CODTECH IT SOLUTIONS PVT.LTD

IT SERVICES & IT CONSULTING

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

Java Programming Material

OUR PARTNERS & CERTIFICATIONS

Google Partner

aws PARTNER

CERTIFIED ISO 9001 CERTIFIED

MSME MICRO, SMALL & MEDIUM ENTERPRISES

सत्यमेव जयते

AICTE All India Council for Technical Education

सत्यमेव जयते

MINISTRY OF CORPORATE AFFAIRS

GOVERNMENT OF INDIA

CODTECH IT SOLUTIONS PVT.LTD

IT SERVICES & IT CONSULTING

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

1. Introduction to Java

- Overview of Java
- Features of Java
- Java Development Kit (JDK), Java Runtime Environment (JRE), Java Virtual Machine (JVM)
- Setting up Java and IDE (Eclipse, IntelliJ IDEA, VS Code)
- Writing and running the first Java program

2. Java Fundamentals

- Data Types and Variables
- Operators in Java
- Input and Output (Scanner, BufferedReader)
- Control Statements (if-else, switch-case)
- Loops (for, while, do-while)
- Break and Continue Statements
-

3. Object-Oriented Programming (OOP) in Java

- Classes and Objects
- Constructors and Methods
- Method Overloading and Overriding
- Access Modifiers (public, private, protected, default)
- Static and Instance Variables & Methods
- Encapsulation
- Inheritance (Single, Multilevel, Hierarchical)
- Polymorphism (Compile-time and Run-time)
- Abstraction (Abstract Classes, Interfaces)

4. Arrays and Strings

- One-Dimensional and Multi-Dimensional Arrays
- Array Operations (Sorting, Searching, Copying)
- Strings in Java (String, StringBuffer, StringBuilder)
- String Methods and Manipulation

5. Exception Handling

- Types of Exceptions (Checked and Unchecked)
- try, catch, finally, throw, and throws
- Custom Exceptions

6. Java Collections Framework (JCF)

- List (ArrayList, LinkedList)
- Set (HashSet, TreeSet)
- Map (HashMap, TreeMap, LinkedHashMap)
- Queue and Deque
- Iterators and Stream API

7. File Handling in Java

- Reading and Writing Files
- BufferedReader and BufferedWriter
- File Handling with FileInputStream and FileOutputStream
- Serialization and Deserialization

9. JDBC (Java Database Connectivity)
   - Introduction to JDBC
   - Connecting Java with MySQL
   - CRUD Operations using JDBC
   - Prepared Statements and Callable Statements

10. Advanced Java Concepts
   - Lambda Expressions
   - Streams and Functional Programming
   - Java 8 Features (Optional, Method References, Default Methods)
   - Reflection API

CHAPTER -1
Introduction to Java

What is Java?

Java is a high-level, object-oriented programming language developed by James Gosling at Sun Microsystems (now owned by Oracle) in 1995. It is designed to be platform-independent, meaning that Java programs can run on any device that has a Java Virtual Machine (JVM).

Key Features of Java

1. **Platform Independence – "Write Once, Run Anywhere" (WORA).**
2. **Object-Oriented – Follows OOP principles (Encapsulation, Inheritance, Polymorphism, and Abstraction).**
3. **Simple and Easy to Learn – Syntax is similar to C++ but removes complex features like pointers.**
4. **Secure – Provides built-in security features such as bytecode verification and exception handling.**
5. **Robust – Strong memory management and exception handling prevent crashes.**
6. **Multithreading – Supports concurrent execution of multiple tasks.**
7. **High Performance – Uses Just-In-Time (JIT) compiler to improve speed.**
8. **Rich API – Provides a vast set of built-in libraries for various functionalities.**

Java Architecture

- **Java Development Kit (JDK) –** Includes tools to write and compile Java programs.
- **Java Runtime Environment (JRE) –** Contains the libraries needed to run Java programs.
- **Java Virtual Machine (JVM) –** Converts Java bytecode into machine code for execution.

Basic Java Syntax

```
// Hello World Program in Java
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, World!");
   }
}
```

2.Overview of Java

Java is a high-level, object-oriented programming language developed by James Gosling at Sun Microsystems (now owned by Oracle) in 1995. It is widely used for developing web applications, mobile applications, enterprise software, and more. Java follows the "Write Once, Run Anywhere" (WORA) principle, meaning compiled Java code can run on any platform that supports Java without modification.

Key Features of Java

1. Platform-Independent

- Java code is compiled into bytecode, which runs on the Java Virtual Machine (JVM), making it independent of the operating system.

2.Object-Oriented

- Java follows object-oriented programming (OOP) concepts like Encapsulation, Inheritance, Polymorphism, and Abstraction.

3.Simple & Secure

- Java eliminates complex features like pointers and provides security mechanisms like the Java Security Manager and bytecode verification.

4.Robust & Reliable

- It includes strong memory management, exception handling, and automatic garbage collection.

5.Multithreading

- Java allows the execution of multiple threads simultaneously, improving performance.

6.Distributed Computing

- Java supports distributed applications using technologies like RMI (Remote Method Invocation) and CORBA.

7.Rich API & Libraries

- Java provides extensive libraries (Java Standard Edition, Java Enterprise Edition) for development.

8.Scalability & Performance

- Java can be used for both small applications and large-scale enterprise solutions.

Java Editions

Java comes in different editions suited for various applications:

1. **Java SE (Standard Edition) – Used for core Java development (desktop & simple applications).**
2. **Java EE (Enterprise Edition) – Used for web and enterprise-level applications.**
3. **Java ME (Micro Edition) – Used for mobile and embedded system applications.**
4. **JavaFX – Used for building rich internet applications.**

Java Development Tools

- **JDK (Java Development Kit) – Includes the compiler (javac), Java runtime (JRE), and development tools.**
- **JRE (Java Runtime Environment) – Includes JVM and libraries to run Java applications.**
- **JVM (Java Virtual Machine) – Executes Java bytecode on different platforms.**

Basic Java Syntax Example

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, Java!");
    }
}
```

Applications of Java

- **Web Development –** (Spring, Hibernate, Servlets, JSP)
- **Mobile Development –** (Android using Java)
- **Enterprise Applications –** (Banking, ERP, CRM)
- **Big Data & AI –** (Apache Hadoop, Spark)
- **Game Development –** (LibGDX, jMonkeyEngine)

3.Features of Java

Java is a powerful, object-oriented programming language known for its platform independence, security, and robust features. Here are the key features of Java:

1. Platform Independent

- Java follows the "Write Once, Run Anywhere" (WORA) principle.
- Java programs are compiled into bytecode, which runs on the Java Virtual Machine (JVM), making it independent of the underlying OS.

2. Object-Oriented

- Everything in Java is treated as an object.
- Follows OOP principles: Encapsulation, Inheritance, Polymorphism, and Abstraction.

3. Simple and Easy to Learn

- Java syntax is similar to C++, but with simpler memory management (automatic garbage collection).

4. Secure

Java programs run in a sandbox environment to prevent

- unauthorized access.
- Bytecode verification ensures no security breaches.

5. Robust

- Java has strong memory management with built-in exception handling and automatic garbage collection.

6. Multithreading Support

- Java supports multithreading, allowing multiple tasks to execute simultaneously, improving performance.

7. High Performance

- Java uses Just-In-Time (JIT) Compiler, which improves execution speed by compiling bytecode into native machine code at runtime.

8. Distributed Computing

- Java supports distributed computing using RMI (Remote Method Invocation) and CORBA.

9. Dynamic and Extensible

- Java supports dynamic class loading, meaning classes are loaded into memory when required.
- Uses native libraries like JNI (Java Native Interface) for extending functionality.

10. Automatic Memory Management

- Java has a built-in Garbage Collector that automatically manages memory allocation and deallocation.

11. Rich API

- Java provides a vast set of built-in libraries for networking, data structures, database connection, and GUI development.

12. Open Source and Community Support

- Java has a large developer community and is backed by Oracle.
- OpenJDK provides a free and open-source implementation.

4.Java Development Kit (JDK), Java Runtime Environment (JRE), Java Virtual Machine (JVM)

Java Development Kit (JDK), Java Runtime Environment (JRE), Java Virtual Machine (JVM)

Java is a platform-independent, object-oriented programming language that runs on the principle of "Write Once, Run Anywhere" (WORA). The execution of Java applications involves three key components: JDK, JRE, and JVM.

1. Java Virtual Machine (JVM)

JVM (Java Virtual Machine) is an abstract machine that provides a runtime environment for executing Java applications. It is responsible for converting Java bytecode into machine-specific code and executing it.

Key Functions of JVM:

- **Loads Java Bytecode: Converts .class files (bytecode) into native machine code.**
- **Manages Memory: Handles memory allocation and garbage collection.**
- **Ensures Security: Provides a secure execution environment.**
- **Enables Platform Independence: Java programs run on any OS with a compatible JVM.**

💡 JVM Components:

- Class Loader – Loads class files (.class) at runtime.
- Runtime Memory Areas:
  - Method Area
  - Heap
  - Stack
  - Program Counter Register
  - Native Method Stack
- Execution Engine:
  - Interpreter (executes bytecode line by line)
  - Just-In-Time (JIT) Compiler (compiles bytecode into native code for faster execution)
- Garbage Collector – Automatically manages memory by deallocating unused objects.

2. Java Runtime Environment (JRE)

JRE (Java Runtime Environment) is a package that provides the libraries and components required to run Java applications. It includes the JVM along with essential libraries and runtime resources.

JRE Components:

- JVM (Java Virtual Machine)
- Core Libraries (like Java API, Collections, IO, Networking)
- Deployment Technologies (Java Web Start, Java Plug-in)

💡 JRE does not include development tools like a compiler or debugger. It is meant only for running Java applications.

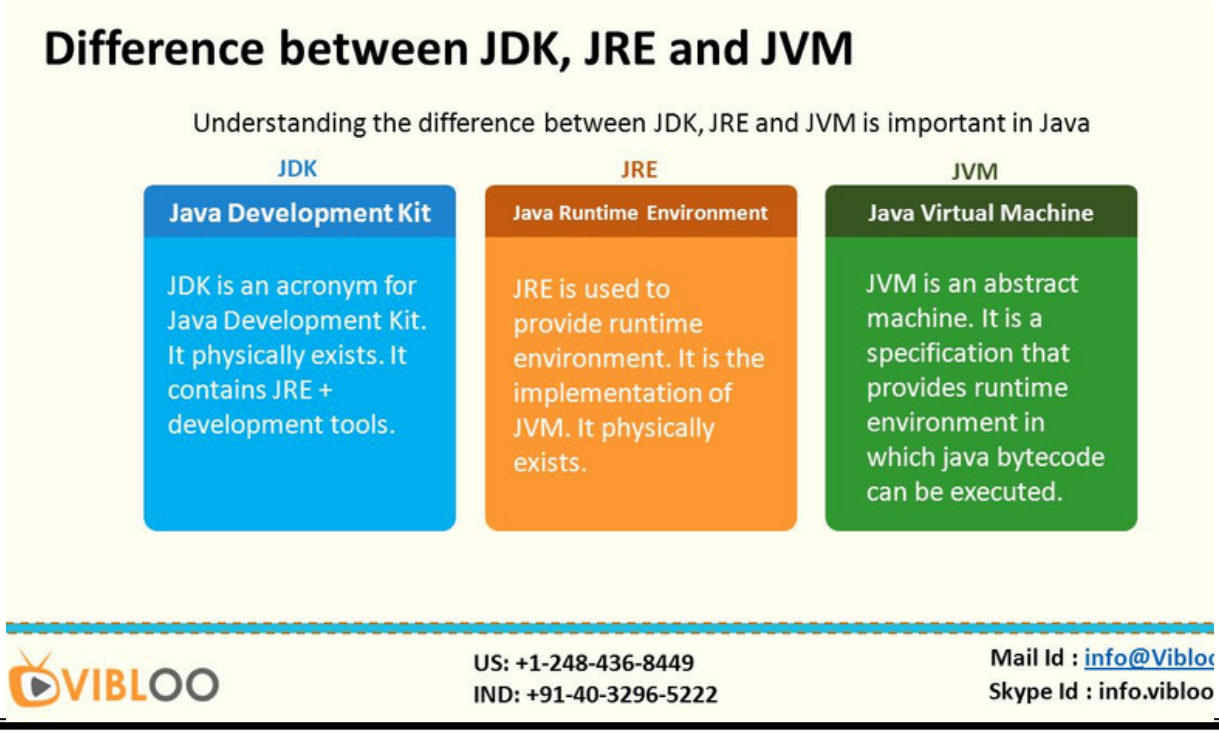
3. Java Development Kit (JDK)

JDK (Java Development Kit) is a complete software development package that includes the JRE, JVM, and additional development tools like the compiler (javac), debugger, and JavaDoc.

JDK Components:

- **JRE (Java Runtime Environment)**
- **JVM (Java Virtual Machine)**
- **Java Compiler (javac) – Converts Java source code into bytecode.**
- **Development Tools – Debugger (jdb), JavaDoc, JavaFX, and other utilities.**

💡 **JDK is required for Java development, whereas JRE is only needed to run Java applications.**



5.Setting up Java and IDE (Eclipse, IntelliJ IDEA, VS Code)

To set up Java and an IDE (Eclipse, IntelliJ IDEA, or VS Code), follow these steps:
Step 1: Install Java Development Kit (JDK)
Java is required to run and develop Java applications.

1. Download and Install JDK:

- **Visit the official Oracle JDK page: Oracle JDK**
- **Alternatively, you can install OpenJDK: OpenJDK**
- **Choose the latest version and install it based on your OS (Windows, Mac, or Linux).**

2. Verify Installation:

After installation, open a terminal (Command Prompt/PowerShell or Terminal) and type:
java -version
javac -version

3. Set Up Environment Variables (Windows Users Only):

- **Go to Control Panel → System → Advanced System Settings → Environment Variables.**
- **Under System Variables, add/edit:**
  - **JAVA_HOME → Set it to the JDK installation path (e.g., C:\Program Files\Java\jdk-17).**
  - **Path → Add C:\Program Files\Java\jdk-17\bin.**

Step 2: Install an IDE

Choose one of the following IDEs:

🔹 Eclipse IDE

1. Download and Install:

- Go to Eclipse Downloads.
- Download "Eclipse IDE for Java Developers".
- Install and launch Eclipse.

2. Set Up a Java Project in Eclipse:

- Open Eclipse and go to File → New → Java Project.
- Enter a project name and click Finish.
- Add a new class: Right-click src → New → Class.
- Write Java code and run it.

🔹 IntelliJ IDEA

1. Download and Install:

- Go to IntelliJ IDEA.
- Download the Community Edition (free) or Ultimate Edition (paid).
- Install and launch IntelliJ IDEA.

2. Create a Java Project in IntelliJ:

- Click New Project → Java.
- Select the installed JDK.
- Click Next → Finish.
- Create a new class in the src folder and write Java code.

🔹 Visual Studio Code (VS Code)

1. Install VS Code:

- Download from VS Code.
- Install it on your system.

2. Install Java Extensions:

- Open VS Code, go to Extensions (Ctrl+Shift+X).
- Install the following extensions:
  - Java Extension Pack (includes Java support).
  - Debugger for Java.

3. Create a Java Project:

- Open a new folder in VS Code.
- Create a new Java file (HelloWorld.java).
- Write your Java code and run it using the Run button or terminal.

5.Writing and running the first Java program

Writing and running your first Java program is simple. Follow these steps:

Step 1: Install Java

Ensure you have Java Development Kit (JDK) installed. You can download it from Oracle's website or use an open-source version like OpenJDK.
To check if Java is installed, run:
java -version
javac -version

Step 2: Write Your First Java Program

1. Open a text editor (like Notepad, VS Code, or IntelliJ IDEA).

2.Write the following Java program:

