Introduction to Java

1. History of Java.

- Java was developed by sun microsystems in 1991.
- The language was initially named as Oak, but in 1995 it was renamed as Java.
- The first official version of Java was released in 1996.
- James Gosling known as “Father of Java”.
- Java is one of the most widely used programming languages in the world and consistently ranks in the top 3 most popular programming languages.
- Twitter, LinkedIn, Netflix and many enterprise applications were developed using Java.

2. Features of Java (Platform Independent, Object-Oriented, etc.)

- Simple
- Object-oriented
- Portable
- Platform independent
- Secured
- Robust
- Architecture neutral
- Interpreted
- High performance
- Multithreaded
- Distributed
- Dynamic

3. Understanding JVM, JRE, and JDK.

JDK:

- JDK stands for “Java Development Kit”.
- JDK is the complete package used to develop and run Java application.
- It contains JRE (JVM + Libraries) and Development tools
- Used for both development and execution of java programs.

JVM:

- JVM stands for “Java Virtual Machine”.
- JVM is runtime environment that executes Java bytecode.
- It makes java platform-independent by converting byte code into machine code (.class file) of the operating system.
- It runs the java programs.

JRE:

- Java Runtime Environment is the software package that provides the environment to run java applications.
- It contains JVM and Libraries but it does not have development tools like compiler.
- Allows running of java programs but does not provide development tools.

4. Setting up the Java environment and IDE (e.g., Eclipse, IntelliJ)

- Go to the Oracle JDK download page or use OpenJDK.
- Download the latest **JDK** for your OS (Windows/Mac/Linux).
- Install it by following the installation wizard.
- Set JAVA_HOME environment variable (Windows).
- Search for "Environment Variables".
- Under System Variables click New →JAVA_HOME → set it to your JDK path(e.g. C:\PROGRAM_FILES\JAVA\JDK-21)

Setting up the IDE

- Go to https://www.eclipse.org/downloads.
- Download "Eclipse IDE for Java Developers".
- Run the installer and follow the instruction.

5. Java Program Structure (Packages, Classes, Methods).

- **Packages** → Collection of related classes and interfaces, used for organizing code.
- **Classes** → Basic building blocks of Java, contain attributes (fields) and behaviors (methods).
- **Methods** → Define the actions or behaviors of a class, used to perform operations or logic.

A Java program is organized into **packages → classes → methods**.

Data Types, Variables, and Operators

1. Primitive Data Types in Java (int, float, char, etc.)

Java has **8 primitive data types**:

- **byte** → 8-bit integer
- **short** → 16-bit integer
- **int** → 32-bit integer
- **long** → 64-bit integer
- **float** → 32-bit decimal
- **double** → 64-bit decimal
- **char** → 16-bit single character
- **boolean** → true or false values

2. Variable Declaration and Initialization.

- **Declaration** → Telling the compiler about the variable's name and type.
  Example: int age;
- **Initialization** → Assigning a value to the variable.
  Example: age = 20;
- **Combined** → Can declare and initialize in one step.
  Example: int age = 20;

Declaration defines the variable, initialization gives it a value.

3. Operators: Arithmetic, Relational, Logical, Assignment, Unary, and Bitwise.

An operator in java is a symbol that performs an operation one or more operands.

Arithmetic Operator

Used for basic mathematical operations.

Operator	Meaning	Example	Result
+	Addition	10+5	15
-	Subtraction	10-5	5
*	Multiplication	10*5	50
/	Division	10/5	2
%	Modulus	10%5	0

Relational Operator

Used to compare two values. Return True or False.

Operator	Meaning	Example	Result
==	Equal to	10==5	false
!=	Not Equal to	10!=5	true
>	Greater than	10>5	true
<	Less than	10<5	false
>=	Greater than or equal	10>=5	true
<=	Less than Or equal	10<=5	false

Logical Operators

Used for combining multiple conditions.

Operator	Meaning	Example	Result
&&	Logical AND	10>5 && 5<10	true

||	Logical OR	10==5 || 10>5	true
!	Logical NOT	!(10>5)	false

Assignment Operators

Used to assign values to variables.

Operator	Example	Result
=	x=10	x=10
+=	x+=10	x=x+10
-=	x-=10	x=x-10
*=	x*=10	x=x*10
/=	x/=10	x=x/10
%=	x%=10	x=x%10

Unary Operator

Work on single operand.

perform operation like increment and decrement.

Operator	Meaning	Example	Result
+	Positive	+10	10
-	negative	-10	-10
++	Increment	++10	11
--	Decrement	--10	9

Bitwise Operator

Bitwise operator used to perform operations at bit level.

Here we use binary values 5=0101 and 3= 0011

Operator	Meaning	Example	Result
&	AND	5&3	1

|	OR	5 | 3	7
^	XOR	5^3	6
~	Not	5~3	-6
<<	Left shift	5<<3	40
>>	Right shift	5>>3	0
>>>	unsigned Right shift	5>>>3	0

4. Type Conversion and Type Casting.

- **Type Conversion (Implicit / Widening)** → Automatic conversion by Java when a smaller data type is assigned to a larger type.
  Example: int → long → float → double.
- **Type Casting (Explicit / Narrowing)** → Manual conversion by the programmer when a larger data type is assigned to a smaller type.
  Example: double → int.

