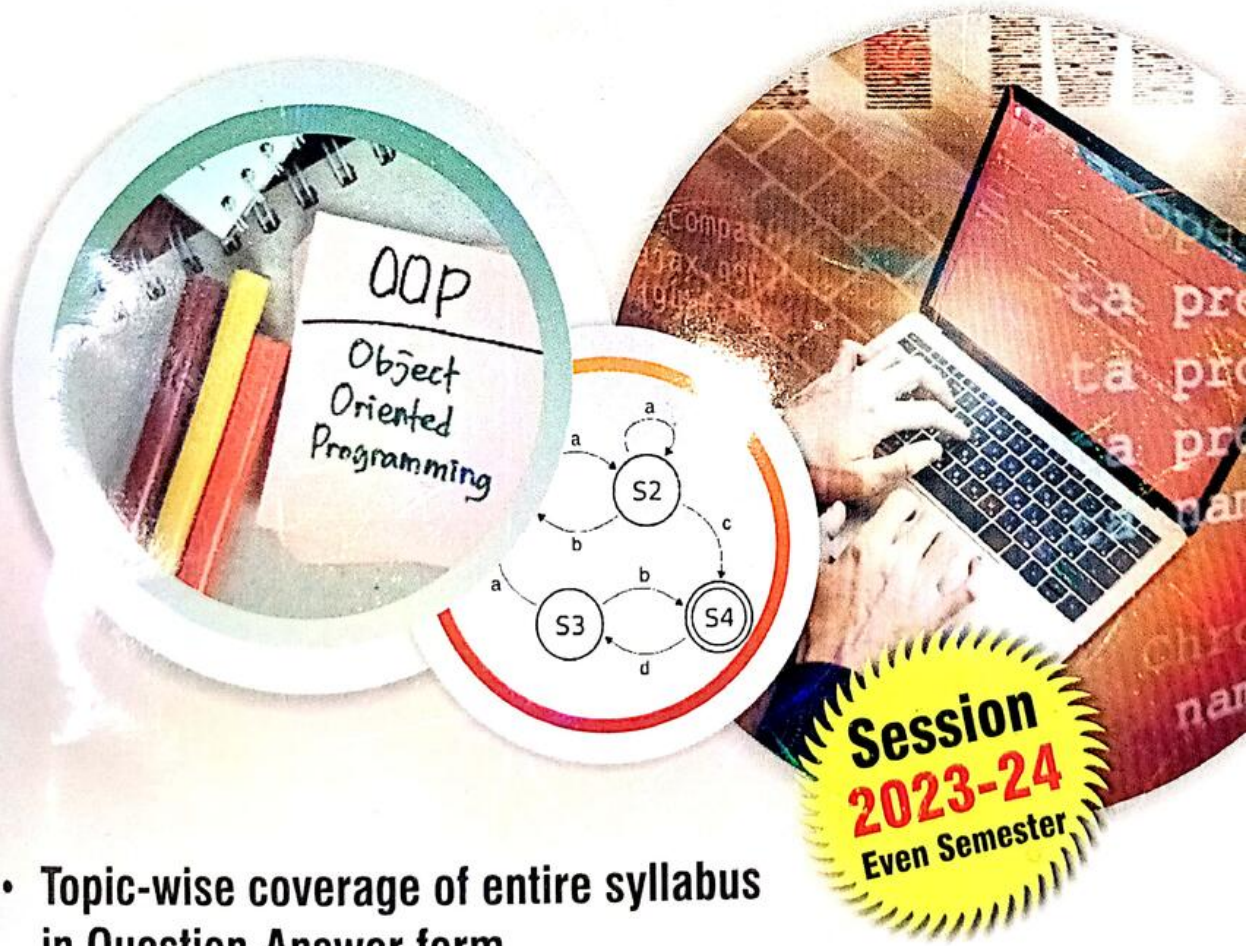
Quantum Series®

ENGINEERING

QUANTUM *Series*

Sem - 4 | CSE/IT & Allied Branches

Object Oriented Programming with Java



- Topic-wise coverage of entire syllabus in Question-Answer form.
- Short Questions (2 Marks)

CONTENTS

BCS 403 : Object Oriented Programming with Java

UNIT-3 : JAVA NEW FEATURES

(3–1 F to 3–23 F)

Functional Interfaces, Lambda Expression, Method References, Stream API, Default Methods, Static Method, Base64 Encode and Decode, ForEach Method, Try-with-resources, Type Annotations, Repeating Annotations, Java Module System, Diamond Syntax with Inner Anonymous Class, Local Variable Type Inference, Switch Expressions, Yield Keyword, Text Blocks, Records, Sealed Classes.

UNIT-4 : JAVA COLLECTIONS FRAMEWORK

(4–1 F to 4–20 F)

Collection in Java, Collection Framework in Java, Hierarchy of Collection Framework, Iterator Interface, Collection Interface, List Interface, ArrayList, LinkedList, Vector, Stack, Queue Interface, Set Interface, HashSet, LinkedHashSet, SortedSet Interface, TreeSet, Map Interface, HashMap Class, LinkedHashMap Class, TreeMap Class, Hashtable Class, Sorting, Comparable Interface, Comparator Interface, Properties Class in Java.

UNIT-5 : SPRING FRAMEWORK & SPRING BOOT

(5–1 F to 5–32 F)

Spring Framework: Spring Core Basics-Spring Dependency Injection concepts, Spring Inversion of Control, AOP, Bean Scopes- Singleton, Prototype, Request, Session, Application, Web Socket, Auto wiring, Annotations, Life Cycle Call backs, Bean Configuration styles.

Spring Boot: Spring Boot Build Systems, Spring Boot Code Structure, Spring Boot Runners, Logger, BUILDING RESTFUL WEB SERVICES, Rest Controller, Request Mapping, Request Body, Path Variable, Request Parameter, GET, POST, PUT, DELETE APIs, Build Web Applications.

SHORT QUESTIONS

(SQ–1 F to SQ–18 F)

1

UNIT

Introduction

CONTENTS

PART-1

Why Java, History of Java.

Que 1.1. What is Java and why is it used ? What are some of the key characteristics and features that distinguish Java from other programming language ?

Answer

A. Java :

1. Java is a versatile, high-level, object-oriented programming language developed by Sun Microsystems.
2. It is a general-purpose programming language made for developers to "write once, run anywhere" Java codes.
3. Java code can run on all platforms that support Java.

B. Why Java is used : Java is used in wide range of applications because :

1. Java has rich API.
2. Java is platform independent.
3. Java has great collection of open source libraries.
4. Java is robust because it utilizes strong memory management.

C. Key characteristics and features that distinguish Java :

1. **Platform independence :** Java programs are compiled into bytecode, which is a platform-independent intermediate representation.
2. **Object-oriented :** Java is a pure object-oriented programming language, which promotes modular and reusable code through classes and objects.
3. **Robust and secure :** Java has features like strong type checking, automatic memory management, and built-in security mechanisms. This makes Java applications less prone to programming errors and security vulnerabilities.
4. **Rich standard library :** Java includes a comprehensive standard library with classes and APIs.
5. **Multithreading :** Java provides built-in support for multithreading, allowing developers to write concurrent and scalable applications.
6. **Exception handling :** Java has a robust exception-handling mechanism that simplifies the management of errors and unexpected situations in code.

Que 1.2. **How does Java achieve platform independence and why is this a significant feature ?**

Answer

1. Java achieves platform independence through the use of Java Virtual Machine (JVM).
2. Java uses JVM to execute its code.
3. When a Java program is compiled, it is compiled into bytecode.
4. This bytecode is then executed by the JVM, which is platform-specific.
5. Each platform has its own implementation of the JVM, which understands and translates bytecode to machine code that can run on that specific platform.

Java's platform independence is significant because of following advantages :

1. Cross-platform compatibility.
2. Reduced development time and costs.
3. Simplified maintenance and updates.
5. Enhanced security and portability.

Que 1.3. **What is the historical background and origin of the Java programming language ?**

Answer

Following is a brief overview of its development and origins :

1. **Sun Microsystems :** The origin of Java can be traced back to Sun Microsystems. Sun recognized the need for a programming language that could provide platform independence and be used in a networked environment.
2. **Oak :** The project, originally called "Oak," was led by James Gosling and his team.
3. **Renaming to Java :** In 1995, Oak was renamed to "Java" due to trademark issues.
4. **Introduction of Applets :** Java applets were a way to bring interactivity to the early web (early 1995). It allowed developers to create small, interactive applications (applets) that could be embedded in web pages.
5. **Java 1.0 :** In 1996, Java 1.0 was released, and it included a complete set of core libraries and features for developing applications.
6. **Sun's Open Approach :** Sun Microsystems took an open approach to Java. This approach helped Java gain widespread adoption and contributed to its "Write Once, Run Anywhere" capability.

7. **Acquisition by Oracle :** In 2010, Oracle Corporation acquired Sun Microsystems, including the rights to Java. Oracle has continued to maintain and develop the Java platform since then.

PART-2

JVM, JRE, Java Environment.

Que 1.4. **What is Java virtual machine (JVM) ? Describe its architecture. What is its primary role in the execution of programs ?**

Answer

A. JVM : JVM is an abstract machine. It is a software-based, virtual computer that provides runtime environment in which java bytecode can be executed.

B. Architecture : Fig. 1.4.1 shows the internal architecture of JVM.

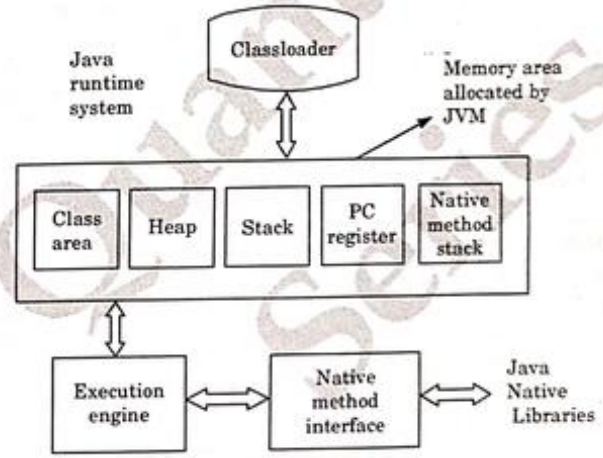


Fig. 1.4.1.

JVM contains the following :

1. **Classloader :** Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader.
2. **Class area :** Class area stores per-class structures such as the runtime constant pool, field and method data.

3. **Heap :** It is the runtime data area in which objects are allocated.
4. **Stack :** Java stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.
5. **Program Counter (PC) register :** PC register contains the address of the Java virtual machine instruction currently being executed.
6. **Native method stack :** It contains all the native methods used in the application.
7. **Execution engine :** It contains :
   i. **A virtual processor**
   ii. **Interpreter :** Read bytecode stream then execute the instructions.
   iii. **Just-In-Time(JIT) compiler :** It is used to improve the performance. JIT compiles parts of the bytecode that have similar functionality at the same time, and hence reduces the amount of time needed for compilation.
8. **Java Native Interface :** Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc.

C. **Role of JVM :** The primary role of the Java Virtual Machine (JVM) in the execution of programs is to act as an intermediary between the compiled Java bytecode and the host operating system and hardware.

Que 1.5. What is a Java Development Kit (JDK) ? Why do we need it ?

Answer

A. **Java Development Kit (JDK) :**

1. JDK is a cross-platformed software development environment that offers a collection of tools and libraries necessary for developing Java-based software applications and applets.
2. It is a core package used in Java, along with the JVM and the JRE (Java Runtime Environment).

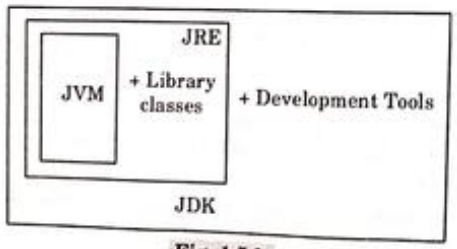


Fig. 1.5.1.

B. **Need of JDK :** The JDK is a fundamental requirement for Java developers, and it serves following purposes :

1. **Compilation :** The JDK includes the Java Compiler, which is used to compile Java source code into bytecode.
2. **Development Tools :** The JDK provides a wide range of development tools.
3. **Libraries and APIs :** The JDK includes the Java Standard Library and APIs (Application Programming Interfaces) for building various types of applications.
4. **Java Virtual Machine (JVM) :** The JDK includes the JVM, which is required for running Java applications.
5. **Compatibility :** The JDK ensures compatibility with the Java language and platform specifications.

Que 1.6. What is Java Runtime Environment (JRE) and what is its primary purpose in Java application ?

Answer

A. **Java Runtime Environment (JRE) :**

1. Java Run-time Environment (JRE) is the part of the Java Development Kit (JDK).
2. It is a software package that provides the runtime environment necessary for executing Java applications.
3. It is a freely available software distribution which has Java Class Library, specific tools, and a stand-alone JVM.

B. **Primary purpose of JRE :**

1. The primary purpose of JRE is to allow Java programs to run on a user's computer or device, regardless of the underlying hardware and operating system.
2. It provides the environment needed to interpret and execute java bytecode.

Que 1.7. How does JRE work with JVM ?

Answer

The JRE and the JVM work closely together to enable the execution of Java applications. Here's how they interact and cooperate :

1. When you write a Java program, you have to save it with .java extension.
2. After saving the program you compile your program. The output of the Java compiler is a byte-code which is platform independent.
3. After compiling, the compiler generates a .class file which has the bytecode.

4. The bytecode is platform independent and runs on any device having the JRE.
5. From here, the work of JRE begins. To run any Java program, you need JRE.
6. The flow of the bytecode to run is as follows :

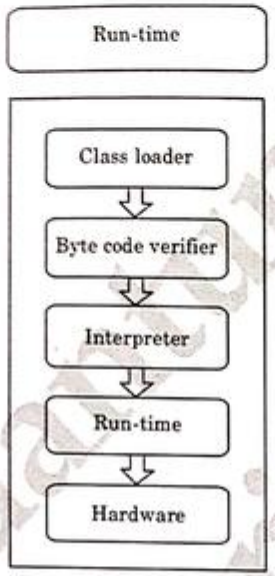


Fig. 1.7.1.

7. The following steps take place at runtime :

   Step 1 : Class Loader : At this step, the class loader loads various classes which are essential for running the program.

   Step 2 : Bytecode verifier : The code is allowed to be interpreted only when it passes the tests of the bytecode verifier which checks the format and checks for illegal code.

   Step 3 : Interpreter : Once the classes get loaded and the code gets verified, then interpreter reads the assembly code line by line and does the following two functions :

   i. Execute the bytecode.

   ii. Make appropriate calls to the underlying hardware.

8. In this way, the program runs in JRE.

Que 1.8. **Differentiate between JDK, JRE and JVM.**

Answer

S. No.	Aspect	JDK	JRE	JVM
1.	Full Form	Java Development Kit	Java Runtime Environment	Java Virtual Machine
2.	Purpose	For Java application development, including compilation and debugging.	For running Java applications.	For executing Java bytecode.
3.	Component	Java Compiler (javac)	Java Virtual Machine (JVM)	Just-In-Time (JIT) Compiler
4.	Platform Independence	No (Platform-specific)	Yes (Platform-independent)	Yes (Platform-independent)
5.	Used by Developers?	Yes	No	No
6.	Used by End Users?	No	Yes	No
7.	Needed for Compilation	Yes	No	No

Que 1.9. **What is Java development environment ? Give some of its features. Also name some of the popular Java development environments.**

Answer

A. Java development environment :

1. A Java development environment, also known as Integrated Development Environment (IDE), is a software suite that provides tools and features to facilitate the development, testing, and debugging of Java applications.
2. It is a comprehensive platform that helps Java developers create, edit, build, and manage Java code efficiently.

B. Features of Java development environment :

1. **Code editor :** A code editor is a central component of the development environment, providing a text editor for writing and editing Java source code.

2. **Compiler :** The development environment integrates the Java Compiler (javac) to compile Java source code into bytecode.
3. **Debugger :** A debugger allows developers to step through their code, set breakpoints, inspect variables, and track program execution to identify and fix issues in the code.
4. **Integrated build tools :** Many development environments integrate build tools for managing dependencies and building Java applications.

C. **Popular Java development environments :** Examples of popular Java development environments include :

1. Eclipse,
2. IntelliJ IDEA,
3. NetBeans, and
4. Oracle JDeveloper.

PART-3

Java Source Structure and Compilation.

Que 1.10. **What is Java source file ? Explain the structure of Java source file.**

Answer

A. **Java source file :**

1. A Java source file, also known as a Java source code file, contains the code written in the Java programming language.
2. The structure of a Java source file follows specific conventions and rules.

B. **Structure of Java source file :** Following explain the typical structure of a Java source file :

1. **Package declaration (Optional) :** This declaration specifies the package to which the class in the file belong. For example : package com.example.myapp;
2. **Import statements (Optional) :** After the package declaration or at the beginning of the file, you can include optional import statements. Import statements are used to avoid fully qualifying class names when you use classes from external packages. For example : import java.util.ArrayList;
3. **Class declaration :** The main body of a Java source file typically contains one or more class declarations. For example :


```
public class MyClass {
    // Class members and methods go here
}
```

4. **Main method (Optional) :** If the class is intended to be an entry point for a Java application, it contains a main method with the following signature :

```
public static void main(String[] args) {
    // Main program logic
}
```

5. **Fields and methods :** Inside the class declaration, you can define fields (variables) and methods (functions). Fields represent the data the class holds, while methods define the behavior and functionality of the class. For example :

```
public class MyClass {
    private int myField; // Field
  public void myMethod() { // Method
        // Method logic here
    }
}
```

6. **Comments :** Java source files commonly include comments to document code. Java supports both single-line and multi-line comments :

```
// This is a single-line comment
/*
* This is a multi-line comment
*/
```

Que 1.11. **Explain the steps involved in the compilation process of a Java source file.**

Answer

Following are the key steps in the compilation process of a Java source file :

1. **Writing Java source code :** The first step is to create or write Java source code using a text editor.
2. **Editing and saving :** After writing the source code, it's essential to review and edit it for correctness and maintainability.
3. **Compiling source code :** Once the Java source code is ready, you compile it using the Java compiler (javac).
4. **Bytecode generation :** The Java compiler generates bytecode instructions for the JVM from the source code.

5. **Bytecode Verification :** The JVM bytecode verifier checks the generated bytecode for correctness and security.
6. **Class loading :** The JVM's classloader loads the compiled classes into memory as they are needed.
7. **Execution :** Once the necessary classes are loaded into memory, the JVM executes the bytecode. It interprets the bytecode or compiles it into native machine code for execution.

PART-4

Programming Structures in Java : Defining Classes in Java, Constructors, Methods.

Que 1.12. What do you understand by 'object' in Java ?

Answer

1. In Java, an object is a fundamental runtime entity that represents an instance of a class.
2. Objects are a key concept in object-oriented programming (OOP).
3. Objects are central to the principles of OOP, including encapsulation, inheritance, and polymorphism.
4. They enable the modeling of real-world entities and the organization of code into manageable and reusable components.
5. In Java, objects are the building blocks of programs, and a well-designed program is often composed of numerous interacting objects.

Que 1.13. What do you understand by 'class' in Java ?

OR

Define class in Java.

Answer

1. In Java, a class is a fundamental blueprint or template for creating objects.
2. It serves as a model for defining the attributes (data) and behaviors (methods) that objects of the class will exhibit.
3. Here are some key points to understand about classes in Java :

i. **Definition :** A class in Java is defined using the 'class' keyword, followed by the class name. For example :

```
public class MyClass {
    // Class members (fields and methods) go here
}
```


ii. **Attributes (Fields) :** Within a class, you can declare attributes. These are used to store data associated with objects of the class. Fields can have different data types, such as integers, strings, etc. For example :

```
public class Person {
    String name; // A string attribute to store the name
    int age;     // An integer attribute to store the age
}
```

iii. **Methods :** A class contains methods that define the behavior of objects created from that class. Methods are functions that can perform actions and manipulate the data stored in the fields. Methods are defined within the class and can take parameters and return values. For example :

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

iv. **Access modifiers :** Java provides access modifiers like 'public', 'private', and 'protected'. Access modifiers determine whether a class member is accessible from other classes or not. For example :

```
public class MyClass {
    private int privateField; // Private field
    public void publicMethod() {
        // Public method
    }
}
```

Que 1.14. Define constructor. What are various types of constructor available in Java ?

Answer

A. Constructor :

1. A constructor is a special method in a class that is automatically called when an object of that class is created.
2. Constructors are called when an object of a class is created.
3. They are responsible for setting the initial state of the object.
4. Constructors have the same name as the class and do not have a return type.

B. Types of constructors in Java :

1. Default constructor :

i. A default constructor is automatically provided by Java if a class does not define any constructors explicitly.

ii. It takes no arguments.

iii. It initializes fields with default values (e.g., 0 for numeric types, null for reference types).

Example :

```
public class MyClass {
    // Default constructor is provided by Java
}
```

2. Parameterized constructor :

i. A parameterized constructor is defined with one or more parameters.

ii. It allows you to provide initial values when creating objects.

iii. Parameterized constructors are useful when you want to set specific values for an object's attributes during object creation.

Example :

```
public class Person {
    private String name;
    private int age;
     / Parameterized constructor
    ublic Person(String name, int age) {
    this.name = name;
        this.age = age;
    }
}
```

3. Private constructor : If a constructor is declared as private, then its objects are only accessible from within the declared class.

Example :

```
public class MyClass {
    private int myValue;
      // Private constructor
      private MyClass(int value) {
        this.myValue = value;
    }
    public static MyClass createInstance(int value) {
    return new MyClass(value);  }
```



```
    public int getMyValue() {
        return myValue;
    }
}
```

4. Copy constructor :

i. A copy constructor is used to create a new object by copying the attributes of an existing object.

ii. This constructor typically takes an object of the same class as a parameter and initializes the new object with the values of the existing object.

Example :

```
public class Point {
    private int x;
    private int y;
    // Copy constructor
        public Point(Point other) {
            this.x = other.x;
            this.y = other.y;
        }
    }
```

Que 1.15. What is 'method' in Java ? How do you define a method in Java ?

Answer

1. A method is a block of code which only runs when it is called.
2. Methods are used to perform certain actions, and they are also known as functions.
3. A method must be declared within a class. It is defined with the name of the method, followed by parentheses ().

Example : Create a method inside Main :

```
public class Main {
    static void myMethod() {
    // code to be executed
        }
            }
```

Que 1.16. Differentiate between constructors and methods in Java.

Answer

S. No.	Aspect	Constructors	Methods
1.	Purpose	Initialize objects when they are created.	Perform operations or provide behavior.
2.	Name	Has the same name as the class.	Can have any name, as long as it follows Java's naming rules.
3.	Return Type	No return type, not even 'void'.	Has a return type, which can be 'void' or any other data type.
4.	Implicit Inheritance	Constructors are not inherited by subclasses.	Methods can be inherited by subclasses.
5.	Default Constructor/ Method	A default constructor is automatically provided by Java if no constructors are defined.	No default methods are provided; they must be defined explicitly.

PART-5

Access Specifiers, Static Members, Final Members, Comments.

Que 1.17. What is an access specifier in Java ? List and explain types of access specifier in Java.

Answer

A. Access specifier :

1. In Java, access specifiers (modifiers) are keywords used to control the visibility and accessibility of classes, methods, variables, and other members within a Java class.
2. They define the level of access that other classes outside the current class have to the members of the class.
3. The access specifiers help in encapsulation and provide a way to hide the internal implementation details of a class.
4. They are important for maintaining the integrity and security of a Java program.

B. Types of access specifier : There are four main access specifiers in Java :

1. **public :** Members marked as 'public' are accessible from any class and package. They have the widest visibility. For example, if a class has a public method, it can be called from any other class.
2. **private :** Members marked as 'private' are only accessible within the same class. They are not visible from other classes or even subclasses of the same class. This is the most restrictive access specifier.
3. **protected :** Members marked as 'protected' are accessible within the same class, its subclasses, and other classes within the same package. They are not accessible outside the package if they are not part of a subclass.
4. **default (no access specifier) :** If no access specifier is specified, the member has package-level visibility. This means it can be accessed only by classes in the same package.

Que 1.18. What are the different types of operators used in Java ?

Answer

Following are the main types of operators used in Java :

A. Arithmetic operators :

1. Arithmetic operators in Java are used to perform mathematical operations on numeric values, including integers and floating-point numbers.
2. Arithmetic operators follow the usual rules of mathematics.
3. When using these operators, you should consider data types and potential division by zero errors.
4. Also mixed-type operations may result in type casting or promotion, depending on the operands involved.

5. Following are the common arithmetic operators used in Java :

Operator	Meaning
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Modulus, remainder after division
++	Increment
--	Decrement

B. Relational operators :

1. Relational operators in Java are used to compare two values and determine the relationship between them.

2. These operators return a boolean value (true or false) based on the comparison result.
3. Relational operators are commonly used in control structures, such as conditional statements and loops, to make decisions or evaluate conditions.
4. Following are the relational operators used in Java :

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

C. Logical operators :

1. Logical operators in Java are used to perform logical operations on boolean values.
2. These operators allow you to combine or modify boolean values and make decisions based on the results.
3. Logical operators are frequently used in conditional statements (e.g., if, while, for) to control program flow and make decisions based on boolean conditions.
4. Following are the logical operators used in Java :

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

D. Assignment operators :

1. Assignment operators in Java are used to assign values to variables.
2. These operators combine the assignment of a value with an arithmetic or bitwise operation.
3. They make it more concise and efficient to update the value of a variable based on its current value.
4. Following are some common assignment operators used in Java :

Operator	Meaning
=	Assignment
+=	Add and assign
-=	Subtract and assign
*=	Multiply and assign

/=	Divide and assign
%=	Modulus and assign
&=	Bitwise AND and assign
|=	Bitwise OR and assign
^=	Bitwise XOR and assign
<<=	Left shift and assign
>>=	Right shift and assign
>>>=	Unsigned right shift and assign

E. Bitwise operators :

1. Bitwise operators in Java are used to perform operations on individual bits of integer types (byte, short, int, long).
2. These operators treat the values as sequences of binary digits (bits) and manipulate them at the bit level.
3. Bitwise operators are often used in low-level programming.
4. Java provides the following bitwise operators :

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	Left shift
>>	Right shift
>>>	Unsigned right shift

F. Conditional operator :

1. The conditional operator in Java, often referred to as the "ternary operator," is a shorthand way of writing an 'if-else' statement in a single line.
2. It provides a compact way to make a decision and assign values based on a condition.
3. The conditional operator has the following syntax :

   condition ? expression1 : expression2
4. 'condition' is a boolean expression that is evaluated first.
5. If the 'condition' is 'true', 'expression' is executed, and its value is returned.
6. If the 'condition' is 'false', 'expression2' is executed, and its value is returned.

G. instanceof operator :

1. The 'instanceof' operator in Java is used to test if an object is an instance of a particular class or interface.
2. It allows you to check whether an object belongs to a specific type or whether it is a subtype of that type.
3. The 'instanceof' operator returns a boolean value, 'true' if the object is an instance of the specified type, and 'false' otherwise.
4. The syntax of the 'instanceof' operator is as follows :

   object instanceof type
5. Here, 'object' is the object you want to test.

   'type' is the class or interface you want to check if the object is an instanceof.

H. Type cast operator :

1. In Java, the type cast operator is used to explicitly convert a value from one data type to another.
2. This operation is known as type casting or type conversion.
3. Type casting can be helpful in situations where you need to work with different data types or need to perform arithmetic or other operations involving mixed data types.
4. The syntax of a type cast is as follows :

   (targetType) value
5. Here, 'targetType' is the data type to which you want to convert the value.

   'value' is the expression or variable you want to convert.

