Theory

1. *Introduction to Java*

-

1. *History of Java*

- The history of Java is very interesting.
- The history of Java starts with the Green Team.
- Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc.
- Java was developed by James Gosling, who is known as the father of Java, in 1995.
- Java is used in internet programming, mobile devices, games, e-business solutions, etc.
- **James Gosling, Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.
- Initially it was designed for small, embedded systems in electronic appliances like set-top boxes.
- Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was .gt.
- After that, it was called **Oak** and was developed as a part of the Green project.
- **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.
- In 1995, Oak was renamed as **"Java"**
- Java is an island in Indonesia where the first coffee was produced (called Java coffee).
- It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.
- Initially developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.
- In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.
- JDK 1.0 was released on January 23, 1996. After the first release of Java, there have been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds new features in Java.

Java Version History

- First version :- JDK Alpha and Beta (1995)
- Last version :- Java 23

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

The features of Java are also known as Java buzzwords.

12 Features in java

1. Simple
2. Object-Oriented
3. Portable
4. Platform independent

5. Secured
6. Robust
7. Architecture neutral
8. Interpreted
9. High Performance
10. Multithreaded
11. Distributed
12. Dynamic

1. Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. Java language is a simple programming language because:

- Java syntax is based on C++.
- Java has removed many complicated and rarely-used features. for example, explicit pointers, operator overloading, etc.
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

2. Object-oriented

Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behavior.

Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

3. Platform Independent

Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines **while Java is a write once, run anywhere language**. A platform is the hardware or software environment in which a program runs.**There are two types of platforms software-based and hardware-based. Java provides a software-based platform.**

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on top of other hardware-based platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

Java code can be executed on multiple platforms, for example, Windows, Linux and Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere (WORA).

4. Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- **No explicit pointer**
- **Java Programs run inside a virtual machine sandbox**

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

5. Robust

The English mining of Robust is strong. Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

6. Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture.

However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

7. Portable

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

8. High-performance

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

9. Distributed

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

10. Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

11. Dynamic

Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.Java supports dynamic compilation and automatic memory management (garbage collection).

12. Interpreted

Java is an interpreted programming language that uses both compilation and interpretation to run code. This is why it's known as a "compiler-interpreter language

3. Understanding JVM, JRE, and JDK

1. JVM (Java Virtual Machine)

The **JVM** is the component responsible for running Java applications. It acts as an interpreter or virtual machine that executes Java bytecode. Java code is compiled into bytecode (an intermediate form), which the JVM interprets and runs on any platform that has a JVM installed.

Key Features of JVM:

- **Platform Independence**: The JVM makes Java a platform-independent language. The same Java bytecode can run on any operating system (Windows, Linux, macOS) as long as the JVM is available.
- **Memory Management**: JVM handles memory allocation, garbage collection (automatic cleanup of unused objects), and exception handling.
- **Execution of Bytecode**: It reads the bytecode from Java classes and translates it into machine code, which the underlying hardware can understand.

2. JRE (Java Runtime Environment)

The **JRE** provides a runtime environment for Java applications. It consists of everything needed to run Java programs, including the JVM, standard libraries, and other resources required for executing Java applications.

Theory

Key Features of JRE:

- **JVM**: The JRE includes the JVM, which is responsible for interpreting and running the bytecode.
- **Libraries**: It contains core libraries like `java.lang`, `java.util`, and others, which provide the essential functionality for Java applications (e.g., file I/O, networking, data manipulation, etc.).
- **Environment for Execution**: The JRE is essentially the environment that allows you to run Java applications. If you only need to run Java programs, you only need to install the JRE.

The JRE **does not** contain development tools such as a compiler or debugger.

3. JDK (Java Development Kit)

The **JDK** is a full-featured software development kit for Java developers. It includes everything in the JRE, along with additional tools needed for Java development, such as compilers, debuggers, and other utilities.

Key Features of JDK:

- **JRE**: The JDK includes the JRE, so it has everything needed to run Java programs.
- **Compiler (`javac`)**: The JDK contains the Java compiler (`javac`), which translates Java source code (`.java` files) into bytecode (`.class` files).
- **Debugger**: It includes a debugger that helps in identifying and fixing bugs in Java programs.
- **Development Tools**: The JDK provides other tools like `javadoc` for generating documentation from Java source code, `jarsigner` for signing Java applications, and more.

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

- ***IntelliJ***

1. Launch IntelliJ IDEA
2. Create a new project
3. Select Java from the list on the left
4. Name the project
5. Select the latest available Oracle OpenJDK version from the JDK list
6. If the JDK is installed but not defined in the IDE, select Add JDK and specify the path to the JDK home directory
7. If the JDK is not installed, select Download JDK
8. Specify the JDK vendor, version, and installation path
9. Click Download
10. Apply the changes and close the dialog

Theory

➢ ***Eclipse***

1 Open Eclipse.
2 Click **Workbench**.
3 Click **Window** > **Open Perspective** > **Java** to open the Java perspective.
4 Click **File** > **Java Project** to create a new Java project.
5 Enter a **Project name**.
6 Click **Finish**.
7 Right-click the new project, then click **Properties**.
8 On the **Properties** window, click **Java Build Path**.
9 Click the **Libraries** tab.
10 Click **Add External JARs**.
11 Add the javatiapi.jar from your TM1 installation directory.
12 Click **OK** to save the properties.
13 Right-click the new project, then click **New** > **File**.
14 Enter extension.xml as the file name.
15 Click **Finish**.

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

A Java program is a collection of classes. Each class is normally written in a separate file and the name of the file is the name of the class contained in the file, with the extension . java.

1. Packages

Organize related classes. Declared at the top of the Java file with `package`.

A **package** is a way to organize related classes and interfaces. It's similar to a directory in a file system that contains classes and other resources. Packages help avoid naming conflicts and make code modular and easier to maintain. Java provides a predefined package (java.lang) and also allows the creation of custom packages.

2. Classes

Blueprint for objects, containing fields and methods. Declared with the `class` keyword.

A class is the blueprint for creating objects. It defines variables and methods that the objects created from the class will have. Every Java application has at least one class that contains the main method, which is the entry point for the program execution.

3. Methods

Functions inside a class that define the behavior of the class. Declared with a return type, name, and parameters.

A method in Java is a block of code that performs a specific task. It can take inputs (parameters), return a value, and may or may not modify the state of the object (instance variables). Every Java program has a special method called the main() method, which is the entry point of the program.

Theory

2.DataTypes,Variables,andOperators

1. PrimitiveDataTypes in Java(int,float,char,etc.)

Data type :- data types are used to specify the type of data to store inside the variables.

Data types in Java categories into two categories:

- Primitive
- Non-primitive

Primitive data type :-

Primitive data types in Java are predefined by the Java language and named as the reserved keywords. A primitive data type does not share a state with other primitive values.

Boolean ,byte ,int ,long ,char ,float ,short,double

1. Boolean data type:- A boolean data type can have two types of values, which are **true** and **false.** It is used to add a simple flag that displays true/false conditions.
2. byte data type :- It is an 8-bit signed 2's complement integer.It is useful for saving memory in large Arrays.
3. int data type :- The int stands for Integer; it is a 32-bit signed two's complement integer. Its default value is zero.
4. long data type:- It is a 64-bit 2's complement integer. It is used for the higher values that can not be handled by the int data type.
5. float data type :- The Float data type is used to declare the floating values (fractional numbers).
6. double data type
7. char data type
8. short data type

Non Primitive Data type :-

A non-primitive data type can be a class, interface, and Array.

2.Variable Declaration and Initialization

1.Variable Declaration

Variable declaration in Java is the process of creating a variable by specifying its type and name. A variable type defines what kind of data the variable will hold (e.g., integer, string, etc.).

Syntax:

```
dataType variableName;
```

- **`dataType`**: Specifies the type of the variable (e.g., `int`, `double`, `String`, etc.).
- **`variableName`**: Specifies the name of the variable.

Example:

```
int age;
String name;
```

2. Variable Initialization

Variable initialization is the process of assigning a value to a declared variable. Initialization is typically done at the time of declaration, but it can also happen later.

Syntax:

```
variableName = value;
```

- **`variableName`**: The name of the variable you are initializing.
- **`value`**: The value you want to assign to the variable.

Example:

```
int age = 25;
String name = "John";
```

3. Local vs. Instance vs. Class Variables

- **Local Variables**: Declared inside a method or block, and can only be used within that method or block.
- **Instance Variables**: Declared inside a class but outside of any method, constructor, or block. These variables belong to an instance of the class (an object).
- **Class Variables**: Declared using the `static` keyword, and belong to the class itself, not any specific object.

```
public class A
{
static int m=100;//static variable
 void method()
   {
          int n=90;//local variable
       }
   public static void main(String args[])
   {
      int data=50;//instance variable
   }
}
```

3.Operators: Arithmetic,Relational,Logical,Assignment,Unary,andBitwise

operators are special symbols used to perform operations on variables and values.

1. Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations.

List of Arithmetic Operators:

- `+`: Addition
- `-`: Subtraction
- `*`: Multiplication
- `/`: Division
- `%`: Modulus (remainder of a division)

Example:
```
int a = 10, b = 3;
System.out.println(a + b); // Output: 13
System.out.println(a - b); // Output: 7
System.out.println(a * b); // Output: 30
System.out.println(a / b); // Output: 3
System.out.println(a % b); // Output: 1 (remainder)
```

2. Relational (Comparison) Operators

Relational operators are used to compare two values and return a boolean result (`true` or `false`).

Theory

List of Relational Operators:

- **==**: Equal to
- **!=**: Not equal to
- **>**: Greater than
- **<**: Less than
- **>=**: Greater than or equal to
- **<=**: Less than or equal to

Example:

```
int a = 10, b = 5;
System.out.println(a == b);  // Output: false
System.out.println(a != b);  // Output: true
System.out.println(a > b);   // Output: true
System.out.println(a < b);   // Output: false
System.out.println(a >= b);  // Output: true
System.out.println(a <= b);  // Output: false
```

3. Logical Operators

Logical operators are used to perform logical operations, often used with boolean values.

List of Logical Operators:

- **&&**: Logical AND
- **||**: Logical OR
- **!**: Logical NOT

Example:

```
boolean x = true, y = false;
System.out.println(x && y);  // Output: false (AND)
System.out.println(x || y);  // Output: true (OR)
System.out.println(!x);      // Output: false (NOT)
```

4. Assignment Operators

Assignment operators are used to assign values to variables.

List of Assignment Operators:

- **=**: Simple assignment
- **+=**: Add and assign
- **-=**: Subtract and assign
- ***=**: Multiply and assign
- **/=**: Divide and assign
- **%=**: Modulus and assign

Example:

```
int a = 10;
```

```
a += 5; // a = a + 5 => 15
a -= 3; // a = a - 3 => 12
a *= 2; // a = a * 2 => 24
a /= 4; // a = a / 4 => 6
a %= 5; // a = a % 5 => 1
System.out.println(a); // Output: 1
```

5. Unary Operators

Unary operators perform operations on a single operand.

List of Unary Operators:

- **+**: Unary plus (indicates a positive value)
- **-**: Unary minus (negates the value)
- **++**: Increment (increases the value by 1)
- **--**: Decrement (decreases the value by 1)
- **!**: Logical NOT (inverts the boolean value)

Example:

```
int a = 5, b = 10;
System.out.println(+a); // Output: 5 (Unary plus, doesn't change the value)
System.out.println(-a); // Output: -5 (Unary minus)
System.out.println(++b); // Output: 11 (Pre-increment)
System.out.println(b--); // Output: 11 (Post-decrement, value is 11 before
decrementing)
System.out.println(b);   // Output: 10 (Value after decrementing)
```

6. Bitwise Operators

Bitwise operators perform operations on the binary representations of integers (bit-level operations). These operators are mostly used for low-level programming and optimization.

List of Bitwise Operators:

- **&**: Bitwise AND
- **|**: Bitwise OR
- **^**: Bitwise XOR
- **~**: Bitwise NOT (inverts the bits)
- **<<**: Left shift
- **>>**: Right shift
- **>>>**: Unsigned right shift (fills 0 on left for positive and negative numbers)

Example:

```
int a = 5, b = 3;  // Binary: 5 = 0101, 3 = 0011
System.out.println(a & b);   // Output: 1  (0101 & 0011 = 0001)
System.out.println(a | b);   // Output: 7  (0101 | 0011 = 0111)
System.out.println(a ^ b);   // Output: 6  (0101 ^ 0011 = 0110)
System.out.println(~a);      // Output: -6 (invert bits of 0101 => 1010,
which is -6 in two's complement)
```

```
System.out.println(a << 1);  // Output: 10 (Shift left by 1, 0101 becomes 
1010)
System.out.println(a >> 1);  // Output: 2  (Shift right by 1, 0101 becomes 
0010)
System.out.println(a >>> 1); // Output: 2  (Unsigned shift right, same as >> 
in this case)
```

4 Type Conversion and TypeCasting

type conversion and **type casting** refer to the process of converting one data type into another. These processes are important because in a strongly-typed language like Java, variables must be explicitly declared with a specific type.

1. Type Conversion

Type conversion refers to the process of converting a variable from one data type to another. There are two types of type conversion in Java:

1.1 Implicit Type Conversion

This happens when a smaller data type is automatically converted to a larger data type. It is done automatically by the Java compiler, and no explicit syntax is required.

Example: Converting `int` to `long`, `float` to `double`, etc.

Example:
```
int num1 = 100;
long num2 = num1;
float num3 = num2;
double num4 = num3;
System.out.println("Double value: " + num4);.
```

1.2 Explicit Type Conversion

This occurs when you convert a larger data type to a smaller one, such as from `double` to `int`. This type of conversion may result in a loss of data, so Java requires you to explicitly specify it using **casting**.

- **Example**: Converting `double` to `int`, `long` to `short`, etc.

Example:
```
double num1 = 9.87;
int num2 = (int) num1;
System.out.println("Integer value: " + num2);
```

Theory

2. Type Casting

Type casting is a way to explicitly convert a variable from one type to another. There are two types of casting in Java:

2.1 Implicit Casting

This is the automatic conversion done by Java when converting from a smaller type to a larger type. It does not require any special syntax.

- **Example**: `byte` to `short`, `short` to `int`, `int` to `long`, etc.

Example:
```
byte byteValue = 10;
int intValue = byteValue;
System.out.println("Int value: " + intValue);
```

2.2 Explicit Casting

This is when you manually convert a larger type to a smaller type.

Example: `double` to `int`, `long` to `short`, `float` to `byte`, etc.

Example:
```
double doubleValue = 9.99;
int intValue = (int) doubleValue;
System.out.println("Int value: " + intValue);
```

3. Autoboxing and Unboxing

In addition to type conversion and casting, Java provides **autoboxing** and **unboxing** mechanisms to automatically convert between primitive types and their corresponding wrapper classes

(e.g., from `int` to `Integer`, from `char` to `Character`, etc.).

- **Autoboxing**: Automatic conversion from a primitive type to its corresponding wrapper class.
- **Unboxing**: Automatic conversion from a wrapper class to its corresponding primitive type.

Example of Autoboxing:
```
int num = 10;
Integer integerObject = num;
System.out.println(integerObject);
```

Example of Unboxing:
```
Integer integerObject = 20;
int num = integerObject;
System.out.println(num);
```

3.Control Flow Statements

Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements.

1.If-elseStatements

`if-else` statements are used to execute a block of code conditionally, based on whether a certain condition evaluates to `true` or `false`.

- **`condition`**: An expression that evaluates to a boolean value (`true` or `false`).
- If the condition is `true`, the code inside the `if` block is executed.
- If the condition is `false`, the code inside the `else` block is executed.

Example:

```
int age = 18;
if (age >= 18) {
    System.out.println("You are eligible to vote.");
} else {
    System.out.println("You are not eligible to vote.");
}
```

Nested `if-else` Statements

You can also nest `if-else` statements within each other to handle more complex conditions.

Example:

```
int score = 85;
if (score >= 90) {
    System.out.println("You got an A grade.");
} else if (score >= 80) {
    System.out.println("You got a B grade.");
} else if (score >= 70) {
    System.out.println("You got a C grade.");
} else {
    System.out.println("You got a D grade.");
}
```

`if-else if-else` Ladder

The `else if` ladder is a series of `if` statements followed by an `else` block. Each `else if` condition is checked in sequence, and the first true condition is executed.

Example:

```
int temperature = 30;
if (temperature > 40) {
    System.out.println("It's extremely hot!");
} else if (temperature > 30) {
    System.out.println("It's hot!");
} else if (temperature > 20) {
    System.out.println("It's warm.");
} else if (temperature > 10) {
    System.out.println("It's cool.");
} else {
    System.out.println("It's cold.");
}
```

The `else` Clause

The `else` clause is optional. If there is no need for an alternative block of code when the condition is `false`, you can omit the `else` block.

Example:

```
int num = 5;
if (num > 0) {
    System.out.println("The number is positive.");
}
```

Ternary Operator (Conditional Operator)

Java also supports a shorthand version of the `if-else` statement called the **ternary operator**. It is often used when you need to assign a value based on a condition.

Example:

```
int age = 18;
String eligibility = (age >= 18) ? "Eligible to vote" : "Not eligible to
vote";
System.out.println(eligibility);
```

2.Switch Case Statements

`switch` statement is a control flow structure used to evaluate a variable or expression and match its value against a series of possible values. This is useful when you need to choose between multiple options based on a single condition, as it makes the code cleaner and easier to read compared to multiple `if-else if` statements.