Conversion is automatic, casting requires explicit instruction.

Control Flow Statements

1. If-Else Statements.

- **If** → Executes a block of code if the condition is true.
  Syntax:
  if (condition) → execute block

- **Else** → Executes a block of code if the condition is false.

Syntax:
if (condition) → execute block1
else → execute block2

- **Nested If-Else** → An if-else inside another if-else, for complex decisions.
  Syntax:
  if (condition1) →
  if (condition2) → execute block1
  else → execute block2
  else → execute block3.

- **Else-If Ladder** → Used when multiple conditions need to be checked in sequence.
  Syntax:
  if (condition1) → execute block1
  else if (condition2) → execute block2
  else if (condition3) → execute block3
  ...
  else → execute blockN

Used for decision-making in programs.

2. Switch Case Statements.

- The **switch-case statement** is a multi-way decision-making statement.
- It allows a variable or expression to be tested against multiple constant values.

- The matching case block executes, and control exits the switch after a break.
- If no case matches, the **default** block executes.

Syntax:

switch (expression) →
  case value1 → execute block1; break;
  case value2 → execute block2; break;
  case value3 → execute block3; break;
  ...
  default → execute blockN;

It is often used as an alternative to the **else-if ladder** for cleaner code when checking many fixed values.

3. Loops (For, While, Do-While)

- **For Loop**
  Used when the number of iterations is known in advance.
  Syntax:
  for (initialization; condition; update) → execute block

- **While Loop**
  Used when the number of iterations is not known; condition is checked before execution.
  Syntax:
  while (condition) → execute block

- **Do-While Loop**
  Similar to while, but executes the block at least once before

checking the condition.

Syntax:

do → execute block while (condition)

4. Break and Continue Keywords.

- **Break**

  The break keyword is used to completely exit from a loop or a switch-case statement before its natural termination.

  Once encountered, it immediately transfers the control to the first statement after the loop or switch.

  It is mainly used when a specific condition is met and no further iterations are required.

  Syntax:

  break;

- **Continue**

  The continue keyword does not terminate the loop but skips the current iteration.

  When encountered, it transfers the control back to the loop condition for the next iteration.

  It is useful when certain conditions need to be skipped without stopping the entire loop.

  Syntax:

  continue;

Classes and Objects

1. Defining a Class and Object in Java.

- **Class**
  A class is a blueprint or template that defines the attributes (fields) and behaviors (methods) of objects.
  It specifies what data and actions the objects of that class will have.

- **Object**
  An object is an instance of a class.
  It represents a real-world entity with its own state (data) and behavior (methods).
  Objects are created using the new keyword and allow access to class attributes and methods.

  Syntax:
  ClassName objectName = new ClassName();

2. Constructors and Overloading.

- **Constructor:**
  A constructor is a special type of method used to initialize an object when it is created.
  It has the same name as the class and **does not have a return type**.
  Constructors can be used to set default values or assign initial values to object attributes.
  If no constructor is explicitly defined, Java provides a **default constructor** automatically.

- **Constructor Overloading**
  Constructor overloading allows a class to have **multiple constructors with the same name** but **different parameter lists** (different number or type of parameters).
  This provides flexibility in object creation, as objects can be initialized in different ways depending on the data provided.
  The compiler determines which constructor to call based on the arguments passed during object creation.
  Multiple constructors can exist in a class with different parameters for overloading.

3. Object Creation, Accessing Members of the Class.

- **Object Creation**
  An object is an instance of a class.
  Objects are created using the new keyword followed by the class constructor.
  Each object has its own copy of the class attributes and can use the class methods.
- **Accessing Members of the Class**
  Class members (fields and methods) can be accessed using the object reference followed by the dot . operator.
  This allows reading or modifying attributes and invoking methods for that specific object.

Syntax:
ClassName objectName = new ClassName();
objectName.attribute → access field
objectName.method() → call method

4. this Keyword.

- Refers to the **current object** of a class.
- Used to **distinguish between class attributes and method or constructor parameters** when they have the same name.
- Can be used to **call another constructor** in the same class (constructor chaining).
- Helps in passing the current object as an argument to methods or constructors.

Syntax:
this.attribute → refers to current object's field
this.method() → calls current object's method
this(parameters) → calls another constructor in the same class

Methods in Java

1. Defining Methods.

- Methods are **blocks of code** that perform a specific task and can be called whenever needed.
- They help in **reusing code**, improving readability, and organizing programs into logical units.
- A method can have **parameters** to accept input and may **return a value** using the return keyword.
- Methods are defined with an **access modifier**, **return type**, **name**, and **parameter list** followed by a body containing the code.

Syntax:

accessModifier returnType methodName(parameters) →

    // code block

    return value; (if applicable)

2. Method Parameters and Return Types.

- **Parameters**
    - Used to pass information or values to a method when it is called.
    - Can be of any data type, including primitives, objects, or arrays.
    - Allow methods to perform operations using different inputs.
- **Return Types**
    - Specify the type of value a method will return after execution.
    - Can be any data type or void if the method does not return anything.
    - The return statement is used to send the result back to the caller.

