Java is a high-level, object-oriented programming language developed by Sun Microsystems in 1995 and later acquired by Oracle Corporation. It is designed to be platform-independent, meaning that programs written in Java can run on any operating system that supports the Java Virtual Machine (JVM), a feature commonly referred to as "write once, run anywhere" (WORA).

Java follows an object-oriented programming (OOP) model, which makes code modular, flexible, and reusable. It supports key OOP principles like inheritance, encapsulation, abstraction, and polymorphism. Java is widely used for building applications ranging from mobile apps (especially Android apps), web applications, enterprise-level solutions, and large-scale systems, to distributed applications and cloud services.

Why Choose Java?

1. **Platform Independence**: Java's WORA capability ensures that code written once can run anywhere without modification, making it versatile across different operating systems and platforms.
2. **Large Ecosystem**: Java has a vast ecosystem of libraries, frameworks (like Spring, Hibernate), and development tools that speed up development and provide solutions to common tasks.
3. **Scalability**: Java is used in large-scale enterprise applications because of its scalability and ability to handle large workloads efficiently.

4. **Strong Community Support**: With a large global community, Java benefits from regular updates, extensive documentation, and active forums where developers can seek advice and share knowledge.

5. **Security**: Java provides built-in security features, such as bytecode verification, exception handling, and the Java Security Manager, which helps create secure and stable applications.

6. **Job Market Demand**: Java remains one of the most in-demand languages in the software development industry, especially for enterprise-level applications, backend services, and Android development.

7. **Mature and Reliable**: Being over two decades old, Java is mature and reliable, making it a good choice for projects that need long-term maintenance and stability.

If you're looking for a language with strong versatility, performance, and community support, Java is an excellent choice, especially for cross-platform applications, Android development, and large-scale enterprise systems.

What is the Java Virtual Machine (JVM)?

The **Java Virtual Machine (JVM)** is a key component of the Java programming environment. It is a virtual machine that allows a computer to run **Java programs** regardless of the underlying operating system or hardware platform. JVM is a crucial part of making Java a platform-independent language, supporting the "write once, run anywhere" (WORA) principle.

Key Responsibilities of the JVM:

1. **Bytecode Execution**: The JVM executes Java **bytecode**, which is the intermediate representation of your code after compilation. When you write Java code and compile it, the source code is translated into bytecode, which the JVM can interpret and execute on any machine.

2. **Memory Management**: JVM handles memory allocation and garbage collection automatically, which helps prevent memory leaks and optimize performance by reclaiming memory that's no longer in use.

3. **Platform Independence**: Since the JVM abstracts away the underlying hardware, it allows Java programs to run on any system that has a compatible JVM, regardless of the operating system (Windows, macOS, Linux, etc.).

4. **Security**: JVM includes a security manager that helps define access levels for Java applications, preventing unauthorized access to system resources.

5. **Performance Optimization**: The JVM incorporates techniques like **Just-In-Time (JIT) compilation**, which compiles bytecode into native machine code during runtime to improve performance.

In short, the JVM is what allows Java to achieve portability, manage memory effectively, and optimize performance, making it a core part of Java's success as a widely-used programming language

---

Steps to Run Java on Your Machine :

The **Java Development Kit (JDK)** is essential for developing and running Java applications because it provides:

1. **Java Compiler (javac)**: Converts Java code into bytecode for execution by the JVM.
2. **Java Runtime Environment (JRE)**: Includes the JVM to run Java applications.
3. **Development Tools**: Offers tools like `javadoc`, `jar`, and `jdb` for compiling, debugging, and packaging Java code.
4. **Standard Libraries**: Gives access to essential Java APIs and libraries for building applications.

Without the JDK, you can't compile or run Java programs on your machine.

1. **Install the Java JDK**

    - Go to the Oracle JDK Downloads page (or you can use an open-source JDK like OpenJDK).
    - Download the appropriate version for your operating system (Windows, macOS, Linux).
    - Follow the installation instructions for your OS.
    - After installation, set the **JAVA_HOME** environment variable:

- **Windows**: Add the path of the JDK to the `JAVA_HOME` environment variable and update the `Path` variable.
- **macOS/Linux**: Add `export JAVA_HOME=/path/to/jdk` to your `.bash_profile` or `.bashrc`, then run `source ~/.bash_profile`.

Verify the installation by running the following command in your terminal:

```
java -version
```

2. **Install Visual Studio Code (VS Code)**
   - Download and install **VS Code** from the official site.
   - After installation, open **VS Code**.
3. **Install the Java Extension Pack in VS Code**
   - Open **VS Code** and go to the **Extensions** tab (or press `Ctrl+Shift+X`).
   - Search for **"Java Extension Pack"**.
   - Install it. This will automatically install several useful extensions for Java development, including support for running and debugging Java applications.
4. **Configure VS Code for Java**
   - Once the **Java Extension Pack** is installed, VS Code will detect the JDK on your machine.
   - Ensure the `JAVA_HOME` variable is set correctly and detected by VS Code.
   - You can open the **Command Palette** (`Ctrl+Shift+P`) and search for "Java: Configure Java Runtime" to ensure the correct JDK is selected.
5. **Create a New Java Project**
   - In VS Code, press `Ctrl+Shift+P` and type "Java: Create Java Project".
   - Choose **No Build Tools** for a basic project or **Maven/Gradle** for more advanced projects.
   - Select a location for the project and create a new folder.
6. **Write a Simple Java Program**
   - In the newly created project folder, create a new file named `Main.java`.
   - Add the following code:

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

7. **Run the Java Program**
    - Right-click inside the `Main.java` file and select **Run Java**.
    - Alternatively, press `F5` to run and debug the program using VS Code's integrated debugging tools.

You should see **"Hello, World!"** printed in the terminal.

Now your Java environment is set up, and you can run Java programs using VS Code!

run java in terminal :

just open your terminal then run the command

```
jshell
```

then you will get a command prompt that you can use to run your java code directly within the terminal

print something :

notion : you can use the print command alone just when dealing with jshell

```
System.out.print(value)
```

Example :

1. print a string [it is necessary to use double quotes , a single quotes will raise an exception]

```
System.out.print("Hello world")
```

quit the jshell :

you can press `Ctrl+D`

print "Hello world" :

1. create a class with the same name as the java file
2. add a `main` function that represents the entry point of the application (it is crucial to pass the `args` to the `main` function)
3. Use the `System.out` to access the `println` function

Example :

```
class Code {
   public static void main(String[] var0) {
      System.out.println("Hello world");
   }
}
```

Run a Java Code:

Java compiles code in two main stages: **compilation** and **execution**. Here's how it works:

1. Writing the Java Code (Source Code)

You write your Java code in a plain text file with the `.java` extension. This file contains human-readable code, known as source code, written using Java syntax.

Example (`HelloWorld.java`):

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

2. Compilation Stage: Converting Source Code to Bytecode

When you compile a Java program, the **Java compiler** (`javac`) translates your source code into **bytecode**. Bytecode is not hardware-specific and can be executed on any machine with a Java Runtime Environment (JRE).

Steps:

- **Compiler (`javac`)**: The Java compiler reads the `.java` file.
- **Syntax Checking**: The compiler checks the source code for errors.
- **Bytecode Generation**: If there are no errors, the compiler generates bytecode (a `.class` file).

Example command:

```
javac HelloWorld.java
```

After running this, a `HelloWorld.class` file is created with bytecode.

3. Execution Stage: Running Bytecode on the JVM

The **Java Virtual Machine (JVM)**, part of the **JRE**, is responsible for running the bytecode. The JVM is platform-independent and works on any system with a JRE installed.

Steps:

- **Class Loader**: The JVM loads the `.class` file into memory.
- **Bytecode Verification**: Ensures the bytecode is safe and valid.
- **JIT Compilation**: The JVM uses a Just-In-Time (JIT) compiler to convert frequently executed bytecode into machine code for faster execution.
- **Execution**: The JVM executes the machine code, handling memory and thread management.

Example command to run the bytecode:

```
java HelloWorld
```

This runs the `HelloWorld.class` file, printing:

```
Hello, World!
```

Key Components:

The **Java Runtime Environment (JRE)** is a key part of the Java ecosystem. Here's a quick summary:

What is the JRE?

- **JRE** stands for **Java Runtime Environment**, and it's the runtime component of Java. It provides the libraries and components needed to run Java applications.

Components of the JRE:

1. **JVM (Java Virtual Machine)**: The core part that runs Java bytecode.
2. **Core Libraries**: Pre-built Java classes (like `java.util`, `java.io`, etc.) that are necessary for running applications.
3. **Other Components**: Such as tools for memory management, exception handling, etc.

What Does the JRE Do?

- The JRE provides the environment in which Java bytecode can be executed. It includes the **JVM** and other resources like libraries that Java programs rely on.
- It does **not** include the Java compiler (`javac`)—that's part of the **JDK** (Java Development Kit).

JRE vs. JDK

- **JRE**: Used for running Java applications (contains the JVM).
- **JDK**: Used for developing Java applications (contains JRE + compiler + other tools).

Summary

- **JRE**: Everything needed to run Java applications.
- **JVM**: The engine within the JRE that executes the bytecode.
- **JDK**: Includes JRE and additional tools for development.

In short, the **JRE** is necessary if you want to **run** Java programs, while the **JDK** is required if you want to **develop** Java programs.

Compilation and Execution Workflow:

1. **Source Code (.java)**: Write the Java program.
2. **Compile** (`javac HelloWorld.java`): Converts the code to bytecode (`HelloWorld.class`).
3. **Bytecode (.class)**: This is platform-independent.
4. **Run** (`java HelloWorld`): The JVM executes the bytecode using JIT for optimization.

Why Bytecode?

- **Portability**: Bytecode can run on any system with a JRE, supporting the "write once, run anywhere" principle.
- **Security**: Bytecode verification protects against malicious code.
- **Performance**: JIT compilation ensures efficient execution.

Summary:

- **Compilation**: Source code → Bytecode via `javac`.
- **Execution**: Bytecode runs on the JVM, optimized by JIT compilation.

Print a string without adding a new line :

```
class Main{
 public static void main(String[] args){
    System.out.print("Hello world")

 }

}
```

Data Types in Java

Data types are a fundamental concept in Java that defines the type of data a variable can hold. Each data type specifies the size and type of values that can be stored, as well as the operations that can be performed on them. Understanding data types is essential for effective programming, as they help in managing memory and ensuring the correctness of operations.

1. Primitive Data Types

Java has eight built-in primitive data types. These data types are predefined and represent the simplest forms of data.

Data Type	Size (in bytes)	Description
`byte`	1	Represents an 8-bit signed integer. Range: -128 to 127.
`short`	2	Represents a 16-bit signed integer. Range: -32,768 to 32,767.
`int`	4	Represents a 32-bit signed integer. Range: -2^31 to 2^31-1.
`long`	8	Represents a 64-bit signed integer. Range: -2^63 to 2^63-1.

Data Type	Size (in bytes)	Description
`float`	4	Represents a single-precision 32-bit IEEE 754 floating-point : ≈ 3.4028235 * 10^38
`double`	8	Represents a double-precision 64-bit IEEE 754 floating-point. ≈ 1.7976931348623157 * 10^308
`char`	2	Represents a single 16-bit Unicode character.
`boolean`	1	Represents one of two values: `true` or `false`.

Example:

```
int age = 30;          // Integer variable
double salary = 50000.50; // Double variable
char initial = 'A';    // Character variable
boolean isActive = true; // Boolean variable
```

2. ** No Primitive Data Types**

We will talk about it later

Variables

In Java, a variable is a container that holds data that can be changed during the execution of a program. Variables are fundamental to programming as they allow you to store and manipulate data effectively. Understanding how to use variables is essential for writing efficient and maintainable Java code.

1. Variable Declaration and Initialization

In Java, variables must be declared before they can be used. The declaration specifies the variable's type and name. Initialization is the process of assigning a value to the variable.

Syntax for declaring and initializing a variable:

```
dataType variableName = initialValue;
```

Example:

```
int age = 25; // Declaration and initialization of an integer variable
float mark = 90.1f; // ensure that the value is not double
long lightSpeedInSecond=300000 ;
char letter='A';
```

2. Types of Variables

Java has three main types of variables:

- **Local Variables**:
  - These are declared within a method, constructor, or block of code.
  - They are created when the method, constructor, or block is entered and destroyed when it is exited.
  - Local variables must be initialized before use.
  - Example:

```
void exampleMethod() {
    int localVar = 10; // Local variable
    System.out.println(localVar);
}
```

- **Instance Variables**:
  - Also known as non-static fields, instance variables are declared inside a class but outside any method or constructor.
  - They are specific to an instance of a class, meaning each object of the class has its own copy of the instance variables.
  - Example:

```
class MyClass {
    int instanceVar; // Instance variable
}
```

- **Static Variables**:
  - These are declared with the `static` keyword within a class but outside any method or constructor.

- Static variables are shared among all instances of a class, meaning they belong to the class itself rather than any individual object.
- Example:

```
class MyClass {
    static int staticVar; // Static variable
}
```

3. Variable Scope

The scope of a variable determines where it can be accessed within the code.

- **Local Variable Scope**: Accessible only within the method or block in which they are declared.
- **Instance Variable Scope**: Accessible to all methods within the class.
- **Static Variable Scope**: Accessible to all methods and instances of the class.

Understanding variable scope is crucial for preventing naming conflicts and ensuring that variables are used correctly within the program.

4. Variable Naming Conventions

When naming variables in Java, follow these conventions:

- Use meaningful names that reflect the variable's purpose (e.g., `studentName`, `totalAmount`).
- Start variable names with a lowercase letter and use camelCase for multi-word names (e.g., `firstName`, `totalScore`).
- Avoid using reserved keywords and special characters in variable names.

5. Type Conversion

Java supports two types of type conversion:

- **Widening Conversion**: Implicit conversion of a smaller primitive type to a larger primitive type (e.g., `int` to `float`). This is done automatically by the compiler.

  Example:

```
int num = 100;
double d = num; // Widening conversion from int to double
```

- **Narrowing Conversion**: Explicit conversion of a larger primitive type to a smaller primitive type (e.g., `float` to `int`). This requires casting and can result in data loss.

  Example:

  ```
  double d = 9.78;
  int num = (int) d; // Narrowing conversion from double to int
  ```

6. Type Checking and Default Values

- **Type Checking**: Java is a statically typed language, meaning that variable types are checked at compile-time. This helps catch type-related errors early.
- **Default Values**: When variables are declared but not initialized, they receive default values:
  - `int`, `short`, `byte`, `long`: `0`
  - `float`: `0.0`
  - `double`: `0.0`
  - `char`: `'\u0000'` (null character)
  - `boolean`: `false` -Primitive Data Types(Reference types): `null`

Conclusion

Variables are an integral part of Java programming. They enable developers to store, manipulate, and access data dynamically. By understanding the different types of variables, their scope, and proper naming conventions, you can write clear and effective Java code.

literal :

1. binary literal :

Use binary to assign an integer variable:

```
int intVariable =0b101
System.out.println(intVariable); // result= 5
```

2. Hexadecimal literal :

Use Hexadecimal to assign an integer variable:

```
int intVariable =0xFF
System.out.println(intVariable); // result= 255
```

3. underscore literal :

Used to clarify the large number format

Example:

```
int intVariable =1_000_000_000
System.out.println(intVariable);// 1000000000
```

4. epsilon literal :

Used to express a number * 10^base

```
 double dblVariable =5e7
System.out.println(dblVariable);// 5 * 10^7
```

5. Increment Literal:

The increment literal is used to increase the value of a variable by one. It is commonly used with numeric types and characters, and can be applied in two ways: post-increment and pre-increment.