- **`expression`**: This is the variable or value being evaluated.
- **`case valueX`**: Each `case` represents a potential value the expression can have. If the expression matches a `case`, that block of code is executed.
- **`break`**: The `break` statement is used to exit the `switch` block after a case has been executed. If `break` is omitted, the program will continue executing subsequent cases (known as **fall-through**).
- **`default`**: The `default` block is optional and is executed if none of the `case` values match the expression. It acts like the `else` in an `if-else` structure.

Example of a Simple `switch` Statement:

```
int day = 3;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    case 4:
        System.out.println("Thursday");
        break;
    case 5:
        System.out.println("Friday");
        break;
    case 6:
        System.out.println("Saturday");
        break;
    case 7:
        System.out.println("Sunday");
        break;
    default:
        System.out.println("Invalid day");
}
```

Fall-through in `switch` Statements:

In Java, if a `case` block does not contain a `break` statement, the control "falls through" to the next `case`. This is known as **fall-through behavior**.

Example of Fall-through:

```
int day = 3;
switch (day) {
    case 1:
    case 2:
    case 3:
        System.out.println("It's a weekday.");
        break;
    case 4:
    case 5:
        System.out.println("It's almost weekend.");
        break;
    default:
        System.out.println("Invalid day.");
}
```

witch with `String`:

Starting from Java 7, you can use `String` values in a `switch` statement. This is particularly useful when you need to compare strings or names.

Example with `String`:

```
String fruit = "Apple";
switch (fruit) {
    case "Apple":
        System.out.println("It's an apple.");
        break;
    case "Banana":
        System.out.println("It's a banana.");
        break;
    case "Orange":
        System.out.println("It's an orange.");
        break;
    default:
        System.out.println("Unknown fruit.");
}
```

Switch with Enum:

You can use **enum types** in a `switch` statement. Enums provide a more type-safe approach when dealing with a fixed set of constants.

Example with `Enum`:

```
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

Day day = Day.WEDNESDAY;
switch (day) {
    case MONDAY:
        System.out.println("Start of the workweek.");
        break;
    case FRIDAY:
        System.out.println("Almost weekend!");
        break;
    case WEDNESDAY:
        System.out.println("Hump day!");
        break;
    default:
        System.out.println("Regular workday.");
}
```

When to Use `switch` vs `if-else`:

- Use `switch` when you have a variable with many possible constant values (e.g., days of the week, months of the year, predefined categories, etc.). It is generally cleaner and more efficient than using multiple `if-else` statements.
- Use `if-else` when you need to check more complex conditions (e.g., ranges, or if the conditions involve relational operators).

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

loops are used to repeatedly execute a block of code as long as a specified condition is true. Java provides three primary types of loops: **for loop**, **while loop**, and **do-while loop**.

1. For Loop

The `for` loop is used when the number of iterations is known beforehand. The loop has three parts:

- **Initialization**: The loop counter is initialized.
- **Condition**: The loop continues as long as this condition is true.
- **Update**: The loop counter is updated after each iteration.

Example:

```
public class ForLoopExample {
    public static void main(String[] args) {
for (int i = 1; i <= 5; i++) {
            System.out.println(i);
        }
    }
}
```

2. While Loop

The `while` loop is used when the number of iterations is not necessarily known in advance. The loop will continue to execute as long as the condition evaluates to true. The condition is checked before each iteration.

Example:

```
public class WhileLoopExample {

 public static void main(String[] args) {

       int i = 1;

        while (i <= 5) {
            System.out.println(i);
            i++;
        }
    }
}
```

Theory

3. Do-While Loop

The `do-while` loop is similar to the `while` loop, but it guarantees that the loop will run at least once, even if the condition is false initially. The condition is checked **after** the code block is executed.

Example:

```
public class DoWhileLoopExample {
    public static void main(String[] args) {
        int i = 1;
        do {
            System.out.println(i);
            i++;
        } while (i <= 5);
    }
}
```

Key Differences:

Loop Type	Condition check	Execution Guarantee	Typical Usage
For Loop	Before each iteration	May not execute if condition is false initially	When the number of iterations is known
While Loop	Before each iteration	May not execute if condition is false initially	When the number of iterations is not known
Do-While Loop	After each iteration	Executes at least once, even if condition is false initially	When the code must run at least once

When to Use Each Loop:

- **For Loop**: Use when the number of iterations is known beforehand or can be calculated easily.
- **While Loop**: Use when the number of iterations is not known and you need to check the condition before executing the loop body.
- **Do-While Loop**: Use when the loop must execute at least once, regardless of the condition.

4.Break and Continue keywords

1. `break` Keyword

The `break` keyword is used to immediately exit from the loop or `switch` statement, regardless of the loop's condition. After the `break` is executed, the control is transferred to the statement following the loop or `switch`.

- **In loops:** When the `break` is executed inside a loop, it terminates the loop and the control moves to the statement immediately after the loop.
- **In `switch`:** When used inside a `switch` statement, `break` exits the switch block, preventing the execution of other case blocks.

Example of `break` in a loop:

```
public class BreakExample {
    public static void main(String[] args) {
        // Print numbers from 1 to 10, but exit the loop when the number is 6
        for (int i = 1; i <= 10; i++) {
            if (i == 6) {
                break; // Exit the loop when i equals 6
            }
            System.out.println(i);
        }
    }
}
```

Example of `break` in a `switch` statement:

```
public class BreakInSwitchExample {
    public static void main(String[] args) {
        int day = 3;

        switch (day) {
            case 1:
                System.out.println("Monday");
                break;
            case 2:
                System.out.println("Tuesday");
                break;
            case 3:
                System.out.println("Wednesday");
                break; // Breaks out of the switch after printing "Wednesday"
            case 4:
                System.out.println("Thursday");
                break;
            default:
                System.out.println("Invalid day");
        }
    }
}
```

2. `continue` Keyword

The `continue` keyword is used to skip the current iteration of a loop and immediately move to the next iteration. Unlike `break`, it does not terminate the loop, but it allows you to skip the remaining code inside the current iteration and proceed to the next cycle of the loop.

- **In loops:** When the `continue` statement is executed, it skips the current iteration and checks the loop condition to determine whether to proceed with the next iteration.

Example of `continue` in a loop:

```
public class ContinueExample {
    public static void main(String[] args) {
        // Print numbers from 1 to 10, skipping 5
        for (int i = 1; i <= 10; i++) {
            if (i == 5) {
                continue;            }
            System.out.println(i);
        }
    }
}
```

Example of `continue` in a `while` loop:

```
public class ContinueInWhileExample {
    public static void main(String[] args) {
      int i = 1;
        while (i <= 5) {
            if (i == 3) {
                i++; // Move to the next iteration
                continue; // Skip printing 3
            }
            System.out.println(i);
            i++;
        }
    }
}
```

Key Differences Between `break` and `continue`:

- **`break`**:
  - Terminates the entire loop or `switch` statement.
  - Control is transferred to the code immediately after the loop or `switch`.
- **`continue`**:
  - Skips the current iteration and proceeds to the next iteration of the loop.
  - Does not terminate the loop, just jumps to the next iteration.

Example with both `break` and `continue`:

```
public class BreakContinueExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            if (i == 4) {
                continue; // Skip 4
            }
            if (i == 8) {
                break; // Exit the loop when i equals 8
            }
            System.out.println(i);
        }
    }
}
```

- Use `break` when you need to exit a loop or switch statement early.
- Use `continue` when you want to skip the current iteration of a loop but continue the loop.

5.Methods in Java

1. Defining Methods

methods are blocks of code that perform a specific task. Methods are used to define reusable code that can be invoked by other parts of the program. Java allows you to define methods with or without parameters and return types.

Method Definition Syntax:

```
returnType methodName(parameters) {
    // Method body: Code to be executed
}
```

- **returnType**: Specifies the type of value the method will return (e.g., `int`, `String`, `void`). If the method does not return a value, `void` is used.
- **methodName**: The name of the method. It should be descriptive and follow Java naming conventions.
- **parameters**: The values or objects the method will use for its operation, enclosed in parentheses. A method can have zero or more parameters. If there are no parameters, it's left empty (`()`).

  Four type of the method

1. Method with a Return Type

A method that returns a value needs to specify a return type. If the method returns no value, it uses `void`.

Example: A method that adds two numbers and returns the result:

```
public class MethodExample {
public static int add(int a, int b) {
        int sum = a + b;
        return sum;
    }

    public static void main(String[] args) {
        int result = add(5, 3);
        System.out.println("The sum is: " + result);
    }
}
```

2. Void Methods (Methods Without a Return Type)

If a method doesn't return any value, its return type is specified as `void`. These methods are generally used for performing actions, like printing messages or modifying data.

Example: A method that prints a message without returning anything:

```
public class VoidMethodExample {
    // Method with void return type that prints a message
    public static void printMessage(String message) {
        System.out.println(message);
    }

    public static void main(String[] args) {
        // Calling the printMessage method
        printMessage("Hello, Java!");
    }
}
```

3. Method Overloading

Java allows multiple methods to have the same name but with different parameters (i.e., **method overloading**). The methods can differ by the number or type of their parameters.

Example: Overloading the `add` method:

```
public class MethodOverloadingExample {
    // Method to add two integers
    public static int add(int a, int b) {
        return a + b;
    }

    // Overloaded method to add three integers
    public static int add(int a, int b, int c) {
        return a + b + c;
    }

    public static void main(String[] args) {
        // Calling overloaded methods
        int result1 = add(5, 3); // Calls the first method
        int result2 = add(1, 2, 3); // Calls the overloaded method
        System.out.println("The sum of two numbers is: " + result1);
        System.out.println("The sum of three numbers is: " + result2);
    }
}
```

4. Method Invocation

Methods are invoked (called) using the following syntax:

```
methodName(arguments);
```

Theory

5. Passing Arguments to Methods

Java methods can accept arguments (parameters) that are passed when the method is called. These arguments are used within the method.

Example: Method with parameters:

```
public class MethodWithParametersExample {
    // Method with two parameters
    public static void greet(String name, int age) {
        System.out.println("Hello " + name + ", you are " + age + " years
old.");
    }

    public static void main(String[] args) {
        // Calling the greet method with arguments
        greet("Alice", 30);
        greet("Bob", 25);
    }
}
```

6. Return Statement

If a method has a return type (other than `void`), it must use the `return` statement to return a value. The type of the value must match the method's declared return type.

Example: A method returning a `double`:

```
public class ReturnExample {
    // Method that calculates the area of a circle
    public static double calculateArea(double radius) {
        return Math.PI * radius * radius;
    }

    public static void main(String[] args) {
        double area = calculateArea(5.0); // Passing argument to the method
        System.out.println("The area of the circle is: " + area);
    }
}
```

7. Method Scope and Lifetime

- **Local variables**: Variables defined inside a method are local to that method. They are created when the method is invoked and destroyed when the method exits.
- **Instance variables**: If a method refers to instance variables (variables declared at the class level), they can be accessed and modified by any method in the class.Bottom of Form

2.Method Parameters and ReturnTypes

Method Parameters and **Return Types** are critical elements when defining and calling methods.

1. Method Parameters in Java

Method parameters are variables that are defined in the method signature to accept input values when the method is called.

- **Method Parameters**: Are variables in the method signature that allow the method to accept values when called. They can be primitive or reference types.

Key Points:

- **Formal Parameters**: These are the parameters defined in the method's declaration. They specify the type and order of the arguments the method will accept.
- **Actual Parameters (Arguments)**: These are the actual values or variables passed to the method when it is called.

Example:

```
public class Calculator {

    // Method with parameters
    public int add(int num1, int num2) {
        return num1 + num2;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int result = calc.add(5, 7);
        System.out.println("Sum: " + result);
    }
}
```

Types of Parameters in Java:

1. **Primitive Data Types**: These include types like `int`, `float`, `char`, etc.
2. **Reference Data Types**: These include objects such as `String`, `Array`, and custom class objects.

Parameter Passing in Java:

Java uses **pass-by-value** for both primitive and reference data types:

- For **primitive types** (like `int`, `float`, `boolean`), the method gets a copy of the value.
- For **reference types** (like `String`, Arrays, Objects), the method gets a copy of the reference, so modifications to the object inside the method will affect the original object.

2. Return Types in Java

The **return type** of a method defines the type of value that the method will return to the caller once it completes execution.

Return Type: Specifies the type of value that the method will return to the caller. If no value is returned, the return type is `void`.

Key Points:

- **Void Return Type**: If a method does not return any value, you declare it with the `void` keyword.
- **Non-Void Return Type**: A method that returns a value must specify the type of the value it will return (e.g., `int`, `String`, `boolean`, custom object types).

Example:

```
public class Calculator {

public int multiply(int num1, int num2) {
        return num1 * num2;
    }

  public void printHello() {
        System.out.println("Hello, world!");
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int result = calc.multiply(3, 4);
System.out.println("Product: " + result);
calc.printHello();      }
}
```

3.Method Overloading

Method Overloading in Java is a feature that allows you to define multiple methods in the same class with the **same name** but **different parameters** (either in number, type, or both). This is a form of **compile-time polymorphism** (also known as static polymorphism) because the method to be invoked is determined at compile time based on the method signature.

Key Points about Method Overloading:

1. **Same Method Name**: All overloaded methods must have the same name.
2. **Different Parameters**: The methods must differ in the number, type, or order of parameters.
3. **Return Type Can Vary**: Although the return type can be different, it is **not sufficient alone** to distinguish overloaded methods. The parameter list must differ.
4. **No Overloading Based on Return Type**: Java does not support method overloading if the only difference is the return type. The parameters must be different.

Theory

Advantages of Method Overloading:

- **Improved Readability**: You can use the same method name to perform similar tasks with different inputs.
- **Code Reusability**: It allows you to reuse the method name for methods performing similar operations, reducing the need for creating multiple method names.
- **Increased Flexibility**: You can perform different operations based on the number or types of arguments passed.

Example of Method Overloading in Java

Overloading Based on the Number of Parameters:

```
public class MathOperations {

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

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

    public static void main(String[] args) {
        MathOperations math = new MathOperations();
        System.out.println("Sum of two numbers: " + math.add(5, 7));
System.out.println("Sum of three numbers: " + math.add(5, 7, 3));      }
}
```

Overloading Based on the Type of Parameters:

```
public class Display {

    // Method that takes an integer
    public void show(int a) {
        System.out.println("Integer: " + a);
    }

    // Overloaded method that takes a string
    public void show(String s) {
        System.out.println("String: " + s);
    }

    // Overloaded method that takes a double
    public void show(double d) {
        System.out.println("Double: " + d);
    }

    public static void main(String[] args) {
        Display display = new Display();
        display.show(5);       // Calls show(int)
        display.show("Hello"); // Calls show(String)
        display.show(3.14);    // Calls show(double)
    }
}
```

Theory

Overloading Based on the Order of Parameters:

```
public class Calculator {


 public int subtract(int a, int b) {
        return a - b;
    }

    public int subtract(double a, double b) {
        return (int)(a - b);
    }

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

Example with varargs:

```
public class Printer {

    // Method that prints multiple integers using varargs
    public void printNumbers(int... numbers) {
        for (int num : numbers) {
            System.out.println(num);
        }
    }

    public static void main(String[] args) {
        Printer printer = new Printer();
        printer.printNumbers(1, 2, 3);  // Passes three integers
        printer.printNumbers(5, 10, 15, 20);  // Passes four integers
    }
}The printNumbers(int... numbers) method allows you to pass any number of
integers, including none.
```

- This method demonstrates **varargs** in method overloading, where you can pass multiple arguments without specifying the exact number of them.

4.Static Methods and Variables

`static` is a keyword used to declare methods and variables that belong to the **class** rather than to any instance (object) of the class. This means that static members can be accessed directly via the class name, without the need for creating an object of the class. Static members are shared among all instances of the class.

Key Characteristics of Static Methods and Variables:

- **Belong to the Class**: Static methods and variables are shared by all instances of the class.
- **Accessed Without Creating an Object**: They can be accessed directly using the class name.
- **Can Be Accessed by Object Instances**: Although static members belong to the class, they can also be accessed through objects, though it's not recommended because it can be misleading.
- **Can Be Used Without Instantiating the Class**: Static members can be used without creating an instance of the class.

Static Variables

A **static variable** is a variable that is shared by all instances (objects) of the class. There is only **one copy** of the static variable for the entire class, regardless of how many objects are created from the class.

Characteristics of Static Variables:

- **Shared Across All Instances**: A static variable is shared by all objects of the class, meaning any change to a static variable reflects across all instances.
- **Memory Efficiency**: Since static variables are shared, they use less memory compared to instance variables, which are stored separately for each object.
- **Initialized Only Once**: Static variables are initialized only once when the class is loaded, and they maintain their state across method calls and objects.

Example of Static Variable:

```
public class Counter {
   static int count = 0;

  public Counter() {
        count++;
    }

   public static void displayCount() {
        System.out.println("Count: " + count);
    }

    public static void main(String[] args) {
        // Creating instances of the Counter class
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        Counter c3 = new Counter();
        // Displaying the count using the static method
        Counter.displayCount();  // Output: Count: 3
    }
}
```

Static Methods

A **static method** is a method that belongs to the class rather than to any object. Static methods can be called without creating an instance of the class. Static methods can access only **static variables** and **static methods**.

Characteristics of Static Methods:

- **No Access to Instance Variables/Methods**: A static method cannot access instance variables and instance methods directly because it doesn't have an instance of the class.
- **Can Be Called Using the Class Name**: Static methods are typically called using the class name, but they can also be called via objects, though this is not recommended.
- **Useful for Utility Methods**: Static methods are often used for utility or helper methods, like mathematical calculations or factory methods.

Example of Static Method:

```
public class MathUtils {


public static int square(int number) {
        return number * number;
    }

public static int cube(int number) {
        return number * number * number;
    }

    public static void main(String[] args) {
        int result1 = MathUtils.square(4);  // Output: 16
        int result2 = MathUtils.cube(3);    // Output: 27

        System.out.println("Square of 4: " + result1);
        System.out.println("Cube of 3: " + result2);
    }
}
```

Static Blocks

A **static block** (also known as a static initialization block) is used for **initializing static variables** when the class is loaded into memory. Static blocks are executed only once when the class is loaded, and they are often used for **complex static initialization**.

Example of Static Block:

```
public class Initialization {    // Static variable
```

Theory

```
    static int value;

    // Static block for initialization
    static {
        value = 10;  // Initializing the static variable
        System.out.println("Static block executed!");
    }

    public static void main(String[] args) {
        System.out.println("Value: " + value);  // Output: Value: 10
    }
}
```

When to Use Static Methods and Variables:

- **Static Variables**: When you need a variable that should be shared across all instances of a class (e.g., counters, configuration settings, constants).
- **Static Methods**: When you need utility methods that don't depend on the object state (e.g., mathematical functions, factory methods, logging).
- **Static Blocks**: When you need to initialize static variables in a more complex way or with specific logic when the class is loaded.

.

6.Object-OrientedProgramming(OOPs)Concepts

1. BasicsofOOP:

Encapsulation,Inheritance,Polymorphism,Abstraction

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes to structure software programs. Java is one of the most popular languages that uses OOP principles. Here are the basic concepts of OOP in Java:

1. Class
2. Object
3. Encapsulation
4. Inheritance
5. Polymorphism
6. Abstraction

1.Class

Collection of objects is called class. It is a logical entity.A class can also be defined as a blueprint from which you can create an individual object. Class does not consume any space.

2.Object

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

3.Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

This is typically done using **private** fields and **public** methods to get and set the field values.

4.Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

a class can inherit from another class using the `extends` keyword.

5.Polymorphism

If *one task is performed in different ways*, it is known as polymorphism.

For example: to convince the customer differently, to draw something,shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism.

- **Compile-time Polymorphism (Method Overloading)**: Same method name but different parameters.
- **Runtime Polymorphism (Method Overriding)**: A subclass provides a specific implementation for a method already defined in its superclass.

6.Abstraction

Hiding internal implementation and showing functionality only to the user is known as abstraction. For example, phone call, we do not know the internal processing.

This is achieved through abstract classes and interfaces.

- **Abstract class**: A class that cannot be instantiated on its own and may contain abstract methods (methods without implementation).
- **Interface**: A contract that a class can implement. It defines methods that must be implemented by the class.

Constructor

A constructor is a special method used to initialize objects. It is called when an object of a class is created. Constructors have the same name as the class and do not have a return type.

2. Inheritance:

Single,Multilevel,Hierarchical

Inheritance in Java is a fundamental concept of object-oriented programming (OOP) that allows one class (called the **subclass** or **child class**) to inherit the properties and behaviors (fields and methods) of another class (called the **superclass** or **parent class**). This facilitates code reusability and method overriding, and helps in creating a hierarchical structure.

Key Concepts of Inheritance:

1. **Super Class (Parent Class)**: The class whose properties and methods are inherited by other classes.
2. **Sub Class (Child Class)**: The class that inherits properties and methods from another class.
3. **Method Overriding**: A subclass can provide its own specific implementation of a method that is already defined in the superclass.

Types of Inheritance in Java:

1. **Single Inheritance**
2. **Multilevel Inheritance**
3. **Hierarchical Inheritance**
4. **Multiple Inheritance** (Not supported directly in Java but can be achieved using interfaces)
5. **Hybrid Inheritance** (Not directly supported, but possible with interfaces)

1. Single Inheritance:

In **Single Inheritance**, a class inherits from just one superclass. This is the simplest form of inheritance in Java.

Example:

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

// Subclass
class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}

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

Theory

2. Multilevel Inheritance:

In **Multilevel Inheritance**, a class is derived from another class, which itself is derived from another class. In this way, a chain of inheritance is formed.

Example:

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

// Parent class
class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}

// Child class
class Puppy extends Dog {
    void play() {
        System.out.println("Playing...");
    }
}

public class Main {
    public static void main(String[] args) {
        Puppy puppy = new Puppy();
        puppy.eat();  // Inherited from Animal
        puppy.bark(); // Inherited from Dog
        puppy.play(); // Own method in Puppy
    }
}
```

3. Hierarchical Inheritance:

In **Hierarchical Inheritance**, multiple subclasses inherit from a single superclass. This allows different subclasses to share common functionality from the superclass.

Example:

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

// Subclass 1
class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}

// Subclass 2
class Cat extends Animal {
    void meow() {
        System.out.println("Meowing...");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Inherited method
        dog.bark(); // Dog-specific method

        Cat cat = new Cat();
        cat.eat();  // Inherited method
        cat.meow(); // Cat-specific method
    }
}
```

4. Multiple Inheritance (via Interfaces):

Java **does not support multiple inheritance** directly (i.e., a class cannot extend more than one class). However, it allows a class to **implement multiple interfaces**. This achieves multiple inheritance by using interfaces.

Example:

```
// First Interface
interface Animal {
    void eat();
}

// Second Interface
interface Pet {
    void play();
}

// Class implementing multiple interfaces
class Dog implements Animal, Pet {
    public void eat() {
        System.out.println("Eating...");
    }

    public void play() {
        System.out.println("Playing...");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Implemented from Animal interface
        dog.play(); // Implemented from Pet interface
    }
}
```

5. Constructor Inheritance:

Constructors are **not inherited** in Java, but the constructor of the superclass can be called in the subclass using the `super()` keyword. The subclass can also define its own constructor.

Example:

```
class Animal {
    Animal() {
        System.out.println("Animal is created");
    }
}

// Subclass
class Dog extends Animal {
    Dog() {
        super();  // Calls constructor of Animal
        System.out.println("Dog is created");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
    }
}
```

6. Accessing Superclass Members Using `super` Keyword:

The `super` keyword can be used to access members (fields and methods) of the superclass.

Example:

```
class Animal {
    String name = "Animal";
}

class Dog extends Animal {
    String name = "Dog";

    void display() {
        System.out.println("Name in subclass: " + name);
        System.out.println("Name in superclass: " + super.name);  // Access
superclass field using super
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.display();
    }
}
```

3. MethodOverriding and DynamicMethodDispatch

Method Overriding in Java

Method Overriding is a feature in Java that allows a subclass to provide a specific implementation for a method that is already defined in its superclass. The method in the subclass must have the same signature (name, return type, and parameters) as the method in the superclass.

Method Overriding allows a subclass to provide a specific implementation for a method that is already defined in its superclass.

Key points to remember about **Method Overriding**:

- **Same method signature**: The overridden method in the subclass must have the same name, return type, and parameters as the method in the superclass.
- **Inheritance**: The subclass inherits the method from the superclass but provides its own version of the method.
- **Use of `@Override` annotation**: While not mandatory, it's good practice to use the `@Override` annotation to indicate that a method is intended to override a superclass method. This helps catch errors if the method doesn't correctly override a superclass method.
- **Access modifiers**: The access level of the overridden method can be the same or more permissive than the superclass method but cannot be more restrictive. For example, a `protected` method in the superclass can be overridden as `protected` or `public` in the subclass but not as `private`.

Example of Method Overriding:

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

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

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Animal myDog = new Dog(); // Upcasting Dog to Animal

        myAnimal.sound();  // Outputs: Animal makes a sound
        myDog.sound();     // Outputs: Dog barks
    }
}
```

Dynamic Method Dispatch in Java

Dynamic Method Dispatch is a mechanism in Java where a method call is resolved at runtime, not at compile-time. It is the process by which Java uses the actual object type (not the reference type) to determine which overridden method to invoke.

Dynamic method dispatch is crucial for achieving **runtime polymorphism** in Java. When a method is called on an object, Java decides which method to invoke based on the actual object (and not the type of the reference variable).

Dynamic Method Dispatch is the process by which Java decides at runtime which method to call based on the actual object type, not the reference type.

Together, these concepts enable **runtime polymorphism**, where the method to be invoked is determined dynamically, allowing for flexible and extensible code.

How Dynamic Method Dispatch Works:

- At runtime, Java determines which version of the overridden method should be called based on the actual object type, not the reference type.
- If you use **upcasting** (e.g., assigning a `Dog` object to an `Animal` reference), Java will call the overridden method in the `Dog` class at runtime.

Key Points of Dynamic Method Dispatch:

- **Polymorphism**: Dynamic method dispatch is a fundamental aspect of polymorphism in Java. It allows objects of different types to be treated uniformly while still behaving in a way that is specific to their actual class.
- **At runtime**: The method that is executed is determined by the object type, not the reference type.

Example with Dynamic Method Dispatch:

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

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

class Cat extends Animal {
    void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Dog();  // Upcasting
        Animal animal2 = new Cat();  // Upcasting

        animal1.sound();  // Outputs: Dog barks
        animal2.sound();  // Outputs: Cat meows
    }
}
```

7. Constructors and Destructors

1. ConstructorTypes (Default,Parameterized)

constructors are special methods used to initialize objects. They are called when an object of a class is created.

There are two main types of constructors in Java.

1..**Default Constructors**

2 **Parameterized Constructors**.

1. Default Constructor

A **default constructor** is a constructor that does not take any parameters. If you do not explicitly define any constructor in a class, the Java compiler automatically provides a default constructor that initializes the object with default values (e.g., `0` for numeric types, `null` for objects, etc.).

If you define a constructor with no parameters explicitly, it's called a **no-argument constructor**.

Example of Default Constructor:

```
class Car {
    String brand;
    int year;

    // Default constructor
    public Car() {
        // Initialize with default values
        brand = "Unknown";
        year = 2000;
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object using the default constructor
        Car car = new Car();
        System.out.println("Brand: " + car.brand);  // Output: Unknown
        System.out.println("Year: " + car.year);    // Output: 2000
    }
}
```

2. Parameterized Constructor

A **parameterized constructor** is a constructor that allows you to provide specific values to the object at the time of creation. This constructor takes parameters and provides more flexibility by allowing different values to be passed when creating objects.

Example of Parameterized Constructor:

```
class Car {
    String brand;
    int year;

    // Parameterized constructor
    public Car(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object using the parameterized constructor
        Car car = new Car("Toyota", 2022);
        System.out.println("Brand: " + car.brand);  // Output: Toyota
        System.out.println("Year: " + car.year);    // Output: 2022
    }
}
```

2. CopyConstructor(EmulatedinJava)

a **copy constructor** is a constructor that creates a new object as a copy of an existing object. Although Java does not provide a built-in copy constructor (unlike languages like C++), you can easily implement one in your class. The copy constructor typically takes another instance of the same class as an argument and copies its field values to the new object.

Syntax of Copy Constructor in Java

```
public class ClassName {
    // Fields
    private int field1;
    private String field2;

    // Default constructor
    public ClassName() {
        field1 = 0;
        field2 = "";
    }

    // Parameterized constructor
    public ClassName(int field1, String field2) {
        this.field1 = field1;
        this.field2 = field2;
    }

    // Copy constructor
    public ClassName(ClassName other) {
        this.field1 = other.field1;
        this.field2 = other.field2;
    }

    // Getter methods
    public int getField1() {
        return field1;
    }

    public String getField2() {
        return field2;
    }
}
```

Usage Example

```
public class Main {
    public static void main(String[] args) {
        // Create an original object
        ClassName original = new ClassName(10, "Hello");

        // Create a copy of the original object using the copy constructor
        ClassName copy = new ClassName(original);

        // Print fields of the copied object
        System.out.println("Field1: " + copy.getField1());  // Output: 10
        System.out.println("Field2: " + copy.getField2());  // Output: Hello
    }
}
```

Key Points to Remember:

1. The copy constructor is typically used to perform a deep copy or shallow copy, depending on how the fields are copied (e.g., for reference types, you may need to clone the object or make a new instance).
2. The default behavior in Java for copying objects involves a shallow copy, meaning if the class contains references to other objects, those references are copied as-is. If you need a deep copy, you must explicitly clone or copy the referenced objects inside the copy constructor.

3. Java also has the `clone()` method for object cloning, but the copy constructor can offer more control and clarity in some cases.

This is a basic implementation, and it can be adapted depending on your needs (e.g., copying more complex objects, handling collections, etc.).

3. ConstructorOverloading

Constructor overloading in Java is the concept of defining multiple constructors with the same name but different parameter lists. This allows you to create objects in different ways, providing flexibility in object initialization. The constructors must differ in the type, number, or order of parameters.

Key Points:

1. **Same Name**: All overloaded constructors must have the same name as the class.
2. **Different Parameter Lists**: The difference in constructors can be in the number of parameters, type of parameters, or both.
3. **No Return Type**: Constructors do not have a return type (not even `void`).

Example:

```
class Vehicle {
    String model;
    int year;

    // Default constructor (no parameters)
    public Vehicle() {
        model = "Unknown";
        year = 0;
    }

    // Constructor with one parameter (model)
    public Vehicle(String model) {
        this.model = model;
        this.year = 2024; // default year
    }

    // Constructor with two parameters (model and year)
    public Vehicle(String model, int year) {
        this.model = model;
        this.year = year;
    }

    // Display method to show vehicle information
    public void display() {
        System.out.println("Model: " + model + ", Year: " + year);
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating objects with different constructors
        Vehicle vehicle1 = new Vehicle(); // Calls the default constructor
        Vehicle vehicle2 = new Vehicle("Toyota"); // Calls constructor with 
one parameter
        Vehicle vehicle3 = new Vehicle("Honda", 2022); // Calls constructor 
with two parameters

        // Displaying the details
        vehicle1.display(); // Model: Unknown, Year: 0
        vehicle2.display(); // Model: Toyota, Year: 2024
        vehicle3.display(); // Model: Honda, Year: 2022
    }
}
```

How It Works:

- **Default Constructor**: When no arguments are passed, the default constructor (`Vehicle()`) is invoked.
- **Constructor with One Parameter**: When only the model name is provided, the constructor (`Vehicle(String model)`) is called, setting the model and defaulting the year.
- **Constructor with Two Parameters**: When both the model and the year are passed, the constructor (`Vehicle(String model, int year)`) is used.

Benefits:

- **Flexibility**: You can create objects with different sets of initial data.
- **Code Readability**: Constructor overloading can make the code more readable and maintainable by allowing initialization in various ways.

Rules:

1. Constructor overloading doesn't involve return types.
2. A constructor can have any number of overloaded versions, as long as the parameter lists differ.
3. You can call one constructor from another using `this()`. This is useful to avoid redundancy.

4. Object LifeCycle and GarbageCollection

Object Life Cycle in Java

In Java, the life cycle of an object refers to the stages an object goes through from its creation to its destruction. This life cycle is managed by the Java Virtual Machine (JVM) and is integral to memory management. Here's an overview of the stages:

1. **Object Creation**:
    - An object in Java is created using the `new` keyword. The creation involves:
        - Memory allocation for the object.
        - Initializing the object through the constructor.

- Example: MyClass obj = new MyClass();

2. **Object Usage**:
   - Once created, the object is used as per the program's logic. It can have methods called on it, and its fields can be accessed or modified.
   - Example: `obj.someMethod(); obj.someField = 10;`
3. **Object Dereferencing**:
   - An object is dereferenced when there are no longer any references pointing to it. This means that the object is no longer accessible from the program, but it still exists in memory until the garbage collector reclaims it.
4. **Garbage Collection**:
   - The object becomes eligible for garbage collection when there are no references pointing to it, meaning it is no longer reachable by any part of the program.
5. **Object Destruction**:
   - When the garbage collector identifies an object that is no longer in use (i.e., has no references), it will remove the object from memory, allowing the memory to be reused.

Garbage Collection in Java

Garbage collection is the process by which Java automatically reclaims memory used by objects that are no longer reachable. The Java garbage collector (GC) runs in the background and automatically frees up memory, helping to prevent memory leaks and improve the efficiency of memory management.

How Garbage Collection Works:

1. **Finalization**:
- Before an object is garbage collected, the `finalize()` method can be called. This method allows an object to release resources (e.g., file handles, database connections) before being destroyed.
- Example:

```
protected void finalize() throws Throwable {

    // Code to release resources
}
```

- However, relying on `finalize()` is discouraged because it introduces unpredictability in when the method is called.

2. **Types of Garbage Collectors in Java**:
- **Serial Garbage Collector**: Uses a single thread for garbage collection. Suitable for small applications with a single processor.
- **Parallel Garbage Collector**: Uses multiple threads to perform the garbage collection, improving performance in multi-processor environments.
- **Concurrent Mark-Sweep (CMS) Collector**: Aims to minimize application pauses by doing most of the work concurrently with the application threads.
- **G1 Garbage Collector**: Divides the heap into regions and performs garbage collection in parallel and incrementally. Suitable for large heap sizes.

8. Arrays and Strings

arrays and **strings** are fundamental data structures that allow you to store and manipulate sequences of data.

1.Array in java

An **array** is a collection of elements of the same type stored in contiguous memory locations. Arrays are used when you need to store multiple values of the same type.

2. **Strings in Java**

In Java, **strings** are objects of the `String` class. They represent sequences of characters and are immutable, meaning once created, they cannot be modified.

. Arrays vs Strings

- **Arrays** are mutable (you can change their contents) while **Strings** are immutable (you cannot modify a string after it is created).
- **Arrays** can store any type of data (primitive or object), whereas **Strings** specifically store sequences of characters.
- **arrays** and **strings** are fundamental data structures that allow you to store and manipulate sequences of data. **arrays** and **strings** are fundamental data structures that allow you to store and manipulate sequences of data. **arrays** and **strings** are fundamental data structures that allow you to store and manipulate sequences of data. **arrays** and **strings** are fundamental data structures that allow you to store and manipulate sequences of data. **arrays** and **strings** are fundamental data structures that allow you to store and manipulate sequences of data.

1.OneDimensional and Multidimensional Arrays

arrays are used to store multiple values of the same type in a single variable. These values are stored in contiguous memory locations and can be accessed using an index. Java supports both **one-dimensional** and **multidimensional** arrays.

1. One-Dimensional Arrays:

A one-dimensional array is essentially a list of elements of the same type. It can be thought of as a row of elements that can be accessed using a single index.

Example of One-Dimensional Array:

```
public class OneDimensionalArray {
    public static void main(String[] args) {
        // Declaration and initialization of an array
        int[] numbers = {1, 2, 3, 4, 5};
        System.out.println("First element: " + numbers[0]);
        System.out.println("Second element: " + numbers[1]);

        System.out.println("Array elements: ");
        for (int i = 0; i < numbers.length; i++) {
            System.out.println(numbers[i]);
        }
    }
}
```

2. Multidimensional Arrays:

A multidimensional array is an array of arrays, meaning it can store more than one row or column of elements. The most common type of multidimensional array is the **two-dimensional array** (which can be thought of as a table or matrix), but Java allows for arrays with more than two dimensions as well.

Example of Two-Dimensional Array:

```
public class TwoDimensionalArray {
    public static void main(String[] args) {
        // Declaration and initialization of a 2D array
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        // Accessing elements of the 2D array
        System.out.println("Element at position (0, 0): " + matrix[0][0]);
        System.out.println("Element at position (2, 1): " + matrix[2][1]);

        // Looping through the 2D array
        System.out.println("2D Array elements: ");
        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();
        }
    }}
```

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

strings are an essential part of programming, and there are different ways to handle and manipulate strings. The main classes for working with strings in Java are `String`, `StringBuffer`, and `StringBuilder`. Let's break down each of these classes:

1. String Class

The `String` class is the most commonly used class for handling strings in Java. Strings in Java are immutable, meaning once a string object is created, its value cannot be changed.

Theory

Key Features:

- **Immutability**: Any operation that modifies a `String` creates a new `String` object rather than modifying the original one.
- **Performance**: Since `String` objects are immutable, Java uses a string pool to optimize memory usage by reusing string literals.
- **Methods**:
  - `length()`: Returns the length of the string.
  - `charAt(int index)`: Returns the character at the specified index.
  - `concat(String str)`: Concatenates the specified string to the end of the current string.
  - `substring(int start, int end)`: Returns a new string that is a substring of the original string.
  - `equals(Object obj)`: Compares the string with another object for equality.
  - `toUpperCase()`, `toLowerCase()`: Converts the string to upper or lower case.

Example:

```
String str = "Hello";
String str2 = str.concat(" World");
System.out.println(str2);  // Output: Hello World
```

Performance Consideration:

Since `String` objects are immutable, if you're performing many concatenations, it can lead to inefficient memory usage because new `String` objects are created with each operation. This is where `StringBuffer` and `StringBuilder` come in.

2. StringBuffer Class

`StringBuffer` is a mutable sequence of characters. It is specifically designed for scenarios where a string undergoes frequent modification, such as appending or inserting characters. Unlike `String`, `StringBuffer` can modify its content without creating new objects.

Key Features:

- **Mutability**: You can change the contents of the `StringBuffer` object directly.
- **Thread-Safety**: `StringBuffer` methods are synchronized, meaning it is thread-safe. This makes it slower than `StringBuilder` in scenarios where thread safety is not required.
- **Performance**: `StringBuffer` is more efficient than `String` when it comes to appending or modifying strings in a loop because it avoids creating new objects every time.

Methods:

- `append(String str)`: Appends the specified string to the current `StringBuffer`.
- `insert(int offset, String str)`: Inserts the specified string at the given index.
- `delete(int start, int end)`: Deletes characters from the `StringBuffer`.
- `reverse()`: Reverses the contents of the `StringBuffer`.

Example:

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

Performance Consideration:

`StringBuffer` is thread-safe due to synchronization, but this synchronization comes at the cost of performance in single-threaded scenarios. If thread-safety is not a concern, `StringBuilder` is a better choice.

3. StringBuilder Class

`StringBuilder` is very similar to `StringBuffer` but without the synchronized methods, making it more efficient for use in single-threaded applications.

Key Features:

- **Mutability**: Like `StringBuffer`, `StringBuilder` is mutable and allows modification of its contents.
- **Not Thread-Safe**: `StringBuilder` is not synchronized, meaning it is not thread-safe. However, it performs better than `StringBuffer` when thread safety is not a requirement.
- **Performance**: `StringBuilder` is faster than `StringBuffer` when string modifications are done in a single-threaded environment because there is no overhead from synchronization.

Methods:

Similar to `StringBuffer`, such as `append()`, `insert()`, `delete()`, and `reverse()`.

Example:

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

Performance Consideration:

`StringBuilder` is ideal for string manipulation when you do not need synchronization, such as in most single-threaded applications.

Comparison Between `String`, `StringBuffer`, and `StringBuilder`

Feature	`String`	`StringBuffer`	`StringBuilder`
Mutability	Immutable (cannot be changed)	Mutable (can modify the string)	Mutable (can modify the string)
Thread	Not applicable	Thread-safe (synchronized)	Not thread-safe

Feature	String	StringBuffer	StringBuilder
Safety	(immutable)		
Performance	Slower for frequent modifications	Slower (due to synchronization)	Faster (no synchronization)
Use Case	When the string is not modified	When modifications are frequent and thread-safety is needed	When modifications are frequent and thread-safety is not required

---

3. ArrayofObjects

an `Array of Objects` refers to an array where each element of the array is a reference to an object. Objects can be instances of any class (user-defined or built-in). These objects are created dynamically, and the array stores references to them.

Example: Array of Objects

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

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

    // Method to display information
    void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an array of Person objects
        Person[] people = new Person[3];

        // Initialize each object in the array
        people[0] = new Person("John", 25);
        people[1] = new Person("Alice", 30);
        people[2] = new Person("Bob", 22);

        // Accessing and displaying information of each person object in the
array
        for (int i = 0; i < people.length; i++) {
            people[i].displayInfo();
            }
        }
}
```

Important Points:

- The elements of the array are references to objects. This means you can assign new objects to specific positions in the array, and the array holds references to these objects.
- The size of the array is fixed once it's created, but you can modify the objects within the array.
- Arrays can hold objects of any class type, including user-defined classes (like `Person` in this example).

---

4. StringMethods(length,charAt,substring,etc.)

`String` class provides several useful methods for manipulating and querying string data. Below is a list of commonly used `String` methods, such as `length()`, `charAt()`, `substring()`, and others, with explanations and examples.

1. length()

- **Purpose**: Returns the length of the string (i.e., the number of characters in it).
- **Example**:

```
String str = "Hello, World!";
int len = str.length();
System.out.println(len);  // Output: 13
```

2. charAt(int index)

- **Purpose**: Returns the character at the specified index in the string.
- **Example**:

```
String str = "Hello, World!";
char ch = str.charAt(7);
System.out.println(ch);  // Output: W
```

3. substring(int beginIndex)

- **Purpose**: Returns a new string that starts from the specified index to the end of the original string.
- **Example**:

```
String str = "Hello, World!";
String subStr = str.substring(7);
System.out.println(subStr);  // Output: World!
```

4. substring(int beginIndex, int endIndex)

- **Purpose**: Returns a new string that starts at the specified `beginIndex` and ends just before `endIndex`.
- **Example**:

```
String str = "Hello, World!";
String subStr = str.substring(0, 5);
System.out.println(subStr);  // Output: Hello
```

5. indexOf(int ch)

- **Purpose**: Returns the index of the first occurrence of the specified character.
- **Example**:

```
String str = "Hello, World!";
int index = str.indexOf('o');
System.out.println(index);  // Output: 4
```

6. indexOf(String str)

- **Purpose**: Returns the index of the first occurrence of the specified substring.
- **Example**:

```
String str = "Hello, World!";
int index = str.indexOf("World");
System.out.println(index);  // Output: 7
```

7. lastIndexOf(int ch)

- **Purpose**: Returns the index of the last occurrence of the specified character.
- **Example**:

```
String str = "Hello, World!";
int lastIndex = str.lastIndexOf('o');
System.out.println(lastIndex);  // Output: 8
```

8. lastIndexOf(String str)

- **Purpose**: Returns the index of the last occurrence of the specified substring.
- **Example**:

```
String str = "Hello, World!";
int lastIndex = str.lastIndexOf("o");
System.out.println(lastIndex);  // Output: 8
```

9. toUpperCase()

- **Purpose**: Converts all the characters of the string to uppercase.
- **Example**:

```
String str = "Hello, World!";
String upper = str.toUpperCase();
System.out.println(upper);  // Output: HELLO, WORLD!
```

10. toLowerCase()

- **Purpose**: Converts all the characters of the string to lowercase.
- **Example**:

```
String str = "Hello, World!";
String lower = str.toLowerCase();
System.out.println(lower);  // Output: hello, world!
```

11. trim()

- **Purpose**: Removes leading and trailing whitespace from the string.
- **Example**:

```
String str = "   Hello, World!   ";
String trimmed = str.trim();
System.out.println(trimmed);  // Output: Hello, World!
```

12. replace(char oldChar, char newChar)

- **Purpose**: Replaces all occurrences of the specified character in the string with a new character.
- **Example**:

```
String str = "Hello, World!";
String replaced = str.replace('o', 'a');
System.out.println(replaced);  // Output: Hella, Warld!
```

13. replaceAll(String regex, String replacement)

- **Purpose**: Replaces all occurrences of a substring that matches the specified regular expression with the provided replacement string.
- **Example**:

```
String str = "abc123xyz";
String replaced = str.replaceAll("\\d", "#");
System.out.println(replaced);  // Output: abc###xyz
```

14. startsWith(String prefix)

- **Purpose**: Checks if the string starts with the specified prefix.
- **Example**:

```
String str = "Hello, World!";
boolean starts = str.startsWith("Hello");
System.out.println(starts);  // Output: true
```

15. endsWith(String suffix)

- **Purpose**: Checks if the string ends with the specified suffix.
- **Example**:

```
String str = "Hello, World!";
boolean ends = str.endsWith("World!");
System.out.println(ends);  // Output: true
```

16. contains(CharSequence sequence)

- **Purpose**: Checks if the string contains the specified sequence of characters.
- **Example**:

```
String str = "Hello, World!";
boolean contains = str.contains("World");
System.out.println(contains);  // Output: true
```

17. split(String regex)

- **Purpose**: Splits the string into an array of substrings based on the specified delimiter (regular expression).
- **Example**:

```
String str = "apple,banana,orange";
String[] fruits = str.split(",");
for (String fruit : fruits) {
    System.out.println(fruit);
}
```

18. valueOf(Object obj)

- **Purpose**: Returns the string representation of the specified object.
- **Syntax**: `String valueOf(Object obj)`
- **Example**:

```
java
Copy code
int num = 100;
String str = String.valueOf(num);
System.out.println(str);  // Output: 100
```

19. equals(Object anObject)

- **Purpose**: Compares the string with another string for equality.
- **Example**:

```
String str1 = "Hello";
String str2 = "Hello";
boolean isEqual = str1.equals(str2);
System.out.println(isEqual);  // Output: true
```

20. equalsIgnoreCase(String anotherString)

- **Purpose**: Compares the string with another string, ignoring case.
- **Example**:

```
String str1 = "Hello";
String str2 = "hello";
boolean isEqual = str1.equalsIgnoreCase(str2);
System.out.println(isEqual);  // Output: true
```

9. Polymorphism

1. Polymorphism Types and Benefits

Polymorphism is one of the four fundamental principles of Object-Oriented Programming (OOP), alongside encapsulation, inheritance, and abstraction. It allows objects of different classes to be treated as objects of a common superclass. The key idea is that a single function, method, or operator can operate on different types of objects, giving flexibility and reusability to the code.

Types of Polymorphism

1. **Compile-Time Polymorphism (Static Polymorphism)**: This type occurs at compile time and is resolved before the program starts running. It is achieved using method overloading and operator overloading.

- **Method Overloading**: This happens when two or more methods in the same class have the same name but differ in parameters (number, type, or both). The correct method is chosen at compile time based on the method signature.

```
class Printer {
    void print(String text) {
        System.out.println(text);
    }
    void print(int number) {
        System.out.println(number);
    }
}
```

- **Operator Overloading**: In some languages like C++, operators can be overloaded to perform different operations based on the operand types.

```
class Complex {
public:
    int real, imag;
    Complex operator + (const Complex& other) {
        Complex temp;
        temp.real = real + other.real;
        temp.imag = imag + other.imag;
        return temp;
    }
```

```
            };
```

2. **Run-Time Polymorphism (Dynamic Polymorphism)**: This type occurs at runtime, and it is resolved when the program is running. It is achieved through method overriding in inheritance hierarchies.

- **Method Overriding**: In this case, a method in a subclass has the same name, return type, and parameters as a method in its superclass. The method to be invoked is determined at runtime based on the object's actual class.

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

class Dog extends Animal {
    void sound() {
        System.out.println("Bark");
    }
}

class Cat extends Animal {
    void sound() {
        System.out.println("Meow");
    }
}
```

Benefits of Polymorphism

1. **Code Reusability**: Polymorphism allows for the reuse of code. For instance, a common interface can be used to handle multiple types of objects, eliminating the need for duplicated code for each specific type.
2. **Flexibility and Extensibility**: Polymorphism enables new classes to be added without modifying existing code. If new subclasses are created, they can be treated in the same way as the parent class objects, improving code maintainability and extensibility.
3. **Simplifies Code Maintenance**: Polymorphism helps simplify code maintenance because it allows developers to work with interfaces or base classes instead of dealing with complex, specific subclasses.
4. **Improves Code Readability**: Polymorphism improves the readability and clarity of code because it allows for the use of a consistent and simplified syntax, even when the underlying objects differ.
5. **Supports Implementing Design Patterns**: Many design patterns like Factory, Strategy, and Command patterns rely heavily on polymorphism. These patterns improve the flexibility, scalability, and maintainability of software systems.

2. MethodOverriding

Method Overriding in Java is a feature that allows a subclass to provide a specific implementation for a method that is already defined in its superclass. The version of the method in the subclass must have the same method signature (name, return type, and parameters) as the method in the superclass.

Method overriding is used to achieve **runtime polymorphism** (also known as dynamic method dispatch), where the method to be executed is determined at runtime based on the object type, rather than the reference type.

Key Concepts of Method Overriding

1. **Method Signature**: The method signature in the subclass should be exactly the same as in the superclass (same name, same parameters, and same return type).
2. **`@Override` Annotation**:
    - This annotation is optional, but it is a good practice to use it, as it tells the compiler that you intend to override a method.
    - If the method does not actually override a method in the superclass, the compiler will generate an error.
3. **Access Modifiers**: The access level of the overriding method should be the same or more permissive than the method in the superclass.
    - For example, if the superclass method is `public`, the overriding method in the subclass must also be `public`.
    - If the superclass method is `protected`, the overriding method can be `protected` or `public`.
    - A `private` method cannot be overridden because it is not accessible in the subclass.
4. **Return Type**:
    - The return type of the overridden method must be the same as the return type of the method in the superclass.
    - Java allows a covariant return type (the return type in the subclass can be a subclass of the return type in the superclass).
5. **Method Body**: The body of the overriding method in the subclass provides the specific implementation, which will replace the superclass method's body when called through a subclass object.

Example: Method Overriding

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

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

class Cat extends Animal {
    @Override
    public void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        animal.sound();  // Output: Animal makes a sound

        Dog dog = new Dog();
        dog.sound();  // Output: Dog barks

        Cat cat = new Cat();
        cat.sound();  // Output: Cat meows
    }
}
```

Rules of Method Overriding

1. **Same Method Signature**: The overriding method must have the same method signature as the method being overridden.
2. **Cannot Change Return Type**: You cannot change the return type of the method, except in the case of covariant return types (i.e., the return type can be a subclass of the original return type).
3. **Access Modifiers**: The overriding method can have the same or a more permissive access modifier than the method being overridden.
    - Example: If the superclass method is `protected`, the subclass can make the method `public`.
4. **Static Methods**: Static methods cannot be overridden. If you define a static method with the same signature in a subclass, it is not considered method overriding. Instead, it is known as **method hiding**.
5. **Final Methods**: If a method is declared `final` in the superclass, it cannot be overridden in the subclass. The `final` modifier indicates that the method cannot be changed or overridden.
6. **Private Methods**: Private methods are not inherited by the subclass and therefore cannot be overridden. They are accessible only within the class they are declared in.

Advantages of Method Overriding

1. **Runtime Polymorphism**: Method overriding facilitates runtime polymorphism. It allows you to implement dynamic method dispatch, which means the method that gets called is determined at runtime based on the object's type.
2. **Behavioral Changes in Subclasses**: It allows a subclass to change the behavior of a method inherited from the superclass. This is useful for customizing functionality for specific subclass needs.
3. **Reusability**: You can override methods to reuse the logic of the superclass method while extending or modifying it to fit specific needs.