Syntax:

accessModifier returnType methodName(parameterList) →

    // code block

    return value; (if returnType is not void)

3. Method Overloading.

- Occurs when a class has **multiple methods with the same name** but **different parameter lists** (number, type, or order of parameters).
- Allows performing similar operations in different ways using the same method name.
- Improves code readability and flexibility.
- The compiler determines which method to call based on the arguments passed.

Syntax:

```
methodName();
methodName(parameterType1 param1);
methodName(parameterType1 param1, parameterType2 param2);
```

4. Static Methods and Variables.

Static Variables:

- Belong to the class rather than any specific object.
- Shared among all instances of the class.
- Useful for values common to all objects.
- Can be accessed using the class name without creating an object.

Static Methods:

- Belong to the class and can be called without creating an object.

- Can directly access only **static variables** and other static methods.
- Cannot access instance variables or methods directly.

Syntax:

static datatype variableName;

static returnType methodName(parameters) →

    // code block

Object-Oriented Programming (OOPs) Concepts

1. Basics of OOP: Encapsulation, Inheritance, Polymorphism, Abstraction.

- **Encapsulation**
  - Combines **data (fields)** and **methods** that manipulate the data into a single class.
  - Protects the internal state of an object by using **access modifiers** (private, public, protected).
  - Provides **getter and setter methods** to control access to private data.
  - Ensures data integrity and security.
- **Inheritance**
  - Allows a class (child/subclass) to inherit **properties and methods** from another class (parent/superclass).
  - Supports **code reusability** and logical hierarchy.

    - Can be **single, multilevel, hierarchical, or multiple (through interfaces)**.
    - Reduces redundancy and makes maintenance easier.
- **Polymorphism**
    - Means “many forms”; an object can behave differently in different contexts.
    - **Compile-time polymorphism**: Method overloading – same method name with different parameters.
    - **Runtime polymorphism**: Method overriding – a subclass provides a specific implementation of a method defined in the parent class.
    - Improves flexibility and scalability of code.
- **Abstraction**
    - Focuses on **what an object does** rather than **how it does it**.
    - Hides implementation details using **abstract classes** and **interfaces**.
    - Helps in designing a clear and simplified interface while hiding complex logic.
    - Supports modularity and maintainability in large programs.

2. Inheritance: Single, Multilevel, Hierarchical.

Definition:
Inheritance is a mechanism in object-oriented programming where a class (called subclass or child class) acquires the properties (fields) and behaviors (methods) of another class (called superclass or parent class). It allows **code reuse**, establishes a **hierarchical relationship**, and supports **polymorphism**.

- **Single Inheritance**
    - A subclass inherits from **one parent class**.
    - The subclass can access the parent's fields and methods, and can also add its own.
    - Simple and easy to manage; promotes **code reuse** for basic relationships.
    - Example: Class Car inherits from class Vehicle.
- **Multilevel Inheritance**
    - A subclass is derived from another subclass, forming a **chain of inheritance**.
    - Attributes and methods are passed down the chain, allowing **gradual extension of behavior**.
    - Useful for modeling hierarchical or layered relationships.
    - Example: Class ElectricCar inherits from Car, which inherits from Vehicle.
- **Hierarchical Inheritance**
    - Multiple subclasses inherit from **a single parent class**.

    - Each subclass can have its **own additional features** while sharing common behavior from the parent.
    - Reduces code duplication and allows specialization of different objects.
    - Example: Classes Car and Bike inherit from class Vehicle.
- **Key Benefits of Inheritance**
    - Promotes **code reusability**.
    - Supports **logical hierarchy and organization** of classes.
    - Makes programs **easier to maintain and extend**.
    - Enables **polymorphism** through method overriding.

3. Method Overriding and Dynamic Method Dispatch.

- **Method Overriding**
    - Occurs when a **subclass provides its own implementation** of a method already defined in its parent class.
    - The method in the subclass must have the **same name, return type, and parameters** as the parent method.
    - Enables a subclass to **modify or extend behavior** of the parent class.
    - Supports **runtime polymorphism**.

- **Dynamic Method Dispatch**
    - Mechanism by which a call to an **overridden method** is resolved at **runtime**, not compile-time.
    - The **type of object** (not reference) determines which method implementation is executed.
    - Allows flexibility: a **parent class reference** can refer to a **child class object**, and the overridden method of the child is called.
- **Benefits**
    - Enables **runtime polymorphism**.
    - Promotes **flexible and maintainable code**.
    - Helps in implementing **generalized code** that can work with multiple subclass objects.

Constructors and Destructors

1. Constructor Types (Default, Parameterized).

- **Default Constructor**
    - A constructor with **no parameters**.
    - Provided automatically by Java if no constructor is explicitly defined.
    - Initializes objects with **default values** (e.g., numeric types → 0, boolean → false, object references → null).
    - Can also be explicitly defined to perform additional initialization.

- **Parameterized Constructor**
    - A constructor that **accepts arguments** to initialize object attributes with specific values.
    - Allows creating objects with **custom initial states**.
    - Supports **constructor overloading**, enabling multiple constructors with different parameters in the same class.
- **Key Points**
    - Constructors have the **same name as the class** and **do not have a return type**.
    - Overloaded constructors improve **flexibility** and **code readability**.
    - They ensure **proper initialization** of objects at the time of creation.
    - Used extensively in object-oriented programming to set **default or user-defined states**.