I. String concatenation operator :

1. In Java, the string concatenation operator, denoted by the '+' symbol, is used to combine or join strings together.
2. It can be used to concatenate (combine) two or more string values, variables, or literals to create a single string.
3. Here's how the string concatenation operator works :

   String result = string1 + string2;
4. Here, the + operator is used to combine the two strings, and the result is stored in the result variable.

J. Unary operators :

1. Unary operators in Java are operators that perform operations on a single operand, which can be a variable or an expression.
2. They are commonly used in incrementing or decrementing variables, changing the sign of a number, and negating boolean or bitwise values.

3. Here are some of the common unary operators in Java:

Operator	Meaning
+	Unary plus, used to indicate a positive value
-	Unary minus, used to negate a value
++	Increment
--	Decrement
!	Logical NOT, also used for Boolean negation

Que 1.19. What are static members in Java ?

Answer

1. Variables and methods declared using keyword static are called static members of a class.
2. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
3. In Java, static members belong to the class itself rather than to instances of the class.
4. Static members are used for various purposes, such as maintaining common data across all instances, utility methods, and constants.
5. There are two types of static members in Java :

A. Static variables (Class variables) :

1. Static variables are declared using the 'static' keyword within a class.
2. They are shared among all instances of the class and have a single copy in memory.
3. Static variables are typically used to store data that should be common to all instances of the class.
4. Static variables are accessed using the class name, followed by the dot (.) operator.

B. Static methods :

1. Static methods are declared using the 'static' keyword and can be called without creating an instance of the class.
2. They are often used for utility methods that don't require access to instance-specific data.
3. Static methods cannot access instance-specific data (non-static members) directly because they do not have access to 'this'.
4. They can only access other static members.
5. Static methods are invoked using the class name, followed by the dot (.) operator.

Que 1.20. **Give the characteristics of static members in Java.**

Answer

Static members have following characteristics :

1. They are loaded into memory when the class is loaded, and there is only one copy of each static member per class, regardless of how many instances of the class are created.
2. They can be accessed even if no instances of the class have been created.
3. They are often used for constants and utility functions that do not depend on the state of a specific instance.
4. They are commonly used for factory methods, where a static method creates and returns instances of the class.
5. They can be used to implement the Singleton design pattern, ensuring that only one instance of a class is created.

Que 1.21. **What is 'final' members in Java ? Explain how 'final' is used for different types of members.**

Answer

1. In Java, a 'final' member refers to a variable, method, or class that cannot be modified or overridden once it has been defined.
2. When applied to members, the 'final' keyword enforces constraints on their usage.
3. Here's how 'final' is used for different types of members :

A. Final Variables (Constants) :

1. When a variable is declared as 'final', it becomes a constant, which means its value cannot be changed after it is initialized.
2. Final variables must be initialized when declared or within the constructor of the class if they are instance variables.
3. They are typically written in uppercase letters to distinguish them from regular variables.
4. **Example :**

```
public class Constants {
final int MAX_VALUE = 100; // A final instance variable (constant)
}
```

B. Final Methods :

1. When a method is declared as 'final' in a class, it cannot be overridden or modified in any subclass.

2. This is often used to prevent a specific behavior from being changed in subclasses.
3. Subclasses can still inherit and use the final method, but they cannot provide a different implementation.
4. **Example :**

```
public class Base {
final void doSomething() {
// Final method implementation
    }
}
```

C. Final Classes :

1. When a class is declared as 'final', it cannot be extended or subclassed by other classes.
2. It essentially marks the class as the final implementation, and it cannot be further specialized.
3. Final classes are often used when you want to prevent inheritance and ensure that the class's behavior remains unchanged.
4. **Example :**

```
final class FinalClass {
// Class implementation
    }
```

Que 1.22. **What do you understand by comments in Java ? What is the purpose of comments in Java ?**

Answer

A. Comments in Java :

1. Comments in Java are non-executable text annotations that are used to provide explanations, documentation, and context within the source code.
2. These comments are ignored by the Java compiler and have no impact on the execution of the program.
3. They are intended solely for programmers/developers to understand the code more easily.
4. In Java, there are two primary ways to write comments :
   a. Single-line comments :
      1. Single-line comments are created using double slashes //.
      2. Everything following // on the same line is considered a comment and is not compiled or executed.

3. **Example :**

```
// This is a single-line comment in Java.
int x = 42; // This comment explains the purpose of the variable.
```

b. **Multi-line comments :**

1. Multi-line comments are enclosed within /* and */ delimiters.
2. They can span multiple lines and are used for longer comments or explanations.
3. **Example :**

```
/*
This is a multi-line comment in Java.
It can span multiple lines and is useful for more extensive explanations.
*/
```

B. **Purpose of comments in Java :** The purposes of comments in Java are as follows :

1. **Code documentation :** Comments help document the code, providing explanations for what the code does.
2. **Improved code readability :** Well-placed comments can enhance the readability of the code.
3. **Debugging and troubleshooting :** Comments can be used to temporarily disable or annotate sections of code for debugging or testing purposes.
4. **Licensing and copyright notices :** Comments can contain licensing information, copyright notices, or attribution information for open-source code.

PART-6

Data types, Variables, Operators, Control Flow, Arrays & String.

Que 1.23. Explain data types in Java.

Answer

1. In Java, data types define the type and size of data that can be stored in variables.
2. They are essential for declaring variables, specifying function return types, and ensuring type safety.
3. Java supports a range of primitive data types and non-primitive (reference) data types.
4. Following is an overview of the different data types in Java :

A. **Primitive data types :**

1. Primitive data types represent simple values and are not objects.
2. They are stored directly in memory and are more efficient in terms of memory usage and performance.
3. Java has eight primitive data types :

i. **byte :** A 1-byte data type with a range from -128 to 127.
ii. **short :** A 2-byte data type with a range from -32,768 to 32,767.
iii. **int :** A 4-byte data type with a range from -2^31 to 2^31-1.
iv. **long :** An 8-byte data type with a range from -2^63 to 2^63-1.
v. **float :** A 4-byte data type for single-precision floating-point numbers.
vi. **double :** An 8-byte data type for double-precision floating-point numbers.
vii. **char :** A 2-byte data type that represents a single Unicode character.
viii. **boolean :** A data type with only two possible values: true or false.

B. **Non-primitive (reference) data types :** Non-primitive data types are used to create objects and work with complex data structures. They include :

i. **Classes :** User-defined data types created using the class keyword.
ii. **Interfaces :** Define contracts for classes to implement.
iii. **Arrays :** Ordered collections of elements of the same data type.
iv. **Enums :** Special data types used to define a set of constant values.

Que 1.24. Explain variables and operators in Java.

Answer

A. **Variables :**

1. Variables are named storage locations that hold data, which can be of various data types, such as integers, floating-point numbers, characters, and more.
2. Variables are declared using a specific data type, a name, and an optional initial value.
3. Once declared, you can assign values to variables, update their values, and use them in expressions.
4. Example of variable declaration and assignment :

i. `int age; // Declaration`
ii. `age = 30; // Assignment`
iii. `double price = 49.99; // Declaration and assignment`

B. **Operators :**

1. Operators are symbols or keywords used to perform various operations on variables and values in expressions.
2. Java supports a wide range of operators for arithmetic, comparison, logical, and other operations.
3. **Types of operators in Java :** Refer Q. 1.18, Page 1–17F, Unit-1.

Que 1.25. What do you understand by control flow statements in Java ?

Answer

1. Control flow statements in Java are used to determine the order in which statements are executed in a program.
2. They allow you to control the flow of your program's execution based on conditions, loops, and method calls.
3. Control flow statements are essential for building logic and decision-making within your Java programs.
4. There are three main categories of control flow statements in Java :

A. **Selection (conditional) statements :** Selection statements allow you to make decisions and execute different blocks of code based on conditions. Java provides the following selection statements :

1. **if :** The if statement is used for conditional execution. It executes a block of code if a specified condition is true.
2. **if-else :** The if-else statement allows you to execute one block of code if a condition is true and another block if the condition is false.
3. **switch :** The switch statement is used for multi-way branching. It evaluates an expression and executes the code block associated with the matching case label.

B. **Looping statements :** Looping statements allow you to repeat a block of code multiple times. Java provides three main looping statements :

1. **for :** The for loop is used to execute a block of code a specified number of times.
2. **while :** The while loop continues executing a block of code as long as a specified condition is true.
3. **do-while :** The do-while loop is similar to the while loop, but it guarantees that the block of code is executed at least once before checking the condition.

C. **Transfer statements :** Transfer statements are used to control the flow of your program by altering the normal sequence of execution. Java provides several transfer statements, including :

1. **break :** The break statement is used to exit from a loop or a switch statement prematurely.
2. **continue :** The continue statement is used to skip the current iteration of a loop and proceed to the next iteration.
3. **return :** The return statement is used to exit from a method and optionally return a value to the calling code.
4. **throw :** The throw statement is used to throw an exception, which can be caught by an exception handler.

Que 1.26. What is an array in Java ? How do you declare an array in Java ?

Answer

A. **Array :**

1. An array is a data structure used to store a collection of elements of the same data type.
2. Arrays provide a way to group multiple values of the same type under a single variable name.
3. Each element in an array can be accessed by its index, which is a non-negative integer.

B. **Array declaration :**

1. In Java, you can declare an array using the following syntax :

```
type[] arrayName;
```

2. Here, type represents the data type of the elements in the array (e.g., int, double, String, etc.), and arrayName is the name you give to your array.
3. You can also declare an array with a specific size :

```
type[] arrayName = new type[size];
```

4. For instance, if you want to create an array of 10 integers, you would write :

```
int[] myArray = new int[10];
```

Que 1.27. Define string in Java. How do you create a string variable in Java ?

Answer

A. **String :**

1. A string is an object that represents a sequence of characters.
2. Strings are used to store and manipulate text-based data.
3. Java provides a built-in class called 'java.lang.String' for creating and working with strings.

4. Strings in Java are immutable, which means their values cannot be changed once they are created.
5. Any operation that appears to modify a string actually creates a new string with the modified value.

B. Creating a string variable : Here's how you create a string variable in Java :

1. **Declaration and initialization :** A string variable can declared and initialize in following ways :

i. Using a string literal enclosed in double quotes :

```
String greeting = "Hello, World!";
```

ii. By creating a string object using the 'new' keyword :

```
String name = new String("John");
```

iii. Initializing an empty string :

```
String emptyString = "";
```

iv. Initializing a string with the value 'null' :

```
String nullString = null;
```

2. **Concatenation :** You can concatenate strings using the '+'operator, which joins two strings together to create a new string.

```
String firstName = "John";
String lastName = "Doe";
String fullName = firstName + " " + lastName;
```

3. **Using string methods :** The 'java.lang.String' class provides various methods for working with strings. You can create strings by calling these methods, such as 'substring()', 'concat()', and more.

```
String original = "Hello, World!";
String subString = original.substring(0, 5); // Creates a new string "Hello" from the original
```

Que 1.28. How do you declare and initialize an array of strings in Java ?

Answer

To declare and initialize an array of strings in Java you follow the given process :

A. Declaration :

1. Declare an array variable with the desired data type (in this case, 'String') and square brackets '[]' to indicate it's an array.
2. You can declare an array of strings as a class-level variable or a local variable within a method.

B. Initialization : Initialize the array using one of the following methods :

1. Using an array initializer (with values enclosed in curly braces {}) when you declare the array.

Example : String[] colors = {"Red", "Green", "Blue", "Yellow"};

2. Creating a new array object and assigning it to the variable.

Example :

```
String[] fruits; // Declare a string array variable
fruits = new String[3]; // Initialize the array with a size of 3
fruits[0] = "Apple";  // Assign values to array elements
fruits[1] = "Banana";
fruits[2] = "Orange";
```

PART-7

Object Oriented Programming : Class, Object.

Que 1.29. What is a class and object in Java ? How do you create an object from a class in Java ?

Answer

A. Class : A class is a template that consists of the data members or variables and functions and defines the properties and methods for a group of objects.

B. Object : An object is nothing but an instance of a class. Each object has its values for the different properties present in its class. The compiler allocates memory for each object.

C. Creating an object from a class in Java : In Java, you can create an object from a class by following these steps :

1. **Declare a class :** First, you need to have a class defined. A class is a blueprint for creating objects. Here's an example of a simple class called 'Person' :

```
public class Person {
    String name;
    int age;
}
```

2. **Instantiate the object :** To create an object from the class, use the new keyword followed by the class name and parentheses. This process is called "instantiation."

```
Person person1 = new Person();
```

3. **Access and modify object properties :** You can access and modify the properties (fields or attributes) of the object using the dot (.) notation. For example, you can set the 'name' and 'age' properties of the person1 object :

```
person1.name = "John";
person1.age = 30;
```

Now you have created an object of the 'Person' class with the name "John" and age 30.

Que 1.30. Differentiate between class and object.

Answer

S.No.	Feature	Class	Object
1.	Definition	A class is a blueprint or template	An object is an instance of a class
2.	Purpose	Defines structure and behavior	Represents specific entity
3.	Instantiation	Not instantiated itself	Created from a class
4.	Multiple Instances	Multiple objects from a class	Distinct instances with own data
5.	Methods	Defines behaviors/ methods	Calls methods for actions
6.	Static Members	Can have static members	Has no static member
7.	Inheritance	Can be inherited by other classes	Inherits properties and behaviors
8.	Usage	Provides structure for objects	Represents specific instances

PART-8

Inheritance, Superclass, Subclass.

Que 1.31. Describe inheritance in Java and explain its importance in object-oriented programming (OOP).

Answer

A. Inheritance in Java :

1. Inheritance allows a new class (subclass) to inherit properties and behaviors (fields and methods) from an existing class (superclass).
2. In Java, you can create a subclass that extends a superclass, inheriting its attributes and adding new ones or modifying existing ones.
3. Inheritance in Java is achieved using the 'extends' keyword.

B. Importance of inheritance in OOP : Inheritance is important in OOP for following reasons :

1. **Code reusability :** Inheritance promotes code reuse.
2. **Hierarchy creation :** Inheritance enables you to create a hierarchy of classes, representing an "is-a" relationship.
3. **Polymorphism :** Inheritance is a key factor in achieving polymorphism in OOP.
4. **Method overriding :** Subclasses can override methods inherited from the superclass. This allows you to customize the behavior of a class.
5. **Abstraction :** Inheritance promotes the concept of abstraction.
6. **Efficiency :** Inheritance can lead to more efficient code, as it eliminates the need to duplicate code shared by multiple classes.

Que 1.32. Explain types of inheritance in Java.

Answer

Following are the common types of inheritance in Java :

1. **Single inheritance :**
   i. Single inheritance involves a subclass inheriting from a single superclass.
   ii. In Java, all classes implicitly inherit from the 'Object' class, which serves as the root of the class hierarchy.
   iii. Therefore, Java supports single inheritance.
2. **Multiple inheritance :**
   i. Java does not support multiple inheritance of classes, which means a class cannot directly inherit from more than one class.
   ii. However, multiple inheritance can be achieved through interfaces.
   iii. A class can implement multiple interfaces, effectively inheriting method signatures from multiple sources.

3. **Multilevel inheritance :**
   i. In multilevel inheritance, a subclass derives from another subclass, creating a chain of inheritance.
   ii. A class "C" extends class "B," which, in turn, extends class "A."
4. **Hierarchical inheritance :**
   i. In hierarchical inheritance, multiple subclasses inherit from a single superclass.
   ii. This creates a branching structure where multiple classes share common features.

Que 1.33. **What is a superclass and subclass ?**

Answer

In object-oriented programming and inheritance, the terms "superclass" and "subclass" are used to describe the relationship between two classes.

A. **Superclass :**
1. A superclass, also known as a base class or parent class, is a class from which other classes (subclasses) inherit properties and behaviors.
2. The superclass defines common attributes and methods that are shared by its subclasses.
3. It serves as a template or blueprint for creating derived classes.
4. In a class hierarchy, a superclass is typically higher in the hierarchy and is more general or abstract.

B. **Subclass :**
1. A subclass, also known as a derived class or child class, is a class that inherits properties and behaviors from a superclass.
2. Subclasses extend or specialize the functionality of the superclass.
3. They can add additional attributes, methods, or override inherited methods to customize their behavior.
4. Subclasses are more specific and detailed compared to the superclass.

PART-9

Overriding, Overloading.

Que 1.34. **What is method overriding in Java, and why is it used ?**

Answer

1. Method overriding in Java allows a subclass to provide a specific implementation of a method that is already defined in its superclass.
2. When a subclass overrides a method, it redefines the method with the same name, return type, and parameters as the superclass method.
3. This allows the subclass to customize the behavior of the inherited method.

Why method overriding is used : Method overriding is used for following reasons :

1. **Customization :** It allows a subclass to provide its own implementation of a method inherited from the superclass.
2. **Polymorphism :** Method overriding enables polymorphism, which allows objects of different classes to be treated as objects of a common superclass.
3. **Extensibility :** It allows for the extension of class behavior without changing the existing code.
4. **Consistency :** Method overriding ensures that when a method is called on an object of a subclass, it behaves consistently with the method's contract defined in the superclass.
5. **Specialization :** Subclasses can specialize the behavior of inherited methods.

Que 1.35. **Provide an example of method overriding in Java.**

Answer

Following is an example of method overriding in Java :

```
class Animal {
    public void makeSound() {
        System.out.println("Some generic sound");
            }
    }
class Dog extends Animal {
    @Override
      public void makeSound() {
        System.out.println("Bark");
            }
    }
```

1. In this example, the 'Animal' class has a method 'makeSound'.

3. The 'Dog' class, which is a subclass of 'Animal', overrides this method to provide a specific implementation.
5. When you call 'makeSound' on a 'Dog' object, it will print "Bark" instead of "Some generic sound", demonstrating method overriding.

Que 1.36. **Describe method overloading and its role in Java.**

Answer

1. Method overloading allows you to define multiple methods with the same name in a class but with different parameter lists.
2. The parameter lists may differ in the number of parameters, their types, or both.
3. When you call an overloaded method, the Java compiler determines which version of the method to execute based on the number and types of arguments provided in the method call.

Role of method overloading in Java : Method overloading plays following important roles in Java :

1. **Improved code readability :** Overloaded methods can have the same name, making the code more readable.
2. **Flexibility :** Method overloading provides flexibility by allowing you to define methods that can handle different types of input data without requiring distinct method names.
3. **Default values :** Overloaded methods can provide default values for optional parameters.
4. **Consistency :** It allows you to maintain consistency in method naming conventions, making it easier for developers to understand how to use your classes and methods.
5. **Polymorphism :** Method overloading, when combined with method overriding, contributes to the concept of polymorphism in Java.

Que 1.37. **Give an example of method overloading in Java.**

Answer

Following is an example of method overloading in Java :

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
    public double add(double a, double b) {
        return a + b;
    }
}
```


1. In this example, the 'Calculator' class has two 'add' methods.
2. One takes two int parameters, and the other takes two double parameters.
3. This allows the class to perform addition with both integer and floating-point numbers.

PART-10

Encapsulation, Polymorphism.

Que 1.38. **Explain the concept of encapsulation in Java. Give its advantages.**

Answer

1. Encapsulation refers to the bundling of data and the methods that operate on that data into a single unit called a class.
2. Encapsulation helps to hide the internal implementation details of a class.
3. It provides controlled access to the data by using access modifiers such as 'private', 'protected', and 'public'.

Advantages of encapsulation :

1. **Data security :** Encapsulation provides a level of security by preventing unauthorized access and modification of data.
2. **Controlled access :** It allows controlled access to the data through well-defined methods.
3. **Code flexibility :** By encapsulating data and providing public methods, you can change the internal implementation of a class without affecting other parts of the program that use the class.
4. **Simplified maintenance :** Encapsulation makes it easier to debug and maintain code.
5. **Improved testing :** Encapsulation allows for easier unit testing.
6. **Enhanced readability :** Code that uses encapsulated classes is often more readable and understandable.
7. **Reusability :** Encapsulation promotes code reusability.

Que 1.39. **Provide an example of a class demonstrating encapsulation in Java.**

Answer

```
class Student {
    private String name;
    private int age;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
public void setAge(int age) {
        if (age >= 0 && age <= 120) {
            this.age = age;
    }
    }
    }
public c ass Main {
    lic static void main(String[] args) {
        Student student = new Student();
        student.setName("John Doe");
        student.setAge(25);
        System.out.println("Name: " + student.getName());
        System.out.println("Age: " + student.getAge());
        student.setAge(150); // Outputs "Invalid age value."
    }
    }
```

1. This code demonstrates encapsulation by keeping the 'name' and 'age' fields private and providing controlled access to them through getter and setter methods.
2. It enforces data validation rules for age and hides the internal implementation details of the 'Student' class.

Que 1.40. **Define polymorphism and give its types.**

Answer

A. Polymorphism :

1. Polymorphism refers to the ability of different objects to respond to the same method call in a way that is appropriate for their specific types.
2. In Java, polymorphism allows objects of different classes to be treated as objects of a common superclass.

B. Types of polymorphism in Java : There are two main types of polymorphism in Java :

1. Compile-time polymorphism (static binding) :

i. Also known as method overloading.

ii. Occurs when multiple methods have the same name but different parameter lists within the same class or between a subclass and its superclass.

iii. The Java compiler determines which method to call based on the number and types of arguments passed during compile time.

iv. The decision on which method to execute is made at compile time, and it's often referred to as early binding.

v. **Example :**

```
class Calculator {
int add(int a, int b) { ... }
double add(double a, double b) { ... }
}
```

2. Runtime Polymorphism (dynamic binding) :

i. Also known as method overriding.

ii. Occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

iii. The decision on which method to call is made at runtime, based on the actual type of the object.

iv. This allows different subclasses to customize the behavior of the method, and it's often referred to as late binding.

v. **Example :**

```
class Animal {
void makeSound() { ... }
}
class Dog extends Animal {
void makeSound() { ... }
}
```

Que 1.41. **Differentiate between compile-time polymorphism and runtime polymorphism.**

Answer

Difference :

S. No.	Compile-time polymorphism	Runtime polymorphism
1.	This is resolved by the compiler.	This is not resolved by the compiler.
2.	This is also known as static/ early binding or overloading.	This is also known as dynamic/late binding or overriding.
3.	The method name is same with differentparameters and return type.	The method name is same with the same parameters and same return type.
4.	Provides fast execution since the method to be executed is known at compile-time.	Provides slow execution since the method to be executed is known at runtime.
5.	Less flexible since all things execute at compile-time.	More flexible since all things execute at runtime.

PART-11

Abstraction, Interfaces,and Abstract Class.

Que 1.42. **What is abstraction in object-oriented programming ?**

Answer

1. Abstraction refers to the process of simplifying complex systems by breaking them down into smaller, more manageable parts while hiding the unnecessary details.
2. It is one of the four main principles of object-oriented programming, along with encapsulation, inheritance, and polymorphism.
3. Abstraction allows you to focus on the essential characteristics and behaviors of objects while ignoring the irrelevant or less important aspects.
4. It allows you to define common interfaces and hide implementation details, making it easier to work with and reason about objects in your code.

Key aspects of abstraction :

1. **Abstract classes and interfaces :** Abstraction is often implemented using abstract classes and interfaces.
2. **Hiding implementation details :** Abstraction allows you to hide the internal details of how an object works, exposing only the essential parts to the outside world.
3. **Modeling real-world concepts :** Abstraction enables you to model real-world concepts as objects in your software.
4. **Reusability and extensibility :** Abstraction promotes code reusability by defining common interfaces or abstract classes.
5. **Reducing complexity :** Abstraction simplifies the development process by breaking down complex systems into smaller, manageable components.

Que 1.43. **How is abstraction implemented in Java using abstract classes and interfaces ?**

Answer

Abstraction implementation in Java using an abstract class :

```
abstract class Animal {
String name;
public Animal(String name) {
this.name = name;
}
    // Abstract method (no implementation)
public abstract void makeSound();
    }
    class Dog extends Animal {
      public Dog(String name) {
super(name);
  }
  // Concrete implementation of the abstract method
  @Override
  public void makeSound() {
        System.out.println(name + " barks");
    }
}
class Cat extends Animal {
    public Cat(String name) {
```

```
        super(name);
    }
    // Concrete implementation of the abstract method
    @Override
    public void makeSound() {
        System.out.println(name + " meows");
        }
    }
    public class Main {
        public static void main(String[] args) {
            Animal dog = new Dog("Buddy");
                Animal cat = new Cat("Tom");
                dog.makeSound(); // Output: Buddy barks
            cat.makeSound(); // Output: Tom meows
        }
    }
```

1. In this example, the 'Animal' abstract class defines the 'makeSound' abstract method.
2. Concrete subclasses 'Dog' and 'Cat' provide their specific implementations for the 'makeSound' method.
3. This allows for the abstraction of common animal behaviors while allowing for specific implementations in subclasses.

Que 1.44. Explain the concept of interfaces in Java.

Answer

1. An interface is like a contract that defines a set of methods that a class must implement if it claims to implement that interface.
2. It only contains method signatures without implementations.
3. In Java, a class can implement multiple interfaces.
4. To declare an interface, you use the 'interface' keyword.
5. Example :

```
interface Drawable {
void draw(); // Method signature
}
```

Que 1.45. Differentiate between abstract class and interface.

Answer

Difference :

S. No.	Abstract	Interface
1.	Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods.
2.	Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
3.	Abstract class can have final, non-final, static and nonstatic variables.	Interface has only static and final variables.
4.	Abstract class can have static methods, main method and constructor.	Interface can't have static methods, main method or constructor.
5.	Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
6.	The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.

7.	**Example :**
	public class Shape{
	public abstract void draw();
	}
	}
		Example :
		public interface Drawable{
		void draw();
		}

Que 1.46. Give an example of a Java interface.

Answer

1. Following an example of a Java interface that defines a simple "Drawable" interface with a single abstract method, "draw" :

```
interface Drawable {
    void draw(); // Abstract method declaration
}
```

2. In above example, the "Drawable" interface has only one abstract method, "draw".
3. Any class that implements this interface must provide a concrete implementation for the "draw" method.
4. Let's create a class that implements this "Drawable" interface :