```
// HelloWorld.java
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, World!");
   }
}
```

Step 3: Compile the Java Program

Open a terminal or command prompt, navigate to the folder where you saved the file, and run:

```
javac HelloWorld.java
```

Step 4: Run the Java Program

Execute the compiled Java program using:

```
java HelloWorld
```

CHAPTER-2
Java Fundamentals

1.Data Types and Variables in Programming

1. Variables

A variable is a name given to a memory location that stores data. It acts as a container for storing values that can be changed during program execution.

2. Rules for Naming Variables

- **Must start with a letter (A-Z or a-z) or an underscore _**
- **Cannot start with a number**
- **Cannot use special characters or spaces**
- **Case-sensitive (age and Age are different variables)**
- **Cannot use reserved keywords (e.g., if, while, class)**

3. Data Types

A data type specifies the type of value a variable can hold. Programming languages support different types of data.

Primitive Data Types

These are basic data types available in most languages.

- **Integer (int) – Whole numbers (e.g., 10, -5, 1000)**
- **Float (float, double) – Decimal numbers (e.g., 3.14, -2.5)**
- **Character (char) – Single character (e.g., 'A', 'z', '5')**
- **String (str) – Sequence of characters (e.g., "Hello" or "Python")**
- **Boolean (bool) – Logical values (True or False)**

Non-Primitive Data Types

More complex types used to store multiple values.

- List/Array – Ordered collection of elements (e.g., [1, 2, 3])
- Tuple – Immutable ordered collection (e.g., (4, 5, 6))
- Dictionary (dict, hashmap) – Key-value pairs (e.g., {"name": "John", "age": 25})
- Set – Unordered collection of unique elements (e.g., {1, 2, 3})

4. Variable Declaration and Initialization

Python Example

```
# Variable declaration and assignment
age = 25        # Integer
pi = 3.14       # Float
name = "Alice"   # String
is_student = True  # Boolean
fruits = ["apple", "banana", "cherry"]  # List
```

Java Example

```
// Variable declaration and initialization
int age = 25;
double pi = 3.14;
String name = "Alice";
boolean isStudent = true;
int[] numbers = {1, 2, 3};  // Array
```

2.Operators in Java

Operators in Java are special symbols that perform operations on variables and values. Java supports different types of operators:

1. Arithmetic Operators

Used for mathematical operations.

Operators	Meaning	Example	Result
+	Addition	4+2	6
-	Subtraction	4-2	2
*	Multiplication	4*2	8
/	Division	4/2	2
%	Modulus operator to get remainder in integer division	5%2	1
+ +	Increment	A = 10; A+ +	11
− −	Decrement	A = 10; A− −	9

2.Relational (Comparison) Operators

==	Equal to	a==b returns 1 if a and b are the same
>	Greater than	a>b returns 1 if a is larger than b
<	Less than	a<b returns 1 if a is smaller than b
>=	Greater than or equal to	a>=b returns 1 if a is larger than or equal to b
<=	Less than or equal to	a<=b returns 1 if a is smaller than or equal to b
!=	Not equal to	a!=b returns 1 if a and b not the same

3. Logical Operators

Used to perform logical operations.

Logical Operator	Logical Symbol
and	$\wedge$
or	$\vee$
exclusive or (*XOR*)	$\oplus$
if … then	$\rightarrow$

4.Bitwise Operators

Used to perform bitwise operations on integers.

Operators	Description	Use
&	Bitwise AND	op1 & op2
|	Bitwise OR	op1 | op2
^	Bitwise Exclusive OR	op1 ^ op2
~	Bitwise Complement	~op
<<	Bitwise Shift Left	op1 << op2
>>	Bitwise Shift Right	op1 >> op2
>>>	Bitwise Shift Right zero fill	op1 >>> op2

5. Assignment Operators

Used to assign values to variables.

Operator	Example	Equivalent Expression (m=15)	Result
+=	m +=10	m = m+10	25
-=	m -=10	m = m-10	5
*=	m *=10	m = m*10	150
/=	m /=	m = m/10	1
%=	m %=10	m = m%10	5

6. Unary Operators

Operate on a single operand.

Operator	Meaning	Example
+	Add two operands or unary plus	x + y
-	Subtract right operand from the left or unary minus	x - y
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	

3.Input and Output (Scanner, BufferedReader)

In Java, we can take input from the user using different methods, including Scanner and BufferedReader. Let's explore both:

1. Using Scanner

The Scanner class is part of java.util and provides a simple way to read input.

Example: Using Scanner

```
import java.util.Scanner;

public class ScannerExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Reading different types of inputs
        System.out.print("Enter an integer: ");
```

```
int num = scanner.nextInt();

 System.out.print("Enter a double: ");
 double decimal = scanner.nextDouble();

 scanner.nextLine(); // Consume the newline left-over

 System.out.print("Enter a string: ");
 String text = scanner.nextLine();

 System.out.println("\nYou entered:");
 System.out.println("Integer: " + num);
 System.out.println("Double: " + decimal);
 System.out.println("String: " + text);

 scanner.close(); // Always close the scanner
 }
}
```

Advantages:

- Easy to use for different data types.
- Supports methods like nextInt(), nextDouble(), nextLine(), etc.

Disadvantages:

- Slightly slower compared to BufferedReader.
- nextLine() issue after nextInt() or nextDouble() (requires extra scanner.nextLine()).

2. Using BufferedReader

The BufferedReader class is part of java.io and provides faster input reading, but it requires additional parsing.

Example: Using BufferedReader

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class BufferedReaderExample {
   public static void main(String[] args) throws IOException {
      BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));

      // Reading input as a string and converting it
      System.out.print("Enter an integer: ");
      int num = Integer.parseInt(reader.readLine());

      System.out.print("Enter a double: ");
      double decimal = Double.parseDouble(reader.readLine());

      System.out.print("Enter a string: ");
      String text = reader.readLine();
      System.out.println("\nYou entered:");
      System.out.println("Integer: " + num);
      System.out.println("Double: " + decimal);
      System.out.println("String: " + text);
```

Advantages:

- Faster than Scanner, especially for large inputs.
- Avoids nextLine() issues.

Disadvantages:

- Requires explicit parsing (Integer.parseInt(), Double.parseDouble()).
- Slightly more complex to use.

4.Control Statements (if-else, switch-case)

Control statements in programming allow you to make decisions and execute different code blocks based on conditions. The two main control statements are:

1. if-else Statement

The if-else statement is used to execute a block of code if a condition is true; otherwise, it executes another block of code.

Syntax:

```
if (condition) {
   // Code to execute if condition is true
} else {
   // Code to execute if condition is false
}
```

2. switch-case Statement

The switch statement is used when a variable needs to be compared against multiple possible values.

Syntax:

```
switch (expression) {
   case value1:
      // Code to execute if expression == value1
      break;
   case value2:
      // Code to execute if expression == value2
      break;
   ...
   default:
      // Code to execute if no cases match
}
```

5.Loops (for, while, do-while)

Loops are used in programming to execute a block of code multiple times. The main types of loops are:

1. For Loop

Used when the number of iterations is known beforehand.

Syntax (C, Java, JavaScript, etc.):

```
for(initialization; condition; increment/decrement) {
   // Code to execute
}
```

Example:

```
for(int i = 1; i <= 5; i++) {
   printf("%d\n", i);
}
```

2. While Loop

Used when the number of iterations is not known in advance. The loop continues as long as the condition is true.

Syntax:

```
while(condition) {
   // Code to execute
```

Example:

```
int i = 1;
while(i <= 5) {
   printf("%d\n", i);
   i++;
```

3. Do-While Loop

Similar to the while loop, but executes at least once before checking the condition.

Syntax:

```
do {
   // Code to execute
} while(condition);
```

Example:

```
int i = 1;
do {
   printf("%d\n", i);
   i++;
} while(i <= 5);
```

6.Break and Continue Statements

In Java, break and continue are control flow statements that help manage loops (for, while, do-while) and switch cases.

1. break Statement

The break statement is used to terminate the loop or switch statement immediately and exit the block.

Syntax:

```
break;
```

Example: Using break in a loop

```
public class BreakExample {
   public static void main(String[] args) {
      for (int i = 1; i <= 10; i++) {
         if (i == 5) {
            break; // Loop terminates when i == 5
         }
         System.out.println(i);
      }
   }
}
```

2. continue Statement

The continue statement skips the current iteration and moves to the next iteration of the loop.

Syntax:

```
continue;
```

6.Break and Continue Statements

Break and Continue Statements in Java
In Java, break and continue are control flow statements used inside loops (for, while, do-while) and switch statements to modify the normal execution flow.

1. Break Statement

The break statement is used to exit a loop or switch statement immediately. When encountered inside a loop, it stops further execution of the loop and control jumps to the statement immediately after the loop.

Syntax:

```
break;
```

Example: Break in a Loop

```
public class BreakExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            if (i == 3) {
                break; // Loop stops when i == 3
            }
            System.out.println("i: " + i);
        }
        System.out.println("Loop terminated");
    }
}
```

2. Continue Statement

The continue statement is used to skip the current iteration of the loop and move to the next iteration. It does not terminate the loop but instead skips the rest of the code in the current iteration.

Syntax:

```
continue;
```

Example: Continue in a Loop

```
public class ContinueExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            if (i == 3) {
                continue; // Skips iteration when i == 3
            }
            System.out.println("i: " + i);
        }
        System.out.println("Loop completed");
    }
}
```

3. Break vs. Continue: Key Differences

3. Break vs. Continue: Key Differences

Sl.No.	break	continue
1.	Used to terminate the loops or to exit loop from a switch.	Used to transfer the control to the start of loop.
2.	The break statement when executed causes immediate termination of loop containing it.	Continue statement when executed causes Immediate termination of the current iteration of the loop.

4. Nested Loops with Break and Continue

Break in Nested Loops

```
public class NestedBreak {
    public static void main(String[] args) {
        for (int i = 1; i <= 3; i++) {
            for (int j = 1; j <= 3; j++) {
                if (j == 2) {
                    break; // Exits inner loop when j == 2
                }
                System.out.println("i: " + i + ", j: " + j);
            }
        }
    }
}
```

Continue in Nested Loops

```
public class NestedContinue {
   public static void main(String[] args) {
      for (int i = 1; i <= 3; i++) {
         for (int j = 1; j <= 3; j++) {
            if (j == 2) {
               continue; // Skips when j == 2
            }
            System.out.println("i: " + i + ", j: " + j);
         }
      }
   }
}
```

5. Labeled Break and Continue

Java provides labeled break and continue to control flow in nested loops.

Labeled Break Example

```
public class LabeledBreak {
   public static void main(String[] args) {
      outer: for (int i = 1; i <= 3; i++) {
         for (int j = 1; j <= 3; j++) {
            if (j == 2) {
               break outer; // Exits both loops
            }
```

```
 System.out.println("i: " + i + ", j: " + j);
 }
 }
 }
}
```

Here, break outer; exits both the inner and outer loops.

Labeled Continue Example

```
public class LabeledContinue {
   public static void main(String[] args) {
      outer: for (int i = 1; i <= 3; i++) {
         for (int j = 1; j <= 3; j++) {
            if (j == 2) {
               continue outer; // Skips to the next iteration of outer
loop
            }
            System.out.println("i: " + i + ", j: " + j);
         }
      }
   }
}
```

CHAPTER -3
Object-Oriented Programming (OOP) in Java

1.Classes and Objects

Classes and Objects in Object-Oriented Programming (OOP)
In object-oriented programming (OOP), classes and objects are fundamental concepts.

1. What is a Class?

A class is a blueprint for creating objects. It defines attributes (data members) and methods (functions) that describe the behavior of the objects.

Syntax:

```
class ClassName:
    # Constructor (Initializer)
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1  # Instance variable
        self.attribute2 = attribute2

    # Method (Function inside a class)
    def display(self):
        print(f"Attribute 1: {self.attribute1}, Attribute 2: {self.attribute2}")
```

2. What is an Object?

An object is an instance of a class. It has specific values assigned to its attributes.