2. Copy Constructor (Emulated in Java).

- A **copy constructor** creates a new object by copying the **state (values of attributes)** from an existing object of the same class.
- Java does not provide a built-in copy constructor, so it must be **emulated manually** by defining a constructor that takes an object of the same class as a parameter.

- Ensures that the new object has the **same attribute values** as the existing object.
- Useful for creating **duplicate objects** without affecting the original object.
- Helps in **deep copying** when combined with object cloning techniques for complex objects.

Key Points

- Takes an object of the same class as a parameter.
- Copies values from the existing object to the new object.
- Prevents unintentional modification of the original object.

3. Constructor Overloading.

- Occurs when a class has **multiple constructors with the same name** but **different parameter lists** (number, type, or order of parameters).
- Allows objects to be initialized in **different ways** depending on the arguments provided.
- Improves **flexibility**, **code readability**, and **reusability**.
- Supports **compile-time polymorphism**, as the compiler determines which constructor to call based on the arguments.
- Can include **default constructor**, **parameterized constructors**, and even a **copy constructor** in the same class.

Key Points

- Constructors must have the **same name as the class**.
- Overloaded constructors **cannot differ only by return type**.
- Provides controlled and versatile object initialization.

4. Object Life Cycle and Garbage Collection.

- **Object Life Cycle**
    - **Creation** → An object is created using the new keyword and a constructor is called.
    - **Usage** → Object's methods and attributes are accessed for performing operations.
    - **Eligibility for Garbage Collection** → When there are **no references** pointing to the object, it becomes eligible for garbage collection.
- **Garbage Collection**
    - Automatic process by the **Java Virtual Machine (JVM)** to **free memory** by destroying unreferenced objects.
    - Helps prevent **memory leaks** and optimizes memory usage.
    - Invoked by JVM, but can be suggested using System.gc().
    - Finalization (finalize() method) can be used to perform cleanup before the object is removed.

Key Points

- Objects are automatically destroyed by the JVM when no longer needed.
- Programmers do not need to explicitly deallocate memory.
- Efficient memory management improves program performance.

Arrays and Strings

1. One-Dimensional and Multidimensional Arrays.

- **One-Dimensional Arrays**
  - Stores elements in a **single row**.
  - All elements are of the **same data type**.
  - Accessed using a **single index**.
  - Useful for storing linear data like a list of numbers or names.
- **Multidimensional Arrays**
  - Stores elements in **two or more dimensions**, like a matrix or table.
  - Accessed using **multiple indices** (row and column for 2D arrays).
  - Useful for representing **tables, grids, or matrices** in programs.

Key Points

- Both types allow efficient storage and manipulation of multiple values.
- Multidimensional arrays extend the concept of one-dimensional arrays to **complex data structures**.
- Indices always start from 0.

2. String Handling in Java: String Class, StringBuffer, StringBuilder.

- **String Class**
  - Represents **immutable sequences of characters**.
  - Once created, the content **cannot be changed**.
  - Supports methods like **length(), substring(), concat(), equals(), charAt()**.
- **StringBuffer**
  - Represents **mutable strings**; content can be modified.
  - Thread-safe (synchronized), so suitable for **multithreaded environments**.
  - Supports methods like **append(), insert(), delete(), reverse()**.
- **StringBuilder**
  - Similar to StringBuffer, but **not synchronized** (faster in single-threaded programs).

- Used for **efficient string manipulation** where thread safety is not required.

Key Points

- Use **String** for fixed text, **StringBuffer** for thread-safe modifications, and **StringBuilder** for fast, non-thread-safe modifications.
- Proper choice improves **performance and memory management**.

3. Array of Objects.

- An array of objects is a **collection where each element is a reference to an object** of a particular class.
- It allows handling **multiple instances of a class** together using a single variable.
- Each element in the array must be **explicitly initialized** with a new object before accessing its members.
- Supports **iteration using loops**, making it easy to perform operations on all objects, such as displaying details or updating attributes.
- Useful in scenarios like managing a **list of students, employees, or products**, where each object represents a separate entity but shares the same class structure.
- Combines the **organization of arrays** with the **capabilities of objects**, enabling structured and efficient data management.

Key Points

- Access object attributes and methods using **array index followed by dot operator**.
- Helps in implementing **batch processing** of objects.
- Reduces the need for multiple separate object references and simplifies **memory management**.

4. String Methods (length, charAt, substring, etc.)

- **length()** → Returns the number of characters in the string.
- **charAt(index)** → Returns the character at the specified index.
- **substring(startIndex, endIndex)** → Extracts a portion of the string from startIndex to endIndex - 1.
- **concat(str)** → Concatenates the specified string to the current string.
- **equals(str)** → Compares two strings for equality (case-sensitive).
- **equalsIgnoreCase(str)** → Compares two strings ignoring case differences.
- **indexOf(ch/str)** → Returns the index of the first occurrence of a character or substring.
- **lastIndexOf(ch/str)** → Returns the index of the last occurrence.
- **toUpperCase() / toLowerCase()** → Converts the string to uppercase or lowercase.