```
class Circle implements Drawable {
```

```
    private int radius;
    public Circle(int radius) {
        this.radius = radius;
    }
@Override
public void draw() {
    System.out.println("Drawing a circle with radius " + radius);
    }
}
```

5. In this example, the "Circle" class implements the "Drawable" interface and provides a specific implementation for the "draw" method.
6. When you create an instance of the "Circle" class and call the "draw" method on it, it will execute the "draw" method defined in the "Circle" class.

Que 1.47. Describe an abstract class and its purpose.

Answer

A. Abstract class :

1. In Java, an abstract class is a class that cannot be instantiated directly but serves as a blueprint for other classes.
2. It is used to define common methods and fields that should be shared among its subclasses, while allowing those subclasses to provide specific implementations for some or all of the abstract methods.
3. Abstract classes are a way to implement abstraction in Java, and they play a central role in creating a hierarchy of classes with shared characteristics.

B. Purposes of an abstract class :

1. **Abstract methods :** Abstract classes can include abstract methods. The purpose of abstract methods is to ensure that all subclasses provide their specific functionality.
2. **Partial implementation :** Partial implementation helps avoid redundant code and enforces consistency across the hierarchy of related classes.
3. **Inheritance :** Subclasses of an abstract class inherit both the abstract methods and the concrete methods. This allows you to create a hierarchical structure of classes.
4. **Polymorphism :** Abstract classes enable polymorphism, which means that you can use references to the abstract class type to work with instances of its concrete subclasses.

Que 1.48. How is an abstract class different from a regular class ?

Answer

S. No.	Feature	Abstract Class	Regular (Concrete) Class
1.	Instantiation	Cannot be instantiated directly with new.	Can be instantiated with new.
2.	Abstract Methods	Can contain abstract methods.	Contains only concrete methods.
3.	Concrete Methods	Can contain concrete methods.	Contains only concrete methods.
4.	Usage in Hierarchy Method	Often used as a base class in class hierarchies.	Used as both base classes and leaf (final) classes in class hierarchies.
5.	Implementation	Contains methods with or without implementations.	Contains methods with concrete implementations.

PART-12

Packages : Defining Package, CLASSPATH Setting for Packages, Making JAR Files for Library Packages.

Que 1.49. What is a package in Java ? Explain the process of defining a package in a Java program.

Answer

A. Package :

1. In Java, a package is a way to organize related classes, interfaces, and sub-packages.
2. It provides a mechanism for grouping related types together, making it easier to manage and maintain a large codebase.
3. A package is essentially a directory that contains a collection of Java files.

B. Defining a package : Here's the process of defining a package in a Java program :

1. **Choose a package name :**
   i. Select a meaningful and unique name for your package.
   ii. The package name is typically in reverse domain name notation, such as 'com.example.myapp'.
2. **Package declaration :**
   i. At the top of each source file that belongs to the package, include a package declaration statement. Example :

   package com.example.myapp;

   ii. This statement informs the Java compiler about the package to which the class or interface belongs.
3. **Organize your directory structure :**
   i. Create a directory structure that mirrors the package name.
   ii. In the example above (com.example.myapp), you would create a directory structure like this :

   com
   └── example
   └── myapp
4. **Compile your code :**
   i. Compile your Java source files using the javac compiler.
   ii. When compiling, make sure your current working directory is the directory that contains the root of your package structure.

Que 1.50. What are the benefits of using packages ?

Answer

Following are some of the key advantages of using packages :

1. **Namespace management :** Packages provide a way to create distinct namespaces for classes. This helps prevent naming conflicts between classes with the same name from different packages.
2. **Code organization :** Code related to a specific functionality can be grouped together in a package, making it easier to locate and manage code files.
3. **Modularity :** Packages promote modularity. This makes the code base easier to maintain.
4. **Access control :** Packages support access control and visibility modifiers like 'public', 'protected', and 'private'.
5. **Code reusability :** Packages make it easier to reuse code across different projects. You can package utility classes and use them in multiple projects.
6. **Encapsulation and abstraction :** Packages allow you to hide the implementation details of classes by using package-private access modifiers. This supports encapsulation and abstraction.

Que 1.51. What is the CLASSPATH in Java ? Describe the steps involved in setting the CLASSPATH for Java packages.

Answer

A. CLASSPATH :

1. The CLASSPATH in Java is a configuration that specifies where the JVM and compiler should look for classes and resources.
2. It's crucial for enabling the JVM to locate and load classes and libraries when running Java applications.
3. Proper CLASSPATH management is essential for Java development, ensuring that the necessary dependencies are available to your programs.

B. Setting the CLASSPATH : Following are the steps involved in setting the CLASSPATH for Java packages :

1. **Determine dependencies :** Identify the directories and JAR files that contain the classes and resources you need for your Java packages.
2. **Define the CLASSPATH :** Determine how you want to set the CLASSPATH. You have following options :

i. **Environment variable :** You can set the CLASSPATH as an environment variable in your operating system.

set CLASSPATH=C:\path\to\directory1;

ii. **Command-line option :** You can specify the CLASSPATH using the '-cp' or '-classpath' option when running the 'java' or 'javac' commands.

java -cp /path/to/directory1:/path/to/myLibrary.jar YourMainClass

iii. **Manifest file :** If you are working with JAR files, you can specify the CLASSPATH in the manifest file of your JAR file using the 'Class-Path' attribute.

Class-Path: lib/myLibrary.jar

3. **Order and delimiters :** If you use multiple paths in the CLASSPATH, separate them with the appropriate delimiter for your operating system (semicolon ';' on Windows).
4. **Verify the setting :** Double-check that the CLASSPATH is set correctly by running the 'echo %CLASSPATH%' (for Windows) to display the current value.
5. **Compile or run your Java program :** After setting the CLASSPATH, you can compile and run your Java program as usual. The JVM will use the CLASSPATH to locate and load the required classes and resources.

Que 1.52. **What is the purpose of setting the CLASSPATH ? What happens if you don't set the CLASSPATH correctly for your package ?**

Answer

A. **Purpose of setting the CLASSPATH :** The purposes of setting the CLASSPATH are as follows :

1. Locating classes and resources.
2. Handling dependencies.
3. Supporting modular development.
4. Avoiding class name conflicts.
5. Class loading and resolution.
6. Managing dependencies for third-party libraries.

B. **Consequences of not setting the CLASSPATH correctly :**

1. **ClassNotFoundException :** If the CLASSPATH is not set correctly, the JVM will not be able to find the classes it needs to run your program. This will result in a ClassNotFoundException.
2. **NoClassDefFoundError :** This error occurs when the class is found at compile time but not at runtime. It usually happens when you run a program that depends on a library that is not in the CLASSPATH.
3. **Resource loading issues :** If your program depends on resources like properties files or XML configurations, they won't be found if the CLASSPATH is set incorrectly.
4. **Library and dependency problems :** If you're using external libraries or dependencies, they need to be included in the CLASSPATH. If a required library or dependency is not included, it can result in compilation or runtime error.

Que 1.53. **How do you create JAR files for libraries in Java ?**

OR

What do you understand by JAR (Java Archive) files in Java ? Explain the steps involved in creating a JAR file.

Answer

JAR (Java Archive) files : In Java, a JAR file is a compressed file format used to package Java classes, associated metadata, and resources into a single file. JAR files are commonly used for distributing Java libraries, applications, or applets.

Creating JAR files : To create a JAR file follow these steps :

1. Compile and generate .class files :
   i. Open your command prompt or terminal.

   ii. Navigate to the directory containing your Java source files.
   iii. Compile your Java files using the javac command. For example, if your main class is named MyLibrary.java, you would run :

   ```
   javac MyLibrary.java
   ```

   iv. This will generate corresponding .class files.
2. **Create a manifest file :**
   i. A manifest file is not always necessary, but it's useful for specifying the entry point of your application if it's an executable JAR.
   ii. Create a text file named Manifest.txt and include a line like this :

   ```
   Main-Class: com.example.MainClass
   ```

   iii. Replace com.example.MainClass with the fully qualified name of the class containing the main method.
3. **Create the JAR file :** Use the jar command to create the JAR file. You can include the Manifest.txt file if needed using the m option. For example :

   ```
   jar cfm MyLibrary.jar Manifest.txt *.class
   ```

4. **Verify the JAR file :**
   i. You can verify the contents of the JAR file using the following command :

   ```
   jar tf MyLibrary.jar
   ```

   ii. This will list the files contained within the JAR.
5. **Test the JAR file :** You can test your JAR by running it using the java -jar command. For example :

   ```
   java -jar MyLibrary.jar
   ```

Que 1.54. **What is the role of JAR files in Java ?**

Answer

Following are the key roles of JAR files in Java :

1. **Packaging classes and resources :** JAR files allow you to package multiple class files, resource files, and other assets into a single, compressed archive.
2. **Classpath management :** JAR files are a standard way to manage dependencies in Java.
3. **Modularity :** JAR files facilitate modularity in Java by organizing code into reusable and manageable units.
4. **Reduced file size :** JAR files are compressed, which reduces their size.
5. **Security :** JAR files support digital signatures and can include a manifest file to specify security attributes.

6. **Cross-platform compatibility :** JAR files are platform-independent, making them suitable for running Java applications on various operating systems without modification.
7. **Version control :** JAR files can include version information, allowing you to manage and track different versions of libraries or applications.

PART-13

Import and Static Import, Naming Convention For Packages.

Que 1.55. Explain the process of importing packages in Java. What is the purpose of the import statement ?

Answer

1. Importing packages in Java is the process of including classes and interfaces from other packages in your Java source code.
2. The 'import' statement is used to specify which classes or packages you want to access in your code.
3. Here's how the process of importing packages in Java works :

A. Import statement syntax :

i. The 'import' statement is followed by the fully qualified name of the class or package you want to import. The syntax is as follows :

```
import package.name.ClassName;
```

ii. You can use the '*' wildcard character to import all classes and interfaces from a specific package, making them available for use in your code. For example :

```
import package.name.*;
```

B. Importing single classes : To import a single class or interface, specify the package name and the class name separated by a dot. For example :

```
import java.util.ArrayList;
```

C. Importing entire packages : To import all classes and interfaces from a package, you can use the '*' wildcard character. For example:

```
import java.util.*;
```

D. Multiple import statements :

i. You can have multiple import statements in your Java source file to import classes and packages from different sources.

ii. They should appear at the top of the file, after the 'package

declaration' (if present) and before the class declaration.

```
package com.example.myapp;
import java.util.ArrayList;
import java.util.HashMap;
import com.example.otherpackage.*;
```

Purpose of the import statement :

1. The purpose of the import statement is to simplify and clarify your Java code.
2. It allows you to reference classes and interfaces from other packages using their simple names, making your code more readable and reducing the need for fully qualified class names.
3. It also promotes code reusability by allowing you to integrate external classes and libraries seamlessly into your projects.

Que 1.56. Explain the concept of static imports in Java. What are the benefits of static imports ?

Answer

A. Static imports :

1. Static imports in Java are a feature introduced in Java 5 (J2SE 5.0) that allows you to import and use static members (fields and methods) of a class directly without having to prefix them with the class name.
2. This feature simplifies code readability and can be particularly useful when working with classes that provide utility methods or constants.

B. Benefits of static imports :

1. **Improved code readability :** Static imports make the code more concise and easier to read by avoiding repetitive class name prefixes for static members.
2. **Convenient access :** It provides a convenient way to access utility methods, constants, and other static members, which can lead to cleaner and more expressive code.
3. **Enhancing maintainability :** When working with libraries or external APIs that expose many static methods or constants, static imports can simplify code maintenance and updates.

Que 1.57. What is the difference between a regular import and a static import in Java ?

Answer

S. No.	Aspect	Regular Import	Static Import
1.	Purpose	Used to import classes or packages.	Used to import static members (fields and methods) from a class.
2.	Syntax	import	import static
3.	Usage	package.ClassName;	package.ClassName.*;
4.	Example	- Enables you to use the class name without the package prefix.	- Allows you to use static members without the class name prefix.
		java import java.util.List;	java import static java.lang.Math.*;

Que 1.58. What are naming conventions for Java packages ?

Answer

Naming conventions for Java packages help maintain a consistent and organized structure for your Java projects. Adhering to these conventions makes it easier for developers to understand and navigate your code. Following are some common naming conventions for Java packages :

1. **Lowercase letters :** Package names should be in lowercase letters.
2. **Unique names :** Package names should be unique to your project to avoid naming conflicts with other libraries or projects.
3. **Reverse domain name :** Using a reverse domain name as the prefix for your package names helps ensure uniqueness. It usually follows the format of com.example.myapp.
4. **Avoid using keywords :** Don't use Java keywords as package names (e.g., package int is not allowed).
5. **Short and descriptive names :** Keep package names short but descriptive. For example, if a package contains utility classes for date manipulation, a suitable name could be util.date.
6. **Avoid underscores :** Avoid using underscores (_) in package names.
7. **No special characters :** Package names should not contain special characters, spaces, or punctuation marks.
8. **Avoid acronyms :** Avoid using acronyms unless they are widely accepted and understood within your organization or the development community.

Following these naming conventions for Java packages will help keep your codebase organized, maintainable, and easily understandable.

☺☺☺

2 UNIT

Exception Handling

CONTENTS

PART-1

Exception Handling : The Idea Behind Exception, Exception and Errors, Types of Exception.

Que 2.1. What do you mean by exception in Java ? What is the fundamental idea behind exception handling in Java ? How does Java handle exceptions ?

Answer

A. Exception in Java :

1. In Java, an exception is an event that disrupts the normal flow of program execution.
2. It represents an unexpected condition or error that occurs during runtime.
3. Exceptions can arise due to various reasons, such as invalid user input, file not found, or programming errors.

B. Fundamental idea behind exception handling :

1. The fundamental idea behind exception handling in Java is to provide a mechanism for dealing with runtime errors in a structured and graceful manner.
2. Exception handling allows you to anticipate and handle exceptional situations that may arise during program execution.

C. Exception handling in Java :

1. Java handles exceptions through a combination of try, catch, and finally blocks.
2. Statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown.
3. Your code can catch this exception using catch and handle it.
4. Any code that absolutely must be executed before a method returns is put in a finally block.
5. Following is a basic structure of exception handling in Java :

```
try {
    // Code that might throw an exception
} catch (ExceptionType e) {
    // Handle the exception
} finally {
    // Code that always executes, regardless of whether an exception occurred
}
```


Que 2.2. What do you mean by errors in Java ?

Answer

1. In Java, errors represent serious, unrecoverable problems that occur during the execution of a program.
2. Unlike exceptions, which can be caught and handled programmatically, errors generally are beyond the control of the application.
3. It usually requires intervention at a higher level, such as the JVM or the operating system.
4. Errors in Java are instances of classes that extend the 'java.lang.Error' class.
5. Some common errors include :
   - i. **OutOfMemoryError :** This error occurs when the Java Virtual Machine (JVM) runs out of memory and is unable to allocate more memory to the application.
   - ii. **StackOverflowError :** This error occurs when the execution stack grows beyond its maximum size.
   - iii. **VirtualMachineError :** This is the superclass of all errors thrown by the JVM.
   - iv. **LinkageError :** This error occurs when there is a problem with the classloading mechanism.
   - v. **AssertionError :** This error occurs when an assertion made by the 'assert' statement fails.

Que 2.3. Explain the difference between exceptions and errors in Java.

Answer

S. No.	Aspect	Exceptions	Errors
1.	Parent class	java.lang.Exception.	java.lang.Error.
2.	Nature of occurrence	Usually recoverable and handled.	Severe, often unrecoverable.
3.	Cause	Generally caused by the application's logic or environment.	Typically caused by external factors such as the JVM or hardware.
4.	Handling	Can be caught and handled using try-catch blocks.	Usually not caught or handled.
5.	Example	FileNotFoundException.	OutOfMemoryError

Que 2.4. **What are the different types of exceptions in Java ? Provide examples for each type.**

Answer

In Java, exceptions are categorized into two main types: checked exceptions and unchecked exceptions.

A. Checked exceptions :

1. Checked exceptions are exceptions that are checked by the compiler at compile-time.
2. The compiler ensures that these exceptions are either caught or declared to be thrown by the method using the throws clause.
3. Examples of checked exceptions include 'IOException', 'FileNotFoundException', and 'ClassNotFoundException'.
4. These exceptions usually represent conditions that a well-behaved application should anticipate and recover from.
5. Example :

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
public class Main {
    public static void main(String[] args) {
        try {
        // Attempt to open a file
        FileInputStream file = new FileInputStream("file.txt");
        } catch (FileNotFoundException e) {
        // Handle the exception
        System.out.println("File not found: " + e.getMessage());
        }
    }
}
```

B. Unchecked exceptions :

1. Unchecked exceptions, also known as runtime exceptions, are not checked by the compiler at compile-time.
2. Instead, they occur at runtime and are typically caused by programming errors.
3. Examples of unchecked exceptions include 'NullPointerException', 'ArrayIndexOutOfBoundsException', and 'ArithmeticException'.
4. These exceptions often indicate programming errors that could have been avoided with proper coding practices.

5. Example :

```
public class Main {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3};
        try {
        // Attempt to access an index out of bounds
        int value = arr[5];
    } catch (ArrayIndexOutOfBoundsException e) {
        // Handle the exception
    System.out.println("Array index out of bounds: " +
    e.getMessage());
    }
    }
}
```

PART-2

Control Flow in Exceptions, JVM Reaction to Exceptions.

Que 2.5. **Describe the control flow in exception handling when an exception occurs.**

OR

How does Java manage control flow when an exception occurs ?

Answer

When an exception occurs in Java, the control flow changes to handle the exception. Here's a step-by-step description of the control flow in exception handling :

1. **Exception occurs :** An exceptional condition occurs due to various reasons such as invalid input, arithmetic errors, file I/O errors, or other unexpected conditions.
2. **Exception thrown :** The code that detects the exceptional condition creates an object representing the exception and throws it using the throw statement.
3. **Search for matching catch blocks :** The Java runtime searches for a matching catch block to handle the thrown exception.
4. **Control transfers to matching catch block :** If a matching catch block is found, the control flow transfers to the corresponding catch block, skipping the remaining code in the try block.
5. **Exception handling :** The code inside the matching catch block is executed to handle the exception.

6. **Program continues execution :** After the exception is handled, the control flow continues from the end of the catch block.

Que 2.6. **How does the JVM react to exceptions during program execution ?**

OR

Describe how the JVM reacts to an unhandled exception in a program.

Answer

When a Java Virtual Machine (JVM) encounters an unhandled exception in a program, it follows these steps :

1. **Exception propagation :** The exception is propagated up through the call stack. It looks for a catch block that matches the type of the exception.
2. **Stack unwinding :** If a matching catch block is found, the JVM "unwinds" the call stack. This means it starts popping off the call stack frames until it finds a matching catch block.
3. **Exception handling :** Once a matching catch block is found, the exception is caught, and the code inside the catch block is executed. If no matching catch block is found, the program terminates, and an error message is displayed.
4. **Finally block (Optional) :** If there's a finally block associated with the try-catch structure, it will be executed after the catch block, regardless of whether an exception was thrown or not.
5. **Exiting the program (Optional) :** After all relevant catch and finally blocks are executed, the program continues its execution. If no suitable catch block is found, the JVM may terminate the program.

PART-3

Use of Try, Catch, Finally, Throw, Throws in Exception Handling.

Que 2.7. **Explain the purpose and usage of the try, catch, and finally blocks in exception handling.**

Answer

Following is an explanation of each block and its purpose :

A. try block :

1. The try block encloses the code that might throw an exception.
2. It is used to define a block of code where exceptions might occur.
3. The syntax is :

```
try {
    // Code that might throw an exception
}
```

4. The try block must be followed by either a catch block, a finally block, or both.

B. catch block :

1. The catch block follows a try block and specifies the type of exception it can handle.
2. It is used to catch and handle exceptions that occur within the associated try block.
3. If an exception occurs in the try block, Java searches for a matching catch block.
4. The syntax is :

```
catch (ExceptionType e) {
    // Handle the exception
}
```

5. Multiple catch blocks can follow a single try block to handle different types of exceptions.

C. finally block :

1. The finally block follows a try block and contains code that always executes, regardless of whether an exception occurred or not.
2. It is used for releasing resources, performing cleanup operations, or finalizing tasks.
3. The finally block executes even if an exception is thrown and caught, allowing for essential cleanup actions.
4. The syntax is :

```
finally {
    // Code that always executes, regardless of whether an exception occurred
}
```

5. The finally block is optional, but if used, it must follow the last catch block (if any).

Que 2.8. **What is the role of the throw keyword in exception handling ? Provide an example.**

Answer

1. In Java, the throw keyword is used to explicitly throw an exception within a method.
2. It allows developers to create and throw their own exceptions or to re-throw exceptions that were caught earlier.

3. The throw statement is typically used when a method encounters an exceptional condition that it cannot handle itself.
4. Here's the syntax of the throw statement :

```
throw throwableObject;
```

5. Where 'throwableObject' is an instance of a subclass of 'Throwable', such as 'Exception' or 'Error'.
6. Example :

```
public class ThrowExample {
    public static void main(String[] args) {
        try {
        // Call a method that throws an exception
        divideByZero();
        } catch (ArithmeticException e) {
        // Handle the exception
        System.out.println("Exception caught: " + e.getMessage());
        }
    }
    public static void divideByZero() {
        int dividend = 10;
        int divisor = 0;
        if (divisor == 0) {
        // If divisor is 0, throw an ArithmeticException
        throw new ArithmeticException("Cannot divide by zero");
        } else {
        // Otherwise, perform the division
        int result = dividend / divisor;
        System.out.println("Result of division: " + result);
        }
    }
}
```

Que 2.9. **How does the throws keyword contribute to exception handling in Java ?**

Answer

1. In Java, the throws keyword is used to indicate that the method may throw certain types of exceptions during its execution.

2. It is part of the method signature and provides information to the caller about the types of exceptions that the method might throw.
3. This helps in informing the caller about the potential exceptions that need to be handled.
4. Here's how the throws keyword contributes to exception handling in Java :
   i. **Informing callers :** It informs the caller about the potential exceptions that might occur during the method's execution. This allows the caller to be aware of the exceptions that need to be handled.
   ii. **Propagation of exceptions :** If a method encounters an exceptional condition that it cannot handle itself, it can choose to propagate the exception. This allows the exception to be propagated further up the call stack.
   iii. **Compile-time checking :** The throws keyword enables compile-time checking of exception handling. This helps in ensuring that exceptions are handled appropriately at compile time.

PART-4

In-Built and User Defined Exceptions, Checked and Unchecked Exceptions.

Que 2.10. **What are inbuilt exceptions ? Give an example.**

Answer

1. In Java, inbuilt exceptions refer to the exceptions that are provided by the Java standard library.
2. These exceptions are predefined and are available for use in Java programs.
3. Inbuilt exceptions are typically organized into a hierarchy of exception classes, with the base class being 'java.lang.Exception'.
4. Examples of inbuilt exceptions include :
   i. **NullPointerException :** This exception is thrown when a program attempts to access or manipulate an object reference that has a null value.

```
String str = null;
System.out.println(str.length()); // Throws NullPointerException
```

   ii. **ArithmeticException :** This exception is thrown when an arithmetic operation such as division by zero occurs.

```
int result = 10 / 0; // Throws ArithmeticException
```

Que 2.11. **What are user-defined exceptions ?**

Answer

1. User-defined (custom) exceptions are exceptions that are defined by the programmer/user.
2. These exceptions allow developers to create more specific and meaningful error handling mechanisms.
3. In Java developers can define their own exception classes by extending the base exception class provided by the language.
4. This allows them to create exception types that are relevant to their specific application domain.
5. By raising a user-defined exception, developers can communicate specific error conditions, improving code readability and maintainability.
6. Additionally, it allows for more granular error handling.
7. Here's an example of how you can define a custom exception in Java :

```
// Define a custom exception by extending Exception class
class MyCustomException extends Exception {
    // Constructor that takes a message as parameter
public MyCustomException(String message) {
    // Call the constructor of the superclass (Exception) with the message
    super(message);
    }
}
```

Que 2.12. Differentiate between in-built and user-defined exceptions with examples.

Answer

S. No.	Aspect	Built-in exceptions	User-defined exceptions
1.	Definition	Exceptions provided by the programming language.	Exceptions defined by the programmer.
2.	Raised automatically?	Yes, automatically raised by the interpreter.	No, must be explicitly raised by the programmer.
3.	Inheritance	All built-in exceptions inherit from 'BaseException'.	User-defined exceptions can inherit from 'Exception' or its subclasses.
4.	Usage	Used for general error handling across applications.	Tailored for specific error scenarios in the application.
5.	Example	ZeroDivisionError	FileNotFoundError

Que 2.13. Distinguish checked and unchecked exceptions in Java ?

Answer

S. No.	Aspect	Checked exceptions	Unchecked exceptions
1.	Definition	Must be caught or declared.	Need not be explicitly handled or declared.
2.	Example	IOException	NullPointerException
3.	Compilation	Checked by the compiler.	Not checked by the compiler.
4.	Handling	Must be handled with try-catch or throws clause.	Can be handled optionally with try-catch if necessary.
5.	Purpose	Typically for external factors (e.g., I/O).	Typically for programming errors or exceptional cases.
6.	Inheritance	Extend 'Exception' class.	Extend 'RuntimeException' or its subclasses.

PART-5

Input / Output Basics : Byte Streams and Character Streams.

Que 2.14. What is byte stream in Java ? Give example of commonly used byte stream classes in Java.

Answer

A. Byte stream in Java :

1. In Java, a byte stream is a stream of raw bytes.
2. It is used to perform I/O operations on binary data, such as images, audio, video, or any other type of non-textual data.
3. Byte streams are suitable for reading and writing raw bytes, making them efficient for handling binary data.

B. Commonly used byte stream classes :

1. **InputStream :** This abstract class is the superclass of all classes representing an input stream of bytes. It is used for reading bytes from a source.
2. **OutputStream :** This abstract class is the superclass of all classes representing an output stream of bytes. It is used for writing bytes to a destination.
3. **FileInputStream :** This class is used to read data from a file as a stream of bytes.
4. **FileOutputStream :** This class is used to write data to a file as a stream of bytes.
5. **ByteArrayInputStream :** This class is used to read data from a byte array as a stream of bytes.
6. **ByteArrayOutputStream :** This class is used to write data to a byte array as a stream of bytes.

Que 2.15. What do you mean by character stream in Java ? Name commonly used character stream classes in Java.

Answer

A. Character stream in Java :

1. In Java, character streams are used to perform I/O operations for textual data.
2. Character stream deals with characters instead of raw bytes.
3. Character streams are designed to handle Unicode characters efficiently.
4. Character stream provides a convenient mean for reading and writing text data in various character encodings.

B. Commonly used character stream classes :

1. **Reader :** This abstract class is the superclass of all classes representing an input stream of characters. It is used for reading characters from a source.
2. **Writer :** This abstract class is the superclass of all classes representing an output stream of characters. It is used for writing characters to a destination.
3. **FileReader :** This class is used to read data from a file as a stream of characters.
4. **FileWriter :** This class is used to write data to a file as a stream of characters.
5. **BufferedReader :** This class reads text from a character-input stream, buffering characters to provide efficient reading of characters, arrays, and lines.

6. **BufferedWriter :** This class writes text to a character-output stream, buffering characters to provide efficient writing of characters, arrays, and lines.

Que 2.16. Differentiate between byte streams and character streams in Java I/O.

Answer

S. No.	Aspect	Byte streams	Character streams
1.	Purpose	Used for handling binary data, like images, audio, video, etc.	Used for handling textual data, such as strings of characters.
2.	Handling data	Handle raw binary data (bytes).	Handle characters, automatically encoding and decoding them.
3.	Data type	Deal with bytes (binary data).	Deal with characters (textual data).
4.	Encoding	No automatic encoding or decoding of data.	Automatic encoding and decoding using specified character encoding.
5.	Usage	Suitable for handling non-textual data.	Suitable for handling textual data.
6.	Efficiency	Less efficient for text-based operations.	More efficient for text-based operations.

PART-6

Reading and Writing File in Java.

Que 2.17. Explain the process of reading and writing files using byte streams.

Answer

A. Writing files using byte streams : Writing files using byte streams in Java involves the use of classes such as 'FileOutputStream'. Following is an overview of the process :

1. **Create a FileOutputStream :**
   i. Instantiate a 'FileOutputStream' object, passing the file path as a parameter.

ii. This class represents an output stream for writing raw bytes to a file.

2. **Write data :**
   i. Use the 'write()' method of the 'FileOutputStream' to write bytes to the file.
   ii. You can write byte arrays, individual bytes, or portions of byte arrays.
3. **Close the stream :**
   i. After writing the data, it's essential to close the 'FileOutputStream' using the 'close()' method.
   ii. This releases any system resources associated with it and ensure data is flushed to the file.
4. **Example of writing files with byte streams :**

```
import java.io.FileOutputStream;
import java.io.IOException;
public class ByteStreamWriteExample {
    public static void main(String[] args) {
        try {
            // Create a FileOutputStream
            FileOutputStream outputStream = new
                                    FileOutputStream("output.txt");
            // Write data to the file
            String data = "Hello, world!";
            byte[] bytes = data.getBytes(); // Convert string to byte array
            outputStream.write(bytes);
            // Close the stream
            outputStream.close();
            } catch (IOException e) {
            e.printStackTrace();
            }
    }
}
```

B. Reading files using byte streams : Reading files using byte streams in Java involves the use of classes such as 'FileInputStream'. Following is an overview of the process :

1. **Create a FileInputStream :**
   i. Instantiate a 'FileInputStream' object, passing the file path as a parameter.

ii. This class represents an input stream for reading raw bytes from a file.