1. Post-increment (`var++`):

In **post-increment**, the current value of the variable is used in an expression or operation, and then the variable is incremented after that operation.

- **Syntax**: `var++`
- **Example**:

```
class PostIncrementExample {
    public static void main(String[] args) {
        int num = 5;
        int result = num++; // result gets the value 5, then num becomes 6

        System.out.println("Post-increment result: " + result); // 5
        System.out.println("Value of num after increment: " + num); // 6
    }
}
```

2. Pre-increment (`++var`):

In **pre-increment**, the variable is incremented first, and then the new value is used in the expression or operation.

- **Syntax**: `++var`
- **Example**:

```
class PreIncrementExample {
    public static void main(String[] args) {
        int num = 5;
        int result = ++num; // num becomes 6, and result is 6

        System.out.println("Pre-increment result: " + result); // 6
        System.out.println("Value of num after increment: " + num); // 6
    }
}
```

6. Decrement Literal:

The decrement literal is used to decrease the value of a variable by one. Similar to incrementing, it can be done using post-decrement and pre-decrement operations.

1. Post-decrement (`var--`):

In **post-decrement**, the current value of the variable is used in an expression or operation, and then the variable is decremented after that operation.

- **Syntax**: `var--`
- **Example**:

```
class PostDecrementExample {
    public static void main(String[] args) {
        int num = 5;
        int result = num--; // result gets the value 5, then num becomes 4

        System.out.println("Post-decrement result: " + result); // 5
        System.out.println("Value of num after decrement: " + num); // 4
    }
}
```

2. Pre-decrement (`--var`):

In **pre-decrement**, the variable is decremented first, and then the new value is used in the expression or operation.

- **Syntax**: `--var`
- **Example**:

```
class PreDecrementExample {
    public static void main(String[] args) {
        int num = 5;
        char b = 'B';
        char a = --b; // a=A
        int result = --num; // num becomes 4, and result is 4

        System.out.println("Pre-decrement result: " + result); // 4
        System.out.println("Value of num after decrement: " + num); // 4
    }
}
```

Summary:

- **Post-increment (`var++`)** uses the current value of the variable first, then increments it.
- **Pre-increment (`++var`)** increments the variable first, then uses the new value.

- **Post-decrement (`var--`)** uses the current value of the variable first, then decrements it.
- **Pre-decrement (`--var`)** decrements the variable first, then uses the new value.

Arithmetic operations :

`+` : Addition `-` : Subtraction `*` : Multiplication `/` : Division `%` : Modulo `^` : Bitwise = ((a) base 2) Xor ((b) base 2), Exp: 6 ^ 5 = 0110 ^ 0101 = 011 = 3

Relational operators :

`>` : Grater than `>=` : Grater than or equal `<` : Less than `<=` : Less than or equal `==` : Equal `!=` : Different `!(boolean Expression)` : Not (boolean Expression)

Logical operators :

`&&` : And `||` : Or `^` : `Bitwise` <=> (BooleanExp1 || BooleanExp2) && !(BooleanExp1 && BooleanExp2)

Conditional block [if , else] :

```
if(booleanCondition){
        // do instructions
        }
        else if(booleanCondition2){ // booleanCondition=false

        }
else{ // booleanCondition=false  && // booleanCondition2=false
      // do instructions
        }
```

Example :

```
class code {
    public static void main(String[] args) {
    int x = 14;
    int y = 12;
    int z = 10;
    if (x > y && x > z) System.out.println(x);
    else if (y > z) System.out.println(y);
    else System.out.println(z);

    }
}
```

Ternary operator :

```
result = condition ? returrnedValueIfIsTrue:returrnedValueIfIsFalse;
```

Example :

Example 1 :

```
class code {
    public static void main(String[] args) {
        int x = 5;
        int y = 7;
        int max = x > y ? x:y;
        System.out.println("The max value is " + max);
    }
}
```

Example 2 :

```
class code {
    public static void main(String[] args) {
        int x = 5;
        int y = 7;
        int z = 10;
        int max = x > y ? x > z ? x : z : y > z ? y : z;
        System.out.println("The max value is " + max);
    }
}
```

Switch statement :

```
switch(n){
 case v1 : // v1 ==n ?
    //instructions
    break;
  case v2 : // v2 ==n ?
    //instructions
    break;
     case v3 : // v3 ==n ?
    //instructions
    break;
    default : // else
    // instructions
}
```

Example :

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

        int n = 7;
        switch (n) {
            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("Error");
```

```
        }
    }
}
```

Combine multiple cases :

```
switch(n){
 case v1,v2 : // v1 ==n || v2 ==n ?
    //instructions
    break;
     case v3 : // v3 ==n ?
    //instructions
    break;
    default : // else
    // instructions
}
```

Using a switch without breaks

Example :

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

        int n = 7;
        switch (n) {
            case 1 -> {
                System.out.println("Monday");
                System.out.println("Monday is the first day in the week");
            }

            case 2 -> System.out.println("Tuesday");

            case 3 -> System.out.println("Wednesday");

            case 4 -> System.out.println("Thursday");

            case 5 -> System.out.println("Friday");

            case 6 -> System.out.println("Saturday");

            case 7 -> System.out.println("Sunday");

            default ->
                System.out.println("Error");
```

```
        }
    }
}
```

Using a switch to return a value

Example :

Methode 1 :

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

        int n = 7;
        String result = switch (n) {
            case 1 -> "Monday";

            case 2 -> "Tuesday";

            case 3 -> "Wednesday";

            case 4 -> "Thursday";

            case 5 -> "Friday";

            case 6 -> "Saturday";

            case 7 -> "Sunday";

            default -> "Error";

        };
        System.out.println(result);
    }
}
```

Method 2 :

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

        int n = 7;
        String result = switch (n) {
            case 1 : yield  "Monday";

            case 2 : yield "Tuesday";
```

```
            case 3 : yield "Wednesday";

            case 4 : yield "Thursday";

            case 5 : yield "Friday";

            case 6 : yield "Saturday";

            case 7 : yield "Sunday";

            default : yield "Error";

        };
        System.out.println(result);
    }
}
```

Loops :

It is a way to repeat a piece of code as long as a condition is valid

While loop :

Repeat instructions[within the loop] as long as the condition is valid

```
while(condition){
         // instructions
      }
```

While loop :

Repeat instructions[within the loop] as long as the condition is valid [run at least one time]

```
do{
     // instructions
  }while(condition);
```

Example :

```
class code {
    public static void main(String[] args) {
        do {
            System.out.println("Hello World");
        } while (5 > 6);
       // Output   : Hello world
    }
}
```

for loop

```
 for(int i =start , conditionToStop,stepsPerCycle){
  //instructions
 }
```

Example :

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

        for (int i = 4; i >= 0; i--) {
            System.out.println(i);
        }

    }
}
```

object-oriented-programming (OOP):

object is a set of Properties and behaviors

class :

it's a blueprint of an object

```
class className{
    int attr1;
    int attr2;
```

```
    // empty constructor :
    public  className(){

    }
    // parametrized constructor :
    public className(int attr1,int attr2){
        this.attr1=attr1;
        this.attr2=attr2;
    }
}
```

Array

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

        // define an array of fore instances
        int[] nums = {1,2,3,4};
        int[] nums2= new int[4]; // default values :{0,0,0,0}

        // access an element of an array :
        nums[1]=10;

        // get array size :
        int size=nums.length;

        // iterate over an array :
        // using enhanced for loop  :
        for( int item : nums){
            System.out.println(item);
        }

        // using for loop :
        for( int i =0 ; i < nums.length ; i++ ){
            System.out.println(nums[i]);
        }


    }
}
```

Multi Dimensional Array

```
class code {
    public static void main(String[] args) {
       //declare a multi dimensional array :
        int nums[][] = new int[3][4];
```

```
      // iterate over a 2d array :
         for(int i = 0 ; i <nums.length ;i++){
            System.out.print("{");
            for(int j = 0 ; j <nums[i].length ; j++){
                // access an element of a 2d array :
                 System.out.print(nums[i][j] + ",");

            }
            System.out.print("}");
            System.out.println();
        }

        // using enhanced for loop  :
        for(int[] line :nums){
            System.out.print("{");
            for(int column : line){
                System.out.print(column + ",");

            }
            System.out.print("}");
            System.out.println();
        }


    }
}
```

Array of object in java :

Example :

```
class Student {
    int rollno;
    String name;
    int marks;

}
class code {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.marks = 10;
        s1.name = "John";
        s1.rollno = 10;

        Student[] arrStudents = new Student[2];
        arrStudents[0] = s1;
        arrStudents[1] = new Student();
        for (Student item : arrStudents) {
            System.out.println(item.name);
        }
```

```
    }
}
```

Here's a structured and organized version of your content:

Differences Between `String` and `StringBuffer` in Java

In Java, `String` and `StringBuffer` both represent sequences of characters but have distinct characteristics and use cases.

---

1. Immutability vs. Mutability

- **`String`**: Immutable, meaning once a `String` object is created, it cannot be changed. Any operation that modifies a `String` (e.g., concatenation) creates a new `String` object.
- **`StringBuffer`**: Mutable, allowing modifications to the existing sequence of characters without creating a new object. This makes it ideal for frequent changes like appending or deleting characters.

2. Performance

- **`String`**: Due to immutability, each modification (e.g., concatenation) creates a new `String` object, which can impact memory and performance, especially when repeated.
- **`StringBuffer`**: More efficient for operations that involve repeated modifications, as it modifies the original object directly. Methods like `append()` add characters to the existing sequence, which conserves memory.

3. Thread Safety

- **`String`**: Since it’s immutable, `String` is inherently thread-safe.
- **`StringBuffer`**: Synchronized, making it thread-safe for concurrent access, though with a slight performance cost due to synchronization.

4. Usage Scenarios

- `String`: Suitable for text that doesn't change frequently or requires minimal modifications.
- `StringBuffer`: Suitable for scenarios where strings undergo frequent changes, like in loops or dynamic concatenations.

5. StringBuilder vs. StringBuffer

- `StringBuilder`: Similar to `StringBuffer` but without synchronization, making it faster in single-threaded contexts.

Code Examples

Using `String`

```
String str = "Hello";
str += " World";  // Creates a new String object each time you modify it.
System.out.println(str); // Output: Hello World

String s1 = "Hello";
String s2 = s1 + " World";  // Creates a new String in the heap, not in the
constant pool
/*
 String Concatenation Creates a New Object:
 In Java, s1 + " World" triggers the creation of a new String object because String
objects are immutable.
 The + operator internally uses a StringBuilder to concatenate, then converts the
result back into a String,
 resulting in a new String object in the heap.
 */
```

Using `StringBuffer`

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

Memory Management and the String Constant Pool

The `StringBuffer` class does **not** use the **String Constant Pool**. Instead, it creates a separate object in the **heap memory**.

1. String Constant Pool

- The String Constant Pool is a special area within the heap to optimize memory usage for `String` literals.
- When a `String` literal (e.g., `String str = "Hello";`) is created, Java checks the pool to see if an identical string already exists. If it does, Java reuses the reference instead of creating a new object.
- This optimization only applies to `String` literals or strings interned with the `.intern()` method.

2. StringBuffer in Heap Memory

- Each `StringBuffer` object is allocated memory directly in the **heap** and doesn't interact with the String Constant Pool.
- Since each `StringBuffer` object is mutable and distinct, pooling would be ineffective, as modifying one reference would impact all other references to that pooled object.

3. StringBuffer and Internal Strings

- When performing operations, `StringBuffer` may contain a `String` internally.
- For instance, calling `toString()` on a `StringBuffer` creates a new `String` object based on the buffer's content. This `String` resides in the heap and can enter the constant pool if interned.

Memory Example

```
StringBuffer sb1 = new StringBuffer("Hello");
StringBuffer sb2 = new StringBuffer("Hello");

// sb1 and sb2 are distinct objects in the heap
System.out.println(sb1 == sb2);  // false

String str1 = sb1.toString();
String str2 = sb2.toString();

// str1 and str2 are new, separate `String` objects in the heap
```

```
System.out.println(str1 == str2); // false

// Interning the strings
String internedStr1 = str1.intern();
String internedStr2 = str2.intern();

// Now they refer to the same constant in the pool
System.out.println(internedStr1 == internedStr2); // true
```

Summary

- `String`: Use for immutable sequences.
- `StringBuffer`: Use for mutable, thread-safe sequences with frequent modifications.
- `StringBuilder`: Use for mutable strings in single-threaded contexts.

Static members :

Example :

```
class Mobile{
     String brand;
     int price;
     String network;
     static String name="Smartphone";


     Mobile(String brand, int price, String network){
         this.brand = brand;
         this.price = price;
         this.network = network;
         String name;
         name="Smartphone";
         // access name using class to prevent shadowing by the local variable
`name
         Mobile.name=name;

     }

final void show(){
    System.out.println("\n"+ brand);
    System.out.println(price);
    System.out.println(network);
```

```
    }
    public static  void printName(){
         System.out.println( "\nname : "+name);
    }

}
class code {
    public static void main(String[] args) {

        Mobile obj1 = new Mobile("iphone",50,"nice");
        Mobile obj2 = new Mobile("Summsung",50,"nice");
        Mobile obj3 = new Mobile("iphone",50,"nice");
         obj1.show();
         obj2.show();
         obj3.show();
         Mobile.printName();

    }
}
```

Static block :

It's an area that you can use to initialize your static variables in a class :

Example :

```
class Mobile {
    static String name;

    static {
        name = "Smartphone";
    }

    String brand;
    int price;
    String network;

    Mobile(String brand, int price, String network) {
        this.brand = brand;
        this.price = price;
        this.network = network;
        String name;
        name = "Smartphone";
        // access name using class to prevent shadowing by the local variable `name
        Mobile.name = name;

    }

    public static void printName() {
        System.out.println("\nname : " + name);
```

```
    }


}
```

load a class :

```
    Class.forName("packageName.className");
```

The levels of access of the class members [Access Modifiers] :

1. **public**: Accessible from anywhere.
2. **protected**: Accessible in the same package and by subclasses.
3. **package-private** (default): Accessible in the same package (no modifier needed).
4. **private**: Accessible only within its own class.

Naming convention :

1. using camelCase
2. class should always start with a capital letter
3. constants should be all capital with the using of snake casing
4. variables[not constant] and methods should always start with a small letter

Anonymous object :

it's an object without a name Example :

```
class A {
    public A() {
        System.out.println("\nobject created");
    }

    public void show() {
        System.out.println("show object");
    }
```

```
}

class code {
    public static void main(String[] args) {
        new A().show();
    }
}
```

Inheritance

Example :

```
 class Calc{
    public  int add(int a,int b){
        return a+b;
    }
    public  int sub(int a,int b){
        return a-b;

    }
    public  int mul(int a,int b){
        return a*b;
    }
    public  int div(int a,int b){
        return a/b;
    }

}

public class AdvCalc  extends Calc{

    public int mod(int a, int b) {
        return a % b;
    }

}
class Main {
    public static void main(String[] args) {
        // base/super/parent class
        Calc cal = new Calc();
        int sum = cal.add(5,4);
        System.out.println( "5 + 4 = "+ sum);

        int mul = cal.mul(5,4);
        System.out.println( "5 * 4 = "+  mul);

        int div = cal.div(5,4);
        System.out.println("5 / 4 = "+  div);

        // sub/derived/child class
        AdvCalc advCalc = new AdvCalc();
```

```
        int mod = advCalc.mod(5,4);
        System.out.println( "5 % 4 = "+ mod);


    }
}
```

Types of inheritance :

1. **Single level Inheritance** : class [B] inherit from class [A]
2. **Multi level Inheritance** : class [B] inherit from class [A]. class [c] inherit from class [B]
3. **Multiple Inheritance** : class [c] inherit from class [A] and [B].