- **trim()** → Removes leading and trailing whitespace.
- **replace(oldChar/newChar or oldStr/newStr)** → Replaces characters or substrings in the string.

Key Points

- Strings in Java are **immutable**, so these methods **return new strings** instead of modifying the original.
- Useful for **manipulating and analyzing text data** efficiently.

Inheritance and Polymorphism

1. Inheritance Types and Benefits.

- **Types of Inheritance**
    1. **Single Inheritance** → A subclass inherits from one parent class.
    2. **Multilevel Inheritance** → A subclass inherits from another subclass, forming a chain.
    3. **Hierarchical Inheritance** → Multiple subclasses inherit from a single parent class.
- **Benefits of Inheritance**
    - **Code Reusability** → Inherited methods and fields can be used in subclasses.
    - **Logical Hierarchy** → Organizes classes in a parent-child relationship.
    - **Extensibility** → New features can be added to subclasses without modifying existing code.

- **Polymorphism Support** → Enables method overriding for runtime flexibility.
- **Maintenance** → Reduces redundancy, making programs easier to maintain and update.

2. Method Overriding.

- Occurs when a **subclass provides its own implementation** of a method already defined in the parent class.
- The overridden method must have the **same name, return type, and parameters** as the parent method.
- Enables a subclass to **modify or extend the behavior** of the parent class method.
- Supports **runtime polymorphism**, allowing the program to decide which method to call based on the **object type at runtime**.
- Improves **flexibility, code reuse, and maintainability**.

Key Points

- Only **inherited methods** can be overridden.
- Access level of the overridden method **cannot be more restrictive** than the parent method.
- Constructors **cannot be overridden**.

3. Dynamic Binding (Run-Time Polymorphism).

- Dynamic binding occurs when a **method call is resolved at runtime** rather than compile-time.
- Achieved through **method overriding**, where a parent class reference points to a child class object.
- The **actual method that gets executed** is determined by the **type of object**, not the reference.
- Enables **flexible and extensible code**, allowing a single interface to work with multiple implementations.
- Supports **runtime polymorphism**, making programs more **modular and maintainable**.

Key Points

- Parent class reference can refer to **any subclass object**.
- Method execution is **decided during program execution**, not at compile time.
- Helps in implementing **generalized code** for different object types.

4. Super Keyword and Method Hiding.

- **Super Keyword**
    - Refers to the **immediate parent class** of the current object.
    - Used to **access parent class members** (fields, methods) that are hidden or overridden in the subclass.

    - Can be used to **call parent class constructor** from a subclass constructor.
    - Helps in **reusing and extending parent class functionality**.
- **Method Hiding**
    - Occurs when a **static method** in a subclass has the **same name** as a static method in the parent class.
    - Unlike overriding, **method hiding is resolved at compile-time**.
    - The method called depends on the **type of reference**, not the object.

Key Points

- super ensures access to **parent class implementation**.
- Method hiding applies **only to static methods**, not instance methods.
- Provides control over **which class's method or field** is being used.

Interfaces and Abstract Classes

1. Abstract Classes and Methods.

- **Abstract Class**
    - A class declared with the abstract keyword.
    - Cannot be instantiated directly; serves as a **base class** for other classes.

    - Can contain **abstract methods** (without body) and **concrete methods** (with body).
    - Used to provide a **common template** for subclasses.
- **Abstract Method**
    - A method declared without a body using the abstract keyword.
    - Must be **implemented by subclasses**.
    - Defines a **contract** that all subclasses must follow.

Key Points

- Supports **abstraction** by hiding implementation details.
- Promotes **code reuse and consistency** across subclasses.
- Helps in designing **flexible and extensible systems**.

2. Interfaces: Multiple Inheritance in Java.

- An **interface** is a collection of abstract methods and constants.
- Java supports **multiple inheritance** using interfaces, allowing a class to implement **multiple interfaces**.
- A class that implements an interface must **provide implementations** for all its abstract methods.
- Helps avoid the **ambiguity and complexity** of multiple class inheritance.

Key Points

- Supports **abstraction** and **polymorphism**.

- Enables a class to **inherit behavior from multiple sources**.
- Promotes **modular and maintainable code**.
- Syntax for multiple inheritance:
  ClassName implements Interface1, Interface2, Interface3 → { ... }

3. Implementing Multiple Interfaces.

- A class can **implement more than one interface** to inherit behavior from multiple sources.
- The class must **override all abstract methods** from all the implemented interfaces.
- Helps achieve **multiple inheritance** safely in Java without the issues of class-based multiple inheritance.
- Useful for creating **flexible, modular, and reusable code** where a class can have multiple capabilities.

Key Points

- Use a **comma-separated list** of interface names after the implements keyword.
- Ensures **contract compliance** by enforcing implementation of all interface methods.
- Supports **polymorphism**, allowing objects to be treated as instances of multiple types.

