

**JAVASCALER**

# **JAVA**

## **ZERO TO HERO**

From Novice to Java Virtuoso – Unleash  
Your Inner Hero!

**MD KASHIF ALI**

# Copyrights

Title: Java Zero to Hero by JavaScaler

Author: Md Kashif Ali

Copyright © 2023 by Md Kashif Ali

All rights reserved.

No part of this eBook may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the copyright holder, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

For permissions requests, contact:

Email: [javascaler@gmail.com](mailto:javascaler@gmail.com)

Website: <https://javascaler.com>

This eBook, "Java Zero to Hero," is the intellectual property of JavaScaler by Md Kashif Ali. The content contained herein, including but not limited to text, graphics, images, and code samples, is intended for educational purposes only and is provided "as is" without warranties or guarantees of any kind. The author and JavaScaler shall not be liable for any errors, omissions, or damages arising from the use of the information provided in this eBook.

## Disclaimer:

The information presented in this eBook is based on the author's experience and research at the time of writing. While every effort has been made to ensure the accuracy of the information, technology and best practices evolve, and some information may become outdated or subject to change. Readers are advised to verify the information and use their discretion when applying it to their specific circumstances. The author and JavaScaler disclaim any liability for any direct, indirect, incidental, consequential, or special damages arising out of or in any way connected with the use of this eBook or the information presented herein.

Please note that this eBook is not intended to replace professional advice. Readers should consult with qualified experts or professionals in the relevant fields for specific advice or guidance.

By reading and using this eBook, you agree to abide by the terms and conditions mentioned herein. Unauthorized use or distribution of this eBook may be subject to

legal action.

Thank you for respecting the intellectual property rights and supporting the author and JavaScaler's efforts to provide valuable educational content.

# Table of Contents

## Java Basics

- What is Java
- History of Java
- Features of Java
- C++ vs Java
- Hello Java Program
- Program Internal
- How to set a path?
- JDK, JRE and JVM
- JVM: Java Virtual Machine
- Java Variables
- Java Data Types
- Unicode System
- Operators
- Keywords

## Control Statements

- Java Control Statements
- Java If-else
- Java Switch
- Java For Loop
- Java While Loop
- Java Do While Loop
- Java Break
- Java Continue
- Java Comments

## Java Object Class

- Java OOPs Concepts
- Naming Convention
- Object and Class
- Method
- Constructor
- static keyword
- this keyword

## Java Inheritance

- Inheritance(IS-A)
- Aggregation(HAS-A)

## Java Polymorphism

- Method Overloading
- Method Overriding
- Covariant Return Type
- super keyword
- Instance Initializer block
- final keyword
- Runtime Polymorphism
- Dynamic Binding
- instanceof operator

## Java Abstraction

- Abstract class
- Interface
- Abstract vs Interface

## Java Encapsulation

- Package
- Access Modifiers Encapsulation

## Java Array

- Java Array

## Java OOPs Misc

- Object class
- Object Cloning
- Math class
- Wrapper Class
- Java Recursion
- Call By Value
- strictfp keyword
- javadoc tool
- Command Line Arg
- Object vs Class
- Overloading vs Overriding

## Java String

- What is String
- Immutable String
- String Comparison
- String Concatenation
- Substring
- Methods of String class
- StringBuffer class
- StringBuilder class
- String vs StringBuffer
- StringBuffer vs Builder
- Creating Immutable class toString method StringTokenizer class

## Java Regex

- Matcher class
- Pattern class
- Example of Java Regular Expressions
- Java regex API

## Exception Handling

- Java Exceptions
- Java Try-catch block
- Java Multiple Catch Block
- Java Nested try
- Java Finally Block
- Java Throw Keyword
- Java Exception Propagation
- Java Throws Keyword
- Java Throw vs Throws
- Final vs Finally vs Finalize
- Exception Handling with Method Overriding
- Java Custom Exceptions

## Java Inner Class

- What is inner class
- Member Inner class
- Anonymous Inner class
- Local Inner class
- static nested class
- Nested Interface

## Java Multithreading

- What is Multithreading
- Life Cycle of a Thread
- How to Create Thread
- Thread Scheduler
- Sleeping a thread
- Start a thread twice
- Calling run() method
- Joining a thread
- Naming a thread
- Thread Priority
- Daemon Thread
- Thread Pool
- Thread Group
- ShutdownHook
- Performing multiple task
- Garbage Collection
- Runtime class

## Java Synchronisation

- Synchronisation in java
- Synchronised block
- static synchronisation
- Deadlock in Java
- Inter-thread Comm
- Interrupting Thread
- Reentrant Monitor

## Java I/O

- Java Input/Output
- FileOutputStream
- FileInputStream
- BufferedOutputStream
- BufferedInputStream
- SequenceInputStream
- ByteArrayOutputStream
- ByteArrayInputStream
- DataOutputStream
- DataInputStream
- Java FilterOutputStream
- Java FilterInputStream
- Java ObjectOutputStream
- Java ObjectOutputStreamField
- Console
- FilePermission

- Writer
- Reader
- FileWriter
- FileReader
- BufferedWriter
- BufferedReader
- CharArrayReader
- CharArrayWriter
- PrintStream
- PrintWriter
- OutputStreamWriter
- InputStreamReader
- PushbackInputStream
- PushbackReader
- StringWriter
- StringReader
- PipedWriter
- PipedReader
- FilterWriter
- FilterReader
- File
- FileDescriptor
- RandomAccessFile
- java.util.Scanner

## **Java Serialization**

- Java Serialization
- Java transient keyword

## **Java Networking**

- Networking Concepts
- Socket Programming
- URL class
- URLConnection class
- HttpURLConnection
- InetAddress class
- DatagramSocket class

## **Java Collections**

- Collection Framework
- Java ArrayList
- Java LinkedList
- ArrayList vs LinkedList
- Java List Interface

- Java HashSet
- Java LinkedHashSet
- Java TreeSet
- Queue & PriorityQueue
- Deque & ArrayDeque
- Java Map Interface
- Java HashMap
- Working of HashMap
- Java LinkedHashMap
- Java TreeMap
- Java Hashtable
- HashMap vs Hashtable
- Java EnumSet
- Java EnumMap
- Collections class
- Sorting Collections
- Comparable interface
- Comparator interface
- Comparable vs Comparator
- Properties class
- ArrayList vs Vector
- Java Vector
- Java Stack
- Java Collection Interface
- Java Iterator Interface
- Java Deque Interface
- Java ConcurrentHashMap
- Java ConcurrentLinkedQueue

# Java Zero to Hero

## Java Basics

### What is Java?

Java is a high-level, object-oriented programming language that is designed to be portable, secure, and robust. It was developed by James Gosling and his team at Sun Microsystems (later acquired by Oracle Corporation) and first released in 1995. Java is known for its "write once, run anywhere" (WORA) principle, which means that Java code can be written once and executed on any platform that has a Java Virtual Machine (JVM) installed, without the need for recompilation. This platform independence is achieved by using the Java bytecode format, which is an intermediate representation of the code that is platform-neutral.

Java is widely used for building various types of applications, including web applications, desktop applications, mobile apps, enterprise systems, and more. It has become one of the most popular programming languages due to its versatility, ease of use, and robustness.

### History of Java

- **1991:** The development of Java began under the name "Oak" at Sun Microsystems by James Gosling and his team. The initial focus was on creating a programming language for consumer electronic devices.
- **1995:** The first public release of Java (JDK 1.0) was made on May 23, 1995. It included the core features of the Java language and the Java applet technology, which allowed small Java programs to be embedded into web pages.
- **1996:** JDK 1.1 was released, introducing new features and improvements to the language and the standard library.
- **1998:** JDK 1.2 (Java 2) was released, bringing significant updates to the language and the platform, including the introduction of the Swing GUI toolkit and the Collections Framework.
- **2004:** JDK 1.5 (Java 5) was released, introducing several important language features like generics, enhanced for loop, autoboxing, and annotations.
- **2011:** Oracle Corporation acquired Sun Microsystems, becoming the new steward of the Java platform.

- **2014:** JDK 1.8 (Java 8) was released, introducing lambda expressions, the Stream API, and other language enhancements.
- **2017:** JDK 9 introduced modularity with the Java Platform Module System (JPMS).
- **2020:** JDK 14, 15, 16, and 17 were released successively, each bringing new features and improvements to the language and platform.

## Features of Java

1. **Platform Independence:** Java code is compiled into bytecode, which is executed by the JVM. This bytecode can run on any platform with a compatible JVM, making Java platform-independent.
2. **Object-Oriented:** Java follows the object-oriented programming paradigm, supporting concepts like classes, objects, inheritance, polymorphism, and encapsulation.
3. **Garbage Collection:** Java has an automatic garbage collection mechanism that manages memory, freeing developers from manual memory management.
4. **Robustness:** Java's strong type-checking, exception handling, and memory management contribute to its robustness.
5. **Security:** Java provides a secure execution environment by using a sandbox model for applets and built-in security mechanisms.
6. **Multi-threading Support:** Java has built-in support for concurrent programming through threads, enabling the creation of multithreaded applications.
7. **Rich Standard Library:** Java comes with a vast standard library that provides a wide range of functionalities, making development faster and more efficient.
8. **High Performance:** Although Java is an interpreted language, its bytecode is compiled at runtime by the JVM, resulting in high performance.

## C++ vs. Java

C++ and Java are both widely used programming languages, but they have some fundamental differences:

- **Syntax:** C++ uses a more complex syntax compared to Java. Java was designed to have a simpler and more user-friendly syntax.
- **Memory Management:** C++ requires manual memory management using pointers, whereas Java has automatic garbage collection, making it less prone to

memory-related bugs.

- **Platform Independence:** C++ code is compiled into platform-specific machine code, while Java code is compiled into platform-independent bytecode.
- **Performance:** C++ is generally considered to have better performance than Java because it is compiled directly to machine code. However, Java performance has improved significantly with advancements in JIT (Just-In-Time) compilation.
- **Object-Oriented Features:** Both languages support object-oriented programming, but Java enforces stricter object-oriented principles, like not allowing multiple inheritances and using interfaces.
- **Standard Library:** Java has a more extensive and consistent standard library compared to C++, which can sometimes have a more fragmented ecosystem due to various libraries and implementations.

## Hello Java Program

Below is a simple "Hello World" program in Java:

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

Explanation:

- We define a class named `HelloWorld`. In Java, all code must be inside classes.
- The `public` keyword indicates that the class is accessible from other classes.
- The `static` keyword is used to define a static method that belongs to the class itself, not to any specific instance of the class.
- The `void` keyword indicates that the `main` method does not return any value.
- `main` is the entry point of the program. It is the method that will be executed when we run the program.
- The `String[] args` is an array of strings representing the command-line arguments passed to the program.
- `System.out.println()` is a method used to print the text "Hello, Java!" to the console.

## Program Internal

The "Program Internal" is not a standard term in Java. However, if you are referring to the internal structure of a Java program, we can briefly discuss the Java compilation and execution process.

### 1. Java Compilation Process:

- Java source code is written in `.java` files.
- The Java compiler ( `javac` ) translates the human-readable Java source code into platform-independent bytecode. The bytecode is stored in `.class` files.

### 2. Java Execution Process:

- The Java Virtual Machine (JVM) is responsible for executing Java bytecode.
- When you run a Java program, the JVM loads the necessary classes and starts executing the `main` method of the specified class (the entry point of the program).

### 3. Java Bytecode:

- Java bytecode is an intermediate representation of the Java source code. It is not directly executed by the operating system but by the JVM.
- The bytecode consists of instructions that the JVM can understand and execute.

It is important to note that understanding the internal details of the Java program, such as bytecode, is not usually required for typical Java programming tasks. However, having a basic understanding of these concepts can be helpful when dealing with advanced topics or troubleshooting performance issues.

## Setting Path:

Setting the path in Java is essential to make the Java Development Kit (JDK) tools, including the Java compiler ( `javac` ) and Java runtime ( `java` ), accessible from any directory on your computer. The path allows the operating system to find these tools when you run Java commands from the command prompt or terminal.

Here's how to set the path in Windows and Unix-based systems (Linux and macOS):

### Windows:

1. Find the JDK installation directory. It usually looks like `C:\Program Files\Java\jdk_version`.
2. Right-click on "This PC" or "My Computer" and select "Properties."

3. Click on "Advanced system settings" (on the left side of the window).
4. In the System Properties window, click the "Environment Variables" button.
5. Under "System Variables," find the "Path" variable, and click "Edit."
6. Click "New" and add the path to the JDK's "bin" directory (e.g., `C:\Program Files\Java\jdk_version\bin`).
7. Click "OK" on all the windows to save the changes.

### Unix-based Systems (Linux and macOS):

1. Open the terminal.
2. Find the JDK installation directory. It's usually located at `/usr/lib/jvm/jdk_version`.
3. Edit the `.bashrc` or `.bash_profile` file in your home directory using a text editor (e.g., `nano`, `vi`, or `gedit`).
4. Add the following line to the file:  
Replace `/usr/lib/jvm/jdk_version` with the actual path to your JDK installation.

```
export PATH="/usr/lib/jvm/jdk_version/bin:$PATH"
```

5. Save the file and exit the text editor.
6. Run `source ~/.bashrc` or `source ~/.bash_profile` in the terminal to apply the changes to the current session.

After setting the path, you can verify that it worked by opening a new terminal or command prompt and running `java -version` and `javac -version` commands. They should display the version information of your installed JDK.

### JDK, JRE, and JVM:

- **JDK (Java Development Kit):** JDK is a software development kit provided by Oracle (or other vendors) that includes tools needed to develop, compile, and run Java applications. It contains the Java compiler (`javac`), Java runtime (`java`), Java documentation generator (`javadoc`), and other utilities. JDK is essential for Java developers.
- **JRE (Java Runtime Environment):** JRE is a subset of JDK and is required to run Java applications. It includes the Java Virtual Machine (JVM) and the Java class libraries needed for executing Java bytecode. Users who only need to run Java applications but not develop them can install the JRE.

- **JVM (Java Virtual Machine):** JVM is the heart of the Java platform. It is responsible for executing Java bytecode. JVM is an abstract machine that provides a runtime environment in which Java bytecode can be executed. JVM implementations are platform-specific, allowing Java programs to be platform-independent.

### Java Variables:

In Java, variables are used to store data values that can be manipulated within a program. Before using a variable, it needs to be declared with a specific data type. Java supports the following variable types:

#### 1. Primitive Data Types:

These are basic data types built into the language.

- Numeric Types: `byte`, `short`, `int`, `long`, `float`, and `double`.
- Characters: `char`.
- Boolean: `boolean`.

#### 2. Reference Data Types:

These types are references to objects in memory.

- Examples: `String`, user-defined classes, arrays, etc.

### Java Data Types:

#### 1. Primitive Data Types:

- `byte`: 8-bit signed integer. Range: -128 to 127.
- `short`: 16-bit signed integer. Range: -32,768 to 32,767.
- `int`: 32-bit signed integer. Range:  $-2^{31}$  to  $2^{31} - 1$ .
- `long`: 64-bit signed integer. Range:  $-2^{63}$  to  $2^{63} - 1$ .
- `float`: 32-bit floating-point. Can represent decimal numbers.
- `double`: 64-bit floating-point. More precise than `float`.
- `char`: 16-bit Unicode character. Represents single characters.
- `boolean`: Represents `true` or `false`.

#### 2. Reference Data Types:

- `String`: A sequence of characters.
- Arrays: Collections of elements of the same type.

### Unicode System:

Unicode is a character encoding standard that aims to support all the characters from all the writing systems in the world. In Java, characters are represented using the Unicode character set, allowing Java to be used with multiple languages and character sets.

For example, you can declare a Unicode character in Java like this:

```
char unicodeChar = '\\u0041'; // Represents the character 'A'
```

The escape sequence `\\u` is used to indicate a Unicode character, followed by the four-digit hexadecimal representation of the Unicode code point.

### Operators:

Java supports various types of operators, which are used to perform operations on variables and values. Here are some common operators in Java:

- **Arithmetic Operators:** `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulus).
- **Assignment Operators:** `=`, `+=`, `-=`, `*=`, `/=`, `%=`.
- **Increment and Decrement Operators:** `++` (increment), `--` (decrement).
- **Relational Operators:** `==` (equals), `!=` (not equals), `>`, `<`, `>=`, `<=`.
- **Logical Operators:** `&&` (logical AND), `||` (logical OR), `!` (logical NOT).
- **Bitwise Operators:** `&` (bitwise AND), `|` (bitwise OR), `^` (bitwise XOR), `~` (bitwise NOT).

### Keywords:

Keywords in Java are reserved words that have predefined meanings and cannot be used as identifiers (e.g., variable names, class names). They are an integral part of the language and serve specific purposes. Some examples of Java keywords include:

```
abstract, boolean, break, byte, case, catch, class, continue, default, do, double, else, extends, final, finally, float, for, if, implements, import, instanceof, int, interface, long, new, package, private, protected, public, return, short, static, super, switch, this, throw, throws, try, void, while
```

These keywords are used for various purposes, such as defining classes, control flow, variable declaration, method definition, and more. It's essential to be familiar

with these keywords when programming in Java.

## Java Control Statements:

Control statements in Java are used to control the flow of a program's execution based on certain conditions or iterations. They help make decisions and repeat actions as needed.