3. DynamicBinding(Run-TimePolymorphism)

Dynamic Binding (also known as **Runtime Polymorphism**) is a concept in Java where the method to be called is determined at runtime based on the object type, not the reference type. This is a key feature of object-oriented programming and enables the flexibility of method overriding.

In Java, dynamic binding occurs when a method is called on an object, and the JVM determines which version of the method (overridden or inherited) should be executed based on the actual type of the object at runtime.

Key Points of Dynamic Binding (Runtime Polymorphism)

- **Method Overriding**: Dynamic binding happens when a method is overridden in a subclass.
- **Object Type vs. Reference Type**: The method call is resolved at runtime, depending on the type of the object being referred to, not the reference type.
- **Achieved Through Inheritance**: Dynamic binding requires inheritance and method overriding.
- **Method Resolution**: The Java Virtual Machine (JVM) decides which method to invoke at runtime based on the actual object type.

Example of Dynamic Binding in Java

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

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

class Cat extends Animal {
    @Override
    public void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
    Animal myDog = new Dog();
    Animal myCat = new Cat();
         myAnimal.sound();  // Output: Animal makes a sound
        myDog.sound();     // Output: Dog barks
        myCat.sound();     // Output: Cat meows
    }
}
```

Advantages of Runtime Polymorphism (Dynamic Binding)

1. **Flexibility**: It allows you to write more flexible and reusable code. The method that gets executed is based on the object type, not the reference type, making the code adaptable to future changes.
2. **Loose Coupling**: It enables loose coupling between classes. Subclasses can override methods in the superclass without modifying the superclass itself.
3. **Extensibility**: New classes can be introduced and integrated with existing code without changing much of the existing code, as long as the new class overrides the appropriate methods.
4. **Code Maintainability**: It makes the code easier to maintain and extend because method behavior can be dynamically determined.

4. SuperKeyword and MethodHiding

`super` keyword and method hiding are concepts related to inheritance, but they serve different purposes and work in different ways. Let's break down each concept:

1. `super` Keyword in Java

The `super` keyword is used in Java to refer to the superclass (parent class) of the current object.

`super` Keyword: Used to refer to the superclass, helping access its methods, fields, or constructors.

It is used in several contexts, such as:

- **Accessing Parent Class Methods**: When a subclass overrides a method from its superclass, `super` can be used to call the method from the superclass.
- **Accessing Parent Class Constructors**: You can call the constructor of the superclass from a subclass constructor using `super()`.
- **Accessing Parent Class Fields**: If the subclass has a field with the same name as a field in the superclass, `super` can be used to refer to the field in the superclass.

Example of `super`:

```
class Animal {
    void speak() {
        System.out.println("Animal speaks");
    }
}

class Dog extends Animal {
    void speak() {
        super.speak();  // Call the superclass method
        System.out.println("Dog barks");
    }
}

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

2. Method Hiding in Java

Method hiding occurs when a subclass defines a static method with the same name and signature as a static method in the superclass. Unlike method overriding, which is for instance methods, method hiding is a concept that applies to static methods.

When a static method in a subclass has the same name and signature as a static method in the superclass, the subclass method "hides" the superclass method. The method that gets called depends on the type of the reference, not the actual object.

Method Hiding: Occurs when a subclass declares a static method with the same signature as a static method in the superclass. It hides the superclass method based on the reference type.

Example of Method Hiding:

```
class Animal {
    static void speak() {
        System.out.println("Animal speaks");
    }
}

class Dog extends Animal {
    static void speak() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Animal dog = new Dog();  // Reference type is Animal, object type is
Dog

        animal.speak();  // Output: Animal speaks
        dog.speak();     // Output: Animal speaks
    }
}
```

Key Differences Between Method Overriding and Method Hiding:

- **Method Overriding** applies to instance methods and is based on the object's runtime type.
- **Method Hiding** applies to static methods and is based on the reference type.

10. Interfaces and AbstractClasses

interfaces and **abstract classes** are both used to achieve abstraction, but they differ in their purpose, usage, and characteristics.

1. AbstractClasses and Methods

Abstract Classes in Java

An **abstract class** is a class that cannot be instantiated on its own and must be subclassed. Abstract classes are meant to provide a common base for other classes and can contain both abstract (without implementation) and concrete (with implementation) methods.

Key Points about Abstract Classes:

- **Abstract Methods**: Abstract methods are methods declared without a body. Subclasses must implement these methods.
- **Concrete Methods**: An abstract class can have methods with full implementations. This allows an abstract class to provide some common functionality while still leaving some methods to be implemented by subclasses.
- **Fields and Constructors**: Abstract classes can have instance fields, constructors, and non-static methods.
- **Single Inheritance**: A class can inherit only one abstract class due to Java's restriction on single inheritance (a class cannot extend more than one class, abstract or not).
- **Cannot Be Instantiated**: You cannot create an object of an abstract class directly; you need to create a subclass that implements all abstract methods.

```
abstract class Animal {
    // Abstract method (no body)
    abstract void sound();

    // Concrete method (with body)
    void sleep() {
        System.out.println("Sleeping...");
    }
}

class Dog extends Animal {
    // Implementing the abstract method
    @Override
    void sound() {
        System.out.println("Bark");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();  // Output: Bark
        dog.sleep();  // Output: Sleeping...
    }
}
```

Key Features of Abstract Classes:

- Abstract classes can have **instance variables** and **constructors**.
- They can have both **abstract methods** and **concrete methods**.
- A class can extend only **one abstract class** (due to Java's single inheritance model).

2. Interfaces: Multiple Inheritance in Java

An **interface** in Java is a reference type, similar to a class, that can contain only constants, method signatures, default methods, and static methods. Interfaces cannot contain instance fields or constructors. A class implements an interface to provide concrete implementations for the methods defined in the interface.

Key Points about Interfaces:

- **Abstract Methods**: By default, methods in an interface are abstract, meaning they do not have a body. A class that implements the interface must provide the implementation for these methods.
- **Multiple Inheritance**: A class can implement multiple interfaces, allowing Java to overcome the limitation of single inheritance (since a class can only inherit from one superclass).
- **No Constructor**: Interfaces cannot have constructors because you cannot instantiate an interface directly.
- **Default and Static Methods**: Java 8 introduced default and static methods in interfaces.
  - **Default Methods**: Allow interfaces to have method implementations. This is useful for adding new methods to an interface without breaking the existing implementations.
  - **Static Methods**: Can be defined in interfaces, and they are called using the interface name.

```
interface Animal {
    // Abstract method
    void sound();


    default void sleep() {
        System.out.println("Sleeping...");
    }


    static void breathe() {
        System.out.println("Breathing...");
    }
}

class Dog implements Animal {
    @Override
    public void sound() {
        System.out.println("Bark");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();  // Output: Bark
        dog.sleep();  // Output: Sleeping...

        // Calling static method from interface
        Animal.breathe();  // Output: Breathing...
    }
}
```

Key Features of Interfaces:

- A class can implement **multiple interfaces**.
- Interfaces support **multiple inheritance** of method signatures (unlike classes).
- A class can implement an interface and still extend another class.

multiple inheritance is allowed **through interfaces**, but not through classes. This means that a class can implement multiple interfaces, inheriting the method signatures defined in those interfaces. Java allows a class to implement more than one interface, thus providing a form of multiple inheritance for behavior (method signatures) and allowing a class to inherit from multiple sources without the issues associated with multiple inheritance in classes.

Key Points:

- **A class can implement multiple interfaces**: Unlike class inheritance, which is limited to single inheritance, a class in Java can implement multiple interfaces.
- **Interfaces can provide default methods**: Since Java 8, interfaces can also have `default` methods (methods with implementations), allowing interfaces to provide some default behavior, which can be inherited by classes.
- **No ambiguity**: Java avoids the issues (like the diamond problem) that come with multiple inheritance in classes by ensuring that interfaces only provide method signatures (except for default methods, which can be overridden).

3. Implementing MultipleInterfaces

```
interface Animal {

  void eat();  // Abstract method, no implementation

}
```

Theory

```
interface Flyable {

   void fly();  // Abstract method, no implementation

}


interface Swimable {

   void swim();  // Abstract method, no implementation

}


class Duck implements Animal, Flyable, Swimable {

   @Override

   public void eat() {

      System.out.println("Duck is eating.");

   }

   @Override

   public void fly() {

      System.out.println("Duck is flying.");

   }

   @Override

   public void swim() {

      System.out.println("Duck is swimming.");

   }

}

public class Main {

   public static void main(String[] args) {
      Duck duck = new Duck();

      duck.eat();  // Output: Duck is eating.

      duck.fly();  // Output: Duck is flying.

      duck.swim(); // Output: Duck is swimming.

   }

}
```

11.Packages and AccessModifiers

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

packages are used to group related classes and interfaces, making the code easier to manage and organize. There are two types of packages in Java:

1. **Built-in Packages**
2. **User-Defined Packages**

1. Built-in Packages

Java provides a vast set of built-in packages that contain pre-written classes and interfaces, which can be used to perform various tasks like input/output (I/O), networking, utilities, and much more.

Built-in packages are pre-defined in Java, and they provide a wide range of functionality that can be directly used in your program.

Some common built-in packages in Java:

- **java.lang**,**java.util**,**java.io**,**java.net**,**java.sql**,**java.awt**

```
import java.util.ArrayList;

import java.io.File;
```

2. User-Defined Packages

In addition to the built-in packages, Java allows developers to define their own packages to organize their classes and interfaces. This is particularly useful in large projects to avoid naming conflicts and to improve code structure.

User-defined packages allow you to organize your own classes into logical units and improve code maintainability, especially for large projects.

Benefits of User-Defined Packages:

- **Organization:** Helps in logically grouping related classes.
- **Namespace Management:** Avoids class name conflicts.
- **Reusability:** Classes in packages can be reused in multiple programs.

2.AccessModifiers: Private,Default,Protected,Public

access modifiers (also known as **access specifiers**) control the visibility and accessibility of class members (variables, methods, etc.) from other classes or code outside the class. There are four primary access modifiers used in languages like Java, C#, C++, etc.:

1. Private

- **Definition**: Members marked as `private` are only accessible within the same class.
- **Visibility**: These members are **not accessible** from outside the class, not even by instances of other classes.
- **Use Case**: It's used to encapsulate data and hide implementation details.
- **Private** is the most restrictive, limiting access to the same class.

2. Default (Package-Private in Java)

- **Definition**: If no access modifier is specified, it is considered **default** (in Java) or **package-private**.
- **Visibility**: Members are accessible only **within the same package** but not outside it. This is the "default" level of visibility when no modifier is provided.
- **Use Case**: It's used when you want to allow access within the same package but hide it from other packages.
- **Default** (or package-private) allows access within the same package.

3. Protected

- **Definition**: Members marked as `protected` are accessible within the same package and **subclasses** (including those in different packages).
- **Visibility**: These members are accessible by the class itself, classes in the same package, and subclasses, even if they are in different packages.
- **Use Case**: This modifier is often used when creating class hierarchies and you want to allow subclasses to access and modify certain data.
- **Protected** allows access to subclasses and classes within the same package.

4. Public

- **Definition**: Members marked as `public` are accessible **from anywhere**, whether inside the same package, different packages, or even outside the program (if appropriate).
- **Visibility**: These members have the **widest visibility**.
- **Use Case**: It's typically used for methods and variables that are intended to be accessed by any other class.

- **Public** provides the broadest access, allowing visibility from anywhere.

3. Importing Packages and Classpath

importing packages and setting up the **classpath** are crucial concepts to enable efficient code management, organization, and access to libraries or other classes within a project.

Importing Packages: Allows you to use classes from other packages or external libraries without needing to specify their full paths.

1. Importing Packages in Java

In Java, **packages** are used to group related classes and interfaces. Importing allows you to use classes and interfaces from other packages in your current class.

How to Import Packages

- **Import Specific Classes**: To import a specific class, you can use the `import` statement.
- **Import All Classes in a Package**: To import all the classes from a package, you can use a wildcard (`*`). This is useful when you want to import many classes from the same package.

Where to Place the `import` Statement

- The `import` statement must appear **after the package declaration** (if any) and before the class declaration.
- If no package is specified, the class is in the **default package**, and you don't need to import other classes unless they are in different packages.

Static Import (Java 5+)

You can also import static members (fields and methods) of a class, so you don't need to qualify them with the class name.

Theory

2. Classpath in Java

The **classpath** is an environment variable or a command-line parameter that tells the Java Virtual Machine (JVM) and Java compiler where to find compiled classes and libraries (like JAR files).

The classpath specifies the **directories** or **JAR files** that contain the compiled `.class` files of Java programs.

Classpath: The JVM and compiler use the classpath to find classes and JAR files. You can set the classpath via command-line flags, environment variables, or IDE configurations

Setting the Classpath

1. **Classpath in Command-Line**: When running or compiling a Java program, you can specify the classpath using the `-cp` or `-classpath` flag.
2. **Classpath Using Environment Variable**: You can set the classpath globally for all Java programs by setting the `CLASSPATH` environment variable. This can be done in the system environment settings or in the terminal/command prompt.
3. **Default Classpath**:
   - By default, the **current directory** (denoted by `.`) is included in the classpath.
   - If you don't specify a classpath, Java will automatically search the current directory for `.class` files.
4. **Classpath in JAR Files**: You can include a **JAR file** in the classpath to load classes stored inside it. JAR files are commonly used to bundle Java libraries.
5. **Classpath in IDE (Integrated Development Environment)**:
   - In an IDE like **Eclipse** or **IntelliJ IDEA**, you can specify the classpath by adding external libraries (JARs) to the project settings.
   - These IDEs automatically handle the classpath for you when you compile or run your Java programs.

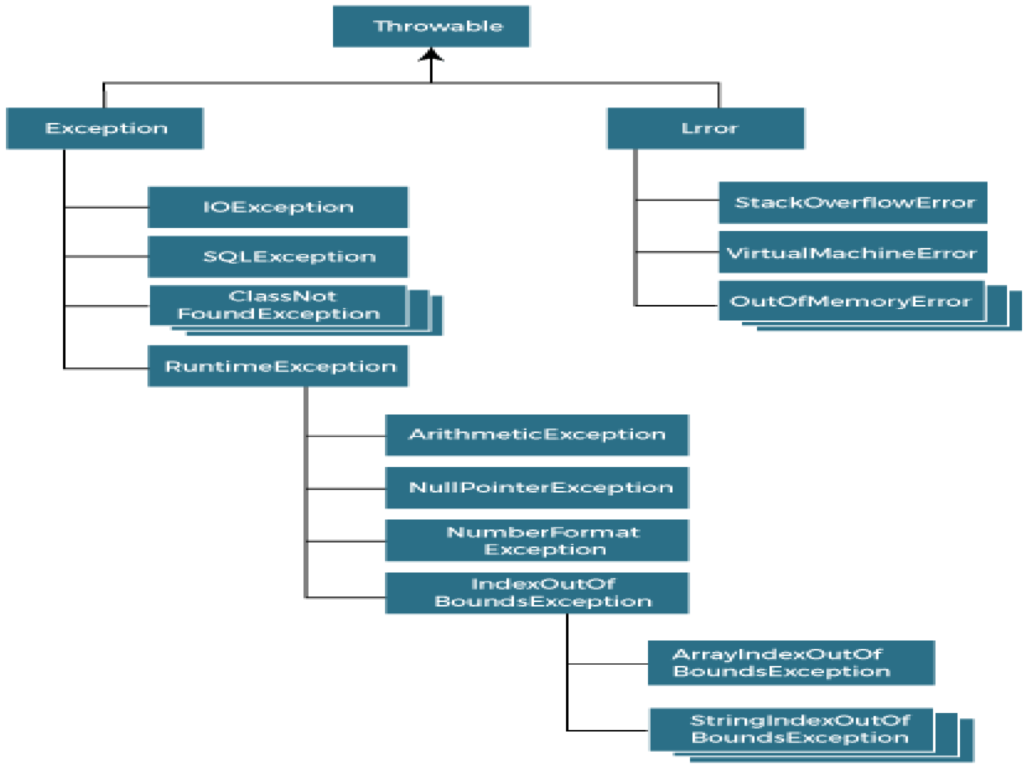
.

3.Setting Classpath: Ensure your classes and libraries (JARs) are included in the classpath to avoid `ClassNotFoundException` and related erro

__12.ExceptionHandling__

Exception handling in Java is a mechanism to handle runtime errors, ensuring that the normal flow of the program is not interrupted. It allows developers to manage errors gracefully and maintain the stability of the application.

The hierarchy of Java Exception classes is given below:



__1. Types of Exceptions: Checked and Unchecked__

exceptions are classified into two main categories based on whether they are checked or unchecked at compile time:

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

These categories are determined by whether or not the exception must be explicitly handled by the programmer at compile time.

Theory

1. Checked Exceptions

Checked exceptions are exceptions that **must** be either caught in a `try-catch` block or declared in the method signature using the `throws` keyword. The Java compiler checks these exceptions during **compile time** to ensure that the program handles them properly.

- **Characteristics**:
  - Checked exceptions are **explicitly checked** by the compiler during compilation.
  - The programmer is required to handle them (either by using a `try-catch` block or by declaring them with `throws`).
  - They usually represent **external conditions** like network failures, file I/O errors, database connection issues, etc.
- **Examples**: `IOException`,`SQLException`,`FileNotFoundException`,`ClassNotFoundException`
- **Handling Checked Exceptions**:
  - You must either catch these exceptions in a `try-catch` block or declare them in the method signature using `throws`.