Example:

```
# Creating a class
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def display_info(self):
        print(f"Car: {self.brand} {self.model}, Year: {self.year}")

# Creating objects
car1 = Car("Toyota", "Corolla", 2022)
car2 = Car("Honda", "Civic", 2023)

# Calling methods
car1.display_info()  # Output: Car: Toyota Corolla, Year: 2022
car2.display_info()  # Output: Car: Honda Civic, Year: 2023
```

3. More Examples

Example: Bank Account Class

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"{amount} deposited. New balance: {self.balance}")
    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient funds!")
        else:
            self.balance -= amount
            print(f"{amount} withdrawn. Remaining balance:
{self.balance}")

# Creating an account object
account = BankAccount("Santhosh", 5000)

# Performing transactions
account.deposit(2000)  # Output: 2000 deposited. New
balance: 7000
account.withdraw(3000)  # Output: 3000 withdrawn.
Remaining balance: 4000
account.withdraw(5000)  # Output: Insufficient funds!
```

Constructors and Methods

1. Constructors in Java

A constructor is a special method in Java that is used to initialize objects. It is called when an object of a class is created.

Characteristics of a Constructor:

- **It has the same name as the class.**
- **It does not have a return type (not even void).**
- **It is automatically called when an object is created.**
- **It can be overloaded (multiple constructors in the same class with different parameters).**

Types of Constructors:

1. **Default Constructor (No-Argument Constructor)**
2. **If no constructor is defined, Java provides a default constructor.**

```
class Student {
    String name;

    // Default constructor
    Student() {
        System.out.println("Default Constructor Called!");
        name = "Unknown";
    }

    void display() {
        System.out.println("Name: " + name);
    }
```

```
}

public class Main {
 public static void main(String[] args) {
 Student s1 = new Student(); // Calls default constructor
 s1.display();
 }
}
```

2.Parameterized Constructor

A constructor with parameters is used to initialize objects with specific values.

```
class Student {
   String name;

   // Parameterized constructor
   Student(String studentName) {
      name = studentName;
   }

   void display() {
      System.out.println("Name: " + name);
   }
}

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

```
 Student s1 = new Student("Hari"); // Calls parameterized
constructor
 s1.display();
 }
}
```

3.Copy Constructor

Java does not have a built-in copy constructor, but it can be created manually.

```
class Student {
   String name;

   // Parameterized constructor
   Student(String studentName) {
      name = studentName;
   }

   // Copy constructor
   Student(Student s) {
      name = s.name;
   }

   void display() {
      System.out.println("Name: " + name);
   }
}
```

```
public class Main {
 public static void main(String[] args) {
 Student s1 = new Student("Hari");
 Student s2 = new Student(s1); // Copy constructor

 s1.display();
 s2.display();
 }
}
```

2. Methods in Java

A method is a block of code that performs a specific task. It helps in code reusability.

Types of Methods:

1. Instance Methods
2. These methods belong to an instance of the class and can access instance variables.

```
class Calculator {
   int add(int a, int b) {
      return a + b;
   }
}
public class Main {
   public static void main(String[] args) {
      Calculator calc = new Calculator();
      int sum = calc.add(5, 10);
```

```
System.out.println("Sum: " + sum);
 }
}
```

1.Static Methods

These methods belong to the class and do not require an instance to be called.

```
class Calculator {
   static int multiply(int a, int b) {
      return a * b;
   }
}

public class Main {
   public static void main(String[] args) {
      int result = Calculator.multiply(5, 10);  // Calling without an object
      System.out.println("Product: " + result);
   }
}
```

2.Method Overloading

Methods with the same name but different parameters.

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

```
 double add(double a, double b) {
 return a + b;
 }
}

public class Main {
 public static void main(String[] args) {
 MathOperations obj = new MathOperations();
 System.out.println(obj.add(5, 10)); // Calls int version
 System.out.println(obj.add(5.5, 10.5)); // Calls double version
 }
```

Method Overriding

When a subclass provides a specific implementation of a method from the superclass.

```
class Animal {
   void sound() {
      System.out.println("Animal makes a sound");
   }
}

class Dog extends Animal {
   @Override
   void sound() {
      System.out.println("Dog barks");
   }
}
```

```
public class Main {
 public static void main(String[] args) {
 Animal myDog = new Dog();
 myDog.sound(); // Calls overridden method
 }
}
```

3.Method Overloading and Overriding

Both Method Overloading and Method Overriding are concepts of Polymorphism in Java, but they differ in how they are implemented.

1. Method Overloading (Compile-time Polymorphism)

Method Overloading occurs when multiple methods in the same class have the same name but different parameters (different type, number, or order of parameters).

Key Points:

- Happens within the same class.
- Methods have the same name but different parameter lists.
- Return type may or may not be different.
- It is a type of compile-time polymorphism (decision made at compile time).

Example of Method Overloading:

```
class Calculator {
   // Method with two int parameters
   int add(int a, int b) {
      return a + b;
   }

   // Method with three int parameters
   int add(int a, int b, int c) {
      return a + b + c;
   }

   // Method with two double parameters
   double add(double a, double b) {
      return a + b;
   }

   public static void main(String[] args) {
      Calculator calc = new Calculator();
      System.out.println(calc.add(5, 10));       // Calls add(int, int)
      System.out.println(calc.add(5, 10, 15));    // Calls add(int,
int, int)
      System.out.println(calc.add(5.5, 10.5));    // Calls
add(double, double)
   }
}
```

2. Method Overriding (Runtime Polymorphism)

Method Overriding occurs when a subclass (child class) provides a specific implementation of a method that is already defined in its superclass (parent class).

Key Points:

- **Happens in an inheritance relationship (parent-child class).**
- **Method name, parameters, and return type must be exactly the same.**
- **The overridden method in the subclass must have the same signature as the method in the parent class.**
- **It is a type of runtime polymorphism (decision made at runtime).**
- **The @Override annotation is used for better readability and to avoid mistakes.**
- **Access specifier of the overridden method in the subclass cannot be more restrictive than in the superclass.**

Example of Method Overriding:

```
class Animal {
   // Parent class method
   void makeSound() {
      System.out.println("Animal makes a sound");
   }
}
```

```
class Dog extends Animal {
 // Overriding the makeSound() method
 @Override
 void makeSound() {
 System.out.println("Dog barks");
 }
}

class Main {
 public static void main(String[] args) {
 Animal myAnimal = new Animal(); // Parent class object
 myAnimal.makeSound(); // Calls parent class method

 Animal myDog = new Dog(); // Parent class reference, child
class object
 myDog.makeSound(); // Calls child class method (Overriding)
 }
}
```

4.Access Modifiers (public, private, protected, default)

Access Modifiers in Java

Access modifiers in Java determine the visibility and accessibility of classes, methods, and variables. Java provides four main types of access modifiers:

1. Public

- **Keyword: public**
- **Access Level: Accessible from anywhere (inside and outside the package).**
- **Usage: Used when a class, method, or variable should be accessible from anywhere.**

Example:

```
public class Example {
   public int number = 10; // Public variable

   public void display() { // Public method
      System.out.println("This is a public method.");
   }
}
```

2. Private

- **Keyword: private**
- **Access Level: Accessible only within the same class.**
- **Usage: Used for data encapsulation, restricting access to variables and methods.**

Example:

```
class Example {
   private int number = 10; // Private variable

   private void display() { // Private method
      System.out.println("This is a private method.");
   }
}
```

3. Protected

- Keyword: protected
- Access Level: Accessible within the same package and in subclasses (even if they are in different packages).
- Usage: Used for inheritance, allowing child classes to access parent class members.

Example:

```
package package1;

class Parent {
   protected int number = 10; // Protected variable

   protected void display() { // Protected method
      System.out.println("This is a protected method.");
   }
}
```

```
package package2;
import package1.Parent;

class Child extends Parent {
 void show() {
 System.out.println(number); // ✅ Accessible due to
inheritance
 display(); // ✅ Accessible due to inheritance
 }
}
```

4. Default (No Modifier)

- **Keyword:** (No keyword specified)
- **Access Level:** Accessible within the same package only.
- **Usage:** Used when you want to restrict access to the package level but not beyond.

Example:

```
class Example {
   int number = 10; // Default variable

   void display() { // Default method
      System.out.println("This is a default method.");
   }
}
```

4.Static and Instance Variables & Methods

In Java, variables and methods are categorized as either static or instance, depending on how they are declared and accessed.

1. Static Variables and Methods

Static Variables (Class Variables)

- Declared using the static keyword inside a class but outside any method or constructor.
- Shared among all instances of the class.
- Stored in the Method Area of memory.
- Belong to the class rather than any specific instance.

Example of a Static Variable

```
class Employee {
   static String companyName = "Tech Solutions"; // Static
Variable
}
```

Static Methods (Class Methods)

- Declared using the static keyword.
- Can be called without creating an instance of the class.
- Can only access static variables directly (not instance variables).
- Cannot use this or super (because they refer to instance-level data).

Example of a Static Method

```
class Employee {
    static String companyName = "Tech Solutions";

    static void displayCompany() {
        System.out.println("Company Name: " + companyName);
    }
}

public class Main {
    public static void main(String[] args) {
        Employee.displayCompany(); // Calling static method
without object
    }
}
```

2. Instance Variables and Methods

Instance Variables

- Declared without the static keyword.
- Each object of the class has its own copy.
- Stored in the Heap Memory.
- Belong to an instance of the class.

Example of an Instance Variable

```
class Employee {
   String name; // Instance Variable
   int age;
}
```

Instance Methods

- Do not have the static keyword.
- Can access both instance and static variables.
- Need an object to be invoked.

Example of an Instance Method

```
class Employee {
   String name;
   int age;
      void displayInfo() { // Instance Method
      System.out.println("Name: " + name + ", Age: " + age);
   }
}
public class Main {
   public static void main(String[] args) {
      Employee emp1 = new Employee();
      emp1.name = "Santhosh";
      emp1.age = 22;
      emp1.displayInfo(); // Calling instance method using object
   }
}
```

6.Encapsulation

What is Encapsulation?

Encapsulation is one of the four fundamental Object-Oriented Programming (OOP) principles in Java, along with Abstraction, Inheritance, and Polymorphism. It is the process of wrapping data (variables) and code (methods) together into a single unit, typically a class.

Encapsulation ensures that the internal details of a class are hidden from the outside world, and access to the data is controlled through getter and setter methods.

Key Points of Encapsulation

1. **Data Hiding: Restricts direct access to class fields, protecting data from accidental modification.**
2. **Increased Security: Prevents unauthorized access and modification of data.**
3. **Modular Code: Makes it easier to modify and maintain code without affecting other parts of the program.**
4. **Improved Readability and Flexibility: Enhances code reusability by providing controlled access.**
5.

How to Achieve Encapsulation in Java?

Encapsulation is implemented in Java using the following steps:

1. **Declare variables as private (to restrict direct access).**
2. **Provide public getter and setter methods to access and modify the variables.**

Example of Encapsulation in Java

```
// Encapsulation Example in Java
class Student {
   // Private data members (data hiding)
   private String name;
   private int age;

   // Public getter method to access the private variable
   public String getName() {
      return name;
   }

   // Public setter method to modify the private variable
   public void setName(String newName) {
      this.name = newName;
   }

   public int getAge() {
      return age;
   }

   public void setAge(int newAge) {
      if (newAge > 0) { // Ensuring valid data
         this.age = newAge;
      } else {
         System.out.println("Age cannot be negative!");
      }
```

```
 }
}

public class EncapsulationDemo {
 public static void main(String[] args) {
 // Creating an object of Student class
 Student s = new Student();

 // Setting values using setter methods
 s.setName("Hari");
 s.setAge(20);

 // Getting values using getter methods
 System.out.println("Student Name: " + s.getName());
 System.out.println("Student Age: " + s.getAge());
 }
}
```

7.Inheritance (Single, Multilevel, Hierarchical)

Inheritance in Java

Inheritance is one of the fundamental concepts of Object-Oriented Programming (OOP) in Java. It allows a class (child/subclass) to inherit properties and behavior (fields and methods) from another class (parent/superclass). This helps in code reusability, reducing redundancy, and improving maintainability.

Types of Inheritance in Java

Java supports three main types of inheritance:

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance

1. Single Inheritance

In single inheritance, a subclass inherits from only one superclass.

Example of Single Inheritance:

```
// Parent Class
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

// Child Class (Inheriting from Animal)
class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}

// Main Class
public class SingleInheritance {
```

```
 public static void main(String[] args) {
 Dog myDog = new Dog();
 myDog.eat(); // Inherited method from Animal
 myDog.bark(); // Method from Dog class
 }
}
```

2. Multilevel Inheritance

In multilevel inheritance, a class inherits from another class, and then another class inherits from it, forming a chain.

Example of Multilevel Inheritance:

```
// Grandparent Class
class Animal {
   void eat() {
      System.out.println("This animal eats food.");
   }
}

// Parent Class (Inheriting from Animal)
class Dog extends Animal {
   void bark() {
      System.out.println("The dog barks.");
   }
}

// Child Class (Inheriting from Dog)
```

```
class Puppy extends Dog {
 void weep() {
 System.out.println("The puppy weeps.");
 }
}

// Main Class
public class MultilevelInheritance {
 public static void main(String[] args) {
 Puppy myPuppy = new Puppy();
 myPuppy.eat(); // Inherited from Animal
 myPuppy.bark(); // Inherited from Dog
 myPuppy.weep(); // Defined in Puppy
 }
}
```

3. Hierarchical Inheritance

In hierarchical inheritance, multiple child classes inherit from the same parent class.

