

IMP : why are you leaving your current company?

I am currently seeking a career move to broaden my skills and experience after gaining valuable insights in my current role.

Exploring new opportunities will allow me to take on fresh challenges and contribute to a different professional landscape.

IMP : Why are you leaving your current company after just one year?

I have gained valuable experience in my current role, but I am seeking a position that offers more opportunities for learning, growth, and career advancement.

I believe making this move at this stage will allow me to take on more responsibilities and make a greater impact.

IMP : Expected Salary?

Considering the skills and experience I have acquired, I am expecting a salary range between 18 to 20 LPA.

Question: "Why are you looking to leave your current company after just one year?"

"I had an enriching experience during my first three years at my previous company, where I built a strong technical foundation.

However, in my current role, which I've held for a year, I have noticed that the scope of my responsibilities has become limited.

I am now looking for a position that offers a broader range of challenges and opportunities for my career growth and develop my skills in Java Full Stack, React, and AWS."

IMP : How long will you stay with this company, and how can we trust you since you are leaving your current job after just one year?

I am looking for a long-term opportunity where I can grow and contribute meaningfully. My move is driven by career growth, not frequent job changes.

If I find the right environment with challenges and learning opportunities, I am committed to staying and adding value.

Subject: Application for Java Full Stack Developer Position

Hi,

I hope this message finds you well. I'm reaching out to express my interest in the Java Developer or Java Full Stack Developer role at your company.

With 4 years of hands-on experience in Java, Spring Boot, Angular and AWS, I'm keen on contributing my skills to your team.

I have attached my resume for your consideration.

I believe my skills and experience make me a strong candidate for the position, and I am eager to discuss how I can contribute to the success of your team.

Please let me know if additional information is required from my end.

Thank you!. I look forward to hearing from you soon.

Thanks & Regards,

Nani Babu Pallapu,

nanipallapu369@mail,

93925-90089,

96765-41438 (Alternative)

Candidate Name:

Mobile Number:

Mail id:

Total Experience:

Experience on Core Java:

Experience on Springboot:

Experience on Microservices:

Current CTC:

Expected CTC:

Notice period:

Location:

Interested in Hyderabad Location: Y/N

Thank you for the update and for the opportunity to discuss the salary for the Java Full Stack Developer role. I am comfortable with the proposed salary of 5500RM per month.

Looking forward to moving forward with the next steps.

ABOUT HTC GLOBAL SERVICE

HTC Global Services is an IT company started in 1990, based in **Troy, Michigan**. It has offices in many countries and over **6000 employees** worldwide.

The **CEO of HTC Global Services** is **Madhava Reddy**, who is also the **founder** of the company.

The company provides **software and business solutions** to industries like **insurance, healthcare, manufacturing, and retail**.

HTC is known for its **quality work, long-term client relationships**, and strong global presence.

SELF-INTRODUCTION:

Good Morning,

I'm Nani Babu Pallapu, a Java Full Stack Developer holding a Bachelor's degree in Computer Science from Nova College of Engineering & Technology, graduated in 2020.

I was born and brought up in Eluru, Andhra Pradesh.

My professional journey began as a Trainee & Contract-based Java Developer at TechEra IT Consulting, where I completed an intensive 6-month training program

before joining Innova Solutions in December 2021 as a Contract-based Java Developer. Within four months, I demonstrated hard work and dedication, leading to a transition to a permanent employee role.

I completed a total of three years at Innova Solutions, including the contract period.

Currently, I am working as a Java Full Stack Developer at HTC Global Services with around 4 years of hands-on experience.

My expertise includes Java, Spring Boot, Angular, React, Hibernate, MySQL, SQL Server, Apache NiFi, and microservices.

Additionally, I have experience with GitLab, BitBucket, Agile methodologies, CI/CD using Jenkins, Docker, and AWS services.

I have worked with GitLab CI/CD pipelines to automate build and deployment processes and have experience managing repositories for efficient version control.

I am also familiar with AWS services like EC2, S3, Lambda, and RDS for cloud-based deployments.