Super and this keyword :

1. **super** : it's an object reference of the super class
2. **this** : it's an object reference of the current class

Example :

```
class A {
    int a;
    A() {
        System.out.println(" in A");
    }

    A(int a) {
        this.a = a;
        System.out.println(" in A [a] ");
    }
}

class B extends A {

    B() {
        System.out.println(" in B");
    }

    B(int a) {
        super(a);
        System.out.println(" in B [a] ");
    }
}
```

Method Overriding

it's represent the operation of redefine a method from the base class in the sub class
Example :

```
package code;

class A {

    int a;

    A(int a) {
        this.a = a;
    }
    public void show(){
        System.out.println(" in A.show()");
    }
}

class B extends A {
    B(int a) {
        super(a);

    }
    @Override
    public void show(){
        System.out.println(" in B.show()");
    }
}

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

        B b = new B(5);
        b.show();


    }
}
```

In Java, `import` statements allow you to bring classes, interfaces, or static members from other packages into your code, making them accessible without needing to reference the fully qualified names. Here's a breakdown of how `import` works in Java, including its usage, types, and some best practices.

1. The Basic Structure of `import`

The `import` statement is used to declare that you are using a specific class, interface, or package in your code. Typically, it comes after the package declaration (if present) and before the class definition.

Syntax:

```
import package_name.ClassName;
import package_name.*;
import static package_name.ClassName.staticMember;
import static package_name.ClassName.*;
```

2. Types of Import Statements

Java provides different types of import statements:

a. Single-Type Import

This is the most commonly used import, where you import a specific class or interface.

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

This allows you to use `ArrayList` directly in your code without needing to fully qualify it with `java.util.ArrayList`.

b. On-Demand (Wildcard) Import

Using `*` in an import statement lets you import all classes and interfaces from a package. However, it **does not import subpackages**.

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

This brings all classes and interfaces in `java.util` into scope (like `ArrayList`, `HashMap`, `Date`, etc.). It's important to note that wildcard imports don't affect performance since the JVM handles them efficiently, but excessive use can lead to ambiguity and lower readability.

c. Static Import

Static import allows you to import static members (fields and methods) of a class so they can be used directly without qualifying them with the class name.

```
import static java.lang.Math.PI;
import static java.lang.Math.*;
```

With `import static`, you can use `PI` and `abs()` directly rather than `Math.PI` or `Math.abs()`.

3. Import Rules

- **Java.lang Package**: Classes from `java.lang` (like `String`, `System`, `Math`) are imported automatically, so you don't need an import statement for them.
- **Fully Qualified Names**: If you don’t use an import statement, you can still use any class by its fully qualified name (e.g., `java.util.ArrayList`), though it can make code verbose.
- **No Multiple Imports for the Same Class**: If you import a class using both single-type import and on-demand import, the compiler ignores the redundancy without errors.

4. Examples of Import Usage

Example 1: Basic Import

```
import java.util.ArrayList;

public class Example {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Hello");
        System.out.println(list.get(0));
    }
}
```

Example 2: Wildcard Import

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

```
public class Example {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        HashMap<String, String> map = new HashMap<>();
    }
}
```

Example 3: Static Import

```
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;

public class Example {
    public static void main(String[] args) {
        double radius = 5;
        double circumference = 2 * PI * radius;
        System.out.println("Circumference: " + circumference);
        System.out.println("Square Root of 4: " + sqrt(4));
    }
}
```

5. Best Practices

- **Avoid Overuse of Wildcard Imports**: While convenient, wildcard imports can lead to ambiguity if two packages contain classes with the same name. Prefer specific imports to improve code readability.
- **Use Static Import Sparingly**: Static import can make code cleaner by removing class references, but overuse may reduce readability, especially for common methods.
- **Organize Imports**: Modern IDEs help organize and manage imports. Remove unused imports to keep code clean, and group imports logically, often with project-specific imports last.

6. Import Optimization in IDEs

Most IDEs, like IntelliJ IDEA or Eclipse, automatically manage imports. They can:

- Add necessary imports when you reference a class not currently imported.
- Optimize imports by removing unused ones.
- Offer settings to avoid wildcard imports.

In Java, a **package** is a way to group related classes and interfaces together in a structured way, providing better code organization, access control, and namespace management. Packages can have different levels or **hierarchies**, allowing developers to organize code into nested structures. Here's an in-depth look at packages, including creating multi-level packages, accessing them, and best practices.

1. What is a Package?

A package in Java is a namespace that organizes classes and interfaces. Java has two main types of packages:

- **Built-in packages** (e.g., `java.util`, `java.io`, etc.).
- **User-defined packages** created by developers for organizing code.

2. Creating Packages

A package is created using the `package` keyword, and it must be the first statement in a Java file (excluding comments). The package structure mirrors the folder structure of the file system.

Basic Example

```
package mypackage;

public class MyClass {
    public void display() {
        System.out.println("Hello from MyClass in mypackage.");
    }
}
```

In this example, the file should be saved in a directory named `mypackage`.

3. Multi-Level (Hierarchical) Packages

Java supports nested package structures, allowing you to create packages within packages, creating a multi-level hierarchy. A **multi-level package** structure might look like this:

```
com
└── mycompany
    ├── project
    │   ├── service
    │   └── model
```

Here:

- `com.mycompany.project.service` and `com.mycompany.project.model` are multi-level packages.
- The folders match the package structure.

Creating Multi-Level Packages

To define a class in a nested package, specify the full package hierarchy at the top of your Java file:

```java
package com.mycompany.project.service;

public class ServiceClass {
    public void printService() {
        System.out.println("This is a service class.");
    }
}
```

4. Importing Multi-Level Packages

Once you have classes organized in a multi-level package structure, you can import them using their fully qualified names.

Importing a Specific Class

```java
import com.mycompany.project.service.ServiceClass;
```

Importing All Classes in a Multi-Level Package

```
import com.mycompany.project.service.*;
```

5. Example of Using Multi-Level Packages

Suppose you have the following package structure with classes `ServiceClass` and `ModelClass` in different sub-packages:

```
com
└── mycompany
    └── project
        ├── service
        │   └── ServiceClass.java
        └── model
            └── ModelClass.java
```

Here's how to use these classes in another class:

```
import com.mycompany.project.service.ServiceClass;
import com.mycompany.project.model.ModelClass;

public class Main {
    public static void main(String[] args) {
        ServiceClass service = new ServiceClass();
        service.printService();

        ModelClass model = new ModelClass();
        model.printModel();
    }
}
```

6. Access Control in Multi-Level Packages

Java packages also help control access between classes:

- **Public** classes and members are accessible from any package.
- **Protected** members are accessible to classes in the same package and subclasses.
- **Default (package-private)** members are accessible only within the same package.
- **Private** members are accessible only within the class itself.

7. Best Practices for Multi-Level Packages

1. **Use Meaningful Names**: Each level of the package should represent a logical grouping.
    - Example: `com.mycompany.project.controller` could contain controller classes, while `com.mycompany.project.service` could contain service classes.
2. **Avoid Too Many Levels**: Over-nesting can make packages harder to navigate. Usually, three or four levels are enough.
3. **Consistent Naming Conventions**: Use lowercase letters for package names (Java standard), separating words with dots.
4. **Reflect Project Structure**: Organize packages based on your project's features or components (e.g., `service`, `controller`, `model`, `repository`).
5. **Separate External Libraries**: If integrating third-party libraries, consider putting them in their own package (e.g., `com.mycompany.project.external`).

8. Example: Real-World Multi-Level Package Structure

Suppose you're building an e-commerce application. Here's how a multi-level package structure might look:

```
com
└── ecommerce
    ├── auth
    │   ├── LoginService.java
    │   └── UserService.java
    ├── products
    │   ├── Product.java
    │   └── ProductService.java
    ├── orders
    │   ├── Order.java
    │   └── OrderService.java
    └── util
        └── Utilities.java
```

In this structure:

- `com.ecommerce.auth` handles authentication-related classes.
- `com.ecommerce.products` handles product-related classes.
- `com.ecommerce.orders` manages orders.
- `com.ecommerce.util` includes utility classes.

Polymorphism in Java is a fundamental concept in object-oriented programming that allows methods to perform differently based on the context, particularly the type of object calling the method. Derived from Greek, *polymorphism* means "many forms." In Java, polymorphism enables a single method or class to handle different types or act in various ways. It helps in achieving flexibility, code reusability, and maintainability.

Here's a detailed guide on polymorphism, covering its types, examples, and usage in Java.

1. Types of Polymorphism in Java

Java has two main types of polymorphism:

1. **Compile-Time Polymorphism (Static Binding)**
2. **Run-Time Polymorphism (Dynamic Binding)**

2. Compile-Time Polymorphism (Method Overloading)

Compile-time polymorphism, also known as *static polymorphism* or *method overloading*, occurs when multiple methods in the same class have the same name but different parameters. The method to call is determined at compile-time based on the method signature.

Key Features:

- **Method Overloading**: Defining multiple methods with the same name but different parameter lists (either by number, type, or order of parameters).
- **Compile-Time Binding**: The method to be executed is determined during compilation.

Example of Method Overloading

```java
class Printer {
    // Method to print an integer
    public void print(int num) {
        System.out.println("Integer: " + num);
    }

    // Method to print a string
    public void print(String str) {
        System.out.println("String: " + str);
    }

    // Method to print two numbers
    public void print(int num1, int num2) {
        System.out.println("Two numbers: " + num1 + " and " + num2);
    }
}

public class Main {
    public static void main(String[] args) {
        Printer printer = new Printer();
        printer.print(10);          // Calls print(int num)
        printer.print("Hello");      // Calls print(String str)
        printer.print(5, 15);        // Calls print(int num1, int num2)
    }
}
```

In this example:

- The `Printer` class has overloaded `print` methods.
- The correct method is chosen based on the arguments passed at compile time.

3. Run-Time Polymorphism (Method Overriding)

Run-time polymorphism, also known as *dynamic polymorphism* or *method overriding*, allows a subclass to provide a specific implementation for a method that is already defined in its superclass. The decision about which method to invoke is made at runtime, based on the object's actual type.

Key Features:

- **Method Overriding**: A subclass provides a specific implementation of a method that is already defined in its superclass.
- **Run-Time Binding**: The method call is resolved at runtime.

- **Inheritance and Upcasting**: Run-time polymorphism requires inheritance and usually involves a superclass reference pointing to a subclass object.

Example of Method Overriding

```
class Animal {
    public void sound() {
        System.out.println("This is an animal sound");
    }
}

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

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

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

        myDog.sound();               // Calls Dog's sound method
        myCat.sound();               // Calls Cat's sound method
    }
}
```

In this example:

- `Animal` has a method `sound`, which is overridden in `Dog` and `Cat`.
- The `sound` method behaves differently depending on the actual type of the object (`Dog` or `Cat`), determined at runtime.

4. Upcasting and Polymorphism

Upcasting is essential for achieving polymorphism. Upcasting is the process of treating a subclass object as an instance of its superclass. This allows us to write code that works with the superclass while still leveraging the subclass's specific behavior.

```
Animal animal = new Dog(); // Upcasting Dog to Animal
animal.sound();            // Calls Dog's overridden sound() method
```

In this case:

- `animal` is of type `Animal`, but at runtime, it refers to a `Dog` object.
- This enables polymorphic behavior by calling the overridden `sound` method of `Dog`.

5. Advantages of Polymorphism

- **Code Reusability**: Common code can be defined in the superclass and reused by subclasses.
- **Extensibility**: New functionality can be added by defining new subclasses without modifying existing code.
- **Flexibility**: Polymorphism provides the flexibility to call methods that have the same name but different behaviors.

6. Disadvantages of Polymorphism

- **Complexity**: Understanding and debugging polymorphic code can sometimes be challenging, especially in complex hierarchies.
- **Performance Overhead**: Run-time polymorphism may have a slight performance overhead due to dynamic method dispatch.
- **Risk of Inappropriate Overriding**: Misuse of overriding can lead to unexpected behaviors, especially when not following the principles of Liskov Substitution.

Summary

Polymorphism in Java is a powerful concept that allows methods to perform differently based on the calling object's type. With compile-time polymorphism (method overloading) and run-time polymorphism (method overriding), Java allows for flexibility, scalability, and maintainable code.

Final keyword :

1. **Variables** : used to declare a constant variable
2. **functions** : used to declare a function that can't be overwriting in a subclass
3. **class** : used to declare a class that can't be inherited Example :

```
final class A {
    // You can't change the value of a
    final int a = 5;

    public void show() {

        System.out.println(a);
    }

}

//Error : class B can not inherit class A because it's constant
class B extends A {
    final int b = 10;

    // Error : You can't override the show function from the A class because it's a
final function
    @Override
    final public void show() {

        System.out.println(a);
        System.out.println(b);
    }
}
```

In Java, the `toString`, `equals`, and `hashCode` methods are essential methods inherited from the `Object` class. They are commonly overridden to provide meaningful and consistent behavior when objects are printed, compared, and stored in data structures like hash-based collections.

1. `toString()` Method

The `toString()` method is used to return a string representation of an object. By default, it returns a string that includes the class name and memory address of the object (not very informative). Overriding `toString()` allows you to provide a human-readable representation of the object's state.

Syntax:

```
@Override
public String toString() {
```

```
    return "Your class string representation";
}
```

Example:

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

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

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

Usage:

```
Person person = new Person("Alice", 30);
System.out.println(person); // Output: Person{name='Alice', age=30}
```

2. `equals()` Method

The `equals()` method is used to compare two objects for equality. The default implementation checks if the references of the objects are the same, not if their content is the same. Overriding `equals()` allows you to define equality based on the actual content (state) of the objects.

Syntax:

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true; // check for self-comparison
    if (obj == null || getClass() != obj.getClass()) return false;

    // Downcast and compare fields
    YourClass other = (YourClass) obj;
    return field1.equals(other.field1) && field2 == other.field2;
}
```

Example:

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

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

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;

        Person person = (Person) obj;
        return age == person.age && name.equals(person.name);
    }
}
```

Usage:

```
Person person1 = new Person("Alice", 30);
Person person2 = new Person("Alice", 30);
System.out.println(person1.equals(person2)); // Output: true
```

3. `hashCode()` Method

The `hashCode()` method returns an integer hash code for an object. It is mainly used in hash-based collections like `HashMap`, `HashSet`, etc. The contract between `equals` and `hashCode` requires that if two objects are considered equal (via `equals`), they must also have the same `hashCode`.