### Java If-else:

The `if-else` statement is used to execute a block of code conditionally. If the given condition is true, the code inside the `if` block is executed; otherwise, the code inside the `else` block (if provided) is executed.

```
int num = 10;

if (num > 0) {
    System.out.println("The number is positive.");
} else {
    System.out.println("The number is non-positive.");
}
```

### Java Switch:

The `switch` statement is used to perform different actions based on the value of a variable or an expression. It provides an alternative to using multiple `if-else` statements.

```
int dayOfWeek = 3;

switch (dayOfWeek) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    default:
        System.out.println("Invalid day");
}
```

### Java For Loop:

The `for` loop is used to execute a block of code repeatedly for a fixed number of times. It consists of three parts: initialization, condition, and increment or decrement.

```
for (int i = 1; i <= 5; i++) {  
    System.out.println("Iteration: " + i);  
}
```

### Java While Loop:

The `while` loop repeatedly executes a block of code as long as the given condition is true.

```
int count = 1;  
  
while (count <= 5) {  
    System.out.println("Count: " + count);  
    count++;  
}
```

### Java Do-While Loop:

The `do-while` loop is similar to the `while` loop, but it ensures that the block of code is executed at least once before checking the condition.

```
int x = 1;  
  
do {  
    System.out.println("Value of x: " + x);  
    x++;  
} while (x <= 5);
```

### Java Break:

The `break` statement is used to terminate the loop or switch statement prematurely, based on a condition.

```
for (int i = 1; i <= 10; i++) {  
    if (i == 5) {  
        break; // Terminate the loop when i equals 5  
    }  
    System.out.println("i: " + i);  
}
```

### Java Continue:

The `continue` statement is used to skip the current iteration and continue with the next iteration of a loop based on a condition.

```
for (int i = 1; i <= 5; i++) {  
    if (i == 3) {  
        continue; // Skip the iteration when i equals 3  
    }  
    System.out.println("i: " + i);  
}
```

## Java Comments:

Comments in Java are used to provide explanations or documentation within the code. They are ignored by the compiler and not executed as part of the program.

- Single-line comments start with `//` and continue until the end of the line:

```
// This is a single-line comment.
```

- Multi-line comments start with `/*` and end with `*/`, allowing multiple lines to be commented at once:

```
/*  
This is a multi-line comment.  
It spans across multiple lines.  
*/
```

- Javadoc comments are used for generating documentation. They start with `/**` and end with `*/`. Javadoc comments are placed before classes, methods, or fields and can include additional tags for documentation generation:

```
/**  
 * This method calculates the sum of two numbers.  
 * @param a The first number.  
 * @param b The second number.  
 * @return The sum of a and b.  
 */  
public int sum(int a, int b) {  
    return a + b;  
}
```

Using comments effectively helps improve code readability and makes it easier for other developers to understand your code.

## Java Object Class:

In Java, the `Object` class is the root class of all other classes. It is defined in the `java.lang` package and is automatically inherited by all other classes, either directly or indirectly. Every class in Java is a subclass of the `Object` class.

The `Object` class provides several important methods that are inherited by all classes, such as `equals()`, `hashCode()`, `toString()`, `getClass()`, and more. These methods can be overridden in user-defined classes to provide custom behavior.

```
public class MyClass {  
    // MyClass inherits from the Object class implicitly  
}
```

## Java OOPs Concepts:

Java is an object-oriented programming language, and its primary focus is on the concepts of Object-Oriented Programming (OOP). Some of the key OOPs concepts in Java include:

1. **Encapsulation:** Encapsulation is the bundling of data (attributes) and methods that operate on that data within a single unit, called a class. It helps in data hiding and prevents direct access to the internal data from outside the class.
2. **Inheritance:** Inheritance allows a class (subclass) to inherit the properties and behaviors of another class (superclass). It promotes code reusability and allows new classes to extend existing classes.
3. **Polymorphism:** Polymorphism allows an object to take on multiple forms. It can be achieved through method overloading (compile-time polymorphism) and method overriding (run-time polymorphism).
4. **Abstraction:** Abstraction involves the creation of abstract classes and interfaces, which define the structure and behavior of classes without specifying their implementation details.
5. **Association:** Association represents a relationship between two classes, where one class uses the functionalities provided by another class.
6. **Aggregation:** Aggregation is a special form of association that represents a "has-a" relationship. It implies a relationship where one class contains another class as part of its structure.

7. **Composition:** Composition is a stronger form of aggregation, where one class is composed of one or more other classes. The composed classes cannot exist independently.

### Naming Convention:

In Java, following a consistent and standard naming convention is crucial for code readability and maintainability. Here are some common Java naming conventions:

1. **Class Names:** Start with an uppercase letter, and use camel case (capitalize the first letter of each word) for multiple words. Example: `MyClass`, `CarModel`.
2. **Variable Names:** Start with a lowercase letter, and use camel case for multiple words. Example: `age`, `firstName`.
3. **Constant Names:** Use uppercase letters and separate words with underscores. Example: `MAX_VALUE`, `PI`.
4. **Method Names:** Start with a lowercase letter, and use camel case for multiple words. Example: `calculateArea`, `getName`.

### Object and Class:

In Java, a class is a blueprint or a template for creating objects. It defines the attributes (fields) and behaviors (methods) that the objects of the class will have. An object is an instance of a class, representing a specific entity based on the class definition.

For example, consider a `Car` class:

```
public class Car {  
    String brand;  
    String model;  
  
    void startEngine() {  
        // Code to start the car's engine  
    }  
}
```

Here, `Car` is a class, and `brand`, `model`, and `startEngine()` are its members. When you create an object of the `Car` class, it represents a specific car with its own brand and model.

### Method:

A method in Java is a block of code that defines the behavior of an object. It is defined within a class and can be invoked on objects of that class. Methods are used

to perform specific actions or operations.

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

In this example, `add()` is a method of the `Calculator` class that takes two integer parameters and returns their sum.

### Constructor:

A constructor in Java is a special method used to initialize objects of a class. It has the same name as the class and is called automatically when an object is created. Constructors are used to set initial values for the object's attributes or perform other setup tasks.

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

In this example, the `Person` class has a constructor that takes the `name` and `age` as parameters and initializes the corresponding attributes.

### static Keyword:

The `static` keyword is used to declare members (variables and methods) that belong to the class itself, rather than to individual objects. They are shared by all objects of the class and can be accessed directly using the class name, without creating an instance of the class.

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

In this example, the `add()` method is declared as `static`, so you can call it using `MathUtils.add(2, 3)` without creating an instance of the `MathUtils` class.

### this Keyword:

The `this` keyword in Java is a reference to the current instance of the class. It is used to distinguish between instance variables and parameters with the same name, and to access instance variables and methods within the class.

```
public class Person {
    String name;

    public void setName(String name) {
        this.name = name;
    }
}
```

In this example, `this.name` refers to the instance variable `name`, while `name` refers to the method parameter `name`.

## Java Inheritance:

Inheritance is one of the fundamental concepts of object-oriented programming (OOP) that allows one class (subclass or derived class) to inherit the properties and behaviors of another class (superclass or base class). The subclass can extend the functionality of the superclass by adding new attributes and methods or by overriding existing ones.

In Java, inheritance is achieved using the `extends` keyword.

### IS-A Relationship (Inheritance):

Inheritance represents an IS-A relationship between classes. If Class B inherits from Class A, it means that objects of Class B are also objects of Class A, and they share a common "is-a" relationship.

Consider the following example:

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

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

```
}  
}
```

In this example, `Dog` is a subclass of `Animal`. So, a `Dog` IS-A `Animal`. When we create a `Dog` object, it can call methods defined in both `Dog` and `Animal`.

```
Animal animal = new Dog();  
animal.makeSound(); // Output: Dog barks.
```

## Java Aggregation:

Aggregation represents a HAS-A relationship between classes, where one class contains another class as part of its structure. It is a form of association that implies a relationship between a whole and its parts.

In Java, aggregation is achieved by creating a class with a reference to another class.

Consider the following example:

```
class Address {  
    String street;  
    String city;  
  
    Address(String street, String city) {  
        this.street = street;  
        this.city = city;  
    }  
}  
  
class Person {  
    String name;  
    Address address;  
  
    Person(String name, Address address) {  
        this.name = name;  
        this.address = address;  
    }  
}
```

In this example, `Person` has a HAS-A relationship with `Address`, as `Person` contains an `Address` object as one of its attributes.

```
Address address = new Address("Main Street", "New York");  
Person person = new Person("John Doe", address);
```

Here, the `person` object contains an `address` object, forming an aggregation relationship.

In summary, inheritance (IS-A) is used to achieve code reusability and represents a relationship between a superclass and its subclasses, while aggregation (HAS-A) represents a relationship between a whole class and its parts, allowing one class to contain another as an attribute. Both concepts are essential in designing object-oriented systems and promoting modularity and flexibility in code.

## Java Polymorphism:

Polymorphism is a key concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It provides a way to perform a single action in different ways based on the context. There are two main types of polymorphism in Java: compile-time polymorphism and runtime polymorphism.

### Method Overloading:

Method overloading is an example of compile-time (static) polymorphism. It allows a class to have multiple methods with the same name but different parameters. The compiler determines which method to call based on the number or types of arguments passed.

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

In this example, the `add()` method is overloaded with two different parameter types (integers and doubles).

### Method Overriding:

Method overriding is an example of runtime (dynamic) polymorphism. It occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The method in the subclass must have the same name, return type, and parameters as the method in the superclass.

```

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

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

```

In this example, the `Dog` class overrides the `makeSound()` method of the `Animal` class to provide a specific implementation for dogs.

### Covariant Return Type:

Covariant return type is a feature introduced in Java 5 that allows a subclass to override a method in its superclass with a more specific return type. The return type in the subclass method can be a subclass of the return type in the superclass method.

```

class Animal {
    Animal getAnimal() {
        return this;
    }
}

class Dog extends Animal {
    Dog getAnimal() {
        return this;
    }
}

```

In this example, the `getAnimal()` method in the `Dog` class returns a `Dog` object, which is a subclass of the `Animal` class's return type.

### super Keyword:

The `super` keyword is used to refer to the superclass's members (fields, methods, and constructors) from the subclass. It is often used to call the superclass's constructor or to access overridden methods or fields.

```

class Animal {
    String name;

    Animal(String name) {

```

```

        this.name = name;
    }
}

class Dog extends Animal {
    String breed;

    Dog(String name, String breed) {
        super(name); // Call the superclass constructor
        this.breed = breed;
    }
}

```

In this example, the `super(name)` call in the `Dog` constructor is used to invoke the `Animal` class's constructor with the `name` parameter.

### Instance Initializer Block:

The instance initializer block is a block of code that is executed whenever an instance of the class is created, before the constructor is called. It is defined within curly braces without any method name.

```

class MyClass {
    {
        // Instance initializer block
        System.out.println("Instance initializer block is executed.");
    }

    MyClass() {
        System.out.println("Constructor is called.");
    }
}

```

When you create an object of the `MyClass`, the instance initializer block is executed before the constructor.

### final Keyword:

The `final` keyword is used to declare that a variable, method, or class cannot be changed or overridden after its initialization or definition.

- Final Variable: Once initialized, the value of a final variable cannot be modified.

```

final int x = 10;

```

- Final Method: A final method cannot be overridden by subclasses.

```
class Parent {
    final void display() {
        // Method implementation
    }
}
```

- **Final Class:** A final class cannot be subclassed.

```
final class MyClass {
    // Class definition
}
```

### Runtime Polymorphism:

Runtime polymorphism is also known as dynamic polymorphism or late binding. It occurs when the method to be executed is determined at runtime based on the actual type of the object, not at compile time.

In Java, method overriding is an example of runtime polymorphism, where the decision about which method to call is made during runtime.

### Dynamic Binding:

Dynamic binding is the process of linking a method call to its method definition during runtime. It allows Java to support runtime polymorphism and method overriding.

When a method is overridden in a subclass, the actual method to be called is determined dynamically based on the type of the object at runtime.

### instanceof Operator:

The `instanceof` operator is used to check whether an object belongs to a specific class or its subclasses. It returns `true` if the object is an instance of the specified class or `false` otherwise.

```
class Animal { }

class Dog extends Animal { }

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();

        if (animal instanceof Dog) {
            System.out.println("The animal is a Dog.");
        }
    }
}
```

```
}  
}
```

In this example, `animal instanceof Dog` checks whether the `animal` object is an instance of the `Dog` class or any of its subclasses. Since `animal` is an instance of `Dog`, the output will be "The animal is a Dog."

## Java Abstraction:

Abstraction is one of the core principles of object-oriented programming (OOP) and refers to the process of hiding the implementation details of an object and exposing only the relevant features or behaviors to the outside world. It allows us to focus on what an object does rather than how it does it. Abstraction is achieved in Java through abstract classes and interfaces.

### Abstract Class:

An abstract class is a class that cannot be instantiated directly and is intended to be subclassed. It serves as a blueprint for other classes and can contain abstract methods (methods without a body) that must be implemented by its subclasses. An abstract class can also have concrete methods with an implementation.

```
abstract class Shape {  
    int x, y;  
  
    abstract void draw();  
  
    void moveTo(int newX, int newY) {  
        this.x = newX;  
        this.y = newY;  
    }  
}
```

In this example, `Shape` is an abstract class that has an abstract method `draw()` and a concrete method `moveTo()`. Classes that inherit from `Shape` must provide an implementation for the `draw()` method.

### Interface:

An interface in Java is a blueprint of a class that defines a contract for the classes that implement it. It contains only abstract methods and constants (implicitly `public`, `static`, and `final`). Interfaces are used to achieve multiple inheritance in Java.

```
interface Drawable {
    void draw();

    int MAX_WIDTH = 800;
    int MAX_HEIGHT = 600;
}
```

In this example, `Drawable` is an interface that has an abstract method `draw()` and two constants `MAX_WIDTH` and `MAX_HEIGHT`. A class can implement multiple interfaces and provide implementations for all the methods defined in those interfaces.

### Abstract Class vs. Interface:

1. **Instantiation:** Abstract classes cannot be instantiated directly, but interfaces cannot be instantiated at all. An abstract class can have a constructor, which is called when a subclass is instantiated, while an interface cannot have a constructor.
2. **Methods:** An abstract class can have both abstract and concrete methods, while an interface can only have abstract methods. In Java 8 and later, interfaces can have default methods (with an implementation) and static methods.
3. **Implementation:** A class can extend only one abstract class (single inheritance), but it can implement multiple interfaces (multiple inheritance).
4. **Fields:** Abstract classes can have instance variables, which can be inherited by their subclasses. Interfaces can only have `public`, `static`, and `final` variables (constants).
5. **Accessibility:** Abstract class members can have different access modifiers (e.g., `public`, `protected`, `private`, or package-private), while interface members are implicitly `public`.
6. **Purpose:** Abstract classes are used to provide a common base for related classes and can contain shared behavior. Interfaces are used to define a contract for unrelated classes to implement.

When to use Abstract Class:

- When you want to provide a common base class with some shared implementation for related classes.
- When you need to create a class hierarchy, and there is a "is-a" relationship between classes.

When to use Interface:

- When you want to define a contract that unrelated classes should implement.
- When you want to achieve multiple inheritance (a class can implement multiple interfaces but extend only one class).

In summary, both abstract classes and interfaces are used for abstraction in Java, but they serve different purposes and have different capabilities. Choosing between an abstract class and an interface depends on the specific design requirements and the relationship between the classes involved.

## Java Encapsulation:

Encapsulation is one of the four fundamental principles of object-oriented programming (OOP) and is closely related to the concept of data hiding. It refers to the bundling of data (attributes) and methods that operate on that data within a single unit (i.e., a class). The data is hidden or made private within the class, and access to the data is controlled through public methods, providing a layer of abstraction.

The main goals of encapsulation are to:

1. Hide the internal details of how an object's data is represented and manipulated.
2. Protect the data from unauthorized access or modification by external code.
3. Provide controlled access to the data through getter and setter methods.

### Package:

A package in Java is a mechanism to organize related classes and interfaces into a single unit. It helps in avoiding naming conflicts and provides a namespace for the classes. Packages allow you to group classes that have similar functionality or are part of the same module or application.

To declare a class inside a package, use the `package` keyword followed by the package name at the beginning of the source file:

```
package com.example.myapp;

public class MyClass {
    // Class implementation
}
```

In this example, `MyClass` is part of the `com.example.myapp` package.

### Access Modifiers:

Access modifiers control the visibility and accessibility of classes, variables, methods, and constructors in Java. There are four types of access modifiers:

1. **public**: Members with the `public` modifier are accessible from any class in any package.
2. **protected**: Members with the `protected` modifier are accessible within the same package and also by subclasses (even if they are in a different package).
3. **default (package-private)**: Members with no access modifier (i.e., no `public`, `protected`, or `private` keyword) are accessible only within the same package.
4. **private**: Members with the `private` modifier are accessible only within the same class.

Here's an example of using access modifiers:

```
package com.example.myapp;

public class MyClass {
    public int publicVariable;
    protected int protectedVariable;
    int defaultVariable;
    private int privateVariable;

    public void publicMethod() {
        // Method implementation
    }

    protected void protectedMethod() {
        // Method implementation
    }

    void defaultMethod() {
        // Method implementation
    }

    private void privateMethod() {
        // Method implementation
    }
}
```

In this example, `publicVariable`, `publicMethod`, `protectedVariable`, `protectedMethod`, `defaultVariable`, `defaultMethod`, `privateVariable`, and `privateMethod` have different access modifiers.

### Encapsulation (Continued):

Encapsulation is achieved in Java by using access modifiers to control the access to the attributes (data) and methods of a class. By making the attributes private and

providing public getter and setter methods, we can ensure that the data is accessed and modified in a controlled manner.

```
public class Person {  
    private String name;  
    private int age;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public void setAge(int age) {  
        if (age >= 0) {  
            this.age = age;  
        } else {  
            System.out.println("Invalid age. Age must be non-negative.");  
        }  
    }  
}
```

In this example, the `name` and `age` attributes are made private, and public getter and setter methods (`getName()`, `setName()`, `getAge()`, `setAge()`) are provided to access and modify these attributes. The setter method for `age` includes a validation check to ensure that the age is non-negative.

