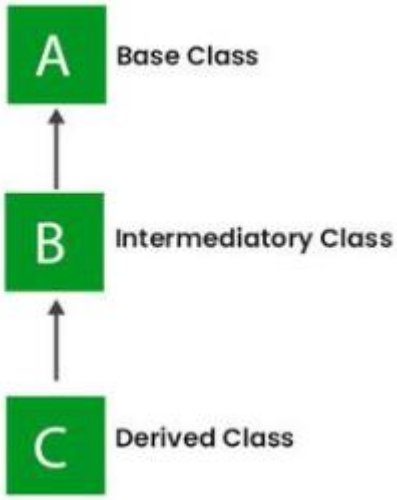
A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.

3. Explain Multilevel Inheritance in Java with suitable diagram and explain.

Ans.

Explanation :

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.



Example :

```
class Shape {
  public void display() {
    System.out.println("Inside display");
  }
}
class Rectangle extends Shape {
  public void area() {
    System.out.println("Inside area");
  }
}
class Cube extends Rectangle {
  public void volume() {
    System.out.println("Inside volume");
  }
}
public class Tester {
  public static void main(String[] arguments) {
    Cube cube = new Cube();
    cube.display();
    cube.area();
    cube.volume();
```

```
  }
}
```

4. Write a program to implement Interface in Java.

Ans.

Interfaces are **blueprints** for classes that define a **contract** of **methods** and **fields** that implementing **classes must provide**.

The interface in Java is a mechanism to achieve **abstraction**. **There can be only abstract methods in the Java interface, not the method body. It is used to achieve abstraction and multiple inheritances in Java using Interface**. In other words, you can say that **interfaces can have abstract methods and variables**. **It cannot have a method body**. Java Interface also represents the IS-A relationship.

Syntax for Java Interfaces

```
interface {

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

```
// A simple interface

interface Player
{
    final int id = 10;
    int move();
}
```

Program :

```
interface Player {
   final int id = 10; // Constant member
   int move(); // Abstract method
}
// Class implementing the Player interface
class ChessPlayer implements Player {
   @Override
   public int move() {
      // Chess-specific move logic
```

```
        System.out.println("Making a chess move...");
        return 5; // Example return value
    }
}
public class InterfaceDemo {
    public static void main(String[] args) {
        Player player = new ChessPlayer();
        int playerMove = player.move();
        System.out.println("Player ID: " + Player.id);
        System.out.println("Move result: " + playerMove);
    }
}
```

5. Write a program Demonstrating factorial of first n number using class and object concept.

Ans.

```
class Factorial {
    int number;
    Factorial(int n) {
        this.number = n;
    }
    int calculateFactorial() {
        if (number == 0) {
            return 1;
        } else {
            return number * new Factorial(number - 1).calculateFactorial();          // Recursion is used
        }
    }
}
public class FactorialDemo {
    public static void main(String[] args) {
```

```
        int n = 5; // Number for which factorial is to be calculated
        Factorial factorialObj = new Factorial(n);                  // Object creation
        int factorial = factorialObj.calculateFactorial();
        System.out.println("Factorial of " + n + " is: " + factorial);
    }
}
```

6. Make use of abstract class to find area of square, cube and square root.

Ans.

```
abstract class Shape {                          // Abstract class
    protected double side;
    public Shape(double side) {                 // parameterised constructor
        this.side = side;
    }
    abstract double getArea();
    // Method for square root calculation
    double getSquareRoot() {
        return Math.sqrt(side);
    }
}
class Square extends Shape {
    public Square(double side) {
        super(side);
    }
    @Override
    double getArea() {
        return side * side;
    }
}
class Cube extends Shape {
```

```
    public Cube(double side) {
        super(side);
    }
    @Override
    double getArea() {
        return 6 * side * side;
    }
}
public class AbstractShapeDemo {
    public static void main(String[] args) {
        Shape square = new Square(5);
        Shape cube = new Cube(4);
        System.out.println("Area of square: " + square.getArea());
        System.out.println("Area of cube: " + cube.getArea());
        System.out.println("Square root of cube's side: " + cube.getSquareRoot());
    }
}
```

7. Implement a Java Program to find the area of a circle using a parameterized constructor. Make use of 'this' keyword

Ans.

```
class Circle {
    double radius;
    Circle(double r) {                    // Parameterized constructor to initialize radius
        this.radius = r;                  // Using 'this' to refer to the current object's radius
    }
    double calculateArea() {
        return Math.PI * radius * radius;
    }
    public static void main(String[] args) {
```

```
        double radius = 5.0;
        Circle circle = new Circle(radius); // Call constructor to create Circle object
        double area = circle.calculateArea();
        System.out.println("Area of the circle: " + area);
    }
}
```

8. What is Inheritance? Explain the types of inheritances in Java.

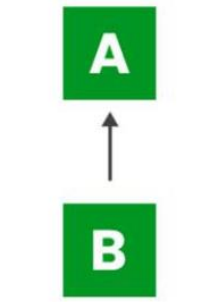
Ans.

Inheritance is an important pillar of **OOP** (Object-Oriented Programming). **It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class**. **In Java, Inheritance means creating new classes based on existing ones**. **A class that inherits from another class can reuse the methods and fields of that class**. In addition, **you can add new fields and methods to your current class as well.**

Java Inheritance Types -

1. **Single Inheritance :**

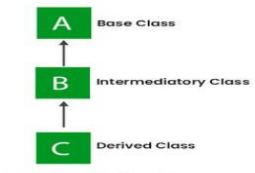
In single inheritance**, subclasses inherit the features of one superclass.** In the image below, class A serves as a base class for the derived class B.



Single Inheritance

2. **Multilevel Inheritance**

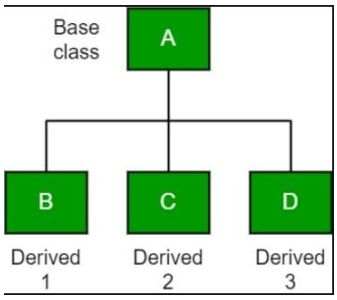
In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.



Multilevel Inheritance

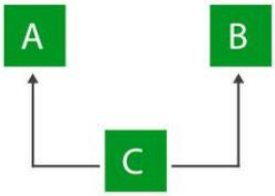
3. **Hierarchical Inheritance**

In Hierarchical Inheritance, **one class serves as a superclass (base class) for more than one subclass**. In the below image, class A serves as a base class for the derived classes B, C, and D.



4. **Multiple Inheritance (Through interfaces)**

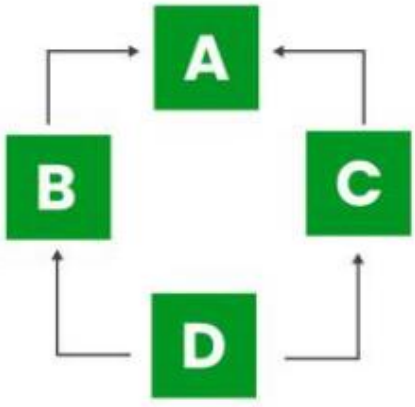
In Multiple inheritances, **one class can have more than one superclass and inherit features from all parent classes**. Please note that Java does not support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interfaces A and B.



Multiple Inheritance

5. **Hybrid Inheritance**

It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritances with classes, hybrid inheritance involving multiple inheritance is also not possible with classes. In Java, **we can achieve hybrid inheritance only through Interfaces if we want to involve multiple inheritance to implement Hybrid inheritance.**



Hybrid Inheritance

9. Explain packages in Java? Explain how to create userdefined package in java with example

Ans. First question (repeated one)

10. Differentiate method Overriding and method Overloading

Ans.

Method Overriding	Method Overloading
Method overriding is a run-time polymorphism.	Method overloading is a compile-time polymorphism.
Method overriding is used to grant the specific implementation of the method which is already provided by its parent class or superclass.	Method overloading helps to increase the readability of the program.
It is performed in two classes with inheritance relationships.	It occurs within the class.
Method overriding always needs inheritance.	Method overloading may or may not require inheritance.
In method overriding, methods must have the same name and same signature.	In method overloading, methods must have the same name and different signatures.
In method overriding, the return type must be the same or co-variant.	In method overloading, the return type can or can not be the same, but we just have to change the parameter.
Dynamic binding is being used for overriding methods.	Static binding is being used for overloaded methods.
It gives better performance. The reason behind this is that the binding of overridden methods is being done at runtime.	Poor Performance due to compile time polymorphism.
Private and final methods can't be overridden.	Private and final methods can be overloaded.
The argument list should be the same in method overriding.	The argument list should be different while doing method overloading.

11. How static, final keywords are used in Java with an example.

Ans.

The **static** keyword in Java is mainly used for **memory management**. **The static keyword in Java is used to share the same variable or method of a given class**. The users can **apply** static keywords with **variables**, **methods**, **blocks**, and **nested classes**. The static keyword belongs to the class than an instance of the class. **The static keyword is used for a constant variable or a method that is the same for every instance of a class.**

The *static* keyword is a non-access modifier in Java that is applicable for the following:

1. **Blocks**
2. **Variables**
3. **Methods**
4. **Classes**