Syntax:

```
@Override
public int hashCode() {
    return Objects.hash(field1, field2);
}
```

Example:

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

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

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }
}
```

Why Override `hashCode()` with `equals()`

In Java, if you override `equals()`, you must also override `hashCode()` to fulfill the **hashCode-Equals contract**. This ensures that equal objects have the same hash code, which is critical for the correct operation of hash-based collections like `HashMap` and `HashSet`.

Example with both `equals` and `hashCode`:

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

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

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;

        Person person = (Person) obj;
        return age == person.age && name.equals(person.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }
}
```

Usage in Hash-Based Collections:

```
Set<Person> people = new HashSet<>();
people.add(new Person("Alice", 30));
people.add(new Person("Alice", 30));

System.out.println(people.size()); // Output: 1, as duplicates are avoided
```

Summary

Method	Purpose	Default Behavior	When to Override
`toString()`	Provides a string representation of an object.	Class name + hash code.	To give a meaningful description of the object.
`equals()`	Checks for logical equality between two objects.	Compares memory addresses.	To define equality based on object content.
`hashCode()`	Generates a hash code for the object, used in hash tables.	Returns a unique integer.	When `equals` is overridden to keep objects equal by `equals` in the same hash bucket.

This setup enables you to use your classes effectively in data structures, print them meaningfully, and compare objects for equality based on their fields.

In Java, **downcasting** is the process of converting a reference of a superclass type to a subclass type. Downcasting is generally used when you have a superclass reference (like an object of a parent class) but need to access specific functionality or properties available only in a subclass.

When to Use Downcasting

1. **Accessing Specific Methods or Properties in the Subclass**: If a superclass reference points to an instance of a subclass, you can only access the methods

and properties defined in the superclass. Downcasting allows access to subclass-specific methods and fields.

```
Animal animal = new Dog(); // Upcasting
Dog dog = (Dog) animal;    // Downcasting
dog.bark(); // Now you can call Dog-specific methods
```

2. **Handling Polymorphism in Collections**: When working with collections of superclass types, downcasting is necessary to access subclass-specific methods on individual elements.

```
List<Animal> animals = new ArrayList<>();
animals.add(new Dog());
animals.add(new Cat());

for (Animal animal : animals) {
    if (animal instanceof Dog) {
        Dog dog = (Dog) animal; // Downcasting
        dog.bark();
    } else if (animal instanceof Cat) {
        Cat cat = (Cat) animal; // Downcasting
        cat.meow();
    }
}
```

Important Considerations with Downcasting

- `instanceof` **Check**: Before downcasting, use `instanceof` to verify that the object is an instance of the desired subclass. This avoids a `ClassCastException` at runtime.

```
if (animal instanceof Dog) {
    Dog dog = (Dog) animal;
    dog.bark();
}
```

- **Avoiding Overuse**: Downcasting can make the code harder to read and maintain. If you often need to downcast, it may indicate a design issue that could benefit from refactoring.

- **Polymorphism as an Alternative**: Favor polymorphism when possible. For example, use method overriding in subclasses to provide specific behavior rather than casting frequently.

In Java, **wrapper classes** are classes that encapsulate primitive data types (like `int`, `double`, `char`, etc.) into objects. Java provides these classes to allow primitives to be used as objects when needed, for example, when working with collections like `ArrayList`, which require objects instead of primitives. Each primitive type has a corresponding wrapper class in Java.

Wrapper Classes for Each Primitive Type

Primitive Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Purpose of Wrapper Classes

1. **Object Requirements**: Some data structures, like `ArrayList`, only work with objects and not primitives. Wrapper classes allow primitives to be used in these scenarios by "wrapping" them in an object.

   ```
   ArrayList<Integer> numbers = new ArrayList<>();
   numbers.add(5); // `int` 5 is wrapped into an `Integer` object
   ```

2. **Utility Methods**: Wrapper classes provide utility methods for conversions and manipulations of data, such as parsing strings into primitives, comparing values,

and obtaining primitive values from strings.

```
int parsedInt = Integer.parseInt("123"); // Converts a String to an int
```

3. **Null Values**: In some cases, it's useful to have the option to represent the absence of a value. Wrapper classes can be `null`, unlike primitives, which have no null value.

4. **Type Conversion**: Wrapper classes provide type conversion methods between different numeric types.

```
Double d = 10.5;
int i = d.intValue(); // Convert Double to int
```

Autoboxing and Unboxing

Autoboxing is the automatic conversion of a primitive to its corresponding wrapper class when needed, and **unboxing** is the automatic conversion of a wrapper class back to its primitive type. This was introduced in Java 5 to make it easier to work with wrappers.

- **Autoboxing Example**:

```
Integer num = 10; // Automatically converts `int` 10 to `Integer` 10
```

- **Unboxing Example**:

```
int n = num; // Automatically converts `Integer` 10 to `int` 10
```

Wrapper classes in Java provide a wide array of **utility functions** that make it easier to work with data conversions, comparisons, parsing, and other common operations. Here's an in-depth look at some of the most commonly used utility functions for each wrapper class:

1. Integer Utility Methods

- `Integer.parseInt(String s)`: Converts a `String` to an `int`.

```
int num = Integer.parseInt("123"); // Returns 123 as an int
```

- `Integer.valueOf(String s)`: Converts a `String` to an `Integer` object.

```
Integer numObj = Integer.valueOf("123"); // Returns Integer 123
```

- `Integer.compare(int x, int y)`: Compares two `int` values.

```
int result = Integer.compare(10, 20); // Returns -1 because 10 < 20
```

- `Integer.toString(int i)`: Converts an `int` to a `String`.

```
String str = Integer.toString(123); // Returns "123"
```

- `Integer.max(int a, int b)` / `Integer.min(int a, int b)`: Finds the maximum or minimum of two integers.

```
int max = Integer.max(10, 20); // Returns 20
```

2. Double Utility Methods

- `Double.parseDouble(String s)`: Converts a `String` to a `double`.

```
double d = Double.parseDouble("45.67"); // Returns 45.67
```

- `Double.valueOf(String s)`: Converts a `String` to a `Double` object.

```
Double dObj = Double.valueOf("45.67"); // Returns Double 45.67
```

- `Double.isNaN(double v)`: Checks if a value is `NaN` (Not-a-Number).

```
boolean isNaN = Double.isNaN(0.0 / 0.0); // Returns true
```

- `Double.compare(double d1, double d2)`: Compares two `double` values.

```
int result = Double.compare(5.5, 3.3); // Returns 1 because 5.5 > 3.3
```

- `Double.toString(double d)`: Converts a `double` to a `String`.

```
String str = Double.toString(45.67); // Returns "45.67"
```

3. Boolean Utility Methods

- `Boolean.parseBoolean(String s)`: Converts a `String` to a `boolean`.

```
boolean b = Boolean.parseBoolean("true"); // Returns true
```

- `Boolean.valueOf(String s)`: Converts a `String` to a `Boolean` object.

```
Boolean bObj = Boolean.valueOf("true"); // Returns Boolean true
```

- `Boolean.logicalAnd(boolean a, boolean b)`: Performs a logical AND operation on two `boolean` values.

```
boolean result = Boolean.logicalAnd(true, false); // Returns false
```

- `Boolean.logicalOr(boolean a, boolean b)`: Performs a logical OR operation on two `boolean` values.

```
boolean result = Boolean.logicalOr(true, false); // Returns true
```

4. Character Utility Methods

- `Character.isDigit(char ch)`: Checks if a character is a digit.

```
boolean isDigit = Character.isDigit('5'); // Returns true
```

- `Character.isLetter(char ch)`: Checks if a character is a letter.

```
boolean isLetter = Character.isLetter('A'); // Returns true
```

- `Character.isUpperCase(char ch)` / `Character.isLowerCase(char ch)`: Checks if a character is uppercase or lowercase.

```
boolean isUpperCase = Character.isUpperCase('A'); // Returns true
```

- `Character.toUpperCase(char ch)` / `Character.toLowerCase(char ch)`: Converts a character to uppercase or lowercase.

```
char upper = Character.toUpperCase('a'); // Returns 'A'
```

5. Long Utility Methods

- `Long.parseLong(String s)`: Converts a `String` to a `long`.

```
long l = Long.parseLong("123456789"); // Returns 123456789
```

- `Long.valueOf(String s)`: Converts a `String` to a `Long` object.

```
Long longObj = Long.valueOf("123456789"); // Returns Long 123456789
```

- `Long.compare(long x, long y)`: Compares two `long` values.

```
int result = Long.compare(123L, 456L); // Returns -1
```

6. Float Utility Methods

- `Float.parseFloat(String s)`: Converts a `String` to a `float`.

```
float f = Float.parseFloat("3.14"); // Returns 3.14f
```

- `Float.valueOf(String s)`: Converts a `String` to a `Float` object.

```
Float fObj = Float.valueOf("3.14"); // Returns Float 3.14
```

- `Float.isNaN(float v)`: Checks if a value is `NaN`.

```
boolean isNaN = Float.isNaN(0.0f / 0.0f); // Returns true
```

7. Byte and Short Utility Methods

- `Byte.parseByte(String s)` / `Short.parseShort(String s)`: Converts a `String` to a `byte` or `short`.

```
byte b = Byte.parseByte("10");   // Returns 10 as a byte
short s = Short.parseShort("20"); // Returns 20 as a short
```

- `Byte.valueOf(String s)` / `Short.valueOf(String s)`: Converts a `String` to a `Byte` or `Short` object.

```
Byte bObj = Byte.valueOf("10");   // Returns Byte 10
Short sObj = Short.valueOf("20"); // Returns Short 20
```

- **Comparison Methods**:

```
int byteComparison = Byte.compare((byte) 10, (byte) 20); // Returns -1
int shortComparison = Short.compare((short) 10, (short) 5); // Returns 1
```

Example Usage in Java Code

Here's an example showing how to use some of these utility methods in practice:

```
import java.util.ArrayList;

public class WrapperUtilityExample {
    public static void main(String[] args) {
        // Parsing strings to primitive values
        int intValue = Integer.parseInt("42");
        double doubleValue = Double.parseDouble("3.14159");
        boolean boolValue = Boolean.parseBoolean("true");

        System.out.println("Parsed int: " + intValue);
        System.out.println("Parsed double: " + doubleValue);
        System.out.println("Parsed boolean: " + boolValue);

        // Comparing values
        int max = Integer.max(5, 10);
        System.out.println("Max of 5 and 10: " + max);

        // Using Character utilities
        char ch = 'a';
        System.out.println("Is 'a' a letter? " + Character.isLetter(ch));
        System.out.println("Uppercase of 'a': " + Character.toUpperCase(ch));

        // Collections with wrapper classes
        ArrayList<Double> doubles = new ArrayList<>();
        doubles.add(Double.valueOf("1.23"));
        doubles.add(Double.parseDouble("4.56")); // Autoboxing will wrap it into
Double

        System.out.println("Doubles list: " + doubles);
    }
}
```

Summary

- **Parsing**: Convert strings to primitive values.
- **Conversion**: Convert between types and from primitives to objects.
- **Comparison**: Compare two values using `compare`, `max`, or `min`.
- **Character Checks**: Validate if a character is a letter, digit, etc.
- **Logical Operations**: `Boolean` has logical operations like `logicalAnd`.

These utilities provide convenience and allow primitives to be more flexible in complex applications.

When to Use Wrapper Classes

- **Working with Collections**: Collections in Java, like `List`, `Map`, and `Set`, require objects. Wrapper classes allow primitive values to be stored in these collections.
- **Using Methods that Require Objects**: Some libraries and frameworks require object parameters rather than primitives.
- **Nullability Requirement**: If you need a variable that can be `null`, you must use a wrapper class instead of a primitive type.
- **Type Conversions**: When converting between types or parsing strings to numbers, wrapper classes provide methods to facilitate these conversions.

Wrapper classes offer flexibility, but keep in mind they add a bit of memory overhead compared to primitives.

Abstract Classes in Java

Definition

An **abstract class** is a class that cannot be instantiated on its own and is designed to be subclassed. It can contain abstract methods (without a body) that must be implemented by subclasses, as well as concrete methods (with a body) that provide default behavior.

Key Characteristics

1. **Cannot be instantiated**: You cannot create an instance of an abstract class directly.
2. **Can have constructors**: Abstract classes can have constructors, which are used to initialize instance variables of the class.
3. **Can have instance variables**: They can contain instance variables (also known as fields) to hold state.
4. **Can have both abstract and concrete methods**: Abstract classes can define both abstract methods (which must be implemented by subclasses) and concrete methods (which have an implementation).

Use Cases

- Use abstract classes when you have a base class that should provide some common functionality but also define specific behaviors that must be implemented by derived classes.
- They are suitable for situations where you want to share code among closely related classes but also enforce a certain structure or behavior.

Example

```
abstract class Animal {
    String name; // Instance variable

    // Constructor
    Animal(String name) {
        this.name = name; // Initializing the instance variable
    }

    abstract void makeSound(); // Abstract method
}

class Dog extends Animal {
    Dog(String name) {
        super(name); // Calling the constructor of the abstract class
    }

    @Override
    void makeSound() {
        System.out.println(name + " says: Bark");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Rex");
        dog.makeSound(); // Output: Rex says: Bark
    }
}
```

Interfaces in Java

Definition

An **interface** is a reference type in Java that is a collection of abstract methods. Interfaces define a contract that classes must follow if they choose to implement the interface.

Key Characteristics

1. **Cannot have constructors**: Interfaces cannot have constructors because they cannot be instantiated.
2. **Cannot have instance variables**: All fields declared in an interface are implicitly `public`, `static`, and `final`, meaning they are constants and cannot hold instance-specific data.
3. **Can have default methods (from Java 8)**: Interfaces can have default methods that provide a default implementation, allowing new methods to be added without breaking existing implementations.
4. **Can extend multiple interfaces**: A class can implement multiple interfaces, which allows for a form of multiple inheritance.

Use Cases

- Use interfaces to define a contract for behaviors that can be implemented by any class, regardless of its position in the class hierarchy.
- Interfaces are ideal for providing common functionality that can be shared among unrelated classes.

Example

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

    // Default method (Java 8 and later)
    default void eat() {
        System.out.println("This animal eats food.");
    }
}

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

```
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.makeSound(); // Output: Bark
        dog.eat(); // Output: This animal eats food.
    }
}
```

Differences Between Abstract Classes and Interfaces

Feature	Abstract Class	Interface
Instantiation	Cannot be instantiated directly	Cannot be instantiated
Constructors	Can have constructors	Cannot have constructors
Instance Variables	Can have instance variables	Cannot have instance variables
Method Types	Can have both abstract and concrete methods	Can only have abstract methods (until Java 8), but can have default and static methods from Java 8
Inheritance	A class can extend only one abstract class	A class can implement multiple interfaces
Access Modifiers	Can use any access modifier	Methods are public by default
Default Implementation	No default implementation (except for concrete methods)	Can have default methods

Summary

- **Abstract Classes** are used when you need to define a base class with some shared functionality and state for related subclasses. They can have instance

variables, constructors, and a mix of abstract and concrete methods.
- **Interfaces** are used to define a contract for behaviors that classes can implement, allowing for flexibility and multiple inheritance. They cannot have instance variables or constructors, but can have default methods starting from Java 8.

Understanding these concepts is crucial for designing robust and maintainable object-oriented software in Java. Choosing between an abstract class and an interface depends on the specific requirements of your application and the relationships between the classes involved.