Example of Handling Checked Exceptions:

```
import java.io.*;

public class CheckedExceptionExample {
    public static void readFile(String filename) throws IOException {
        FileReader file = new FileReader(filename);  // May throw 
FileNotFoundException
        BufferedReader reader = new BufferedReader(file);
        String line = reader.readLine();
        System.out.println(line);
        reader.close();
    }

    public static void main(String[] args) {
        try {
            readFile("nonexistentfile.txt");
        } catch (IOException e) {
            System.out.println("Caught an IOException: " + e.getMessage());
        }
    }
}
```

2. Unchecked Exceptions

Unchecked exceptions, also called **runtime exceptions**, are exceptions that **do not need to be declared** or explicitly handled by the programmer. These exceptions occur during the **runtime** of the program, and the Java compiler does not check for them at compile time.

- **Characteristics**:
  - Unchecked exceptions are **not checked** by the compiler at compile time.

  - They typically represent **programming bugs** or **logical errors**, such as accessing an array element out of bounds or performing an illegal arithmetic operation (like dividing by zero).
  - These exceptions are **optional** to catch; you are not required to catch them, although you can choose to do so if you want to handle them specifically.
- **Examples**:
  - `ArithmeticException`,`NullPointerException`,`ArrayIndexOutOfBoundsException`,`ClassCastException`,`IllegalArgumentException`
- **Handling Unchecked Exceptions**:
  - You **don't have to handle** unchecked exceptions with a `try-catch` block or declare them using `throws`. However, it's still good practice to handle them if needed.

Example of Handling Unchecked Exceptions:

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;  // ArithmeticException: divide by zero
        } catch (ArithmeticException e) {
            System.out.println("Caught an ArithmeticException: " +
e.getMessage());
        }

        try {
            String str = null;
            System.out.println(str.length());  // NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Caught a NullPointerException: " +
e.getMessage());
        }
    }
}
Differences Between Checked and Unchecked Exceptions
```

Aspect	Checked Exceptions	Unchecked Exceptions
Definition	Exceptions that are checked at compile-time.	Exceptions that are not checked at compile-time.
Handling Requirement	Must be either **caught** or **declared** in method signature.	Can be **ignored** or **caught**, but not required.
Example	`IOException`, `SQLException`, `ClassNotFoundException`.	`ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException`.
When to Use	Used for **external issues** (e.g., I/O, network failures).	Used for **programming bugs** or **logic errors** (e.g., invalid arithmetic, null dereference).
Compile-time	Checked by the compiler at compile-	Not checked by the compiler at compile-time.

Aspect	Checked Exceptions	Unchecked Exceptions
Check	time.	
Inheritance	Direct subclass of `Exception`.	Direct subclass of `RuntimeException`.

2. try,catch,finally,throw,throws

1. `try` Block

The `try` block is used to wrap code that may throw an exception. If an exception occurs inside the `try` block, control is passed to the corresponding `catch` block (if one exists).

- **Example**:

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

2. `catch` Block

The `catch` block follows the `try` block and is used to handle exceptions. When an exception is thrown in the `try` block, it is "caught" in the `catch` block. You can have multiple `catch` blocks to handle different types of exceptions.

- **Example**:

```
try {
    String str = null;
    System.out.println(str.length());  // NullPointerException
} catch (NullPointerException e) {
    System.out.println("Caught a NullPointerException: " +
e.getMessage());
}
```

3. `finally` Block

The `finally` block is optional, and it will always be executed after the `try` block, whether or not an exception was thrown. It is typically used to release resources (like closing files, database connections, etc.) regardless of whether an exception occurred or not.

- **Example**:

```
try {
    String str = "Java";
```

```
        System.out.println(str.charAt(10));  //
    StringIndexOutOfBoundsException
    } catch (StringIndexOutOfBoundsException e) {
        System.out.println("Caught an exception: " + e.getMessage());
    } finally {
        System.out.println("This block will always execute.");
    }
```

Important Points about `finally`:

- The `finally` block is always executed, **even if an exception is not thrown**.
- If an exception occurs in the `try` block but is **not caught**, the `finally` block will still execute.
- If there is a `return` statement inside the `try` or `catch` block, the `finally` block will still execute before returning from the method.

4. `throw` Keyword

The `throw` keyword is used to explicitly **throw** an exception. You can use `throw` when you want to create a custom exception or throw a predefined exception based on specific conditions in your code.

- **Example**:

```
public class Test {
    public static void checkAge(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Age must be 18 or
older.");
        } else {
            System.out.println("Age is valid.");
        }
    }

    public static void main(String[] args) {
        checkAge(16);  // This will throw an IllegalArgumentException
    }
}
```

5. `throws` Keyword

The `throws` keyword is used in a method signature to **declare** that the method might throw one or more exceptions. It tells the caller of the method that they should be prepared to handle those exceptions. Unlike `throw`, which actually throws an exception, `throws` is used to signal potential exceptions that the method might throw.

- **Example**:

```
public class Test {
    public static void readFile(String fileName) throws IOException {
        FileReader file = new FileReader(fileName);  //
FileNotFoundException may be thrown
        BufferedReader reader = new BufferedReader(file);
        String line = reader.readLine();
        System.out.println(line);
        reader.close();
    }

    public static void main(String[] args) {
        try {
            readFile("nonexistentfile.txt");  // This may throw
IOException
        } catch (IOException e) {
            System.out.println("Caught an exception: " +
e.getMessage());
        }
    }
}
```

Key Differences between `throw` and `throws`

Keyword	**Purpose**	**Usage**
`throw`	Used to **explicitly throw** an exception.	Inside the body of a method, typically followed by `new` to instantiate an exception.
`throws`	Used to **declare** exceptions that a method may throw.	Used in the method signature to declare possible exceptions.

Conclusion

- **`try`**: A block of code that can potentially throw an exception.
- **`catch`**: A block that handles the exception thrown by the `try` block.
- **`finally`**: A block of code that always executes after the `try` block, regardless of whether an exception was thrown or caught. Typically used for cleanup operations.
- **`throw`**: Used to explicitly throw an exception within your code.
- **`throws`**: Used in a method signature to declare exceptions that the method might throw, signaling to the caller that they must handle those exceptions.

3. Custom Exception Classes

- Custom exceptions are a powerful tool for handling specific error scenarios in Java.
- By defining your own exception classes, you can provide more meaningful error messages and tailor exception handling to your application's needs.
- You can create both **checked** exceptions (by extending `Exception`) and **unchecked** exceptions (by extending `RuntimeException`).

Theory

- The custom exceptions can have multiple constructors, allowing for flexible error reporting and more control over how errors are communicated within your application.

```
class InvalidAgeException  extends Exception
{
   public InvalidAgeException (String str)
   {
      // calling the constructor of parent Exception
      super(str);
   }
}

// class that uses custom exception InvalidAgeException
public class TestCustomException1
{

   // method to check the age
   static void validate (int age) throws InvalidAgeException{
     if(age < 18){

      // throw an object of user defined exception
      throw new InvalidAgeException("age is not valid to vote");
   }
     else {
      System.out.println("welcome to vote");
      }
    }
  // main method
 public static void main(String args[])
   {
      try
      {
         // calling the method
         validate(13);
      }
      catch (InvalidAgeException ex)
      {
         System.out.println("Caught the exception");

         // printing the message from InvalidAgeException object
         System.out.println("Exception occured: " + ex);
      }

      System.out.println("rest of the code...");
   }
}
```

Theory

13.Multithreading

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

1. Introduction to Threads

a **thread** is the smallest unit of execution within a process. It represents a single sequence of instructions that can be executed independently, but it shares the resources of its parent process, such as memory and open files. Threads allow for **concurrent execution**, which can improve the performance of applications, especially on multi-core processors.

Key Concepts of Threads:

1. **Process vs Thread**:
   - A **process** is a program in execution that has its own memory space.
   - A **thread** is a lightweight sub-unit of a process that shares the same memory and resources as other threads within the same process.
2. **Concurrency vs Parallelism**:
   - **Concurrency** refers to the ability of a system to handle multiple tasks at the same time, but not necessarily simultaneously.
   - **Parallelism** is when multiple tasks are actually executed simultaneously, often on different processors or cores.
3. **Types of Threads**:
   - **User-level threads**: Managed by user-level libraries, not the operating system. These threads are faster but lack full integration with OS scheduling.
   - **Kernel-level threads**: Managed directly by the operating system. These threads benefit from better scheduling and multitasking support.

  - o **Hybrid threads**: A combination of user and kernel-level threads.

4. **Multithreading**:
   - o **Multithreading** is a technique that allows multiple threads to run concurrently within a process. It can be used to perform multiple operations at once, leading to more efficient execution, especially in CPU-bound and I/O-bound applications.
5. **Benefits of Multithreading**:
   - o **Improved performance**: Tasks can be divided into smaller, independent threads that can run in parallel on multi-core processors.
   - o **Better resource utilization**: Threads can take advantage of idle CPU cycles or wait on I/O operations while other threads continue executing.
   - o **Responsiveness**: In interactive applications (e.g., GUI applications), one thread can handle user input while others perform background tasks.
6. **Challenges with Threads**:
   - o **Synchronization**: Since threads share memory, proper synchronization mechanisms (like locks, semaphores, and mutexes) are needed to avoid data corruption when multiple threads access shared data.
   - o **Context switching overhead**: While threads are lightweight, managing many threads can introduce overhead from the operating system switching between them.
   - o **Deadlock**: This occurs when two or more threads are blocked indefinitely because they are waiting for each other to release resources.

2.CreatingThreadsbyExtendingThreadClassorImplementing RunnableInterface

creating threads can be done in two primary ways: by **extending the `Thread` class** or by **implementing the `Runnable` interface**. Both approaches allow you to define the code that the thread will execute, but there are key differences in how they work.

1. Creating Threads by Extending the `Thread` Class

When you extend the `Thread` class, you override its `run()` method to specify the code that will execute in the new thread. You don't need to implement any interface, as `Thread` already provides a `run()` method, but you need to call `start()` to begin the execution of the thread.

Example:

```
class MyThread extends Thread {
    @Override
    public void run() {
        // Code to be executed in the thread
        System.out.println("Thread is running.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create a new thread by extending the Thread class
        MyThread thread = new MyThread();
        thread.start(); // Start the thread
    }
}
```

Advantages:

- Simpler and more direct for small, single-threaded tasks.
- No need to implement an additional interface.

Disadvantages:

- Java supports single inheritance, so if you extend the `Thread` class, you cannot extend any other class.
- Less flexible if you want to reuse the same code in different contexts (you can't extend multiple classes).

---

2. Creating Threads by Implementing the `Runnable` Interface

Another approach is to implement the `Runnable` interface, which requires you to implement the `run()` method. Unlike extending `Thread`, implementing `Runnable` allows you to create threads in a more flexible way, as you can still extend other classes (because Java allows multiple interfaces but only one class inheritance).

Example:

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        // Code to be executed in the thread
        System.out.println("Thread is running.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create a Runnable instance
        MyRunnable myRunnable = new MyRunnable();

        // Create a new Thread object and pass the Runnable to the
constructor
        Thread thread = new Thread(myRunnable);
        thread.start(); // Start the thread
    }
}
```

Advantages:

- **Better for code reuse**: You can extend other classes while still implementing `Runnable`.
- **More flexible**: The `Runnable` interface can be passed to multiple threads, allowing the same task to be executed by different threads.

- **Recommended for multi-threading**: It is often the preferred approach because it allows better separation of concerns and cleaner code design.

Disadvantages:

- Requires you to create a `Thread` object in addition to implementing `Runnable`.
- Slightly more complex if you don't need the flexibility of reusing the task.

Comparing Both Approaches

Feature	Extending `Thread` Class	Implementing `Runnable` Interface
Inheritance	Single inheritance (can only extend `Thread`)	Can implement multiple interfaces (flexible with inheritance)
Code Reusability	Limited, because you can only extend `Thread`	More reusable, can be used with multiple threads
Task Definition	Override `run()` method of `Thread` class	Implement `run()` method of `Runnable` interface
Best Used For	Simple cases with no need for multiple inheritance	More complex scenarios, better for separation of concerns and scalability

When to Use Each Approach:

- **Extending the `Thread` class**:
  - Use when you need a simple, one-off thread.
  - It's fine if you don’t need to extend any other class.
  - You don't need to reuse or share the `run()` code across different threads.
- **Implementing the `Runnable` interface**:
  - Use when you need to reuse the same task across multiple threads.
  - It's a better choice when your task can be shared across different threads.
  - It offers more flexibility, especially when dealing with more complex multi-threaded applications.

3. Thread LifeCycle

A **thread** in Java goes through several distinct phases during its existence, known as the **thread life cycle**. Understanding these phases is crucial for working with threads, as it helps manage thread execution and synchronization effectively.

Theory

Here's an overview of the **Thread Life Cycle** in Java, along with each state a thread can go through:

Thread Life Cycle States

1. **New (Born) State**
   - **Description**: When a thread is created but has not yet started execution, it is in the **New** state. At this point, the thread is just an object, and its `run()` method has not been invoked.
   - **Transition**: A thread enters this state after being instantiated, but before the `start()` method is called.
2. **Runnable (Ready) State**
   - **Description**: After calling the `start()` method on the thread, the thread enters the **Runnable** state. In this state, the thread is ready for execution and is waiting for CPU time. **Runnable** doesn't necessarily mean the thread is currently running; it just means the thread is ready to be scheduled by the **Thread Scheduler**.
   - **Transition**: A thread enters this state from the **New** state when `start()` is called, or it may return to the **Runnable** state after yielding the CPU, sleeping, or being waiting to be resumed.
3. **Blocked (Waiting for Resources) State**
   - **Description**: A thread enters the **Blocked** state when it is waiting to acquire a lock or resource that is currently held by another thread. For example, this happens when multiple threads try to access the same synchronized block or method and one has to wait for the other to release the lock.
   - **Transition**: A thread enters the **Blocked** state when it is trying to enter a synchronized block and the lock is already held by another thread.
4. **Waiting (Idle) State**
   - **Description**: A thread enters the **Waiting** state when it is waiting indefinitely for another thread to perform a particular action. This happens, for example, when a thread calls `wait()`, `join()`, or `sleep()`.
   - **Transition**: A thread enters this state after calling methods like:
     - `wait()`
     - `join()`
     - `sleep()`
5. **Timed Waiting State**
   - **Description**: A thread enters the **Timed Waiting** state when it is waiting for a specific period of time before it can resume. This happens, for example, when `sleep()` or `join()` is called with a specified duration.
   - **Transition**: A thread enters this state when it calls methods like `sleep(millis)`, `join(millis)`, or `wait(millis)`, and after the specified time has elapsed, the thread is moved back to the **Runnable** state.
6. **Terminated (Dead) State**
   - **Description**: A thread enters the **Terminated** state when it has finished executing its `run()` method or when it is forcibly terminated (for example, via an exception or calling `stop()`, although `stop()` is deprecated due to its unsafe nature).
   - **Transition**: A thread enters the **Terminated** state after it completes its execution or is terminated due to an exception or error.

4 Synchronization and Inter-threadCommunication

1. Synchronization in Java

Synchronization is a technique used to ensure that only one thread can access a shared resource at any given time. It is necessary when multiple threads are modifying shared data to prevent inconsistency. Java provides two main mechanisms to implement synchronization:

- **Synchronized Methods**
- **Synchronized Blocks**

1.1 Synchronized Methods

By marking a method with the `synchronized` keyword, you ensure that only one thread at a time can execute that method on a given object. If a thread is already executing a synchronized method, other threads trying to invoke the same method on the same object will be blocked until the first thread finishes.

Example:

```
class Counter {
    private int count = 0;

    // Synchronized method to ensure thread safety
    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();  // Incrementing count
            }
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();  // Incrementing count
            }
        });

        t1.start();
        t2.start();
    }
}
```

Theory

1.2 Synchronized Blocks

Sometimes, you don't need to synchronize the entire method, but only a specific part of the code where shared resources are accessed. This is where **synchronized blocks** come into play. You can lock only a particular block of code, making it more efficient than synchronizing the entire method.

Example:
```
class Counter {
    private int count = 0;

    // Synchronized block inside the method
    public void increment() {
        synchronized (this) {  // Locking a specific block
            count++;
        }
    }

    public int getCount() {
        return count;
    }
}
```

1.3 Locks and ReentrantLocks (Advanced)

Java also provides **Locks** (from the `java.util.concurrent.locks` package), which offer more advanced synchronization mechanisms than the `synchronized` keyword, such as explicit lock acquisition and timed locks.

2. Inter-Thread Communication in Java

Inter-thread communication refers to mechanisms that allow threads to communicate with each other. This is necessary when threads need to cooperate or wait for a certain condition before proceeding. Java provides methods like `wait()`, `notify()`, and `notifyAll()` to facilitate communication between threads.

These methods are defined in the `Object` class, meaning they can be used on any object.

2.1 The `wait()` Method

The `wait()` method causes the current thread to release the lock and enter the **waiting** state. The thread remains in this state until another thread sends a notification (`notify()` or `notifyAll()`).

- The `wait()` method must be called within a **synchronized** context (i.e., inside a synchronized method or block).