Packages and Access Modifiers

1. Java Packages: Built-in and User-Defined Packages.

- **Package**
  - A package is a **collection of related classes, interfaces, and sub-packages**.
  - Provides **modularity, organization, and namespace management**.
- **Built-in Packages**
  - Provided by Java for **common functionalities**.
  - Examples: java.util (collections, date), java.io (file handling), java.lang (core classes).
- **User-Defined Packages**
  - Created by programmers to **organize their own classes and interfaces**.
  - Helps in **reusability and better project structure**.

Key Points

- Classes from a package can be accessed using the import statement.
- Packages prevent **name conflicts** in large projects.
- Promote **modular and maintainable code design**.

2. Access Modifiers: Private, Default, Protected, Public.

- **Private**
  - Members are accessible **only within the same class**.
  - Provides **maximum data hiding**.
- **Default (Package-Private)**
  - No modifier is specified.
  - Members are accessible **only within the same package**.
- **Protected**
  - Members are accessible **within the same package** and in **subclasses (even in different packages)**.
- **Public**
  - Members are accessible from **anywhere** in the program.

Key Points

- Control **visibility and accessibility** of classes, methods, and variables.
- Help achieve **encapsulation** and maintain **data security**.
- Choose modifiers based on **scope and intended access**.

3. Importing Packages and Classpath.

- **Importing Packages**
  - Allows access to classes and interfaces from other packages.
  - Done using the import keyword.
  - Can import a **single class** or all classes in a package using *.
  - Example: import java.util.Scanner; or import java.util.*;
- **Classpath**
  - Specifies the **location of user-defined and built-in classes** for the Java compiler and JVM.
  - Helps the program **locate classes and packages** at compile-time and runtime.
  - Can be set using the **CLASSPATH environment variable** or -cp option in command line.

Key Points

- Importing reduces the need to **use fully qualified class names** repeatedly.
- Correct classpath ensures that **all dependencies** are found and the program runs without errors.

Exception Handling

1. Types of Exceptions: Checked and Unchecked.

- **Checked Exceptions**
  - Exceptions that are **checked at compile-time**.
  - Must be **handled using try-catch** or declared with throws.
  - Examples: IOException, SQLException, FileNotFoundException.
- **Unchecked Exceptions**
  - Exceptions that occur **at runtime** and are **not checked by the compiler**.
  - Typically caused by **programming errors** like invalid input or arithmetic mistakes.
  - Examples: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException.

Key Points

- Checked exceptions enforce **mandatory handling**, improving program reliability.
- Unchecked exceptions indicate **logical errors** that should be fixed in code.
- Proper exception handling ensures **robust and error-resilient programs**.

2. try, catch, finally, throw, throws.

- **try**
  - Block where **code that might throw an exception** is placed.
  - Must be followed by **catch** or **finally**.
- **catch**
  - Handles exceptions thrown in the try block.
  - Can have **multiple catch blocks** for different exception types.
- **finally**
  - Block that **always executes**, regardless of whether an exception occurs.
  - Used for **cleanup activities**, like closing files or releasing resources.
- **throw**
  - Used to **explicitly throw an exception** from a method or block.
  - Syntax: throw new ExceptionType("message");
- **throws**
  - Declares **exceptions that a method can throw** to the caller.
  - Used in method signature to indicate **potential checked exceptions**.
  - Syntax: void method() throws IOException, SQLException { ... }

Key Points

- Together, these keywords provide **structured exception handling** in Java.
- Ensure **program stability** by managing runtime errors gracefully.

3. Custom Exception Classes.

- A custom exception is a **user-defined class** that extends Exception (for checked exceptions) or RuntimeException (for unchecked exceptions).
- Allows creating **meaningful and specific error messages** for particular program requirements.
- Helps in **handling application-specific conditions** more effectively than generic exceptions.
- Can be thrown using the throw keyword and declared with throws if it's a checked exception.

Key Points

- Extend **Exception** for checked or **RuntimeException** for unchecked exceptions.
- Include constructors to **pass custom error messages**.
- Improves **readability, maintainability, and clarity** of exception handling.

Multithreading

1. Introduction to Threads.

- A **thread** is the **smallest unit of execution** within a program.
- Java supports **multithreading**, allowing multiple threads to run **concurrently**.
- Threads share the **same memory space** but have their **own execution path**.
- Used to **perform multiple tasks simultaneously**, improving performance and responsiveness.
- Can be created by **extending the Thread class** or **implementing the Runnable interface**.

Key Points

- Enables **parallel execution** of code.
- Helps in **efficient CPU utilization** and **responsive applications**.
- Each thread has **its own lifecycle** including states like New, Runnable, Running, and Terminated.

2. Creating Threads by Extending Thread Class or Implementing Runnable Interface.

- **By Extending Thread Class**
    - Create a class that **extends Thread**.
    - Override the run() method with the code to execute.

    - Create an object of the class and call start() to begin the thread.
- **By Implementing Runnable Interface**
    - Create a class that **implements Runnable**.
    - Implement the run() method.
    - Create a Thread object by passing the Runnable object, then call start().

Key Points

- Both methods allow **concurrent execution** of code.
- Implementing Runnable is preferred when the class already **extends another class**.
- start() begins the thread, while run() contains the thread's task.