Inner class :

Example :

```
package code;

class A {
    int age;

    public void show() {
        System.out.println(age);
    }

    class B {

        public void config() {
            System.out.println("B config");
        }
    }
    static class C {

        public void config() {
            System.out.println("B config");
        }
    }
}

class Main {
    public static void main(String[] args) {
        A a = new A();
        A.B b = a.new B();
        b.config();
        A.C  c= new A.C();
        c.config();

    }
}
```

Anonymous inner class :

Example :

```
package code;

class A {
    int age;

    public void show() {
        System.out.println("hello");
    }

    public void show2() {
        System.out.println("hello");
    }


}

class Main {
    public static void main(String[] args) {
        A a = new A() {

            public void show() {
                System.out.println("hi");
            }
        };
        a.show(); // hi
        a.show2(); // hello


    }
}
```

Example 2 :

```
package code;

abstract class A {

    public abstract void show();

}


class Main {
    public static void main(String[] args) {
        A a = new A() {
            public void show() {
                System.out.println("show");
```

```
            }

        };


    }
}
```

dependency injection :

Example ;

```
package code;

interface   Computer{
    void code();
}

class Laptop implements Computer{
    public void code(){
        System.out.println("Laptop coding");
    }
}
class Desktop implements Computer{
    public void code(){
        System.out.println("Desktop coding");
    }
}
class Developer {

    void devApp(Computer device){
        device.code();
    }
}
class Main {

    public static void main(String[] args) {

        Computer computer = new Desktop();
         Developer dev1 = new Developer();
         // dependency injection
         dev1.devApp(computer);

        Developer dev2 = new Developer();
        computer = new Laptop();
        // dependency injection
        dev1.devApp(computer);

    }
}
```

#the inheritance keyword : /* class -> class : extends interface -> class : implements interface -> interface : extends */

enums :

Example :

```
enum Status {
    pending(10), inProgress(20), completed(30);
    // enum constructor :

    private final double price;

    Status(int price) {
        this.price = price;

    }

    public double getPrice() {
        return this.price;
    }
}


class StatusHandler {
    final static String pending = "pending";

    public String getStatus(Status st) {

        return switch (st) {
            case pending -> "pending";
            case inProgress -> "inProgress";
            case completed -> "completed";

        };

    }
}

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

        StatusHandler statusHandler = new StatusHandler();
        String currentStatus = statusHandler.getStatus(Status.pending);

        // get the index of the  inProgress object  :
        System.out.println(Status.inProgress.ordinal());

        // get the price of the inProgress object :
        System.out.println(Status.inProgress.getPrice());
```

```
        // get the string value of the pending object :
        System.out.println(Status.pending); // pending
        System.out.println(Status.valueOf(currentStatus)); // pending

        // enum in java extend enum class
        System.out.println(Status.completed.getClass().getSuperclass()); //  class
java.lang.Enum

        // iterate over all status :
        for (Status stat : Status.values()) {
            System.out.println(  stat.ordinal()+  "-the price of " + stat + " : " +
stat.getPrice());
        }

    }
}
```

Annotations in Java are a form of metadata, which you can use to provide additional information about the code to the Java compiler, tools, and runtime environments. Annotations are widely used in Java to influence the behavior of code, validate data, mark classes and methods, and generate boilerplate code automatically through libraries and frameworks. Here's an overview of the essential aspects of Java annotations:

1. What are Annotations?

Annotations are a special kind of syntactic metadata that can be added to Java classes, methods, fields, parameters, and other elements. They are prefixed by the `@` symbol and don't directly affect the execution of code but provide information for the compiler or runtime.

2. Types of Annotations

- **Standard Annotations**: Built-in annotations provided by Java (e.g., `@Override`, `@Deprecated`, `@SuppressWarnings`).
- **Meta-Annotations**: Annotations that apply to other annotations and control how they behave (e.g., `@Retention`, `@Target`).
- **Custom Annotations**: Annotations you define to suit specific needs within your project.

3. Common Standard Annotations

- `@Override`: Indicates a method overrides a method in a superclass. This helps prevent errors by ensuring the method signature matches.
- `@Deprecated`: Marks a method, class, or field as obsolete, signaling it should not be used and may be removed in the future.
- `@SuppressWarnings`: Suppresses compiler warnings for a specific section of code.
- `@FunctionalInterface`: Ensures an interface has only one abstract method, making it a functional interface.
- `@SafeVarargs`: Suppresses warnings related to varargs usage with generic types.

Example :

```
package code;
// annotation :  a hint for the compiler
@FunctionalInterface
interface I1 {
    void show();
}

class A {

    public void showTheDataWhichBelongsToThisClass() {
        System.out.println("This is the data which belongs to the class A");
    }

    public void show() {
        System.out.println("A");
    }
}

class B extends A {
    // annotation :  a hint for the compiler
    @Override
    public void showTheDataWhichBelongsToThisClass() {
        System.out.println("This is the data which belongs to the class B");
    }

    // annotation :  a hint for the compiler
    @Override
    public void show() {
        System.out.println("B");
    }
}

class Main {

    public static void main(String[] args) {
        B obj = new B();
        obj.showTheDataWhichBelongsToThisClass();
```

```
    }
}
```

lambda function :

```
// SAM : single abstract method
@FunctionalInterface
interface A {
    void show(int x);

}

@FunctionalInterface
interface B {
    int add(int x, int y);
}

class Main {

    public static void main(String[] args) {
        A ob1 = new A() {
            public void show(int x) {
                System.out.println(x + "-Hello World");
            }
        };
        // lambda expression
        A ob2 = (int x) -> {
            System.out.println(x + "-Hello World");
        };

        // lambda expression
        A ob3 = (x) -> System.out.println(x + "-Hello World");

        // lambda expression
        A ob4 = x -> System.out.println(x + "-Hello World");

        // lambda expression
        B o1 = new B() {
            public int add(int x, int y) {
                return x + y;
            }
        };

        // lambda expression
        B o2 = (int x, int y) -> {
            return x + y;
```

```
        };

        // lambda expression
        B o3 = (int x, int y) -> x + y;
        System.out.println(o3.add(5, 4)); // 9


    }
}
```

Types of interfaces :

1. *normal* : it's an interface with more than one abstract function
2. *functional* : an interface with only one abstract method
3. *marker* : blank interface with no method

Types of Errors :

1. compile time error
2. runtime error
3. logical error

1. Definition

- **Exception**: An exception is an event that disrupts the normal flow of a program's instructions due to an abnormal condition. Exceptions are usually conditions that a program can anticipate and potentially handle. For instance, trying to divide by zero, accessing a file that doesn't exist, or parsing invalid input data could all trigger exceptions.
- **Error**: An error typically represents a more severe issue that is not intended to be recovered from within the program. Errors are usually conditions that indicate problems beyond the program’s control or problems in the runtime environment, like running out of memory, stack overflow, or hardware failure.

2. Recoverability

- **Exceptions** are usually recoverable. A program can often catch an exception, handle it (e.g., provide a default value or retry the operation), and continue running. For example, if a file is not found, the program can prompt the user to check the file path and try again.
- **Errors** are usually not recoverable. They indicate serious problems that may require the program to terminate or restart. For example, a memory allocation error (OutOfMemoryError in Java) is typically fatal and difficult to recover from within the program.

3. Types of Exceptions

- **Exceptions** can be of various types, such as:
  - **Checked exceptions** (Java): These are exceptions that must be declared in the method signature and caught or thrown. For example, `IOException` when dealing with file handling.
  - **Unchecked exceptions** (Java): These do not need to be declared or caught, such as `NullPointerException` or `ArithmeticException`.
  - **Runtime exceptions**: Typically indicate programming errors, like trying to access an array index that doesn't exist. Many programming languages treat these as recoverable exceptions.

5. Handling

- **Exceptions** can be handled with `try-catch` blocks (in Java, Python, and other languages) or equivalent structures, allowing developers to provide alternate logic when an error condition is encountered.
- **Errors** are usually not handled with `try-catch` blocks because they represent serious problems. Attempting to handle them might not be effective or recommended. Instead, they may be logged, and the program might terminate or alert the user, depending on the nature of the application.

6. Impact on Program Flow

- **Exceptions** allow for controlled handling and continuation of the program flow. The program may continue after an exception if it’s properly handled.

- **Errors** often cause the program to crash or halt because they represent unrecoverable conditions.

Summary Table

Aspect	Exception	Error
Definition	A condition that disrupts normal flow but can often be handled	A serious issue usually beyond program control
Recoverability	Often recoverable	Rarely recoverable
Examples	`FileNotFoundException`, `NullPointerException`	`OutOfMemoryError`, `StackOverflowError`
Handling Mechanism	Handled with `try-catch` blocks	Generally not handled
Impact on Program	May allow program to continue if handled	Often causes program termination

In summary, **exceptions** are typically for predictable, recoverable issues, whereas **errors** are for severe, usually unrecoverable problems. Knowing this distinction helps developers decide when to attempt recovery and when to allow the program to terminate or restart.

Example :

```
try {
            // Arithmetic Exception :
            int result = 5 / 0;
            System.out.println(result);

            // ArrayIndexOutOfBounds  Exception:
            int[] nums = new int[5];
            System.out.println(nums[5]);

            // NullPointer Exception :
            String str = null;
            System.out.println("The length : " + str.length());

        } catch (ArithmeticException e) {
            System.out.println("handling an Arithmetic Exception " +
e.getMessage());
        } catch (ArrayIndexOutOfBoundsException e) {
```

```
            System.out.println("handling an ArrayIndexOutOfBounds Exception " +
e.getMessage());
        } catch (NullPointerException e) {
            System.out.println("handling a NullPointer Exception" +
e.getMessage());
        } catch (Exception e) {
            // other Exceptions :
            System.out.println("handling  an other Exception " + e.getMessage());

        }
```

Create a custom Exception :

```
  try{
            int result =  5/1;

            // create a custom exception :
             if(result==0 ) throw new ArithmeticException("I dont' want to divide
per 0");
             System.out.println(result);
        }catch (ArithmeticException e){
            System.out.println(e.getMessage());
        }
```

Create a custom Exception :

```
class MyException extends Exception {
    public MyException() {
        //
        super("I dont' want to divide per 0");
    }

}
public class Main {

    public static void main(String[] args) {

        try{
            int result =  5/6;

            // create a custom exception :
             if(result==0 ) throw new MyException();
             System.out.println(result);
        }catch(MyException e){
            System.out.println(e.getMessage());
        }
        catch (ArithmeticException e){
```

```
            System.out.println(e.getMessage());
        }
    }
}
```

throws Keyword (Ducking exceptions):

The `throws` keyword in Java is used in method declarations to specify that a method can throw one or more exceptions. It indicates to the caller that they must handle or propagate these exceptions.

Example :

```
class A {


};

public class Main {


    // move the responsibility of handling the Exception to the caller function
    static public void show() throws  ClassNotFoundException {


            Class.forName("code.Mainf");
            System.out.println("Class A found");
    }

    public static void main(String[] args){

        // handle the ClassNotFound Exception from the show method :
        try {
            show();

        }
        catch (ClassNotFoundException e) {
            System.out.println(e);
        }
        finally {
            System.out.println("continue the execution of the code:");
        }

    }
}
```

Scanner object :

Here's a complete Java file that demonstrates and explains the usage of the `Scanner` class with comments to help you understand each part:

```
import java.util.Scanner;

public class ScannerExample {

    public static void main(String[] args) {
        // Create a Scanner object to read input from the standard input (keyboard)
        Scanner scanner = new Scanner(System.in);

        // Using nextLine() to read a full line of input as a String
        System.out.print("Enter your full name: ");
        String fullName = scanner.nextLine(); // Reads a line of text
        System.out.println("Your full name is: " + fullName);

        // Using next() to read a single word as a String
        System.out.print("Enter a single word: ");
        String singleWord = scanner.next(); // Reads a single word until whitespace
        System.out.println("You entered the word: " + singleWord);

        // Using nextInt() to read an integer
        System.out.print("Enter your age: ");
        int age = scanner.nextInt(); // Reads an integer
        System.out.println("Your age is: " + age);

        // Using nextDouble() to read a double
        System.out.print("Enter your height in meters (e.g., 1.75): ");
        double height = scanner.nextDouble(); // Reads a double value
        System.out.println("Your height is: " + height + " meters");

        // Using nextBoolean() to read a boolean value
        System.out.print("Are you a student? (true/false): ");
        boolean isStudent = scanner.nextBoolean(); // Reads a boolean (true or
false)
        System.out.println("Student status: " + isStudent);

        // Checking for input with hasNext methods before reading
        System.out.println("\nAdditional input checks:");
        System.out.print("Enter an integer (or type 'done' to skip): ");
        if (scanner.hasNextInt()) { // Checks if the next input is an integer
            int number = scanner.nextInt();
            System.out.println("You entered the integer: " + number);
        } else {
            System.out.println("Skipping integer input.");
        }

        // Clear scanner buffer after reading non-String types
        scanner.nextLine(); // Clear the buffer after using nextInt(),
nextDouble(), etc.
```

```
        // Using delimiter to change the default whitespace delimiter
        System.out.print("\nEnter three words separated by commas (e.g.,
word1,word2,word3): ");
        scanner.useDelimiter(","); // Sets a comma as the delimiter
        String word1 = scanner.next();
        String word2 = scanner.next();
        String word3 = scanner.next();
        System.out.println("You entered: " + word1 + ", " + word2 + ", " + word3);

        // Reset delimiter to default (whitespace)
        scanner.useDelimiter("\\s+");

        // Using hasNextLine() in a loop to read multiple lines
        System.out.println("\nEnter lines of text (type 'exit' to finish):");
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            if (line.equalsIgnoreCase("exit")) {
                break; // Exit the loop if "exit" is entered
            }
            System.out.println("You entered: " + line);
        }

        // Close the scanner to free resources
        scanner.close();
    }
}
```

Explanation of Key Parts

- **Creating a Scanner:** `Scanner scanner = new Scanner(System.in);` initializes a scanner to read input from the console.
- **Reading different data types:**
  - `nextLine()`: Reads an entire line.
  - `next()`: Reads a single word.
  - `nextInt()`, `nextDouble()`, `nextBoolean()`: Reads specific data types.
- **Using `hasNext` methods:** `hasNextInt()`, `hasNextDouble()`, etc., are used to check the next input's type before reading it.
- **Changing delimiters:** By default, the scanner uses whitespace as a delimiter, but you can change it with `useDelimiter()`.
- **Clearing the buffer:** After reading non-String types like `nextInt()`, use `nextLine()` to clear the buffer.
- **Closing the scanner:** `scanner.close();` is essential to free resources after input is done.

This example covers most of `Scanner`'s features commonly used in Java.

Why we need to clear the buffer :

We need to clear the buffer in `Scanner` because of how the `nextInt()`, `nextDouble()`, and similar methods work. These methods only consume the specific type of input (like an integer or double) and leave any remaining characters, including a newline character, in the buffer.

Here's a breakdown of why and when we need to clear the buffer:

Example Without Clearing the Buffer

Consider the following code:

```
Scanner scanner = new Scanner(System.in);

System.out.print("Enter an integer: ");
int number = scanner.nextInt(); // Reads an integer but leaves the newline in the
buffer

System.out.print("Enter your name: ");
String name = scanner.nextLine(); // Immediately reads the leftover newline
System.out.println("Your name is: " + name);
```

Explanation

1. **`nextInt()`** reads only the integer, leaving the newline (`\n`) in the buffer.
2. **`nextLine()`** is called immediately after `nextInt()`. Since there's a leftover newline from the previous input, `nextLine()` reads this newline as a blank input, without actually waiting for the user's name.