```
class Test
{
    // static method
    static void m1()
    {
        System.out.println("from m1");
    }

    public static void main(String[] args)
    {
          // calling m1 without creating
          // any object of class Test
           m1();
    }
}
```

Final keyword :

Final is a **non-access modifier applicable only to a variable, a method, or a class**.

We must initialize a final variable, otherwise, the compiler will throw a compile-time error. **A final variable can only be initialized once**, either via an initializer or an assignment statement. There are **three ways to initialize a final variable:**

1. **You can initialize a final variable when it is declared**. This approach is the most common. **A final variable is called a blank final variable if it is not initialized while declaration**. Below are the two ways to initialize a blank final variable.
2. **A blank final variable can be initialized inside an instance-initializer block or inside the constructor. If you have more than one constructor in your class then it must be initialized in all of them,** otherwise, a compile-time error will be thrown.
3. **A blank final static variable can be initialized inside a static block.**

The only **difference** between a **normal variable** and a **final variable** is that **we can re-assign the value to a normal variable but we cannot change the value of a final variable once assigned.** Hence final variables must be used only for the values that we want to remain constant throughout the execution of the program.

12. Explain abstract class with examples. Differentiate between Abstract class and Interfaces in Java

Ans.

Abstract class in java :

Java abstract class is a class that can not be initiated by itself, it needs to be subclassed by another class to use its properties. An abstract class is declared using the "**abstract**" keyword in its class definition.

1. An instance of an abstract class can not be created.
2. Constructors are allowed.
3. We **can have an abstract class without any abstract method.**
4. **There can be a final method in abstract class but any abstract method in class(abstract class) can not be declared as final** or in simpler terms **final method can not be abstract itself as it will show an error:** "Illegal combination of modifiers: abstract and final"
5. We can define static methods in an abstract class
6. We can use the **abstract keyword** for declaring **Outer class as well as inner classes** as abstract
7. If a **class** contains at least **one abstract method** then **compulsory** should **declare a class as abstract**
8. If the **Child class** is unable to provide implementation to all abstract methods of the **Parent class** then we should declare that **Child class as abstract** so that the next level Child class should provide implementation to the remaining abstract method

Difference between abstract class and interfaces in Java :

S.No.	Abstract Class	Interface
1.	An abstract class can contain both abstract and non-abstract methods.	Interface contains only abstract methods.
2.	An abstract class can have all four; static, non-static and final, non-final variables.	Only final and static variables are used.
3.	To declare abstract class abstract keywords are used.	The interface can be declared with the interface keyword.
4.	It supports multiple inheritance.	It does not support multiple inheritance.
5.	The keyword 'extend' is used to extend an abstract class	The keyword implement is used to implement the interface.
6.	It has class members like private and protected, etc.	It has class members public by default.

13. Explain Constructor in Java with example

Ans.

In Java, a **Constructor is a block of codes similar to the method**. **It is called when an instance of the class is created**. **At the time of calling the constructor, memory for the object is allocated in the memory.** It is a special type of method that is **used to initialize the object. Every time an object is created using the new () keyword, at least one constructor is called.**

```
class Geek
{
    .......
    // A Constructor
    Geek() {
    }
    .......
}
```

Each time an object is created using a new () keyword, at least one constructor (it could be the default constructor) is invoked to assign initial values to the data members of the same class. Rules for writing constructors are as follows:

- **The constructor(s) of a class must have the same name as the class name in which it resides.**
- **A constructor in Java can't be abstract, final, static, or Synchronized.**
- **Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.**

Types of Constructors in Java:

Now is the correct time to discuss the types of the constructor, so primarily there are three types of constructors in Java are mentioned below:

- **Default Constructor**
- **Parameterized Constructor**
- **Copy Constructor**

1. Default Constructor in Java

A constructor that has no parameters is known as default the constructor. **A default constructor is invisible, and if we write a constructor with no arguments, the compiler does not create a default constructor**. It is taken out. It is being overloaded and called a parameterized constructor. The default constructor changed into the parameterized constructor. But Parameterized constructor can't change the default constructor.

```java
// Java Program to demonstrate
// Default Constructor
import java.io.*;

// Driver class
class GFG {

    // Default Constructor
    GFG() { System.out.println("Default constructor"); }

    // Driver function
    public static void main(String[] args)
    {
        GFG hello = new GFG();
    }
}
```

2. Parameterized Constructor in Java

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.

```java
// Java Program for Parameterized Constructor
import java.io.*;
class Geek {
    // data members of the class.
    String name;
    int id;
    Geek(String name, int id)
    {
        this.name = name;
        this.id = id;
    }
}
class GFG {
    public static void main(String[] args)
    {
        // This would invoke the parameterized constructor.
        Geek geek1 = new Geek("avinash", 68);
        System.out.println("GeekName :" + geek1.name
                           + " and GeekId :" + geek1.id);
    }
}
```

3. Copy Constructor in Java

Unlike other constructors copy constructor is passed with another object which copies the data available from the passed object to the newly created object.

```java
// Java Program for Copy Constructor
import java.io.*;

class Geek {
    // data members of the class.
    String name;
    int id;

    // Parameterized Constructor
    Geek(String name, int id)
    {
        this.name = name;
        this.id = id;
    }

    // Copy Constructor
    Geek(Geek obj2)
    {
        this.name = obj2.name;
        this.id = obj2.id;
    }
}
class GFG {
    public static void main(String[] args)
    {
        // This would invoke the parameterized constructor.
        System.out.println("First Object");
        Geek geek1 = new Geek("avinash", 68);
        System.out.println("GeekName :" + geek1.name
                           + " and GeekId :" + geek1.id);

        System.out.println();

        // This would invoke the copy constructor.
        Geek geek2 = new Geek(geek1);
        System.out.println(
            "Copy Constructor used Second Object");
        System.out.println("GeekName :" + geek2.name
                           + " and GeekId :" + geek2.id);
    }
}
```

 Warning : Do Not fully depend on this

14. Explain the use of static variable and static method in java Variable with example

Ans.

Static Variables:

- **Declaration: Declared using the static keyword before the variable type.**
- Scope: **Belong to the class itself, not individual objects**.
- Memory allocation: **Only one copy exists in memory**, **shared by all objects of the class**.
- Access: Accessed directly using the class name (e.g., ClassName.staticVariable).
- **Common use cases:**
    - Storing constants that are common to all objects (e.g Math.PI).
    - Keeping track of global state information (e.g. number of objects created).

Example of static variables :

```
class Circle {
   static final double PI = 3.14159;          // Static variable for PI declared initially
   private double radius;
   private static int numberOfCircles = 0; // Static variable to count objects
   public Circle(double radius) {
      this.radius = radius;
      numberOfCircles++;
   public static int getNumberOfCircles() { // Static method to access static variable
      return numberOfCircles;
   }
}
```

Static Methods:

- **Declaration: Declared using the static keyword before the method's return type.**
- **Association: Belong to the class, not objects.**
- Invocation: **Called directly using the class name** (e.g., ClassName.staticMethod()).
- Access restrictions: **Can only access static variables and call other static methods within the class.**
- **Common use cases:**

- **Utility methods that don't require object state (e.g., Math.sqrt()).**
- **Factory methods to create objects (e.g., Calendar.getInstance()).**

Example of static method :

```
class MathUtils {
   public static double calculateArea(double radius) {                    // Static method for area calculation
      return Math.PI * radius * radius;
   }
}
MathUtils.calculateArea(5.0); // Calling the static method
```

15. Write significance of keyword 'super'? Demonstrate with example each of the cases.

Ans.

Significance:

- Accessing Superclass Members: **Allows a subclass to directly access members (fields or methods) of its superclass**, even if they have the same name in both classes.
- Calling Superclass Constructors: **Used to invoke a constructor of the superclass, ensuring proper object initialization across the inheritance hierarchy.**

The super keyword in Java is a reference variable that is used to refer to parent class when we're working with objects.

The Keyword "super" came into the picture with the concept of Inheritance.

Use of super keyword in Java

- **Use of super with Variables :**

This scenario occurs when a derived class and base class have the same data members.

```java
// Base class vehicle
class Vehicle {
    int maxSpeed = 120;
}

// sub class Car extending vehicle
class Car extends Vehicle {
    int maxSpeed = 180;

    void display()
    {
        // print maxSpeed of base class (vehicle)
        System.out.println("Maximum Speed: "
                           + super.maxSpeed);
    }
}

// Driver Program
class Test {
    public static void main(String[] args)
    {
        Car small = new Car();
        small.display();
    }
}
```

- **Use of super with Methods :**

This is used when we want to call the parent class method. So whenever a parent and child class have the same-named methods then to resolve ambiguity we use the super keyword.