3. Thread Life Cycle.

- A thread in Java goes through **different states** during its lifetime:

1. **New (Created)**
    - Thread object is created but start() is not yet called.
2. **Runnable**
    - start() is called; thread is ready to run and waiting for CPU scheduling.
3. **Running**

    - The thread scheduler selects the thread, and its run() method executes.
4. **Waiting/Blocked**
    - Thread waits for another thread to perform a specific action or resource availability.
5. **Timed Waiting**
    - Thread waits for a specified period using methods like sleep() or join(timeout).
6. **Terminated (Dead)**
    - Thread finishes execution or is stopped; cannot be restarted.

Key Points

- Thread transitions between states are managed by the **JVM and thread scheduler**.
- Understanding the lifecycle helps in **efficient thread management** and synchronization.

4. Synchronization and Inter-thread Communication.

- **Synchronization**
    - Controls access to **shared resources** to prevent **data inconsistency** when multiple threads access them simultaneously.
    - Achieved using the synchronized keyword with **methods or blocks**.

    - Ensures **only one thread** can execute the synchronized code at a time.

- **Inter-Thread Communication**
    - Mechanism that allows threads to **communicate and coordinate** with each other.
    - Uses methods like wait(), notify(), and notifyAll() for **cooperative multitasking**.
    - wait() → thread pauses until notified.
    - notify() → wakes up a single waiting thread.
    - notifyAll() → wakes up all waiting threads.

Key Points

- Synchronization prevents **race conditions**.
- Inter-thread communication improves **thread cooperation** and **resource management**.
- Essential for writing **safe and efficient multithreaded programs**.

File Handling

1. Introduction to File I/O in Java (java.io package).

- **File I/O (Input/Output)** in Java allows programs to **read from and write to files**.

- Provided by the **java.io package**, which contains classes like File, FileReader, FileWriter, BufferedReader, and BufferedWriter.
- Supports handling of **text and binary data**.
- Enables **persistent storage** of data outside the program's memory.
- Essential for tasks like **data logging, configuration, and file-based communication**.

Key Points

- Streams are used to **transfer data**:
  - **Byte streams** (InputStream/OutputStream) for binary data.
  - **Character streams** (Reader/Writer) for text data.
- Proper handling of **exceptions** like IOException is necessary.
- Closing streams is important to **release system resources**.

2. FileReader and FileWriter Classes.

- **FileReader**
  - A **character-based input stream** used to read data from files.
  - Reads **text data** character by character or using arrays.
  - Automatically converts bytes to characters using the **default character encoding**.

- **FileWriter**
    - A **character-based output stream** used to write data to files.
    - Writes **text data** to files, either overwriting or appending.
    - Can write single characters, arrays, or strings.

Key Points

- Both are part of the **java.io package**.
- Used for **simple file handling** with text files.
- Must **close the streams** after use to release system resources.
- Handle **IOException** to manage errors during file operations.

3. BufferedReader and BufferedWriter.

- **BufferedReader**
    - A **character-based input stream** that reads text from a character input stream efficiently.
    - Uses an internal **buffer** to reduce the number of I/O operations.
    - Provides methods like read(), readLine() to read characters or lines of text.
- **BufferedWriter**
    - A **character-based output stream** that writes text to an output stream efficiently.

- Uses an internal **buffer** to minimize direct writes to the file.
- Provides methods like write() and newLine() for writing characters or lines.

Key Points

- Both improve **performance** compared to FileReader/FileWriter for large text files.
- Require **closing the stream** after use to flush buffers and release resources.
- Commonly used for **line-by-line reading and writing** of text files.

4. Serialization and Deserialization.

- **Serialization**
  - Converts an object into a **sequence of bytes** to save its state to a file or transmit over a network.
  - Implemented by having the class implement the **Serializable** interface.
  - All non-transient fields are saved; **transient fields** are skipped.
  - Helps in **persistence, caching, and remote communication** (RMI, web services).
- **Deserialization**
  - Converts the byte stream back into a **replica of the original object**.

    - Uses classes like ObjectInputStream to read objects from a file or network.
    - Restores the exact state of the object at the time of serialization.

Key Points

- Both processes require handling **IOException**; deserialization also needs **ClassNotFoundException**.
- Enables **object transfer between JVMs** and **storage of complex objects**.
- Care must be taken with **class versioning**; incompatible changes can lead to InvalidClassException.
- Used widely in **distributed applications, caching, and session management**.

Collections Framework

1. Introduction to Collections Framework.

- The **Collections Framework** in Java provides **predefined interfaces, classes, and algorithms** to store, manage, and manipulate groups of objects.
- Part of the **java.util package**.
- Includes **core interfaces** like List, Set, Queue, and Map.
- Provides **implementing classes** such as ArrayList, LinkedList, HashSet, HashMap, and TreeMap.

- Supports **dynamic data structures**, unlike arrays which have fixed size.

Key Points

- Offers **efficient storage, retrieval, and manipulation** of data.
- Reduces the need to **write custom data structures**.
- Supports **generic types**, allowing type-safe collections.
- Includes **utility classes** like Collections and Arrays for sorting, searching, and other operations.