During my tenure, I have worked on the 3 projects:

1. ****LYNC-OMS Project**** – Developed an Order Management System (OMS) application for internal sales and order processing teams. This system streamlined the process of placing business orders, where Account Executives (AEs) could place orders on behalf of customers, which were then reviewed and approved by the Order Processing (OP) team.
2. ****NiFi Development Project**** – Led the development of customized NiFi processors for PDF/RTF to text conversion. As the sole team member, I analyzed client requirements and developed specialized processors to convert files into plain text format, storing them in a MySQL database.
3. ****Insurance Partner Portal** (HTC Global Services – B2B Team)**** – Worked on a secure, role-based Business Partner Portal that allows external partners like suppliers, medical providers, and lenders to manage their services, billing, and communication with the insurance enterprise.

These projects have helped me enhance my technical skills and provided valuable experience in problem-solving and client collaboration.

Outside of work, I am known for my positive attitude, hard work, and dedication. I approach challenges with confidence and always strive to find solutions and grow professionally.

In my spare time, I enjoy listening to music and dancing.

That concludes my introduction. Thank you for giving me the opportunity to present myself.

WHAT IS JAVA

Java is an object-oriented, class-based, concurrent, secured and general-purpose computer-programming language. It is a widely used robust technology.

Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995.

James Gosling is known as the father of Java. Before Java, its name was Oak.

Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

Platform: Any hardware or software environment in which a program runs, is known as a platform.

Since Java has a runtime environment (JRE) and API, it is called a platform.

Applications :

1. Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irctc.co.in, javatpoint.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games, etc.

Types of Java Applications

There are mainly 4 types of applications that can be created using Java programming:

1) Standalone Application

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

2) Web Application

An application that runs on the server side and creates a dynamic page is called a web application. Currently, Servlet, JSP, Struts, Spring, Hibernate, JSF, etc. technologies are used for creating web applications in Java.

3) Enterprise Application

An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like high-level security, load balancing, and clustering. In Java, EJB is used for creating enterprise applications.

4) Mobile Application

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

Java Platforms / Editions

There are 4 platforms or editions of Java:

1) Java SE (Java Standard Edition)

It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, String, Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

2) Java EE (Java Enterprise Edition)

It is an enterprise platform that is mainly used to develop web and enterprise applications. It is built on top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, JPA, etc.

3) Java ME (Java Micro Edition)

It is a micro platform that is dedicated to mobile applications.

4) JavaFX

It is used to develop rich internet applications. It uses a lightweight user interface API.

Features of Java

=====

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as Java buzzwords.

A list of the most important features of the Java language is given below.

Java Features

-
- 1.Simple
 - 2.Object-Oriented
 - 3.Portable
 - 4.Platform independent
 - 5.Secured
 - 6.Robust
 - 7.Architecture neutral
 - 8.Interpreted
 - 9.High Performance
 - 10.Multithreaded
 - 11.Distributed
 - 12.Dynamic

JVM , JDK AND JRE IN JAVA

JVM, JDK, and JRE are important components in the Java programming and runtime environment. Here's what each of them represents:

- 1. **JVM (Java Virtual Machine)**:

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist.

It's a specification that provides a runtime environment in which Java bytecode can be executed.

It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other.

However, Java is platform independent. There are three notions of the JVM: specification, implementation, and instance.

The JVM performs the following main tasks:

- i. Loads code
- ii. Verifies code
- iii. Executes code
- iv. Provides runtime environment

2. **JRE (Java Runtime Environment)**:

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment.

It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

3. **JDK (Java Development Kit)**:

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets.

It physically exists. It contains JRE + development tools.

How the JVM executes a Java program:

=====

Compile Time:

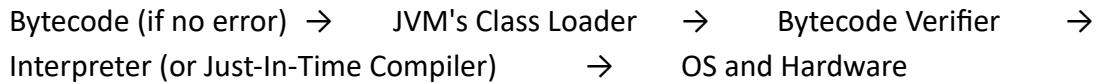
Java Code (Student.java) → Compiler → Bytecode (checks Error | No Error) → Student.class

- **Java Code (Student.java)**: This is the source code written by the programmer in Java.