2. **Read data :**
   i. Use the 'read()' method of the 'FileInputStream' to read bytes from the file.
   ii. You can read byte arrays or individual bytes.
3. **Close the stream :**
   i. After reading the data, it's essential to close the 'FileInputStream' using the 'close()' method.
   ii. This releases any system resources associated with it.
4. **Example of reading files with byte streams :**

```
import java.io.FileInputStream;
import java.io.IOException;
public class ByteStreamReadExample {
    public static void main(String[] args) {
        try {
            // Create a FileInputStream
            FileInputStream inputStream = new
                                    FileInputStream("input.txt");
            // Read data from the file
            int data;
            while ((data = inputStream.read()) != -1) {
            System.out.print((char) data); // Convert byte to char and
            print
            }
            // Close the stream
            inputStream.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Que 2.18. Describe the steps involved in reading and writing files using character streams.

Answer

A. **Writing files using character streams :** Writing files using character streams in Java involves the use of classes such as 'FileWriter' and 'BufferedWriter'. Here are the steps involved in the process :

1. **Create a FileWriter :**
   i. Instantiate a 'FileWriter' object, passing the file path as a parameter.
   ii. This class represents an output stream for writing characters to a file.
2. **Write data :**
   i. Use the 'write()' method of the 'FileWriter' (or 'BufferedWriter') to write characters to the file.
   ii. You can write strings, characters, or portions of strings.
3. **Close the stream :**
   i. After writing the data, it's essential to close the 'FileWriter' (and 'BufferedWriter', if used) using the 'close()' method.
   ii. This releases any system resources associated with it and ensure data is flushed to the file.
4. **Example of writing files with character streams :**

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
public class CharacterStreamWriteExample {
    public static void main(String[] args) {
        try {
        // Create a FileWriter
        FileWriter writer = new FileWriter("output.txt");
        // Create a BufferedWriter for efficient writing (optional)
        BufferedWriter bufferedWriter = new
        BufferedWriter(writer);
        // Write data to the file
        String data = "Hello, world!";
        bufferedWriter.write(data);
        // Close the streams
        bufferedWriter.close();
        writer.close();
    } catch (IOException e) {
    e.printStackTrace();
```



```
        }
    }
}
```

B. **Reading files using character streams :** Reading files using character streams in Java involves the use of classes such as 'FileReader' and 'BufferedReader'. Here are the steps involved in the process :

1. **Create a FileReader :**
   i. Instantiate a 'FileReader' object, passing the file path as a parameter.
   ii. This class represents an input stream for reading characters from a file.
2. **Read data :**
   i. Use the 'read()' method of the 'FileReader' (or 'BufferedReader') to read characters from the file.
   ii. You can read strings, characters, or portions of strings.
3. **Close the stream :**
   i. After reading the data, it's essential to close the 'FileReader' (and 'BufferedReader', if used) using the 'close()' method.
   ii. This releases any system resources associated with it.
4. **Example of reading files with character streams :**

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class CharacterStreamReadExample {
    public static void main(String[] args){
    try {
    // Create a FileReader
    FileReader reader = new FileReader("input.txt");
    // Create a BufferedReader for efficient reading (optional)
    BufferedReader bufferedReader = new
    BufferedReader(reader);
    // Read data from the file
    String line;
    while ((line = bufferedReader.readLine()) != null) {
    System.out.println(line); // Print each line
    }
    // Close the streams
    bufferedReader.close();
```

```
        reader.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Que 2.18. **What are the advantages of using character streams over byte streams for file I/O operations ?**

Answer

Using character streams over byte streams for file I/O operations offers following advantages :

1. **Character encoding handling :** Character streams automatically handle character encoding and decoding. This eliminates the need for manual encoding/decoding when working with textual data.
2. **Textual data handling :** Character streams are specifically designed for handling textual data, making them more convenient for reading and writing text files.
3. **Internationalization support :** Character streams support Unicode characters, making them suitable for handling internationalized text.
4. **Efficiency for text operations :** Character streams, especially when used with buffering, offer improved performance for text-based I/O operations.
5. **Platform independence :** Character streams abstract away platform-specific details related to character encoding, making Java code more portable across different platforms.
6. **Ease of use :** Character streams provide higher-level abstractions for working with different text processing tasks. This makes them easier to use.

Que 2.20. **How do you handle exceptions during file I/O operations in Java ?**

Answer

Here's how you can handle exceptions during file I/O operations :

1. **Using try-catch blocks :** Surround file I/O operations with a try-catch block to catch any exceptions that might occur during the operation.

```
try {
    // File I/O operations
} catch (IOException e) {
    // Handle IOException
    e.printStackTrace();
}
```


2. **Handle specific exceptions :** Catch specific exceptions that might be thrown during file I/O operations, such as IOException, to provide more specific error handling.

```
try {
    // File I/O operations
} catch (IOException e) {
    // Handle IOException
    e.printStackTrace();
}
```

3. **Propagate exceptions :** Propagate exceptions to the calling method if they cannot be handled locally. This allows higher-level code to handle the exception appropriately.

```
public void readFile() throws IOException {
    FileReader reader = null;
        try {
        reader = new FileReader("file.txt");
        // File I/O operations
        } finally {
        if (reader != null) {
        reader.close();
        }
    }
}
```

PART-7

Multithreading : Thread, Thread Life Cycle, Creating Threads.

Que 2.21. **What is a thread in Java ? Explain its significance in concurrent programming.**

Answer

A. **Thread :**

1. In Java, a thread refers to a lightweight process that exists within a larger process (JVM) and operates independently.

2. Threads allow concurrent execution of multiple tasks within a single application.
3. This enables developers to perform multiple operations simultaneously.

B. Significance of thread in concurrent programming :

1. **Concurrency :** Threads enable concurrent execution of tasks within a Java application.
2. **Multitasking :** Threads allow a Java program to perform multiple tasks simultaneously.
3. **Responsiveness :** Threads can help ensure that an application remains responsive even when performing time-consuming tasks.
4. **Parallelism :** Threads can be used to achieve parallelism, where multiple tasks are executed simultaneously on multi-core processors.
5. **Asynchronous programming :** Threads facilitate asynchronous programming, allowing certain tasks to execute independently of the main program flow.
6. **Resource sharing :** Threads can share resources such as memory, files, and network connections within the same process.

Que 2.22. Describe the life cycle of a thread in Java.

Answer

Lifecycle of thread :

1. **New and Runnable States :** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread, which places it in the runnable state. A thread in the runnable state is considered to be executing its task.
2. **Waiting State :** Sometimes a runnables thread transitions to the waiting state while it waits for another thread to perform a task. A waiting thread transitions back to the runnable state only when another thread notifies it to continue executing.
3. **TimedWaiting State :** A runnable thread can enter the timed waiting state for a specified interval of time. Timed waiting and waiting threads cannot use a processor, even if one is available.
4. **Blocked State :** A runnable thread transitions to the blocked state when it attempts to perform a task that cannot be completed immediately and it must temporarily wait until that task completes.
5. **Terminated State :** A runnable thread enters the terminated state (sometimes called the dead state) when it successfully completes its task or otherwise terminates (perhaps due to an error).

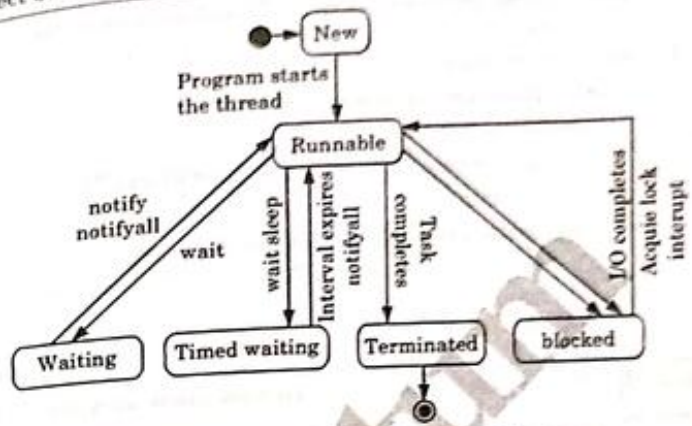


Fig. 2.22.1. Thread lif-cycle UML state diagram.

Que 2.23. How do you create threads in Java ? Provide examples using both Thread class and Runnable interface.

Answer

In Java, a thread can be created in two main ways :

A. Extending the Thread class :

1. You can create a class that extends the Thread class and overrides the run() method.
2. The run() method contains the code that will be executed in the new thread.
3. Here's an example :

```
class MyThread extends Thread {
    public void run() {
        System.out.println("This is a new thread.");
    }}
public class Main {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.start();
    }}
```

B. Implementing the Runnable interface :

1. You can create a class that implements the Runnable interface and overrides the run() method.

2. Then, you can create an instance of Thread and pass an instance of your class to its constructor.
3. Here's an example :

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("This is a new thread."); }}
public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable());
        thread.start();
    }}
```

Que 2.24. **Differentiate between Thread class and Runnable interface.**

Answer

S. No.	Aspect	Thread class	Runnable interface
1.	Inheritance	Extends the Thread class.	Does not extend any class.
2.	Implementation	Directly implements the run() method.	Provides a single run() method to be implemented.
3.	Resource consumption	Consumes more system resources due to inheritance.	Consumes less system resources as it's an interface.
4.	Sharing	Cannot be shared among multiple threads.	Can be shared among multiple threads.
5.	Encapsulation	Encapsulates both thread logic and thread creation.	Encapsulates only the thread logic.

PART-8

Thread Priorities, Synchronizing Threads, Inter-thread Communication.

Que 2.25. **What is thread priority and why is it important in a multi-threaded environment ?**

Answer

A. **Thread Priority :**

1. Every Java thread has a thread priority (from MIN_PRIORITY to MAX_PRIORITY) that helps the operating system determine the order in which threads are scheduled.
2. It is represented as integer value ranging from 1 to 10.
3. By default, every thread is given priority NORM_PRIORITY (a constant of 5). Each new thread inherits the priority of the thread that created it.
4. The job of an operating system's thread scheduler is to determine which thread runs next.
5. When a higher-priority thread enters the ready state, the operating system generally preempts the currently running thread.
6. Depending on the operating system, higher-priority threads could postpone the execution of lower-priority threads.

B. **Importance of thread priority :** In multi-threaded environment thread priority is important because it helps the operating system's thread scheduler decide which thread to execute when there are more threads ready to run than there are available CPU cores.

Que 2.26. **Explain how thread priorities influence thread scheduling.**

Answer

In Java, thread priorities influence thread scheduling by providing hints to the thread scheduler about the importance of one thread relative to others. Here's how thread priorities influence thread scheduling :

1. **Thread scheduler :** The underlying operating system's thread scheduler is responsible for scheduling threads for execution on the CPU.
2. **Priority-based scheduling :** When multiple threads are competing for CPU time, the thread scheduler uses thread priorities to determine which thread to execute next. Threads with higher priorities are more likely to be scheduled for execution.
3. **Preemptive scheduling :** In preemptive scheduling, the thread scheduler may preempt a lower-priority thread to allow a higher-priority thread to run. This ensures that more important tasks are given CPU time when needed.

4. **Interactions between threads :** The thread scheduler continuously monitors the status of all threads and may adjust the scheduling of threads based on their priorities and the current system load.

5. **Fairness and starvation :** While higher priority threads are given preference, the thread scheduler also ensures fairness by not completely ignoring lower priority threads. This prevents starvation.

Que 2.27. What is thread synchronization ? Discuss key concepts related to thread synchronization.

Answer

A. **Thread synchronization :**

1. Thread synchronization refers to the coordination of multiple threads to ensure proper and orderly access to shared resources.
2. In a multi-threaded environment, where multiple threads are executing concurrently, synchronization is essential to prevent race conditions.
3. Thread synchronization is typically achieved using synchronization constructs such as locks, mutexes, semaphores, and monitors.
4. These constructs help ensure that only one thread can access a shared resource at a time.
5. Thread synchronization is crucial in multi-threaded applications to maintain data consistency, prevent race conditions, and ensure correct program behavior.

B. **Key concepts related to thread synchronization :**

1. **Critical sections :** These are segments of code that access shared resources or modify shared state. It's essential that only one thread executes critical section at a time to maintain data consistency.
2. **Mutual exclusion :** Mutual exclusion (or mutex) ensures that only one thread can access a critical section at any given time.
3. **Synchronized methods and blocks :** Methods or blocks of code marked as 'synchronized' can only be executed by one thread at a time, ensuring thread safety.
4. **Locks and monitors :** Java provides low-level synchronization constructs such as 'Lock' and 'Condition' interfaces, as well as high-level constructs such as 'synchronized' blocks and 'wait()' methods, to implement thread synchronization.
5. **Inter-thread communication :** Synchronization also involves coordinating communication between threads. This is often achieved using methods like 'wait()', 'notify()', and 'notifyAll()'.

Que 2.28. Discuss the need for synchronizing threads in a multi-threaded environment.

Answer

In a multi-threaded environment the need for synchronizing threads arises due to following reasons :

1. **Prevent data corruption :** Synchronization prevents multiple threads from concurrently modifying shared data, avoiding data corruption.
2. **Avoid race conditions :** Synchronization ensures that the outcome of the program is not affected by the unpredictable interleaving of thread execution.
3. **Maintain data consistency :** Synchronization ensures that changes made by one thread are visible to other threads, maintaining data consistency.
4. **Ensure thread safety :** Synchronization prevents concurrent access to critical sections of code or resources, ensuring correct behavior in a multi-threaded environment.
5. **Prevent deadlocks and livelocks :** Proper synchronization helps avoid deadlocks and livelocks by ensuring that threads can safely acquire and release resources.
6. **Enforce ordering constraints :** Synchronization allows developers to enforce ordering constraints on the execution of threads, ensuring certain operations occur in a specific sequence.
7. **Optimize performance :** Although synchronization introduces overhead, it's crucial for maintaining correctness and reliability, ultimately optimizing the performance of multi-threaded applications.

Que 2.29. Describe various mechanisms for synchronizing threads in Java.

Answer

Following are some of the most commonly used mechanisms :

A. **Synchronized methods and blocks :**

1. Java provides the 'synchronized' keyword to synchronize methods or blocks of code.
2. When a method or block is marked as synchronized, only one thread can execute it at a time.
3. This ensures thread safety and prevents concurrent access to critical sections of code or shared resources.
4. Example :

```
public synchronized void synchronizedMethod() {
    // Synchronized method implementation
}
```

B. **Reentrant locks (java.util.concurrent.locks.Lock) :**

1. Reentrant locks provide a more flexible alternative to synchronized methods and blocks.
2. They allow finer-grained control over locking and unlocking and support features like deadlock avoidance, timeout, and condition variables.
3. Example :

```
Lock lock = new ReentrantLock();
lock.lock(); // Acquire the lock
try {
    // Critical section of code
} finally {
lock.unlock(); // Release the lock
}
```

C. **Semaphore (java.util.concurrent.Semaphore) :**

1. A semaphore is a synchronization primitive that maintains a set of permits.
2. Threads can acquire permits from the semaphore and release them when done.
3. Semaphores are useful for controlling access to a shared resource with limited capacity.
4. Example :

```
Semaphore semaphore = new Semaphore(1); // Initialize with one permit
semaphore.acquire(); // Acquire a permit
try {
        // Critical section of code
} finally {
    semaphore.release(); // Release the permit
}
```

Que 2.30. **What is inter-thread communication ? How is it achieved in Java ?**

Answer

1. Inter-thread communication is the process of coordinating and exchanging information between multiple threads in a multi-threaded application.

2. It allows threads to synchronize their activities, share data, and coordinate their execution in a controlled manner.
3. This is essential for building concurrent programs where threads need to work together to accomplish tasks efficiently.
4. In Java, inter-thread communication is typically achieved using three fundamental methods provided by the Object class: wait(), notify(), and notifyAll().
5. These methods enable threads to wait for certain conditions to be met and to signal other threads when those conditions are satisfied.

Que 2.31. **Provide examples illustrating the use of wait(), notify(), and notifyAll() methods for inter-thread communication.**

Answer

Example 1 :

Using wait() and notify() for single-thread communication :

```
class SharedObject {
    boolean flag = false;
    synchronized void waitForFlagChange() {
        while (!flag) {
            try {
            wait(); // Wait until flag changes
            } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            }
        }
        System.out.println("Flag changed");
        }
        synchronized void changeFlag() {
        flag = true;
        notify(); // Notify waiting thread that flag has changed
        }
}
    public class InterThreadCommunicationExample {
        public static void main(String[] args) {
        SharedObject sharedObject = new SharedObject();
        Thread thread1 = new Thread(() ->
        sharedObject.waitForFlagChange());
```

```
        Thread thread2 = new Thread(() -> sharedObject.changeFlag());
        thread1.start(); // Thread 1 waits for flag change
        thread2.start(); // Thread 2 changes the flag
        }
    }
```

In this example, Thread 1 waits until Thread 2 changes the flag. Once the flag changes, Thread 1 is notified and continues its execution.

Example 2 :

Using wait() and notifyAll() for multi-thread communication :

```
class SharedObject {
    boolean flag = false;
    synchronized void waitForFlagChange() {
    while (!flag) {
        try {
            wait(); // Wait until flag changes
            } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            }
        }
        System.out.println("Flag changed");
        }
    synchronized void changeFlag() {
    flag = true;
        notifyAll(); // Notify all waiting threads that flag has changed
        }
    }
public class InterThreadCommunicationExample {
    public static void main(String[] args) {
    SharedObject sharedObject = new SharedObject();
    Thread thread1 = new Thread(() -> sharedObject.waitForFlagChange());
    Thread thread2 = new Thread(() -> sharedObject.waitForFlagChange());
    Thread thread3 = new Thread(() -> sharedObject.changeFlag());
    thread1.start(); // Thread 1 waits for flag change
    thread2.start(); // Thread 2 waits for flag change
```



```
    thread3.start(); // Thread 3 changes the flag
    }
}
```

In this example, both Thread 1 and Thread 2 wait until Thread 3 changes the flag. Once the flag changes, both Thread 1 and Thread 2 are notified and continue their execution.

3

UNIT

Java New Features

CONTENTS

PART-1

Functional Interfaces.

Que 3.1. What are functional interfaces in Java ? Give some predefined functional interfaces.

Answer

A. Functional interfaces :

1. Functional interfaces in Java are interfaces that contain only one abstract method.
2. Functional interface is also known as Single Abstract Method (SAM) interfaces.
3. They are a key feature introduced in Java to support functional programming paradigms.
4. These interfaces are often used to represent functions or actions, allowing them to be passed around as parameters or returned from methods.

B. Predefined functional interfaces in Java :

1. Some predefined functional interfaces in Java include :
   - i. **java.lang.Runnable :** Represents a task that can be executed concurrently.
   - ii. **java.util.Comparator :** Defines a comparison function for ordering elements.
   - iii. **java.util.function.Predicate :** Represents a predicate (boolean-valued function) of one argument.
   - iv. **java.util.function.Function :** Represents a function that accepts one argument and produces a result.
   - v. **java.util.function.Supplier :** Represents a supplier of results.
2. These predefined functional interfaces provide common functional constructs.
3. They are extensively used in conjunction with lambda expressions and the Stream API.

PART-2

Lambda Expression.

Que 3.5. **Explain lambda expressions in Java and provide examples of their usage.**

Answer

1. Lambda expressions in Java are a concise way to represent anonymous functions (functions without a name).
2. Lambda expressions are similar to methods, but they do not need a name and can be implemented right in the body of a method.
3. The lambda expression is used to provide the implementation of an interface which has functional interface.
4. Lambda expressions are particularly useful when you need to pass behavior (code) as an argument to a method.
5. They are a key feature introduced in Java 8 to support functional programming style.
6. Syntax of Lambda expression :

   (argument-list) -> {body}
7. Java lambda expression consisted of three components :
   i. **Argument-list :** It can be empty or non-empty as well.
   ii. **Arrow-token :** It is used to link arguments-list and body of expression.
   iii. **Body :** It contains expressions and statements for lambda expression.
8. Example :
   i. **Example of a simple lambda expression :**

```
// Lambda expression to print a message
() -> System.out.println("Hello, Lambda!");
```

   ii. **Example of using lambda expression with a functional interface :**

```
// Using a lambda expression with Comparator interface to sort a list of strings
List<String> names = Arrays.asList("Arjun", "Bhim", "Ram", "Hari");
Collections.sort(names, (String a, String b) -> a.compareTo(b));
```

   iii. **Example of using lambda expression with the Stream API:**

```
// Using a lambda expression with Stream API to filter even numbers
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
List<Integer> evenNumbers = numbers.stream()
                                   .filter(num -> num % 2 == 0)
                                   .collect(Collectors.toList());
```


Que 3.6. **How functional interfaces in Java relate to lambda expressions ?**

Answer

1. Functional interfaces in Java are closely related to lambda expressions because lambda expressions can only be used with functional interfaces.
2. Lambda expressions enable the creation of instances of functional interfaces without having to explicitly define a new class.
3. Instead, they allow you to specify the implementation of the single abstract method directly inline.
4. This makes code more concise and readable.
5. Following example illustrate how functional interfaces and lambda expressions are related :

```
// Define a functional interface with a single abstract method
interface MyFunction {
    void performAction(String message);
}
public class Main {
    public static void main(String[] args) {
    // Using a lambda expression to create an instance of MyFunction interface
    MyFunction myFunction = (String message) -> {
    System.out.println("Message: " + message);
    };
    // Calling the method defined in the functional interface
    myFunction.performAction("Hello, Lambda!");
    }
}
```

6. In this example, 'MyFunction' is a functional interface with a single abstract method 'performAction'.
7. We then use a lambda expression '(String message) -> { System.out.println("Message: " + message); }' to provide the implementation of the 'performAction' method directly.
8. This demonstrates how lambda expressions enable concise implementation of functional interfaces in Java.

PART-3

Method References.

Que 3.4. **Explain method references in Java with its types.**

Answer

A. Method references in Java :

1. Method references in Java provide a shorthand syntax for writing lambda expressions that call a single method.
2. They allow you to refer to methods or constructors without invoking them.
3. They provide a more concise and readable alternative to lambda expressions.

B. Types of method references :

1. **Reference to a static method :**
   i. **Syntax :** ClassName::staticMethodName
   ii. **Example :** String::valueOf or Math::max
2. **Reference to an instance method of a particular object :**
   i. **Syntax :** objectReference::instanceMethodName
   ii. **Example :** System.out::println
3. **Reference to an instance method of an arbitrary object of a particular type :**
   i. **Syntax :** ClassName::instanceMethodName
   ii. **Example :** String::length
4. **Reference to a constructor :**
   i. **Syntax :** ClassName::new
   ii. **Example :** ArrayList::new or String::new

Que 3.5. **How do method references simplify code in Java ? Provide examples.**

Answer

Following are several ways method references simplify code :

A. Readability :

1. Method references often provide a clearer and more intuitive representation of the code's intent compared to lambda expressions, especially when the method being referenced has a descriptive name.
2. Example :

```
// Lambda expression
numbers.forEach(number -> System.out.println(number));
// Method reference
numbers.forEach(System.out::println);
```


B. Elimination of redundant code :

1. Method references help eliminate redundancy by allowing you to directly reference existing methods instead of rewriting similar code in lambda expressions.
2. Example :

```
// Lambda expression
List<String> upperCaseStrings = strings.stream()
    .map(string -> string.toUpperCase())
    .collect(Collectors.toList());
// Method reference
List<String> upperCaseStrings = strings.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());
```

C. Improved maintainability :

1. Using method references can make code easier to maintain because they explicitly indicate the method being invoked, making it easier for other developers to understand and modify the code.
2. Example :

```
// Lambda expression
numbers.stream()
    .map(number -> Math.sqrt(number))
    .forEach(System.out::println);
// Method reference
numbers.stream()
    .map(Math::sqrt)
    .forEach(System.out::println);
```

D. Simplified constructor invocation :

1. Method references can simplify the instantiation of objects by referencing constructors directly, eliminating the need for lambda expressions to call constructors.
2. Example :

```
// Lambda expression
Supplier<List<String>> listSupplier = () -> new ArrayList<>();
// Method reference
Supplier<List<String>> listSupplier = ArrayList::new;
```

PART-4

Stream API.

Que 3.6. Describe stream API. Write down the features of stream API.

Answer

A. **Stream API :**

1. The Stream API in Java provides a powerful and flexible way to process collections of objects in a functional style.
2. It allows developers to perform aggregate operations on sequences of elements with concise and expressive syntax.

B. **Features of Stream API :** Following are the key features of the Stream API :

1. **Declarative and functional style :** Stream API encourages a declarative and functional programming style.
2. **Lazy evaluation :** Stream operations are typically lazy, meaning that they are not evaluated until a terminal operation is called. This enables efficient processing of large datasets.
3. **Parallel execution :** Stream API supports parallel execution of operations, allowing concurrent processing of elements when appropriate.
4. **Interoperability :** Stream API seamlessly integrates with existing collections framework in Java.
5. **Pipelining :** Stream operations can be chained together to form a pipeline. This enables concise and expressive code for complex data transformations.
6. **Common Operations :** Stream API provides a wide range of intermediate and terminal operations for common data processing tasks.

Que 3.7. Discuss the purpose and usage of the Stream API in Java.

Answer

A. **Purpose of Stream API :**

1. **Data processing :** The primary purpose of the Stream API is to facilitate efficient processing of data, such as lists, arrays, or other data structures.

2. **Functional programming :** Stream API promotes a functional programming style leading to more concise and readable code.
3. **Concurrency :** Stream API supports parallel execution of operations, enabling concurrent processing of data.
4. **Intermediate and terminal operations :** Stream API provides a wide range of intermediate and terminal operations for common data processing tasks.

B. **Usage of Stream API :**

1. **Creating streams :** Streams can be created from various data sources using factory methods provided by the 'Stream' interface.
2. **Intermediate operations :** Intermediate operations transform or filter the elements of a stream and return a new stream as a result.
3. **Terminal operations :** Terminal operations consume the elements of a stream and produce a final result, such as a value, a collection, or an action.
4. **Chaining operations :** Stream operations can be chained together to form a pipeline, where the output of one operation serves as the input for the next.

PART-5

Default Methods, Static Method.

Que 3.8. What are default methods in Java interfaces ? Give key features of default methods.

Answer

A. **Default methods :**

1. Default methods are a feature introduced in Java 8 to enable adding new methods to interfaces without breaking existing implementations.
2. Prior to Java 8, interfaces in Java could only declare abstract methods.
3. This means that any new method added to an interface would require all implementing classes to provide an implementation for that method.
4. Default methods address this limitation by allowing interfaces to provide default implementations for methods.

B. **Key features of default methods :** Key features of default methods in Java interfaces include :

1. **Default method syntax :** A default method is defined using the 'default' keyword followed by the method signature and implementation.
2. **Backward compatibility :** Default methods enable backward compatibility.
3. **Override default methods :** Classes implementing the interface can choose to override default methods if they need custom behavior.
4. **Multiple inheritance :** Default methods enable multiple inheritance of behavior in Java interfaces.
5. **Extending interfaces :** Interfaces can extend other interfaces and inherit their default methods.

Que 3.9. **How do default methods enable backward compatibility?**

Answer

1. Default methods enable backward compatibility by allowing existing interfaces to evolve without breaking existing implementations.
2. Existing classes that implement the interface will automatically inherit the default implementation for the new method.
3. This means that classes implementing the interface do not need to provide an implementation for the new method unless they choose to override it.

Que 3.10. **Explain the concept of static methods in Java interfaces. What are the features of static methods ?**

Answer

A. Static methods :

1. Static methods are a feature introduced in Java 8 to allow interfaces to contain static methods.
2. Prior to Java 8, interfaces could only declare instance methods.
3. Any method declared in an interface was implicitly abstract and instance-based.
4. With the introduction of static methods in interfaces, Java interfaces gained the ability to declare static methods.
5. These static methods can be called directly on the interface itself, without requiring an instance of the implementing class.
6. Overall, static methods in Java interfaces enhance the flexibility, utility, and expressiveness of Java interfaces.
7. They play a crucial role in improving code organization and promoting code reuse.

B. Features of static methods : Following are the key features of static methods :

1. **Belong to the class :** Static methods are associated with the class in which they are defined rather than with instances of the class.
2. **No access to instance variables :** Static methods cannot access instance variables directly because they do not operate on any particular instance of the class.
3. **Cannot be overridden :** Unlike instance methods, static methods cannot be overridden by subclasses.
4. **Can access other static members :** Static methods can access other static members of the class directly without the need for an instance reference.
5. **Memory allocation :** Static methods are loaded into memory along with the class definition, regardless of whether any instances of the class are created.

PART-6

Base64 Encode and Decode.

Que 3.11. **Explain Base64 encode and decode in detail.**

Answer

1. Base64 encoding and decoding are techniques used to convert binary data into ASCII characters and vice versa.
2. This conversion is commonly used in scenarios where binary data needs to be represented as text.

A. Base64 encoding :

1. Base64 encoding is a method of encoding binary data into a text format using a set of 64 ASCII characters.
2. The process involves dividing the input binary data into groups of 6 bits, which are then mapped to one of the 64 printable ASCII characters.
3. Each group of 6 bits corresponds to one character in the Base64 alphabet.
4. The Base64 alphabet consists of 64 characters, typically the following :
   i. A-Z (26 characters)
   ii. a-z (26 characters)
   iii. 0-9 (10 characters)

iv. '+' (plus sign)

v. '/' (forward slash)

vi. Additionally, a padding character '=' may be used to ensure that the input binary data can be evenly divided into groups of 6 bits.

B. Base64 decoding :

1. Base64 decoding is the reverse process of Base64 encoding.
2. It involves converting Base64-encoded text back into its original binary representation.

Que 3.12. How can you perform Base64 encoding and decoding in Java ?

Answer

A. Base64 encoding in Java : The steps for Base64 encoding in Java are as follows :

1. Divide the input binary data into groups of 6 bits.
2. Convert each 6-bit group into a decimal value.
3. Map each decimal value to the corresponding character in the Base64 alphabet.
4. If necessary, pad the encoded data with '=' characters to ensure it is a multiple of 4 characters in length.
5. Here's an example of Base64 encoding in Java :