```java
// superclass Person
class Person {
    void message()
    {
        System.out.println("This is person class\n");
    }
}
// Subclass Student
class Student extends Person {
    void message()
    {
        System.out.println("This is student class");
    }
    // Note that display() is
    // only in Student class
    void display()
    {
        // will invoke or call current
        // class message() method
        message();

        // will invoke or call parent
        // class message() method
        super.message();
    }
}
// Driver Program
class Test {
    public static void main(String args[])
    {
        Student s = new Student();

        // calling display() of Student
        s.display();
    }
}
```

- **Use of super with Constructors :**

The super keyword can also be used to access the parent class constructor. One more important thing is that 'super' can call both parametric as well as non-parametric constructors depending on the situation.

```java
// superclass Person
class Person {
    Person()
    {
        System.out.println("Person class Constructor");
    }
}

// subclass Student extending the Person class
class Student extends Person {
    Student()
    {
        // invoke or call parent class constructor
        super();

        System.out.println("Student class Constructor");
    }
}

// Driver Program
class Test {
    public static void main(String[] args)
    {
        Student s = new Student();
    }
}
```

16. Describe Using 'super' to call super class constructer

Ans.

Purpose:

- Ensures proper **initialization of inherited fields and resources from the superclass.**
- **Establishes a chain of constructor calls across the inheritance hierarchy.**

Syntax:

- Within a subclass constructor, use super(parameter-list); as the first statement.
- parameter-list specifies arguments for the superclass constructor being invoked.

Example : Calling no argument Super class constructor.

```
class Animal {
   public Animal() {
      System.out.println("Animal constructor");
   }
}
class Dog extends Animal {
   public Dog() {
      super(); // Calls Animal()
```

```
        System.out.println("Dog constructor");
    }
}
```

- First Statement Rule: The super call must be the first statement within a subclass constructor.
- Implicit Call: If not called explicitly, the compiler automatically inserts a call to the superclass's no-argument constructor (if it exists).
- Constructor Chaining: **This process continues up the inheritance chain, ensuring all superclasses are properly initialized.**
- Static Constructors: **super cannot be used within static constructors, as they're not associated with instances of classes.**

17. Explain Garbage collector and Finalize().

Ans.

Garbage collection :

Garbage collection in Java is the process by which Java programs perform automatic memory management. **Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program**. Eventually, some objects will no longer be needed. **The garbage collector finds these unused objects and deletes them to free up memory.**

What is Garbage Collection?

In C/C++, a programmer is responsible for both the creation and destruction of objects. Usually, programmer neglects the destruction of useless objects. Due to this negligence, at a certain point, sufficient memory may not be available to create new objects, and the entire program will terminate abnormally, causing OutOfMemoryErrors.

But in Java, the programmer need not care for all those objects which are no longer in use. Garbage collector destroys these objects. **The main objective of Garbage Collector is to free heap memory by destroying unreachable objects**.

How Does Garbage Collection in Java works?

Java garbage collection is an automatic process. Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused or unreferenced object is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed. The programmer does not need to mark objects to be deleted explicitly. The garbage collection implementation lives in the JVM.

Finalize :

The Java finalize() method of Object class is a method that the Garbage Collector always calls just before the deletion/destroying the object which is eligible for Garbage Collection to perform clean-up activity. Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection, or we can say resource de-allocation. Remember, it is not a reserved keyword. Once the finalize() method completes immediately, Garbage Collector destroys that object.

Finalization: Just before destroying any object, the garbage collector always calls finalize() method to perform clean-up activities on that object. This process is known as Finalization in Java.

The finalize method, which is present in the Object class, has an empty implementation. In our class, clean-up activities are there. Then we have to override this method to define our clean-up activities.

18. Explain the concept of dynamic method dispatch with example.

Ans. Repeated Question …

19. Explain use of Interface in Java. How it is different from a class?

Ans.

Use Cases:

- Defining common behavior: Establish a shared contract for unrelated classes to follow.
- Segregating responsibilities: Separate interfaces for different functionalities, promoting modularity.
- Facilitating polymorphism: Allow objects of different classes to be treated uniformly as long as they implement the same interface.
- Achieving loose coupling: Reduce dependencies between classes by relying on interfaces rather than concrete implementations.
- Enabling pluggable components: Allow different implementations to be swapped without affecting client code.

Interfaces:

- Blueprint for behavior: Define a contract that classes can implement, specifying what methods they must have without defining their implementation.
- Abstraction: Promote loose coupling and flexibility by separating the "what" from the "how."
- **Key members:**
    - **Abstract methods (without bodies)**
    - **Constant variables (public static final)**
    - **Default methods (with bodies, added in Java 8)**

- **Static methods (added in Java 8)**

Feature	Interface	Class
Purpose	Define a contract	Define a blueprint for objects
Members	Abstract methods, constants, default methods, static methods	Fields (variables), constructors, methods (concrete and abstract)
Implementation	Implemented by classes using "implements" keyword	Instantiated to create objects
Inheritance	Multiple inheritance supported	Single inheritance (except for interfaces)
Instantiation	Cannot be instantiated directly	Can be instantiated

Examples:

Comparable interface for ordering objects

List interface for collections of objects

Runnable interface for threads

ActionListener interface for event handling

20. Illustrate use of Packages? How access protection is provided to packages?

Ans.

Packages:

- Organizational Units: Group related classes and interfaces together, promoting better code structure, maintainability, and namespace management.
- Declaration: Use the package keyword at the beginning of a Java source file.
- Hierarchical Structure: Packages can be nested within each other, forming a logical tree-like structure.

Access Protection:

Rules of Access:

- Classes: Only public classes are accessible from outside the package.
- Members: Access level depends on the modifier used (e.g., private field only accessible within the class).
- Inheritance: Subclasses inherit protected and public members of the superclass, but not private members.

4. Benefits of Access Protection:

- **Access Modifiers: Control visibility and accessibility of classes and members within and across packages.**
    - public: Accessible from anywhere.
    - protected: Accessible within the same package and subclasses.
    - default (no modifier): Accessible within the same package only.
    - private: Accessible only within the same class.

Unit 4

1. What is Exception? How is it handled? Explain with suitable example.

Ans. In Java, Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

All exception and error types are subclasses of the class Throwable, which is the base class of the hierarchy. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception

Default Exception Handling: Whenever inside a method, if an exception has occurred, the method creates an Object known as an Exception Object and hands it off to the run-time system(JVM). The exception object contains the name and description of the exception and the current state of the program where the exception has occurred. Creating the Exception Object and handling it in the run-time system is called throwing an Exception. There might be a list of the methods that had been called to get to the method where an exception occurred. This ordered list of methods is called **Call Stack**. Now the following procedure will happen.

- The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred exception. The block of the code is called an **Exception handler**.
- The run-time system starts searching from the method in which the exception occurred and proceeds through the call stack in the reverse order in which methods were called.
- If it finds an appropriate handler, then it passes the occurred exception to it. An appropriate handler means the type of exception object thrown matches the type of exception object it can handle.
- If the run-time system searches all the methods on the call stack and couldn't have found the appropriate handler, then the run-time system handover the Exception Object to the **default exception handler**, which is part of the run-time system. This handler prints the exception information in the following format and terminates the program **abnormally**.

Example :

```
class GFG {

    // Method 1
    // It throws the Exception(ArithmeticException).
    // Appropriate Exception handler is not found
    // within this method.
    static int divideByZero(int a, int b)
    {

        // this statement will cause ArithmeticException
        // (/by zero)
        int i = a / b;

        return i;
    }

    // The runTime System searches the appropriate
    // Exception handler in method also but couldn't have
    // found. So looking forward on the call stack
    static int computeDivision(int a, int b)
    {

        int res = 0;

        // Try block to check for exceptions
        try {

            res = divideByZero(a, b);
        }

        // Catch block to handle NumberFormatException
        // exception Doesn't matches with
        // ArithmeticException
        catch (NumberFormatException ex) {
            // Display message when exception occurs
            System.out.println(
                "NumberFormatException is occurred");
        }
        return res;
    }

    // Method 2
    // Found appropriate Exception handler.
    // i.e. matching catch block.
    public static void main(String args[])
    {

        int a = 1;
        int b = 0;

        // Try block to check for exceptions
        try {
            int i = computeDivision(a, b);
        }

        // Catch block to handle ArithmeticException
        // exceptions
        catch (ArithmeticException ex) {