- **Compiler**: The Java compiler ('javac') translates the Java source code into bytecode. During this process, the compiler checks for syntax errors and other compile-time errors. If there are no errors, it generates bytecode.

- **Bytecode (Student.class)**: Bytecode is the intermediate representation of the Java program. It's a platform-independent format that can be executed by the Java Virtual Machine (JVM).

Run-Time:



- **Bytecode (Student.class)**: If there are no compilation errors, the bytecode is loaded into memory by the JVM's class loader.

- **Class Loader**: The class loader loads the bytecode into memory and resolves references to other classes.

- **Bytecode Verifier**: The bytecode verifier ensures that the bytecode adheres to the rules of the Java language and doesn't violate security constraints. It checks for illegal bytecode that could potentially harm the JVM or the underlying system.

- **Interpreter (or Just-In-Time Compiler)**: The interpreter reads and executes bytecode instructions one by one. In some cases, the JVM may use a Just-In-Time (JIT) compiler to translate bytecode into native machine code for better performance.

- **OS and Hardware**: The native machine code generated by the interpreter or JIT compiler is executed by the operating system and the hardware, producing the desired output or performing the required operations.

This process ensures that Java programs are compiled into bytecode, which is then executed by the JVM on different platforms, making Java a platform-independent language.

JVM Memory

1. Method area: In the method area, all class level information like class name, immediate parent class name, methods and variables information etc are stored, including static variables.

There is only one method area per JVM, and it is a shared resource.

2. Heap area: Information of all objects is stored in the heap area. There is also one Heap Area per JVM. It is also a shared resource.

3. Stack area: For every thread, JVM creates one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which stores methods calls.

All local variables of that method are stored in their corresponding frame. After a thread terminates, its run-time stack will be destroyed by JVM. It is not a shared resource.

(The stack area simply stores method calls and local variables per stack frame.)

4. PC Registers: Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers.

5. Native method stacks: For every thread, a separate native stack is created. It stores native method information.

Access Modifiers:

=====

Private: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

Default: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Protected: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

Public: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

DATA TYPES :

=====

In Java, data types can be categorized into primitive and non-primitive (reference) data types.

1. **Primitive Data Types:**

- These are basic data types predefined by the language and are not objects.
- They store simple values.
- There are eight primitive data types in Java:

1. **byte**: 8 bits, with a range of -128 to 127. ex : byte myByte = 10;
2. **short**: 16 bits, with a range of -32,768 to 32,767. ex : short myShort = 1000;
3. **int**: 32 bits, with a range of -2³¹ to 2³¹ - 1. ex : int myInt = 100000;
4. **long**: 64 bits, with a range of -2⁶³ to 2⁶³ - 1. ex : long myLong = 1000000000L;
5. **float**: 32 bits, IEEE 754 floating-point representation. ex : float myFloat = 3.14f;
6. **double**: 64 bits, IEEE 754 floating-point representation. ex : double myDouble = 3.14159265359;
7. **char**: 16 bits, representing Unicode characters. ex : char myChar = 'A';
8. **boolean**: Not defined, but typically occupies 1 bit, with values true or false. ex : boolean myBoolean = true;

2. **Non-Primitive (Reference) Data Types:**

- These data types are not predefined by Java and are created by the programmer.

- They are also known as reference types because they reference objects.
- Examples include:
 - **Arrays**: Ordered collection of elements of the same or different data types.
 - **Classes**: User-defined types that can contain fields, methods, constructors, etc.
 - **Interfaces**: Similar to classes but define a set of methods that implementing classes must provide.
 - **Strings**: A sequence of characters treated as an object in Java.

Primitive types are usually faster and require less memory because they directly contain their values. Non-primitive types, being objects, have overhead associated with them (like memory allocation and garbage collection), but they offer more functionality and flexibility.

Java Variables

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

Variable

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

2) Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.

It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

Final Keyword :

=====

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable : If you make any variable as final, you cannot change the value of final variable(It will be constant).