Example of Hierarchical Inheritance:

```
// Parent Class
class Animal {
   void eat() {
      System.out.println("This animal eats food.");
   }
}
```

```
// Child Class 1 (Inheriting from Animal)
class Dog extends Animal {
 void bark() {
 System.out.println("The dog barks.");
 }
}

// Child Class 2 (Inheriting from Animal)
class Cat extends Animal {
 void meow() {
 System.out.println("The cat meows.");
 }
}

// Main Class
public class HierarchicalInheritance {
 public static void main(String[] args) {
 Dog myDog = new Dog();
 myDog.eat(); // Inherited from Animal
 myDog.bark(); // Defined in Dog

 Cat myCat = new Cat();
 myCat.eat(); // Inherited from Animal
 myCat.meow(); // Defined in Cat
 }
}
```

Polymorphism (Compile-time and Run-time)

Polymorphism in Java allows objects to be treated as instances of their parent class while executing different behaviors based on their actual type. There are two types of polymorphism in Java:

1. **Compile-time Polymorphism (Method Overloading)**
2. **Run-time Polymorphism (Method Overriding)**

1. Compile-time Polymorphism (Method Overloading)

- **Achieved through method overloading, where multiple methods in the same class have the same name but different parameters.**
- **The method to be executed is determined at compile-time based on the method signature.**

Example:

```
class Calculator {
   // Method with two int parameters
   int add(int a, int b) {
      return a + b;
   }

   // Method with three int parameters
   int add(int a, int b, int c) {
      return a + b + c;
   }
```

```
 // Method with two double parameters
 double add(double a, double b) {
 return a + b;
 }
}

public class Main {
 public static void main(String[] args) {
 Calculator calc = new Calculator();
 System.out.println(calc.add(5, 10)); // Calls add(int, int)
 System.out.println(calc.add(5, 10, 15)); // Calls add(int, int, int)
 System.out.println(calc.add(5.5, 10.5)); // Calls add(double,
double)
 }
}
```

2. Run-time Polymorphism (Method Overriding)

- Achieved through method overriding, where a subclass provides a specific implementation of a method already defined in its superclass.
- The method to be executed is determined at runtime based on the actual object type (dynamic method dispatch).

Example:

```
class Animal {
   void makeSound() {
      System.out.println("Animal makes a sound");
```

```
 }
}

class Dog extends Animal {
 @Override
 void makeSound() {
 System.out.println("Dog barks");
 }
}

class Cat extends Animal {
 @Override
 void makeSound() {
 System.out.println("Cat meows");
 }
}
public class Main {
 public static void main(String[] args) {
 Animal myAnimal; // Reference of type Animal

 myAnimal = new Dog();
 myAnimal.makeSound(); // Calls Dog's makeSound() method

 myAnimal = new Cat();
 myAnimal.makeSound(); // Calls Cat's makeSound() method
 }
}
```

Differences Between Compile-time and Run-time Polymorphism

Compile time Polymorphism	Run time Polymorphism
Call is resolved by the compiler.	Call is not resolved by the compiler.
Also known as Static binding or Early binding or overloading.	Also known as Dynamic binding or Late binding or overriding.
Overloading is compile time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	Overriding is run time polymorphism having same method with same parameters or signature, but associated in a class & its subclass.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution because known early at compile time.	It provides slow execution as compare to early binding because it is known at runtime.
It is less flexible as all things execute at compile time.	It is more flexible as all things execute at run time.

7.Abstraction (Abstract Classes, Interfaces)

Abstraction in Java (Abstract Classes & Interfaces)
Abstraction is one of the four pillars of Object-Oriented Programming (OOP), which helps hide implementation details and only exposes essential functionalities. In Java, abstraction can be achieved using:

1. **Abstract Classes**
2. **Interfaces**

1. Abstract Classes in Java

An abstract class is a class that cannot be instantiated and is

meant to be inherited by other classes. It can contain both abstract methods (without a body) and concrete methods (with implementations).

Syntax: Abstract Class

```
// Abstract class
abstract class Vehicle {
    // Abstract method (no implementation)
    abstract void start();
    // Concrete method (with implementation)
    void stop() {
        System.out.println("Vehicle stopped.");
    }
}
// Concrete subclass
class Car extends Vehicle {
    // Providing implementation for the abstract method
    void start() {
        System.out.println("Car started with a key.");
    }
}
public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        myCar.start(); // Output: Car started with a key.
        myCar.stop();  // Output: Vehicle stopped.
    }
```

2. Interfaces in Java

An interface is a blueprint of a class that only contains abstract methods (Java 7 and earlier). Since Java 8, it can have default and static methods as well.

Syntax: Interface

```
// Interface
interface Animal {
   void makeSound(); // Abstract method (by default in
interfaces)
}

// Implementing the interface
class Dog implements Animal {
   public void makeSound() {
      System.out.println("Dog barks.");
   }
}

public class Main {
   public static void main(String[] args) {
      Animal myDog = new Dog();
      myDog.makeSound(); // Output: Dog barks.
   }
}
```

Differences: Abstract Class vs. Interface

Abstract Class	Interface
1. Abstract class contains both **DECLARATION & DEFINITION** of methods.	Mostly Interfaces contain **DECLARATION** of methods. From C# 8.0 definition is also possible.
2. Abstract class keyword: **ABSTRACT**	2. Interface keyword: **INTERFACE**
3. Abstract class does not support **multiple inheritance**	3. Interface supports **multiple inheritance.**
4. Abstract class can have **constructors**.	4. Interface do not have constructors.

Example: Using Both Abstract Class & Interface

```
// Abstract class
abstract class Vehicle {
   abstract void start();
   void stop() {
      System.out.println("Vehicle stopped.");
   }
}

// Interface
interface Electric {
   void chargeBattery();
}
```

```
// Concrete class implementing both
class ElectricCar extends Vehicle implements Electric {
 void start() {
 System.out.println("Electric Car started silently.");
 }

 public void chargeBattery() {
 System.out.println("Battery is charging...");
 }
}

public class Main {
 public static void main(String[] args) {
 ElectricCar myCar = new ElectricCar();
 myCar.start();
 myCar.chargeBattery();
 myCar.stop();
 }
}
```

Output

```
Electric Car started silently.
Battery is charging...
Vehicle stopped.
```

CHAPTER -4
Arrays and Strings

1.One-Dimensional and Multi-Dimensional Arrays

1. One-Dimensional Arrays

A one-dimensional array is a linear collection of elements of the same data type, stored in contiguous memory locations. It is declared using square brackets ([]).

Declaration and Initialization:

```
// Declaration
int[] arr;

// Memory Allocation
arr = new int[5];

// Initialization
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
arr[3] = 40;
arr[4] = 50;
```

Alternative Initialization:

```
int[] arr = {10, 20, 30, 40, 50};
```

Accessing Elements:

```
System.out.println(arr[2]); // Output: 30
```

Iterating Over an Array:

```
for (int i = 0; i < arr.length; i++) {
   System.out.println(arr[i]);
}
```

Using an enhanced for loop:

```
for (int num : arr) {
   System.out.println(num);
}
```

2. Multi-Dimensional Arrays

A multi-dimensional array is an array of arrays, allowing for matrix-like structures.

2D Array Declaration and Initialization:

```
// Declaration
int[][] matrix;

// Memory Allocation
matrix = new int[3][3]; // 3 rows and 3 columns

// Initialization
matrix[0][0] = 1;
matrix[0][1] = 2;
matrix[0][2] = 3;
```

```
matrix[1][0] = 4;
matrix[1][1] = 5;
matrix[1][2] = 6;
matrix[2][0] = 7;
matrix[2][1] = 8;
matrix[2][2] = 9;
```

Alternative Initialization:

```
int[][] matrix = {
   {1, 2, 3},
   {4, 5, 6},
   {7, 8, 9}
};
```

Accessing Elements:

```
System.out.println(matrix[1][2]); // Output: 6
```

Iterating Over a 2D Array:

```
for (int i = 0; i < matrix.length; i++) {
   for (int j = 0; j < matrix[i].length; j++) {
      System.out.print(matrix[i][j] + " ");
   }
   System.out.println();
}
```

Using an enhanced for loop:

```
for (int[] row : matrix) {
    for (int num : row) {
        System.out.print(num + " ");
    }
    System.out.println();
}
```

3. Jagged Arrays (Irregular Multi-Dimensional Arrays)

A jagged array is a multi-dimensional array where sub-arrays can have different lengths.

Declaration and Initialization:

```
int[][] jaggedArr = new int[3][];
jaggedArr[0] = new int[]{1, 2};
jaggedArr[1] = new int[]{3, 4, 5};
jaggedArr[2] = new int[]{6, 7, 8, 9};
```

Iterating Over a Jagged Array:

```
for (int i = 0; i < jaggedArr.length; i++) {
    for (int j = 0; j < jaggedArr[i].length; j++) {
        System.out.print(jaggedArr[i][j] + " ");
    }
    System.out.println();
}
```

2.Array Operations (Sorting, Searching, Copying)

Array operations like sorting, searching, and copying are fundamental in programming. Here's an overview of each operation with examples in Python and Java:

Sorting

Sorting is used to arrange elements in ascending or descending order.

Python Sorting

- Using sorted() (returns a new sorted list)
- Using .sort() (modifies the list in-place)

```
arr = [5, 2, 9, 1, 5, 6]

# Using sorted() (does not modify the original list)
sorted_arr = sorted(arr)
print(sorted_arr)  # Output: [1, 2, 5, 5, 6, 9]

# Using sort() (modifies the original list)
arr.sort()
print(arr)  # Output: [1, 2, 5, 5, 6, 9]
```

Java Sorting

- Using Arrays.sort()

```
import java.util.Arrays;

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

```
 int[] arr = {5, 2, 9, 1, 5, 6};
 Arrays.sort(arr);
 System.out.println(Arrays.toString(arr)); // Output: [1, 2, 5, 5, 6,
9]
 }
}
```

Searching

Searching is used to find an element's index in an array.

Python Searching

- Linear Search (O(n))
- Binary Search (O(log n), requires sorted array)

```
arr = [10, 20, 30, 40, 50]

# Linear Search
def linear_search(arr, target):
   for i in range(len(arr)):
      if arr[i] == target:
         return i
   return -1
print(linear_search(arr, 30))  # Output: 2
# Binary Search using bisect module
import bisect

arr.sort()
index = bisect.bisect_left(arr, 30)
print(index)  # Output: 2
```

Java Searching

- Linear Search
- Binary Search (using Arrays.binarySearch())

```
import java.util.Arrays;

public class SearchingExample {
   public static void main(String[] args) {
      int[] arr = {10, 20, 30, 40, 50};

      // Linear Search
      int target = 30;
      int index = -1;
      for (int i = 0; i < arr.length; i++) {
         if (arr[i] == target) {
            index = i;
            break;
         }
      }
      System.out.println("Linear Search Index: " + index); //
Output: 2

      // Binary Search
      Arrays.sort(arr);
      int binaryIndex = Arrays.binarySearch(arr, 30);
      System.out.println("Binary Search Index: " + binaryIndex); //
Output: 2
   }
```

Copying

Copying is used to duplicate an array.

Python Copying

- Using copy()
- Using slicing [:]
- Using list()

```
arr = [1, 2, 3, 4, 5]

copy1 = arr.copy()
copy2 = arr[:]
copy3 = list(arr)

print(copy1)  # Output: [1, 2, 3, 4, 5]
```

Java Copying

- Using Arrays.copyOf()
- Using clone()

```
import java.util.Arrays;

public class CopyingExample {
   public static void main(String[] args) {
      int[] arr = {1, 2, 3, 4, 5};

      // Using clone()
      int[] copy1 = arr.clone();

      // Using Arrays.copyOf()
```

```
int[] copy2 = Arrays.copyOf(arr, arr.length);

 System.out.println(Arrays.toString(copy1)); // Output: [1, 2, 3, 4, 5]
 System.out.println(Arrays.toString(copy2)); // Output: [1, 2, 3, 4, 5]
 }
}
```

3.Strings in Java (String, StringBuffer, StringBuilder)

In Java, strings can be handled using three main classes:

1. String (Immutable)
2. StringBuffer (Mutable & Thread-safe)
3. StringBuilder (Mutable & Not Thread-safe)

1. String (Immutable)

- Definition: A String is an immutable sequence of characters.
- Why Immutable? Any modification creates a new String object in memory.
- Memory Efficiency: Uses String Pool for memory optimization.
- Performance: Slower for frequent modifications.

Example:

```
String s1 = "Hello";  // Stored in String Pool
String s2 = new String("Hello"); // Stored in Heap memory
s1 = s1 + " World";  // Creates a new object, original remains unchanged
```

2. StringBuffer (Mutable & Thread-safe)

- Definition: A StringBuffer is a mutable sequence of characters.
- Thread-Safety: Methods are synchronized, making it safe for multi-threaded environments.
- Performance: Slightly slower than StringBuilder due to synchronization.

Example:

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World"); // Modifies the existing object
System.out.println(sb); // Output: Hello World
```

3. StringBuilder (Mutable & Not Thread-safe)

- Definition: A StringBuilder is a mutable sequence of characters.
- Thread-Safety: Not synchronized, making it faster than StringBuffer.
- Performance: Best for single-threaded applications.