By using encapsulation, we can ensure that the internal state of an object is protected from unauthorized access and modification, promoting better code maintainability and reusability. Clients of the class interact with the object through its public methods, and the internal details remain hidden from them.

## Java Array:

An array in Java is a collection of elements of the same data type that are stored in contiguous memory locations. Arrays provide a way to store multiple values under a single variable name, making it easier to manage and access data. Each element in an array is identified by its index, starting from 0.

### Types of Array in Java:

1. **Single-Dimensional Array:** An array that contains elements in a linear sequence is called a single-dimensional array. It is also known as a one-dimensional array.
2. **Multidimensional Array:** An array that contains multiple arrays as its elements is called a multidimensional array. Common examples include two-dimensional (2D) and three-dimensional (3D) arrays.

### Single-Dimensional Array in Java:

A single-dimensional array in Java is declared using square brackets `[]` after the data type, followed by the array name.

```
// Declare a single-dimensional array to store integers
int[] numbers;
```

### Multidimensional Array:

A multidimensional array in Java is an array of arrays. It can have multiple dimensions, such as 2D or 3D arrays.

```
// Declare a 2D array to store integers
int[][] matrix;

// Declare a 3D array to store characters
char[][][] cube;
```

### Example of Java Array:

Here's an example of how to declare, instantiate, and initialize a single-dimensional array in Java:

```
public class ArrayExample {
    public static void main(String[] args) {
        // Declare an array to store integers
        int[] numbers;

        // Instantiate the array with 5 elements
        numbers = new int[5];

        // Initialize the elements of the array
        numbers[0] = 10;
        numbers[1] = 20;
        numbers[2] = 30;
        numbers[3] = 40;
        numbers[4] = 50;

        // Access and print the elements of the array
    }
}
```

```

        for (int i = 0; i < numbers.length; i++) {
            System.out.println("Element at index " + i + ": " + numbers[i]);
        }
    }
}

```

Output:

```

Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
Element at index 4: 50

```

In this example, we declare an array `numbers` to store integers, instantiate it with 5 elements using `new int[5]`, and then initialize the elements with values. The elements are accessed and printed using a `for` loop.

### Declaration, Instantiation, and Initialization of Java Array:

1. Declaration: The syntax for declaring an array involves specifying the data type of the elements followed by square brackets `[]` and the array name.

```
int[] numbers;
```

1. Instantiation: To create an array and allocate memory for its elements, use the `new` keyword followed by the data type and the number of elements in square brackets.

```
numbers = new int[5];
```

1. Initialization: After creating the array, you can initialize its elements with specific values using the array index.

```

numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;
numbers[4] = 50;

```

Alternatively, you can declare, instantiate, and initialize an array in a single line:

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

This is a shorthand way of creating an array and assigning values to its elements in one step. The size of the array is automatically determined based on the number of values provided.

## Java OOPs Misc

### 1. Object class:

The `Object` class is the root class of the Java class hierarchy. It is the superclass of all other classes in Java, either directly or indirectly. Every class in Java implicitly extends the `Object` class. The `Object` class provides several important methods that are inherited by all classes, such as `equals()`, `hashCode()`, `toString()`, `getClass()`, and more. These methods can be overridden in user-defined classes to provide custom behavior.

#### Example:

```
public class ObjectClassExample {
    public static void main(String[] args) {
        Object obj1 = new Object();
        Object obj2 = new Object();

        // equals() method compares object references
        System.out.println("obj1 equals obj2: " + obj1.equals(obj2)); // Output: false

        // hashCode() method returns the hash code of the object
        System.out.println("HashCode of obj1: " + obj1.hashCode());
        System.out.println("HashCode of obj2: " + obj2.hashCode());

        // toString() method returns the string representation of the object
        System.out.println("String representation of obj1: " + obj1.toString());
        System.out.println("String representation of obj2: " + obj2.toString());

        // getClass() method returns the runtime class of the object
        System.out.println("Class of obj1: " + obj1.getClass());
        System.out.println("Class of obj2: " + obj2.getClass());
    }
}
```

### 2. Object Cloning:

Object cloning in Java is the process of creating a copy of an object. The `Cloneable` interface, available in the `java.lang` package, is used to indicate that a class supports cloning. The `clone()` method is used to create a shallow copy of the object.

To perform deep cloning (i.e., copying both the object and its internal references), additional customization is needed.

### Example:

```
class Student implements Cloneable {
    private String name;

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

    // Override the clone() method
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    @Override
    public String toString() {
        return "Student: " + name;
    }
}

public class ObjectCloningExample {
    public static void main(String[] args) {
        try {
            Student student1 = new Student("John");
            Student student2 = (Student) student1.clone();

            System.out.println("Original Object: " + student1);
            System.out.println("Cloned Object: " + student2);
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

### 3. Math class:

The `Math` class in Java is part of the `java.lang` package and provides several static methods for common mathematical operations. It includes methods for basic arithmetic operations (e.g., `add`, `subtract`, `multiply`, `divide`) as well as methods for more complex mathematical functions (e.g., `sqrt`, `pow`, `sin`, `cos`, `log`, etc.). The methods of the `Math` class are static, meaning they can be accessed directly using the class name, without the need to create an instance of the class.

### Example:

```

public class MathClassExample {
    public static void main(String[] args) {
        int x = 5;
        int y = 3;

        // Basic arithmetic operations
        int sum = Math.addExact(x, y);
        int difference = Math.subtractExact(x, y);
        int product = Math.multiplyExact(x, y);
        int quotient = Math.floorDiv(x, y);
        int remainder = Math.floorMod(x, y);

        System.out.println("Sum: " + sum);
        System.out.println("Difference: " + difference);
        System.out.println("Product: " + product);
        System.out.println("Quotient: " + quotient);
        System.out.println("Remainder: " + remainder);

        // More complex mathematical functions
        double squareRoot = Math.sqrt(x);
        double power = Math.pow(x, y);
        double sinValue = Math.sin(Math.toRadians(30));
        double cosValue = Math.cos(Math.toRadians(60));
        double logValue = Math.log(x);

        System.out.println("Square Root: " + squareRoot);
        System.out.println("Power: " + power);
        System.out.println("Sin Value: " + sinValue);
        System.out.println("Cos Value: " + cosValue);
        System.out.println("Log Value: " + logValue);
    }
}

```

#### 4. Wrapper Class:

In Java, a wrapper class is used to convert a primitive data type into an object. Each primitive data type (e.g., `int`, `double`, `boolean`, etc.) has a corresponding wrapper class (e.g., `Integer`, `Double`, `Boolean`, etc.). Wrapper classes are useful when you need to use primitive data types in situations that require objects, such as using collections or generics.

##### Example:

```

public class WrapperClassExample {
    public static void main(String[] args) {
        // Using primitive data types
        int intValue = 10;
        double doubleValue = 3.14;
        boolean booleanValue = true;

        // Using wrapper classes
        Integer integerObj = Integer.valueOf(intValue);
    }
}

```

```

    Double doubleObj = Double.valueOf(doubleValue);
    Boolean booleanObj = Boolean.valueOf(booleanValue);

    // Auto-boxing (automatic conversion from primitive to wrapper)
    Integer anotherIntegerObj = intValue;
    Double anotherDoubleObj = doubleValue;
    Boolean anotherBooleanObj = booleanValue;

    // Unboxing (automatic conversion from wrapper to primitive)
    int unboxedIntValue = integerObj;
    double unboxedDoubleValue = doubleObj;
    boolean unboxedBooleanValue = booleanObj;

    System.out.println("Using Wrapper Classes:");
    System.out.println("Integer Object: " + integerObj);
    System.out.println("Double Object: " + doubleObj);
    System.out.println("Boolean Object: " + booleanObj);

    System.out.println("Auto-boxing and Unboxing:");
    System.out.println("Another Integer Object: " + anotherIntegerObj);
    System.out.println("Another Double Object: " + anotherDoubleObj);
    System.out.println("Another Boolean Object: " + anotherBooleanObj);

    );

    System.out.println("Unboxed Values:");
    System.out.println("Unboxed Int Value: " + unboxedIntValue);
    System.out.println("Unboxed Double Value: " + unboxedDoubleValue);
    System.out.println("Unboxed Boolean Value: " + unboxedBooleanValue);
}
}

```

## 5. Java Recursion:

Recursion is a programming technique where a method calls itself to solve a problem. In Java, recursion can be used to solve problems that can be broken down into smaller, similar subproblems. A well-known example of recursion is the calculation of factorial or Fibonacci series.

### Example:

```

public class RecursionExample {
    // Factorial using recursion
    public static int factorial(int n) {
        if (n == 0 || n == 1) {
            return 1;
        } else {
            return n * factorial(n - 1);
        }
    }

    // Fibonacci series using recursion
    public static int fibonacci(int n) {

```

```

        if (n <= 1) {
            return n;
        } else {
            return fibonacci(n - 1) + fibonacci(n - 2);
        }
    }

    public static void main(String[] args) {
        int num = 5;
        System.out.println("Factorial of " + num + " is: " + factorial(num));
        System.out.println("Fibonacci series up to " + num + " terms:");
        for (int i = 0; i < num; i++) {
            System.out.print(fibonacci(i) + " ");
        }
    }
}

```

## 6. Call By Value:

Java uses call by value to pass arguments to methods. When you pass a primitive data type (e.g., `int`, `double`, `char`, etc.) to a method, a copy of the value is passed, and changes made to the parameter inside the method have no effect on the original variable outside the method. Similarly, when you pass an object to a method, the reference to the object is passed by value, not the actual object. Therefore, the method can modify the object's properties, but it cannot change the reference itself.

### Example:

```

class Student {
    String name;

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

    // Method to change the name of the student
    public void changeName(String newName) {
        this.name = newName;
    }
}

public class CallByValueExample {
    public static void main(String[] args) {
        // Primitive data type - int
        int x = 10;

        // Pass by value - a copy of 'x' is passed to the method
        changeValue(x);

        // 'x' remains unchanged outside the method
        System.out.println("x after method call: " + x);
    }
}

```

```

// Object - Student
Student student = new Student("John");

// Pass by value - a copy of the reference 'student' is passed to the method
changeName(student);

// The object is modified inside the method
System.out.println("Student's name after method call: " + student.name);
}

public static void changeValue(int value) {
    value = value + 5;
}

public static void changeName(Student student) {
    student.changeName("Alice");
}
}

```

## 7. strictfp keyword:

The `strictfp` keyword is used to enforce strict floating-point precision in Java. When a class or method is declared with the `strictfp` keyword, all floating-point calculations within that class or method are performed using strict floating-point rules defined by the IEEE 754 standard. This ensures consistent results across different platforms and architectures.

### Example:

```

public strictfp class StrictFPEExample {
    // Perform strict floating-point calculation
    public strictfp double performCalculation(double x, double y) {
        return x * y + Math.sqrt(x) / y;
    }

    public static void main(String[] args) {
        StrictFPEExample example = new StrictFPEExample();
        double result = example.performCalculation(5.5, 2.0);
        System.out.println("Result: " + result);
    }
}

```

## 8. javadoc tool:

The `javadoc` tool is a utility in Java used to generate HTML documentation from Java source code. It parses the source code, extracts the comments written in the Javadoc format, and generates API documentation in HTML format. Javadoc comments start with `/**` and can include additional tags to provide information about classes, methods, and fields.

To generate Javadoc documentation, you need to include Javadoc comments in your source code and run the `javadoc` command from the command line. The generated documentation will include information about classes, methods, constructors, fields, and comments.

## 9. Command Line Arg:

Command-line arguments allow you to pass information to a Java program when running it from the command line. They are specified after the name of the Java class and separated by spaces. Java programs can access command-line arguments through the `args` parameter in the `main` method. The `args` parameter is an array of strings, where each element represents a command-line argument.

### Example:

```
public class CommandLineArgExample {
    public static void main(String[] args) {
        // Check if command-line arguments are provided
        if (args.length > 0) {
            System.out.println("Command-line arguments:");
            for (String arg : args) {
                System.out.println(arg);
            }
        } else {
            System.out.println("No command-line arguments provided.");
        }
    }
}
```

## 10. Object vs. Class:

- **Class:** A class is a blueprint or template for creating objects. It defines the structure and behavior of objects but does not occupy memory space. It is like a user-defined data type.
- **Object:** An object is an instance of a class. It is a runtime entity that occupies memory and can be used to access the members (fields and methods) defined in its class.

### Example:

```
class Car {
    String brand;
    String model;

    // Constructor
    public Car(String brand, String model) {
```

```

        this.brand = brand;
        this.model = model;
    }

    // Method
    public void start() {
        System.out.println("Starting the car...");
    }
}

public class ObjectVsClassExample {
    public static void main(String[] args) {
        // Class - Car
        // Objects - car1, car2
        Car car1 = new Car("Toyota", "Camry");
        Car car2 = new Car("Honda", "Accord");

        // Accessing fields and methods of objects
        System.out.println("Car 1: " + car1.brand + " " + car1.model);
        System.out.println("Car 2: " + car2.brand + " " + car2.model);

        car1.start();
        car2.start();
    }
}

```

## 11. Overloading vs. Overriding:

- **Overloading:** Overloading is a concept in Java that allows a class to have multiple methods with the same name but different parameters. The methods must have different method signatures, i.e., the number or types of parameters should be different. Overloading is a form of compile-time polymorphism.
- **Overriding:** Overriding is a concept in Java that occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The method in the subclass must have the same name, return type, and parameters as the method in the superclass. Overriding is a form of runtime polymorphism.

### Example:

```

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

class Dog extends Animal {
    // Method overriding
    @Override
    public void makeSound() {

```

```

        System.out.println("Dog barks");
    }

    // Method overloading
    public void makeSound(String soundType) {
        System.out.println("Dog " + soundType);
    }
}

public class OverloadingOverridingExample {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Animal dog = new Dog();

        // Method overriding - Polymorphism
        animal.makeSound(); // Output: Animal makes a sound
        dog.makeSound();    // Output: Dog barks

        // Method overloading
        Dog myDog = new Dog();
        myDog.makeSound();           // Output: Dog barks
        myDog.makeSound("howls");   // Output: Dog howls
    }
}

```

In this example, the `Animal` class has a `makeSound()` method that is overridden in the `Dog` class. The `Dog` class also has an overloaded `makeSound()` method that takes a `String` parameter. The appropriate method is called based on the reference type at compile time (overloading) and the object type at runtime (overriding).

## Java String

In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string.

### CharSequence Interface:

The `CharSequence` interface is a part of the `java.lang` package and is implemented by the `String`, `StringBuilder`, and `StringBuffer` classes in Java. It represents a sequence of characters and provides a read-only access to the character data. As it is an interface, it cannot be directly instantiated, but its implementing classes can be used to work with character sequences in Java.

### What is String in Java?

In Java, a `String` is a sequence of characters. It is a class in the `java.lang` package and is one of the most commonly used classes in Java. `String` objects are immutable, which means that once a `String` is created, its value cannot be changed. Any modification to a `String` results in a new `String` object being created.

## How to create a String object?

There are two ways to create a `String` object in Java:

1. **String Literal:** A string literal is a sequence of characters enclosed in double quotes. When you create a string using a string literal, Java automatically creates a `String` object in the string pool if it does not already exist.
2. **By new keyword:** You can also create a `String` object using the `new` keyword. This creates a new `String` object in the heap memory, even if a similar string already exists in the string pool.

### Java String Example:

```
public class StringExample {
    public static void main(String[] args) {
        // String Literal
        String str1 = "Hello, World!";

        // By new keyword
        String str2 = new String("Java is fun!");

        System.out.println("String 1: " + str1);
        System.out.println("String 2: " + str2);
    }
}
```

### Java String class methods:

The `String` class in Java provides various methods to manipulate and work with strings. Here are some common methods of the `String` class:

```
public class StringMethodsExample {
    public static void main(String[] args) {
        String str = "Hello, Java!";

        // Length of the string
        int length = str.length();
        System.out.println("Length of the string: " + length);

        // Get character at a specific index
        char ch = str.charAt(4);
        System.out.println("Character at index 4: " + ch);

        // Concatenate strings
        String concatStr = str.concat(" Welcome");
        System.out.println("Concatenated string: " + concatStr);

        // Check if the string contains a specific substring
        boolean containsJava = str.contains("Java");
    }
}
```

```

        System.out.println("Contains 'Java': " + containsJava);

        // Check if the string starts with a specific prefix
        boolean startsWithHello = str.startsWith("Hello");
        System.out.println("Starts with 'Hello': " + startsWithHello);

        // Check if the string ends with a specific suffix
        boolean endsWithExclamation = str.endsWith("!");
        System.out.println("Ends with '!': " + endsWithExclamation);

        // Find the index of a substring
        int indexJava = str.indexOf("Java");
        System.out.println("Index of 'Java': " + indexJava);

        // Convert the string to uppercase and lowercase
        String uppercaseStr = str.toUpperCase();
        String lowercaseStr = str.toLowerCase();
        System.out.println("Uppercase: " + uppercaseStr);
        System.out.println("Lowercase: " + lowercaseStr);

        // Replace characters or substrings
        String replacedStr = str.replace("Java", "Python");
        System.out.println("Replaced string: " + replacedStr);

        // Split the string into an array of substrings
        String[] words = str.split(", ");
        for (String word : words) {
            System.out.println("Word: " + word);
        }

        // Remove leading and trailing whitespaces
        String trimmedStr = str.trim();
        System.out.println("Trimmed string: " + trimmedStr);
    }
}

```

These are just some of the many methods provided by the `String` class. The `String` class offers a wide range of methods to handle string manipulation, searching, and formatting in Java.

### Immutable String:

In Java, a `String` object is immutable, which means once it is created, its content cannot be changed. When you modify a `String`, a new `String` object is created with the modified content. The immutability of strings ensures their safety in multithreaded environments and allows for efficient string manipulation because the same string can be shared by multiple references without fear of modification.

### Example:

```

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

```

```

String str = "Hello";
str.concat(" Java"); // This will not modify the original string
System.out.println("Original String: " + str); // Output: Hello
    }
}

```

## String Comparison:

In Java, you can compare strings using the `equals()` method to check if the content of two strings is the same. For reference-based comparison, you can use the `==` operator, but it only checks if the two string references point to the same memory location, not the content.

### Example:

```

public class StringComparisonExample {
    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = "Hello";
        String str3 = new String("Hello");

        // Content-based comparison using equals()
        System.out.println("str1 equals str2: " + str1.equals(str2)); // Output: true
        System.out.println("str1 equals str3: " + str1.equals(str3)); // Output: true

        // Reference-based comparison using ==
        System.out.println("str1 == str2: " + (str1 == str2)); // Output: true
        System.out.println("str1 == str3: " + (str1 == str3)); // Output: false
    }
}

```

## String Concatenation:

String concatenation in Java can be performed using the `+` operator or the `concat()` method of the `String` class. When concatenating multiple strings using the `+` operator, the Java compiler automatically converts the expression into a `StringBuilder` or `StringBuffer` object for efficiency.

### Example:

```

public class StringConcatenationExample {
    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = " Java";

        // Using + operator
        String concatenatedStr1 = str1 + str2;

        // Using concat() method
    }
}

```

```

        String concatenatedStr2 = str1.concat(str2);

        System.out.println("Concatenated String (using +): " + concatenatedStr1); // Output: Hello Java
        System.out.println("Concatenated String (using concat()): " + concatenatedStr2); // Output: Hello Java
    }
}

```

## Substring:

The `substring()` method in Java is used to extract a portion of a string. It takes two parameters: the starting index (inclusive) and the ending index (exclusive) of the substring to be extracted.

## Example:

```

public class SubstringExample {
    public static void main(String[] args) {
        String str = "Hello, Java";

        // Extract a substring from index 0 to index 5 (exclusive)
        String subStr1 = str.substring(0, 5);
        System.out.println("Substring 1: " + subStr1); // Output: Hello

        // Extract a substring from index 7 to the end of the string
        String subStr2 = str.substring(7);
        System.out.println("Substring 2: " + subStr2); // Output: Java
    }
}

```

## Methods of String class:

The `String` class in Java provides many useful methods for string manipulation and analysis. Some common methods are `length()`, `charAt()`, `equals()`, `equalsIgnoreCase()`, `startsWith()`, `endsWith()`, `contains()`, `indexOf()`, `lastIndexOf()`, `replace()`, `toUpperCase()`, `toLowerCase()`, `trim()`, `split()`, and more.

## Example usage of some methods:

```

public class StringMethodsExample {
    public static void main(String[] args) {
        String str = "Hello, Java!";

        // Length of the string
        int length = str.length();
        System.out.println("Length of the string: " + length);

        // Get character at a specific index
        char ch = str.charAt(4);
    }
}

```

```

        System.out.println("Character at index 4: " + ch);

        // Check if the string contains a specific substring
        boolean containsJava = str.contains("Java");
        System.out.println("Contains 'Java': " + containsJava);

        // Convert the string to uppercase and lowercase
        String uppercaseStr = str.toUpperCase();
        String lowercaseStr = str.toLowerCase();
        System.out.println("Uppercase: " + uppercaseStr);
        System.out.println("Lowercase: " + lowercaseStr);

        // Replace characters or substrings
        String replacedStr = str.replace("Java", "Python");
        System.out.println("Replaced string: " + replacedStr);

        // Split the string into an array of substrings
        String[] words = str.split(" ");
        for (String word : words) {
            System.out.println("Word: " + word);
        }

        // Remove leading and trailing whitespaces
        String trimmedStr = str.trim();
        System.out.println("Trimmed string: " + trimmedStr);
    }
}

```

## StringBuffer class:

The `StringBuffer` class in Java provides a mutable sequence of characters. It is similar to `String` but allows modification of the content without creating a new object. This makes `StringBuffer` suitable for situations where frequent string modifications are required, such as in a loop.

Example usage of `StringBuffer`:

```

public class StringBufferExample {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello");

        // Append strings
        sb.append(", Java!");
        System.out.println("Appended string: " + sb); // Output: Hello, Java!

        // Insert a string at a specific position
        sb.insert(5, " World");
        System.out.println("Inserted string: " + sb); // Output: Hello World, Java!

        // Reverse the string
        sb.reverse();
        System.out.println("Reversed string: " + sb); // Output: !avaJ ,dlrow olleH
    }
}

```

```
}  
}
```

## StringBuilder class:

The `StringBuilder` class in Java is similar to `StringBuffer` in functionality but is not synchronized, making it more efficient for single-threaded environments. It is used when you need a mutable sequence of characters in a single-threaded application.

Example usage of `StringBuilder`:

```
public class StringBuilderExample {  
    public static void main(String[] args) {  
        StringBuilder sb = new StringBuilder("Hello");  
  
        // Append strings  
        sb.append(", Java!");  
        System.out.println("Appended string: " + sb); // Output: Hello, Java!  
  
        // Insert a string at a specific position  
        sb.insert(5, " World");  
        System.out.println("Inserted string: " + sb); // Output: Hello World, Java!  
  
        // Reverse the string  
        sb.reverse();  
        System.out.println("Reversed string:  
  
" + sb); // Output: !avaJ ,dlrow olleH  
    }  
}
```

## String vs. StringBuffer:

- `String`: `String` objects are immutable. Once created, their content cannot be changed. Every string modification results in a new `String` object, which may lead to unnecessary memory usage and reduced performance when dealing with frequent modifications.
- `StringBuffer`: `StringBuffer` objects are mutable. They allow modifications of the content without creating a new object. This makes them suitable for scenarios where frequent string modifications are needed, such as in loops. `StringBuffer` is synchronized and thread-safe.

## StringBuffer vs. StringBuilder:

- `StringBuffer`: `StringBuffer` is synchronized and thread-safe, making it suitable for multithreaded environments. It ensures that multiple threads can safely access and modify the content of the `StringBuffer`.

- `StringBuilder`: `StringBuilder` is not synchronized and not thread-safe, making it more efficient in single-threaded environments. It is preferred over `StringBuffer` when thread safety is not a concern.

### Creating Immutable class:

To create an immutable class, follow these steps:

1. Declare the class as `final` so that it cannot be subclassed.
2. Declare all fields as `final` and initialize them either in the constructor or at the time of declaration.
3. Provide only getter methods to access the fields. Do not provide setter methods.
4. Make sure to not allow any modification of the fields once the object is created.

### Example:

```
public final class ImmutableClassExample {
    private final String name;
    private final int age;

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

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

public class ImmutableClassExampleTest {
    public static void main(String[] args) {
        ImmutableClassExample immutableObj = new ImmutableClassExample("John", 30);

        // Access fields using getter methods
        System.out.println("Name: " + immutableObj.getName()); // Output: John
        System.out.println("Age: " + immutableObj.getAge());    // Output: 30
    }
}
```

## Java String Methods

1. `charAt(int index)`: Returns the character at the specified `index` within the string.

Example:

```
String str = "Hello";  
char result = str.charAt(1); // Returns 'e'
```

1. `compareTo(String anotherString)`: Compares the current string with `anotherString` lexicographically.

Example:

```
String str1 = "apple";  
String str2 = "banana";  
int result = str1.compareTo(str2); // Returns a negative value, because "apple" comes  
before "banana" lexicographically.
```

1. `concat(String str)`: Concatenates the specified `str` to the end of the current string.

Example:

```
String str1 = "Hello";  
String str2 = " World";  
String result = str1.concat(str2); // Returns "Hello World"
```

1. `contains(CharSequence sequence)`: Checks if the current string contains the specified `sequence`.

Example:

```
String str = "The quick brown fox";  
boolean result = str.contains("fox"); // Returns true
```

1. `endsWith(String suffix)`: Checks if the current string ends with the specified `suffix`.

Example:

```
String str = "Hello, World!";  
boolean result = str.endsWith("World!"); // Returns true
```

1. `equals(Object anObject)`: Compares the current string with the `anObject` for equality.

Example:

```
String str1 = "Hello";
String str2 = "hello";
boolean result = str1.equals(str2); // Returns false, because of case sensitivity
```

1. `equalsIgnoreCase(String anotherString)`: Compares the current string with `anotherString`, ignoring case considerations.

Example:

```
String str1 = "Hello";
String str2 = "hello";
boolean result = str1.equalsIgnoreCase(str2); // Returns true, as it ignores case
```

1. `format(String format, Object... args)`: Returns a formatted string using the specified `format` and `args`.

Example:

```
String name = "John";
int age = 30;
String result = String.format("My name is %s, and I am %d years old.", name, age);
// Returns "My name is John, and I am 30 years old."
```

1. `getBytes()`: Converts the string into a byte array using the platform's default charset encoding.

Example:

```
String str = "Hello";
byte[] bytes = str.getBytes();
```

1. `getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`: Copies characters from the string into the `dst` array.

Example:

```
String str = "Hello";
char[] dest = new char[3];
str.getChars(1, 4, dest, 0); // Copies "ell" into dest array
```

1. `indexOf(int ch)` : Returns the index of the first occurrence of the specified character `ch` within the string.

Example:

```
String str = "Hello";
int index = str.indexOf('o'); // Returns 4
```

1. `indexOf(int ch, int fromIndex)` : Returns the index of the first occurrence of the specified character `ch` within the string, starting the search from the specified `fromIndex`.

Example:

```
String str = "Hello, World!";
int index = str.indexOf('o', 5); // Returns 8 (searching from index 5)
```

1. `indexOf(String str)` : Returns the index of the first occurrence of the specified substring `str` within the string.

Example:

```
String str = "Hello, World!";
int index = str.indexOf("World"); // Returns 7
```

1. `indexOf(String str, int fromIndex)` : Returns the index of the first occurrence of the specified substring `str` within the string, starting the search from the specified `fromIndex`.

Example:

```
String str = "Hello, World!";
int index = str.indexOf("o", 5); // Returns 8 (searching from index 5)
```

1. `intern()` : Returns the canonical representation of the string, i.e., the unique instance of the string in the string pool.

Example:

```
String str1 = "Hello";
String str2 = new String("Hello");
String str3 = str2.intern();
boolean result = (str1 == str2); // Returns false, as they are different instances
boolean result2 = (str1 == str3); // Returns true, as they refer to the same instance
in the string pool
```

1. `isEmpty()`: Checks if the string is empty (i.e., it has a length of 0).

Example:

```
String str = "";
boolean result = str.isEmpty(); // Returns true
```

1. `join(CharSequence delimiter, CharSequence... elements)`: Joins the elements together into a single string using the specified `delimiter` between each element.

Example:

```
String[] fruits = {"apple", "orange", "banana"};
String result = String.join(", ", fruits);
// Returns "apple, orange, banana"
```

1. `lastIndexOf(int ch)`: Returns the index of the last occurrence of the specified character `ch` within the string.

Example:

```
String str = "Hello, World!";
int index = str.lastIndexOf('o'); // Returns 8
```

1. `lastIndexOf(int ch, int fromIndex)`: Returns the index of the last occurrence of the specified character `ch` within the string, searching backward from the specified `fromIndex`.

Example:

```
String str = "Hello, World!";
int index = str.lastIndexOf('o', 7); // Returns 4 (searching backward from index 7)
```

1. `lastIndexOf(String str)`: Returns the index of the last occurrence of the specified substring `str` within the string.

Example:

```
String str = "Hello, World!";  
int index = str.lastIndexOf("o"); // Returns 8
```

1. `lastIndexOf(String str, int fromIndex)`: Returns the index of the last occurrence of the specified substring `str` within the string, searching backward from the specified `fromIndex`.

Example:

```
String str = "Hello, World!";  
int index = str.lastIndexOf("o", 7); // Returns 4 (searching backward from index 7)
```

1. `length()`: Returns the length (number of characters) of the string.

Example:

```
String str = "Hello";  
int length = str.length(); // Returns 5
```

1. `replace(char oldChar, char newChar)`: Replaces all occurrences of `oldChar` with `newChar` and returns a new string with the replacements.

Example:

```
String str = "Hello, World!";  
String result = str.replace('o', 'X'); // Returns "HellX, WXXrld!"
```

1. `replace(CharSequence target, CharSequence replacement)`: Replaces all occurrences of the `target` sequence with the `replacement` sequence and returns a new string with the replacements.

Example:

```
String str = "Hello, World!";  
String result = str.replace("  
  
World", "Universe"); // Returns "Hello, Universe!"
```

1. `replaceAll(String regex, String replacement)`: Replaces all substrings that match the `regex` with the `replacement` string and returns a new string with the replacements.

Example:

```
String str = "Hello, World!";  
String result = str.replaceAll("[aeiou]", "-"); // Returns "H-l-l-, W-rld!"
```

1. `split(String regex)`: Splits the string into an array of substrings based on the provided `regex`.

Example:

```
String str = "apple,orange,banana";  
String[] result = str.split(","); // Returns ["apple", "orange", "banana"]
```

1. `startsWith(String prefix)`: Checks if the string starts with the specified `prefix`.

Example:

```
String str = "Hello, World!";  
boolean result = str.startsWith("Hello"); // Returns true
```

1. `startsWith(String prefix, int toffset)`: Checks if the substring of the string, starting from the specified `toffset`, starts with the specified `prefix`.

Example:

```
String str = "Hello, World!";  
boolean result = str.startsWith("World", 7); // Returns true (starts from index 7)
```

1. `substring(int beginIndex)`: Returns a new string that is a substring of the current string, starting from the specified `beginIndex`.

Example:

```
String str = "Hello, World!";  
String result = str.substring(7); // Returns "World!"
```

1. `substring(int beginIndex, int endIndex)` : Returns a new string that is a substring of the current string, starting from `beginIndex` and ending at `endIndex-1`.

Example:

```
String str = "Hello, World!";  
String result = str.substring(7, 12); // Returns "World"
```

1. `toCharArray()` : Converts the string to a character array.

Example:

```
String str = "Hello";  
char[] charArray = str.toCharArray(); // Returns ['H', 'e', 'l', 'l', 'o']
```

1. `toLowerCase()` : Converts all characters in the string to lowercase and returns a new string.

Example:

```
String str = "Hello, World!";  
String result = str.toLowerCase(); // Returns "hello, world!"
```

1. `toUpperCase()` : Converts all characters in the string to uppercase and returns a new string.

Example:

```
String str = "Hello, World!";  
String result = str.toUpperCase(); // Returns "HELLO, WORLD!"
```

1. `trim()` : Removes leading and trailing whitespaces from the string and returns a new string.

Example:

```
String str = "  Hello, World!  ";  
String result = str.trim(); // Returns "Hello, World!"
```

1. `valueOf(boolean b)` : Converts the boolean `b` into a string representation.

Example:

```
boolean flag = true;  
String result = String.valueOf(flag); // Returns "true"
```

1. `valueOf(char c)`: Converts the character `c` into a string representation.

Example:

```
char letter = 'A';  
String result = String.valueOf(letter); // Returns "A"
```

1. `valueOf(int i)`: Converts the integer `i` into a string representation.

Example:

```
int number = 42;  
String result = String.valueOf(number); // Returns "42"
```

1. `valueOf(long l)`: Converts the long `l` into a string representation.

Example:

```
long bigNumber = 1234567890L;  
String result = String.valueOf(bigNumber); // Returns "1234567890"
```

1. `valueOf(float f)`: Converts the float `f` into a string representation.

Example:

```
float pi = 3.14159f;  
String result = String.valueOf(pi); // Returns "3.14159"
```

1. `valueOf(double d)`: Converts the double `d` into a string representation.

Example:

```
double distance = 1000.45;  
String result = String.valueOf(distance); // Returns "1000.45"
```

These examples demonstrate the usage of various Java String methods for manipulating and working with strings in different scenarios.

## Java Regex:

Java Regular Expressions, also known as Java Regex, are used to match patterns in strings. They are a powerful tool for text processing and pattern matching. Java provides built-in support for regular expressions through two main classes: `Pattern` and `Matcher`.

### Matcher class:

The `Matcher` class is used to search for patterns defined by a `Pattern` object within a given input string. It provides methods to perform various operations on the matched patterns, such as finding all occurrences, replacing, and extracting groups.

### Pattern class:

The `Pattern` class is used to compile a regular expression into a pattern that can be used by the `Matcher` class for matching against input strings. It provides methods to create and manipulate regular expression patterns.

## Example of Java Regular Expressions:

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

public class RegexExample {
    public static void main(String[] args) {
        String input = "The quick brown fox jumps over the lazy dog.";

        // Define the regular expression pattern
        String regex = "quick.*fox";

        // Compile the pattern into a Pattern object
        Pattern pattern = Pattern.compile(regex);

        // Create a Matcher object to search for the pattern in the input string
        Matcher matcher = pattern.matcher(input);

        // Perform the matching operation
        if (matcher.find()) {
            System.out.println("Pattern found: " + matcher.group());
        } else {
            System.out.println("Pattern not found.");
        }
    }
}
```

Output:

```
Pattern found: quick brown fox
```

## Regex Character classes:

Character classes in regular expressions allow you to match any one character from a specific set of characters. They are enclosed within square brackets `[ ]`.

Commonly used character classes include:

- `[abc]`: Matches any single character 'a', 'b', or 'c'.
- `[0-9]`: Matches any single digit (0 to 9).
- `[a-zA-Z]`: Matches any single uppercase or lowercase letter.
- `[^0-9]`: Matches any character that is not a digit (negation).