```
import java.util.Base64;
public class Base64Example {
    public static void main(String[] args) {
    // Input binary data
    byte[] binaryData = "Hello, World!".getBytes();
    // Encode the binary data to Base64
    String base64Encoded =
            Base64.getEncoder().encodeToString(binaryData);
    System.out.println("Base64 Encoded: " + base64Encoded);
    }
}
```

B. Base64 decoding in Java : The steps for Base64 decoding in Java are as follows :

1. Convert each character in the Base64-encoded text back to its corresponding 6-bit binary value.
2. Combine the binary values into groups of 8 bits.

3. Convert each group of 8 bits into a byte.
4. Repeat steps 1-3 for each character in the Base64-encoded text.
5. Remove any padding characters '=' from the end of the decoded data.
6. Here's an example of Base64 decoding in Java :

```
import java.util.Base64;
public class Base64Example {
    public static void main(String[] args) {
        // Base64 encoded text
    String base64Encoded = "SGVsbG8sIFdvcmxkIQ==";
    // Decode the Base64-encoded text
    byte[] decodedBytes =
                    Base64.getDecoder().decode(base64Encoded);
    String decodedString = new String(decodedBytes);
    System.out.println("Decoded String: " + decodedString);
    }
}
```

PART-7

ForEach Method, Try-with-resources.

Que 3.13. Describe the forEach method in Java and its significance in collections processing.

Answer

A. forEach method in Java :

1. In Java, the forEach method is a terminal operation provided by the Stream API.
2. It allows you to perform a specified action for each element in a stream, iterating over the elements sequentially.
3. The forEach method accepts a functional interface as an argument, which defines the action to be performed on each element of the stream.
4. Here's the general syntax of the forEach method :

```
stream.forEach(action);
```

where,

stream : The stream of elements over which the operation will be performed.

action : A functional interface representing the action to be performed on each element of the stream.

B. **Significance of forEach method in collections processing :**

1. **Concise and readable code :** The forEach method provides a concise and expressive way to iterate over elements of a collection or a stream.
2. **Functional programming paradigm :** The forEach method promotes a functional programming style.
3. **Declarative style :** Using the forEach method allows you to express the operation to be performed on each element in a declarative manner.
4. **Parallel execution :** In addition to sequential execution, the forEach method supports parallel execution when used with parallel streams.
5. **Stream integration :** The forEach method seamlessly integrates with other Stream API operations.

Que 3.14. What is the try-with-resources statement in Java ? How does it simplify resource management ?

Answer

A. **try-with-resources statement in Java :**

1. The try-with-resources statement is a feature use to simplify the management of resources that require closing or releasing after being used.
2. It allows you to declare one or more resources within the parentheses of the try statement.
3. It also ensures that these resources are automatically closed at the end of the try block, regardless of whether an exception occurs or not.
4. Following is the general syntax of the try-with-resources statement :

```
try (resource initialization) {
    // Code that uses the resource
} catch (ExceptionType e) {
    // Exception handling
}
```

B. **Simplification of resource management :**

1. The try-with-resources statement in Java simplifies resource management by automating the process of closing resources and reducing the likelihood of resource leaks.

2. It improves code readability and enhances the robustness of exception handling in Java programs.

PART-8

Type Annotations, Repeating Annotations.

Que 3.15. Explain type annotations and their role in Java.

Answer

A. **Type annotations in Java :**

1. Type annotations in Java allow developers to apply annotations to various types in addition to just declarations.
2. These annotations provide additional metadata about types, which can be used by tools and frameworks.
3. Type annotations can be applied to a wide range of program elements.

B. **Role of type annotations in Java :**

1. **Providing additional type information :** Type annotations provide additional metadata about types, which can be used by tools and frameworks to perform various tasks.
2. **Custom annotations :** Developers can define custom type annotations to express domain-specific constraints or requirements.
3. **Improved code quality and safety :** Type annotations help improve code quality, safety, and maintainability by providing additional compile-time and runtime checks.

Que 3.16. What are repeating annotations, and how are they used in Java ?

Answer

A. **Repeating annotations :**

1. Repeating annotations, introduced in Java SE 8, allow multiple instances of the same annotation to be applied to a single program element.
2. Prior to Java SE 8, each annotation could only be applied once to a given element, which limited their flexibility in certain scenarios.
3. Repeating annotations address this limitation by allowing annotations to be repeated.
4. Repeating annotations provide a more flexible and expressive way to annotate program elements.

5. It improves code readability and maintainability in situations where multiple annotations of the same type are needed.

B. **Repeating annotations use in Java :** Here's how repeating annotations are used in Java :

1. **Declaring repeating annotations :** To declare a repeating annotation type, you use the '@Repeatable' meta-annotation along with the container annota. on type.
2. **Applying repeating annotations :** Once a repeating annotation type is declared, you can apply it to program elements using its container annotation.

PART-9

Java Module System.

Que 3.17. Discuss the Java Module System and its advantages.

Answer

A. **Java Module System :**

1. The Java Module System, introduced in Java 9, provides a way to modularize Java applications by encapsulating code into discrete units called modules.
2. This allows developers to organize codebase into logical units with well-defined dependencies and boundaries.
3. The Java Module System offers significant advantages for developing and maintaining large-scale Java applications.
4. By promoting modular design, encapsulation, and dependency management, it helps improve code quality, scalability, and security.

B. **Advantages of Java Module System :**

1. **Modularization :** The Java Module System allows developers to partition their codebase into modules.
2. **Encapsulation :** Modules enforce strong encapsulation, allowing developers to control which types and members are accessible outside the module.
3. **Improved performance :** The Java Module System improves application startup time and reduces memory footprint.
4. **Isolation and security :** Modules provide a level of isolation. This enhances security by preventing unauthorized access to internal APIs.
5. **Scalability :** Modular applications are easier to scale and maintain as they grow.

PART-10

Diamond Syntax with Inner Anonymous Class.

Que 3.18. What is diamond syntax in Java ?

Answer

1. Diamond syntax, also known as the diamond operator (<>), is a feature introduced in Java 7.
2. It allows you to instantiate generic classes without explicitly specifying the type arguments.
3. The diamond syntax is used when the type arguments to a generic class can be inferred from the context in which the class is being instantiated.
4. Here's an example to illustrate the use of diamond syntax :

```
// Before Java 7
List<String> list1 = new ArrayList<String>();
// With diamond syntax (Java 7 and later)
List<String> list2 = new ArrayList<>();
```

5. The diamond syntax is particularly useful in situations where the type arguments are redundant or obvious from the context.
6. Diamond syntax can only be used with anonymous inner classes and generic class instantiation.
7. It cannot be used with constructors that have no arguments.
8. Additionally, diamond syntax does not introduce any new type inference capabilities.

Que 3.19. What do you mean by inner anonymous classes in Java ?

Answer

1. Inner anonymous classes in Java are unnamed classes defined and instantiated inline, typically within the body of another class or method.
2. They are used for providing one-off implementations of interfaces or extending classes without the need to define separate named classes.
3. These classes are limited in scope and have access to enclosing method's local variables if they are effectively final or explicitly declared as final.
4. Inner anonymous classes are handy for creating quick implementations of interfaces or extending classes without adding extra named classes.
5. However, they should be used carefully to avoid cluttering code and compromising readability and maintainability.

Que 3.20. **How does the diamond syntax work with inner anonymous classes in Java ?**

Answer

1. Here's an example of how the diamond syntax can be used with inner anonymous classes :

```
// Example with ArrayList and inner anonymous class
List<String> list = new ArrayList<>() {
    // Inner anonymous class with initializer block
{
    add("Hello");
    add("World");
    }
};
```

2. In this example we're creating an instance of ArrayList using the diamond syntax (<>) without specifying the type arguments.
3. Inside the curly braces {}, we define an inner anonymous class that extends ArrayList.
4. This class contains an initializer block where we add elements to the list.
5. The compiler infers that the type argument for ArrayList should be String, based on the type of the variable list declared as List<String>.

PART-11

Local Variable Type Inference.

Que 3.21. **What is local variable type inference, and how does it improve code readability ?**

Answer

A. Local variable type inference :

1. Local variable type inference is a feature in Java 10 that allows the compiler to infer the data type of a variable based on the value assigned to it.
2. This allows the developer to skip the type declaration associated with local variables.

3. So, instead of explicitly stating the type of a variable, we can simply use the var keyword and let the compiler figure out the type based on the context.
4. For example, in Java, instead of writing :

```
List<String> names = new ArrayList<String>();
```

We can write :

```
var names = new ArrayList<String>();
```

5. The compiler will infer that 'names' is of type 'ArrayList<String>' based on the assignment.

B. Improving code readability : Local variable type inference improves code readability in following ways :

1. **Conciseness :** By reducing boilerplate code related to type declarations, local variable type inference makes code shorter and more concise.
2. **Focus on intent :** It allows developers to focus more on the intent of their code rather than the details of types.
3. **Reduced redundancy :** With type inference, you don't need to repeat type names, which can reduce redundancy.
4. **Improved maintainability :** As code becomes more concise and focused on intent, it tends to become easier to maintain.

PART-12

Switch Expressions.

Que 3.22. **Explain switch expressions in Java and how they differ from traditional switch statements.**

Answer

1. Switch expressions were introduced in Java 12 as a preview feature and became a permanent feature starting from Java 14.
2. They provide a more concise and expressive way to write switch statements compared to traditional switch statements.

Difference :

S. No.	Feature	Switch statements	Switch expressions
1.	Syntax	Uses switch keyword, case labels, code blocks.	Uses switch keyword, -> arrow syntax, expression-based cases.
2.	Return values	Each case may contain statements.	Each case must provide a value to return.
3.	Use of break	Typically requires break statements to prevent fall-through.	No fall-through behavior by default, no need for break.
4.	Default case	Optional, can be omitted.	Mandatory, must be present with a value to return.

PART-13

Yield Keyword.

Que 3.23. What is the 'yield' keyword in Java ? Provide examples of its usage.

Answer

1. The 'yield' keyword in Java is used within switch expressions to specify the value to be returned from a case.
2. It indicates the result of evaluating a particular case and terminates the execution of the switch expression.
3. 'yield' can only be used within switch expressions, and it's not permitted in other contexts in Java.
4. It's specifically designed to work with switch expressions to provide a concise way to return values.
5. Following is an example demonstrating the usage of 'yield' keyword :

```
int dayNumber = 3;
String dayName = switch (dayNumber) {
    case 1 -> "Monday";
    case 2 -> "Tuesday";
    case 3 -> "Wednesday";
    case 4 -> "Thursday";
    case 5 -> "Friday";
    default -> {
```



```
        // Handle invalid day numbers
        yield "Invalid day";
    }
};
System.out.println("The day is: " + dayName);
```

6. In this example, the 'yield' keyword is used within the 'default' case to return the value "Invalid day".
7. The switch expression terminates at the 'yield' statement, and the value is assigned to the variable 'dayName'.

PART-14

Text Blocks.

Que 3.24. Discuss text blocks in Java and their benefits in string manipulation.

Answer

A. **Text blocks :**

1. Text blocks were introduced in Java 13 as a preview feature and made a permanent feature starting from Java 15.
2. Text blocks provide a more natural way to write multiline strings in Java.
3. Instead of concatenating multiple string literals or using escape characters for line breaks, you can use text blocks to represent multiline strings directly in your code.
4. Here's a simple example of a text block :

```
String html = """
        <html>
            <body>
                <p>Hello, world!</p>
            </body>
        </html>
    """;
```

5. In this example, the triple double quotes (""") indicate the start and end of the text block.
6. The content within the text block, including the indentation, is preserved as-is.

B. **Benefits :** Benefits of text blocks in string manipulation include :

1. **Readability :** Text blocks improve code readability by representing multiline strings in a more natural and readable format.
2. **Maintainability :** With text blocks, multiline strings are easier to maintain because the indentation and formatting are preserved.
3. **String manipulation :** Text blocks facilitate string manipulation operations, such as substring extraction, searching, or replacement.
4. **Reduced boilerplate :** Text blocks reduce the amount of boilerplate code. This leads to cleaner, more concise code.

PART-15

Records.

Que 3.25. What are records in Java, and how do they simplify the creation of immutable data ?

Answer

A. **Records in Java :**

1. A record in Java is a compact way to declare classes that are mainly intended to hold immutable data.
2. They provide a concise syntax for declaring classes that are primarily used to model data rather than behavior.
3. They promote best practices for modeling data, making code more maintainable and easier to understand.
4. Records were introduced as a preview feature in Java 14 and became a permanent feature starting from Java 16.
5. Here's a basic example of a record declaration :

```
public record Point(int x, int y) {}
```

6. This single line of code declares a record named 'Point' with two components 'x' and 'y'.
7. Behind the scenes, the Java compiler automatically generates several standard methods, such as 'equals()', 'hashCode()', and 'toString()', as well as a constructor to initialize the record's components.

B. **Simplifying the creation of immutable data :** Records simplify the creation of immutable data in following ways :

1. **Concise syntax :** Records provide a concise syntax for declaring classes that are primarily used to hold data.

2. **Implicit finality :** By default, all components of a record are implicitly final. This ensures that instances of the record are immutable.
3. **Compact initialization :** Records generate a compact constructor to initialize the record's components. This allows you to create instances of the record using a concise syntax.

PART-16

Sealed Classes.

Que 3.26. Explain sealed classes in Java and their role in controlling inheritance hierarchies.

Answer

A. **Sealed classes :**

1. Sealed classes provide a way to restrict which classes can be subclasses of a particular class or interface.
2. They allow you to define a finite set of classes that are permitted to extend or implement a sealed class or interface.
3. Sealed classes in Java provide a powerful mechanism for controlling inheritance hierarchies, enhancing encapsulation, and improving API design.
4. They promote stronger type safety, better code organization, and easier maintenance of complex class hierarchies.
5. Following is a basic example of a sealed class declaration :