Example:

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World"); // Modifies the existing object
System.out.println(sb); // Output: Hello World
```

4.String Methods and Manipulation

String Methods and Manipulation in Python

Python provides various built-in string methods to manipulate and process text efficiently. Here's an overview of some of the most commonly used string methods and techniques:

1. Creating Strings

```
s1 = "Hello, World!"
s2 = 'Python Programming'
s3 = """This is a multiline string"""
```

2. String Indexing & Slicing

```
s = "Hello, Python"

# Indexing (0-based)
print(s[0])   # H
print(s[-1])  # n

# Slicing
print(s[0:5])  # Hello
print(s[:5])   # Hello
print(s[7:])   # Python
print(s[::2])  # Hlo yhn (every second character)
```

3. Common String Methods

Case Conversion

```
s = "hello world"
print(s.upper())   # HELLO WORLD
print(s.lower())   # hello world
print(s.title())   # Hello World
print(s.capitalize())  # Hello world
print(s.swapcase())    # HELLO WORLD
```

String Checking Methods

```
s = "Hello123"
print(s.isalpha())   # False (contains numbers)
print(s.isdigit())   # False
print(s.isalnum())   # True (letters + numbers)
```

```
print(s.isspace()) # False
print(" ".isspace()) # True
```

String Modification

```
s = "   Python Programming   "
print(s.strip())   # Removes leading/trailing spaces
print(s.lstrip())  # Removes leading spaces
print(s.rstrip())  # Removes trailing spaces
```

Replacing and Splitting

```
s = "Hello, World!"
print(s.replace("World", "Python"))  # Hello, Python!

s = "apple,banana,cherry"
print(s.split(","))  # ['apple', 'banana', 'cherry']
```

Joining Strings

```
words = ["Python", "is", "awesome"]
print(" ".join(words))  # Python is awesome
```

4. String Searching

```
s = "Python is amazing"
print(s.find("is"))     # 7 (index of first occurrence)
print(s.rfind("is"))    # 7 (searches from right)
print(s.startswith("Py"))  # True
print(s.endswith("ing"))   # True
```

6. Escape Characters

```
print("Hello\nWorld")   # New line
print("Hello\tWorld")   # Tab space
print("Hello \"Python\"")  # Double quotes inside string
```

7. Reversing a String

```
s = "Python"
print(s[::-1])  # nohtyP
```

8. Counting Occurrences

```
s = "banana"
print(s.count("a"))  # 3
```

9. Checking Substring

```
s = "Python programming"
print("Python" in s)   # True
print("Java" not in s) # True
```

10. Encoding & Decoding Strings

```
s = "Hello"
encoded_s = s.encode('utf-8')
print(encoded_s)  # b'Hello'

decoded_s = encoded_s.decode('utf-8')
print(decoded_s)  # Hello
```

CHAPTER-5
Exception Handling

1.Types of Exceptions (Checked and Unchecked)

In Java, exceptions are divided into two main categories:

1. **Checked Exceptions**
2. **Unchecked Exceptions**

1. Checked Exceptions

Checked exceptions are exceptions that the compiler checks at compile time. If a method throws a checked exception, it must either handle it using try-catch or declare it using the throws keyword in the method signature.

Examples of Checked Exceptions:

- **IOException (when dealing with file handling)**
- **SQLException (when interacting with databases)**
- **FileNotFoundException (when a file is not found)**
- **ClassNotFoundException (when a class is not found in the classpath)**
- **InterruptedException (when a thread is interrupted)**
- **InstantiationException (when trying to create an instance of an abstract class or interface)**

Example of Checked Exception (Handling using try-catch):

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class CheckedExceptionExample {
   public static void main(String[] args) {
      try {
         File file = new File("test.txt");
         FileReader fr = new FileReader(file); // This can throw
FileNotFoundException
      } catch (IOException e) {
         System.out.println("File not found: " + e.getMessage());
      }
   }
}
```

Example of Checked Exception (Declaring using throws)

```
import java.io.FileNotFoundException;
import java.io.FileReader;

public class CheckedExceptionExample {
   public static void readFile() throws FileNotFoundException {
      FileReader fr = new FileReader("test.txt"); //
FileNotFoundException
   }

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

```
try {
readFile();
} catch (FileNotFoundException e) {
System.out.println("Handled Exception: " + e.getMessage());
}
}
}
```

2. Unchecked Exceptions (Runtime Exceptions)

Unchecked exceptions are exceptions that occur at runtime and are not checked at compile time. They are usually caused by programming mistakes, such as invalid input, incorrect calculations, or null references. These exceptions extend the RuntimeException class.

Examples of Unchecked Exceptions:

- NullPointerException (when trying to access a null object)
- ArithmeticException (when dividing by zero)
- ArrayIndexOutOfBoundsException (when accessing an invalid array index)
- IllegalArgumentException (when passing an invalid argument to a method)
- NumberFormatException (when converting an invalid string to a number)
- ClassCastException (when performing an invalid type casting)

Example of Unchecked Exception (NullPointerException)

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        String str = null;
        System.out.println(str.length()); // NullPointerException
    }
}
```

Example of Uncheck(ArrayIndexOutOfBoundsException)

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        int[] arr = new int[5];
        System.out.println(arr[10]); //
ArrayIndexOutOfBoundsException
    }
}
```

Key Differences Between Checked and Unchecked Exceptions

	Checked Exceptions	Unchecked Exceptions
Definition	Checked at compile-time	Not checked at compile-time
Exception Types	IOException, ClassNotFoundException, etc.	NullPointerException, ArrayIndexOutOfBoundsException, etc.
Handling Requirement	Must be caught or declared using throws keyword	Optional to catch or declare using throws keyword
Exception Handling	Handled using try-catch blocks or throws keyword	Not required to handle explicitly
Flow Control	Program flow is interrupted and transferred to catch block	Program execution is halted with an error message
Examples	File operations, database operations, etc.	Logical errors, invalid arguments, etc.

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

In Java, try, catch, finally, throw, and throws are used for exception handling. They help in managing runtime errors and ensuring smooth execution of the program.

1. try Block

- The try block contains code that might throw an exception.
- If an exception occurs, it is handled by the catch block.
- If no exception occurs, the catch block is skipped.

```
try {
   int num = 10 / 0;  // This will cause ArithmeticException
   System.out.println(num);
}
```

2. catch Block

- The catch block handles exceptions thrown inside the try block.
- It must follow immediately after the try block.

```
try {
   int num = 10 / 0;
} catch (ArithmeticException e) {
   System.out.println("Cannot divide by zero!");
}
```

3. finally Block

- The finally block executes whether an exception occurs or not.
- It is mainly used for resource cleanup (e.g., closing files, database connections).

```
try {
   int num = 10 / 0;
} catch (ArithmeticException e) {
   System.out.println("Error: " + e.getMessage());
} finally {
   System.out.println("Execution completed."); // This will always execute
}
```

4. throw Statement

- The throw keyword is used to explicitly throw an exception.
- It is used inside a method or block.

```
class Test {
   static void checkAge(int age) {
      if (age < 18) {
         throw new IllegalArgumentException("Age must be 18 or above.");
      }
      System.out.println("Access granted.");
   }

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

```
 checkAge(16); // This will throw an exception
 }
}
```

5. throws Keyword

- The throws keyword is used in method signatures to declare exceptions that might be thrown.
- The calling method must handle these exceptions.

```
class Test {
   static void divide() throws ArithmeticException {
      int num = 10 / 0;
   }

   public static void main(String[] args) {
      try {
         divide();
      } catch (ArithmeticException e) {
         System.out.println("Exception caught: " +
e.getMessage());
      }
```

Custom Exceptions

In Java, exceptions are used to handle errors and unexpected situations that occur during the execution of a program. While Java provides many built-in exceptions (like NullPointerException, IOException, etc.), sometimes you need to create custom exceptions to handle specific business logic errors.

1. Why Use Custom Exceptions?

- To make error handling more meaningful and specific.
- To improve code readability and maintainability.
- To enforce business rules (e.g., an InsufficientFundsException in a banking app).

2. Creating a Custom Exception

To create a custom exception, extend either:

- Checked Exception: Extend Exception (must be handled using try-catch or throws).
- Unchecked Exception: Extend RuntimeException (optional handling).

Example 1: Creating a Checked Custom Exception

```
// Custom checked exception (must be handled)
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class CustomExceptionDemo {
    static void checkAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or
above.");
        } else {
            System.out.println("You are eligible to vote.");
        }
```

```
 }

 public static void main(String[] args) {
 try {
 checkAge(16); // This will throw an exception
 } catch (InvalidAgeException e) {
 System.out.println("Caught Exception: " + e.getMessage());
 }
 }
}
```

Output:
Caught Exception: Age must be 18 or above.

Example 2: Creating an Unchecked Custom Exception

```
// Custom unchecked exception (optional handling)
class InsufficientFundsException extends RuntimeException {
    public InsufficientFundsException(String message) {
        super(message);
    }
}

class BankAccount {
    private double balance = 5000;

    public void withdraw(double amount) {
        if (amount > balance) {
            throw new InsufficientFundsException("Insufficient
```

```
funds. Available balance: " + balance);
 }
 balance -= amount;
 System.out.println("Withdrawal successful. Remaining balance:
" + balance);
 }
}

public class Main {
 public static void main(String[] args) {
 BankAccount account = new BankAccount();
 account.withdraw(6000); // This will throw an exception
 }
}
```

CHAPTER-6
Java Collections Framework (JCF)

1.List (ArrayList, LinkedList)

In Java, List is an interface under the java.util package, and it is implemented by classes like ArrayList and LinkedList. Both provide dynamic array-like structures but have different internal implementations and performance characteristics.

1. ArrayList in Java

ArrayList is a resizable array implementation of the List interface. It allows dynamic resizing and provides fast random access.

Declaration and Initialization

```
import java.util.ArrayList;

public class ArrayListExample {
   public static void main(String[] args) {
      ArrayList<String> list = new ArrayList<>(); // Creating an ArrayList

      // Adding elements
      list.add("Apple");
      list.add("Banana");
      list.add("Cherry");

      // Accessing elements
```

```
System.out.println("Element at index 1: " + list.get(1));

// Iterating through the list
for (String fruit : list) {
System.out.println(fruit);
}

// Removing an element
list.remove("Banana");

System.out.println("After removal: " + list);
}
```

Key Features of ArrayList

- Uses a dynamic array to store elements.
- Provides O(1) time complexity for get(index), making random access fast.
- add() at the end is O(1) (amortized), but add(index, element) and remove(index) are O(n) due to shifting elements.
- Suitable for scenarios where frequent read operations are required.

2. LinkedList in Java

LinkedList implements both List and Deque interfaces and provides a doubly-linked list implementation.
Declaration and Initialization

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<>(); // Creating a
LinkedList

        // Adding elements
        list.add("Dog");
        list.add("Cat");
        list.add("Rabbit");

        // Accessing elements
        System.out.println("First element: " + list.getFirst());

        // Iterating through the list
        for (String animal : list) {
            System.out.println(animal);
        }

        // Removing an element
        list.remove("Cat");

        System.out.println("After removal: " + list);
    }
}
```

Key Features of LinkedList

- **Uses a doubly-linked list internally.**
- **Provides O(1) insertion and deletion at the beginning or end.**
- **get(index) is O(n) since traversal is needed.**
- **Best suited for scenarios with frequent insertions and deletions.**

Comparison: ArrayList vs. LinkedList

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

2.Set (HashSet, TreeSet)

In Java, Set is an interface in the java.util package that represents a collection of unique elements. The two common implementations of Set are:

1. HashSet
2. TreeSet

1. HashSet

- Implements the Set interface using a hash table.
- Does not maintain any specific order of elements.
- Allows null values.
- Fast performance for operations like add, remove, and contains (O(1) on average).
- Not synchronized (not thread-safe).

Example:

HashSet in Java

```
import java.util.HashSet;

public class HashSetExample {
   public static void main(String[] args) {
      HashSet<String> set = new HashSet<>();

      set.add("Apple");
      set.add("Banana");
      set.add("Orange");
      set.add("Apple"); // Duplicate, will be ignored
```

```
 System.out.println(set); // Output: [Banana, Orange, Apple]
(Order may vary)

 set.remove("Banana");
 System.out.println(set.contains("Banana")); // Output: false
 }
}
```

2. TreeSet

- Implements the Set interface using a Red-Black Tree (self-balancing BST).
- Maintains elements in sorted order (natural ordering or a custom comparator).
- Does not allow null values.
- Slower than HashSet for most operations (O(log n) for add, remove, contains).
- Not synchronized.

Example:

TreeSet in Java

```
import java.util.TreeSet;

public class TreeSetExample {
   public static void main(String[] args) {
      TreeSet<Integer> set = new TreeSet<>();
      set.add(50);
      set.add(10);
```

```
set.add(30);
 set.add(20);

 System.out.println(set); // Output: [10, 20, 30, 50] (Sorted
order)

 System.out.println(set.first()); // Output: 10
 System.out.println(set.last()); // Output: 50
 }
}
```

Key Differences: HashSet vs. TreeSet

HashSet	TreeSet
HashSet uses HashMap internally to store it's elements.	TreeSet uses TreeMap internally to store it's elements.
A HashSet is unordered .	TreeSet orders the elements according to supplied Comparator. If no comparator is supplied, elements will be placed in their natural ascending order.
HashSet is faster than TreeSet	TreeSet gives less performance than the HashSet as it has to sort the elements after each insertion and removal operations.
HashSet uses equals() and hashCode() methods to compare the elements and thus removing the possible duplicate elements.	TreeSet uses compare() or compareTo() methods to compare the elements and thus removing the possible duplicate elements. It doesn't use equals() and hashCode() methods for comparision of elements.
HashSet allows maximum one null element.	TreeSet doesn't allow even a single null element. If you try to insert null element into TreeSet, it throws NullPointerException.
HashSet requires less memory than TreeSet as it uses only HashMap internally to store its elements	TreeSet also requires more memory than HashSet as it also maintains Comparator to sort the elements along with the TreeMap.

3.Map (HashMap, TreeMap, LinkedHashMap)

In Java, the Map interface represents a collection of key-value pairs. The three commonly used implementations of Map are:

1. HashMap
2. TreeMap
3. LinkedHashMap

Each of these implementations has distinct properties and use cases.

1. HashMap

- Implementation: Uses a hash table.
- Order: Does not maintain any order of keys.
- Performance: Offers O(1) time complexity for put(), get(), and remove() operations in the average case.
- Allows null keys and values: Yes (only one null key, multiple null values).
- Synchronization: Not thread-safe.

Example:

```
import java.util.HashMap;
public class HashMapExample {
   public static void main(String[] args) {
      HashMap<Integer, String> map = new HashMap<>();
      map.put(1, "Apple");
      map.put(3, "Orange");
      map.put(2, "Banana");
```

```
 System.out.println(map); // Order is not guaranteed
 }
}
```

Use Case:

Use HashMap when you don’t need ordering and require fast performance.

2. TreeMap

- Implementation: Uses a Red-Black Tree.
- Order: Maintains keys in sorted (ascending) order.
- Performance: Operations like put(), get(), and remove() take O(log n) time.
- Allows null keys and values: Does not allow null keys, but allows multiple null values.
- Synchronization: Not thread-safe.

Example:

```
import java.util.TreeMap;

public class TreeMapExample {
   public static void main(String[] args) {
      TreeMap<Integer, String> map = new TreeMap<>();
      map.put(1, "Apple");
      map.put(3, "Orange");
      map.put(2, "Banana");