## Regular Expression Character classes Example:

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

public class CharacterClassExample {
    public static void main(String[] args) {
        String input = "The car costs $500, the bike costs $200.";

        // Define the regular expression pattern to find all digits
        String regex = "[0-9]+";

        // Compile the pattern into a Pattern object
        Pattern pattern = Pattern.compile(regex);

        // Create a Matcher object to search for the pattern in the input string
        Matcher matcher = pattern.matcher(input);

        // Perform the matching operation and print all occurrences of digits
        while (matcher.find()) {
            System.out.println("Found: " + matcher.group());
        }
    }
}
```

Output:

Found: 500

Found: 200

## Regex Quantifiers:

Quantifiers in regular expressions allow you to specify how many times a character or group should occur. They modify the behavior of character classes and metacharacters.

Commonly used quantifiers include:

- `*`: Matches zero or more occurrences of the preceding character.
- `+`: Matches one or more occurrences of the preceding character.
- `?`: Matches zero or one occurrence of the preceding character.
- `{n}`: Matches exactly 'n' occurrences of the preceding character.

## Regular Expression Character classes and Quantifiers Example:

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

public class QuantifiersExample {
    public static void main(String[] args) {
        String input = "abbbbbbcde";

        // Define the regular expression pattern to find 'b' occurring 2 to 5 times
        String regex = "b{2,5}";

        // Compile the pattern into a Pattern object
        Pattern pattern = Pattern.compile(regex);

        // Create a Matcher object to search for the pattern in the input string
        Matcher matcher = pattern.matcher(input);

        // Perform the matching operation and print all occurrences
        while (matcher.find()) {
            System.out.println("Found: " + matcher.group());
        }
    }
}
```

Output:

Found: bbbbb

## Regex Metacharacters:

Metacharacters in regular expressions are characters that have a special meaning and are used to define the structure of the pattern.

Commonly used metacharacters include:

- `.`: Matches any single character except a newline.
- `^`: Matches the start of a line.
- `$`: Matches the end of a line.
- `|`: Acts as an OR operator to match either the expression before or after it.

## Regular Expression Metacharacters Example:

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

public class MetacharactersExample {
    public static void main(String[] args) {
        String input = "The cat sat on the mat.";

        // Define the regular expression pattern to find words that start with 'c' or 'm'
        String regex = "\\b(c|m)\\w+\\b";

        // Compile the pattern into a Pattern object
        Pattern pattern = Pattern.compile(regex);

        // Create a Matcher object to search for the pattern in the input string
        Matcher matcher = pattern.matcher(input);

        // Perform the matching operation and print all occurrences
        while (matcher.find()) {
            System.out.println("Found: " + matcher.group());
        }
    }
}
```

Output:

```
Found: cat
Found: mat
```

These examples illustrate some essential concepts of Java Regular Expressions, including character classes, quantifiers, and metacharacters. Regular expressions provide a powerful and flexible way to work with text patterns in Java.

## Exception Handling in Java:

Exception handling is a mechanism in Java that deals with runtime errors or exceptional situations that may occur during the program's execution. When an exception occurs, it disrupts the normal flow of the program, and without proper handling, it can lead to program termination.

### Example:

```
public class ExceptionHandlingExample {
    public static void main(String[] args) {
        int num1 = 10;
        int num2 = 0;

        try {
            int result = num1 / num2; // This will throw an ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }

        System.out.println("Program continues after exception handling.");
    }
}
```

Output:

```
Exception caught: / by zero
Program continues after exception handling.
```

In this example, we attempt to divide `num1` by `num2`, which is 0. This operation causes an `ArithmeticException` to be thrown. We handle the exception using a `try-catch` block, and the program continues to execute after the exception is caught.

## Java Try-Catch Block:

The `try-catch` block is used to enclose the code that may throw exceptions. If an exception occurs within the `try` block, it is caught by the corresponding `catch` block, which contains code to handle the exception.

## Example:

```
public class TryCatchExample {
    public static void main(String[] args) {
        try {
            int[] arr = {1, 2, 3};
            System.out.println(arr[5]); // This will throw an ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }

        System.out.println("Program continues after exception handling.");
    }
}
```

## Output:

```
Exception caught: 5
Program continues after exception handling.
```

In this example, we try to access the 6th element of an array with only 3 elements, causing an `ArrayIndexOutOfBoundsException`. The exception is caught and handled within the `catch` block.

## Java Multiple Catch Block:

Java allows us to use multiple `catch` blocks to handle different types of exceptions that may arise within the `try` block.

## Example:

```
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            String str = null;
            System.out.println(str.length()); // This will throw a NullPointerException
        } catch (NullPointerException e) {
            System.out.println("NullPointerException caught: " + e.getMessage());
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught: " + e.getMessage());
        }

        System.out.println("Program continues after exception handling.");
    }
}
```

Output:

```
NullPointerException caught: null
Program continues after exception handling.
```

In this example, we try to call the `length()` method on a null string, leading to a `NullPointerException`. We have multiple `catch` blocks to handle different exceptions.

## Java Nested Try:

We can use nested `try` blocks to handle exceptions at different levels of code execution.

### Example:

```
public class NestedTryExample {
    public static void main(String[] args) {
        try {
            int[] arr = {1, 2, 3};
            int num1 = 10;
            int num2 = 0;

            try {
                int result = num1 / num2; // This will throw an ArithmeticException
                System.out.println("Result: " + result);
            } catch (ArithmeticException e) {
                System.out.println("Inner Exception caught: " + e.getMessage());
            }

            System.out.println(arr[5]); // This will throw an ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Outer Exception caught: " + e.getMessage());
        }

        System.out.println("Program continues after exception handling.");
    }
}
```

Output:

```
Inner Exception caught: / by zero
Outer Exception caught: 5
Program continues after exception handling.
```

In this example, we have an outer `try-catch` block that handles an `ArrayIndexOutOfBoundsException`. Within this block, there's an inner `try-catch` block that handles an `ArithmeticException`.

## Java Finally Block:

The `finally` block is used to specify code that should be executed regardless of whether an exception occurs or not. It is often used to release resources, close files, or clean up operations.

### Example:

```
import java.io.*;

public class FinallyExample {
    public static void main(String[] args) {
        BufferedReader br = null;

        try {
            br = new BufferedReader(new FileReader("input.txt"));
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("IOException: " + e.getMessage());
        } finally {
            try {
                if (br != null)
                    br.close(); // Close the BufferedReader in the finally block
            } catch (IOException e) {
                System.out.println("Error closing BufferedReader: " + e.getMessage());
            }
        }
    }
}
```

In this example, we read data from a file using a `BufferedReader`. We close the `BufferedReader` in the `finally` block to ensure that it gets closed regardless of whether an exception occurs or not.

## Java Throw Keyword:

The `throw` keyword is used to explicitly throw an exception. It is often used when the program encounters an exceptional situation that cannot be handled by the program

itself.

## Example:

```
public class ThrowExample {
    public static void main(String[] args) {
        int age = 15;
        if (age < 18) {
            throw new ArithmeticException("Age must be greater than or equal to 18.");
        }
        System.out.println("You are eligible to vote.");
    }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Age must be greater than or
equal to 18.
    at ThrowExample.main(ThrowExample.java:6)
```

In this example, we explicitly throw an `ArithmeticException` when the age is less than 18.

## Java Exception Propagation:

Java allows exceptions to be propagated from a method to its caller method. If a method does not handle an exception, it can declare the exception using the `throws` keyword, and the caller method can handle it.

## Example:

```
public class ExceptionPropagationExample {
    public static void main(String[] args) {
        try {
            method1();
        } catch (NullPointerException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }

    public static void method1() throws NullPointerException {
        method2();
    }

    public static void method2() throws NullPointerException {
        String str = null;
    }
}
```

```

        System.out.println(str.length());
    }
}

```

Output:

```
Exception caught: null
```

In this example, the `method2()` throws a `NullPointerException`, which is propagated to `method1()`, and finally caught in the `main()` method.

## Java Throws Keyword:

The `throws` keyword is used in a method signature to indicate that the method may throw one or more exceptions. It allows the caller method to handle the declared exceptions or propagate them further.

### Example:

```

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

    public static void method() throws Exception {
        throw new Exception("This is an exception.");
    }
}

```

Output:

```
Exception caught: This is an exception.
```

In this example, the `method()` throws an `Exception`, which is handled in the `main()` method using a catch block.

## Java Throw vs. Throws:

- `throw` is used to explicitly throw an exception within a method.
- `throws` is used in a method signature to indicate that the method may throw one or more exceptions.

## Final vs. Finally vs. Finalize:

- `final`: It is a keyword used to declare a constant variable, a method that cannot be overridden, or a class that cannot be subclassed.
- `finally`: It is a block used in exception handling to specify code that should be executed regardless of whether an exception occurs or not.
- `finalize`: It is a method in the `Object` class that is called by the garbage collector before reclaiming the memory occupied by an object.

## Exception Handling with Method Overriding:

When a subclass overrides a method from its superclass, it can choose to throw the same exception as the superclass method or a subclass of that exception.

### Example:

```
class Parent {
    void method() throws Exception {
        System.out.println("Parent's method");
    }
}

class Child extends Parent {
    @Override
    void method() throws ArithmeticException {
        System.out.println("Child's method");
    }
}

public class ExceptionHandlingWithMethodOverriding {
    public static void main(String[] args) {
        Parent parent = new Child();
        try {
            parent.method();
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

Output:

Child's method

In this example, the `Child` class overrides the `method()` from its `Parent` class. The `Child` class chooses to throw an `ArithmeticException` in the overridden method. When the method is called through the reference of the `Parent` class but pointing to a `Child` object, the `Child` class's method is executed.

## Java Custom Exceptions

In Java, you can create custom exceptions by extending the built-in `Exception` or `RuntimeException` classes. Custom exceptions allow you to handle specific exceptional situations in your application and provide meaningful error messages for better debugging and error reporting.

Here's an example of creating and using a custom exception:

```
// Custom exception class
class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}

// Class with a method that throws the custom exception
class ExampleClass {
    public void performOperation(int num) throws CustomException {
        if (num < 0) {
            throw new CustomException("Number cannot be negative.");
        }
        // Perform some operation here
        System.out.println("Operation successful.");
    }
}

// Main class to demonstrate using the custom exception
public class CustomExceptionExample {
    public static void main(String[] args) {
        ExampleClass example = new ExampleClass();
        try {
            example.performOperation(-5); // This will throw CustomException
            example.performOperation(10); // This will execute successfully
        } catch (CustomException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

Output:

```
Exception caught: Number cannot be negative.  
Operation successful.
```

In this example, we create a custom exception class `CustomException`, which extends the `Exception` class. The `CustomException` class has a constructor that takes a message as an argument and passes it to the `super()` constructor of the `Exception` class.

The `ExampleClass` has a method `performOperation()` that throws the `CustomException` if the input number is negative. We handle the exception in the `main()` method using a try-catch block.

By creating a custom exception, you can define your application-specific exception handling logic and provide more descriptive error messages to users or developers for better understanding of the issues that occur during the program's execution.

## Java Inner Classes

In Java, an inner class is a class defined within another class. Inner classes provide a way to logically group classes that belong together, making the code more readable and maintainable. There are several types of inner classes in Java: Member Inner class, Anonymous Inner class, Local Inner class, Static Nested class, and Nested Interface.

### Member Inner Class:

A Member Inner class is defined as a non-static class inside another class. It has access to all members of the enclosing class, including private members. To create an instance of a Member Inner class, it must be associated with an instance of the enclosing class.

#### Example:

```
class Outer {  
    private int outerField = 10;  
  
    // Member Inner class  
    class Inner {  
        private int innerField = 20;  
  
        void display() {  
            System.out.println("Outer Field: " + outerField);  
            System.out.println("Inner Field: " + innerField);  
        }  
    }  
}
```

```

    }
}

public class MemberInnerClassExample {
    public static void main(String[] args) {
        Outer outerObj = new Outer();
        Outer.Inner innerObj = outerObj.new Inner();
        innerObj.display();
    }
}

```

Output:

```

Outer Field: 10
Inner Field: 20

```

In this example, we have an outer class `Outer` with a member inner class `Inner`. We create an instance of the outer class and then use it to create an instance of the inner class. The inner class can access both the `outerField` and `innerField` variables.

## Anonymous Inner Class:

An Anonymous Inner class is a type of inner class that does not have a name. It is declared and instantiated at the same time. It is used when you need to create a class instance only once.

### Example:

```

interface Greeting {
    void greet();
}

public class AnonymousInnerClassExample {
    public static void main(String[] args) {
        Greeting greeting = new Greeting() {
            public void greet() {
                System.out.println("Hello, Anonymous Inner Class!");
            }
        };
        greeting.greet();
    }
}

```

Output:

```
Hello, Anonymous Inner Class!
```

In this example, we define an interface `Greeting` with a single method `greet()`. We then create an instance of the interface using an anonymous inner class and override the `greet()` method.

## Local Inner Class:

A Local Inner class is defined inside a block of code, typically within a method. It is only accessible within the block where it is declared.

### Example:

```
public class LocalInnerClassExample {
    private int outerField = 100;

    void display() {
        // Local Inner class
        class Inner {
            void innerMethod() {
                System.out.println("Outer Field from Inner: " + outerField);
            }
        }

        Inner innerObj = new Inner();
        innerObj.innerMethod();
    }

    public static void main(String[] args) {
        LocalInnerClassExample outerObj = new LocalInnerClassExample();
        outerObj.display();
    }
}
```

Output:

```
Outer Field from Inner: 100
```

In this example, we have an outer class `LocalInnerClassExample` with a method `display()`. Inside the `display()` method, we define a local inner class `Inner`. The `innerMethod()` of the inner class accesses the `outerField` of the enclosing class.

## Static Nested Class:

A Static Nested class is a static class defined inside another class. It is similar to a regular class but is just nested for packaging convenience. A static nested class does not have access to the non-static members of the enclosing class.

## Example:

```
class Outer {
    private static int outerStaticField = 50;

    // Static Nested class
    static class StaticNested {
        void display() {
            System.out.println("Outer Static Field from Static Nested: " + outerStaticField);
        }
    }
}

public class StaticNestedClassExample {
    public static void main(String[] args) {
        Outer.StaticNested nestedObj = new Outer.StaticNested();
        nestedObj.display();
    }
}
```

Output:

```
Outer Static Field from Static Nested: 50
```

In this example, we have an outer class `Outer` with a static nested class `StaticNested`. We access the `outerStaticField` from the static nested class.

## Nested Interface:

A Nested Interface is an interface declared within another class or interface. It is implicitly static and accessible within the enclosing class or interface. It is used for logical grouping of related interfaces.

## Example:

```
class Outer {
    interface NestedInterface {
        void display();
    }
}
```

```

public class NestedInterfaceExample {
    public static void main(String[] args) {
        Outer.NestedInterface nestedObj = new Outer.NestedInterface() {
            public void display() {
                System.out.println("Hello, Nested Interface!");
            }
        };
        nestedObj.display();
    }
}

```

Output:

```

Hello, Nested Interface!

```

In this example, we have an outer class `Outer` with a nested interface `NestedInterface`. We create an instance of the nested interface using an anonymous inner class and provide an implementation for the `display()` method.

## Java Multithreading

1. **Multithreading:** Multithreading is a programming concept that allows multiple threads to execute independently within a single Java program. It enables efficient utilization of CPU resources and helps in building responsive and concurrent applications.
2. **Life Cycle of a Thread:** The life cycle of a thread in Java consists of several states:
  - **New:** The thread is created, but not yet started.
  - **Runnable:** The thread is ready to run and waiting for CPU time.
  - **Running:** The thread is currently executing its task.
  - **Blocked/Waiting:** The thread is waiting for a specific condition to be satisfied or for another thread to release a lock.
  - **Terminated:** The thread has finished its execution.
3. **How to Create a Thread:**
  - By extending the Thread class:

```
class MyThread extends Thread {
    public void run() {
        // The code to be executed by the thread.
    }
}
MyThread thread = new MyThread();
thread.start(); // This will start the thread's execution.
```

- By implementing the Runnable interface:

```
class MyRunnable implements Runnable {
    public void run() {
        // The code to be executed by the thread.
    }
}
Thread thread = new Thread(new MyRunnable());
thread.start(); // This will start the thread's execution.
```

4. Thread Scheduler: The thread scheduler is responsible for assigning CPU time to threads based on their priorities and states.
5. Sleeping a Thread: The `sleep()` method is used to make a thread pause its execution for a specified amount of time.

```
try {
    Thread.sleep(1000); // Sleep for 1 second.
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

6. Starting a Thread Twice: It's not possible to start a thread twice. Once a thread has been started and executed or terminated, it cannot be started again. If you try to start a thread again, it will throw an `IllegalThreadStateException`.
7. Calling the `run()` Method: You should never directly call the `run()` method to start a new thread. Instead, use the `start()` method. Calling `run()` directly will run the code in the current thread, not in a new thread.
8. Joining a Thread: The `join()` method allows one thread to wait for the completion of another thread.

```
Thread thread = new Thread(() -> {
    // Some lengthy task
});
```

```
thread.start();
thread.join(); // The current thread waits until 'thread' completes its task.
```

9. Naming a Thread: You can set a name for a thread using the `setName()` method. This helps in identifying threads easily during debugging.

```
Thread thread = new Thread(() -> {
    // Thread task
});
thread.setName("MyThread");
```

10. Thread Priority: Threads can have different priorities, such as `MIN_PRIORITY`, `NORM_PRIORITY`, and `MAX_PRIORITY`. Higher priority threads get more CPU time.

```
Thread thread = new Thread(() -> {
    // Thread task
});
thread.setPriority(Thread.MAX_PRIORITY);
```

11. Daemon Thread: A daemon thread is a thread that runs in the background and does not prevent the JVM from exiting when the main thread finishes. You can set a thread as a daemon thread using `setDaemon(true)`.

```
Thread daemonThread = new Thread(() -> {
    // Daemon thread task
});
daemonThread.setDaemon(true);
daemonThread.start();
```

12. Thread Pool: A thread pool is a collection of pre-initialized reusable threads managed by an executor. It helps in managing the number of threads in an application to avoid the overhead of creating and destroying threads frequently.
13. Thread Group: A thread group is a way to group multiple threads together for better management. However, it's not commonly used, and modern Java applications tend to use thread pools instead.
14. Shutdown Hook: A shutdown hook is a piece of code that is executed when the JVM is about to shut down. It's useful for performing cleanup tasks before the application exits.