```
public sealed class Shape permits Circle, Rectangle, Triangle {
    // Class definition
}
```

6. In this example, 'Shape' is declared as a sealed class that permits only 'Circle', 'Rectangle', and 'Triangle' to be its direct subclasses.

B. **Controlling inheritance hierarchies in Java :** Sealed classes play a significant role in controlling inheritance hierarchies in Java in following ways :

1. **Limited subclassing :** Sealed classes restrict the set of classes that can extend them. This limits the inheritance hierarchy to a predefined set of subclasses.
2. **Enhanced encapsulation :** By explicitly specifying which classes can extend a sealed class, you can enforce stronger encapsulation and control over the design of your code.

3. **Pattern matching :** Sealed classes work seamlessly with pattern matching. Pattern matching allows you to easily handle different subclasses of a sealed class with clarity and brevity.
4. **Improved API design :** Sealed classes encourage better API design by explicitly declaring the intended subclasses upfront.
5. **Future extensibility :** Sealed classes can be extended in future versions of the code by adding new permitted subclasses.

☺☺☺

4 UNIT

Java Collections Framework

CONTENTS

PART-1

Collection in Java , Collection Framework in Java.

Que 4.1. **What do you mean by Collections in Java ? Also give key features of Collections.**

Answer

A. Collections in Java :

1. In Java, "Collections" refers to a framework provided in the Java API to manage and manipulate groups of objects.
2. These objects are commonly referred to as elements or items.
3. The Collections framework provides a unified architecture for working with collections of objects.
4. It allows developers to easily store, retrieve, manipulate, and iterate over these collections.

B. Key features of Collections :

1. **Unified architecture :** The Collections framework provides a unified architecture for representing and manipulating collections.
2. **Generics support :** Most classes and interfaces in the Collections framework support generics, allowing developers to specify the type of elements a collection can hold.
3. **Dynamic resizing :** Many collection classes automatically resize themselves as needed to accommodate the addition or removal of elements.
4. **Algorithms :** The Collections framework includes various utility methods and algorithms for sorting, searching, and manipulating collections efficiently.
5. **Iterators :** Iterators provide a way to traverse the elements of a collection sequentially, enabling easy iteration over collection elements.

Que 4.2. **What is the Java Collections framework ? Give its advantages.**

Answer

A. Java Collections framework :

1. The Java Collections framework is a set of classes and interfaces that provide implementations of commonly reusable collection data structures in Java.
2. The data structures can be lists, sets, maps, queues, etc.
3. It provides a unified architecture for manipulating and storing groups of objects.

B. Advantages of the Java Collections framework :

1. **Reusable implementations :** The framework provides ready-to-use implementations of common data structures, saving developers time and effort.
2. **Consistency :** All collections in the framework follow a common set of interfaces and conventions.
3. **Efficiency :** The framework offers efficient implementations of data structures, optimized for various use cases.
4. **Type safety :** Java generics are extensively used in the collections framework, providing compile-time type safety.
5. **Scalability :** The framework supports scalable data structures, enabling developers to handle large datasets efficiently.

PART-2

Hierarchy of Collection Framework.

Que 4.3. **Explain the hierarchy of the Collection framework in Java.**

Answer

1. The utility package, (java.util) contains all the classes and interfaces that are required by the collection framework.
2. The collection framework contains an interface named an iterable interface which provides the iterator to iterate through all the collections.
3. This interface is extended by the main collection interface which acts as a root for the collection framework.
4. All the collections extend this collection interface thereby extending the properties of the iterator and the methods of this interface.
5. The following figure illustrates the hierarchy of the collection framework :

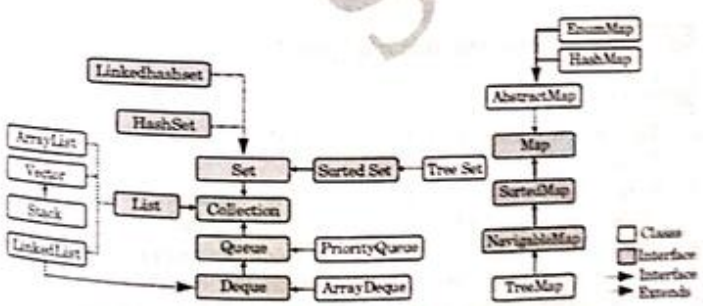


Fig. 4.3.1.

PART-3

Iterator Interface, Collection Interface, List Interface.

Que 4.4. What is Iterator interface ? What is the purpose of the Iterator interface in Java Collections framework ?

Answer

A. **Iterator interface :**

1. This is the root interface for the entire collection framework.
2. The Iterator interface is used to traverse through elements of a collection.
3. It provides a uniform way to access elements of various collection types without exposing the underlying implementation details.

B. **Purpose of the Iterator interface :** The purpose of the Iterator interface can be summarized as follows :

1. **Traversal :** It allows sequential access to the elements of a collection. This enables iterating over the elements of a collection without needing to know its internal structure.
2. **Uniformity :** Iterator provides a common way to iterate over different types of collections. This uniformity simplifies the process of iterating through collections and promotes code reusability.
3. **Safe removal :** Iterator supports safe removal of elements from a collection during iteration. This prevents concurrent modification exceptions.
4. **Enhanced for loop support :** The Iterator interface is utilized implicitly in the enhanced 'for' loop syntax. This syntax provides a concise and readable way to iterate over collections without explicitly dealing with iterators.

Que 4.5. What is Collection interface ?

Answer

1. The Collection interface is the root interface in the Java Collections framework hierarchy.
2. It represents a group of objects, known as elements, and provides a unified way to work with collections of objects in Java.
3. The Collection interface defines a set of operations that can be performed on collections, regardless of their specific implementation.
4. The Collection interface does not specify any particular ordering of elements.
5. The Collection interface allows duplicate elements.

6. Overall, the Collection interface serves as a fundamental building block for working with collections of objects in Java.
7. It provides a common set of operations and behaviors that can be used across different types of collections.

Que 4.6. Differentiate between Collection and Collections in Java.

Answer

S.No.	Aspect	Collection	Collections
1.	Purpose	Represents a group of objects (interface).	Provides utility methods for working with collections class.
2.	Usage	Interface used for defining collection behaviors.	Utility class providing static methods for collection manipulation.
3.	Genericity	Can be parameterized with a type.	Not parameterized (operates on collections generically).
4.	Instances	Collections can be implemented by various classes.	Collections class itself cannot be instantiated.
5.	Examples	List, Set, Queue, etc.	Methods like sort, reverse, shuffle, etc.

Que 4.7. What is List interface ? What are the key characteristics of the List interface ?

Answer

A. **List interface :**

1. This is a child interface of the collection interface.
2. This interface is dedicated to the data of the list type in which we can store all the ordered collections of the objects.
3. This also allows duplicate data to be present in it. This list interface is implemented by various classes like ArrayList, Vector, Stack, etc.
4. Since all the subclasses implement the list, we can instantiate a list object with any of these classes.

B. **Key characteristics :** Following are the key characteristics of the List interface :

1. **Ordered collection :** Lists maintain the order of elements in which

they are inserted.

2. **Indexed access :** Elements in a list can be accessed by their index.
3. **Dynamic size :** Lists are resizable, meaning they can grow or shrink dynamically.
4. **Iterable :** Lists implement the Iterable interface, which means they can be traversed using iterators.
5. **Search operations :** Lists support search operations to find the index of a specified element.

PART-4

ArrayList, LinkedList, Vector, Stack.

Que 4.8. Which classes implement link interface ?

OR

Explain ArrayList, LinkedList, Vector, Stack in reference to link interface.

Answer

The classes that implement the list interface are :

A. **ArrayList :**

1. **Implementation :** ArrayList is a dynamic array-based implementation of the List interface.
2. **Data structure :** It internally uses an array to store elements, which allows for fast random access and retrieval of elements by index.
3. **Performance :** ArrayList provides O(1) time complexity for adding or retrieving elements by index. However, inserting or deleting elements in the middle of the list can be slower due to the need to shift elements.
4. **Use cases :** ArrayList is suitable for scenarios where random access and retrieval of elements are frequent, and the list size is expected to remain relatively stable.

B. **LinkedList :**

1. **Implementation :** LinkedList is a doubly linked list-based implementation of the List interface.
2. **Data structure :** It consists of a sequence of elements, where each element is stored in a node that contains a reference to the previous and next elements in the sequence.
3. **Performance :** LinkedList provides O(1) time complexity for adding or removing elements at the beginning or end of the list. However, accessing

elements by index requires traversing the list, resulting in $O(n)$ time complexity.

4. **Use cases :** LinkedList is suitable for scenarios where frequent insertion or deletion of elements is required, especially at the beginning or end of the list.

C. **Vector :**

1. **Implementation :** Vector is a synchronized, dynamic array-based implementation of the List interface.
2. **Data structure :** Similar to ArrayList, Vector internally uses an array to store elements.
3. **Performance :** Vector provides the same performance characteristics as ArrayList. However, Vector ensures thread safety by synchronizing access to its methods, which can impact performance in multi-threaded environments.
4. **Use cases :** Vector is suitable for scenarios where thread safety is a concern and multiple threads need to access or modify the list concurrently.

D. **Stack :**

1. **Implementation :** Stack is a subclass of Vector and represents a Last-In-First-Out (LIFO) stack data structure.
2. **Data structure :** It supports two main operations: push and pop.
3. **Performance :** Stack inherits its performance characteristics from Vector, with the additional overhead of managing stack-specific operations.
4. **Use cases :** Stack is suitable for implementing algorithms and applications that require LIFO behavior, such as expression evaluation, backtracking, and undo mechanisms.

Que 4.9. Compare and contrast ArrayList, LinkedList, Vector, and Stack classes in Java.

Answer

No.	Feature	ArrayList	LinkedList	Vector	Stack
1.	Implementation	Dynamic array based	Doubly linked list based	Dynamic array based, synchronized	Subclass of Vector, LIFO structure
2.	Data Structure	Array	Doubly linked list	Array, resizable	Array, resizable, synchronized
3.	Thread Safety	Not synchronized	Not synchronized	Synchronized	Synchronized
4.	Use Cases	Random access to elements	Frequent insertion/deletion at beginning/end	Thread-safe operations	LIFO operations
5.	Performance vs Usage	Efficient for random access	Efficient for insertion/deletion at beginning/end	Provides thread safety, suitable for multi-threaded environments	Suitable for LIFO operations, thread-safe version of Vector

PART-5

Queue Interface.

Que 4.10. **Describe queue interface. Which classes implement queue interface ?**

Answer

A. Queue interface :

1. The Queue interface in Java represents a collection of elements that follows the First-In-First-Out (FIFO) principle.
2. The Queue interface extends the Collection interface and adds specific methods for adding, removing, and inspecting elements in a queue.

B. Key characteristics of the Queue interface :

1. **FIFO ordering :** Elements are inserted at the end of the queue and removed from the front, maintaining the order in which they were added.
2. **Adding and removing elements :** Provides methods for adding elements to the end of the queue (`offer()`, `add()`) and removing elements from the front (`poll()`, `remove()`).
3. **Peeking :** Allows accessing the element at the front of the queue without removing it (`peek()`).
4. **Size and empty check :** Methods to check the size of the queue (`size()`) and whether it is empty (`isEmpty()`).

C. Classes that implement the Queue interface :

1. **LinkedList :** LinkedList class implements the Queue interface and provides a doubly linked list-based implementation of a queue.
2. **PriorityQueue :** PriorityQueue class implements the Queue interface using a priority heap.
3. **ArrayDeque :** ArrayDeque class implements the Deque interface, which extends the Queue interface.

PART-6

Set Interface, HashSet, LinkedHashSet.

Que 4.11. **What do you mean by Set interface ? Which classes implement Set interface ?**

Answer

A. **Set interface :**

1. The Set interface in Java represents a collection of unique elements.
2. Meaning that no two elements in the set can be equal according to the equals() method.
3. Sets do not allow duplicate elements.
4. Also they do not maintain the insertion order of elements.

B. **Key characteristics of the Set interface :**

1. **Uniqueness :** Sets contain only unique elements. If an element is added to a set and it already exists in the set, the add operation will have no effect.
2. **No ordering :** Sets do not maintain the order in which elements are inserted.
3. **Equality :** Elements in a set are compared for equality using the equals() method.
4. **No index :** Sets do not support index-based access to elements. Elements can only be accessed through iteration or specific search operations.

C. **Classes that implement the Set interface :**

1. **HashSet :** HashSet is a widely-used implementation of the Set interface that stores elements in a hash table.
2. **TreeSet :** TreeSet is an implementation of the Set interface that stores elements in a sorted tree structure (specifically, a red-black tree).
3. **LinkedHashSet :** LinkedHashSet is an implementation of the Set interface that maintains the insertion order of elements, in addition to ensuring uniqueness. It achieves this by using a hash table and a linked list.

Que 4.12. Write a short note on HashSet and LinkedHashSet.

Answer

A. **HashSet :**

1. HashSet is an implementation of the Set interface in Java that stores elements in a hash table.
2. It provides constant-time average-case performance for basic operations like add, remove, and contains, making it highly efficient for handling large datasets.
3. HashSet does not guarantee the order of elements, as it uses hashing to store elements and does not maintain insertion order.
4. Key features of HashSet include :
   i. **Uniqueness :** HashSet ensures that no duplicate elements are present within the collection.

   ii. **Efficient lookup :** HashSet provides fast lookup operations, due to its underlying hash table implementation.
   iii. **No ordering :** Elements in a HashSet are not stored in any particular order.

B. **LinkedHashSet :**

1. LinkedHashSet combines the features of HashSet with predictable iteration order.
2. It maintains a doubly linked list alongside the hash table, ensuring that elements are stored in the order they were inserted.
3. This allows LinkedHashSet to provide predictable iteration order while still offering fast lookup operations.
4. Key features of LinkedHashSet include :
   i. **Uniqueness :** Like HashSet, LinkedHashSet ensures that no duplicate elements are present within the collection.
   ii. **Predictable iteration order :** Unlike HashSet, LinkedHashSet guarantees that elements will be iterated over in the order they were inserted.
   iii. **Efficient lookup :** While LinkedHashSet maintains insertion order, it still provides efficient lookup operations similar to HashSet.

Que 4.13. Discuss the significance of the Set interface and its implementations like HashSet, LinkedHashSet, and TreeSet.

Answer

The Set interface and its implementations (HashSet, LinkedHashSet, and TreeSet) are vital in Java programming, especially when uniqueness and unordered collections are needed. Here's why :

1. **Uniqueness :** Sets ensure no duplicates within the collection, crucial for scenarios like unique identifiers or eliminating duplicates from datasets.
2. **Efficient lookup :** HashSet, LinkedHashSet, and TreeSet offer efficient element lookup. HashSet uses hashing for constant-time performance, TreeSet uses a sorted tree for binary search operations, and LinkedHashSet combines HashSet's efficiency with predictable iteration order.
3. **Iterating over unique elements :** Sets allow for efficient iteration over unique elements, beneficial when processing each element once without duplication.
4. **Use in Collections framework :** Sets are integral to the Java Collections framework, enabling flexible manipulation of unique collections alongside other collection types like lists and maps.

5. **Implementation flexibility :** HashSet, LinkedHashSet, and TreeSet provide different performance and iteration characteristics. Developers can choose based on specific requirements.

Que 4.14. **How does the Queue interface differ from the Set interface in Java Collections framework ?**

Answer

S.No.	Feature	Queue interface	Set interface
1.	Purpose	Represents a collection of elements with FIFO ordering.	Set interface represents a collection of unique elements.
2.	Ordering	Maintains FIFO (First-In-First-Out) ordering.	No specific ordering
3.	Duplicates	Allows duplicates	Does not allow duplicates.
4.	Key Methods	offer(), poll(), peek()	add(), remove(), contains()
5.	Common Implementations	LinkedList, PriorityQueue	HashSet, LinkedHashSet, TreeSet

PART-7

SortedSet Interface, TreeSet.

Que 4.15. **Describe SortedSet interface. Also give its key characteristics.**

Answer

A. SortedSet interface :

1. The SortedSet interface in Java represents a special type of Set that maintains its elements in sorted order.
2. It extends the Set interface and adds methods for accessing and manipulating elements based on their sorted order.
3. The SortedSet interface provides a powerful way to work with sorted collections of unique elements in Java.
4. It offers efficient access to elements based on their sorted order.

5. It is commonly used in scenarios where sorted collections are required, such as maintaining sorted dictionaries, sorted lists, or implementing algorithms that require sorted data.

B. Key characteristics :

1. **Sorted order :** Elements in a SortedSet are stored in sorted order.
2. **Uniqueness :** Like other Set implementations, a SortedSet does not allow duplicate elements.
3. **Efficient retrieval :** SortedSet provides efficient methods for retrieving elements based on their sorted order.
4. **No index-based access :** Unlike List implementations, SortedSet does not support index-based access to elements.

Que 4.16. **Explain TreeSet. Also give its key characteristics.**

Answer

A. TreeSet :

1. TreeSet is a class in Java that implements the SortedSet interface, providing a sorted collection of unique elements.
2. It uses a Red-Black Tree data structure to maintain elements in sorted order.
3. TreeSet provides a convenient and efficient way to work with sorted collections of unique elements in Java.
4. It is commonly used in scenarios where elements need to be stored and accessed in sorted order.

B. Key characteristics :

1. **Sorted order :** TreeSet maintains its elements in sorted order according to their natural ordering or a specified comparator.
2. **Unique elements :** Like other Set implementations, TreeSet does not allow duplicate elements.
3. **Efficient operations :** TreeSet offers efficient operations for adding, removing, and accessing elements.
4. **Iterating in sorted order :** Iterating over a TreeSet provides elements in sorted order.
5. **Use of Red-Black tree :** Internally, TreeSet uses a Red-Black tree data structure to maintain elements in sorted order.

PART-8

Map Interface, HashMap Class, LinkedHashMap Class, TreeMap Class, Hashtable Class.

Que 4.17. **Explain Map interface in Java. Write down the characteristics of Map interface.**

Answer

A. **Map interface :**

1. The Map interface in Java represents a collection of key-value pairs, where each key is associated with a corresponding value.
2. It provides a way to store and retrieve elements based on their keys.
3. Keys are unique within a map, meaning that each key can map to at most one value.
4. The Map interface provides an efficient retrieval and manipulation of values based on their keys.
5. This makes maps suitable for scenarios where quick access to values based on unique identifiers is required.

B. **Key characteristics :**

1. **Key-value pairs :** Maps store elements as key-value pairs, where each key is associated with a corresponding value.
2. **Uniqueness of keys :** Each key in a map is unique.
3. **No ordering :** Maps do not maintain any inherent ordering of their elements.
4. **Efficient retrieval :** Maps provide efficient methods for retrieving values based on their corresponding keys.

Que 4.18. **What is the role of the Map interface in Java Collections framework ?**

Answer

Following are some key roles of the Map interface :

1. **Associative data structure :** Maps provide an associative data structure that allows elements to be stored and retrieved based on unique keys.
2. **Flexible storage :** Maps offer flexibility in storing and organizing data.
3. **Key uniqueness :** Maps enforce the uniqueness of keys within the collection.
4. **Integration with Collections framework :** The Map interface integrates seamlessly with other interfaces and classes in the Java Collections framework.
5. **Multiple implementations :** The Map interface allows multiple implementations with different performance characteristics and usage patterns.

Que 4.19. **Explain HashMap, LinkedHashMap, TreeMap, and Hashtable classes.**

Answer

A. **HashMap class :**

1. **Implementation :** HashMap is a widely-used implementation of the Map interface that stores key-value pairs in a hash table data structure.
2. **Performance :** HashMap provides constant-time average-case performance for basic operations like put and get, assuming a good hash function and a properly-sized backing array. It offers efficient insertion, deletion, and retrieval of elements.
3. **Ordering :** HashMap does not guarantee any specific order of elements. The order of elements may change over time as elements are added or removed from the map.

B. **LinkedHashMap class :**

1. **Implementation :** LinkedHashMap is an implementation of the Map interface that extends HashMap to maintain a doubly linked list alongside the hash table, preserving the insertion order of elements.
2. **Performance :** LinkedHashMap provides similar performance characteristics to HashMap for basic operations like put and get. It offers efficient insertion, deletion, and retrieval of elements, with slightly higher memory overhead due to maintaining the linked list.
3. **Ordering :** LinkedHashMap maintains the insertion order of elements, making it suitable for scenarios where iteration order matters. Elements are iterated over in the same order in which they were inserted into the map.

C. **TreeMap class :**

1. **Implementation :** TreeMap is an implementation of the SortedMap interface that uses a Red-Black Tree data structure to store key-value pairs in sorted order based on the keys.
2. **Performance :** TreeMap provides guaranteed log(n) time complexity for basic operations like put, get, and remove. It offers efficient insertion, deletion, and retrieval of elements, with additional overhead for maintaining the sorted tree structure.
3. **Ordering :** TreeMap maintains the elements in sorted order based on the natural ordering of keys or a specified comparator. This allows for efficient range queries and iteration over elements in sorted order.

D. **Hashtable class :**

1. **Implementation :** Hashtable is a legacy implementation of the Map interface that predates the Java Collections Framework. It is similar to HashMap but is synchronized, making it thread-safe for concurrent access from multiple threads.

2. **Performance :** Hashtable provides similar performance characteristics to HashMap for basic operations like put and get. However, the synchronization overhead may impact performance in multi-threaded environments.
3. **Ordering :** Hashtable does not guarantee any specific order of elements, similar to HashMap.

Que 4.20. Explain the differences between HashMap, LinkedHashMap, TreeMap, and Hashtable classes.

Answer

S.No.	Feature	HashMap	LinkedHash Map	TreeMap	Hashtable
1.	Implementation	Uses hash table	Uses hash table + linked list for ordering	Uses Red-Black Tree	Uses hash table
2.	Ordering	No guaranteed ordering	Maintains insertion order	Sorted order based on keys	No guaranteed ordering
3.	Thread Safety	Not synchronized	Not synchronized	Not synchronized	Synchronized
4.	Null Values/Keys	Allows null values and one null key	Allows null values and one null key	Does not allow null keys	Does not allow null values or keys

PART-9

Sorting.

Que 4.21. How can sorting be performed in Java Collections framework ?

Answer

In the Java Collections Framework, sorting can be performed using the following approaches :

i. **Using Comparable interface :**
   1. Many classes in Java, such as String, Integer, and other wrapper classes, implement the Comparable interface.
   2. This interface defines a method compareTo() that specifies the natural ordering of objects.

   3. To sort a collection of objects that implement Comparable, you can use methods provided by the Collections utility class, such as Collections.sort(List<T> list).
   4. This method sorts the elements of the specified list into ascending order according to their natural ordering.

ii. **Using Comparator interface :**
   1. If the objects you want to sort do not implement Comparable or if you want to sort them based on different ordering criteria, you can use the Comparator interface.
   2. Comparator defines a method compare() that compares two objects.
   3. It returns a negative integer, zero, or a positive integer depending on whether the first object is less than, equal to, or greater than the second object.
   4. You can create a custom Comparator implementation and pass it to methods like Collections.sort() or Arrays.sort() to sort the elements.

iii. **Sorting arrays :**
   1. Java also provides utility methods for sorting arrays.
   2. The Arrays class contains overloaded versions of the sort() method.
   3. It accept arrays of different types, such as sort(int[] a) for sorting arrays of integers and sort(Object[] a) for sorting arrays of objects that implement Comparable.

PART-10

Comparable Interface.

Que 4.22. Explain Comparable interface in Java Collections framework.

Answer

1. The Comparable interface in the Java Collections framework is used to define the natural ordering of objects.
2. It provides a way for objects to specify how they should be compared to one another for the purpose of sorting.
3. Classes that implement Comparable can be sorted into their natural order using utility methods provided by the Collections framework.
4. The Comparable interface is defined in the java.lang package.
5. It consists of a single method called compareTo(), which takes another object of the same type as an argument and returns an integer value indicating the comparison result.

Que 4.23. **Describe the use of the Comparable interface in Java Collections framework.**

Answer

The Comparable interface is used in following ways in Java Collections framework :

1. **Sorting collections :** The primary use of Comparable is to enable sorting of collections of objects.
2. **Natural ordering :** Comparable allows objects to define their natural ordering.
3. **Consistent sorting :** Implementing Comparable allows for consistent sorting behavior.
4. **Ease of use :** Comparable provides a standardized way to define the natural ordering of objects.
5. **Integration with collections framework :** Comparable integrates seamlessly with the Collections framework.

PART-11

Comparator Interface.

Que 4.24. **Explain Comparator interface in Java Collections framework.**

Answer

1. The Comparator interface in the Java Collections Framework is used to define custom ordering of objects.
2. The Comparator interface allows for the definition of multiple comparison strategies for a given class.
3. The Comparator interface is part of the java.util package.
4. It defines a single method called compare(), which compares two objects and returns an integer value indicating the comparison result.
5. The Comparator interface provides a powerful mechanism for defining custom sorting strategies for objects.
6. It allows for flexible and customizable sorting based on different criteria, enhancing the functionality and versatility of the Collections framework.

Que 4.25. **When would you use the Comparator interface instead of the Comparable interface ?**

Answer

You would use the Comparator interface instead of the Comparable interface in the following situations :

1. **Sorting objects with multiple criteria :** When you need to sort objects based on criteria other than their natural ordering.
2. **Sorting objects in different ways :** When you need to sort objects based on different criteria depending on the context or requirements of your application.
3. **Dynamic sorting :** When you need to sort objects based on different criteria at different times, or when you want to switch between different sorting strategies dynamically.
4. **Sorting immutable classes :** When the class representing objects is immutable and cannot be modified to implement Comparable.
5. **Custom sorting order :** When you want to sort objects in a specific order that is different from their natural ordering.

Que 4.26. **Differentiate between Comparable and Comparator interface.**

Answer

S.No.	Feature	Comparable	Comparator
1.	Purpose	Defines natural ordering of objects.	Defines custom ordering of objects.
2.	Interface	Implemented by the class being sorted.	Implemented by a separate class or lambda function.
3.	compareTo() method	Implemented within the class.	Not implemented within the class.
4.	Usage	Objects sorted based on their natural ordering.	Objects sorted based on custom criteria.
5.	Flexibility	Less flexible, fixed ordering.	More flexible, allows for dynamic ordering.

PART-12

Properties Class in Java.

Que 4.27. **What is properties class in Java Collections framework ? Explain the purpose and usage of the Properties class.**

Answer

A. Properties class :

1. The Properties class in the Java Collections Framework is a subclass of Hashtable and represents a persistent set of properties, where each property consists of a key-value pair.
2. It is commonly used for handling configuration settings, such as application settings or parameters, where key-value pairs are stored in a text file.

B. Purpose :

1. The main purpose of the Properties class is to manage configuration data in key-value pairs.
2. It provides a convenient way to store and retrieve configuration settings, such as database connection parameters, application settings, or system properties.

C. Usage :

1. The Properties class is typically used in scenarios where configuration settings need to be managed, such as desktop applications, web applications, and server applications.
2. It provides a simple and efficient way to handle configuration data in a key-value format.

☺☺☺

5 UNIT

Spring Framework and Spring Boot

CONTENTS

PART-1

Spring Core Basics : Spring Dependency Injection Concepts

Que 5.1. What is Spring Core and what are its advantages ?

Answer

1. Spring is a lightweight framework. It can be thought of as a framework of frameworks because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF, etc.
2. The framework can be defined as a structure where we find solution of the various technical problems.
3. The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc.

Advantages of Spring framework :

1. **Predefined templates :** Spring framework provides templates for JDBC, Hibernate, JPA technologies. So there is no need to write too much code. It hides the basic steps of these technologies.
2. **Loose coupling :** The Spring applications are loosely coupled because of dependency injection.
3. **Easy to test :** The dependency injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework does not require server.
4. **Lightweight :** Spring framework is lightweight because of its POJO implementation. The Spring framework does not force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.
5. **Fast development :** The dependency injection feature of Spring framework and it support to various frameworks makes the easy development of JavaEE application.

Que 5.2. What is Spring framework ? Discuss feature of the Spring.

Answer

Spring framework :

1. The Spring framework is an open-source Java framework that provides comprehensive support for building enterprise-level applications.
2. It offers a lightweight and modular approach to develop robust, scalable, and maintainable applications

3. The Spring framework focuses on the concept of dependency injection and inversion of control (IoC).
4. This enables loose coupling and promoting testability and reusability.

Features of Spring framework :

1. **Inversion of Control (IoC) :** The core principle of the Spring framework is IoC, which allows the framework to manage the creation and lifecycle of objects.
2. **Aspect-Oriented Programming (AOP) :** AOP enables modularization of concerns such as logging, transaction management, security, and caching.
3. **Spring container :** It manages the creation, configuration, and lifecycle of objects (beans).
4. **Spring MVC :** Spring MVC is a web framework built on top of the Spring framework, providing a robust and flexible MVC (Model-ViewController) architecture.
5. **Data access and integration :** Spring provides powerful abstractions and integrations for working with various data access technologies.
6. **Security :** Spring security is a highly flexible and customizable security framework.
7. **Testability and test integration :** The Spring framework promotes testability by supporting integration testing and providing mock objects and testing utilities.

Que 5.3. What is dependency injection in Spring ? How does it work ?

Answer

Dependency injection :

1. Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application.
2. Dependency Injection makes our programming code loosely coupled.

Working :

1. Suppose class 1 needs the object of class 2 to instantiate or operate a method, then class 1 is said to be dependent on class 2.
2. Now though it might appear okay to depend a module on the other but, in the real world, this could lead to a lot of problems, including system failure. Hence such dependencies need to be avoided.
3. Spring IOC resolves such dependencies with Dependency Injection, which makes the code easier to test and reuse.

4. Loose coupling between classes can be possible by defining interfaces for common functionality and the injector will instantiate the objects of required implementation.
5. The task of instantiating objects is done by the container according to the configurations specified by the developer.

Que 5.4. What are the types of dependency injection in Spring ?

Answer

There are two types of spring dependency injection. They are :

1. **Setter Dependency Injection (SDI) :** This is the simpler of the two DI methods. In this, the DI will be injected with the help of setter and/or getter methods. Now to set the DI as SDI in the bean, it is done through the bean-configuration file. For this, the property to be set with the SDI is declared under the tag in the bean-config file.
2. **Constructor Dependency Injection (CDI) :** In this, the DI will be injected with the help of constructors. Now to set the DI as CDI in bean, it is done through the bean-configuration file. For this, the property to be set with the CDI is declared under the tag in the bean-config file.

Que 5.5. Compare constructor injection and setter injection in Spring with suitable example.

Answer

S. No.	Aspect	Constructor injection	Setter injection
1.	Definition	Dependencies are provided to a class through its constructor.	Dependencies are provided to a class through setter methods.
2.	Dependency order	The order and number of constructor parameters define the dependencies.	Each dependency can be set independently using corresponding setter methods.
3.	Required dependencies	All dependencies are required, and they must be provided at the time of object instantiation.	Dependencies can be optional, and they can be set later using setter methods.

4.	Immutable objects	Suitable for injecting dependencies into immutable objects.	Not suitable for injecting dependencies into immutable objects.
5.	Initialization	Dependencies are initialized once during object creation and cannot be changed afterward.	Dependencies can be set multiple times, allowing for dynamic changes.

PART-2

Spring Inversion of Control.

Que 5.6. Explain the concept of Inversion of Control (IoC) in Spring.

Answer

1. Spring IoC (Inversion of Control) Container is the core of Spring Framework.
2. It creates the objects, configures and assembles their dependencies, manages their entire life cycle.
3. The Container uses Dependency Injection (DI) to manage the components that make up the application.
4. It gets the information about the objects from a configuration file (XML) or Java Code or Java Annotations and Java POJO class. These objects are called Beans.
5. Since the controlling of Java objects and their lifecycle is not done by the developers, hence the name Inversion Of Control.
6. The followings are some of the main features of Spring IoC :
   i. Creating Object for us,
   ii. Managing our objects,
   iii. Helping our application to be configurable,
   iv. Managing dependencies
7. There are two types of IoC containers. They are :
   i. BeanFactory
   ii. ApplicationContext

Que 5.7. What are the advantages of using Spring for dependency injection and inversion of control ?

Answer

Advantages of using Spring for dependency injection :

1. Dependency Injection (DI) improves modularity, increased testability, enhanced flexibility, and simplified maintenance.
2. DI separates the creation and consumption of dependencies, allowing for code to be organized into smaller units with clear responsibilities and interfaces.
3. It also allows for dependencies to be replaced with mocks or stubs in unit tests without modifying the original code.
4. DI decouples code from specific implementations, making it easy to swap or update dependencies without affecting the rest of the system.
5. DI reduces the complexity and interdependence of your code, making it easier to understand, debug, and refactor.

Advantages of using Spring for inversion of control :

1. Inversion of control maintains the creation, configuration, provisioning and lifecycle of all container-managed objects separately from the code where they are referenced.
2. IOC decouples the execution of a task from its implementation.
3. IOC makes it easier to switch between different implementations.
4. IOC gives greater modularity of a program.
5. IOC makes greater ease in testing a program by isolating a component or mocking its dependencies, and allowing components to communicate through contracts.
6. IOC makes your code loosely coupled.
7. IOC makes it easy for the programmer to write great unit tests.

PART-3

AOP.

Que 5.8. How does Aspect-Oriented Programming (AOP) work in Spring ? Provide examples.

OR

Illustrate Aspect Oriented Programming in spring framework.

Answer

Aspect Oriented Programming (AOP) :

1. Aspect Oriented Programming (AOP) is defined as the breaking of code into different modules where the aspect is the key unit of modularity.

2. AOP addresses cross-cutting concerns in an application.
3. Cross-cutting concerns are functionalities that cut across multiple modules or components, such as logging, caching, authentication, and transaction management.
4. AOP provides a modular approach to separate and manage these concerns.
5. This helps in promoting code modularity, reusability, and maintainability.
6. In the Spring framework, AOP is seamlessly integrated and provides robust support for aspect-oriented programming.
7. AOP in Spring is primarily based on proxy-based AOP and is implemented using runtime proxies.

Key concepts in AOP :

1. **Aspect :** An aspect is a modular unit of cross-cutting functionality that encapsulates a concern, such as logging or security.
2. **Join point :** A join point represents a specific point in the execution of a program, typically corresponding to method invocations or exception handling.
3. **Pointcut :** A pointcut is a predicate that selects join points, defining the specific points in the code where advice should be applied.
4. **Advice :** Advice is the code that runs at a particular join point. It represents the behavior associated with a cross-cutting concern.
5. **Aspect configuration :** Aspect configuration defines how the aspect is applied to the target objects and specifies the pointcuts and advice to be executed.

PART-4

Bean Scopes : Singleton, Prototype, Request, Session, Application.

Que 5.9. What are the different types of bean scopes in Spring ? Explain each one.

Answer

Following are the different types of scopes in Spring :

1. **Singleton :** This is the default scope in Spring. It means that only one instance of a bean is created per Spring container. Singleton beans are shared across the application. Any changes made to the bean's state are visible to all components that use it.

2. **Prototype :** Prototype scope creates a new instance of a bean every time it is requested. This means that multiple instances of the bean can coexist within the application. Prototype beans are not shared and have independent states.
3. **Request :** Request scope is specific to web applications. It creates a new instance of a bean for each HTTP request. The bean is available only within the scope of that request and is destroyed once the request is completed.
4. **Session :** Session scope is also applicable to web applications. It creates a new instance of a bean for each user session. The bean is destroyed when the session ends.
5. **Global Session :** This scope is similar to the Session scope but applies to applications that use a global session. It creates a single instance of a bean per global session and is available across multiple HTTP sessions.
6. **Custom Scopes :** Spring also allows developers to define their own custom scopes. This gives flexibility to create scopes tailored to specific requirements.

Que 5.10. What is the Singleton scope in Spring ? How is it different from other scopes ?

Answer

1. If a scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition.
2. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.
3. The default scope is always singleton.
4. However, when we need one and only one instance of a bean, we can set the scope property to singleton in the bean configuration file

Difference :

1. **Singleton :** Only one instance of the spring bean will be created for the spring container. This is the default spring bean scope. While using this scope, make sure bean does not have shared instance variables otherwise it might lead to data inconsistency issues.
2. **Prototype :** A new instance will be created every time the bean is requested from the spring container.
3. **Request :** This is same as prototype scope, however it's meant to be used for web applications. A new instance of the bean will be created for each HTTP request.
4. **Session :** A new bean will be created for each HTTP session by the container.

5. **Global-session :** This is used to create global session beans for Portlet applications.

Que 5.11. What is the Prototype scope in Spring ? When would you use it ?

Answer

1. If the scope is declared prototype, then spring IOC container will create a new instance of that bean every time a request is made for that specific bean.
2. A request can be made to the bean instance either programmatically using getBean() method or by XML for Dependency Injection of secondary type.
3. Generally, we use the prototype scope for all beans that are stateful, while the singleton scope is used for the stateless beans.

Que 5.12. Explain the Request scope in Spring. How is it related to web applications ?

Answer

A. Request scope in Spring :

1. In Spring, the request scope is one of the bean scopes available for managing object instances.
2. The request scope defines that a new instance of a bean is created for each individual HTTP request made to a web application.
3. The bean remains active throughout the processing of that specific request and is then destroyed once the request is completed.

B. Request scope relation to web applications :

1. The request scope is primarily relevant in the context of web applications, where each user request is processed independently.
2. When a bean is scoped as "request" in Spring, it ensures that each user request receives a separate instance of that bean, isolating the state and behavior specific to that request.
3. The request scope is commonly used for beans that hold request specific data or perform operations related to a specific HTTP request.
4. For example, in a web application, you may have a controller that handles user requests.
5. Within that controller, you might have a request-scoped bean responsible for validating and processing the incoming request data.
6. By using the request scope, you ensure that each user request is handled by a separate instance of that bean, preventing interference between concurrent requests.

Que 5.13. What is the Session scope in Spring ? How is it different from other scopes ?

Answer

A. Session scope in Spring :

1. In Spring, the session scope is one of the bean scopes available for managing object instances.
2. The session scope defines that a single instance of a bean is created and associated with an HTTP session.
3. The bean remains active throughout the lifespan of that specific session and is destroyed when the session ends or expires.
4. The session scope is particularly relevant in web applications, where user sessions are maintained.
5. By using the session scope, you can create beans that store and manage session-specific data or perform operations related to a particular user session.

B. Difference : The Session scope in Spring is different from other scopes, such as Singleton, Prototype, and Request scopes, in the following ways :

1. **Scope duration :** The session scope lasts longer than the request scope but shorter than the singleton scope.
2. **Instance association :** In the session scope, each user session will have its own unique instance of a bean. In contrast, singleton scope has a single shared instance across the entire application, while prototype scope creates a new instance per request or dependency injection.
3. **Scoped proxy :** To manage the session scope effectively, Spring uses a scoped proxy mechanism. In contrast, other scopes like singleton and Prototype do not require scoped proxies.
4. **Web application context :** The session scope is specific to web applications and is relevant when using Spring's web-related components. Other scopes like singleton and prototype are applicable in various types of Spring applications, not limited to web environments.

Que 5.14. How does the Application scope (Singleton scope) work in Spring ? In what scenarios would you use it ?

Answer

A. Working of application scope in Spring :

1. **Single instance :** In the application scope, only one instance of a bean is created and shared throughout the application. Any subsequent requests for that bean will receive the same instance.

2. **Shared state :** Since the same instance is shared across the application, any changes made to the state of the bean will be reflected across all components that use that bean. This can be advantageous when you want to maintain a shared state or share data among different parts of the application.
3. **Thread safety :** When using the application scope, it's important to consider thread safety. If multiple threads access and modify the shared bean instance concurrently, you need to ensure proper synchronization or use thread-safe techniques to prevent race conditions and data inconsistencies.

B. **Scenarios :** Here are some common scenarios where you would use the application scope :

1. **Configuration beans :** Beans that provide application-wide configuration settings or manage shared resources can be scoped as application beans. For example, database connection pools, configuration managers, or caching components can be scoped as singletons to ensure consistent behavior and resource sharing.
2. **Stateful beans :** Beans that maintain mutable state or manage application-wide data can be scoped as singletons. This allows different components to access and modify the shared state, promoting data consistency and efficient resource utilization.
3. **Performance optimization :** In some cases, creating multiple instances of a particular bean might be expensive in terms of memory or resource usage. By using the application scope, you can ensure that a single instance is reused, reducing overhead and improving performance.

PART-5

Web Socket.

Que 5.15. What is WebSocket in Spring ? How does it facilitate real-time communication ?

Answer

A. **WebSocket in Spring :**

1. WebSocket is a communication protocol that provides full-duplex communication channels over a single TCP connection.
2. It enables real-time and bidirectional communication between a client and a server.
3. In the context of Spring, WebSocket support allows you to build applications that facilitate real-time communication between clients and servers.

B. **Real-time communication :** Here's how WebSocket in Spring facilitates real-time communication :

1. **Connection establishment :** With WebSocket in Spring, client sends a WebSocket handshake request. The server, upon receiving the request, upgrades the connection to the WebSocket protocol, enabling bi-directional communication between the client and server.
2. **Message exchange :** Once the WebSocket connection is established, the client and server can exchange messages in real time.
3. **Event-driven model :** WebSocket in Spring follows an event driven model, where the server and client can react to events asynchronously.
4. **Broadcasting and Pub/Sub :** WebSocket in Spring supports broadcasting messages to multiple clients or implementing pub/sub communication patterns. This enables real-time updates and notifications to be efficiently distributed among multiple clients.
5. **Integration with Spring components :** WebSocket in Spring can be seamlessly integrated with other Spring components. This allows you to apply security measures, authenticate users, and leverage existing Spring infrastructure.

Que 5.16. How does Spring support WebSocket communication ? Explain the relevant components.

Answer

Spring provides support for WebSocket communication through the Spring WebSocket module. This module integrates WebSocket functionality into the Spring framework, making it easy to develop WebSocket-enabled applications.

Relevant components : Following are the relevant components provided by Spring for WebSocket communication :

1. **WebSocketHandler :** The WebSocketHandler interface is implemented by a Spring bean to handle WebSocket messages and events. It defines methods such as handleMessage() and handleTransportError() that allow you to process incoming WebSocket messages and handle WebSocket-related errors.
2. **WebSocketSession :** The WebSocketSession object represents a connection between a client and the server. It provides methods to send messages, close the session, retrieve attributes, and get information about the session.
3. **WebSocketConfigurer :** The WebSocketConfigurer interface is used to configure WebSocket handling in Spring. By implementing this interface, you can register WebSocketHandler instances and set up WebSocket-related settings.

4. **WebSocketMessageBrokerConfigurer :** This interface is used for configuring WebSocket message broker functionality, which provides support for pub/sub communication patterns and broadcasting messages.
5. **SimpMessagingTemplate :** This class is a convenient way to send messages to WebSocket clients. It simplifies the process of sending messages to specific topics or clients by abstracting the details of message routing and subscription management.
6. **STOMP :** STOMP (Simple Text Oriented Messaging Protocol) protocol provides a higher-level abstraction for working with WebSocket communication. Spring provides annotations such as '@MessageMapping' and '@SendTo' that can be used with STOMP for mapping message handling methods.

PART-6

Auto Wiring.

Que 5.17. What is Auto-wiring in Spring ? How does it simplify bean configuration ?

Answer

A. Auto-wiring in Spring :

1. Auto-wiring in Spring is a feature that automatically resolves dependencies between beans without the need for explicit configuration.
2. It simplifies bean configuration by automatically connecting beans based on their types, eliminating the need for manual wiring.

B. Simplification of bean configuration : Here's how Auto-wiring simplifies the process :

1. **Dependency resolution :** With auto-wiring, you do not need to explicitly define and wire dependencies in XML configuration files. Instead, the Spring IoC (Inversion of Control) container analyzes the dependencies of a bean.
2. **Automatic injection :** Once the dependencies are identified, the Spring container automatically injects the corresponding beans into the dependent bean. You do not have to write explicit code to wire the dependencies.
3. **Reduction in configuration code :** Auto-wiring eliminates the need for manual wiring and reduces the amount of configuration code.
4. **Convention over configuration :** Auto-wiring follows the convention-over-configuration principle. This allows the container to automatically wire the beans without requiring explicit configuration, which reduces the need for manual wiring instructions.
5. **Flexibility and customization :** While auto-wiring simplifies the configuration process, Spring also provides ways to customize and control the auto-wiring behavior. You can use annotations like '@Autowired', '@Qualifier', and '@Primary' to specify dependencies and resolve ambiguities.

PART-7

Annotations.

Que 5.18. Explain the concept of annotations in Spring. How are they used in bean configuration ?

OR

Classify all the annotations those are exclusively used for spring boot applications.

Answer

Spring boot annotations :

1. Spring boot annotation is a form of metadata that provides data about a program.
2. Annotations are used to provide supplemental information about a program. It is not a part of the application that we develop.
3. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program.

Core spring framework annotations :

1. **@Required :** It applies to the bean setter method. It indicates that the annotated bean must be populated at configuration time with the required property; else it throws an exception BeanInitializationException.
2. **@Autowired :** Spring provides annotation-based auto-wiring by providing @Autowired annotation. It is used to autowire spring bean on setter methods, instance variable, and constructor. When we use @Autowired annotation, the spring container auto-wires the bean by matching data-type.
3. **@Configuration :** It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.
4. **@ComponentScan :** It is used when we want to scan a package for beans. It is used with the annotation @Configuration. We can also specify the base packages to scan for Spring Components.

5. **@Bean :** It is a method-level annotation. It is an alternative of XML tag. It tells the method to produce a bean to be managed by spring container.

PART-8

Life Cycle Call Backs.

Que 5.19. Can we call a Java class with annotations as a POJO class or not ?

Answer

1. Technically, you can call a Java class with annotations a POJO (Plain Old Java Object) class, but it depends on the purpose and usage of the annotations.
2. The term "POJO" generally refers to a simple Java class that encapsulates data and provides getter and setter methods for accessing and modifying that data.
3. It is a lightweight class that does not depend on any external frameworks or libraries.
4. Annotations, on the other hand, are a way to add metadata or additional information to Java code elements, such as classes, methods, or fields.
5. Annotations can be used for a variety of purposes, including configuration, documentation, or runtime behavior modification.
6. If the annotations used in a Java class are purely for metadata or configuration purposes, then you can still consider the class as a POJO.
7. However, if the annotations used in the class introduce specific behavior or dependencies tied to a particular framework or library, then the class may be considered more than just a simple POJO.
8. It may have additional responsibilities or dependencies associated with those annotations.

Que 5.20. What is Bean in Spring ? How bean life cycle can be controlled in Spring ?

Answer

Bean :

1. In Spring, a bean is an object that is managed by the Spring IoC (Inversion of Control) container.
2. It is a core concept in the Spring framework and represents the building blocks of an application.

3. Beans in Spring are instantiated, assembled, and managed by the container.
4. Beans provide various benefits such as dependency injection, aspect oriented programming, and modularity.

Controlling bean life cycle :

1. The lifecycle of a bean in Spring consists of several phases, including bean instantiation, initialization, usage, and destruction.
2. Spring provides mechanisms to control and customize the lifecycle of beans.

Here are the main ways to control the bean lifecycle in Spring :

A. **Bean instantiation :**
   1. **Constructor injection :** Beans can be instantiated using constructors and dependencies can be injected via constructor parameters.
   2. **Setter injection :** Beans can be instantiated using a default constructor and dependencies can be injected using setter methods.

B. **Initialization :**
   1. **InitializingBean and @PostConstruct :** Beans can implement the InitializingBean interface or use the @PostConstruct annotation to define initialization logic.
   2. **XML configuration :** XML configuration allows specifying initialization methods using the attribute.

C. **Usage :**
   1. **Bean methods :** Beans can define methods that perform specific tasks or provide functionality. These methods can be called after the bean is initialized.

D. **Destruction :**
   1. **DisposableBean and @PreDestroy :** Beans can implement the DisposableBean interface or use the @PreDestroy annotation to define cleanup logic.
   2. **XML configuration :** XML configuration allows specifying destruction methods using the attribute.

Que 5.21. What are the different types of life cycle callbacks in Spring ? Explain each one.

Answer

Spring bean factory controls the creation and destruction of beans. To execute some custom code, the bean factory provides the callback methods, which can be categorized as follow :

1. **Initialization callbacks :**
   i. **@PostConstruct Annotation :** This annotation is placed on a method that should be executed after a bean has been instantiated and its dependencies have been injected. It is commonly used to perform initialization tasks.
   ii. **InitializingBean Interface :** Beans implementing the InitializingBean interface need to implement the afterPropertiesSet() method. This method is called after the bean's properties have been set by the container.
2. **Destruction callbacks :**
   i. **@PreDestroy Annotation :** This annotation is placed on a method that should be executed just before a bean is destroyed. It's commonly used to release resources or perform cleanup tasks.
   ii. **DisposableBean Interface :** Beans implementing the DisposableBean interface need to implement the destroy() method. This method is called when the container is shutting down and the bean is being removed from the container.
3. **Custom initialization and destruction methods :**
   i. You can define custom initialization and destruction methods in your bean configuration. These methods can have any name and signature.
   ii. Use the init-method and destroy-method attributes in XML configuration or the corresponding annotations (@Bean(initMethod = "...") and @Bean(destroyMethod = "...")) to specify the methods to be called.

PART-9

Bean Configuration Styles.

Que 5.22. What are the different bean configuration styles in Spring ? Provide examples.

Answer

Following are several ways to configure beans in Spring :

1. **XML-based configuration :** XML-based configuration is the traditional approach to configure beans in Spring using XML files. You define beans, their dependencies, and configuration details in XML format.

   Example :