            // getMessage() will print description
            // of exception(here / by zero)
            System.out.println(ex.getMessage());
        }
    }
}
```

 Warning : Do Not fully depend on this

2. Discuss exception handling in Java in detail? explain the advantages of exception handling

Ans. Explanation is done in first question about Exception handling

Advantages are :

1. Separation of Concerns:

- Isolates error-handling code: Exception handling allows you to cleanly separate the code that deals with normal execution from the code that handles exceptional situations. This makes the main program logic easier to read and understand, as it's not cluttered with error-checking statements.
- Enhances code readability and maintainability: By separating error-handling logic, the code becomes more modular, organized, and easier to maintain.

2. Graceful Termination:

- Prevents abrupt program crashes: Instead of crashing unexpectedly, programs can handle errors gracefully and either recover from them or terminate in a controlled manner.
- Provides user-friendly error messages: You can display informative messages to the user, explaining the error and suggesting potential actions.
- Enables corrective actions: You can implement code to attempt recovery from errors, such as retrying operations or suggesting alternative actions.

3. Improved Code Organization:

- Enforces error handling: Checked exceptions in Java require developers to explicitly handle potential errors, promoting a more robust and error-aware coding approach.
- Structures error management: The try-catch-finally block provides a clear structure for handling exceptions, making it easier to reason about error flow.

4. Centralized Error Management:

- Common exception handling: You can create centralized exception handlers to manage errors in a consistent manner across different parts of your application.
- Logging and debugging: Exception handling facilitates logging of errors and debugging by providing information about the error type, location, and stack trace.

5. Improved Scalability and Performance:

- Reduces unnecessary processing: By catching and handling exceptions, you can avoid redundant operations or resource consumption, leading to better performance.
- Optimizes resource usage: Proper exception handling can help release resources appropriately, preventing memory leaks or other resource-related issues.

6. Enhanced Code Reusability:

- Safer component integration: Exception handling promotes safer integration of reusable components by allowing them to signal errors without compromising the integrity of the calling code.

- Clearer component interfaces: Well-defined exception handling in components makes their usage and error-handling expectations more transparent.

3. State with example the use of following built in exception in Java

- **IllegalArgument Exception :**

Thrown when a method receives an illegal or inappropriate argument

Example :

```
public void calculateArea(int length, int width) {
   if (length <= 0 || width <= 0) {
      throw new IllegalArgumentException("Length and width must be positive");
   }
   // Calculate area
}
```

- **Arithmetic Exception :**

Thrown when an exceptional arithmetic condition occurs, such as division by zero.

Example :

int result = 10 / 0; // ArithmeticException: / by zero

- **NumberFormatException :**

Thrown when a string cannot be parsed into a number of the specified type.

Example :

int number = Integer.parseInt("hello"); // NumberFormatException: For input string: "hello"

- **StringIndexOutOfBoundException :**

- Thrown when an index is out of bounds for a string operation.

Example :

String name = "John";

char firstChar = name.charAt(10); // StringIndexOutOfBoundsException: index 10 out of bounds for length 4

- **Null Pointer Exception :**

- Thrown when an attempt is made to use a null reference.

Example :

String myString = null;

myString.length(); // NullPointerException: Cannot invoke "length()" on a null object

- **ArrayIndexOutOfBoundException :**

- Thrown when an array is accessed with an illegal index.

Example :

int[] numbers = new int[5];

numbers[10] = 15; // ArrayIndexOutOfBoundsException: 10

4. Explain Chained exceptions.

Ans.

Chained Exceptions allows to relate one exception with another exception, i.e one exception describes cause of another exception. For example, consider a situation in which a method throws an ArithmeticException because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw only ArithmeticException to the caller. So the caller would not come to know about the actual cause of exception. Chained Exception is used in such type of situations. Constructors Of Throwable class Which support chained exceptions in java :

1. **Throwable**(Throwable cause) :- Where cause is the exception that causes the current exception.
2. **Throwable**(String msg, Throwable cause) :- Where msg is the exception message and cause is the exception that causes the current exception.

Methods Of Throwable class Which support chained exceptions in java :

1. **getCause() method** :- This method returns actual cause of an exception.
2. **initCause(Throwable cause) method** :- This method sets the cause for the calling exception.

```java
// Java program to demonstrate working of chained exceptions
public class ExceptionHandling
{
    public static void main(String[] args)
    {
        try
        {
            // Creating an exception
            NumberFormatException ex =
                       new NumberFormatException("Exception");

            // Setting a cause of the exception
            ex.initCause(new NullPointerException(
                      "This is actual cause of the exception"));

            // Throwing an exception with cause.
            throw ex;
        }

        catch(NumberFormatException ex)
        {
            // displaying the exception
            System.out.println(ex);

            // Getting the actual cause of the exception
            System.out.println(ex.getCause());
        }
    }
}
```

Chained exceptions, also known as nested exceptions, allow you to associate a cause with an exception in Java. This is useful when you want to propagate information about the original cause of an exception.

5. Explain nested try statements in Java with example.

Ans.

Concept:

- Nested try statements allow you to place one try block within another try block.
- This creates a hierarchy for handling exceptions at different levels of code execution.

Purpose:

- Handling multiple types of exceptions: Different try blocks can catch specific exceptions, providing more granular control over error handling.
- Protecting critical code: Inner try blocks can safeguard sensitive operations within a broader try block, ensuring their execution even if outer operations fail.
- Handling exceptions within loops: You can retry operations within loops using inner try blocks, preventing premature termination due to exceptions.

Execution Flow:

1. Exception occurs in inner try:
    - The corresponding inner catch block attempts to handle it.
    - If not handled, the exception propagates to the outer try block.
2. Exception occurs in outer try (and not within an inner try):
    - The outer catch blocks attempt to handle it.
    - If not handled, the exception propagates to the next outer try block or the JVM.

Key Points:

- Each try block has its own context, represented as a stack.
- Exceptions propagate upwards through nested try blocks until a matching catch block is found.
- Finally blocks execute even if exceptions occur, ensuring resource cleanup.

```java
class NestedTry {

    // main method
    public static void main(String args[])
    {
        // Main try block
        try {

            // initializing array
            int a[] = { 1, 2, 3, 4, 5 };

            // trying to print element at index 5
            System.out.println(a[5]);

            // try-block2 inside another try block
            try {

                // performing division by zero
                int x = a[2] / 0;
            }
            catch (ArithmeticException e2) {
                System.out.println("division by zero is not possible");
            }
        }
        catch (ArrayIndexOutOfBoundsException e1) {
            System.out.println("ArrayIndexOutOfBoundsException");
            System.out.println("Element at such index does not exists");
        }
    }
    // end of main method
}
```

6. What are the different types of errors? What is the use of throw, throws, finally.

Ans.

1. Run Time Error:

Run Time errors occur or we can say, are detected during the execution of the program. Sometimes these are discovered when the user enters an invalid data or data which is not relevant. Runtime errors occur when a program does not contain any syntax errors but asks the computer to do something that the computer is unable to reliably do. During compilation, the compiler has no technique to detect these kinds of errors. It is the JVM (Java Virtual Machine) that detects it while the program is running. To handle the error during the run time we can put our error code inside the try block and catch the error inside the catch block.

2. Compile Time Error:

Compile Time Errors are those errors which prevent the code from running because of an incorrect syntax such as a missing semicolon at the end of a statement or a missing bracket, class not found, etc. These errors are detected by the java compiler and an error message is displayed on the screen while compiling. Compile Time Errors are sometimes also referred to as Syntax errors. These kind of errors are easy to spot and rectify because the java compiler finds them for you. The compiler will tell you which piece of code in the program got in trouble and its best guess as to what you did wrong. Usually, the compiler indicates the exact line where the error is, or sometimes the line just before it, however, if the problem is with incorrectly nested braces, the actual error may be at the beginning of the block.

3.Logical Error: A logic error is when your program compiles and executes, but does the wrong thing or returns an incorrect result or no output when it should be returning an output. These errors are

detected neither by the compiler nor by JVM. The Java system has no idea what your program is supposed to do, so it provides no additional information to help you find the error. Logical errors are also called Semantic Errors. These errors are caused due to an incorrect idea or concept used by a programmer while coding. Syntax errors are grammatical errors whereas, logical errors are errors arising out of an incorrect meaning. For example, if a programmer accidentally adds two variables when he or she meant to divide them, the program will give no error and will execute successfully but with an incorrect result.

Throw :

- **Purpose: To signal the occurrence of an exception within a method.**
- Syntax: throw new ExceptionName(optional message);
- **Process:**
    1. Creates an object of the specified exception class.
    2. Halts the normal execution of the method.
    3. Passes control to the nearest enclosing try-catch block that can handle the exception.

```
if (balance < amount) {
   throw new InsufficientBalanceException("Balance is too low");
}
```

Throws :

- Purpose: To declare that a method might throw certain types of exceptions.
- Syntax: public method_name() throws Exception1, Exception2, ...
- Informational purpose: Informs the compiler and callers of the method about potential exceptions.
- Does not handle exceptions: Doesn't handle the exceptions itself; it's a warning for callers.