15. Performing Multiple Tasks: You can perform multiple tasks using multiple threads in a concurrent manner. For example, you can use a thread pool to execute multiple tasks simultaneously.
16. Garbage Collection: Garbage collection is a process in Java where the JVM automatically reclaims memory occupied by objects that are no longer in use. It helps in managing memory efficiently.
17. Runtime Class: The `Runtime` class in Java provides access to the Java runtime system, and it allows you to interact with the environment in which the application is running. It also provides methods for executing system commands and managing the JVM.

Here's a code example demonstrating the creation and execution of threads, setting thread name, priority, and using sleep:

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(getName() + ": Count " + i);
            try {
                Thread.sleep(1000); // Sleep for 1 second
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread thread1 = new MyThread();
        MyThread thread2 = new MyThread();

        thread1.setName("Thread-1");
        thread2.setName("Thread-2");

        thread1.setPriority(Thread.MAX_PRIORITY);
        thread2.setPriority(Thread.MIN_PRIORITY);

        thread1.start();
        thread2.start();
    }
}
```

In this example, two threads are created using the `MyThread` class (extending `Thread`). The threads will execute their tasks concurrently, and `Thread.sleep(1000)` will make each thread sleep for 1 second between each count. The threads are given

different names and priorities. The output will show how threads execute concurrently with different priorities.

## Java Synchronization:

1. Synchronization in Java: Synchronization is a technique used to control access to shared resources in a multi-threaded environment. When multiple threads try to access shared resources concurrently, synchronization ensures that only one thread can access the critical section (the synchronized block) at a time. This prevents race conditions and maintains data consistency.
2. Synchronized Block: In Java, you can use the `synchronized` keyword to create a synchronized block. A synchronized block allows you to specify a block of code that should be executed by only one thread at a time. The block is synchronized on a specified object, ensuring that only one thread at a time can enter that block for that particular object.

Example:

```
class SharedResource {
    public void synchronizedMethod() {
        // This synchronized method can be accessed by only one thread at a time.
        // Critical section code
    }

    public void nonSynchronizedMethod() {
        // Non-synchronized method, can be accessed by multiple threads concurrently.
    }
}
```

1. Static Synchronization: When multiple threads are trying to access a static synchronized method or block, only one thread at a time can execute the critical section for that method or block. It ensures that only one thread can access the static synchronized method or block across all instances of the class.

Example:

```
class SharedResource {
    public static synchronized void staticSyncMethod() {
        // Static synchronized method can be accessed by only one thread at a time.
        // Critical section code
    }

    public static void staticSyncBlock() {
        synchronized (SharedResource.class) {
            // Static synchronized block can be accessed by only one thread at a time.
        }
    }
}
```

```

        // Critical section code
    }
}

```

1. **Deadlock in Java:** Deadlock occurs when two or more threads are unable to proceed because each of them is waiting for a resource that is held by another thread. It leads to a circular waiting scenario, causing the threads to get stuck indefinitely. Deadlocks are undesirable and can lead to application hang-ups.

Example of Deadlock:

```

class SharedResource {
    public synchronized void method1(SharedResource resource) {
        // This method acquires a lock on this object (current instance of SharedResource).
        // It will now try to call method2 on the 'resource' object.
        resource.method2(this);
    }

    public synchronized void method2(SharedResource resource) {
        // This method acquires a lock on the 'resource' object.
        // It will now try to call method1 on the 'resource' object.
        resource.method1(this);
    }
}

```

1. **Inter-thread Communication:** Inter-thread communication allows threads to communicate with each other by waiting and notifying on a shared object. This is achieved using the `wait()`, `notify()`, and `notifyAll()` methods from the `Object` class. Threads can wait for certain conditions to be satisfied before proceeding.

Example of Inter-thread Communication:

```

class SharedResource {
    public synchronized void produce() {
        while (queueIsFull()) {
            try {
                // Wait until the queue is not full
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // Produce an item and add it to the queue
        addToQueue();
        // Notify waiting consumer threads that a new item is available in the queue
        notifyAll();
    }
}

```

```

public synchronized void consume() {
    while (queueIsEmpty()) {
        try {
            // Wait until the queue is not empty
            wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    // Consume an item from the queue
    removeFromQueue();
    // Notify waiting producer threads that space is available in the queue
    notifyAll();
}
}

```

1. Interrupting Thread: The `interrupt()` method is used to interrupt a thread in Java. When a thread is interrupted, it sets the thread's interrupt status to true. A thread can check its interrupt status using `isInterrupted()` method or by catching `InterruptedException`.

Example:

```

Thread myThread = new Thread(() -> {
    while (!Thread.currentThread().isInterrupted()) {
        // Thread's task
    }
});
myThread.start();

// Somewhere else in the code, you can interrupt the thread:
myThread.interrupt();

```

1. Reentrant Monitor: Java's synchronization is based on a concept called a "reentrant monitor." It means that a thread can acquire the lock on an object multiple times without being blocked by itself. Reentrant synchronization ensures that a thread can re-enter a synchronized block that it already holds the lock for, without deadlocking itself.

Example:

```

class SharedResource {
    public synchronized void method1() {
        System.out.println("Inside method1");
        method2();
    }
}

```

```

        public synchronized void method2() {
            System.out.println("Inside method2");
        }
    }

    public class Main {
        public static void main(String[] args) {
            SharedResource resource = new SharedResource();
            resource.method1();
        }
    }
}

```

In this example, `method1` and `method2` are synchronized methods. When `method1` is called, it acquires the lock on the `resource` object. Within `method1`, it calls `method2`, which is also synchronized. The thread can successfully enter `method2` even though it already holds the lock for the `resource` object, thanks to reentrant synchronization.

## Java I/O

Java I/O (Input/Output):

Java I/O deals with reading data from input sources and writing data to output destinations. It involves classes and methods to perform various I/O operations, such as reading and writing files, streams, and objects.

### 1. FileInputStream and FileOutputStream:

`FileInputStream` and `FileOutputStream` classes are used for reading from and writing to files, respectively.

Example - Writing to a file using `FileOutputStream`:

```

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

public class FileOutputStreamExample {
    public static void main(String[] args) {
        String content = "Hello, this is some content to write to a file!";
        try (FileOutputStream fos = new FileOutputStream("output.txt")) {
            byte[] data = content.getBytes();
            fos.write(data);
            System.out.println("Content written to the file successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

## 2. BufferedOutputStream and BufferedInputStream:

`BufferedOutputStream` and `BufferedInputStream` classes provide buffering capabilities to enhance the performance of I/O operations by reducing the number of direct system calls.

Example - Reading from a file using `BufferedInputStream`:

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class BufferedInputStreamExample {
    public static void main(String[] args) {
        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream("input.txt"))) {
            int data;
            while ((data = bis.read()) != -1) {
                System.out.print((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

## 3. SequenceInputStream:

`SequenceInputStream` is used to concatenate multiple input streams into a single stream, reading from them one by one.

Example - Reading from multiple files using `SequenceInputStream`:

```
import java.io.*;

public class SequenceInputStreamExample {
    public static void main(String[] args) {
        try (FileInputStream fis1 = new FileInputStream("file1.txt");
            FileInputStream fis2 = new FileInputStream("file2.txt");
            SequenceInputStream sis = new SequenceInputStream(fis1, fis2)) {

            int data;
            while ((data = sis.read()) != -1) {
                System.out.print((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

#### 4. ByteArrayOutputStream and ByteArrayInputStream:

`ByteArrayOutputStream` and `ByteArrayInputStream` are used for reading from and writing to byte arrays, respectively.

Example - Writing to a byte array using `ByteArrayOutputStream`:

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;

public class ByteArrayOutputStreamExample {
    public static void main(String[] args) {
        try (ByteArrayOutputStream baos = new ByteArrayOutputStream()) {
            String content = "Hello, this is some content to write to a byte array!";
            byte[] data = content.getBytes();
            baos.write(data);
            System.out.println("Content written to the byte array successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

#### 1. DataOutputStream and DataInputStream:

`DataOutputStream` and `DataInputStream` provide methods to write and read primitive data types and strings to and from streams, respectively.

Example - Writing and reading primitive data types using `DataOutputStream` and `DataInputStream`:

```
import java.io.*;

public class DataOutputStreamExample {
    public static void main(String[] args) {
        try (DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.txt"))) {
            dos.writeInt(42);
            dos.writeDouble(3.14);
            dos.writeUTF("Hello, this is a string.");
            System.out.println("Data written to the file successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }

        try (DataInputStream dis = new DataInputStream(new FileInputStream("data.txt"))) {
            int intValue = dis.readInt();
            double doubleValue = dis.readDouble();
            String stringValue = dis.readUTF();
            System.out.println("Read values: " + intValue + ", " + doubleValue + ", " + stringValue);
        } catch (IOException e) {
        }
    }
}
```

```

        e.printStackTrace();
    }
}
}

```

### 1. Java FilterOutputStream and FilterInputStream:

`FilterOutputStream` and `FilterInputStream` are abstract classes that allow you to filter or modify the data while reading from or writing to streams. The concrete implementations, such as `BufferedOutputStream`, `DataOutputStream`, and `ObjectOutputStream`, extend `FilterOutputStream`.

Example - Using `FilterOutputStream` to write to a file with additional formatting:

```

import java.io.*;

public class FilterOutputStreamExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("filtered_output.txt");
            FilterOutputStream fosFiltered = new FilterOutputStream(fos)) {
            String content = "Hello, this is some content to write with additional formatting!";
            byte[] data = content.getBytes();
            fosFiltered.write(data);
            fosFiltered.write('\n');
            System.out.println("Content written to the file with additional formatting!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

### 1. Java ObjectOutputStream and Java ObjectInputStream:

`ObjectOutputStream` and `ObjectInputStream` are used for serializing and deserializing Java objects, respectively. They allow objects to be written to streams and read back later.

Example - Writing and reading Java objects using `ObjectOutputStream` and `ObjectInputStream`:

```

import java.io.*;

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

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

```

```

        this.age = age;
    }

    public String toString() {
        return "Name: " + name + ", Age: " + age;
    }
}

public class ObjectOutputStreamExample {
    public static void main(String[] args) {
        // Writing objects to file
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("objects.dat"))) {
            Person person1 = new Person("Alice", 30);
            Person person2 = new Person("Bob", 25);

            oos.writeObject(person1);
            oos.writeObject(person2);
            System.out.println("Objects written to the file successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Reading objects from file
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("objects.dat"))) {
            Person person1 = (Person) ois.readObject();
            Person person2 = (Person) ois.readObject();

            System.out.println("Read objects:");
            System.out.println(person1);
            System.out.println(person2);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

```

Java I/O provides powerful mechanisms to handle various types of data and formats, making it a crucial part of developing applications that interact with external resources such as files, streams, and serialized objects.

### 1. Console:

The `Console` class in Java provides a simple way to interact with the console or command-line interface. It allows reading input and printing output directly to the console.

Example:

```
import java.io.Console;

public class ConsoleExample {
    public static void main(String[] args) {
        Console console = System.console();

        if (console != null) {
            String input = console.readLine("Enter your name: ");
            console.printf("Hello, %s!\n", input);
        } else {
            System.out.println("Console not available.");
        }
    }
}
```

### 1. FilePermission:

The `FilePermission` class in Java is used to represent the access rights to files and directories. It is a part of the security framework to control what resources a codebase can access.

Example:

```
import java.io.FilePermission;
import java.security.Permission;

public class FilePermissionExample {
    public static void main(String[] args) {
        String filePath = "/path/to/file.txt";
        Permission permission = new FilePermission(filePath, "read");

        if (permission.implies(new FilePermission(filePath, "read"))) {
            System.out.println("Read access granted.");
        } else {
            System.out.println("Read access denied.");
        }
    }
}
```

### 1. Writer:

The `Writer` class is an abstract base class for writing characters to an output stream. It provides various methods to write character data to a destination, like a file or a network socket.

Example:

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

```

public class WriterExample {
    public static void main(String[] args) {
        try (Writer writer = new FileWriter("output.txt")) {
            writer.write("Hello, Java!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

## 1. Reader:

The `Reader` class is an abstract base class for reading character data from an input stream. It provides various methods to read characters from a source, such as a file or a network socket.

Example:

```

import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;

public class ReaderExample {
    public static void main(String[] args) {
        try (Reader reader = new FileReader("input.txt")) {
            int data;
            while ((data = reader.read()) != -1) {
                System.out.print((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

## 1. FileWriter:

The `FileWriter` class is a subclass of `Writer` that writes characters to a file.

Example:

```

import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;

public class FileWriterExample {
    public static void main(String[] args) {
        try (Writer writer = new FileWriter("output.txt")) {
            writer.write("Hello, FileWriter!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

```

    }
}

```

## 1. FileReader:

The `FileReader` class is a subclass of `Reader` that reads characters from a file.

Example:

```

import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;

public class FileReaderExample {
    public static void main(String[] args) {
        try (Reader reader = new FileReader("input.txt")) {
            int data;
            while ((data = reader.read()) != -1) {
                System.out.print((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

## 1. BufferedWriter:

The `BufferedWriter` class is a subclass of `Writer` that adds buffering to improve performance when writing large amounts of character data.

Example:

```

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

public class BufferedWriterExample {
    public static void main(String[] args) {
        try (Writer writer = new FileWriter("output.txt");
            BufferedWriter bufferedWriter = new BufferedWriter(writer)) {
            bufferedWriter.write("Hello, BufferedWriter!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

### 1. BufferedReader:

The `BufferedReader` class is a subclass of `Reader` that adds buffering to improve performance when reading large amounts of character data.

Example:

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

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

### 1. CharArrayReader:

The `CharArrayReader` class is a subclass of `Reader` that reads characters from an array of characters.

Example:

```
import java.io.CharArrayReader;
import java.io.IOException;
import java.io.Reader;

public class CharArrayReaderExample {
    public static void main(String[] args) {
        char[] data = "Hello, CharArrayReader!".toCharArray();
        try (Reader reader = new CharArrayReader(data)) {
            int ch;
            while ((ch = reader.read()) != -1) {
                System.out.print((char) ch);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

### 1. CharArrayWriter:

The `CharArrayWriter` class is a subclass of `Writer` that writes characters to an internal character array.

Example:

```
import java.io.CharArrayWriter;
import java.io.IOException;
import java.io.Writer;

public class CharArrayWriterExample {
    public static void main(String[] args) {
        try (Writer writer = new CharArrayWriter()) {
            writer.write("Hello, CharArrayWriter!");
            char[] data = ((CharArrayWriter) writer).toCharArray();
            System.out.println(data);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

### 1. PrintStream:

The `PrintStream` class provides convenient methods to print various data types to an output stream, such as the console or a file.

Example:

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintStream;

public class PrintStreamExample {
    public static void main(String[] args) {
        try (PrintStream printStream = new PrintStream(new FileOutputStream("output.txt"))) {
            printStream.println("Hello, PrintStream!");
            printStream.printf("Value of pi: %.2f%n", Math.PI);
            printStream.println("Bye!");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

### 1. PrintWriter:

The `PrintWriter` class is a subclass of `Writer` that provides convenient methods to write various data types to an output stream.

Example:

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintWriter;

public class PrintWriterExample {
    public static void main(String[] args) {
        try (PrintWriter printWriter = new PrintWriter(new FileOutputStream("output.txt"))) {
            printWriter.println("Hello, PrintWriter!");
            printWriter.printf("Value of pi: %.2f%n", Math.PI);
            printWriter

.println("Bye!");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

#### 1. OutputStreamWriter:

The `OutputStreamWriter` class is a subclass of `Writer` that bridges character streams to byte streams. It converts character data into bytes using a specified character encoding.

Example:

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.Writer;

public class OutputStreamWriterExample {
    public static void main(String[] args) {
        try (OutputStream outputStream = new FileOutputStream("output.txt");
            Writer writer = new OutputStreamWriter(outputStream, "UTF-8")) {
            writer.write("Hello, OutputStreamWriter!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

#### 1. InputStreamReader:

The `InputStreamReader` class is a subclass of `Reader` that bridges byte streams to character streams. It converts bytes from a byte stream into characters using a specified character encoding.

Example:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.Reader;

public class InputStreamReaderExample {
    public static void main(String[] args) {
        try (InputStream inputStream = new FileInputStream("input.txt");
            Reader reader = new InputStreamReader(inputStream, "UTF-8")) {
            int data;
            while ((data = reader.read()) != -1) {
                System.out.print((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

### 1. PushbackInputStream:

The `PushbackInputStream` class is a subclass of `FilterInputStream` that allows pushing back bytes into the stream, enabling one-byte "unreading."

Example:

```
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.PushbackInputStream;

public class PushbackInputStreamExample {
    public static void main(String[] args) {
        byte[] data = "Hello, PushbackInputStream!".getBytes();
        try (InputStream inputStream = new ByteArrayInputStream(data);
            PushbackInputStream pushbackInputStream = new PushbackInputStream(inputStream)) {

            int dataByte;
            while ((dataByte = pushbackInputStream.read()) != -1) {
                if (dataByte == ',') {
                    pushbackInputStream.unread('!');
                }
                System.out.print((char) dataByte);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

## 1. PushbackReader:

The `PushbackReader` class is a subclass of `FilterReader` that allows pushing back characters into the stream, enabling one-character "unreading."