      System.out.println(map); // Output: {1=Apple, 2=Banana,
3=Orange} (sorted order)
```

Use Case:

Use TreeMap when you need keys to be sorted.

3. LinkedHashMap

- Implementation: Uses a hash table + doubly linked list.
- Order: Maintains insertion order.
- Performance: Similar to HashMap (O(1) for put(), get(), and remove()).
- Allows null keys and values: Yes.
- Synchronization: Not thread-safe.

Example:

```
import java.util.LinkedHashMap;

public class LinkedHashMapExample {
   public static void main(String[] args) {
      LinkedHashMap<Integer, String> map = new
LinkedHashMap<>();
      map.put(1, "Apple");
      map.put(3, "Orange");
      map.put(2, "Banana");

      System.out.println(map); // Output: {1=Apple, 3=Orange,
2=Banana} (insertion order)
   }
}
```

Use Case:

Use LinkedHashMap when you need predictable iteration order.

4.Queue and Deque

Queue and Deque in Java (Data Structures)
In Java, Queue and Deque (Double-Ended Queue) are part of the java.util package and are implemented using various data structures like linked lists, arrays, or priority queues.

1. Queue in Java

A Queue follows the FIFO (First In, First Out) principle, where elements are added at the rear and removed from the front.

Implementation of Queue in Java

Java provides the Queue interface, which is implemented by various classes like:

- **LinkedList**
- **PriorityQueue**
- **ArrayDeque**

Example of Queue Using LinkedList

```
import java.util.*;

public class QueueExample {
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();

        // Adding elements
        queue.add(10);
        queue.add(20);
```

```
queue.add(30);

 System.out.println("Queue: " + queue);

 // Removing elements
 System.out.println("Removed: " + queue.poll());

 // Peek element
 System.out.println("Front element: " + queue.peek());

 System.out.println("Final Queue: " + queue);
 }
}
```

Output:

```
Queue: [10, 20, 30]
Removed: 10
Front element: 20
Final Queue: [20, 30]
```

2. Deque (Double-Ended Queue)

A Deque allows insertion and deletion from both ends. It supports FIFO and LIFO operations.

Types of Deque

1. ArrayDeque – Resizable array-based implementation.
2. LinkedList – Doubly linked list-based implementation.

Example of Deque Using ArrayDeque

```
import java.util.*;

public class DequeExample {
    public static void main(String[] args) {
        Deque<Integer> deque = new ArrayDeque<>();

        // Adding elements at both ends
        deque.addFirst(10);
        deque.addLast(20);
        deque.addLast(30);

        System.out.println("Deque: " + deque);

        // Removing elements
        System.out.println("Removed First: " + deque.pollFirst());
        System.out.println("Removed Last: " + deque.pollLast());

        // Peek elements
        System.out.println("First Element: " + deque.peekFirst());
        System.out.println("Last Element: " + deque.peekLast());

        System.out.println("Final Deque: " + deque);
    }
}
```

Output:

Deque: [10, 20, 30]
Removed First: 10
Removed Last: 30
First Element: 20
Last Element: 20
Final Deque: [20]

Iterators and Stream API in java

In Java, Iterators and the Stream API are both used for iterating over collections of data, but they work in different ways and offer different functionalities.

Iterators in Java

An Iterator is an object that allows you to traverse a collection, typically in a sequential manner. It is part of the java.util package and provides methods to iterate over a collection, remove elements, and check the next element.

Key Methods of Iterator:

1. **hasNext() – Returns true if there is at least one more element in the collection.**
2. **next() – Returns the next element in the iteration.**
3. **remove() – Removes the current element (optional operation).**

Example of using an Iterator:

```
import java.util.*;

public class IteratorExample {
   public static void main(String[] args) {
      List<String> list = new ArrayList<>(Arrays.asList("Apple",
"Banana", "Cherry"));

      Iterator<String> iterator = list.iterator();

      while (iterator.hasNext()) {
         System.out.println(iterator.next());
      }
```

Stream API in Java

The Stream API was introduced in Java 8 and allows you to process collections of objects in a functional-style, making it easier to work with sequences of data (collections, arrays, I/O, etc.). It supports operations like map, filter, and reduce and can also process elements in parallel.

Key Features of Stream API:

1. **Functional-style operations.**
2. **Lazy evaluation: Operations on streams are evaluated only when needed (on terminal operations).**
3. **Parallel processing: Streams can be processed in parallel, leveraging multi-core architectures for better performance.**

Example of using Stream API:

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

public class StreamExample {
   public static void main(String[] args) {
      List<String> list = Arrays.asList("Apple", "Banana", "Cherry",
"Date");

      // Stream operation to filter and print items with length
greater than 5
      list.stream()
         .filter(item -> item.length() > 5)
         .forEach(System.out::println);
   }
}
```

In this example, the stream() method converts the list into a stream. The filter() method applies a condition (string length > 5), and forEach() prints each element in the filtered stream.

Some Useful Stream Operations:

- map() – Transforms elements.
- filter() – Filters elements based on a condition.
- reduce() – Reduces elements to a single value.
- collect() – Collects the results into a collection (e.g., list, set).
- forEach() – Applies an action to each element.

CHAPTER-7
File Handling in Java

1.Reading and Writing Files

In Java, reading from and writing to files is commonly done using classes from the java.io package, such as FileReader, FileWriter, BufferedReader, BufferedWriter, FileInputStream, and FileOutputStream.

Reading a File:

To read a file, you can use either FileReader (for text files) or FileInputStream (for binary files). BufferedReader can be used to read text files efficiently by buffering the input.

Example: Reading a text file with BufferedReader

```
import java.io.*;

public class ReadFileExample {
   public static void main(String[] args) {
      String filePath = "example.txt";
      try (BufferedReader reader = new BufferedReader(new
FileReader(filePath))) {
         String line;
         while ((line = reader.readLine()) != null) {
            System.out.println(line);
         }
      } catch (IOException e) {
```

```
  e.printStackTrace();
 }
 }
}
```

Explanation:

- BufferedReader is used to read the file line by line.
- readLine() reads one line at a time.
- The try-with-resources statement ensures that the file is closed automatically after reading.

Writing to a File:

You can use FileWriter or BufferedWriter to write text data to a file. The FileWriter class writes directly to a file, while BufferedWriter buffers the output to improve efficiency.

Example: Writing to a text file with BufferedWriter

```
import java.io.*;

public class WriteFileExample {
   public static void main(String[] args) {
      String filePath = "example.txt";
      try (BufferedWriter writer = new BufferedWriter(new
FileWriter(filePath))) {
         writer.write("Hello, this is a test file.");
         writer.newLine();
         writer.write("Writing multiple lines.");
```

```
is a test file.");
        writer.newLine();
        writer.write("Writing multiple lines.");
```

Explanation:

- BufferedWriter is used to write text data efficiently.
- write() writes a string to the file.
- newLine() inserts a new line.
- The try-with-resources statement ensures the file is closed automatically after writing.
-

Reading and Writing Binary Files:

For binary files, you can use FileInputStream and FileOutputStream.

Example: Reading a binary file with FileInputStream

```
import java.io.*;

public class ReadBinaryFileExample {
   public static void main(String[] args) {
      String filePath = "example.bin";
      try (FileInputStream fis = new FileInputStream(filePath)) {
         int byteData;
         while ((byteData = fis.read()) != -1) {
            System.out.print((char) byteData);  // Assuming the
binary data is text-based
         }
```

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

Example: Writing to a binary file with FileOutputStream

```
import java.io.*;

public class WriteBinaryFileExample {
   public static void main(String[] args) {
      String filePath = "example.bin";
      try (FileOutputStream fos = new
FileOutputStream(filePath)) {
         String data = "This is binary data!";
         fos.write(data.getBytes());
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

BufferedReader and BufferedWriter

In Java, BufferedReader and BufferedWriter are classes used for efficient reading and writing of data from and to a character stream. They wrap around other stream objects like FileReader or FileWriter to provide buffering, which helps in improving performance, especially for large data transfers.

BufferedReader:

BufferedReader is used to read text from a character-based input stream. It buffers the input to provide efficient reading of characters, arrays, and lines.

- Constructor:
  - BufferedReader(Reader in)
  - Creates a BufferedReader that uses a default buffer size.
  - BufferedReader(Reader in, int sz)
  - Creates a BufferedReader that uses a custom buffer size.
- Common Methods:
  - read(): Reads a single character from the stream.
  - readLine(): Reads a line of text (returns null at the end of the stream).
  - read(char[] cbuf): Reads characters into an array.
  -

Example:

```
import java.io.*;

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

```
try (BufferedReader reader = new BufferedReader(new
FileReader("input.txt"))) {
 String line;
 while ((line = reader.readLine()) != null) {
 System.out.println(line); // Prints each line from the file
 }
 } catch (IOException e) {
 e.printStackTrace();
 }
 }
}
```

3.File Handling with FileInputStream and FileOutputStream

In Java, FileInputStream and FileOutputStream are used to read and write binary data to files. Here's a basic overview of both and an example for each:

1. FileInputStream

FileInputStream is used to read data in the form of bytes from a file.

- Constructor:
  - FileInputStream(String name): Creates a FileInputStream object to read the file with the specified file name.
  - FileInputStream(File file): Creates a FileInputStream object for the specified File object.
- Methods:
  - read(): Reads the next byte of data from the input stream. Returns -1 when the end of the file is reached.
  - read(byte[] b): Reads bytes into the array b. Returns the number of bytes read, or -1 if the end of the file is reached.
  - close(): Closes the stream and releases any system resources associated with it.

2. FileOutputStream

FileOutputStream is used to write data in the form of bytes to a file.

- Constructor:
  - FileOutputStream(String name): Creates a

- FileOutputStream object to write to a file with the specified name.
- FileOutputStream(File file): Creates a FileOutputStream object for the specified File object.
- FileOutputStream(String name, boolean append): Creates a FileOutputStream with an option to append data to an existing file.

- **Methods:**
  - write(int b): Writes the specified byte to the file.
  - write(byte[] b): Writes an array of bytes to the file.
  - close(): Closes the stream and releases system resources.

Example: File Handling Using FileInputStream and FileOutputStream

Write Data to File Using FileOutputStream

```
import java.io.FileOutputStream;
import java.io.IOException;

public class FileOutputStreamExample {
   public static void main(String[] args) {
      FileOutputStream fos = null;
      try {
         fos = new FileOutputStream("example.txt");
         String data = "Hello, this is a test!";
         byte[] byteArray = data.getBytes();
         fos.write(byteArray);
```

```
System.out.println("Data written to file successfully.");
} catch (IOException e) {
e.printStackTrace();
} finally {
try {
if (fos != null) {
fos.close();
}
} catch (IOException e) {
e.printStackTrace();
}
}
}
```

4.Serialization and Deserialization

Serialization and deserialization in Java are concepts that allow you to convert objects into a stream of bytes and then reconstruct them back to their original state. This is useful, for example, when saving an object to a file or sending it over a network.

1. Serialization

Serialization is the process of converting an object's state into a byte stream so it can be easily saved to a file, transmitted over a network, or stored in memory. Java provides built-in support for serialization through the Serializable interface.

- **Steps for Serialization:**
  - **Mark the class of the object to be serialized with the Serializable interface.**
  - **Use ObjectOutputStream to write the object to an output stream (e.g., a file or network).**

Example:

```
import java.io.*;

class Person implements Serializable {
   String name;
   int age;

   Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
}

public class SerializationExample {
   public static void main(String[] args) {
      Person person = new Person("Alice", 30);

      try (ObjectOutputStream oos = new
ObjectOutputStream(new FileOutputStream("person.ser"))) {
         oos.writeObject(person); // Serialize the object to a file
```

```
System.out.println("Object serialized successfully!");
} catch (IOException e) {
e.printStackTrace();
}
}
}
```

Explanation:

- The class Person implements the Serializable interface.
- The object person is written to a file person.ser using ObjectOutputStream.

2. Deserialization

Deserialization is the process of reading an object's byte stream and converting it back into a Java object. You can use ObjectInputStream to read the serialized data and reconstruct the object.

- **Steps for Deserialization:**
  a. Use ObjectInputStream to read the object from an input stream (e.g., a file or network).
  b. The object is restored back to its original form.

Example:

```
import java.io.*;

class Person implements Serializable {
   String name;
   int age;
```

```
 Person(String name, int age) {
 this.name = name;
 this.age = age;
 }

 @Override
 public String toString() {
 return "Person{name='" + name + "', age=" + age + '}';
 }
}

public class DeserializationExample {
 public static void main(String[] args) {
 try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("person.ser"))) {
 Person person = (Person) ois.readObject(); // Deserialize the
object from the file
 System.out.println("Deserialized Object: " + person);
 } catch (IOException | ClassNotFoundException e) {
 e.printStackTrace();
 }
 }
}
```

Explanation:

- The object is read from the file person.ser using ObjectInputStream.
- It is cast back to the Person class and printed out.

Example with transient and serialVersionUID:

```
import java.io.*;

class Employee implements Serializable {
    private static final long serialVersionUID = 1L;

    String name;
    transient String password;  // This field will not be serialized

    Employee(String name, String password) {
        this.name = name;
        this.password = password;
    }

    @Override
    public String toString() {
        return "Employee{name='" + name + "', password='" +
password + "'}";
    }
}

public class SerializationExample {
    public static void main(String[] args) {
        Employee employee = new Employee("John", "12345");

        try (ObjectOutputStream oos = new
ObjectOutputStream(new FileOutputStream("employee.ser")))
```

```
{
 oos.writeObject(employee);
 } catch (IOException e) {
 e.printStackTrace();
 }

 try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("employee.ser"))) {
 Employee deserializedEmployee = (Employee) ois.readObject();
 System.out.println("Deserialized Employee: " +
deserializedEmployee);
 } catch (IOException | ClassNotFoundException e) {
 e.printStackTrace();
 }
 }
}
```

Explanation:

- The password field is marked transient, so it won't be serialized.
- The serialVersionUID is added to prevent issues when the class changes.