```
public void withdraw(int amount) throws InsufficientBalanceException {
   // ...
}
```

Finally :

- Purpose: To ensure that a block of code is always executed, regardless of whether an exception occurs or not.
- Placement: Used with try-catch blocks.

- Typical usage: For releasing resources (e.g., closing files, connections), performing cleanup tasks, or logging.
- Execution: Always executed, even if a return statement is encountered within try or catch blocks.
- **Example:**

```
try {
   // Code that might throw exceptions
} catch (Exception e) {
   // Handle exception
} finally {
   // Code that will always execute
   file.close(); // Ensure file is closed
}
```

7. Differentiate between throw and throws keywords in Java.

Ans.

S. No.	Key Difference	throw	throws
1.	Point of Usage	The **throw** keyword is used inside a function. It is used when it is required to throw an Exception logically.	The **throws** keyword is used in the function signature. It is used when the function has some statements that can lead to exceptions.
2.	Exceptions Thrown	The **throw** keyword is used to throw an exception explicitly. It can throw only one exception at a time.	The **throws** keyword can be used to declare multiple exceptions, separated by a comma. Whichever exception occurs, if matched with the declared ones, is thrown automatically then.
3.	Syntax	Syntax of **throw** keyword includes the instance of the Exception to be thrown. Syntax wise throw keyword is followed by the instance variable.	Syntax of **throws** keyword includes the class names of the Exceptions to be thrown. Syntax wise throws keyword is followed by exception class names.
4.	Propagation of Exceptions	**throw** keyword cannot propagate checked exceptions. It is only used to propagate the unchecked Exceptions that are not checked using the throws keyword.	**throws** keyword is used to propagate the checked Exceptions only.

8. Implement a Java Program to handle Multiple Exceptions.

Ans.

```
public class HandlingMultipleExceptions {
    public static void main(String[] args) {
        try {
            // Code that might throw multiple exceptions
            int[] array = new int[5];
            array[10] = 30;   // ArrayIndexOutOfBoundsException
            int result = 30 / 0;  // ArithmeticException
            String str = null;
            System.out.println(str.length());                    // NullPointerException
        } catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {
            // Handle both ArithmeticException and ArrayIndexOutOfBoundsException
            System.out.println("Arithmetic or Array Index Exception: " + e.getMessage());
        } catch (NullPointerException e) {
            // Handle NullPointerException
            System.out.println("NullPointerException: " + e.getMessage());
        } catch (Exception e) {
            // Handle any other exception
            System.out.println("General Exception: " + e.getMessage());
```

9. Demonstrate how user defined Exceptions are created

Ans.

```
public class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message); // Pass the message to the parent constructor
    }
}
```

```
public class AgeCheck {
    public static void main(String[] args) {
        int age = 15;
        try {
            checkAge(age);
        } catch (InvalidAgeException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
    public static void checkAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or older");
        } else {
            System.out.println("Valid age");
        }
    }
}
```

- User-defined exceptions inherit methods from the Exception class, like getMessage() and printStackTrace().
- You can create a hierarchy of custom exceptions for more specific error handling.
- Use checked exceptions (extending Exception) for errors that must be handled explicitly, and unchecked exceptions (extending RuntimeException) for errors that can be handled optionally.

10. Illustrate with example how Synchronization is achieved in Java.

Ans.

- **Shared Resources**: Objects or data that multiple threads can access simultaneously.
- **Race Conditions**: Unpredictable results when multiple threads access and modify shared resources without coordination.
- **Synchronization**: Mechanism to ensure only one thread can access a shared resource at a time, preventing race conditions.

- **Monitors** (Intrinsic Locks): Internal locks associated with every object in Java, used to implement synchronization.

Mechanisms for Synchronization:

1. **Synchronized Methods:**
    - Declared using the synchronized keyword.
    - Acquires the intrinsic lock of the object on which the method is called.
    - Only one thread can execute a synchronized method of a given object at a time.

Java Synchronization is used to make sure by some synchronization method that only one thread can access the resource at a given point in time.

Java Synchronized Blocks

Java provides a way of creating threads and synchronizing their tasks using synchronized blocks.

A synchronized block in Java is synchronized on some object. All synchronized blocks synchronize on the same object and can only have one thread executed inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
public class Printer {
    synchronized void printDocuments(int num) {
        for (int i = 1; i <= num; i++) {
            System.out.println("Printing document " + i);
            try {
                Thread.sleep(500); // Simulate document printing time
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

11. Apply the concept of thread to reserve berth in railway reservation system.

Ans. Typically, we can define threads as a subprocess with lightweight with the smallest unit of processes and also has separate paths of execution. These threads use shared memory but they act independently hence if there is an exception in threads that do not affect the working of other threads despite them sharing the same memory.

```
public class TrainTicket implements Runnable {
   private int totalBerthsAvailable;
   private int berthsRequested;
   public TrainTicket(int totalBerthsAvailable) {
      this.totalBerthsAvailable = totalBerthsAvailable;
   }
   @Override
   public void run() {
      synchronized (this) {
         if (totalBerthsAvailable > 0) {
            System.out.println(Thread.currentThread().getName() + " reserving a berth");
            try {
               Thread.sleep(2000); // Simulate processing time
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
            totalBerthsAvailable--;
            System.out.println(berthsRequested + " berth reserved for " +
Thread.currentThread().getName());
         } else {
            System.out.println("No berths available for " + Thread.currentThread().getName());
         }
      }
   }
}

public class RailwayReservationSystem {
   public static void main(String[] args) {
      TrainTicket ticket = new TrainTicket(1); // Only one berth available

      Thread passenger1 = new Thread(ticket, "Passenger 1");
```

```
        Thread passenger2 = new Thread(ticket, "Passenger 2");


        passenger1.start();

        passenger2.start();
    }
}
```

- TrainTicket class:
    - Represents a train ticket with available berths.
    - Implements Runnable to allow multiple threads to access it.
    - Uses a synchronized block to ensure only one thread can access the totalBerthsAvailable variable at a time.
    - Simulates processing time using Thread.sleep().
- RailwayReservationSystem class:
    - Creates a TrainTicket object with one available berth.
    - Creates two threads representing passengers trying to book the same berth.
    - Starts both threads to simulate concurrent booking attempts.

12. Explain threads lifecycle in detail.

Ans.

There are different states Thread transfers into during its lifetime, let us know about those states in the following lines: in its lifetime, a thread undergoes the following states, namely:

1. New State
2. Active State
3. Waiting/Blocked State
4. Timed Waiting State
5. Terminated State

1. New State

By default, a Thread will be in a new state, in this state, code has not yet been run and the execution process is not yet initiated.

2. Active State

A Thread that is a new state by default gets transferred to Active state when it invokes the start() method, his Active state contains two sub-states namely:

- Runnable State: In This State, The Thread is ready to run at any given time and it's the job of the Thread Scheduler to provide the thread time for the runnable state preserved threads. A program that has obtained Multithreading shares slices of time intervals which are shared between threads hence, these threads run for some short span of time and wait in the runnable state to get their schedules slice of a time interval.
- Running State: When The Thread Receives CPU allocated by Thread Scheduler, it transfers from the "Runnable" state to the "Running" state. and after the expiry of its given time slice session, it again moves back to the "Runnable" state and waits for its next time slice.

3. Waiting/Blocked State

If a Thread is inactive but on a temporary time, then either it is a waiting or blocked state, for example, if there are two threads, T1 and T2 where T1 needs to communicate to the camera and the other thread T2 already using a camera to scan then T1 waits until T2 Thread completes its work, at this state T1 is parked in waiting for the state, and in another scenario, the user called two Threads T2 and T3 with the same functionality and both had same time slice given by Thread Scheduler then both Threads T1, T2 is in a blocked state. When there are multiple threads parked in a Blocked/Waiting state Thread Scheduler clears Queue by rejecting unwanted Threads and allocating CPU on a priority basis.

4. Timed Waiting State

Sometimes the longer duration of waiting for threads causes starvation, if we take an example like there are two threads T1, T2 waiting for CPU and T1 is undergoing a Critical Coding operation and if it does not exist the CPU until its operation gets executed then T2 will be exposed to longer waiting with undetermined certainty, In order to avoid this starvation situation, we had Timed Waiting for the state to avoid that kind of scenario as in Timed Waiting, each thread has a time period for which sleep() method is invoked and after the time expires the Threads starts executing its task.

5. Terminated State

A thread will be in Terminated State, due to the below reasons:

- Termination is achieved by a Thread when it finishes its task Normally.
- Sometimes Threads may be terminated due to unusual events like segmentation faults, exceptions…etc. and such kind of Termination can be called Abnormal Termination.
- A terminated Thread means it is dead and no longer available.

13. Explain in detail Thread priorities.

Ans.

Priority:

- A mechanism to influence the scheduling of threads by the JVM.
- Assigned an integer value between 1 (lowest) and 10 (highest).
- Higher priority threads are generally favored for execution.

Constants:

Thread.MIN_PRIORITY = 1

Thread.NORM_PRIORITY = 5 (default)

Thread.MAX_PRIORITY = 10

thread.setPriority(Thread.MAX_PRIORITY);

- set priorities judiciously, primarily for long-running, CPU-bound tasks.
- Focus on synchronization and coordination for multithreading correctness.
- Consider alternative concurrency mechanisms like thread pools for managing tasks effectively.
- Experiment with priorities to see their impact in your specific application context.

- Overuse can lead to scheduling issues: Too many high-priority threads can starve lower-priority ones.
- Not a replacement for proper synchronization: Doesn't prevent race conditions or deadlocks.
- May not have a significant impact in I/O-bound tasks: Threads often wait for I/O regardless of priority.

14. Implement a program that creates 3 threads?

Ans.