Example:

```
import java.io.CharArrayReader;
import java.io.IOException;
import java.io.PushbackReader;
import java.io.Reader;

public class PushbackReaderExample {
    public static void main(String[] args) {
        char[] data = "Hello, PushbackReader!".toCharArray();
        try (Reader reader = new CharArrayReader(data);
            PushbackReader pushbackReader = new PushbackReader(reader)) {

            int dataChar;
            while ((dataChar = pushbackReader.read()) != -1) {
                if (dataChar == ',') {
                    pushbackReader.unread('!');
                }
                System.out.print((char) dataChar);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

## 1. StringWriter:

The `StringWriter` class is a subclass of `Writer` that writes character data to an internal string buffer.

Example:

```
import java.io.IOException;
import java.io.StringWriter;
import java.io.Writer;

public class StringWriterExample {
    public static void main(String[] args) {
        try (Writer writer = new StringWriter()) {
            writer.write("Hello, StringWriter!");
            String data = writer.toString();
            System.out.println(data);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
}  
}
```

### 1. StringReader:

The `StringReader` class is a subclass of `Reader` that reads characters from a string.

Example:

```
import java.io.IOException;  
import java.io.Reader;  
import java.io.StringReader;  
  
public class StringReaderExample {  
    public static void main(String[] args) {  
        String data = "Hello, StringReader!";  
        try (Reader reader = new StringReader(data)) {  
            int ch;  
            while ((ch = reader.read()) != -1) {  
                System.out.print((char) ch);  
            }  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

### 1. PipedWriter:

The `PipedWriter` class is used to write data to a `PipedReader`, which can be read by another thread.

Example:

```
import java.io.IOException;  
import java.io.PipedReader;  
import java.io.PipedWriter;  
import java.io.Reader;  
import java.io.Writer;  
  
public class PipedWriterExample {  
    public static void main(String[] args) {  
        try (PipedWriter pipedWriter = new PipedWriter();  
            PipedReader pipedReader = new PipedReader(pipedWriter)) {  
  
            // Start a separate thread to read from the pipedReader  
            new Thread(() -> {  
                try (Reader reader = pipedReader) {  
                    int data;  
                    while ((data = reader.read()) != -1) {  
                        System.out.print((char) data);  
                    }  
                }  
            }).start();  
        }  
    }  
}
```

```

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

    pipedWriter.write("Hello, PipedWriter!");
    pipedWriter.close();
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

## 1. PipedReader:

The `PipedReader` class is used to read data from a `PipedWriter`, which is written by another thread.

Example:

```

import java.io.IOException;
import java.io.PipedReader;
import java.io.PipedWriter;
import java.io.Reader;
import java.io.Writer;

public class PipedReaderExample {
    public static void main(String[] args) {
        try (PipedWriter pipedWriter = new PipedWriter();
            PipedReader pipedReader = new PipedReader(pipedWriter)) {

            // Start a separate thread to write to the pipedWriter
            new Thread(() -> {
                try (Writer writer = pipedWriter) {
                    writer.write("Hello, PipedReader!");
                    writer.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }).start();

            int data;
            while ((data = pipedReader.read()) != -1) {
                System.out.print((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

## 1. FilterWriter:

The `FilterWriter` class is an abstract subclass of `Writer` that provides a base class for filtering character output streams.

Example:

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

public class FilterWriterExample {
    public static void main(String[] args) {
        try (Writer writer = new FileWriter("output.txt");
            FilterWriter filterWriter = new FilterWriter(writer) {
                @Override
                public void write(String str, int off, int len) throws IOException {
                    // Filter some characters before writing to the underlying writer
                    String filteredStr = str.replace("
Java", "Java SE");
                    super.write(filteredStr, off, len);
                }
            }) {

            filterWriter.write("Hello, Java! This is Java Programming.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

## 1. FilterReader:

The `FilterReader` class is an abstract subclass of `Reader` that provides a base class for filtering character input streams.

Example:

```
import java.io.CharArrayReader;
import java.io.FilterReader;
import java.io.IOException;
import java.io.Reader;

public class FilterReaderExample {
    public static void main(String[] args) {
        char[] data = "Hello, Java! This is Java Programming.".toCharArray();
        try (Reader reader = new CharArrayReader(data);
            FilterReader filterReader = new FilterReader(reader) {
                @Override
                public int read(char[] cbuf, int off, int len) throws IOException {
                    int bytesRead = super.read(cbuf, off, len);
                }
            }) {

        }
    }
}
```

```

        if (bytesRead != -1) {
            // Filter some characters before returning to the caller
            for (int i = off; i < off + bytesRead; i++) {
                if (cbuf[i] == '!') {
                    cbuf[i] = '.';
                }
            }
        }
        return bytesRead;
    }
}

char[] buffer = new char[16];
int bytesRead;
while ((bytesRead = filterReader.read(buffer)) != -1) {
    System.out.print(new String(buffer, 0, bytesRead));
}
} catch (IOException e) {
    e.printStackTrace();
}
}
}
}

```

## 1. File:

The `File` class represents a file or directory pathname in the filesystem. It can be used to manipulate files and directories, create new ones, and retrieve file information.

Example:

```

import java.io.File;

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

        // Check if the file exists
        if (file.exists()) {
            System.out.println("File exists.");
        } else {
            System.out.println("File does not exist.");
        }

        // Get the absolute path of the file
        String absolutePath = file.getAbsolutePath();
        System.out.println("Absolute path: " + absolutePath);

        // Check if it's a file or directory
        if (file.isFile()) {
            System.out.println("It's a file.");
        } else if (file.isDirectory()) {
            System.out.println("It's a directory.");
        }
    }
}

```

```

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

```

### 1. FileDescriptor:

The `FileDescriptor` class represents an opaque handle to an open file, device, or socket. It is used to manipulate low-level file operations.

Example:

```

import java.io.FileDescriptor;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileDescriptorExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("output.txt");
            FileInputStream fis = new FileInputStream(FileDescriptor.in)) {

            // Use FileDescriptor to access System.in directly
            byte[] buffer = new byte[10];
            int bytesRead = fis.read(buffer);
            System.out.println("Read from System.in: " + new String(buffer, 0, bytesRead));

            // Use FileDescriptor to duplicate file descriptor
            FileDescriptor fd = fos.getFD();
            FileOutputStream duplicateFos = new FileOutputStream(fd);
            duplicateFos.write("Hello, FileDescriptor!".getBytes());
            duplicateFos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

### 1. RandomAccessFile:

The `RandomAccessFile` class allows random access to files. It provides methods to read and write at a specific position within the file.

Example:

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomAccessFileExample {
    public static void main(String[] args) {
        try (RandomAccessFile file = new RandomAccessFile("data.txt", "rw")) {
            // Write data at a specific position
            file.write("Hello, RandomAccessFile!".getBytes());

            // Seek to a specific position
            file.seek(10);

            // Read data from the current position
            byte[] buffer = new byte[20];
            file.read(buffer);
            System.out.println(new String(buffer).trim());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

### 1. java.util.Scanner:

The `Scanner` class is used to parse primitive types and strings from a stream. It provides methods to read input from various sources like files, strings, and the console.

Example:

```
import java.util.Scanner;

public class ScannerExample {
    public static void main(String[] args) {
        try (Scanner scanner = new Scanner(System.in)) {
            System.out.print("Enter your name: ");
            String name = scanner.nextLine();
            System.out.println("Hello, " + name + "!");

            System.out.print("Enter your age: ");
            int age = scanner.nextInt();
            System.out.println("You are " + age + " years old.");
        }
    }
}
```

These examples cover a wide range of I/O classes in Java, including reading and writing to files, parsing input, filtering streams, and bridging between character and byte streams. Remember that proper exception handling and resource management

(using try-with-resources) are essential for real-world applications to ensure resources are correctly released.

## Java Networking:

Java Networking provides classes and APIs to enable network communication in Java applications. It allows Java programs to interact with remote systems, access resources over the internet, and perform various networking tasks.

### Networking Concepts:

1. **IP Address:** An IP address is a unique numerical identifier assigned to each device connected to a computer network. It allows devices to locate and communicate with each other over the network.
2. **Port:** A port is a virtual endpoint that allows different applications on a single device to communicate with each other over the network. Ports are identified by a 16-bit number, and each application has a unique port number associated with it.
3. **Socket:** A socket is an endpoint for communication between two machines over a network. Sockets enable bidirectional communication by creating a communication channel between the client and server.

### Socket Programming:

Socket programming in Java allows applications to communicate over the network using sockets. There are two types of sockets:

1. **ServerSocket:** It listens for incoming requests from clients and creates a new socket for each client to handle the communication.
2. **Socket:** It represents the endpoint of a connection to communicate with the server.

### Example of ServerSocket and Socket:

Server (ServerSocket):

```
import java.io.*;
import java.net.*;

public class ServerExample {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(8080)) {
            System.out.println("Server started. Listening on port 8080...");
            while (true) {
                Socket socket = serverSocket.accept();
            }
        }
    }
}
```

```

        System.out.println("Client connected: " + socket.getInetAddress());

        // Handle client communication using the socket
        // ...
    }
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

### Client (Socket):

```

import java.io.*;
import java.net.*;

public class ClientExample {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 8080)) {
            System.out.println("Connected to server.");

            // Communicate with the server using the socket
            // ...
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

### URL class:

The `URL` class in Java represents a Uniform Resource Locator, which is a reference to a web resource on the internet. It provides methods to parse, access, and manipulate various components of a URL, such as the protocol, host, port, path, query, and fragment.

### Example:

```

import java.net.*;

public class URLExample {
    public static void main(String[] args) throws MalformedURLException {
        String urlString = "<https://www.example.com:8080/path?param=value#fragment>";
        URL url = new URL(urlString);

        System.out.println("Protocol: " + url.getProtocol());
        System.out.println("Host: " + url.getHost());
        System.out.println("Port: " + url.getPort());
        System.out.println("Path: " + url.getPath());
        System.out.println("Query: " + url.getQuery());
        System.out.println("Fragment: " + url.getRef());
    }
}

```

```
}  
}
```

### URLConnection class:

The `URLConnection` class is an abstract class that represents a connection to a URL. It provides methods to establish a connection, retrieve data from the URL, and manage headers and cookies.

### Example:

```
import java.io.*;  
import java.net.*;  
  
public class URLConnectionExample {  
    public static void main(String[] args) throws IOException {  
        URL url = new URL("<https://www.example.com>");  
        URLConnection connection = url.openConnection();  
  
        // Get input stream to read data from the URL  
        try (InputStream inputStream = connection.getInputStream();  
            BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream))) {  
            String line;  
            while ((line = reader.readLine()) != null) {  
                System.out.println(line);  
            }  
        }  
    }  
}
```

### HttpURLConnection:

`HttpURLConnection` is a subclass of `URLConnection` specifically designed for handling HTTP requests and responses. It provides methods to set request properties, send data to the server, and read responses from the server.

### Example:

```
import java.io.*;  
import java.net.*;  
  
public class HttpURLConnectionExample {  
    public static void main(String[] args) throws IOException {  
        URL url = new URL("<https://jsonplaceholder.typicode.com/posts>");  
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();  
  
        connection.setRequestMethod("GET");  
        connection.setRequestProperty("Content-Type", "application/json");  
    }  
}
```

```

        int responseCode = connection.getResponseCode();
        System.out.println("Response Code: " + responseCode);

        // Read response data
        try (InputStream inputStream = connection.getInputStream();
             BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        }
    }
}

```

### InetAddress class:

The `InetAddress` class represents an IP address or a hostname. It provides methods to get the IP address of a given host and vice versa.

#### Example:

```

import java.net.*;

public class InetAddressExample {
    public static void main(String[] args) throws UnknownHostException {
        String host = "www.example.com";
        InetAddress inetAddress = InetAddress.getByName(host);

        System.out.println("Host Name: " + inetAddress.getHostName());
        System.out.println("Host Address: " + inetAddress.getHostAddress());
    }
}

```

### DatagramSocket class:

The `DatagramSocket` class represents a UDP (User Datagram Protocol) socket for sending and receiving datagrams. It provides methods to send and receive packets of data without establishing a connection with the recipient.

#### Example:

```

import java.io.*;
import java.net.*;

public class DatagramSocketExample {
    public static void main(String[] args) throws IOException {
        try (DatagramSocket socket = new DatagramSocket()) {
            String message = "Hello, DatagramSocket!";
            InetAddress address = InetAddress.getByName("localhost");
            int port = 8888;

```

```

        byte[] data = message.getBytes();
        DatagramPacket packet = new DatagramPacket(data, data.length, address, port);

        // Sending the packet
        socket.send(packet);
        System.out.println("Packet sent.");

        // Receiving the packet
        byte[] buffer = new byte[1024];
        packet = new DatagramPacket(buffer, buffer.length);
        socket.receive(packet);

        String receivedMessage = new String(packet.getData(), 0, packet.getLength());
        System.out.println("Received: " + receivedMessage);
    }
}

```

These are some of the fundamental concepts and classes in Java Networking. By utilizing these classes, Java applications can effectively communicate over the network, retrieve resources from remote servers, and perform various networking tasks.

## Java Collections Framework

The Java Collections Framework is a set of classes and interfaces that provide implementations of common data structures to store, manage, and manipulate collections of objects in Java. It is a fundamental part of the Java Standard Library and plays a crucial role in Java programming, providing developers with a variety of data structures and algorithms for handling collections of data efficiently.

### Hierarchy of Collection Framework

The Collections Framework has a hierarchy of interfaces and classes:

1. **Collection Interface:** The root of the collection hierarchy. It defines the most general methods applicable to all collections, such as `add`, `remove`, `contains`, etc.
  - **List Interface:** Ordered collections that can contain duplicates.
    - **ArrayList, LinkedList, etc.:** Concrete implementations.
  - **Set Interface:** Collections that cannot contain duplicate elements.

- **HashSet, TreeSet, etc.:** Concrete implementations.
- **Queue Interface:** Collections that hold elements prior to processing.
  - **LinkedList, PriorityQueue, etc.:** Concrete implementations.
- 2. **Map Interface:** It's not a child of Collection but also part of Collections Framework. Maps store key-value pairs.
  - **HashMap, TreeMap, etc.:** Concrete implementations.

## Collection Interface

It provides the standard functionality that every collection will have, like adding and removing elements.

```
import java.util.*;

public class CollectionExample {
    public static void main(String[] args) {
        Collection<String> collection = new ArrayList<>();
        collection.add("Apple");
        collection.add("Banana");
        collection.remove("Banana");
        System.out.println(collection); // Output: [Apple]
    }
}
```

## Iterator Interface

The Iterator interface provides a way to iterate through the collection, using methods like `next()`, `hasNext()`, and `remove()`.

Here's an example of using an Iterator to loop through a collection:

```
import java.util.*;

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

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

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

```
}  
}
```

In this example, the `iterator()` method returns an Iterator, and you can then use `hasNext()` to check if there are more elements, and `next()` to get the next element.

Overall, the Java Collections Framework offers a wide range of functionalities for dealing with collections of objects, whether you need lists, sets, maps, or other structures. It provides a unified architecture for representing and manipulating collections, enabling them to be manipulated independently of the details of their representation.

## Collection Framework

The Collection Framework is the foundation of the Java Collections Framework and consists of several interfaces and classes. It organizes collections into four major interfaces: List, Set, Queue, and Map.

1. **List:** An ordered collection that allows duplicate elements. It is implemented by classes like `ArrayList`, `LinkedList`, etc.
2. **Set:** A collection that does not allow duplicate elements. It is implemented by classes like `HashSet`, `LinkedHashSet`, and `TreeSet`.
3. **Queue:** A collection designed for holding elements prior to processing. It typically follows the first-in-first-out (FIFO) order. The `Queue` interface is implemented by classes like `PriorityQueue`, `LinkedList`, etc.
4. **Map:** A collection that stores key-value pairs and does not allow duplicate keys. It is implemented by classes like `HashMap`, `LinkedHashMap`, and `TreeMap`.

## Java ArrayList

`ArrayList` is a dynamic array implementation of the `List` interface. It allows you to store elements in an ordered manner with dynamic resizing. When elements are added to an `ArrayList`, it automatically grows to accommodate more elements.

Example of creating an `ArrayList`:

```
import java.util.ArrayList;  
  
public class ArrayListExample {  
    public static void main(String[] args) {  
        ArrayList<String> names = new ArrayList<>();  
        names.add("Alice");  
    }  
}
```

```

        names.add("Bob");
        names.add("Charlie");

        System.out.println(names); // Output: [Alice, Bob, Charlie]
    }
}

```

## Java LinkedList

`LinkedList` is another implementation of the `List` interface, but it differs from `ArrayList` in terms of its underlying data structure. A `LinkedList` is composed of nodes, where each node contains the data and a reference to the next node.

Example of creating a `LinkedList`:

```

import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<String> names = new LinkedList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        System.out.println(names); // Output: [Alice, Bob, Charlie]
    }
}

```

## ArrayList vs. LinkedList

`ArrayList` and `LinkedList` are both implementations of the `List` interface, but they have different performance characteristics.

- **ArrayList:** Provides fast access and retrieval of elements using index-based operations. However, insertion and deletion in the middle of the list can be slow, as it may require shifting elements.
- **LinkedList:** Provides fast insertion and deletion in the middle of the list. However, accessing elements by index is slower compared to `ArrayList`.