CHAPTER-9
JDBC (Java Database Connectivity)

1.Introduction to JDBC

JDBC (Java Database Connectivity) is an API (Application Programming Interface) that enables Java applications to interact with relational databases. It provides a standard interface for connecting to databases, executing SQL queries, and processing the results.

JDBC allows Java developers to build database-driven applications that can perform operations like retrieving, inserting, updating, and deleting data from a database. It acts as a bridge between Java applications and databases like MySQL, Oracle, PostgreSQL, and others.

Key Concepts of JDBC:

1. **JDBC Driver:**
   - **The JDBC driver is responsible for establishing a connection between Java applications and databases. It translates Java calls into database-specific calls and vice versa.**
   - **There are four types of JDBC drivers:**
     - **Type 1: JDBC-ODBC Bridge Driver**
     - **Type 2: Native-API Driver**
     - **Type 3: Network Protocol Driver**
     - **Type 4: Thin Driver (Pure Java Driver)**
2. **Connection Interface:**

Represents a connection to a specific database. Using

Sample Code (JDBC Example):

```
import java.sql.*;

public class JDBCExample {
   public static void main(String[] args) {
      Connection conn = null;
      Statement stmt = null;
      ResultSet rs = null;

      try {
         // Load the JDBC driver
         Class.forName("com.mysql.cj.jdbc.Driver");

         // Establish a connection to the database
         conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/m
ydb", "root", "password");

         // Create a statement
         stmt = conn.createStatement();

         // Execute a query
         String sql = "SELECT id, name, age FROM students";
         rs = stmt.executeQuery(sql);

         // Process the result set
         while (rs.next()) {
```

```
 int id = rs.getInt("id");
 String name = rs.getString("name");
 int age = rs.getInt("age");

 System.out.println("ID: " + id + ", Name: " + name + ", Age: " +
age);
 }
 } catch (SQLException | ClassNotFoundException e) {
 e.printStackTrace();
 } finally {
 try {
 // Close resources
 if (rs != null) rs.close();
 if (stmt != null) stmt.close();
 if (conn != null) conn.close();
 } catch (SQLException se) {
 se.printStackTrace();
 }
 }
 }
}
```

Connecting Java with MySQL
To connect Java with MySQL, you need to follow these steps:

1. Install MySQL Server

Make sure you have MySQL server installed and running on your machine.

2. Download MySQL JDBC Driver

You need to download the MySQL JDBC driver (Connector/J). You can download it from the official MySQL website. After downloading, add the .jar file to your project.
If you're using a build tool like Maven or Gradle, you can add the dependency instead.

Maven Dependency Example:

```
<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <version>8.0.26</version>
</dependency>
```

3. Set Up the Database
Create a database in MySQL, for example:
CREATE DATABASE testdb;

Create a table:

```
USE testdb;
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL
);
```

4. Java Code to Connect to MySQL

Now, let's write the Java code to connect to MySQL.

Example Java Code:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class MySQLConnectionExample {
    public static void main(String[] args) {
        // MySQL URL, username and password
        String url = "jdbc:mysql://localhost:3306/testdb"; // Change
the URL if your MySQL is running on a different host or port
        String username = "root"; // Change this with your MySQL
username
        String password = "yourpassword"; // Change this with your
MySQL password
```

```
try {
 // 1. Load and register the JDBC driver
 Class.forName("com.mysql.cj.jdbc.Driver");

 // 2. Establish the connection
 Connection connection = DriverManager.getConnection(url,
username, password);
 System.out.println("Connected to the database!");

 // 3. Create a Statement to execute SQL queries
 Statement statement = connection.createStatement();

 // 4. Query the database
 String query = "SELECT * FROM users";
 ResultSet resultSet = statement.executeQuery(query);

 // 5. Process the ResultSet
 while (resultSet.next()) {
 int id = resultSet.getInt("id");
 String name = resultSet.getString("name");
 String email = resultSet.getString("email");

 System.out.println("ID: " + id + ", Name: " + name + ", Email: " +
email);
 }

 // 6. Close the resources
```

```
resultSet.close();
 statement.close();
 connection.close();

 } catch (ClassNotFoundException e) {
 System.out.println("MySQL JDBC Driver not found");
 e.printStackTrace();
 } catch (SQLException e) {
 System.out.println("Connection failed!");
 e.printStackTrace();
 }
 }
}
```

Breakdown of the Code:

1. Loading the Driver:
   - Class.forName("com.mysql.cj.jdbc.Driver"); loads the MySQL JDBC driver into memory.
2. Establishing the Connection:
   - DriverManager.getConnection(url, username, password) establishes a connection to the MySQL database.
3. Creating a Statement:
   - connection.createStatement() creates a Statement object for executing SQL queries.
4. Executing a Query:
   - statement.executeQuery(query) executes the SQL query and returns a ResultSet.

5. **Processing the Result:**
   - resultSet.next() iterates through the result set.
6. **Closing the Resources:**
   - Close ResultSet, Statement, and Connection to avoid memory leaks.

-

Common Issues and Troubleshooting:

1. **Driver Not Found:** Ensure that the MySQL JDBC driver JAR file is correctly added to the classpath.
2. **Connection Issues:** Make sure MySQL is running, and the URL, username, and password are correct.
3. **SQL Errors:** Ensure your SQL syntax and table names are correct.

3.CRUD Operations using JDBC

CRUD (Create, Read, Update, Delete) operations are fundamental for interacting with a database in Java using JDBC (Java Database Connectivity). Here's an overview of how to implement each operation.

Prerequisites

1. **JDBC Driver: Ensure that the correct JDBC driver is included in your project (e.g., for MySQL, you'll need mysql-connector-java).**
2. **Database Setup: Ensure you have a database and table created. For example, consider a simple table like:**

```
CREATE TABLE users (
   id INT PRIMARY KEY AUTO_INCREMENT,
   name VARCHAR(100),
   email VARCHAR(100)
);
```

Steps for JDBC CRUD Operations

1. **Set up JDBC connection: Use Connection to connect to your databa**

```
import java.sql.*;

public class DatabaseUtil {
   public static Connection getConnection() throws
```

```
SQLException {
 try {
 // Load the JDBC driver (for MySQL, for example)
 Class.forName("com.mysql.cj.jdbc.Driver");
 return
DriverManager.getConnection("jdbc:mysql://localhost:3306/yo
ur_database", "username", "password");
 } catch (Exception e) {
 e.printStackTrace();
 throw new SQLException("Database connection failed");
 }
 }
}
```

1. Create (Insert)

To insert data into the database, use the INSERT INTO SQL statement.

```
import java.sql.*;

public class UserDao {

   public void createUser(String name, String email) {
      String query = "INSERT INTO users (name, email) VALUES (?,
?)";

      try (Connection conn = DatabaseUtil.getConnection();
          PreparedStatement stmt =
```

```
        conn.prepareStatement(query)) {

 stmt.setString(1, name);
 stmt.setString(2, email);

 int rowsAffected = stmt.executeUpdate();
 if (rowsAffected > 0) {
 System.out.println("User added successfully!");
 }
 } catch (SQLException e) {
 e.printStackTrace();
 }
 }
}
```

2. Read (Select)

To retrieve data from the database, use the SELECT statement.

```
import java.sql.*;

public class UserDao {

   public void getUserById(int id) {
      String query = "SELECT * FROM users WHERE id = ?";

      try (Connection conn = DatabaseUtil.getConnection();
          PreparedStatement stmt =
conn.prepareStatement(query)) {
```

```
    stmt.setInt(1, id);
 ResultSet rs = stmt.executeQuery();

 while (rs.next()) {
 System.out.println("ID: " + rs.getInt("id"));
 System.out.println("Name: " + rs.getString("name"));
 System.out.println("Email: " + rs.getString("email"));
 }
 } catch (SQLException e) {
 e.printStackTrace();
 }
 }
}
```

3. Update

To modify an existing record, use the UPDATE SQL statement.

```
import java.sql.*;

public class UserDao {

   public void updateUser(int id, String name, String email) {
      String query = "UPDATE users SET name = ?, email = ?
WHERE id = ?";

      try (Connection conn = DatabaseUtil.getConnection();
          PreparedStatement stmt =
conn.prepareStatement(query)) {
```

```
stmt.setString(1, name);
stmt.setString(2, email);
stmt.setInt(3, id);

int rowsAffected = stmt.executeUpdate();
if (rowsAffected > 0) {
System.out.println("User updated successfully!");
}
} catch (SQLException e) {
e.printStackTrace();
}
}
}
```

4. Delete

To remove a record from the database, use the DELETE FROM SQL statement.

```
import java.sql.*;

public class UserDao {

   public void deleteUser(int id) {
      String query = "DELETE FROM users WHERE id = ?";

      try (Connection conn = DatabaseUtil.getConnection();
          PreparedStatement stmt =
conn.prepareStatement(query)) {
```

```
 stmt.setInt(1, id);

 int rowsAffected = stmt.executeUpdate();
 if (rowsAffected > 0) {
 System.out.println("User deleted successfully!");
 }
 } catch (SQLException e) {
 e.printStackTrace();
 }
 }
}
```

Putting it all together:

```
public class Main {
   public static void main(String[] args) {
      UserDao userDao = new UserDao();
            // Create a new user
      userDao.createUser("John Doe", "johndoe@example.com");
      // Read user details
      userDao.getUserById(1);
     // Update user details
      userDao.updateUser(1, "John Smith",
"johnsmith@example.com");
      // Delete user
      userDao.deleteUser(1);
   }
}
```

4.Prepared Statements and Callable Statements

In Java, PreparedStatement and CallableStatement are both types of Statement objects used to interact with a database, but they serve different purposes and have distinct features. Here's an explanation of each:

1. PreparedStatement

A PreparedStatement is an extension of the Statement interface. It is used to execute SQL queries that may be executed multiple times, with the ability to bind input parameters to the query dynamically. Using prepared statements can help prevent SQL injection attacks, improve performance (especially when running the same query multiple times), and provide better readability.

Key Features:

- **Parameterized Queries: You can use placeholders (?) in your SQL query for parameters, and then bind values to those placeholders at runtime.**
- **Pre-compilation: The SQL query is pre-compiled, meaning the database can optimize the execution plan for repeated executions, which can lead to performance improvements.**
- **Prevents SQL Injection: By using placeholders and binding values rather than concatenating strings, it mitigates the risk of SQL injection attacks.**

Example:

```
import java.sql.*;

public class PreparedStatementExample {
   public static void main(String[] args) {
      String url = "jdbc:mysql://localhost:3306/mydb";
      String username = "root";
      String password = "password";

      try (Connection conn = DriverManager.getConnection(url,
username, password)) {
         String sql = "SELECT * FROM users WHERE username = ?
AND age = ?";
         try (PreparedStatement stmt =
conn.prepareStatement(sql)) {
            stmt.setString(1, "john_doe");  // Bind first parameter
            stmt.setInt(2, 25);  // Bind second parameter
            ResultSet rs = stmt.executeQuery();

            while (rs.next()) {
               System.out.println(rs.getString("username") + ": " +
rs.getInt("age"));
            }
         }
      } catch (SQLException e) {
         e.printStackTrace();
      }
   }
```

2. CallableStatement

A CallableStatement is used to execute SQL stored procedures or functions in a database. Stored procedures are precompiled SQL code that can be called and executed by the database server, often used for repetitive or complex operations.

Key Features:

- Stored Procedures: CallableStatement allows you to execute SQL stored procedures or functions on the database, which may include input and output parameters.
- Output Parameters: CallableStatement can retrieve output parameters, making it suitable for functions that return values.
- More Complex Queries: It's useful for executing more complex database operations that are encapsulated within a stored procedure.

Example:

```
import java.sql.*;

public class CallableStatementExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mydb";
        String username = "root";
        String password = "password";

        try (Connection conn = DriverManager.getConnection(url,
username, password)) {
            String sql = "{call getUserDetails(?, ?)}";  // Calling a stored
```

```
procedure with two input parameters
 try (CallableStatement stmt = conn.prepareCall(sql)) {
 stmt.setInt(1, 1001); // Bind first input parameter
 stmt.registerOutParameter(2, Types.VARCHAR); // Register
output parameter
 stmt.execute();

 // Retrieve the output parameter
 String userDetails = stmt.getString(2);
 System.out.println("User Details: " + userDetails);
 }
 } catch (SQLException e) {
 e.printStackTrace();
 }
 }
}
```

Key Differences between PreparedStatement and CallableStatement:

- Usage:
  - PreparedStatement is used for executing parameterized SQL queries (usually SELECT, INSERT, UPDATE, DELETE).
  - CallableStatement is used for executing stored procedures or functions.
- SQL Type:
  - PreparedStatement deals with regular SQL queries.

CallableStatement deals with stored procedures, which

- can contain complex logic and multiple SQL statements.

- **Parameter Binding:**
  - Both allow parameter binding, but PreparedStatement is used with regular SQL queries, while CallableStatement is for calling stored procedures with input/output parameters.
- **Performance:**
  - PreparedStatement can offer performance benefits when executing the same query multiple times, as it uses precompiled SQL. CallableStatement is optimized for executing prewritten stored procedures, which may also improve performance depending on the procedure's complexity.

CHAPTER-10.
Advanced Java Concepts

1.Lambda Expressions

Lambda Expressions in Java

Lambda expressions were introduced in Java 8 and allow you to write more concise and readable code, especially for functional interfaces (interfaces with a single abstract method). They enable functional-style programming and make working with streams, collections, and multi-threading more efficient.

Syntax of Lambda Expression

```
(parameter) -> expression
(parameter) -> { statements; }
```

Example 1: Lambda Expression with Functional Interface

```
@FunctionalInterface
interface MathOperation {
   int operation(int a, int b);
}

public class LambdaExample {
   public static void main(String[] args) {
      // Lambda expression for addition
      MathOperation add = (a, b) -> a + b;

      // Lambda expression for multiplication
      MathOperation multiply = (a, b) -> a * b;
```

```
        // Lambda expression for multiplication
        MathOperation multiply = (a, b) -> a * b;