Clearing the Buffer

To avoid this issue, we add a `scanner.nextLine();` after `nextInt()` to consume the newline left by `nextInt()`:

```
System.out.print("Enter an integer: ");
int number = scanner.nextInt(); // Reads the integer

scanner.nextLine(); // Clears the buffer by consuming the newline
```

```
System.out.print("Enter your name: ");
String name = scanner.nextLine(); // Now reads the actual name
System.out.println("Your name is: " + name);
```

Why Clearing the Buffer is Necessary

Clearing the buffer ensures:

- **Accurate Reading of Input**: `nextLine()` will now correctly wait for a full line of input from the user rather than picking up leftover characters.
- **Prevents Unexpected Behavior**: If the buffer isn't cleared, input methods like `nextLine()` might behave unpredictably, causing confusion in input handling.

In short, clearing the buffer is necessary to ensure smooth, predictable reading of user input when switching between different `Scanner` methods.

using Scanner with try resources :

```
// try with resources {no need to close the resources in the finally block}
   try(Scanner sc=new Scanner(System.in)){
      System.out.println("Enter number");
      num=sc.nextInt();
      System.out.println(num);
   }catch (Exception e){
       System.out.println("something went wrong");
   }

   }
```

Multithreading :

Yes, the JVM can run multiple threads at the same time, but whether it does depends on the **number of available CPU cores** and how the **operating system schedules threads**. Here's a more detailed explanation:

Multithreading and Concurrency in the JVM

1. **Multithreading in the JVM:**
    - The JVM is capable of running multiple threads concurrently, meaning that if your system has multiple CPU cores, the JVM can execute different threads on different cores at the same time. This is true for both Java threads that you create and internal threads like those used for garbage collection.
2. **How Multithreading Works on Multiple Cores:**
    - If your system has **multiple CPU cores** (e.g., a quad-core processor), the operating system's scheduler can assign separate threads to different cores. This allows the JVM to execute multiple threads simultaneously on different cores, so the threads truly run **at the same time**, rather than just switching between them.
    - For example, if you have 4 cores and 4 threads, each thread can run on a separate core simultaneously.
3. **How Multithreading Works on a Single Core:**
    - On a **single-core system**, the JVM can't truly execute multiple threads simultaneously. Instead, the OS uses **time slicing** (also known as **context switching**), where the CPU rapidly switches between threads, giving the illusion of concurrent execution. This is called **concurrent execution**, but not **parallel execution**.
    - Even though threads are not running simultaneously on a single core, the rapid switching can make it appear as though multiple threads are executing at once.
4. **Garbage Collection:**
    - The JVM's garbage collector itself uses multiple threads (especially in modern garbage collection algorithms like **G1GC** or **ParallelGC**), and these threads can run in parallel across multiple cores. The garbage collector operates on a separate thread, allowing background work to happen while the main program continues running.
5. **Thread Synchronization and Blocking:**
    - Even though threads can run concurrently on multiple cores, if threads need to access shared resources (like variables or objects), synchronization mechanisms (such as `synchronized` blocks or locks) may be used to ensure

that only one thread at a time can access a particular resource. This prevents race conditions.
    - Threads can also be blocked (e.g., by calling `Thread.sleep()` or `wait()`), allowing the OS to switch to another thread while the blocked thread is waiting. This doesn't stop other threads from executing.

Key Points:

- On **multi-core systems**, the JVM can run multiple threads **simultaneously** (in parallel) on different cores.
- On a **single-core system**, the JVM will simulate concurrent execution by rapidly switching between threads, but only one thread runs at a time.
- **Garbage collection** and other JVM internal processes can also run in parallel using multiple threads.
- **Thread scheduling** and management (including whether threads run in parallel or are switched) are determined by the **operating system's scheduler**.

Conclusion

So, in summary, the JVM can indeed run multiple threads **at the same time** on multi-core systems (parallel execution). On single-core systems, it will switch between threads very quickly to give the illusion of concurrency, but only one thread runs at any given moment (this is concurrent execution, not parallel).

types of classes :

1. ordinary : a normal class
2. thread class : a class that inherit from the thread class

Example Thread class:

```
package code;

class A extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("hello");
        }
    }
}
```

```
class B extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("hi");
        }
    }
}


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

        System.out.println("1");
        A a = new A();
        B b = new B();
        System.out.println("2");
        a.start();
        Thread.sleep(2000);
        System.out.println("3");
        b.start();
        System.out.println("4");
    }

}
```

Example Runnable interface :

```
package code;

class A implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("hello");
        }
    }
}

class B implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("hi");
        }
    }
}


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

        System.out.println("1");
        A a = new A();
        B b = new B();
```

```
        System.out.println("2");
        Thread t1 = new Thread(a);
        Thread t2 = new Thread(b);
        t1.start();
        Thread.sleep(2000);
        System.out.println("3");
        t2.start();
        System.out.println("4");
    }

}
```

Example 3 Functional interface :

```
package code;

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

        System.out.println("1");
        System.out.println("2");
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                System.out.println("hello");
            }
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                System.out.println("hi");
            }
        });
        t1.start();
        Thread.sleep(2000);
        System.out.println("3");
        t2.start();
        System.out.println("4");
    }

}
```

Final Code

```
package code;

class Counter {
    public static int counter = 0;

    // A synchronized function to make it thread-safe
    public synchronized static void Increment() {
```

```
            counter++;
        }
    }

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

            Thread t1 = new Thread(() -> {
                for (int i = 0; i < 1000; i++) {
                    Counter.Increment();
                }
            });
            Thread t2 = new Thread(() -> {
                for (int i = 0; i < 1000; i++) {
                    Counter.Increment();
                }
            });

            // Start both threads
            t1.start();
            t2.start();

            // Wait for threads to complete
            t1.join();
            t2.join();

            // Print final counter value
            System.out.println(Counter.counter); // Always 2000
        }
    }
```

the goal is to increment a shared counter variable from multiple threads (`t1` and `t2`) and ensure that the final value is correct (2000). Let's break down the two concepts in question:

Why Use `synchronized`?

The `synchronized` keyword ensures that only one thread can execute the `Increment` method at a time, making it **thread-safe**. Without `synchronized`, multiple threads could modify the `counter` variable simultaneously, leading to a race condition where the final value is unpredictable.

Problem Without `synchronized`:

1. Both threads (`t1` and `t2`) might read the same value of `counter` (e.g., `1000`) at the same time.

2. Each increments it separately and writes back the same value (`1001`), effectively losing one increment.

Solution with `synchronized`:

By synchronizing the `Increment` method, the JVM ensures:

- Only one thread can execute `Increment` at any given moment.
- Any updates to the `counter` variable are visible to all threads immediately.

Example with Race Condition (No `synchronized`):

Output might be:

```
1994
1997
1988
```

With `synchronized`, the output is always:

```
2000
```

Why Use `join()`?

The `join()` method ensures that the main thread waits for `t1` and `t2` to finish execution before continuing. Without `join()`, the main thread might execute the `System.out.println(Counter.counter)` statement before `t1` and `t2` complete their tasks.

Problem Without `join()`:

- `System.out.println(Counter.counter)` might execute before `t1` and `t2` finish, printing an incomplete result (e.g., `1500` or `1800`).

Solution with `join()`:

- `t1.join()` and `t2.join()` make the main thread wait until both threads finish execution, ensuring that `Counter.counter` is fully updated before printing.

How the Code Runs Step-by-Step

1. **Main Thread Starts:**
    - Two threads (`t1` and `t2`) are created.
    - Both threads increment the `Counter.counter` variable 1000 times.
2. **Threads Execute:**
    - The JVM assigns CPU time to both `t1` and `t2`.
    - `synchronized` ensures that only one thread at a time can execute the `Increment` method.
3. **Main Thread Waits (`join()`):**
    - The main thread waits for `t1` and `t2` to finish using `join()`.
4. **Output:**
    - After both threads finish, the final value of `Counter.counter` (2000) is printed.

Output

```
2000
```

Summary

- **`synchronized`** ensures that `Increment()` is thread-safe, avoiding race conditions.
- **`join()`** ensures the main thread waits for `t1` and `t2` to complete, guaranteeing accurate output. The syntax `synchronized (lock) { ... }` in Java is **not a**

function. It is a **synchronization block**, a built-in Java construct used for thread synchronization.

What is a Synchronized Block?

- A **synchronized block** ensures that only one thread at a time can execute the code inside the block for the specified object (`lock` in this case).
- The `lock` is an object that serves as a **monitor** (or mutex). The thread must acquire the monitor (lock) before entering the block and releases it when leaving the block.

Syntax Breakdown

```
synchronized (lock) {
    // Critical section
    // Code inside this block can only be executed by one thread at a time.
}
```

- `synchronized`: A Java keyword indicating that the block of code is synchronized.
- `(lock)`: The monitor object on which synchronization is applied.
  - The thread must acquire the monitor for this object before executing the block.
  - Other threads trying to synchronize on the same object (`lock`) must wait until the monitor is released.
- `{ ... }`: The block of code (critical section) that is protected by the synchronization mechanism.

Is It a Function?

No, `synchronized (lock) { ... }` is not a function. It is:

1. A **language construct** for synchronizing threads.
2. A way to protect shared resources and prevent race conditions.

How It Works

1. A thread entering the `synchronized` block must:
    - Acquire the lock on the `lock` object.
    - If the lock is already held by another thread, the thread is blocked until the lock is available.
2. After acquiring the lock:
    - The thread executes the code inside the block.
3. When the thread exits the block:
    - The lock is released, allowing other threads waiting for the lock to proceed.

Example: Synchronized Block in Action

Here is an example showing how a synchronized block is used to safely increment a counter in a multithreaded environment:

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

    public void increment() {
        synchronized (this) { // Synchronize on the current instance
            count++;
        }
    }

    public int getCount() {
        return count;
    }
}

class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        Thread t2 = new Thread(() -> {
```

```
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final Count: " + counter.getCount());
    }
}
```

Why Use a Synchronized Block Instead of a Function?

You can synchronize at the block level instead of the whole method for finer-grained control. For example:

- **Synchronized Block**: Only critical parts of the code are synchronized, allowing other threads to execute non-critical sections simultaneously.

  ```
  public void increment() {
      synchronized (this) {
          count++;
      }
  }
  ```

- **Synchronized Method**: The entire method is synchronized, which might be less efficient.

  ```
  public synchronized void increment() {
      count++;
  }
  ```

Summary

- `synchronized (lock) { ... }` is **not a function**; it is a language construct for thread synchronization.
- It allows a thread to acquire a lock on an object (`lock`) and execute the block exclusively.
- It is used to prevent concurrent access to shared resources, avoiding issues like race conditions.

Yes, your understanding is correct. Here's how the `sleep` function interacts with thread execution and locks:

Key Behaviors of `Thread.sleep`

1. `Thread.sleep` **pauses the current thread**:
    - When a thread calls `Thread.sleep(milliseconds)`, it goes into a **TIMED_WAITING** state for the specified duration.
    - During this time, the thread does not consume CPU but is still considered alive.
2. **The thread does not release locks**:
    - If the thread that called `sleep` is holding a lock (e.g., it is inside a `synchronized` block), it will **continue to hold the lock** during the sleep period.
    - Other threads waiting for that lock cannot proceed until the sleeping thread wakes up and exits the synchronized block.
3. **After sleeping, the thread resumes execution**:
    - Once the sleep duration ends, the thread becomes runnable again, re-enters the running state when scheduled, and continues executing from where it left off.

Does Sleep Allow Other Threads to Enter the Critical Section?

No, the `sleep` function does not allow other threads to enter the critical section **if the sleeping thread is holding the lock**.

- While the sleeping thread is inside a `synchronized` block, the lock is not released. This means other threads waiting for the lock will remain blocked until the sleeping thread wakes up, finishes its work in the critical section, and releases the lock.

Example: Sleep Inside a Synchronized Block

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

    public void increment() {
        synchronized (this) { // Critical section
            System.out.println(Thread.currentThread().getName() + " has entered the
critical section.");
            try {
                Thread.sleep(100); // Thread sleeps but keeps the lock
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            count++;
            System.out.println(Thread.currentThread().getName() + " is leaving the
critical section.");
        }
    }

    public int getCount() {
        return count;
    }
}

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

        Thread t1 = new Thread(() -> counter.increment(), "Thread 1");
        Thread t2 = new Thread(() -> counter.increment(), "Thread 2");

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

Output Explanation

1. When `Thread 1` enters the critical section, it acquires the lock.
2. It then calls `Thread.sleep(100)`. During this time:
    - `Thread 1` is paused (TIMED_WAITING state).
    - **The lock is still held by `Thread 1`**, so `Thread 2` cannot enter the critical section and must wait.
3. After 100 ms, `Thread 1` wakes up, finishes its critical section, and releases the lock.
4. `Thread 2` then acquires the lock and proceeds to execute its critical section.

Output Example:

```
Thread 1 has entered the critical section.
Thread 1 is leaving the critical section.
Thread 2 has entered the critical section.
Thread 2 is leaving the critical section.
```

Key Difference Between `sleep` and `wait`

1. `Thread.sleep`:
    - Pauses the thread for a specified time.
    - The thread **does not release any locks it holds**.
    - Other threads cannot enter synchronized blocks protected by the same lock.
2. `Object.wait`:
    - Causes the thread to release the lock and enter the waiting state.
    - Other threads can now acquire the lock and enter the critical section.
    - The waiting thread can only proceed if another thread calls `notify` or `notifyAll` on the same lock object.

Example of `wait` Behavior

```
class SharedResource {
    public void criticalSection() {
        synchronized (this) {
            try {
                System.out.println(Thread.currentThread().getName() + " is
waiting.");
                wait(); // Releases lock and waits to be notified
                System.out.println(Thread.currentThread().getName() + " is running
again.");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public void notifyThread() {
        synchronized (this) {
            System.out.println("Notifying threads...");
            notify(); // Wakes up one waiting thread
        }
    }
}

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

        Thread t1 = new Thread(resource::criticalSection, "Thread 1");
        Thread t2 = new Thread(resource::notifyThread, "Thread 2");

        t1.start();
        try {
            Thread.sleep(100); // Ensure t1 starts and waits before t2 notifies
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        t2.start();
    }
}
```

Output Example:

```
Thread 1 is waiting.
Notifying threads...
Thread 1 is running again.
```

Summary

- `Thread.sleep` pauses the thread but does not release the lock, preventing other threads from entering the critical section.
- `Object.wait` pauses the thread **and releases the lock**, allowing other threads to enter the critical section. The thread resumes only after receiving a `notify` or `notifyAll` call.

Skip the critical section?:

In Java, if a thread encounters a **synchronized block** (critical section) and finds that the lock is held by another thread, the thread **does not skip the critical section or execute the code below it**. Instead, the thread will **wait (block)** until the lock is released. Once the lock is available, the thread will acquire it and then execute the critical section before proceeding with the code below it.

How it Works

1. When a thread reaches a `synchronized(lock)` block:
    - If the lock is available, the thread acquires it and proceeds to execute the critical section.
    - If the lock is held by another thread, the thread is placed in a **BLOCKED state**. It will wait until the lock becomes available.
2. Once the thread acquires the lock:
    - It executes the critical section inside the `synchronized(lock)` block.
    - After completing the critical section, it releases the lock and continues executing any code below the critical section.
3. The thread cannot skip the critical section, as it must execute it once the lock is acquired. Skipping would break synchronization guarantees and potentially lead to inconsistent data.

Example

Here’s an example to demonstrate this behavior:

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

    public void increment() {
        synchronized (this) { // Critical section
            System.out.println(Thread.currentThread().getName() + " is inside the
critical section.");
            count++;
            try {
                Thread.sleep(100); // Simulate some work in the critical section
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + " is leaving the
critical section.");
        }
        System.out.println(Thread.currentThread().getName() + " is executing code
below the critical section.");
    }

    public int getCount() {
        return count;
    }
}

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

        Thread t1 = new Thread(() -> counter.increment(), "Thread 1");
        Thread t2 = new Thread(() -> counter.increment(), "Thread 2");

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

Output Explanation

- When `Thread 1` enters the critical section, `Thread 2` must wait until the lock is released.
- While `Thread 2` is waiting, it cannot execute the code below the critical section. It must first execute the synchronized block before continuing.
- The output will show the threads taking turns in the critical section, followed by the code outside the critical section.

Example Output:

```
Thread 1 is inside the critical section.
Thread 1 is leaving the critical section.
Thread 1 is executing code below the critical section.
Thread 2 is inside the critical section.
Thread 2 is leaving the critical section.
Thread 2 is executing code below the critical section.
```

Key Points

- A thread **must acquire the lock** before entering a synchronized block.
- While waiting for the lock, the thread is in a BLOCKED state, not executing any part of the critical section or the code below it.
- Once the lock is released, the thread resumes execution from the start of the synchronized block and proceeds to execute the code below after finishing the critical section.

This ensures proper synchronization and prevents issues like race conditions.

Thread States in Java

Java threads go through several states during their lifecycle. These states are managed by the JVM and the thread scheduler. Below are the key thread states, their meaning, and how they are transitioned:

---

1. NEW

- **Description**:
  - When a thread is created but hasn't started yet.
  - The thread exists as an object in memory but isn't active.
- **How to Enter**:
  - A thread enters the `NEW` state when an instance of `Thread` or its subclass is created using the `new` keyword.
- **Example**:

```
Thread t = new Thread(() -> System.out.println("Thread is running"));
// t is in NEW state.
```

---

2. RUNNABLE

- **Description**:
  - The thread is ready to run but is waiting for CPU time.
  - This state means the thread has been scheduled by the JVM but is not guaranteed to be running immediately.
- **How to Enter**:
  - A thread moves to the `RUNNABLE` state when `start()` is called.
- **Example**:

```
t.start(); // Now t is in the RUNNABLE state.
```

---

3. RUNNING

- **Description**:
  - The thread is actively executing its `run()` method.
  - In Java, a thread is in the `RUNNING` state whenever it gets CPU time while being in the `RUNNABLE` state.
- **How to Enter**:
  - The thread scheduler selects a thread in the `RUNNABLE` state and allocates CPU time to it.
  - Note: You cannot directly manipulate or observe the `RUNNING` state—it is an internal JVM state.

---

4. WAITING

- **Description**:
  - The thread is waiting indefinitely for another thread to perform a specific action (e.g., notify it).

- The thread does not consume CPU cycles while in this state.

- **How to Enter**:
  - A thread enters the `WAITING` state when it calls:
    - `wait()` (inside a synchronized block).
    - `join()` on another thread.
    - `LockSupport.park()`.

- **Example**:

```
synchronized (lock) {
    lock.wait(); // Thread is in WAITING state.
}
```

---

5. TIMED_WAITING

- **Description**:
  - Similar to `WAITING`, but the thread will wait for a fixed amount of time before proceeding.
- **How to Enter**:
  - A thread enters the `TIMED_WAITING` state when it calls:
    - `sleep(milliseconds)`.
    - `wait(milliseconds)` (inside a synchronized block).
    - `join(milliseconds)` on another thread.
    - `LockSupport.parkNanos()` or `LockSupport.parkUntil()`.
- **Example**:

```
Thread.sleep(100); // Thread is in TIMED_WAITING state.
```

---

6. BLOCKED

- **Description**:
  - The thread is waiting for a lock to enter a synchronized block or method.
  - It happens when another thread is already holding the required lock.
- **How to Enter**:

- A thread enters the `BLOCKED` state if it tries to access a synchronized block/method and the lock is already held by another thread.

- **Example**:

```
synchronized (lock) {
    // Another thread will be in the BLOCKED state if it tries to access this block.
}
```

---

7. TERMINATED (DEAD)

- **Description**:
  - The thread has finished executing its `run()` method.
  - The thread is no longer active and cannot be restarted.
- **How to Enter**:
  - A thread enters the `TERMINATED` state after completing its execution or being stopped abruptly.
- **Example**:

```
t.join(); // After this, t will be in TERMINATED state if its execution is complete.
```

Thread State Transitions

Here's a summary of common transitions between states:

1. **NEW → RUNNABLE**:
   - Calling `start()` on a thread object.
2. **RUNNABLE → RUNNING**:
   - The thread scheduler allocates CPU time to the thread.
3. **RUNNING → WAITING / TIMED_WAITING**:
   - The thread calls `wait()`, `sleep()`, or similar methods.

4. **RUNNING → BLOCKED**:
    - The thread tries to acquire a lock held by another thread.
5. **WAITING / TIMED_WAITING → RUNNABLE**:
    - The thread is notified or the wait/sleep time expires.
6. **RUNNING → TERMINATED**:
    - The thread completes its task or is explicitly stopped.

Code Example Demonstrating Thread States

```
public class ThreadStatesExample {

    public static void main(String[] args) {
        Object lock = new Object();

        Thread thread1 = new Thread(() -> {
            synchronized (lock) {
                System.out.println(Thread.currentThread().getName() + " is
RUNNING.");
                try {
                    System.out.println(Thread.currentThread().getName() + " is
WAITING.");
                    lock.wait(); // Moves to WAITING state.
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName() + " is RUNNING
again.");
            }
        }, "Thread 1");

        Thread thread2 = new Thread(() -> {
            try {
                Thread.sleep(100); // Moves to TIMED_WAITING state.
                synchronized (lock) {
                    System.out.println(Thread.currentThread().getName() + " is
NOTIFYING.");
                    lock.notify(); // Wakes up thread1.
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "Thread 2");

        thread1.start(); // Moves to RUNNABLE, then RUNNING.
        thread2.start(); // Moves to RUNNABLE, then TIMED_WAITING.
```

```
    }
}
```

Output

```
Thread 1 is RUNNING.
Thread 1 is WAITING.
Thread 2 is NOTIFYING.
Thread 1 is RUNNING again.
```

Summary

- Threads progress through states (`NEW`, `RUNNABLE`, `RUNNING`, etc.) based on their actions and interactions with other threads.
- Understanding these states helps to write efficient, synchronized, and thread-safe programs.

No, the `notify()` method does **not** inform all threads. It wakes up **only one thread** that is waiting on the same lock object. If multiple threads are waiting on the lock, the thread scheduler will choose **one thread** arbitrarily (or based on thread priority) to wake up and move it from the `WAITING` state to the `BLOCKED` state. The thread will then attempt to acquire the lock.

If You Want to Inform All Waiting Threads

You can use the `notifyAll()` method instead. This will wake up **all threads** waiting on the same lock object, moving them all from the `WAITING` state to the `BLOCKED` state. These threads will then compete for the lock, and only one of them can acquire it at a time.

Difference Between `notify()` and `notifyAll()`

Method	Behavior
`notify()`	Wakes up **only one thread** waiting on the lock. The rest remain in the `WAITING` state.
`notifyAll()`	Wakes up **all threads** waiting on the lock. They all compete for the lock.

Code Example

```
public class NotifyExample {
    public static void main(String[] args) {
        Object lock = new Object();

        Runnable waitingTask = () -> {
            synchronized (lock) {
                try {
                    System.out.println(Thread.currentThread().getName() + " is
WAITING.");
                    lock.wait(); // Moves to WAITING state.
                    System.out.println(Thread.currentThread().getName() + " is
RUNNING again.");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        Thread thread1 = new Thread(waitingTask, "Thread 1");
        Thread thread2 = new Thread(waitingTask, "Thread 2");
        Thread thread3 = new Thread(waitingTask, "Thread 3");

        thread1.start();
        thread2.start();
        thread3.start();

        new Thread(() -> {
            try {
                Thread.sleep(1000); // Allow other threads to enter WAITING state.
                synchronized (lock) {
                    System.out.println("Notifier Thread is calling notify().");
                    lock.notify(); // Wakes up one waiting thread.
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

Output Example

```
Thread 1 is WAITING.
Thread 2 is WAITING.
Thread 3 is WAITING.
Notifier Thread is calling notify().
Thread 1 is RUNNING again.  // Only one thread wakes up.
```

To Wake All Threads

Replace `lock.notify();` with `lock.notifyAll();`:

```
lock.notifyAll();
```

Output Example With `notifyAll()`

```
Thread 1 is WAITING.
Thread 2 is WAITING.
Thread 3 is WAITING.
Notifier Thread is calling notifyAll().
Thread 1 is RUNNING again.
Thread 2 is RUNNING again.
Thread 3 is RUNNING again.
```

Key Points

- `notify()`: Wakes up **one waiting thread**.
- `notifyAll()`: Wakes up **all waiting threads**, and they will compete for the lock.
- The thread that gets the lock first depends on the thread scheduler.

Real Example :

Threads in Java are widely used in real-world applications where concurrency is needed. Here's an example of using threads in a **real-world scenario**: a **web server simulation** that handles multiple client requests concurrently.

Scenario: Simulating a Web Server

In this example, each thread represents a client request to a server. The server processes multiple requests in parallel using threads.

Code Example: Web Server Simulation

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

// Simulating a server handling multiple client requests
class ClientRequest implements Runnable {
    private final int clientId;

    public ClientRequest(int clientId) {
        this.clientId = clientId;
    }

    @Override
    public void run() {
        System.out.println("Processing request from Client " + clientId + " by " +
Thread.currentThread().getName());
        try {
            // Simulate processing time
            Thread.sleep(2000); // 2 seconds
        } catch (InterruptedException e) {
            System.err.println("Thread interrupted for Client " + clientId);
        }
        System.out.println("Completed request from Client " + clientId + " by " +
Thread.currentThread().getName());
    }
}

public class WebServer {
    public static void main(String[] args) {
        // Simulating a thread pool for handling client requests
        ExecutorService threadPool = Executors.newFixedThreadPool(3); // 3 threads
in the pool

        // Simulating 10 client requests
        for (int i = 1; i <= 10; i++) {
            ClientRequest request = new ClientRequest(i);
            threadPool.execute(request); // Submit request to the thread pool
        }
```

```
        // Shutdown the thread pool after all tasks are submitted
        threadPool.shutdown();
        System.out.println("Server has received all requests. Processing...");
    }
}
```

Explanation:

1. **ClientRequest Class**:
   Represents a task for the server. Each task is a client request that takes 2 seconds to process.

2. **Thread Pool**:
   - We use a fixed thread pool with three threads to simulate the server's ability to handle multiple requests concurrently.
   - The `Executors.newFixedThreadPool(3)` creates a thread pool with three worker threads. This means the server can process up to three requests simultaneously.

3. **Simulating Requests**:
   - We create 10 client requests using a `for` loop and submit each request to the thread pool for processing.

4. **Output Behavior**:
   - Initially, the first three requests are processed by the three threads in the pool.
   - Once a thread completes a task, it picks up the next request from the queue.

Sample Output:

```
Server has received all requests. Processing...
Processing request from Client 1 by pool-1-thread-1
Processing request from Client 2 by pool-1-thread-2
Processing request from Client 3 by pool-1-thread-3
Completed request from Client 1 by pool-1-thread-1
Processing request from Client 4 by pool-1-thread-1
```

```
Completed request from Client 2 by pool-1-thread-2
Processing request from Client 5 by pool-1-thread-2
Completed request from Client 3 by pool-1-thread-3
Processing request from Client 6 by pool-1-thread-3
...
```

Real-Life Use Cases of Threads in Java:

1. **Web Servers**: Handling multiple client connections.
2. **Database Systems**: Executing parallel queries or background indexing.
3. **Gaming Engines**: Updating game state, physics, and rendering graphics simultaneously.
4. **IoT Systems**: Handling multiple sensor inputs concurrently.
5. **File Servers**: Reading and writing files from/to multiple users.

Threads are essential when tasks can execute independently or benefit from parallel execution.

Collections in Java

The **Java Collections Framework** is a key part of the Java programming language, providing a set of interfaces, classes, and algorithms for storing and manipulating groups of objects efficiently. It is part of the `java.util` package and supports various operations such as searching, sorting, and iterating over data.

Core Concepts

1. **Collection Interface**: Represents a group of objects known as elements. Subinterfaces include:
    - **List**: Ordered collection allowing duplicate elements (e.g., `ArrayList`, `LinkedList`).
    - **Set**: Collection with no duplicate elements (e.g., `HashSet`, `TreeSet`).
    - **Queue**: Designed for holding elements before processing (e.g., `PriorityQueue`, `LinkedList`).

    - **Deque**: A double-ended queue allowing insertion and removal at both ends (e.g., `ArrayDeque`).

2. **Map Interface**: Represents key-value pairs. Examples include:
    - **HashMap**: Allows null keys and values, unordered.
    - **TreeMap**: Sorted based on natural ordering or a custom comparator.
    - **LinkedHashMap**: Maintains insertion order.

Commonly Used Classes

- **ArrayList**: A resizable array. Offers fast access but slower insert/delete operations compared to linked structures.
- **LinkedList**: A doubly linked list, ideal for frequent insertions and deletions.
- **HashSet**: Implements the `Set` interface using a hash table, ensuring no duplicate entries.
- **TreeSet**: Implements the `Set` interface with a red-black tree, maintaining sorted order.
- **HashMap**: Offers fast lookups by using a hash table to store key-value pairs.
- **PriorityQueue**: Implements a priority heap, where elements are ordered based on their priority.

Key Features

1. **Generics**: Collections support generics to ensure type safety at compile time (e.g., `List<String>`).
2. **Iterators**: Enable traversal over collections using methods like `hasNext()` and `next()`.
3. **Algorithms**: The framework provides algorithms such as sorting (`Collections.sort()`), searching (`Collections.binarySearch()`), and synchronization (`Collections.synchronizedList()`).
4. **Streams**: Modern Java versions (Java 8+) integrate the `Stream` API for functional-style operations like filtering and mapping.

Example

```java
import java.util.*;

public class CollectionExample {
    public static void main(String[] args) {
        // List example
        List<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");
        System.out.println("List: " + list);

        // Set example
        Set<String> set = new HashSet<>(list);
        System.out.println("Set: " + set);

        // Map example
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "One");
        map.put(2, "Two");
        System.out.println("Map: " + map);
    }
}
```

Advanced Topics

- **Concurrent Collections**: Specialized classes like `ConcurrentHashMap` and `CopyOnWriteArrayList` for thread-safe operations.
- **Custom Comparators**: Implement `Comparator` or `Comparable` for custom sorting.
- **Immutable Collections**: Methods like `List.of()` in Java 9+ for creating unmodifiable collections.

The Java Collections Framework is essential for efficient data handling and manipulation in Java applications, making it a cornerstone of modern Java development. For detailed examples and guides, visit Oracle's official documentation.

In Java, `Comparator` and `Comparable` are interfaces used to compare objects. They serve similar purposes but are used in different scenarios and implemented differently.

---

1. Comparable Interface

- **Purpose**: Defines the natural ordering of objects.
- **Implementation**: Implemented by the class whose objects need to be compared.
- **Single sorting logic**: You can define only one way to compare objects.
- **Method to Override**: `compareTo(Object o)`

Syntax Example:

```
public class Student implements Comparable<Student> {
    private int id;
    private String name;

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

    @Override
    public int compareTo(Student other) {
        return this.id - other.id; // Sort by id (ascending order)
    }

    // Getters and toString() for demonstration
}
```

Usage:

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student(2, "Alice"));
        students.add(new Student(1, "Bob"));

        Collections.sort(students); // Uses compareTo for sorting
        System.out.println(students);
    }
}
```

---

2. Comparator Interface

- **Purpose**: Defines custom comparison logic that can be used in addition to the natural ordering.