Example:
```
class Data {
    private int number;
    public synchronized void produce() throws InterruptedException {
        while (number != 0) {
            wait();  // Wait until the number is consumed
        }
        number = (int) (Math.random() * 100);
        System.out.println("Produced: " + number);
        notify();  // Notify the consumer thread
    }

    public synchronized void consume() throws InterruptedException {
        while (number == 0) {
            wait();  // Wait until the number is produced
        }
        System.out.println("Consumed: " + number);
        number = 0;
        notify();  // Notify the producer thread
    }
}

public class Main {
    public static void main(String[] args) {
        Data data = new Data();

        Thread producer = new Thread(() -> {
            try {
                while (true) {
                    data.produce();
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        Thread consumer = new Thread(() -> {
            try {
                while (true) {
                    data.consume();
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        producer.start();
        consumer.start();
    }
}.
```

2.2 The `notify()` Method

The `notify()` method wakes up a single thread that is waiting on the object's monitor. It’s important to note that the `notify()` method does not guarantee that the waiting thread will resume immediately; it just signals that it can potentially resume once it acquires the lock.

2.3 The `notifyAll()` Method

The `notifyAll()` method wakes up all the threads that are waiting on the object’s monitor. This is useful when you want all waiting threads to check if they can proceed.

2.4 The `sleep()` Method

The `sleep()` method is used to pause the execution of the current thread for a specified period of time. Unlike `wait()`, it does not release the lock associated with the thread, and it doesn’t require synchronization.

3. Deadlock and Avoidance

A **deadlock** is a situation where two or more threads are blocked forever, waiting for each other to release resources. To avoid deadlocks, it's essential to follow these practices:

- **Avoid circular waits**: Ensure that threads acquire resources in a predefined order.
- **Use timeouts**: Use try-locks or timeouts to avoid indefinite blocking.
- **Minimize synchronized blocks**: Keep synchronized blocks as small as possible to reduce the chances of deadlock.

14.FileHandling

file handling refers to the process of performing operations such as reading from, writing to, and manipulating files in a system. Java provides a comprehensive set of classes and methods for file handling through the **java.io** and **java.nio** (New I/O) packages.

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

File Input/Output (File I/O) in Java refers to the process of reading from and writing to files in a file system. The **`java.io`** package provides a rich set of classes that help in performing file operations. The `java.io` package allows developers to read from and write to files, create files, delete files, and perform other file manipulations.

In Java, files can be handled in both text and binary formats. The classes in the `java.io` package are designed to handle these types of files and provide streams for input and output.

Theory

Key Classes for File I/O in `java.io` package

1. **File Class**:
    - The `File` class is used to represent file and directory pathnames in an abstract manner.
    - You can use this class to create, delete, and check file properties (like existence, readability, etc.).
2. **InputStream & OutputStream**:
    - These are abstract classes for reading and writing byte-based data (binary data). Classes like `FileInputStream`, `FileOutputStream`, `BufferedInputStream`, and `BufferedOutputStream` extend these classes.
3. **Reader & Writer Classes**:
    - These are used for reading and writing character-based data (text data).
    - Classes like `FileReader`, `FileWriter`, `BufferedReader`, `BufferedWriter`, and `PrintWriter` are used for text file operations.
4. **Buffered Classes**:
    - Classes such as `BufferedReader` and `BufferedWriter` are used to read and write data in large chunks, improving the performance for large files.

Basic File I/O Operations in Java

1. **Creating a File**: You can create a new file using the `createNewFile()` method of the `File` class.

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

public class FileCreation {
    public static void main(String[] args) {
        File file = new File("example.txt");

        try {
            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
            } else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

2. **Reading from a File**:
    - Java provides classes such as `FileReader` and `BufferedReader` for reading files. `BufferedReader` allows you to read the file line by line.

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

```
public class FileRead {
    public static void main(String[] args) {
        try {
            FileReader fr = new FileReader("example.txt");
            BufferedReader br = new BufferedReader(fr);

            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }

            br.close();
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

3. **Writing to a File**:
    - To write to a file, you can use `FileWriter` or `BufferedWriter`. These classes allow writing data to the file. You can use `FileWriter` to overwrite the file or `BufferedWriter` for efficient writing.

```
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;

public class FileWrite {
    public static void main(String[] args) {
        try {
            FileWriter fw = new FileWriter("example.txt");
            BufferedWriter bw = new BufferedWriter(fw);

            bw.write("Hello, World!");
            bw.newLine();
            bw.write("This is a test file in Java.");

            bw.close();
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

4. **Appending to a File**:
    - To append data to an existing file (instead of overwriting it), use `FileWriter` with the `true` flag.

```
import java.io.FileWriter;
import java.io.IOException;

public class FileAppend {
    public static void main(String[] args) {
        try {
            FileWriter fw = new FileWriter("example.txt", true);  //
'true' flag to append
            fw.write("\nAppending some text.");
            fw.close();
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

5. **Checking if a File Exists**:
   - You can check if a file exists using the `exists()` method of the `File` class.

```
import java.io.File;

public class FileExists {
    public static void main(String[] args) {
        File file = new File("example.txt");

        if (file.exists()) {
            System.out.println("The file exists.");
        } else {
            System.out.println("The file does not exist.");
        }
    }
}
```

6. **Deleting a File**:
   - You can delete a file using the `delete()` method of the `File` class.

```
import java.io.File;

public class FileDelete {
    public static void main(String[] args) {
        File file = new File("example.txt");

        if (file.delete()) {
            System.out.println("File deleted successfully.");
        } else {
            System.out.println("Failed to delete the file.");
        }
    }
}
```

Important Classes for File I/O

1. **File**:
   - Represents files and directory pathnames in an abstract manner. It is used for file manipulation operations such as creating, deleting, checking properties, etc.

```
File file = new File("filename.txt");
```

```
file.exists();  // Check if file exists
file.delete();  // Delete file
```

2. **FileReader**:
   - A character-based stream used for reading files. It is used to read data from a text file.

```
FileReader fr = new FileReader("filename.txt");
int character;
while ((character = fr.read()) != -1) {
    System.out.print((char) character);  // Print characters one by one
}
```

3. **BufferedReader**:
   - A wrapper class around `FileReader` to efficiently read text from a file. It allows reading one line at a time.

```
java
Copy code
BufferedReader br = new BufferedReader(new FileReader("filename.txt"));
String line;
while ((line = br.readLine()) != null) {
    System.out.println(line);  // Print each line
}
```

4. **FileWriter**:
   - A character-based stream used for writing to files. It writes data to the file in text format.

```
FileWriter fw = new FileWriter("filename.txt");
fw.write("Hello, World!");
fw.close();
```

5. **BufferedWriter**:
   - A wrapper class around `FileWriter` for efficient writing to a file. It is used to write data in chunks, reducing the number of write operations.

```
BufferedWriter bw = new BufferedWriter(new FileWriter("filename.txt"));
bw.write("Hello, World!");
bw.newLine();
bw.close();
```

6. **PrintWriter**:
   - A convenient class that can be used for writing formatted text to files. It can be used for both text and binary files.

```
PrintWriter pw = new PrintWriter(new FileWriter("filename.txt"));
pw.println("This is a print statement.");
pw.close();
```

2. FileReader and FileWriter Classes

FileWriter Class

Java FileWriter class is used to write character-oriented data to a file. It is character-oriented class which is used for file handling in java.

Unlike FileOutputStream class, you don't need to convert string into byte array because it provides method to write string directly.

```
package com.javatpoint;
import java.io.FileWriter;
public class FileWriterExample {
   public static void main(String args[]){
        try{
          FileWriter fw=new FileWriter("D:\\testout.txt");
          fw.write("Welcome to javaTpoint.");
          fw.close();
         }catch(Exception e){System.out.println(e);}
         System.out.println("Success...");
    }
}
```

FileReader Class

Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class.

It is character-oriented class which is used for file handling in java.

```
package com.javatpoint;

import java.io.FileReader;
public class FileReaderExample {
   public static void main(String args[])throws Exception{
         FileReader fr=new FileReader("D:\\testout.txt");
         int i;
         while((i=fr.read())!=-1)
         System.out.print((char)i);
         fr.close();
   }
}
```

3. BufferedReaderandBufferedWriter

BufferedWriter Class

Java BufferedWriter class is used to provide buffering for Writer instances. It makes the performance fast. It inherits Writer class. The buffering characters are used for providing the efficient writing of single arrays, characters, and strings.

```
package com.javatpoint;
import java.io.*;
public class BufferedWriterExample {
public static void main(String[] args) throws Exception {
   FileWriter writer = new FileWriter("D:\\testout.txt");
   BufferedWriter buffer = new BufferedWriter(writer);
   buffer.write("Welcome to javaTpoint.");
   buffer.close();
   System.out.println("Success");
   }
}
```

BufferedReader Class

BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method. It makes the performance fast. It inherits the Reader class.

```
package com.javatpoint;
import java.io.*;
public class BufferedReaderExample {
   public static void main(String args[])throws Exception{
        FileReader fr=new FileReader("D:\\testout.txt");
        BufferedReader br=new BufferedReader(fr);

        int i;
        while((i=br.read())!=-1){
        System.out.print((char)i);
        }
        br.close();
        fr.close();
   }    }
```

4. Serialization and Deserialization

Serialization

Serialization in Java is a mechanism of *writing the state of an object into a byte-stream*. It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

serializing the object, we call the **writeObject()** method of *ObjectOutputStream* class, and for deserialization we call the **readObject()** method of *ObjectInputStream* class.

Advantages of Java Serialization

It is mainly used to travel object's state on the network (that is known as marshalling).

java.io.Serializable interface

Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that the objects of these classes may get a certain capability. The **Cloneable** and **Remote** are also marker interfaces.

The **Serializable** interface must be implemented by the class whose object needs to be persisted.

The String class and all the wrapper classes implement the *java.io.Serializable* interface by default.

```
import java.io.Serializable;
public class Student implements Serializable{
 int id;
 String name;
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}
```

Deserialization

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. Let's see an example where we are reading the data from a deserialized object.

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. Let's see an example where we are reading the data from a deserialized object.

```
import java.io.*;
class Depersist{
 public static void main(String args[]){
  try{
  //Creating stream to read the object
  ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
  Student s=(Student)in.readObject();
  //printing the data of the serialized object
  System.out.println(s.id+" "+s.name);
  //closing the stream
  in.close();
  }catch(Exception e){System.out.println(e);}
 }
}
```

15.CollectionsFramework

1. Introduction to Collections Framework

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

2. List,Set,Map,andQueue Interfaces

1. List Interface:

- **Description**: A collection that allows for ordered storage of elements. Lists allow duplicate elements and provide positional access (i.e., elements are indexed).
- **Key Characteristics**:
  - Elements can be inserted or accessed by their position (index).
  - Allows duplicates (multiple occurrences of the same element).
  - Maintains the insertion order of elements.
- **Common Implementations**:
  - `ArrayList`: A dynamically resizing array-based list.

- o `LinkedList`: A doubly linked list implementation.
- o `Vector`: An older, synchronized implementation (less commonly used today).
- **Methods**:
  - o `add(),get(),remove(),set(),indexOf(),contains()`

Example:

```
List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
list.add("Apple"); // List allows duplicates
System.out.println(list.get(0));  // Output: Apple
```

2. Set Interface:

- **Description**: A collection that does not allow duplicate elements and typically does not maintain the order of elements (although some implementations may).
- **Key Characteristics**:
  - o No duplicates allowed.
  - o The order in which elements are stored is not guaranteed (unless you use a specific implementation like `LinkedHashSet` or `TreeSet`).
- **Common Implementations**:
  - o `HashSet`: A hash-based implementation that does not guarantee any order.
  - o `LinkedHashSet`: A hash-based implementation that maintains insertion order.
  - o `TreeSet`: A set that stores elements in a sorted order (based on natural ordering or a custom comparator).
- **Methods**:
  - o `add(),remove(),contains(),size()`

Example:

```
Set<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
set.add("Apple"); // Duplicates will not be added
System.out.println(set);  // Output: [Apple, Banana] (order may vary)
```

3. Map Interface:

- **Description**: A collection that maps keys to values, where each key is unique and maps to exactly one value. Maps do not allow duplicate keys but can have duplicate values.
- **Key Characteristics**:
  - o Stores key-value pairs.
  - o Keys are unique; values can be duplicated.
  - o The order of key-value pairs is not guaranteed (unless using a specific implementation like `LinkedHashMap` or `TreeMap`).
- **Common Implementations**:
  - o `HashMap`: A hash-based implementation that does not guarantee any order.
  - o `LinkedHashMap`: A hash-based implementation that maintains insertion order.

  - `TreeMap`: A map that stores entries in sorted order of the keys.
- **Methods**:
  - `put(),get(),remove(),containsKey(),keySet(),values(),size()`

Example:

```
Map<String, Integer> map = new HashMap<>();
map.put("Apple", 1);
map.put("Banana", 2);
map.put("Apple", 3); // The value for "Apple" is updated
System.out.println(map);  // Output: {Apple=3, Banana=2}
```

4. Queue Interface:

- **Description**: A collection used to hold elements prior to processing. Typically, a queue follows the FIFO (First-In-First-Out) order.
- **Key Characteristics**:
  - Primarily used for holding elements for processing, typically in a first-come, first-served manner.
  - Supports operations like adding elements to the end (enqueue), removing elements from the front (dequeue), and peeking at the front element.
- **Common Implementations**:
  - `LinkedList`: Often used as a queue since it implements both the `Queue` and `Deque` interfaces.
  - `PriorityQueue`: A queue that orders elements according to their natural ordering or a comparator.
  - `ArrayDeque`: A resizable array implementation of the `Deque` interface that can be used as a queue.
- **Methods**:
  - `offer(),poll(),peek(),size(),remove()`

Example:

```
Queue<String> queue = new LinkedList<>();
queue.offer("Apple");
queue.offer("Banana");
System.out.println(queue.peek()); // Output: Apple (first element)
System.out.println(queue.poll()); // Output: Apple (removes first element)
System.out.println(queue); // Output: [Banana]
```

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

1. ArrayList

- **Implements**: `List`

- **Description**: An `ArrayList` is a resizable array implementation of the `List` interface. It allows for random access and dynamic resizing as elements are added or removed.
- **Key Characteristics**:
  - **Ordered**: Maintains the insertion order of elements.
  - **Indexed**: Allows access to elements by their index.
  - **Dynamic Size**: The size of the list grows automatically when more elements are added.
  - **Fast Random Access**: Offers constant time complexity for accessing elements by index (`O(1)`).
  - **Slower Insertions/Deletions**: Insertions or deletions (except at the end) are slower compared to a `LinkedList` because elements have to be shifted.
- **Performance**:
  - **Access time**: `O(1)` for accessing elements.
  - **Insertions/Removals**: `O(n)` in the worst case, especially when not adding/removing at the end.
- **Example**:

```
List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
System.out.println(list.get(0));  // Output: Apple
```

2. LinkedList

- **Implements**: `List`, `Queue`, `Deque`
- **Description**: A `LinkedList` is a doubly linked list implementation of the `List` and `Queue` interfaces. It consists of nodes that contain references to both the next and the previous elements.
- **Key Characteristics**:
  - **Ordered**: Maintains insertion order.
  - **Indexed**: Allows access to elements by index.
  - **Doubly Linked**: Each element is connected to the next and previous elements, which makes insertions and deletions faster compared to `ArrayList` for elements at the beginning or middle of the list.
  - **Flexible for Queue/Deque Operations**: Can be used as a queue or deque (double-ended queue), meaning elements can be added or removed from both ends efficiently.
- **Performance**:
  - **Access time**: `O(n)` for accessing elements by index (since it must traverse the list).
  - **Insertions/Removals**: `O(1)` for adding/removing at the beginning or end of the list.
- **Example**:

```
List<String> list = new LinkedList<>();
list.add("Apple");
list.add("Banana");
System.out.println(list.get(1));  // Output: Banana
```

3. HashSet

- **Implements**: `Set`

- **Description**: A `HashSet` is a collection that does not allow duplicate elements and does not guarantee any specific order of elements.
- **Key Characteristics**:
    - **No Duplicates**: Automatically eliminates duplicate entries.
    - **Unordered**: Elements are not stored in any particular order.
    - **Uses Hashing**: Internally uses a hash table for storing elements, which ensures constant time complexity for basic operations like adding, removing, and checking for an element (`O(1)` on average).
- **Performance**:
    - **Access/Search/Insert/Delete**: `O(1)` average time for most operations (in the worst case, `O(n)` if hash collisions occur).
- **Example**:

```
Set<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
set.add("Apple");  // Duplicate will be ignored
System.out.println(set);  // Output: [Apple, Banana] (order may vary)
```

4. TreeSet

- **Implements**: `Set`
- **Description**: A `TreeSet` is a NavigableSet that uses a Red-Black tree (a type of balanced binary search tree) to store elements in a sorted order.
- **Key Characteristics**:
    - **No Duplicates**: Like `HashSet`, it eliminates duplicate elements.
    - **Sorted**: Elements are stored in ascending order (by default) or according to a specified comparator.
    - **Performance**: Provides logarithmic time complexity for most operations (`O(log n)`), due to the tree structure.
- **Performance**:
    - **Access/Search/Insert/Delete**: `O(log n)` due to tree balancing.
- **Example**:

```
Set<Integer> set = new TreeSet<>();
set.add(5);
set.add(1);
set.add(3);
System.out.println(set);  // Output: [1, 3, 5]
```

5. HashMap

- **Implements**: `Map`
- **Description**: A `HashMap` is a hash table-based implementation of the `Map` interface. It stores key-value pairs, with each key being unique.
- **Key Characteristics**:
    - **Unordered**: Does not guarantee the order of keys or values.
    - **Hashing**: Internally uses a hash table for storing key-value pairs.
    - **Allows null**: Allows one `null` key and multiple `null` values.

  - **Performance**: Provides constant time complexity for most operations (`O(1)` on average).
- **Performance**:
  - **Put/Get/Remove**: `O(1)` on average for most operations (may degrade to `O(n)` if hash collisions occur).
- **Example**:

```
Map<String, Integer> map = new HashMap<>();
map.put("Apple", 1);
map.put("Banana", 2);
System.out.println(map.get("Apple"));  // Output: 1
```

6. TreeMap

- **Implements**: `Map`
- **Description**: A `TreeMap` is a `NavigableMap` implementation that uses a Red-Black tree to store key-value pairs in a sorted order.
- **Key Characteristics**:
  - **Sorted**: Keys are stored in ascending order by default or according to a specified comparator.
  - **No `null` Keys**: Does not allow `null` keys (but can have `null` values).
  - **Navigable**: Provides methods for navigation (e.g., retrieving the closest key).
  - **Performance**: Provides logarithmic time complexity for most operations (`O(log n)`).
- **Performance**:
  - **Put/Get/Remove**: `O(log n)` due to tree balancing.
- **Example**:

```
Map<Integer, String> map = new TreeMap<>();
map.put(3, "Apple");
map.put(1, "Banana");
map.put(2, "Orange");
System.out.println(map);
```

4. Iterators and ListIterators

Iterator Interface

The `Iterator` interface is used to iterate over collections that implement the `Collection` interface (like `List`, `Set`, etc.). It allows sequential access to elements of the collection without exposing the underlying structure.

Key Methods in `Iterator`:

1. **`hasNext()`**:
   - Returns `true` if there are more elements to iterate over.
   - Returns `false` if the iterator has traversed all elements.
2. **`next()`**:

   - Returns the next element in the iteration.
   - Throws `NoSuchElementException` if there are no more elements.
3. **`remove()`**:
   - Removes the last element returned by the iterator.
   - Throws `IllegalStateException` if `next()` has not been called yet, or if `remove()` has already been called after the last call to `next()`.

Example: Using `Iterator` to Traverse a `List`

```
import java.util.*;

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

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

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

Key Points about `Iterator`:

- It is used to iterate over any collection (such as `List`, `Set`, etc.) that implements the `Collection` interface.
- It allows you to remove elements from the collection during iteration (using `remove()`).
- The `Iterator` can only iterate in one direction (forward) through the collection.