```
public class ThreeThreadsDemo {
   public static void main(String[] args) {
      // Create thread objects using the Runnable interface
      Runnable thread1 = () -> {
```

```
    for (int i = 0; i < 5; i++) {
        System.out.println("Thread 1: Message " + i);
        try {
            Thread.sleep(1000); // Delay for 1 second
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
};
Runnable thread2 = () -> {
    for (int i = 0; i < 5; i++) {
        System.out.println("Thread 2: Hello " + i);
        try {
            Thread.sleep(2000); // Delay for 2 seconds
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
};
Runnable thread3 = () -> {
    for (int i = 0; i < 5; i++) {
        System.out.println("Thread 3: Welcome " + i);
        try {
            Thread.sleep(3000); // Delay for 3 seconds
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
};
// Create and start the threads
new Thread(thread1).start();
```

```
        new Thread(thread2).start();
        new Thread(thread3).start();
    }
}
```

15. Implement a program to throw a user defined exception“String Mismatch” if two strings are not equal.

Ans.

```
class StringMismatchException extends Exception {
    public StringMismatchException(String message) {
        super(message);
    }
}
public class StringComparison {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the first string: ");
        String str1 = scanner.nextLine();
        System.out.print("Enter the second string: ");
        String str2 = scanner.nextLine();
        try {
            if (!str1.equals(str2)) {
                throw new StringMismatchException("Strings are not equal!");
            } else {
                System.out.println("Strings are equal.");
            }
        } catch (StringMismatchException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

16. Explain how threads acting on same object are synchronized.

Ans.

- **Intrinsic Locks (Monitors):** Every object in Java has an intrinsic lock, also known as a monitor. This lock acts as a gatekeeper for accessing the object's state.
- **Synchronized Blocks:**
    - Enclose critical code sections within synchronized(object) blocks.
    - When a thread enters a synchronized block, it acquires the lock of the specified object.
    - Other threads attempting to enter a synchronized block using the same object are blocked until the lock is released.
- **Synchronized Methods:**
    - Declared using the synchronized keyword.
    - Implicitly acquire the lock of the object on which they are called.
- **Reentrant Locks:**
    - A thread can acquire the same lock multiple times without blocking itself.
    - This allows for nested synchronized blocks or recursive calls to synchronized methods.

Mechanism:

1. **Thread Enters Synchronized Block:**
    - A thread attempts to enter a synchronized block or call a synchronized method on an object.
2. **Lock Acquisition:**
    - If the lock is available, the thread acquires it.
    - If the lock is already held by another thread, the current thread is blocked and placed in a waiting queue.
3. **Exclusive Access:**
    - The thread holding the lock has exclusive access to the object's state within the synchronized block or method.
4. **Lock Release:**
    - When the thread exits the synchronized block or method (either normally or through an exception), it releases the lock.
    - The JVM then selects a thread from the waiting queue (if any) to acquire the lock and proceed.

17. Explain use of threads in multitasking.

Ans.

Multi-tasking is the ability of an operating system to run multiple processes or tasks concurrently, sharing the same processor and other resources. In multi-tasking, the operating system divides the CPU time between multiple tasks, allowing them to execute simultaneously. Each task is assigned a time slice, or a portion of CPU time, during which it can execute its code. Multi-tasking is essential for increasing system efficiency, improving user productivity, and achieving optimal resource utilization.

Benefits:

Improved performance: By running independent tasks concurrently, you can better utilize CPU resources, especially on multi-core systems.

Enhanced responsiveness: Applications can remain responsive to user interactions even while performing other tasks in separate threads.

Improved resource utilization: Threads can share resources within the process, reducing overhead compared to creating multiple processes.

18. Explain Messaging.

Ans.

19. Explain Built-in exceptions, and Chained exceptions.

Ans.

Built-in exceptions :

- Java provides a collection of predefined exception classes that represent common errors or exceptional situations that can occur during program execution.
- These exceptions are organized in a hierarchy, with the root class being Throwable.
- Built-in exceptions are used to signal errors and allow for structured error handling using try-catch blocks.

Examples of Built-in Exception:

1. Arithmetic exception : It is thrown when an exceptional condition has occurred in an arithmetic operation.

2. ArrayIndexOutOfBounds Exception : It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

3. ClassNotFoundException : This Exception is raised when we try to access a class whose definition is not found.

4. FileNotFoundException : This Exception is raised when a file is not accessible or does not open.

5. IOException : It is thrown when an input-output operation failed or interrupted

6. NoSuchMethodException : t is thrown when accessing a method which is not found.

Chained exceptions :

Chained Exceptions allows to relate one exception with another exception, i.e. one exception describes cause of another exception. For example, consider a situation in which a method throws an Arithmetic Exception because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw only Arithmetic Exception to the caller. So the caller would not come to know about the actual cause of exception. Chained Exception is used in such type of situations. Constructors Of Throwable class **Which support chained** exceptions in java :

1. **Throwable(Throwable cause) :-** Where cause is the exception that causes the current exception.
2. **Throwable(String msg, Throwable cause) :-** Where msg is the exception message and cause is the exception that causes the current exception.

Methods Of Throwable class Which support chained exceptions in java :

1. **getCause() method :-** This method returns actual cause of an exception.
2. **initCause(Throwable cause) method :-** This method sets the cause for the calling exception.

```java
// Java program to demonstrate working of chained exceptions
public class ExceptionHandling
{
    public static void main(String[] args)
    {
        try
        {
            // Creating an exception
            NumberFormatException ex =
                       new NumberFormatException("Exception");

            // Setting a cause of the exception
            ex.initCause(new NullPointerException(
                      "This is actual cause of the exception"));

            // Throwing an exception with cause.
            throw ex;
        }

        catch(NumberFormatException ex)
        {
            // displaying the exception
            System.out.println(ex);

            // Getting the actual cause of the exception
            System.out.println(ex.getCause());
        }
    }
}
```

20. Differentiate between checked and unchecked exceptions

Ans.

Checked Exceptions

- They occur at compile time.
- The compiler checks for a checked exception.
- These exceptions can be handled at the compilation time.
- It is a sub-class of the exception class.
- The JVM requires that the exception be caught and handled.
- Example of Checked exception- 'File Not Found Exception'

Unchecked Exceptions

- These exceptions occur at runtime.
- The compiler doesn't check for these kinds of exceptions.
- These kinds of exceptions can't be caught or handled during compilation time.
- This is because the exceptions are generated due to the mistakes in the program.
- These are not a part of the 'Exception' class since they are runtime exceptions.
- The JVM doesn't require the exception to be caught and handled.
- Example of Unchecked Exceptions- 'No Such Element Exception'

Unit 5

1. **What is a Lambda function? Explain with example.**

Ans.

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

Lambda Expressions implement the only abstract function and therefore implement functional interfaces lambda expressions are added in Java 8 and provide the below functionalities.

- **Enable to treat functionality as a method argument, or code as data.**
- **A function that can be created without belonging to any class.**
- **A lambda expression can be passed around as if it was an object and executed on demand.**

There are three Lambda Expression Parameters are mentioned below:

1. **Zero Parameter**

No input data needed for the lambda's operation.

() -> expression

2. **Single Parameter**

Lamda takes one input value.

(parameter) -> expression

3. **Multiple Parameters**

Lambda takes multiple input value.

(parameter1, parameter2, ...) -> expression

Example :

```
// Java program to demonstrate lambda expressions
// to implement a user defined functional interface.
// A sample functional interface (An interface with
// single abstract method
interface FuncInterface
{
        // An abstract function
        void abstractFun(int x);

        // A non-abstract (or default) function
        default void normalFun()
        {
        System.out.println("Hello");
        }
}
class Test
{
        public static void main(String args[])
        {
                // lambda expression to implement above
                // functional interface. This interface
```

```
        // by default implements abstractFun()
        FuncInterface fobj = (int x)->System.out.println(2*x);

        // This calls above lambda expression and prints 10.
        fobj.abstractFun(5);
    }
}
```

2. Use arithmetic operators in Lisp.

Ans.

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, and division.