- **Implementation**: Implemented by a separate class or as a lambda expression.
- **Multiple sorting logic**: You can define multiple ways to compare objects.
- **Method to Override**: `compare(Object o1, Object o2)`

Syntax Example:

```
import java.util.*;

class NameComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return s1.getName().compareTo(s2.getName()); // Sort by name (alphabetical)
    }
}
```

Usage:

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student(2, "Alice"));
        students.add(new Student(1, "Bob"));

        // Sort by name using NameComparator
        Collections.sort(students, new NameComparator());
        System.out.println(students);

        // Sort by id using lambda expression
        Collections.sort(students, (s1, s2) -> s1.getId() - s2.getId());
        System.out.println(students);
    }
}
```

Key Differences Between `Comparable` and `Comparator`

Feature	Comparable	Comparator
Package	`java.lang`	`java.util`
Purpose	Defines natural ordering	Defines custom ordering

Feature	Comparable	Comparator
Method	`compareTo(Object o)`	`compare(Object o1, Object o2)`
Implementation	Implemented by the class	Implemented by a separate class
Number of Logics	Single sorting logic per class	Multiple sorting logics

Use `Comparable` when objects have a natural order (e.g., numbers, alphabetical), and use `Comparator` when you need flexibility to sort in different ways. Streams in Java are a core feature introduced in **Java 8** as part of the Java Stream API, enabling functional-style programming to process collections of data (like arrays or lists) in a declarative way.

Key Characteristics of Streams

1. **Declarative**: Use method chaining to describe what needs to be done, not how.
2. **Lazy Evaluation**: Operations on streams are not performed until a terminal operation is invoked.
3. **Parallelizable**: Streams can be executed in parallel for performance improvements.
4. **Non-Mutable**: Streams do not modify the original data source.

Key Stream Components

1. Stream Source

A stream is created from a data source such as a collection, array, or I/O channel.

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
Stream<Integer> stream = numbers.stream();
```

2. Intermediate Operations

These operations transform or filter the stream and are lazy. Examples include:

- `map()`: Transforms elements.
- `filter()`: Filters elements based on a condition.
- `sorted()`: Sorts the stream.

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
Stream<Integer> evenNumbers = numbers.stream()
    .filter(n -> n % 2 == 0) // Filter even numbers
    .map(n -> n * n);        // Square each number
```

3. Terminal Operations

These operations produce a result or a side effect and consume the stream. Examples include:

- `collect()`: Gathers the stream into a collection.
- `forEach()`: Iterates through the stream.
- `reduce()`: Aggregates elements.

```
List<Integer> evenSquares = evenNumbers.collect(Collectors.toList());
```

4. Short-Circuiting Operations

These stop processing when a condition is met:

- `anyMatch()`
- `allMatch()`
- `noneMatch()`
- `findFirst()`
- `findAny()`

Stream API Methods

Creating Streams

```
Stream<String> streamOfStrings = Stream.of("a", "b", "c");
IntStream intStream = IntStream.range(1, 10); // 1 to 9
Stream<String> streamFromCollection = List.of("x", "y", "z").stream();
```

Intermediate Operations

- `filter()`: Filters elements based on a predicate.

```
Stream<Integer> evenNumbers = numbers.stream().filter(n -> n % 2 == 0);
```

- `map()`: Transforms elements.

```
Stream<String> uppercased = names.stream().map(String::toUpperCase);
```

- `sorted()`: Sorts elements.

```
Stream<Integer> sortedNumbers = numbers.stream().sorted();
```

- `distinct()`: Removes duplicates.

```
Stream<Integer> uniqueNumbers = numbers.stream().distinct();
```

Terminal Operations

- `collect()`: Collects elements into a collection.

```
List<Integer> result = numbers.stream().collect(Collectors.toList());
```

- `reduce()`: Combines elements into a single result.

```
int sum = numbers.stream().reduce(0, Integer::sum);
```

- `forEach()`: Iterates through the stream.

```
numbers.stream().forEach(System.out::println);
```

- `count()`: Counts elements in the stream.

```
long count = numbers.stream().count();
```

Parallel Streams

For large datasets, streams can be parallelized for better performance.

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
numbers.parallelStream().forEach(System.out::println);
```

Primitive Streams

Java provides specialized streams for primitive types:

- `IntStream`: For `int` values.
- `LongStream`: For `long` values.
- `DoubleStream`: For `double` values.

```
IntStream.range(1, 10).forEach(System.out::println);
```

Stream Collectors

The `Collectors` utility class provides methods to gather data from a stream:

- **To List**:

```
List<Integer> list = stream.collect(Collectors.toList());
```

- **To Set**:

```
Set<Integer> set = stream.collect(Collectors.toSet());
```

- **To Map**:

```
Map<Integer, String> map = stream.collect(Collectors.toMap(k -> k, v ->
"Value" + v));
```

- **Joining Strings**:

```
String result = stream.collect(Collectors.joining(", "));
```

Examples

Example 1: Filtering and Mapping

```
List<String> names = List.of("John", "Jane", "Doe");
List<String> uppercased = names.stream()
    .filter(name -> name.startsWith("J"))
    .map(String::toUpperCase)
    .collect(Collectors.toList());

System.out.println(uppercased); // Output: [JOHN, JANE]
```

Example 2: Reduce

```
int sum = List.of(1, 2, 3, 4, 5).stream()
    .reduce(0, Integer::sum);

System.out.println(sum); // Output: 15
```

Example 3: Grouping by

```
Map<Integer, List<String>> groupedByLength = List.of("one", "two", "three", "four")
    .stream()
    .collect(Collectors.groupingBy(String::length));
```

```
System.out.println(groupedByLength);
// Output: {3=[one, two], 4=[four], 5=[three]}
```

Advantages of Streams

1. Cleaner and more readable code.
2. Simplified parallel execution.
3. Improved performance for large data sets when used with parallel streams.

Limitations

1. Cannot reuse streams after a terminal operation.
2. Debugging can be challenging due to the declarative nature.
3. Not always the best option for small data sets due to overhead.

Sealed Classes in Java

Sealed classes, introduced in **Java 15 (as a preview)** and finalized in **Java 17**, allow developers to control which classes can extend or implement a given class or interface. They are part of the effort to make Java more expressive and enable stronger guarantees about the design of APIs.

Features of Sealed Classes

1. **Restricts Extensibility**: A sealed class explicitly lists the classes or interfaces that are allowed to extend it.
2. **Improves Exhaustiveness in Pattern Matching**: It enables the compiler to enforce exhaustive checks for pattern matching, ensuring all possible subclasses are handled.
3. **Enhanced API Design**: Provides more control over the hierarchy of classes.

How to Declare Sealed Classes

A sealed class uses the `sealed` keyword and must explicitly list its permitted subclasses with the `permits` clause.

```
public sealed class Shape permits Circle, Rectangle, Triangle {
    // common fields and methods
}

public final class Circle extends Shape {
    // specific implementation for Circle
}

public final class Rectangle extends Shape {
    // specific implementation for Rectangle
}

public final class Triangle extends Shape {
    // specific implementation for Triangle
}
```

Key Modifiers in a Sealed Class Hierarchy

1. `sealed`: Specifies that the class or interface restricts its subclassing.
2. `non-sealed`: Allows unrestricted subclassing beyond the permitted ones.
3. `final`: Prevents further subclassing entirely.

Example: Sealed and Non-Sealed Classes

```
public sealed class Vehicle permits Car, Truck { }

public final class Car extends Vehicle { }

public non-sealed class Truck extends Vehicle {
    // Truck can have unrestricted subclasses
}

public class Pickup extends Truck {
    // Allowed because Truck is non-sealed
}
```

Rules for Sealed Classes

1. Permitted subclasses must be in the same module (or package, if no module).

2. Subclasses must be `final`, `sealed`, or `non-sealed`.
3. If a subclass is `sealed`, it must list its own permitted subclasses.

Benefits of Sealed Classes

1. **Controlled Extensibility**: API designers can control the class hierarchy.
2. **Exhaustive Pattern Matching**: Useful with `switch` expressions.
3. **Readability and Maintenance**: Enforces strict designs.

`var` in Java

The `var` keyword, introduced in **Java 10**, allows you to declare local variables with inferred types. The compiler determines the type of the variable at compile time.

Features of `var`

1. **Type Inference**: The variable type is inferred from the assigned value.
2. **Limited to Local Variables**: Only usable for local variables, including indices in loops.
3. **Readable Code**: Reduces boilerplate while maintaining type safety.

Examples of Using `var`

1. **Basic Declaration**:

```
var message = "Hello, World!";  // Inferred as String
var number = 42;                // Inferred as int
```

2. **With Collections**:

```
var list = List.of("Apple", "Banana", "Cherry");  // Inferred as List<String>
```

3. **Loops**:

```
for (var item : list) {
    System.out.println(item);
}
```

4. **Map Entries**:

```
var map = Map.of(1, "One", 2, "Two");
for (var entry : map.entrySet()) {
    System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```

Limitations of `var`

1. **Readability**: Overuse can reduce code clarity.

```
var something = getSomething(); // What is something?
```

2. **Cannot Be Used for Fields**: Only local variables and loop variables.
3. **Cannot Be `null`**: The type cannot be inferred if `null` is the only assigned value.

```
var value = null; // Compilation error
```

4. **Explicit Initialization Required**:

```
var uninitialized; // Compilation error
```

Best Practices with `var`

1. **Use When the Type is Obvious**:

```
var number = 42; // Obvious that it's an int
```

2. **Avoid in Complex Declarations**:

```
var result = processSomething(); // Avoid unless the method name conveys the type
```

Combining Sealed Classes and `var`

Sealed classes and `var` often complement each other, especially with pattern matching and local variable declarations.

Example: Pattern Matching with Sealed Classes

```
sealed interface Shape permits Circle, Rectangle {}

final class Circle implements Shape {
    double radius;
}

final class Rectangle implements Shape {
    double length, breadth;
}

public class ShapeDemo {
    public static void main(String[] args) {
        var shape = new Circle();
        if (shape instanceof Circle c) {
            System.out.println("Circle with radius: " + c.radius);
        }
    }
}
```

Summary

1. **Sealed Classes**:
    - Restrict extensibility.
    - Ensure exhaustive handling in pattern matching.
    - Used with `sealed`, `non-sealed`, and `final`.
2. `var`:
    - Enables type inference for local variables.

- Improves readability when types are clear.
- Cannot be used for fields, uninitialized variables, or `null`.

Both features aim to make Java more modern, expressive, and easier to maintain. Let me know if you'd like code examples or further clarifications!

Records in Java

Records, introduced in **Java 14** (as a preview) and finalized in **Java 16**, are a special kind of class in Java designed to hold immutable data. They simplify the creation of classes whose primary purpose is to store data by automatically generating boilerplate code like constructors, `toString()`, `equals()`, and `hashCode()` methods.

Key Features of Records

1. **Immutable Data Holders**:
   - Fields in records are final by default.
   - Records are inherently immutable unless mutable objects are stored.
2. **Compact Syntax**:
   - Declaring a record requires significantly less code compared to a traditional class.
3. **Auto-Generated Methods**:
   - **Constructor**: Automatically initializes fields.
   - **Accessors**: Getters are automatically created.
   - `toString()`: Provides a readable string representation.
   - `equals()` **and** `hashCode()`: Generated based on fields.
4. **Sealed by Design**:
   - Records are implicitly final, so they cannot be extended.

Defining a Record

Here’s an example of a record:

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

This single line generates:

1. A `Point` class with fields `x` and `y`.
2. A constructor `Point(int x, int y)`.
3. Getters `x()` and `y()`.
4. Overridden methods:
    - `toString()` → `"Point[x=1, y=2]"`
    - `equals()` and `hashCode()` based on `x` and `y`.

How Records Work

1. **Declaration**:

    ```
    public record Employee(String name, int age) {}
    ```

2. **Usage**:

    ```
    public class Main {
        public static void main(String[] args) {
            Employee emp = new Employee("Alice", 30);
            System.out.println(emp.name()); // Accessor for name
            System.out.println(emp.age());  // Accessor for age
            System.out.println(emp);        // Employee[name=Alice, age=30]
        }
    }
    ```

3. **Equality Check**:

    ```
    Employee e1 = new Employee("Alice", 30);
    Employee e2 = new Employee("Alice", 30);
    System.out.println(e1.equals(e2)); // true
    ```

Customizing Records

You can add custom methods and override generated methods, if necessary.

1. **Custom Methods**:

```
public record Rectangle(int length, int width) {
    public int area() {
        return length * width;
    }
}
```

2. **Custom Constructor**:

   - Records allow custom compact constructors to validate or transform input.

```
public record Point(int x, int y) {
    public Point {
        if (x < 0 || y < 0) {
            throw new IllegalArgumentException("Coordinates must be non-
negative");
        }
    }
}
```

3. **Overriding `toString`**:

```
@Override
public String toString() {
    return "CustomPoint[x=" + x + ", y=" + y + "]";
}
```

Limitations of Records

1. **Immutability**:

   - Fields are `final` and cannot be changed once the record is created.

2. **Cannot Extend Other Classes**:

- Records implicitly extend `java.lang.Record` and cannot extend any other class.

3. **Limited Use of Inheritance**:
   - While you can implement interfaces, you cannot define subclasses of records.
4. **Not Suitable for Complex Behavior**:
   - Records are optimized for data storage and should not have complex logic or mutable state.

When to Use Records

1. **Data Transfer Objects (DTOs)**:
   - Represent data that is passed between layers or services.
2. **Immutable Value Classes**:
   - Create immutable objects with pre-defined equality and hashing.
3. **Simplify Boilerplate**:
   - Reduce verbosity in code for simple models.

Example: Records and Interfaces

Records can implement interfaces:

```
public interface Shape {
    double area();
}

public record Circle(double radius) implements Shape {
    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Circle circle = new Circle(5);
        System.out.println(circle.area()); // 78.53981633974483
    }
}
```

Advanced Features

1. **Nested Records**: Records can be declared inside other classes:

```
public class Main {
    public record Address(String city, String country) {}
}
```

2. **Pattern Matching with Records** (Introduced in Java 19 as a preview): Use records with pattern matching in `switch` expressions:

```
public static String describeShape(Shape shape) {
    return switch (shape) {
        case Circle(double radius) -> "Circle with radius: " + radius;
        case Rectangle(int length, int width) -> "Rectangle with area: " +
(length * width);
        default -> "Unknown shape";
    };
}
```

Key Differences Between Records and Classes

Aspect	Traditional Class	Record
Purpose	General-purpose	Immutable data carriers
Boilerplate Code	Requires manual writing (constructor, equals, etc.)	Automatically generated
Inheritance	Can extend classes	Cannot extend classes

Aspect	Traditional Class	Record
Immutability	Optional, fields can be mutable	Always immutable, fields are final
Accessors	Manually defined	Automatically generated for fields

Conclusion

Records in Java are a modern, concise way to represent immutable data structures. They are ideal for reducing boilerplate in simple classes and enable cleaner, more maintainable code. While they have limitations (e.g., immutability, no inheritance), they are a powerful addition to Java's language features.