ListIterator Interface

`ListIterator` is a specialized subinterface of `Iterator` that is available for collections that implement the `List` interface (e.g., `ArrayList`, `LinkedList`). It extends the `Iterator` interface and adds more functionality, allowing you to iterate both forward and backward, and also modify the list while iterating.

Key Methods in `ListIterator`:

1. **`hasNext()`**:
   - Returns `true` if there are more elements when traversing in the forward direction.
2. **`next()`**:
   - Returns the next element in the list.
3. **`hasPrevious()`**:
   - Returns `true` if there are more elements when traversing in the backward direction.

4. **`previous()`**:
    - Returns the previous element in the list.
5. **`add(E e)`**:
    - Inserts the specified element into the list (it will be inserted before the element that would be returned by `next()` if traversing forward).
6. **`set(E e)`**:
    - Replaces the last element returned by `next()` or `previous()` with the specified element.
7. **`remove()`**:
    - Removes the last element returned by `next()` or `previous()`.

Example: Using `ListIterator` to Traverse a `List` Forward and Backward

```
import java.util.*;

public class ListIteratorExample {
    public static void main(String[] args) {
        List<String> list = new LinkedList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Orange");

        ListIterator<String> listIterator = list.listIterator();

        // Forward iteration
        System.out.println("Forward iteration:");
        while (listIterator.hasNext()) {
            System.out.println(listIterator.next());
        }

        // Backward iteration
        System.out.println("\nBackward iteration:");
        while (listIterator.hasPrevious()) {
            System.out.println(listIterator.previous());
        }

        // Adding and setting elements
        listIterator = list.listIterator();
        listIterator.next();
        listIterator.add("Mango");
        listIterator.set("Pineapple");
        System.out.println("\nAfter modifications: " + list);
    }
}
```

Key Points about `ListIterator`:

- It is only available for collections that implement the `List` interface (e.g., `ArrayList`, `LinkedList`).
- You can iterate both forward and backward.
- It allows modifications to the list (such as adding, replacing, and removing elements) while iterating.

- `ListIterator` is more powerful than `Iterator` because it allows greater flexibility in traversing and modifying the list.

16.JavaInput/Output(I/O)

Input/Output (I/O) refers to the process of reading from and writing to data sources such as files, consoles, or networks. Java provides a comprehensive set of libraries for handling I/O operations in various ways, including classes in the `java.io` package and the more recent `java.nio` (New I/O) package. Below are some of the key concepts and classes in Java I/O.

1. StreamsinJava(InputStream,OutputStream)

the `InputStream` and `OutputStream` classes are part of the `java.io` package and are the foundational classes for handling byte-based I/O operations. They represent a sequence of bytes that can be read from or written to various sources (such as files, memory, or network connections).

1. InputStream (Byte Input):

The `InputStream` class is used for reading byte data. It serves as the base class for all byte input streams in Java. The subclasses of `InputStream` handle the reading of byte data from various sources.

Important Methods in `InputStream`:

- `int read()`: Reads the next byte of data. It returns the byte as an integer, or `-1` if the end of the stream is reached.
- `int read(byte[] b)`: Reads up to `b.length` bytes of data into an array. Returns the number of bytes read, or `-1` if the end of the stream is reached.
- `long skip(long n)`: Skips over `n` bytes of data.
- `int available()`: Returns the number of bytes that can be read without blocking.
- `void close()`: Closes the stream and releases any resources.

Common Subclasses of `InputStream`:

- `FileInputStream`: Reads bytes from a file.
- `ByteArrayInputStream`: Reads bytes from a byte array.
- `BufferedInputStream`: Adds buffering to an existing input stream to improve performance.
- `ObjectInputStream`: Reads objects that have been serialized.

Example (Reading from a file using `FileInputStream`):

```
import java.io.*;

public class InputStreamExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("example.txt")) {
            int byteRead;
            while ((byteRead = fis.read()) != -1) {
                System.out.print((char) byteRead);  // Print each byte as a
character
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. OutputStream (Byte Output):

The `OutputStream` class is the parent class for all output streams in Java. It is used for writing byte data to a destination such as a file, network, or memory. Like `InputStream`, it has several subclasses for different data output scenarios.

Important Methods in `OutputStream`:

- `void write(int b)`: Writes a single byte to the output stream. The byte is provided as an integer value.
- `void write(byte[] b)`: Writes an array of bytes to the output stream.
- `void write(byte[] b, int off, int len)`: Writes a part of the byte array (from `off` to `len`).
- `void flush()`: Flushes the output stream, ensuring that all data is written out.
- `void close()`: Closes the output stream and releases any resources.

Common Subclasses of `OutputStream`:

- `FileOutputStream`: Writes bytes to a file.
- `ByteArrayOutputStream`: Writes bytes to a byte array (useful for in-memory operations).
- `BufferedOutputStream`: Adds buffering to an output stream to improve performance.
- `ObjectOutputStream`: Writes objects to an output stream (used for serialization).

Example (Writing to a file using `FileOutputStream`):

```
import java.io.*;

public class OutputStreamExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("output.txt")) {
            String message = "Hello, Java I/O!";
            fos.write(message.getBytes());  // Convert string to bytes and
write
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Theory

Key Differences Between `InputStream` and `OutputStream`:

Aspect	InputStream	OutputStream
Purpose	Used for reading byte data.	Used for writing byte data.
Primary Methods	`read()`, `read(byte[])`, `skip()`, `close()`	`write()`, `write(byte[])`, `flush()`, `close()`
Direction of Data	Reads from a source (e.g., file, network).	Writes to a destination (e.g., file, network).
Common Subclasses	`FileInputStream`, `BufferedInputStream`, `ObjectInputStream`	`FileOutputStream`, `BufferedOutputStream`, `ObjectOutputStream`

2. ReadingandWritingDataUsingStreams

reading and writing data using streams is a core concept for handling I/O (Input/Output) operations. Streams allow you to read data from a source (like a file or network) and write data to a destination (like a file or console). Below is an overview of how you can read and write data using various types of streams (byte streams and character streams).

1. Reading and Writing Using Byte Streams (using `InputStream` and `OutputStream`):

Byte streams are used for reading and writing binary data (like images, audio files, or any raw data). They handle 8-bit data, making them suitable for all kinds of data, not just text.

Reading Data Using Byte Streams:

To read data from a source (such as a file) using a byte stream, we typically use `FileInputStream` or other subclasses of `InputStream`.

Example: Reading a file using `FileInputStream`:

```
import java.io.*;

public class ReadFileExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("input.txt")) {
            int byteRead;
            while ((byteRead = fis.read()) != -1) {
                System.out.print((char) byteRead);  // Print each byte as a 
character
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Writing Data Using Byte Streams:

To write data to a destination, like a file, we use `FileOutputStream` or other subclasses of `OutputStream`.

Example: Writing data to a file using `FileOutputStream`:

```
import java.io.*;

public class WriteFileExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("output.txt")) {
            String message = "Hello, World!";
            fos.write(message.getBytes());  // Convert string to bytes and 
write to the file
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. Buffered Streams for More Efficient I/O:

Buffered I/O streams like `BufferedInputStream` and `BufferedOutputStream` provide better performance by reducing the number of read and write operations. They use a buffer to store data before actually reading or writing it, which minimizes the overhead of I/O operations.

Reading with Buffered Streams:

You can wrap a `FileInputStream` in a `BufferedInputStream` to improve performance by buffering the input.

Example: Buffered file reading using `BufferedInputStream`:

```
import java.io.*;

public class BufferedReadExample {
    public static void main(String[] args) {
        try (BufferedInputStream bis = new BufferedInputStream(new 
FileInputStream("input.txt"))) {
            int byteRead;
            while ((byteRead = bis.read()) != -1) {
                System.out.print((char) byteRead);  // Print each byte as a 
character
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Writing with Buffered Streams:

Similarly, you can wrap a `FileOutputStream` in a `BufferedOutputStream` to write data more efficiently.

Example: Buffered file writing using `BufferedOutputStream`:

```
import java.io.*;

public class BufferedWriteExample {
    public static void main(String[] args) {
        try (BufferedOutputStream bos = new BufferedOutputStream(new 
FileOutputStream("output.txt"))) {
            String message = "Hello, Buffered World!";
            bos.write(message.getBytes());  // Write bytes using buffered 
stream
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3. Reading and Writing Using Character Streams (using `Reader` and `Writer`):

Character streams are used for handling character data (text) and are more appropriate when dealing with textual data. They automatically handle the encoding and decoding of characters.

Reading Data Using Character Streams:

To read text data from a file, we typically use `FileReader` (which is a subclass of `Reader`).

Example: Reading a file using `FileReader`:

```
import java.io.*;

public class ReadTextFileExample {
    public static void main(String[] args) {
        try (FileReader fr = new FileReader("input.txt")) {
            int charRead;
            while ((charRead = fr.read()) != -1) {
                System.out.print((char) charRead);  // Print each character
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Writing Data Using Character Streams:

To write text data to a file, we use `FileWriter` (which is a subclass of `Writer`).

Example: Writing to a file using `FileWriter`:

```
import java.io.*;

public class WriteTextFileExample {
    public static void main(String[] args) {
        try (FileWriter fw = new FileWriter("output.txt")) {
            String message = "Hello, Java I/O!";
            fw.write(message);  // Write the string directly to the file
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4. Combining Buffered Streams with Character Streams:

It's a good practice to use buffered streams with character streams for more efficient I/O when working with large files.

Example: Reading and Writing Text Files Using Buffered Streams:

```
import java.io.*;

public class BufferedFileExample {
    public static void main(String[] args) {
        // Reading from a file using BufferedReader
        try (BufferedReader br = new BufferedReader(new 
FileReader("input.txt"));
             BufferedWriter bw = new BufferedWriter(new 
FileWriter("output.txt"))) {

            String line;
            while ((line = br.readLine()) != null) {
                bw.write(line);  // Write each line to the output file
                bw.newLine();    // Write a new line
            }

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

5. Handling Exceptions and Closing Streams:

Always make sure to handle exceptions properly and close your streams to release system resources.

Example: Properly closing streams using try-with-resources:

```
try (FileInputStream fis = new FileInputStream("input.txt")) {
    // Read from the file
} catch (IOException e) {
    e.printStackTrace();
} // Automatically closes the stream after use
```

3. HandlingFileI/OOperations

Handling file I/O operations in Java involves reading from and writing to files using the Java I/O API. Java provides both byte-based and character-based streams to facilitate file I/O. Here, we will explore how to handle file I/O operations effectively, including reading, writing, and managing file resources in a safe and efficient manner.

1. Basic File I/O in Java

Java provides two main types of streams for file operations:

- **Byte Streams**: Use classes like `FileInputStream` and `FileOutputStream` to handle raw byte data.
- **Character Streams**: Use classes like `FileReader` and `FileWriter` to handle text data (characters).

Let's go through examples of how to perform basic file I/O operations.

2. Reading from a File

You can read data from a file using byte-based or character-based streams. Below are examples for both approaches.

Using FileInputStream (Byte Stream)

```
import java.io.*;

public class ReadFileExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("input.txt")) {
            int byteRead;
            while ((byteRead = fis.read()) != -1) {
                System.out.print((char) byteRead);  // Print byte as
character
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Using FileReader (Character Stream)

```
import java.io.*;

public class ReadFileExample {
    public static void main(String[] args) {
        try (FileReader fr = new FileReader("input.txt")) {
            int charRead;
            while ((charRead = fr.read()) != -1) {
                System.out.print((char) charRead);  // Print each character
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3. Writing to a File

You can write data to a file using either byte-based or character-based streams, depending on the type of data you are writing.

Using FileOutputStream (Byte Stream)

```
import java.io.*;

public class WriteFileExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("output.txt")) {
            String message = "Hello, World!";
            fos.write(message.getBytes());  // Write bytes to the file
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Using FileWriter (Character Stream)

```
import java.io.*;

public class WriteFileExample {
    public static void main(String[] args) {
        try (FileWriter fw = new FileWriter("output.txt")) {
            String message = "Hello, Java I/O!";
            fw.write(message);  // Write characters to the file
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4. Using Buffered Streams for Efficiency

Buffered streams (e.g., `BufferedInputStream`, `BufferedOutputStream`, `BufferedReader`, and `BufferedWriter`) improve performance by reducing the number of I/O operations. These streams use an internal buffer to store data before actually reading or writing it, making them especially useful for handling large files.

Buffered File Reading (Using BufferedReader)

```
import java.io.*;

public class BufferedFileReaderExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new
FileReader("input.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);  // Print each line from the file
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Buffered File Writing (Using BufferedWriter)

```
import java.io.*;

public class BufferedFileWriterExample {
    public static void main(String[] args) {
        try (BufferedWriter bw = new BufferedWriter(new
FileWriter("output.txt"))) {
            String message = "Hello, Buffered World!";
            bw.write(message);  // Write text to the file efficiently
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

5. Handling File I/O Exceptions

When working with file I/O in Java, it's important to handle `IOExceptions` properly. These exceptions can occur due to various reasons, such as when a file is not found, permission issues, or an I/O error during reading or writing. Java provides several exception handling mechanisms to deal with these situations.

Example of Exception Handling in File I/O

```
import java.io.*;

public class FileIOExceptionExample {
    public static void main(String[] args) {
        try {
            FileReader fr = new FileReader("nonexistent_file.txt");
        } catch (IOException e) {
            System.err.println("File not found or unable to read file: " +
e.getMessage());
        }
    }
}
```

6. Using Try-With-Resources for Automatic Resource Management

The `try-with-resources` statement ensures that resources (such as files, streams, and readers/writers) are automatically closed when they are no longer needed, avoiding resource leaks.

Example: Using Try-With-Resources

```
java
Copy code
import java.io.*;

public class TryWithResourcesExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new
FileReader("input.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);  // Print each line from the file
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

7. File Operations with `java.nio` Package (Java 7 and Later)

Since Java 7, the `java.nio.file` package provides more modern, flexible, and powerful ways to work with files. The `Files` class in the `java.nio.file` package can be used to perform file operations like reading, writing, and copying files.

Reading a File Using `Files.readAllLines()`

```
import java.nio.file.*;
import java.io.*;
import java.util.*;

public class NioFileReadExample {
    public static void main(String[] args) {
        try {
            List<String> lines = Files.readAllLines(Paths.get("input.txt"));
            for (String line : lines) {
                System.out.println(line);  // Print each line
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Writing to a File Using `Files.write()`

```
import java.nio.file.*;
import java.io.*;

public class NioFileWriteExample {
    public static void main(String[] args) {
        try {
            String message = "Hello, NIO World!";
            Files.write(Paths.get("output.txt"), message.getBytes());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

8. File Deletion and Copying with NIO

Java NIO makes it easy to copy, move, and delete files. Here's how you can do it:

Copying a File Using NIO

```
import java.nio.file.*;

public class FileCopyExample {
    public static void main(String[] args) {
        try {
            Path sourcePath = Paths.get("input.txt");
            Path destinationPath = Paths.get("copy.txt");
            Files.copy(sourcePath, destinationPath,
StandardCopyOption.REPLACE_EXISTING);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Deleting a File Using NIO

```
import java.nio.file.*;

public class FileDeleteExample {
    public static void main(String[] args) {
        try {
            Path path = Paths.get("output.txt");
            Files.delete(path);  // Delete the file
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Theory