Operator	Syntax	Description
Addition Operator(+)	+ num1 num2	Add the two numbers
Subtraction Operator(-)	– num1 num2	Subtract the second number from the first number
Multiplication(*)	* num1 num2	Multiply two numbers
Division(/)	/ num1 num2	Divide the two numbers
Modulus(mod)	mod num1 num2	Get the remainder of two numbers
Increment(incf)	incf num value	Increment number by given value
Decrement(decf)	decf num value	Decrement number by given value

```
;set value 1 to 300
; set value 2 to 600
(setq val1 300)
(setq val2 600)

;addition operation
(print (+ val1 val2))

;subtraction operation
(print (- val1 val2))

;multiplication operation
(print (* val1 val2))

;division operation
(print (/ val1 val2))

;modulus operation
(print (MOD val1 val2))

;increment a by 10
(print (incf val1 val2))

;decrement b by 20
(print (decf val1 val2))
```

3. How rules are defined in Lisp.

Ans.

Macros:

- Code transformations that create new code at compile time.
- Used to create specialized syntax for rule-like constructs.
- Common Lisp's defmacro and Scheme's syntax-rules offer macro systems.
- Rules as First-Class Objects: Rules are directly represented as data structures (lists or s-expressions), making them flexible and manipulatable.
- Pattern Matching: Rules match patterns in data using cond or if expressions.
- Unification: New values can be bound to variables during matching, enabling flexible reasoning.
- Dynamic Evaluation: Rules are evaluated at runtime, allowing for adaptive behavior.

Conditionals:

- if, cond, and case constructs for decision-making.
- Can implement basic rule-like behavior

3. Logic Programming Libraries:

- Third-party libraries like MiniKanren and Core.Logic for Prolog-like rule-based programming within Lisp.

4. Pattern Matching:

- Some Lisp dialects offer pattern matching features for rule-like code.
- Check the specific dialect for availability.

5. Function Composition and Higher-Order Functions:

- Combine functions to create rules implicitly.
- Apply functions conditionally based on arguments.

Java:

- Represent Rules with Classes and Methods: Rules are often modeled as classes with methods for pattern matching and execution.
- Conditional Statements and Objects: Use if statements and object-oriented constructs for pattern matching and rule execution.
- No Built-in Unification: Implement unification-like behavior manually or using external libraries.

- Static Typing: Constraints rule flexibility but can enhance type safety and maintainability.

Approaches for Rule-Based Systems in Java:

1. Rule Engines: External libraries or frameworks like Drools or JBoss Rules explicitly designed for rule-based systems.
2. Custom Implementations: Develop rule representations and execution mechanisms using Java's core OOP features.
3. Decision Tables: Model rules in tabular form, often used for business rules and decision automation.

Key Differences:

- Dynamic vs. Static: Lisp's dynamic nature allows for more adaptable rule execution, while Java's static typing provides stronger type safety.
- Unification: Lisp's built-in unification enables pattern matching with variable binding, while Java requires manual implementation or external libraries.
- First-Class Rules: Lisp's direct representation of rules as data structures offers more flexibility in manipulation and reasoning.

Choosing the Right Approach:

- Complexity of Rules: For complex rule systems with frequent changes, a dedicated rule engine might be more suitable.
- Integration Requirements: If rules need to be tightly integrated with existing Java code, custom implementations might be better.
- Domain Specificity: Decision tables can be effective for representing business rules or decision logic in a structured format.

4. Demonstrate with an example how to code in LISP?

Ans.

```
(+ 2 3)   ; Returns 5

(list 1 2 "hello")  ; Creates a list

'(a b c)  ; Quoted list
```

Defining Functions:

```
(defun square (x) (* x x))

(square 4)  ; Returns 16
```

Conditionals:

- if for simple conditionals.
- cond for multiple conditions

```
(if (> x 5) "greater" "lesser")
(cond ((= x 0) "zero")
      ((> x 0) "positive")
      (t "negative"))
```

iteration:

- **loop for general iteration.**
- **dolist and dotimes for list and number iterations.**

```
(loop for i from 1 to 5
      do (print i))
```

Recursion:

```
(defun factorial (n)
  (if (= n 0) 1
      (* n (factorial (- n 1)))))
```

Lists and Cons Cells:

- **Building blocks of Lisp data.**
- **cons to create pairs (first and rest).**
- **car and cdr to access elements.**

```
(cons 1 (cons 2 nil))  ; Returns (1 2)
(car '(a b c))         ; Returns a
(cdr '(a b c))         ; Returns (b c)
```

5. What is the significance of first-class functions in Lisp, and how do they support functional programming paradigms?

Ans.

First-Class Functions in Lisp:

- **Treated as first-class citizens, meaning they can be:**
    - Assigned to variables
    - Passed as arguments to other functions
    - Returned as values from functions
    - Stored in data structures
- This makes functions incredibly versatile building blocks in Lisp.

Higher-Order Functions:

Functions that operate on other functions, promoting code reusability and abstraction.

Examples: map, filter, reduce, apply, compose

Function Composition:

Combining simpler functions to create more complex ones, leading to modular and expressive code.

Lambda Expressions:

Creating anonymous functions inline, often used with higher-order functions.

Closures:

Functions that capture their surrounding environment, enabling powerful techniques like partial application.

Declarative Programming:

Describing "what" to do rather than "how" to do it.

Emphasizes problem-solving through function composition and higher-order functions, often leading to more concise and elegant code.

Recursion:

Functions that call themselves, often used in Lisp due to its functional nature, allowing for elegant solutions to problems like tree traversal or list processing.

- First-class functions are a cornerstone of Lisp's functional programming paradigm.
- They enable powerful techniques like higher-order functions, function composition, lambda expressions, closures, and declarative programming.
- They promote code reusability, abstraction, and elegant problem-solving.
- Lisp's focus on first-class functions has influenced many modern programming languages, contributing to the growing popularity of functional programming concepts.

6. Write a LISP expression using nums, filter, and prime whichis the list of prime numbers in the range 1..100.

Ans.

: (filter 'prime (nums 1 100))

1. **nums 1 100 creates a list of numbers from 1 to 100.**
2. **filter 'prime ... applies the prime function (which presumably checks for primality) to each element of the list, keeping only those elements that return true (i.e., the prime numbers).**
3. **The result is a list containing only the prime numbers within the specified range.**

7. List Features of LISP.

Ans.

1. Homoiconicity:

- **Code and data are represented in the same way (as lists).**
- This enables powerful metaprogramming capabilities, allowing code to create and modify code.

2. Dynamic Typing:

- Data types are determined at runtime, providing flexibility but requiring careful attention to type errors.

3. First-Class Functions:

- Functions can be treated as data, passed as arguments, returned from functions, and stored in data structures.
- This foundation for functional programming enables techniques like higher-order functions, function composition, and recursion.

4. Symbolic Expressions:

- Lisp manipulates symbolic expressions, making it well-suited for tasks involving symbolic manipulation, such as artificial intelligence and natural language processing.

5. List-Based Data Structures:

- Lists are the primary data structure in Lisp, used for representing both code and data.
- Cons cells form the building blocks of lists, enabling the creation of nested and recursive structures.

6. Garbage Collection:

- Automatic memory management frees programmers from manual memory allocation and deallocation.

7. Interactive Read-Eval-Print Loop (REPL):

- A feature common to most Lisp implementations that allows for interactive code evaluation and experimentation, enhancing development and debugging.

8. S-Expressions:

- Lisp code is written as S-expressions (symbolic expressions), a simple and uniform syntax using parentheses for structure.

9. Macros:

- A powerful code transformation facility for creating custom syntax and extending the language.

10. Dialects:

- Lisp has multiple dialects, each with its own strengths and domains, including Common Lisp, Scheme, Emacs Lisp, Clojure, and more.

8. What are 3 elements of Prolog?

Ans.

1. Facts: These are basic truths or statements about the world that the program believes to be true. They are written in the form of subject-predicate-object triplets, like father(john, mary), which signifies that "John is the father of Mary." Facts form the initial knowledge base used for reasoning.

2. Rules: These represent logical implications or relationships between facts. They are written in the form of "head ← body," where the head is the conclusion and the body is a logical expression composed of one or more facts or other rules. For example, parent(X, Y) ← father(X, Y). This rule defines that "X is a parent of Y" if "X is the father of Y."

3. Queries: These are questions posed to the Prolog program, often seeking to prove or find something based on the existing knowledge base. They are written in the same format as facts or the head of a rule. For example, brother(john, peter)? asks "Is John the brother of Peter?" The program tries to match the query with existing facts or infer it using rules and facts.

These three elements work together in Prolog:

- **Facts:** provide the initial information.
- **Rules:** connect and interpret the facts.
- **Queries:** drive the process of reasoning and retrieving information.

By combining these elements, Prolog programs can perform logic-based reasoning, draw conclusions, and answer complex questions based on the knowledge base.

9. Explain Facts in Prolog.

Ans.