```
<!-- Bean definition in XML -->
<bean id ="myBean" class="com.example.MyBean">
```



```
<property name="name" value = "John"/>
</bean>
```

2. **Java-based configuration :** Java-based configuration, also known as JavaConfig, allows you to configure beans using Java classes. You use annotations and Java code to define beans, dependencies, and configuration.

   Example :

```
// Java-based configuration
@Configuration
public class AppConfig{
@Bean
public MyBean myBean()
{ MyBean bean = new MyBean();
bean.setName("John");
return bean;
}
}
```

3. **Annotation-based configuration :** Annotation-based configuration leverages annotations to define beans and their dependencies directly in classes. Spring scans these annotations and automatically configures the beans.

   Example :

```
// Annotation-based configuration
@Component
public class MyBean
{@Value("John")
private String name; //
Bean methods, lifecycle hooks, etc.
}
```

4. **Java configuration with component scanning :** Java configuration with component scanning combines Java-based configuration with automatic component scanning. It allows you to define beans using annotations, and Spring automatically detects and registers those beans.

   Example :

```
// Java Configuration with Component Scanning
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {
```

```
// No explicit bean definition needed
}
```

5. **XML configuration with component scanning :** XML Configuration with Component Scanning combines XML-based configuration with component scanning. You can define beans using annotations in classes, and Spring scans and registers them based on XML configuration.

Example :

```
<!-- XML Configuration with Component Scanning -->
<context:component-scan base-package="com.example"/>
```

Que 5.23. How does Spring handle bean initialization and destruction during its life cycle ?

Answer

i. In Spring, beans go through a life cycle that includes initialization and destruction phases.

ii. Spring provides mechanisms to handle bean initialization and destruction gracefully.

Here's how Spring handles bean life cycle :

1. **Instantiation :** When the Spring container starts, it creates bean instances based on the bean definitions defined in the configuration.
2. **Dependency injection :** After instantiation, Spring performs dependency injection to populate the dependencies of the bean.
3. **Bean post processors :** Spring allows you to register BeanPostProcessor implementations that can intercept bean initialization callbacks. BeanPostProcessors can modify the bean instance before and after initialization.
4. **Initialization callback :** If a bean defines an initialization method with the '@PostConstruct' annotation, Spring invokes this method after all dependencies are injected and before the bean is ready for use. This allows for custom initialization logic to be executed.
5. **Usage of @EventListener :** Spring provides the @EventListener annotation, which allows beans to listen to events and perform specific actions during the application's life cycle.
6. **Destruction callback :** If a bean defines a destruction method with the '@PreDestroy' annotation, Spring invokes this method before the bean is removed from the container. This allows for custom cleanup or resource release operations.

7. **Shutdown hook :** When the Spring application context is closed, Spring triggers the destruction phase for all beans. It ensures that the destruction callbacks are invoked to release resources and perform cleanup operations.

PART-10

Spring Boot Build Systems.

Que 5.24. How dependency management helps in Spring Boot build system.

Answer

1. In Spring Boot, choosing a build system is an important task. So, Maven or Gradle are good build system as they provide a good support for dependency management.
2. Spring Boot team provides a list of dependencies to support the Spring Boot version for its every release.
3. You do not need to provide a version for dependencies in the build configuration file. Spring Boot automatically configures the dependencies version based on the release.
4. When you upgrade the Spring Boot version, dependencies will upgrade automatically

Maven dependency : For Maven configuration, we should inherit the Spring Boot Starter parent project to manage the Spring Boot Starters dependencies. For this, simply we can inherit the starter parent in our pom.xml file as shown below.

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>1.5.8.RELEASE</version>
</parent>
```

We should specify the version number for Spring Boot Parent Starter dependency. Then for other starter dependencies, we do not need to specify the Spring Boot version number. Observe the code given below :

```
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
</dependencies>
```

Gradle dependency : We can import the Spring Boot Starters dependencies directly into build.gradle file. We do not need Spring Boot start Parent dependency like Maven or Gradle. Observe the code given below :

```
Buildscript {
ext {
springBootVersion = '1.5.8.RELEASE'
}
repositories {
mavenCentral()
}
dependencies {
classpath("org.springframework.boot:spring-boot-
gradleplugin:${springBootVersion}")
}
}
```

Similarly, in Gradle, we need not specify the Spring Boot version number for dependencies. Spring Boot automatically configures the dependency based on the version.

```
dependencies {
compile('org.springframework.boot:spring-boot-starter-web')
}
```

PART-11

Spring Boot Code Structure.

Que 5.25. How is the code structured in a Spring Boot application ?

Answer

In a Spring Boot application, the code is typically structured in following certain conventions and best practices. The recommended structure organizes the code in a modular and maintainable way. Following is a typical structure for a Spring Boot application :

1. **Main application class :** The application starts with a main class annotated with @SpringBootApplication. This class serves as the entry point of the application and initializes the Spring Boot environment.
2. **Controller classes :** These classes are responsible for handling incoming requests and returning appropriate responses. They are typically annotated with @RestController or @Controller.

3. **Service classes :** Service classes contain the business logic of the application. They handle complex operations and are often annotated with @Service.
4. **Repository/DAO classes :** These classes interact with the database or other data sources. They are responsible for data retrieval, storage, and manipulation. They are typically annotated with @Repository or @Component.
5. **Model/Entity classes :** Model or entity classes represent the data structure of the application. They define the structure and relationships of the objects used in the application.
6. **Configuration classes :** Configuration classes provide additional configuration and customization options for the application. They can include beans, database configuration, security configuration, etc. These classes are often annotated with @Configuration.
7. **Utility classes :** Utility classes contain helper methods and utility functions that can be used across different parts of the application. They provide common functionality and reusable code snippets.
8. **Resource files :** Resource files include static resources such as HTML, CSS, JavaScript, and other files required for the application. They are typically stored in the src/main/resources directory.
9. **Test classes :** Test classes contain unit tests, integration tests, and other tests to ensure the correctness of the application. They are stored in a separate directory, usually src/test/java.

PART-12

Spring Boot Runners.

Que 5.26. Describe application runner and command line runner.

Answer

1. Spring Boot provides two runner interfaces, ApplicationRunner and CommandLineRunner. Being Functional Interfaces, both the runners have a single functional method, run().
2. When we implement one of these runners, Spring Boot invokes its run() method after it starts the context and before the application starts.
3. That means we can use Spring Boot CommandLineRunner or ApplicationRunner to execute a piece of code when launching an Application or to create a Spring Boot non-web application.
4. Overall, both CommandLineRunner and ApplicationRunner are similar, and we can use them to do exact same things.

5. The only difference between the two interfaces is the signature of the run() method. The run() method in the CommandLineRunner receives the application or program argument as an array of String.
6. However, the run() method in ApplicationRunner receives the program arguments wrapped in an ApplicationArguments instance. The ApplicationArguments provides convenient methods to access the program arguments.

PART-13

Loggers.

Que 5.27. Explain Spring Boot logging.

Answer

1. Spring Boot logging is defined as a framework that enables developers to trace out errors that might occur in the running of the application.
2. Logging in Spring Boot is basically an API that provides tracing out of information along with a recording of any critical failure that might occur in the application during its run.
3. Spring Boot uses a common logging framework to implement all internal logging and leaves the log implementation open.
4. A thin adapter that allows configurable bridging to other logging systems is the USP of Commons logging.
5. The default configurations provided by Spring Boot logging are Java Util logging, Log4J2, Logback. The loggers in default configurations are pre-configured for using console output.
6. A default Spring Boot log file contains the following items :
   i. **Date and time :** This provides the occurrence date and time of the log item.
   ii. **Log level :** This provides the information on what level of information the log is of. It will be one out of the 7 options we will see now, *i.e.*, TRACE, DEBUG, INFO, WARN, ERROR, FATAL or OFF.
   iii. **Process ID :** This provides the information of the process ID on which the spring boot application is running on.
   iv. **Separator :** This is a separator which signifies the next part of the log.
   v. **Thread name :** This is enclosed within a square ([ ]) brackets and mostly contains the thread within which the logging thread or element is present.

   vi. **Logger name :** This is the penultimate element that contains the source class name.
   vii. **Log message :** Finally, this element contains the log message, which explains the methods followed in the application and helps us tracing back to the root cause if an error pops up in the application.

PART-14

Building Restful Web Services.

Que 5.28. What do you understand by Restful web services in Spring Boot ? What are its advantages ?

Answer

1. REST stands for REpresentational State Transfer. The main goal of RESTful web services is to make web services more effective.
2. RESTful web services try to define services using the different concepts that are already present in HTTP.
3. REST is an architectural approach, not a protocol. It does not define the standard message exchange format.
4. We can build REST services with both XML and JSON. JSON is more popular format with REST.
5. The key abstraction is a resource in REST. A resource can be anything. It can be accessed through a Uniform Resource Identifier (URI).
6. **For example :** The resource has representations like XML, HTML, and JSON. The current state capture by representational resource. When we request a resource, we provide the representation of the resource.

   The important methods of HTTP are :
   i. **GET :** It reads a resource.
   ii. **PUT :** It updates an existing resource.
   iii. **POST :** It creates a new resource.
   iv. **DELETE :** It deletes the resource

Advantages of RESTful web services :

1. RESTful web services are platform-independent.
2. It can be written in any programming language and can be executed on any platform.
3. It provides different data format like JSON, text, HTML, and XML.
4. It is fast in comparison to SOAP because there is no strict specification like SOAP.

5. These are reusable.
6. They are language neutral.

Que 5.29. How to build Restful web services in Spring Boot ?

Answer

For building a RESTful Web Services, we need to add the Spring Boot Starter Web dependency into the build configuration file.

For Maven user, use the following code to add the below dependency in pom.xml file :

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

For Gradle user, use the following code to add the below dependency in build.gradle file.

```
compile('org.springframework.boot:spring-boot-starter-web')
```

Que 5.30. Create a RESTful spring boot application for handling the delete and put request.

Answer

1. Following is an example of a simple RESTful Spring Boot application that handles DELETE and PUT requests.
2. In this example, we'll create an API for managing a collection of books.
3. Set up a new spring boot project with the necessary dependencies.
4. Create a book class representing a book entity.
5. The class can have properties like id, title, author, etc., along with getters and setters.

```
public class Book {
private Long id;
private String title;
private String author;
// Constructors, getters, and setters }
```

6. Create a BookController class to handle the RESTful endpoints. It will contain methods for handling DELETE and PUT requests.

```
@RestController
@RequestMapping("/api/books")
public class BookController {
```



```
private List <Book> books = new ArrayList<>();
// GET request to retrieve all books
@GetMapping
public List <Book> getAllBooks() {
return books;
}
// PUT request to update a book
@PutMapping("/{id}")
public ResponseEntity <string> updateBook(@PathVariable Long id,
@RequestBody Book updatedBook) {
// Find the book with the given ID
Optional <Book> optionalBook = books.stream()
.filter(book -> book.getId().equals(id))
.findFirst();
if(optionalBook.isPresent()) {
Book book = optionalBook.get();
// Update the book properties
book.setTitle(updatedBook.getTitle());
book.setAuthor(updatedBook.getAuthor());
return ResponseEntity.ok("Book updated successfully.");
} else {
return ResponseEntity.notFound().build();
}
}
// DELETE request to delete a book
@DeleteMapping("/{id}")
public ResponseEntity <string> deleteBook(@PathVariable Long id) {
// Find the book with the given ID
Optional <Book> optionalBook = books.stream()
.filter(book -> book.getId().equals(id))
.findFirst();
if(optionalBook.isPresent()) {
Book book = optionalBook.get();
books.remove(book);
return ResponseEntity.ok("Book deleted successfully.");
} else {
return ResponseEntity.notFound().build();
```

```
}
}
}
```

7. Configure the main application class.

```
@SpringBootApplication
public class RestfulApplication {
public static void main(String[] args) {
SpringApplication.run(RestfulApplication.class, args);
}
}
```

8. Run the application

PART-15

Rest Controller.

Que 5.31. What is rest controller ?

Answer

1. RestController is used for making restful web services with the help of the @RestController annotation.
2. This annotation is used at the class level and allows the class to handle the requests made by the client.
3. The RestController allows to handle all REST APIs such as GET, POST, Delete, PUT requests.
4. It serves JSON, XML and custom response.
5. Its syntax is as follows :

```
@RestController
public class ProductServiceController {
}
```

PART-16

Request Mapping, Request Body, Path Variable, Request Parameter.

Que 5.32. What is request mapping ?

Answer

Request mapping :

1. To configure the mapping of web requests, we use the @RequestMapping annotation.
2. The @RequestMapping annotation can be applied to class-level and/or method-level in a controller
3. The class-level annotation maps a specific request path or pattern onto a controller.
4. We can then apply additional method-level annotations to make mappings more specific to handler methods.
5. The @RequestMapping annotation is used to define the Request URI to access the REST Endpoints.
6. We can define Request method to consume and produce object.
7. The default request method is GET. @RequestMapping(value = "/products") public ResponseEntity.

Que 5.33. Define the terms :

1. **Request body**
2. **Path variable**
3. **Request parameter**

Answer

1. **Request body :**
   The @RequestBody annotation is used to define the request body content type.

```
public ResponseEntity<Object> createProduct(@RequestBody Product product) {
}
```

2. **Path variable :**
   The @PathVariable annotation is used to define the custom or dynamic request URI. The Path variable in request URI is defined as curly braces {} as given below :

```
public ResponseEntity<Object> updateProduct(@PathVariable("id") String id) {
}
```

3. **Request parameter :**
   The @RequestParam annotation is used to read the request parameters from the Request URL. By default, it is a required parameter. We can also set default value for request parameters as :

```
public ResponseEntity<Object> getProduct(
@RequestParam(value = "name", required = false, defaultValue = "honey") String name) {
}
```

PART-17

GET, POST, PUT, DELETE APIs.

Que 5.34. **Explain GET, POST, PUT, DELETE APIs in context to Spring boot.**

OR

Explain GET, PUT, POST, and DELETE method with respect to REST API.

Answer

GET API : The default HTTP request method is GET. This method does not require any Request Body. You can send request parameters and path variables to define the custom or dynamic URL.

POST API : The HTTP POST request is used to create a resource. This method contains the Request Body. We can send request parameters and path variables to define the custom or dynamic URL.

PUT API : The HTTP PUT request is used to update the existing resource. This method contains a Request Body. We can send request parameters and path variables to define the custom or dynamic URL.

DELETE API : The HTTP Delete request is used to delete the existing resource. This method does not contain any Request Body. We can send request parameters and path variables to define the custom or dynamic URL

PART-18

Build Web Applications.

Que 5.35. **How can you use Spring Boot to build web applications ?**

Answer

Following is a general guide on how to use Spring Boot to build web applications :

1. **Setup project :** Start by creating a new Spring Boot project using either Spring Initializer or your preferred integrated development environment (IDE).
2. **Dependencies :** In your project's build configuration include the necessary dependencies for building web applications. Common dependencies include spring-boot-starter-web for basic web functionality, and additional dependencies for templating engines and database access.

3. **Create controllers :** Create classes annotated with @Controller or @RestController to handle incoming HTTP requests. Use @RequestMapping and related annotations to define mappings between URLs and controller methods.
4. **View templates :** If your application requires rendering dynamic content, include the corresponding Spring Boot starter dependency, create template files in the designated directory, and use the template engine's syntax to generate dynamic content.
5. **Static resources :** Place static resources such as CSS, JavaScript, and images in the src/main/resources/static directory. Spring Boot will serve these resources directly without requiring explicit controller mappings.
6. **Configuration :** Customize application behavior through configuration files like application.properties or application.yml.
7. **Testing :** Write unit tests and integration tests for your controller methods using testing frameworks like JUnit and Spring Test.
8. **Run the application :** Run your Spring Boot application using your IDE or the command line.
9. **Deployment :** Build an executable JAR or WAR file using your build tool. You can deploy this file to various environments, including traditional servers or cloud platforms.

Que 5.36. **Explain the role of templates and view resolvers in Spring Boot web applications.**

Answer

Role of templates in Spring Boot web applications :

1. Templates are files that contain a mix of static content and placeholders for dynamic data.
2. They are used to generate dynamic HTML content that is sent to the client's web browser.
3. Templates allow developers to create reusable and consistent layouts for displaying data.
4. In Spring Boot web applications, template engines process these templates, replacing placeholders with actual data values.
5. Common template engines used in Spring Boot applications include Thymeleaf, FreeMarker, and JSP (Java Server Pages).
6. Each of these template engines has its syntax and features, but they all serve the same purpose of dynamically generating HTML content.

Role of view resolvers in Spring Boot web applications :

1. A view resolver is responsible for determining which template to use for rendering a specific view (web page) and then invoking the template engine to process the template and generate the final HTML output.