        System.out.println("Addition: " + add.operation(5, 3));      //
Output: 8
        System.out.println("Multiplication: " + multiply.operation(5,
3)); // Output: 15
    }
}
```

Types of Lambda Expressions

1. Without parameters

```
() -> System.out.println("Hello, Lambda!");
```

2.With a single parameter

```
name -> System.out.println("Hello, " + name);
```

3.With multiple parameters

```
(a, b) -> a + b;
```

4.With block statements

```
(a, b) -> {
    int sum = a + b;
    return sum;
};
```

Example 2: Lambda with Java Collections (Sorting List)

```
import java.util.*;

public class LambdaSorting {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Santhosh", "Neela",
"Chinni", "Hari");

        // Sorting using Lambda
        Collections.sort(names, (a, b) -> a.compareTo(b));

        System.out.println("Sorted Names: " + names);
    }
}
```

Example 3: Using Lambda in Threads

```
public class LambdaThread {
    public static void main(String[] args) {
        // Traditional way using Anonymous class
        Thread t1 = new Thread(new Runnable() {
            public void run() {
                System.out.println("Thread running using Anonymous
class.");
            }
        });
        t1.start();

        // Using Lambda expression
```

```
Thread t2 = new Thread(() -> System.out.println("Thread
running using Lambda."));
 t2.start();
 }
}
```

Example 4: Using Lambda with Java Streams

```
import java.util.*;

public class LambdaStream {
   public static void main(String[] args) {
      List<Integer> numbers = Arrays.asList(2, 4, 6, 8, 10);

      // Using lambda to print all elements
      numbers.forEach(n -> System.out.print(n + " "));

      // Using lambda with filter
      numbers.stream()
            .filter(n -> n > 5)
            .forEach(n -> System.out.println("\nFiltered: " + n));
   }
}
```

2.Streams and Functional Programming

Streams and Functional Programming in Java

1. Introduction to Streams

A Stream in Java is a sequence of elements that can be processed in a functional-style pipeline. It does not store data but operates on data sources such as collections, arrays, or I/O channels.

- **Streams enable functional programming in Java.**
- **They allow operations such as filtering, mapping, and reducing.**
- **They can be either sequential or parallel.**

2. Key Characteristics of Streams

- **Laziness: Computation is only performed when needed.**
- **Pipeline Processing: Operations are composed into a sequence.**
- **Immutable: Does not modify the source collection.**
- **Internal Iteration: Automates iteration logic.**

3. Functional Programming Concepts in Java

Functional programming treats computation as evaluating mathematical functions and avoids changing state & mutable data.

- **Lambda Expressions: Anonymous functions that simplify code.**
- **Method References: Compact notation to refer to existing methods.**

- **Higher-Order Functions: Functions that take other functions as arguments.**

4. Stream Operations

Streams support two types of operations:

- **Intermediate Operations (Lazy)**
  - **map(): Transforms elements.**
  - **filter(): Selects elements based on a condition.**
  - **sorted(): Sorts elements.**
  - **distinct(): Removes duplicates.**
  - **limit() / skip(): Controls the number of elements.**
- **Terminal Operations (Eager)**
  - **forEach(): Iterates over elements.**
  - **collect(): Gathers results into a collection.**
  - **reduce(): Aggregates elements.**
  - **count(): Counts elements.**
  - **anyMatch() / allMatch() / noneMatch(): Checks conditions.**

5. Example Usage

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

public class StreamExample {
   public static void main(String[] args) {
      List<Integer> numbers = Arrays.asList(2, 4, 5, 8, 10, 3, 6);

      // Filter even numbers and print
```

```
numbers.stream()
filter(n -> n % 2 == 0)
forEach(System.out::println);

// Square each number and collect to a list
List<Integer> squares = numbers.stream()
map(n -> n * n)
collect(Collectors.toList());
System.out.println(squares);

// Sum of all numbers
int sum = numbers.stream().reduce(0, Integer::sum);
System.out.println("Sum: " + sum);
}
}
```

6. Parallel Streams

Parallel processing can be used to speed up computations:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.parallelStream().reduce(0, Integer::sum);
System.out.println("Parallel Sum: " + sum);
```

7. Benefits of Using Streams

- Concise Code: Reduces boilerplate loops.
- Performance Optimization: Parallel execution.
- Functional Approach: Readable and maintainable.

3.Java 8 Features (Optional, Method References, Default Methods)

Java 8 introduced several new features that improved code readability and maintainability. Let's discuss three key features: Optional, Method References, and Default Methods.

1. Optional

Optional<T> is a container object that may or may not contain a non-null value. It helps to avoid NullPointerException and provides a better way to handle missing values.

Example: Using Optional

```
import java.util.Optional;

public class OptionalExample {
    public static void main(String[] args) {
        Optional<String> optional =
Optional.ofNullable(getValue());

        // Using ifPresent
        optional.ifPresent(value -> System.out.println("Value: " +
value));

        // Using orElse
        String result = optional.orElse("Default Value");
        System.out.println("Result: " + result);

        // Using orElseGet
        String result2 = optional.orElseGet(() -> "Generated Default
```

```
Value");
 System.out.println("Result2: " + result2);
 }

 public static String getValue() {
 return null; // Simulating a missing value
 }
}
```

Key Methods of Optional

- Optional.of(value) → Creates an Optional with a non-null value.
- Optional.ofNullable(value) → Creates an Optional that may hold a null.
- isPresent() → Checks if a value is present.
- ifPresent(Consumer<T>) → Executes if the value is present.
- orElse(T other) → Returns the value or a default.
- orElseGet(Supplier<T>) → Returns the value or generates a default.
- orElseThrow(Supplier<Throwable>) → Throws an exception if no value is present.

2. Method References

Method References are a shorthand way of writing lambda expressions for methods that already have a name.
Types of Method References

1.Reference to a Static Method

```
import java.util.function.Function;

public class StaticMethodReference {
    public static void main(String[] args) {
        Function<String, Integer> function = Integer::parseInt;
        int num = function.apply("100");
        System.out.println("Parsed Integer: " + num);
    }
}
```

2.Reference to an Instance Method of a Particular Object

```
class Printer {
    void print(String message) {
        System.out.println(message);
    }
}

public class InstanceMethodReference {
    public static void main(String[] args) {
        Printer printer = new Printer();
        Runnable r = printer::print; // Instead of () -> printer.print()
        r.run();
    }
}
```

3.Reference to an Instance Method of an Arbitrary Object

```
import java.util.Arrays;
import java.util.List;

public class ArbitraryMethodReference {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("John", "Doe", "Alice");
        names.forEach(System.out::println);
    }
}
```

4.Reference to a Constructor

```
import java.util.function.Supplier;

class Person {
    String name;
    Person() {
        this.name = "Default Name";
    }
}

public class ConstructorReference {
    public static void main(String[] args) {
        Supplier<Person> supplier = Person::new;
        Person person = supplier.get();
        System.out.println("Person Name: " + person.name);
    }
}
```

3. Default Methods

Before Java 8, interfaces could only have abstract methods. With default methods, we can now define method implementations in interfaces.

Why Default Methods?

- Allows adding new methods to interfaces without breaking existing implementations.
- Helps in providing default behavior in APIs like Java Collections.

Example: Default Method in Interface

```
interface Vehicle {
    void start(); // Abstract method

    default void stop() {
        System.out.println("Vehicle is stopping...");
    }
}

class Car implements Vehicle {
    public void start() {
        System.out.println("Car is starting...");
    }
}

public class DefaultMethodExample {
    public static void main(String[] args) {
        Car car = new Car();
        car.start();
```

```
car.stop(); // Calls default method
}
}
```

Rules for Default Methods

- A class implementing an interface gets the default method automatically.
- If a class implements multiple interfaces with the same default method, it must override it to resolve conflicts.
- Default methods can be overridden in the implementing class.

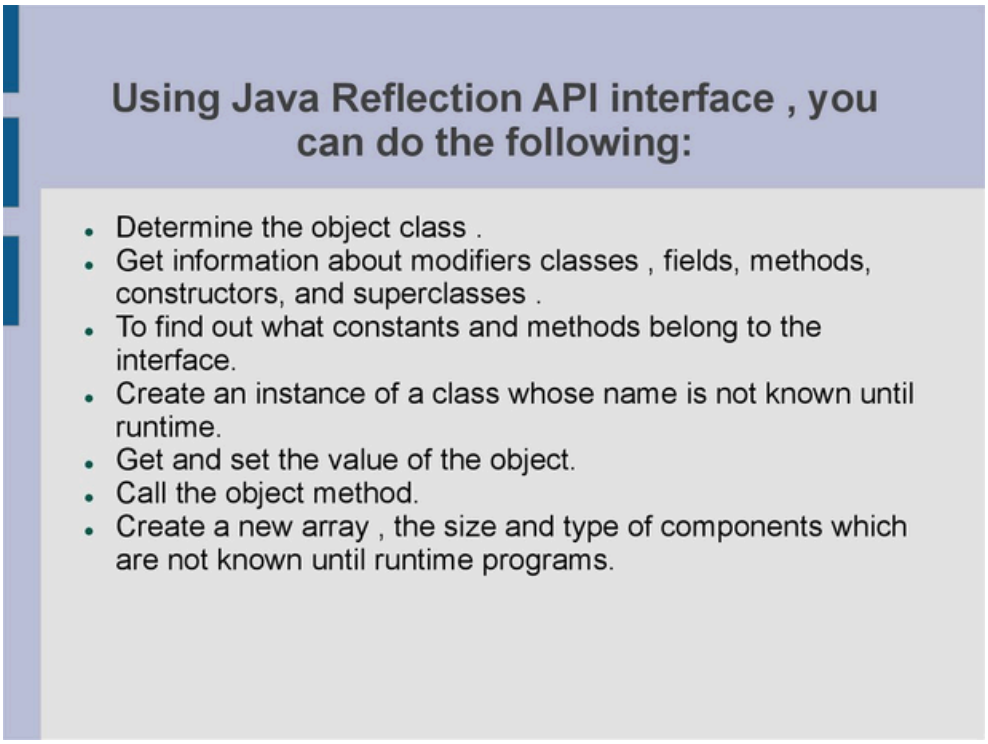
4.Reflection API

The Reflection API in Java is used to inspect and manipulate classes, methods, fields, and constructors at runtime. It allows a Java program to analyze itself and modify its behavior dynamically.

1. Key Features of Reflection API

- **Inspect classes, interfaces, and objects at runtime.**
- **Access private fields and methods.**
- **Create new instances of classes dynamically.**
- **Invoke methods dynamically.**
- **Modify field values at runtime.**

2. Important Classes in Reflection API



3. Getting Class Object

To use reflection, you need a Class object. There are three ways to get it:

Example: Obtaining Class Object

```
class Example {}

public class ReflectionDemo {
   public static void main(String[] args) throws
ClassNotFoundException {
      // Method 1: Using .class
      Class<?> cls1 = Example.class;

      // Method 2: Using getClass()
      Example obj = new Example();
      Class<?> cls2 = obj.getClass();

      // Method 3: Using forName()
      Class<?> cls3 = Class.forName("Example");

      System.out.println(cls1.getName());
      System.out.println(cls2.getName());
      System.out.println(cls3.getName());
   }
}
```

4. Accessing Fields Using Reflection

```
import java.lang.reflect.Field;

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

public class ReflectionFields {
   public static void main(String[] args) throws Exception {
      Person person = new Person();

      // Get the Class object
      Class<?> cls = person.getClass();

      // Get all fields (public only)
      Field[] fields = cls.getFields();
      for (Field field : fields) {
         System.out.println("Public Field: " + field.getName());
      }
      // Access private field
      Field privateField = cls.getDeclaredField("age");
      privateField.setAccessible(true); // Allow access to private
field
      privateField.set(person, 25);
      System.out.println("Age: " + privateField.get(person));
   }
}
```

5. Accessing Methods Using Reflection

```
import java.lang.reflect.Method;

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

public class ReflectionMethods {
    public static void main(String[] args) throws Exception {
        Calculator calc = new Calculator();
        Class<?> cls = calc.getClass();

        // Get method details
        Method method = cls.getMethod("add", int.class, int.class);
        int result = (int) method.invoke(calc, 5, 3);
        System.out.println("Addition Result: " + result);
    }
}
```

6. Creating Objects Dynamically Using Reflection

```
import java.lang.reflect.Constructor;

class Student {
    private String name;

    public Student(String name) {
        this.name = name;
    }

    public void display() {
        System.out.println("Student Name: " + name);
    }
}

public class ReflectionConstructor {
    public static void main(String[] args) throws Exception {
        // Get Constructor
        Constructor<Student> constructor =
Student.class.getConstructor(String.class);

        // Create an object dynamically
        Student student = constructor.newInstance("John Doe");
        student.display();
    }
}
```

7. Modifying Private Methods Using Reflection

```
import java.lang.reflect.Method;

class Secret {
    private void hiddenMessage() {
        System.out.println("This is a secret method.");
    }
}

public class ReflectionPrivateMethod {
    public static void main(String[] args) throws Exception {
        Secret secret = new Secret();
        Class<?> cls = secret.getClass();

        // Access private method
        Method privateMethod =
cls.getDeclaredMethod("hiddenMessage");
        privateMethod.setAccessible(true);
        privateMethod.invoke(secret);
    }
}
```

8. Practical Use Cases of Reflection API

1. **Framework Development: Used in Hibernate, Spring, and JUnit for dependency injection and testing.**
2. **Serialization & Deserialization: Convert objects to/from JSON/XML.**
3. **Dynamic Proxy Classes: Used in Java Dynamic Proxy API.**
4. **Debugging & Profiling: Helps in debugging tools to inspect object structure.**
5. **Developing IDEs & Tools: Used in tools like IntelliJ, Eclipse for code analysis.**

9. Disadvantages of Reflection API

1. **Performance Overhead: Reflection is slower than direct method calls.**
2. **Security Issues: Can break encapsulation by accessing private members.**
3. **Complexity: Code using reflection is harder to read and maintain.**

This Java material is for reference to gain basic knowledge about Java; don't rely solely on it, and also refer to other internet resources for competitive exams. Thank you from CodTech.