Facts in Prolog: Building Blocks of Knowledge

Facts are the foundation of any Prolog program. They represent unconditionally true statements about the world within the program's domain. Think of them as the bricks from which you build the knowledge base

Structure:

- Facts are written as subject-predicate-object triplets, separated by commas and enclosed in parentheses.
- The subject refers to the entity about which the statement is made.
- The predicate describes the relationship between the subject and the object.
- The object specifies additional information about the subject and the predicate.

Example:

- parent(john, mary). - This fact states that john is the parent of mary.
- age(jane, 25). - This fact states that jane is 25 years old.
- likes(tom, ice_cream). - This fact states that tom likes ice cream.

Key Points:

- Facts are static - they remain true once defined and cannot be changed.
- Facts represent ground truths - they are always assumed to be true within the program's context.
- Facts are declarative - they simply state information without specifying how it should be used.
- Multiple facts about the same subject can be defined, building on the understanding of that entity.

Benefits of Facts:

- Simple and intuitive: The straightforward structure makes them easy to understand and write.
- Modular: Facts can be easily added or removed without affecting other parts of the program.
- Scalable: Large knowledge bases can be built by accumulating numerous facts.

10. What is LISP? Give an example of some of the popular applications built in LISP.

Ans.

LISP (List Processing) is a family of programming languages with a long history and distinctive features

- **Homoiconicity:** Code and data are represented in the same way (as lists), enabling powerful metaprogramming capabilities**.**
- **Dynamic Typing:** Data types are determined at runtime, providing flexibility but requiring attention to type errors.
- **First-Class Functions:** Functions can be treated as data, enabling techniques like higher-order functions, function composition, and recursion.
- **List-Based Data Structures**: Lists are the primary data structure, used for code and data.
- **Symbolic Expressions:** LISP excels at manipulating symbolic expressions, making it suitable for tasks involving symbols and logic.
- **Interactive Read-Eval-Print Loop (REPL):** Most LISP implementations offer interactive code evaluation and experimentation.

Popular Applications Built in LISP:

- **Artificial Intelligence:**
    - Early AI systems like SHRDLU and ELIZA were written in LISP.
    - It's still used in AI research for symbolic reasoning, knowledge representation, and problem-solving.
- **Natural Language Processing (NLP):**
    - LISP's ability to manipulate symbols makes it well-suited for NLP tasks like language translation and text analysis.
- **Expert Systems:**
    - Rule-based systems that capture expert knowledge in a specific domain were often built in LISP.
- **Computer Algebra Systems:**
    - Symbolic mathematical manipulation systems like Macsyma and Maxima were written in LISP.
- **Game Development:**
    - **Game engines like Unity have LISP-like scripting languages for game logic.**
- **Web Development:**
    - LISP dialects like Clojure are used for web development, leveraging its functional programming paradigm for building scalable and maintainable web applications.
    -

11. How recursion is achieved in Lisp.

Ans.

1. Function Calls Itself:

- A recursive function directly calls itself within its body, creating a loop that continues until a base case is reached.

2. Base Case:

- A crucial condition that stops the recursion. It's a problem instance that the function can solve directly without further calls.

3. Breaking Down Problems:

- Recursive functions work by breaking down complex problems into smaller, self-similar subproblems. Each recursive call solves a smaller part of the problem.

4. Building Up Solutions:

- The solutions to the subproblems are combined to create the final solution to the original problem.

```
(defun factorial (n)
 (if (= n 0)
   1
   (* n (factorial (- n 1)))))
```

- **Natural fit for Lisp:** Lisp's list-based structure and functional nature make it well-suited for recursion.
- **Tail Recursion Optimization:** Many Lisp implementations optimize tail calls, allowing for efficient recursion without stack overflow issues.
- **Common Use Cases:** Recursive algorithms are often used in Lisp for tree traversal, list processing, mathematical functions, and problem-solving involving divide-and-conquer strategies.

12. Which are the Basic building blocks of Lisp.

Ans.

1. **Atoms:**
    - Fundamental units of data that cannot be further decomposed.
    - Examples: numbers (e.g., 42), symbols (e.g., 'hello), strings (e.g., "world").
2. **Lists:**
    - Ordered sequences of elements, enclosed in parentheses.

- Can contain any combination of atoms and other lists, forming nested structures.
- Examples: (1 2 3), ('a . ('b . 'c)), ("hello" "world").

3. **S-Expressions (Symbolic Expressions):**
   - General representation for both code and data in Lisp.
   - All Lisp code is written as S-expressions, creating a uniform syntax.
4. **Cons Cells:**
   - Fundamental building blocks of lists.
   - Each cons cell has two parts: a car (first element) and a cdr (rest of the list).
   - Cons cells link together to form lists of any length.
5. **Symbols:**
   - Named entities used to represent variables, function names, and other identifiers.
   - Examples: x, my-function, +, car, cdr.
6. **Functions:**
   - First-class citizens in Lisp, meaning they can be treated as data:
     - Assigned to variables
     - Passed as arguments to other functions
     - Returned as values from functions
     - Stored in data structures
7. **Lambda Expressions:**
   - Anonymous functions created inline, often used with higher-order functions.
8. **Macros:**
   - Code transformations that create new code at compile time, offering a powerful metaprogramming facility for extending the language.

13. Explain features of PROLOG Language?

Ans.

1. Logic Programming Paradigm:

- Prolog stands for "Programming in Logic."
- It focuses on describing relationships and rules rather than a sequence of steps to execute.
- Programs are written as a set of facts and rules, forming a knowledge base.

- Computation is driven by queries, asking the program to prove or find something based on the knowledge base.

2. Declarative Programming:

- Prolog programs state what is true, not how to compute it.
- This contrasts with imperative languages that focus on instructions.
- Declarative style often leads to more concise and readable code.

3. Facts and Rules:

- Facts represent unconditional truths about the world (e.g., parent(john, mary).).
- Rules express logical implications (e.g., grandparent(X, Z) :- parent(X, Y), parent(Y, Z).).

4. Unification:

- The core mechanism for matching patterns and solving goals.
- It's the process of finding substitutions for variables that make two expressions identical.
- Prolog uses unification to find solutions to queries and apply rules.

5. Backtracking:

- A search algorithm for exploring multiple possible solutions.
- Prolog backtracks when a rule fails, trying alternative paths to find a solution.

6. Non-Determinism:

- Prolog programs can produce multiple valid solutions for a query.
- This is useful for tasks like generating possibilities or exploring different scenarios.

7. Predicates and Arguments:

- Prolog data is represented as predicates (relationships) with arguments.
- Predicates are like functions in other languages, but with a logical interpretation.
- Arguments represent objects involved in the relationship.

8. Lists:

- The primary data structure for representing collections of items.
- Prolog lists are similar to Lisp lists, using a head-tail structure.

9. Recursion:

- A common technique for defining predicates and solving problems in Prolog.
- Prolog excels at recursive problem-solving due to its logical nature and backtracking mechanism.

10. Built-in Predicates:

- Prolog offers a variety of built-in predicates for common operations like arithmetic, input/output, and list manipulation.

14. Write a PROLOG program to find largest number from a given List

Ans.

```
largest(List, Largest) :-
    max_of_two(List, 0, Largest).

max_of_two([], MaxSoFar, MaxSoFar).
max_of_two([Head|Tail], MaxSoFar, Largest) :-
    Head > MaxSoFar,
    max_of_two(Tail, Head, Largest).
    max_of_two([_|Tail], MaxSoFar, Largest) :-
    max_of_two(Tail, MaxSoFar, Largest).
```

15. Explain following predicates in LISP

•Atom :

- **Purpose:** Tests if an expression is an atom (a fundamental unit of data that cannot be further decomposed).
- **Returns:** t (true) if the expression is an atom, nil (false) otherwise.
- **Examples:**
    - (atom 'symbol) -> t
    - (atom 42) -> t
    - (atom "string") -> t
    - (atom (1 2 3)) -> nil

•Equal :

- **Purpose:** Tests if two expressions are equal in value.
- **Returns:** t if the expressions are equal, nil otherwise.
- **Notes:**
    - Compares values, not object identity.

    - Uses deep comparison for lists and structures.
- **Examples:**
    - (equal 5 5) -> t
    - (equal 'a 'a) -> t
    - (equal "hello" "hello") -> t
    - (equal (1 2) (1 2)) -> t
    - (equal (1 2) '(1 2)) -> nil (different data types)

- **Evenp :**
    - **Purpose**: Tests if a number is even.
    - **Returns**: t if the number is even, nil if odd.
    - **Example**:
        - (evenp 4) -> t

- **Listp :**
    - **Purpose:** Tests if an expression is a list.
    - **Returns:** t if the expression is a list, nil otherwise.
    - **Example:**
        - (listp '(1 2 3)) -> t

- **Numberp :**
    - **Purpose:** Tests if an expression is a number.
    - **Returns:** t if the expression is a number, nil otherwise.
    - **Example:**
        - (numberp 3.14) -> t