2. It essentially resolves the logical view name to the actual template file.
3. In a Spring Boot application, you configure a view resolver to handle the mapping between logical view names and the corresponding template files
4. This enables you to work with logical view names in your controller methods rather than specifying the exact template paths.
5. Spring Boot provides default view resolver configurations, making it easier to get started.

☺☺☺

1 UNIT

Introduction (2 Marks Questions)

1.1. What is Java ?

Ans Java is a versatile, high-level, object-oriented programming language made for developers to "write once, run anywhere" Java codes.

1.2. What are some of the key characteristics and features of Java ?

Ans Key characteristics and features of Java are :

1. Platform independence
2. Object-oriented
3. Robust and secure
4. Rich standard library
5. Multithreading

1.3. How does Java achieve platform independence ?

Ans Java achieves platform independence through the use of Java Virtual Machine (JVM).

1.4. What is Java virtual machine (JVM) ?

Ans JVM is an abstract machine. It is a software-based, virtual computer that provides runtime environment in which java bytecode can be executed.

1.5. What is a Java Development Kit (JDK) ?

Ans JDK is a cross-platform software development environment that offers a collection of tools and libraries necessary for developing Java-based software applications and applets.

1.6. What is Java Runtime Environment (JRE) ?

Ans Java Run-time Environment (JRE) is the part of the Java Development Kit (JDK). It is a software package that provides the runtime environment necessary for executing Java applications.

1.7. What is Java development environment ?

Ans A Java development environment, also known as Integrated Development Environment (IDE), is a software suite that provides tools and features to facilitate the development, testing, and debugging of Java applications.

1.8. Name some of the popular Java development environments.

Ans Some of the popular Java development environments include :

1. Eclipse,
2. IntelliJ IDEA,
3. NetBeans, and
4. Oracle JDeveloper.

1.9. What is Java source file ?

Ans A Java source file, also known as a Java source code file, contains the code written in the Java programming language.

1.10. What do you understand by 'object' in Java ?

Ans In Java, an object is a fundamental runtime entity that represents an instance of a class.

1.11. Define class in Java.

Ans In Java, a class is a fundamental blueprint or template for creating objects. It serves as a model for defining the attributes (data) and behaviors (methods) that objects of the class will exhibit.

1.12. Define constructor.

Ans A constructor is a special method in a class that is automatically called when an object of that class is created.

1.13. What are various types of constructor available in Java ?

Ans Following are various types of constructors in Java :

1. Default constructor
2. Parameterized constructor
3. Private constructor
4. Copy constructor

1.14. What is 'method' in Java ?

Ans A method is a block of code which only runs when it is called. Methods are used to perform certain actions. A method must be declared within a class.

1.15. What is an access specifier in Java ?

Ans In Java, access specifiers (modifiers) are keywords used to control the visibility and accessibility of classes, methods, variables, and other members within a Java class.

1.16. List types of access specifier in Java.

Ans There are four main access specifiers in Java :

1. public
2. private
3. protected
4. default (no access specifier)

1.17. What are static members in Java ?

Ans Variables and methods declared using keyword static are called static members of a class. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.

1.18. What is 'final' member in Java ?

Ans In Java, a 'final' member refers to a variable, method, or class that cannot be modified or overridden once it has been defined.

1.19. What do you understand by comments in Java ?

Ans Comments in Java are non-executable text annotations that are used to provide explanations, documentation, and context within the source code.

1.20. Explain data types in Java.

Ans In Java, data types define the type and size of data that can be stored in variables. They are essential for declaring variables, specifying function return types, and ensuring type safety.

1.21. Explain variables and operators in Java.

Ans **Variables :** Variables are named storage locations that hold data, which can be of various data types, such as integers, floating-point numbers, characters, and more.

Operators : Operators are symbols or keywords used to perform various operations on variables and values in expressions.

1.22. What do you understand by control flow statements in Java ?

Ans Control flow statements in Java are used to determine the order in which statements are executed in a program. They allow you to control the flow of your program's execution based on conditions, loops, and method calls.

1.23. What is an array in Java ?

Ans An array is a data structure used to store a collection of elements of the same data type. Arrays provide a way to group multiple values of the same type under a single variable name.

1.24. Define string in Java.

Ans A string is an object that represents a sequence of characters. Strings are used to store and manipulate text-based data.

1.25. Describe inheritance in Java.

Ans Inheritance allows a new class (subclass) to inherit properties and behaviors (fields and methods) from an existing class (superclass).

1.26. What are different types of inheritance in Java ?

Ans Following are the common types of inheritance in Java :

1. Single inheritance
2. Multiple inheritance
3. Multilevel inheritance
4. Hierarchical inheritance

1.27. What is a superclass and subclass ?

Ans. **Superclass :** A superclass, also known as a base class or parent class, is a class from which other classes (subclasses) inherit properties and behaviors.

Subclass : A subclass, also known as a derived class or child class, is a class that inherits properties and behaviors from a superclass.

1.28. What is method overriding in Java ?

Ans. Method overriding in Java allows a subclass to provide a specific implementation of a method that is already defined in its superclass.

1.29. Describe method overloading in Java.

Ans. Method overloading allows you to define multiple methods with the same name in a class but with different parameter lists.

1.30. Explain the concept of encapsulation in Java.

Ans. Encapsulation refers to the bundling of data and the methods that operate on that data into a single unit called a class. Encapsulation helps to hide the internal implementation details of a class.

1.31. Give the advantages of encapsulation in Java.

Ans. Advantages of encapsulation :

1. Data security
2. Controlled access
3. Code flexibility
4. Simplified maintenance

1.32. Define polymorphism and give its types.

Ans. **Polymorphism :** It refers to the ability of different objects to respond to the same method call in a way that is appropriate for their specific types.

Types of polymorphism in Java : There are two main types of polymorphism in Java :

1. Compile-time polymorphism (static binding)
2. Runtime polymorphism (dynamic binding)

1.33. What is abstraction in object-oriented programming ?

Ans. Abstraction refers to the process of simplifying complex systems by breaking them down into smaller, more manageable parts while hiding the unnecessary details.

1.34. Explain the concept of interfaces in Java.

Ans. An interface is like a contract that defines a set of methods that a class must implement if it claims to implement that interface. It only contains method signatures without implementations.

1.35. Describe an abstract class.

Ans. In Java, an abstract class is a class that cannot be instantiated directly but serves as a blueprint for other classes. It is used to define common methods and fields that should be shared among its subclasses.

1.36. What is a package in Java ?

Ans. In Java, a package is a way to organize related classes, interfaces, and sub-packages. It provides a mechanism for grouping related types together, making it easier to manage and maintain a large codebase.

1.37. What is the CLASSPATH in Java ?

Ans. The CLASSPATH in Java is a configuration that specifies where the JVM and compiler should look for classes and resources.

1.38. What is the purpose of setting the CLASSPATH ?

Ans. The purposes of setting the CLASSPATH are as follows :

1. Locating classes and resources.
2. Handling dependencies.
3. Supporting modular development.
4. Avoiding class name conflicts.

1.39. What do you understand by JAR (Java Archive) files in Java ?

Ans. In Java, a JAR file is a compressed file format used to package Java classes, associated metadata, and resources into a single file. JAR files are commonly used for distributing Java libraries, applications, or applets.

1.40. What is 'importing packages in Java' mean ?

Ans. Importing packages in Java is the process of including classes and interfaces from other packages in your Java source code.

1.41. What is the purpose of the import statement ?

Ans. The purpose of the import statement is to simplify and clarify your Java code. It allows you to reference classes and interfaces from other packages using their simple names, making your code more readable and reducing the need for fully qualified class names.

1.42. Explain the concept of static imports in Java.

Ans. Static imports in Java are a feature introduced in Java 5 that allows you to import and use static members (fields and methods) of a class directly without having to prefix them with the class name.

☺☺☺

2 UNIT

Exception Handling (2 Marks Questions)

2.1. What do you mean by exception in Java ?

Ans. In Java, an exception is an event that disrupts the normal flow of program execution. It represents an unexpected condition or error that occurs during runtime.

2.2. How does Java handle exceptions ?

Ans. Java handles exceptions through a combination of try, catch, and finally blocks.

2.3. What do you mean by errors in Java ?

Ans. In Java, errors represent serious, unrecoverable problems that occur during the execution of a program. Unlike exceptions, which can be caught and handled programmatically, errors generally are beyond the control of the application.

2.4. What are the different types of exceptions in Java ? Provide examples for each type.

Ans. In Java, exceptions are categorized into two main types :

1. **Checked exceptions :** Checked exceptions are exceptions that are checked by the compiler at compile time.
2. **Unchecked exceptions :** Unchecked exceptions, also known as runtime exceptions, are not checked by the compiler at compile-time.

2.5. Explain the purpose of try block in exception handling.

Ans. The try block encloses the code that might throw an exception. It is used to define a block of code where exceptions might occur.

2.6. Give the syntax of try block.

Ans. The syntax is :

```
try {
    // Code that might throw an exception
}
```

2.7. Explain the purpose of catch block in exception handling.

Ans. The catch block follows a try block and specifies the type of exception it can handle. It is used to catch and handle exceptions that occur within the associated try block.

2.8. Give the syntax of catch block.

Ans. The syntax is :

```
catch (ExceptionType e) {
    // Handle the exception
}
```

2.9. Explain the purpose of finally block in exception handling.

Ans. The finally block follows a try block and contains code that always executes, regardless of whether an exception occurred or not. It is used for releasing resources, performing cleanup operations, or finalizing tasks.

2.10. Give the syntax of finally block.

Ans. The syntax is :

```
finally {
    // Code that always executes, regardless of whether an
    exception occurred
}
```

2.11. What is the role of the throw keyword in exception handling ?

Ans. In Java, the throw keyword is used to explicitly throw an exception within a method.

2.12. What is the role of the throws keyword in exception handling ?

Ans. In Java, the throws keyword is used to indicate that the method may throw certain types of exceptions during its execution.

2.13. What are inbuilt exceptions ? Give an example.

Ans. In Java, inbuilt exceptions refer to the exceptions that are provided by the Java standard library. These exceptions are predefined and are available for use in Java programs.

2.14. What are user-defined exceptions ?

Ans. User-defined (custom) exceptions are exceptions that are defined by the programmer/user. These exceptions allow developers to create more specific and meaningful error handling mechanisms.

2.15. What is byte stream in Java ?

Ans. In Java, a byte stream is a stream of raw bytes. It is used to perform I/O operations on binary data, such as images, audio, video, or any other type of non-textual data.

2.16. What do you mean by character stream in Java ?

Ans. In Java, character streams are used to perform I/O operations for textual data. Character stream deals with characters instead of raw bytes.

2.17. How do you handle exceptions during file I/O operations in Java ?

Ans. Here's how you can handle exceptions during file I/O operations :
1. Using try-catch blocks
2. Handle specific exceptions
3. Propagate exceptions

2.18. What is a thread in Java ?

Ans. In Java, a thread refers to a lightweight process that exists within a larger process (JVM) and operates independently. Threads allow concurrent execution of multiple tasks within a single application.

2.19. What is thread priority ?

Ans. Thread priority in Java is an integer value that indicates the importance or priority of a thread's execution relative to other threads. Thread priority values range from 1 (lowest priority) to 10 (highest priority). By default, every thread is given priority 5.

2.20. What is thread synchronization ?

Ans. Thread synchronization refers to the coordination of multiple threads to ensure proper and orderly access to shared resources. In a multi-threaded environment, where multiple threads are executing concurrently, synchronization is essential to prevent race conditions.

2.21. Discuss the need for synchronizing threads in a multi-threaded environment.

Ans. The need for synchronizing threads arises due to following reasons :
1. Prevent data corruption
2. Avoid race conditions
3. Maintain data consistency
4. Ensure thread safety

2.22. Describe various mechanisms for synchronizing threads in Java.

Ans. Following are some of the most commonly used mechanisms :
1. Synchronized methods and blocks
2. Reentrant locks (java.util.concurrent.locks.Lock)
3. Semaphore (java.util.concurrent.Semaphore)

2.23. What is inter-thread communication ? How is it achieved in Java ?

Ans. Inter-thread communication is the process of coordinating and exchanging information between multiple threads in a multi-threaded application. It allows threads to synchronize their activities, share data, and coordinate their execution in a controlled manner.

2.24. How inter-thread communication is achieved in Java ?

Ans. In Java, inter-thread communication is typically achieved using three fundamental methods provided by the Object class: wait(), notify(), and notifyAll(). These methods enable threads to wait for certain conditions to be met and to signal other threads when those conditions are satisfied.

Java New Features (2 Marks Questions)

3.1. What are functional interfaces in Java ?

Ans. Functional interfaces in Java are interfaces that contain only one abstract method. They are a key feature introduced in Java to support functional programming paradigms. Functional interface is also known as Single Abstract Method (SAM) interfaces.

3.2. Explain lambda expressions in Java.

Ans. Lambda expressions in Java are a concise way to represent anonymous functions (functions without a name). Lambda expressions are similar to methods, but they do not need a name and can be implemented right in the body of a method.

3.3. How functional interfaces in Java relate to lambda expressions ?

Ans. Functional interfaces in Java are closely related to lambda expressions because lambda expressions can only be used with functional interfaces.

3.4. Explain method references in Java with its types.

Ans. Method references in Java provide a shorthand syntax for writing lambda expressions that call a single method. They allow you to refer to methods or constructors without invoking them.

3.5. What are different types of method references in Java ?

Ans. Types of method references in Java :
1. Reference to a static method.
2. Reference to an instance method of a particular object.
3. Reference to an instance method of an arbitrary object of a particular type.
4. Reference to a constructor.

3.6. How do method references simplify code in Java ?

Ans. Following are several ways method references simplify code :
1. Enhanced readability.
2. Elimination of redundant code.
3. Improved maintainability.
4. Simplified constructor invocation.

3.7. Describe stream API.

Ans. The Stream API in Java provides a powerful and flexible way to process collections of objects in a functional style. It allows developers

to perform aggregate operations on sequences of elements with concise and expressive syntax.

3.8. What are default methods in Java interfaces ?

Ans. Default methods are a feature introduced in Java 8 to enable adding new methods to interfaces without breaking existing implementations.

3.9. How do default methods enable backward compatibility ?

Ans. Default methods enable backward compatibility by allowing existing interfaces to evolve without breaking existing implementations.

3.10. Explain static methods in Java interfaces.

Ans. Static methods are a feature introduced in Java 8 to allow interfaces to contain static methods. Prior to Java 8, interfaces could only declare instance methods.

3.11. Explain Base64 encode and decode.

Ans. Base64 encoding and decoding are techniques used to convert binary data into ASCII characters and vice versa. This conversion is commonly used in scenarios where binary data needs to be represented as text.

3.12. Describe the forEach method in Java.

Ans. In Java, the forEach method is a terminal operation provided by the Stream API. It allows you to perform a specified action for each element in a stream, iterating over the elements sequentially.

3.13. Give the syntax of forEach method in Java.

Ans. General syntax of the forEach method :

```
stream.forEach(action);
```

where,

stream : The stream of elements over which the operation will be performed.

action : A functional interface representing the action to be performed on each element of the stream.

3.14. What is the try-with-resources statement in Java ?

Ans. The try-with-resources statement is a feature use to simplify the management of resources that require closing or releasing after being used. It allows you to declare one or more resources within the parentheses of the try statement.

3.15. Explain type annotations in Java.

Ans. Type annotations in Java allow developers to apply annotations to various types in addition to just declarations. These annotations provide additional metadata about types, which can be used by tools and frameworks.

3.16. What are repeating annotations ?

Ans. Repeating annotations, introduced in Java SE 8, allow multiple instances of the same annotation to be applied to a single program element.

3.17. Discuss the Java Module System.

Ans. The Java Module System, introduced in Java 9, provides a way to modularize Java applications by encapsulating code into discrete units called modules. This allows developers to organize codebase into logical units with well-defined dependencies and boundaries.

3.18. What is diamond syntax in Java ?

Ans. Diamond syntax, also known as the diamond operator (<>), is a feature introduced in Java 7. It allows you to instantiate generic classes without explicitly specifying the type arguments.

3.19. What do you mean by inner anonymous classes in Java ?

Ans. Inner anonymous classes in Java are unnamed classes defined and instantiated inline, typically within the body of another class or method.

3.20. What is local variable type inference ?

Ans. Local variable type inference is a feature in Java 10 that allows the compiler to infer the data type of a variable based on the value assigned to it.

3.21. Explain switch expressions in Java.

Ans. Switch expressions, introduced in Java 12, offer a concise way to express conditional logic. Unlike traditional switch statements, switch expressions return a value based on the evaluated case.

3.22. What is the 'yield' keyword in Java ?

Ans. The 'yield' keyword in Java is used within switch expressions to specify the value to be returned from a case. It indicates the result of evaluating a particular case and terminates the execution of the switch expression.

3.23. Discuss text blocks in Java.

Ans. Text blocks provide a more natural way to write multiline strings in Java. Instead of concatenating multiple string literals or using escape characters for line breaks, you can use text blocks to represent multiline strings directly in your code.

3.24. What are records in Java ?

Ans. A record in Java is a compact way to declare classes that are mainly intended to hold immutable data. They provide a concise syntax for declaring classes that are primarily used to model data rather than behavior.

3.25. Explain sealed classes in Java.

Ans. Sealed classes provide a way to restrict which classes can be subclasses of a particular class or interface. They allow you to define a finite set of classes that are permitted to extend or implement a sealed class or interface.

☺☺☺

4 UNIT

Java Collections Framework (2 Marks Questions)

4.1. What do you mean by Collections in Java ?

Ans. In Java, "Collections" refers to a framework provided in the Java API to manage and manipulate groups of objects. These objects are commonly referred to as elements or items.

4.2. What is the Java Collections framework ?

Ans. The Java Collections framework is a set of classes and interfaces that provide implementations of commonly reusable collection data structures in Java. The data structures can be lists, sets, maps, queues, etc.

4.3. What is Iterator interface ?

Ans. The Iterator interface is used to traverse through elements of a collection. It provides a uniform way to access elements of various collection types without exposing the underlying implementation details.

4.4. What is Collection interface ?

Ans. The Collection interface is the root interface in the Java Collections framework hierarchy. It represents a group of objects, known as elements, and provides a unified way to work with collections of objects in Java.

4.5 What is List interface ?

Ans. List interface is a child interface of the collection interface. This interface is dedicated to the data of the list type in which we can store all the ordered collections of the objects.

4.6. Which classes implement link interface ?

Ans. The classes that implement the list interface are : ArrayList, LinkedList, Vector, and Stack.

4.7. Describe Queue interface.

Ans. The Queue interface in Java represents a collection of elements that follows the First-In-First-Out (FIFO) principle. The Queue interface extends the Collection interface and adds specific methods for adding, removing, and inspecting elements in a queue.

4.8. What do you mean by Set interface ?

Ans. The Set interface in Java represents a collection of unique elements. Meaning that no two elements in the set can be equal according to the equals() method.

4.9. Which classes implement Set interface ?

Ans. Classes that implement the Set interface :
1. HashSet class 2. TreeSet class 3. LinkedHashSet class

4.10. Describe SortedSet interface.

Ans. The SortedSet interface in Java represents a special type of Set that maintains its elements in sorted order. It extends the Set interface and adds methods for accessing and manipulating elements based on their sorted order.

4.11. Explain TreeSet.

Ans. TreeSet is a class in Java that implements the SortedSet interface, providing a sorted collection of unique elements. It uses a Red-Black Tree data structure to maintain elements in sorted order. TreeSet provides a convenient and efficient way to work with sorted collections of unique elements in Java.

4.12. Explain Map interface in Java.

Ans. The Map interface in Java represents a collection of key-value pairs, where each key is associated with a corresponding value. It provides a way to store and retrieve elements based on their keys.

4.13. How can sorting be performed in Java Collections framework ?

Ans. In the Java Collections Framework, sorting can be performed using the following approaches :
1. Using Comparable interface 2. Using Comparator interface
3. Sorting arrays

4.14. Explain Comparable interface in Java Collections framework.

Ans. The Comparable interface in the Java Collections framework is used to define the natural ordering of objects. It provides a way for objects to specify how they should be compared to one another for the purpose of sorting.

4.15. Explain Comparator interface in Java Collections framework.

Ans. The Comparator interface in the Java Collections Framework is used to define custom ordering of objects. The Comparator interface allows for the definition of multiple comparison strategies for a given class.

4.16. What is properties class in Java Collections framework ?

Ans. The Properties class in the Java Collections Framework is a subclass of Hashtable and represents a persistent set of properties, where each property consists of a key-value pair. It is commonly used for handling configuration settings, such as application settings or parameters, where key-value pairs are stored in a text file.

☺☺☺

5 UNIT

Spring Framework and Spring Boot (2 Marks Questions)

5.1. What is a spring framework ?

Ans.

1. Spring is a lightweight framework. It can be thought of as a framework of frameworks because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF, etc.
2. The framework can be defined as a structure where we find solution of the various technical problems.

5.2. What are the different features of spring framework ?

Ans. Following are some of the major features of spring framework :

1. **Inversion of control (IOC) :** The objects give their dependencies instead of creating or looking for dependent objects. This is called inversion of control.
2. **Aspect oriented programming (AOP) :** Aspect oriented programming in spring supports cohesive development by separating application business logic from system services.
3. **Container :** Spring framework creates and manages the life cycle and configuration of the application objects.

5.3. In how many ways can dependency injection be done ?

Ans. In general, dependency injection can be done in three ways, namely :

1. Constructor injection
2. Setter injection
3. Interface injection

In spring framework, only constructor and setter injections are used.

4.4. List the advantages of spring framework.

Ans. Advantages of spring framework :

1. Light weight
2. Flexible
3. Loose coupling
4. Powerful abstraction spring
5. Declarative support
6. Portable

5.5. What are the different components of a spring application ?

Ans. A spring application generally consists of following components :

1. **Interface :** It defines the functions.
2. **Bean class :** It contains properties, its setter and getter methods, functions etc.
3. **Spring aspect oriented programming (AOP) :** Provides the functionality of cross-cutting concerns.
4. **Bean configuration file :** Contains the information of classes and how to configure them.
5. **User program :** It uses the function.

5.6. How many types of IOC containers are there in spring ?

Ans. Types of IOC containers in spring :

1. **BeanFactory :** BeanFactory is like a factory class that contains a collection of beans. It instantiates the bean whenever asked for by clients.
2. **ApplicationContext :** The ApplicationContext interface is built on top of the BeanFactory interface. It provides some extra functionality on top BeanFactory.

5.7. What is AOP ?

Ans. AOP stands for aspect-oriented programming. It is a programming paradigm and methodology that aims to modularize cross-cutting concerns in software systems. It helps to address common software engineering challenges related to cross-cutting concerns, such as code duplication, tangled dependencies, and reduced modularity.

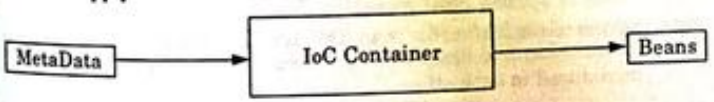
5.8. Explain the importance of IoC in spring.

Ans. IoC is fundamental to the spring framework and plays a vital role in promoting loose coupling, configurability, testability, modular development, and integration with other frameworks. It simplifies application development and maintenance, enhances code quality, and enables building flexible and scalable applications.

5.9. Explain spring beans ?

Ans.

1. They are the objects that form the backbone of the user's application.
2. Beans are managed by the spring IoC container.
3. They are instantiated, configured, wired and managed by a spring IoC container.
4. Beans are created with the configuration metadata that the users supply to the container.



5.10. How many bean scopes are supported by spring ?

Ans. The spring framework supports five scopes. They are :

1. **Singleton :** This provides scope for the bean definition to single instance per spring IoC container.
2. **Prototype :** This provides scope for a single bean definition to have any number of object instances.
3. **Request :** This provides scope for a bean definition to an HTTPrequest.
4. **Session :** This provides scope for a bean definition to an HTTPsession.
5. **Global-session :** This provides scope for a bean definition to a Global HTTP-session.

5.11. Define bean wiring.

Ans. When beans are combined together within the spring container, it's called wiring or bean wiring. The spring container needs to know what beans are needed and how the container should use dependency injection to tie the beans together, while wiring beans.

5.12. What are the limitations with auto wiring ?

Ans. Following are some of the limitations with auto wiring :

1. **Overriding possibility :** You can always specify dependencies using and settings which will override autowiring.
2. **Primitive data type :** Simple properties such as primitives, strings and classes can't be autowired.
3. **Confusing nature :** Always prefer using explicit wiring because autowiring is less precise.

5.13. What do you understand by @Required annotation ?

Ans. @Required is applied to bean property setter methods. This annotation simply indicates that the affected bean property must be populated at the configuration time with the help of an explicit property value in a bean definition or with autowiring. If the affected bean property has not been populated, the container will throw BeanInitializationException.

5.14. What do you understand by @Qualifier annotation ?

Ans. When you create more than one bean of the same type and want to wire only one of them with a property you can use the @Qualifier annotation along with @Autowired to remove the ambiguity by specifying which exact bean should be wired.

5.15. What is spring configuration file ?

Ans. Spring configuration file is an XML file. This file contains the classes information and describes how these classes are configured and introduced to each other.

5.16. What is spring IoC container ?

Ans. The spring IoC creates the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction. The spring container uses dependency injection (DI) to manage the components that make up an application.

5.17. What is a spring boot ?

Ans. Spring Boot provides a good platform for Java developers to develop a stand-alone and production-grade spring application that you can just run. You can get started with minimum configurations without the need for an entire Spring configuration setup.

5.18. What are the advantages of spring boot ?

Ans. Spring boot offers the following advantages to its developers :

1. Easy to understand and develop spring applications.
2. Increases productivity.
3. Reduces the development time.

5.19. What are the goals of spring boot ?

Ans. Spring boot is designed with the following goals :

1. To avoid complex XML configuration in spring.
2. To develop a production ready spring applications in an easier way.
3. To reduce the development time and run the application independently.
4. Offer an easier way of getting started with the application.

5.20. What are the benefits of using spring boot ?

Ans. Benefits of using spring boot :

1. It provides a flexible way to configure Java Beans, XML configurations, and Database Transactions.
2. It offers annotation-based spring application.
3. Eases dependency management.
4. It includes embedded servlet container.

5.21. Difference between spring and spring boot.

Ans.

S. No.	Aspect	Spring	Spring boot
1.	Framework purpose	Comprehensive enterprise application framework.	Simplified development and deployment of spring applications.
2.	Configuration	Requires manual configuration.	Convention-over-configuration approach.
3.	Dependency management	Developers need to manage dependencies manually.	Automatic dependency management.

3.92. What are the advantages RESTful web services ?

Ans. Advantages of RESTful web services :

1. RESTful web services are platform-independent.
2. It can be written in any programming language and can be executed on any platform.
3. It provides different data format like JSON, text, HTML, and XML.
4. It is fast in comparison to SOAP because there is no strict specification like SOAP.

3.93. Explain spring boot logging.

Ans. Spring boot logging is defined as a framework that enables developers to trace out errors that might occur in the running of the application.

3.94. Define spring boot annotations.

Ans. Spring boot annotations is a form of metadata that provides data about a program. In other words, annotations are used to provide supplemental information about a program. It is not a part of the application that we develop. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program.

3.95. What are the effects of running spring boot application as Java application ?

Ans. Running a spring boot application as a Java application provides convenience during development and testing, simplifies deployment as a standalone application, and allows utilizing the embedded web server.

3.96. Discuss, can we call a Java class with annotations as POJO class or not ?

Ans. Yes, a Java class with annotations can still be considered a POJO (Plain Old Java Object) class. The key point is that the class should remain simple, framework-independent, and maintain its core purpose of encapsulating data and behavior. As long as the class adheres to these principles, it can be regarded as a POJO, even if it uses annotation.

☺☺☺