Choosing between `ArrayList` and `LinkedList` depends on the specific use case and the type of operations you perform frequently.

## Java List Interface

The `List` interface in Java extends the `Collection` interface and represents an ordered collection of elements. It allows duplicates and maintains the insertion order. Some commonly used methods of the `List` interface include `add`, `get`, `remove`, `indexOf`, and `size`.

Example of using the `List` interface:

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

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

        System.out.println(fruits); // Output: [Apple, Banana, Orange]
        System.out.println(fruits.get(1)); // Output: Banana
    }
}
```

## Java HashSet

`HashSet` is an implementation of the `Set` interface that uses a hash table for storage. It does not allow duplicate elements and does not guarantee the insertion order.

Example of using `HashSet`:

```
import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> uniqueNames = new HashSet<>();
        uniqueNames.add("Alice");
        uniqueNames.add("Bob");
        uniqueNames.add("Alice"); // This duplicate will not be added

        System.out.println(uniqueNames); // Output: [Alice, Bob]
    }
}
```

## Java LinkedHashSet

`LinkedHashSet` is an implementation of the `Set` interface that maintains the insertion order of elements, like a `LinkedList`. It uses a hash table with a linked list to achieve this behavior.

Example of using `LinkedHashSet` :

```
import java.util.LinkedHashSet;

public class LinkedHashSetExample {
    public static void main(String[] args) {
        LinkedHashSet<String> orderedNames = new LinkedHashSet<>();
        orderedNames.add("Alice");
        orderedNames.add("Bob");
        orderedNames.add("Charlie");

        System.out.println(orderedNames); // Output: [Alice, Bob, Charlie]
    }
}
```

## Java TreeSet

`TreeSet` is an implementation of the `Set` interface that stores elements in a sorted order. It uses a red-black tree data structure for this purpose.

Example of using `TreeSet` :

```
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<String> sortedNames = new TreeSet<>();
        sortedNames.add("Bob");
        sortedNames.add("Charlie");
        sortedNames.add("Alice");

        System.out.println(sortedNames); // Output: [Alice, Bob, Charlie]
    }
}
```

## Queue & PriorityQueue

A `Queue` in Java is a collection that represents a waiting line. It follows the first-in-first-out (FIFO) principle. The `Queue` interface extends the `Collection` interface and adds methods for enqueue (add) and dequeue (remove) operations.

A `PriorityQueue` is a specialized implementation of the `Queue` interface that orders elements based on their natural order or a custom comparator.

Example of using `Queue` and `PriorityQueue` :

```
import java.util.Queue;
import java.util.PriorityQueue;

public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new PriorityQueue<>();
        queue.add("Alice");
        queue.add("Bob");
        queue.add("Charlie");

        System.out.println(queue); // Output: [Alice, Bob, Charlie]

        String removedElement = queue.remove();
        System.out.println("Removed: " + removedElement); // Output: Removed: Alice
    }
}
```

## Deque & ArrayDeque

A **Deque** (pronounced "deck") is a double-ended queue that allows insertion and deletion at both ends. The **Deque** interface extends the **Queue** interface and provides additional methods to support stack-like behavior (LIFO - last-in-first-out).

**ArrayDeque** is an implementation of the **Deque** interface that uses a resizable array for storage.

Example of using **Deque** and **ArrayDeque**:

```
import java.util.Deque;
import java.util.ArrayDeque;

public class DequeExample {
    public static void main(String[] args) {
        Deque<String> deque = new ArrayDeque<>();
        deque.addFirst("Alice");
        deque.add

Last("Bob");
        deque.addLast("Charlie");

        System.out.println(deque); // Output: [Alice, Bob, Charlie]

        String removedElement = deque.removeFirst();
        System.out.println("Removed: " + removedElement); // Output: Removed: Alice
    }
}
```

## Java Map Interface

The `Map` interface in Java represents a collection of key-value pairs, where each key is unique, and each key is associated with a value. It does not allow duplicate keys, but different keys can be associated with the same value. The `Map` interface does not extend the `Collection` interface, but it is an important part of the Java Collections Framework.

The `Map` interface includes methods such as `put`, `get`, `remove`, `containsKey`, and `containsValue`.

## Java HashMap

`HashMap` is one of the most commonly used implementations of the `Map` interface. It uses a hash table to store key-value pairs. The keys are hashed to provide efficient access to values. It allows `null` as both key and value, and it does not guarantee the order of the elements.

Example of using `HashMap`:

```
import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> ages = new HashMap<>();
        ages.put("Alice", 30);
        ages.put("Bob", 25);
        ages.put("Charlie", 35);

        System.out.println(ages.get("Bob")); // Output: 25
    }
}
```

## Working of HashMap

The working of a `HashMap` is based on the concept of hashing. When you put a key-value pair into a `HashMap`, the key's `hashCode()` method is used to calculate a hash value, which is then used to determine the bucket in which the key-value pair is stored.

If multiple keys have the same hash value (collision), they are stored as a linked list in the same bucket. In case the number of collisions becomes significant, the linked list is converted into a balanced tree for better performance.

When you retrieve a value using a key from a `HashMap`, it calculates the hash value for the key again and uses it to look up the corresponding bucket. If the key is found

in the bucket, it returns the associated value.

## Java LinkedHashMap

`LinkedHashMap` is an implementation of the `Map` interface that maintains the insertion order of the keys. It is similar to `HashMap`, but it also maintains a doubly-linked list to keep track of the order in which elements were added.

Example of using `LinkedHashMap` :

```
import java.util.LinkedHashMap;
import java.util.Map;

public class LinkedHashMapExample {
    public static void main(String[] args) {
        LinkedHashMap<String, Integer> ages = new LinkedHashMap<>();
        ages.put("Alice", 30);
        ages.put("Bob", 25);
        ages.put("Charlie", 35);

        System.out.println(ages); // Output: {Alice=30, Bob=25, Charlie=35}
    }
}
```

## Java TreeMap

`TreeMap` is another implementation of the `Map` interface that stores key-value pairs in a sorted order based on the natural ordering of the keys or a custom comparator. It uses a red-black tree data structure for this purpose.

Example of using `TreeMap` :

```
import java.util.TreeMap;
import java.util.Map;

public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<String, Integer> ages = new TreeMap<>();
        ages.put("Alice", 30);
        ages.put("Bob", 25);
        ages.put("Charlie", 35);

        System.out.println(ages); // Output: {Alice=30, Bob=25, Charlie=35}
    }
}
```

## Java Hashtable

`Hashtable` is an older implementation of the `Map` interface that is similar to `HashMap`. However, it is synchronized, making it thread-safe for use in concurrent environments. Despite being thread-safe, `Hashtable` has some drawbacks, such as lower performance compared to `HashMap` and the inability to use `null` as a key or value.

Example of using `Hashtable`:

```
import java.util.Hashtable;
import java.util.Map;

public class HashtableExample {
    public static void main(String[] args) {
        Hashtable<String, Integer> ages = new Hashtable<>();
        ages.put("Alice", 30);
        ages.put("Bob", 25);
        ages.put("Charlie", 35);

        System.out.println(ages); // Output: {Charlie=35, Bob=25, Alice=30}
    }
}
```

## HashMap vs. Hashtable

- **Synchronization:** `Hashtable` is synchronized, meaning it is thread-safe and can be used in concurrent environments without external synchronization. On the other hand, `HashMap` is not synchronized, which makes it more efficient in single-threaded scenarios but requires external synchronization for thread safety.
- **Null Keys and Values:** `HashMap` allows `null` keys and `null` values, whereas `Hashtable` does not allow either `null` keys or `null` values.
- **Performance:** Due to synchronization, `Hashtable` may have slightly lower performance compared to `HashMap` in single-threaded scenarios.
- **Iterating Order:** `HashMap` does not guarantee the order of elements, whereas `Hashtable` returns elements in the order they were inserted.
- **Legacy:** `Hashtable` is part of the older Java Collections Framework and was present before the introduction of the `Map` interface in Java 1.2. It is recommended to use `HashMap` over `Hashtable` in modern Java applications unless thread safety is explicitly required.

In summary, both `HashMap` and `Hashtable` are implementations of the `Map` interface, but `HashMap` is the preferred choice for most use cases in modern Java applications

due to its better performance and the ability to handle `null` keys and values. However, if you need thread safety, `Hashtable` can be used, but you should consider using the newer `ConcurrentHashMap` for better concurrency performance.

## Java EnumSet

`EnumSet` is a specialized collection class in Java that is designed to work with enums. It is part of the `java.util` package and is used to represent a set of elements of a specific enumeration type. `EnumSet` is highly efficient and optimized for enums, offering better performance than other general-purpose set implementations.

Example of using `EnumSet` :

```
import java.util.EnumSet;

enum Days {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public class EnumSetExample {
    public static void main(String[] args) {
        EnumSet<Days> workingDays = EnumSet.range(Days.MONDAY, Days.FRIDAY);
        System.out.println(workingDays); // Output: [MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY]
    }
}
```

## Java EnumMap

`EnumMap` is another specialized collection class that is designed to work with enums. It implements the `Map` interface and is used to represent a map with enum keys and associated values. Like `EnumSet`, `EnumMap` is optimized for enums, offering high performance and memory efficiency.

Example of using `EnumMap` :

```
import java.util.EnumMap;
import java.util.Map;

enum Days {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public class EnumMapExample {
    public static void main(String[] args) {
        EnumMap<Days, String> tasks = new EnumMap<>(Days.class);
        tasks.put(Days.MONDAY, "Clean the house");
    }
}
```

```

        tasks.put(Days.TUESDAY, "Buy groceries");

        System.out.println(tasks); // Output: {MONDAY=Clean the house, TUESDAY=Buy groceries}
    }
}

```

## Collections class

The `Collections` class in Java is a utility class in the `java.util` package that provides various static methods to work with collections. It contains methods for performing common operations on collections such as sorting, searching, shuffling, and creating unmodifiable collections.

Example of using `Collections` class for sorting:

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

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

        Collections.sort(fruits);

        System.out.println(fruits); // Output: [Apple, Banana, Orange]
    }
}

```

## Sorting Collections

As shown in the previous example, the `Collections` class provides a method `sort()` to sort elements in a collection. The `sort()` method uses the natural ordering of elements (if they implement the `Comparable` interface) or a custom `Comparator` to determine the sorting order.

Example of sorting with a custom `Comparator`:

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class LengthComparator implements Comparator<String> {

```

```

    public int compare(String str1, String str2) {
        return Integer.compare(str1.length(), str2.length());
    }
}

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

        Collections.sort(fruits, new LengthComparator());

        System.out.println(fruits); // Output: [Apple, Banana, Orange]
    }
}

```

## Comparable interface

The `Comparable` interface in Java is used to define the natural ordering of objects. If a class implements the `Comparable` interface, it must override the `compareTo()` method, which defines how the object should be compared with another object of the same type.

Example of implementing the `Comparable` interface:

```

class Student implements Comparable<Student> {
    private String name;
    private int age;

    // Constructors, getters, setters, and other methods

    @Override
    public int compareTo(Student other) {
        return this.name.compareTo(other.name);
    }
}

```

## Comparator interface

The `Comparator` interface in Java is used to define custom comparison logic for objects that do not implement the `Comparable` interface or when you need an alternative sorting order. A `Comparator` object can be passed to sorting methods like `Collections.sort()` to determine the sorting order.

Example of implementing the `Comparator` interface:

```
class AgeComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return Integer.compare(s1.getAge(), s2.getAge());
    }
}
```

## Comparable vs. Comparator

The main difference between `Comparable` and `Comparator` is:

- **Comparable:** It defines the natural ordering of objects within the class itself. The `compareTo()` method is implemented in the class whose objects need to be sorted.
- **Comparator:** It is a separate class that defines external comparison logic. It allows sorting objects based on different criteria without modifying the original class.

When to use `Comparable`:

- When you want to define a default or natural ordering for objects of a class.
- The class is not modifiable, or you want to define a single, consistent sorting order for the class.

When to use `Comparator`:

- When you need to define multiple sorting orders for objects of a class.
- When you want to sort objects based on different criteria, depending on the context.
- The class you are sorting is not modifiable, and you cannot implement `Comparable` in it.

In summary, `Comparable` is used when you want to define the natural ordering of objects within the class itself, while `Comparator` is used when you need to define external comparison logic or multiple sorting orders for objects of a class.

## Properties class

The `Properties` class in Java is a subclass of `Hashtable` that represents a persistent set of properties in key-value pairs. It is often used to manage configuration settings in Java applications. The keys and values in a `Properties` object are both strings.

Example of using the `Properties` class:

```
import java.util.Properties;

public class PropertiesExample {
    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.setProperty("username", "john_doe");
        properties.setProperty("password", "mysecretpassword");

        System.out.println(properties.getProperty("username")); // Output: john_doe
    }
}
```

## ArrayList vs. Vector

Both `ArrayList` and `Vector` are classes that implement the `List` interface, and they are used to store elements in an ordered manner with dynamic resizing. However, there are some differences between them:

- **Synchronization:** `Vector` is synchronized, which means it is thread-safe and can be used in concurrent environments without external synchronization. `ArrayList`, on the other hand, is not synchronized and is not suitable for concurrent use without proper synchronization.
- **Performance:** Due to synchronization, `Vector` may have slightly lower performance compared to `ArrayList` in single-threaded scenarios.
- **Legacy:** `Vector` is part of the older Java Collections Framework and was present before the introduction of the `List` interface in Java 1.2. `ArrayList` is the preferred choice in modern Java applications unless thread safety is explicitly required.

## Java Vector

`Vector` is a class in Java that is part of the legacy Java Collections Framework. It implements the `List` interface and is similar to `ArrayList`, but with the added feature of being synchronized, which makes it thread-safe.

Example of using `Vector`:

```
import java.util.Vector;

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

```

        Vector<String> names = new Vector<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        System.out.println(names); // Output: [Alice, Bob, Charlie]
    }
}

```

## Java Stack

**Stack** is a class in Java that represents a last-in-first-out (LIFO) stack of objects. It extends the **Vector** class and provides additional methods like **push()** to add elements to the stack and **pop()** to remove elements from the top of the stack.

Example of using **Stack**:

```

import java.util.Stack;

public class StackExample {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();
        stack.push("Alice");
        stack.push("Bob");
        stack.push("Charlie");

        System.out.println(stack.pop()); // Output: Charlie
    }
}

```

## Java Collection Interface

The **Collection** interface in Java is the root interface of the Java Collections Framework. It represents a group of objects known as elements and provides methods to work with collections such as adding, removing, and querying elements.

The **Collection** interface includes common methods like **add**, **remove**, **contains**, **size**, **isEmpty**, and **iterator**.

Example of using the **Collection** interface:

```

import java.util.Collection;
import java.util.ArrayList;

public class CollectionExample {
    public static void main(String[] args) {
        Collection<String> fruits = new ArrayList<>();
        fruits.add("Apple");
    }
}

```

```

        fruits.add("Banana");
        fruits.add("Orange");

        System.out.println(fruits); // Output: [Apple, Banana, Orange]
    }
}

```

## Java Iterator Interface

The `Iterator` interface in Java is used to iterate over elements in a collection. It provides methods like `next()`, `hasNext()`, and `remove()` to traverse the elements in a collection in a forward direction.

Example of using the `Iterator` interface:

```

import java.util.Iterator;
import java.util.ArrayList;

public class IteratorExample {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        Iterator<String> iterator = names.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}

```

## Java Deque Interface

The `Deque` (pronounced "deck") interface in Java stands for double-ended queue. It extends the `Queue` interface and allows insertion and removal of elements from both ends. It can be used as a stack (LIFO - last-in-first-out) or a queue (FIFO - first-in-first-out) depending on the requirements.

The `Deque` interface includes methods like `addFirst`, `addLast`, `removeFirst`, `removeLast`, `getFirst`, and `getLast`.

Example of using the `Deque` interface:

```

import java.util.Deque;
import java.util.ArrayDeque;

```

```

public class DequeExample {
    public static void main(String[] args) {
        Deque<String> deque = new ArrayDeque<>();
        deque.addFirst("Alice");
        deque.addLast("Bob");
        deque.addLast("Charlie");

        System.out.println(deque.removeFirst()); // Output: Alice
    }
}

```

## Java ConcurrentHashMap

`ConcurrentHashMap` is a class in Java that is part of the Java Concurrency Utilities. It is a highly efficient and thread-safe version of a `HashMap`. It allows multiple threads to read and write concurrently without external synchronization, while still maintaining data consistency.

Example of using `ConcurrentHashMap`:

```

import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample {
    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> ages = new ConcurrentHashMap<>();
        ages.put("Alice", 30);
        ages.put("Bob", 25);
        ages.put("Charlie", 35);

        System.out.println(ages.get("Charlie")); // Output: 35
    }
}

```

## Java ConcurrentLinkedQueue

`ConcurrentLinkedQueue` is a class in Java that is part of the Java Concurrency Utilities. It is a thread-safe implementation of the `Queue` interface, suitable for concurrent use by multiple threads.

Example of using `ConcurrentLinkedQueue`:

```

import java.util.concurrent.ConcurrentLinkedQueue;

public class ConcurrentLinkedQueueExample {
    public static void main(String[] args) {
        ConcurrentLinkedQueue<String> queue = new ConcurrentLinkedQueue<>();
        queue.add("Alice");
        queue.add("Bob");
    }
}

```

```
        queue.add("Charlie");  
  
        System.out.println(queue.poll()); // Output: Alice  
    }  
}
```

In summary, these classes and interfaces in the Java Collections Framework provide a powerful and flexible way to work with collections and handle various concurrency scenarios. Whether you need a synchronized list, a thread-safe map, or a concurrent queue, the Java Collections Framework has appropriate classes to suit your needs.