2. List, Set, Map, and Queue Interfaces.

- **List**
    - An **ordered collection** that allows **duplicate elements**.
    - Elements can be accessed by **index**.
    - Common implementations: ArrayList, LinkedList, Vector.
- **Set**
    - A collection that **does not allow duplicates**.
    - Does not guarantee any specific **order** (unless using LinkedHashSet or TreeSet).
    - Common implementations: HashSet, LinkedHashSet, TreeSet.

- **Map**
    - Stores **key-value pairs**, where keys are **unique** and values can be duplicated.
    - Does not extend the Collection interface.
    - Common implementations: HashMap, TreeMap, LinkedHashMap.
- **Queue**
    - A collection used to hold elements prior to processing.
    - Usually follows **FIFO (First-In-First-Out)** order.
    - Common implementations: LinkedList, PriorityQueue, ArrayDeque.

Key Points

- Interfaces define the **contract**, while implementations provide the **specific behavior**.
- Collections Framework provides **flexible, efficient, and reusable data structures**.

3. ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap.

- **ArrayList**
    - Resizable **array implementation** of the List interface.
    - Allows **duplicates** and maintains **insertion order**.
    - Provides **fast random access** but slower insertions/deletions in the middle.

- **LinkedList**
  - Doubly-linked list implementation of List and Queue interfaces.
  - Maintains **insertion order** and allows **duplicates**.
  - Efficient for **insertions and deletions** but slower for random access.
- **HashSet**
  - Implements the Set interface using a **hash table**.
  - **No duplicates** allowed; does not maintain order.
  - Fast for **add, remove, and search operations**.
- **TreeSet**
  - Implements Set using a **Red-Black tree**.
  - **No duplicates** and **sorted order** maintained.
  - Useful when **ordered elements** are required.
- **HashMap**
  - Implements Map using a **hash table**.
  - Stores **key-value pairs**; keys are unique, values can duplicate.
  - Does **not maintain order** of elements.
- **TreeMap**
  - Implements Map using a **Red-Black tree**.
  - Maintains **sorted order of keys**.
  - Useful for **ordered key-value pairs** and range-based operations.

Key Points

- Choose the implementation based on **performance needs**, **ordering**, and **duplication rules**.
- Provides **dynamic, efficient, and flexible** data management compared to arrays.

4. Iterators and ListIterators.

- **Iterator**
  - An interface used to **traverse elements** of a collection sequentially.
  - Methods include: hasNext(), next(), and remove().
  - Works for **all collection types** like List, Set, but **cannot traverse backward**.
- **ListIterator**
  - Extends Iterator and is specific to **List implementations**.
  - Allows **bidirectional traversal** (forward and backward).
  - Additional methods: hasPrevious(), previous(), add(), set().
  - Provides **more control** over element access and modification.

Key Points

- Iterators provide a **uniform way** to traverse collections.

- ListIterator is **more powerful** but limited to lists.
- Helps avoid **index-based traversal**, improving **flexibility and safety**.

Java Input/Output (I/O)

1. Streams in Java (InputStream, OutputStream).

- **Stream**
  - A sequence of **data elements** that can be read from or written to.
  - Supports **byte-oriented** or **character-oriented** I/O operations.
- **InputStream**
  - Abstract class for **reading bytes** from a source (file, network, etc.).
  - Common methods: read(), read(byte[] b), close().
  - Subclasses include FileInputStream, BufferedInputStream.
- **OutputStream**
  - Abstract class for **writing bytes** to a destination.
  - Common methods: write(int b), write(byte[] b), flush(), close().
  - Subclasses include FileOutputStream, BufferedOutputStream.

Key Points

- Streams provide **sequential access** to data.
- Byte streams handle **binary data**, character streams handle **text data**.
- Proper **closing of streams** is required to release system resources.
- Used extensively in **file handling, network communication, and serialization**.

2. Reading and Writing Data Using Streams.

- **Reading Data**
  - Use **InputStream** or **Reader** classes to read data from a source like a file or network.
  - Methods like read() or read(byte[] buffer) are used to **retrieve bytes or characters**.
  - Can be buffered using BufferedInputStream or BufferedReader for **efficiency**.
- **Writing Data**
  - Use **OutputStream** or **Writer** classes to write data to a destination.
  - Methods like write(int b) or write(byte[] buffer) output bytes or characters.
  - flush() ensures that all buffered data is **actually written** to the destination.

Key Points

- Streams allow **sequential processing** of data.
- Byte streams handle **binary data**, character streams handle **text data**.
- Proper **closing of streams** is necessary to avoid **resource leaks**.
- Buffering improves **performance for large data operations**.

3. Handling File I/O Operations.

- Java provides classes in **java.io** for performing **file operations** like creation, reading, writing, and deletion.
- **Creating a File** → Using the File class to represent file paths and create new files.
- **Reading Files** → Using FileReader, BufferedReader, or Scanner to read text data.
- **Writing Files** → Using FileWriter, BufferedWriter, or PrintWriter to write text data.
- **Checking File Properties** → Methods like exists(), canRead(), canWrite(), length().
- **Deleting Files** → delete() method of the File class.

Key Points

- File I/O requires **exception handling** (IOException) to manage errors.
- Streams must be **closed** after use to release system resources.

- Supports both **text and binary file handling**.
- Enables **persistent storage and retrieval** of program data.