2. method : If you make any method as final, you cannot override it.

3. class : If you make any class as final, you cannot extend it.

Static Keyword:

=====

1. ****Static Variables****: Shared among all instances, initialized once, accessed using the class name.

2. ****Static Methods****: Belong to the class, callable without instance creation, cannot access instance variables directly.

3. ****Static Blocks****: Used for static variables initialization, executed once when the class is loaded than means it is executed before main method.

4. ****Static Nested Classes****: Nested classes declared static, can access only static members of the outer class, can be instantiated without an outer class instance.

JAVA OOPS:

=====

Object-Oriented Programming is a paradigm that provides many concepts, such as inheritance, Abstraction, Encapsulation, polymorphism, etc.

Simula is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.

Smalltalk is considered the first truly object-oriented programming language.

The popular object-oriented languages are Java, C#, PHP, Python, C++, etc.

1. **Classes and Objects**:

Class :

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

It is a logical entity. It can't be physical. A class in Java can contain:

Fields

Methods

Constructors

Object :

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc.

It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

State: represents the data (value) of an object.

Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.

Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

2. ****Encapsulation**:**

Encapsulation is the concept of bundling attributes and methods that operate on that data into a single unit (a class).

Encapsulation in Java is a process of wrapping code and data together into a single unit, for example, a capsule which is mixed of several medicines.

Encapsulation in java : We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The Java Bean class is the example of a fully encapsulated class.

It helps in data hiding, where the internal details of a class are hidden from the outside and can only be accessed through well-defined interfaces (methods).

Access modifiers like `private`, `protected`, and `public` control the visibility and accessibility of class members.

It is a way to achieve data hiding in Java because other class will not be able to access the data through the private data members.

3. **Inheritance**:

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.

It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.

When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

The `extends` keyword is used in Java to create subclasses that inherit from a superclass.

The `implements` keyword is used in java to create subclass that inherit from a interface.

Uses : Runtime polymorphism can be achieved(Method Overriding) and Code Reusability.

Types i. Single Inheritance 2. Multilevel Inheritance 3. Hierarchical Inheritance

Note : To reduce the complexity and simplify the language, multiple inheritance is not supported in java. It is possible with interfaces.

NOTE : Aggregation in Java (HAS - A RELATIONSHIP)

If a class has an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, email etc.

It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc.

it is useful for code reusability.

example :

```
-----
class Employee{
    int id;
    String name;
    Address address;//Address is a class
    ...
}
```

4. **Polymorphism**:

Polymorphism in Java is a concept by which we can perform a single action in different ways

Polymorphism is achieved through method overriding and method overloading.

Method overloading(CompileTime Polymorphism) involves defining multiple methods with the same name in the same class, differing by the number or type of parameters.

Method overriding (RunTime Polymorphism) involves creating a method in a subclass with the same name as a method in the superclass, allowing the subclass to provide a specific implementation.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

5. **Abstraction**:

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java:

- i. Abstract class (0 to 100%)
- ii.Interface (100%)

****Abstract Class:****

i. ****Definition**:**

- An abstract class is a class that can have both abstract (i.e., method without a body) and concrete (i.e., method with a body) methods.

- You can define fields (variables) and constructors in an abstract class.

ii. ****Usage**:**

- Abstract classes are used to provide a common base for multiple related classes.

- They can serve as a blueprint for subclasses, allowing code reuse.

iii. ****Inheritance**:**

- An abstract class can be extended (i.e., a subclass can inherit from it).

- A subclass must implement (override) all the abstract methods of its abstract superclass unless the subclass itself is declared as abstract.

iv. ****Access Modifiers**:**

- Abstract classes can have access modifiers for their fields and methods.

v. ****Multiple Inheritance**:**

- A Java class can extend only one abstract class, so Java supports single inheritance with abstract classes.

vi. ****Constructor**:**

- Abstract classes can have constructors, and these constructors can be called by the constructors of their subclasses.

In an abstract class, you can have:

Concrete (normal) methods

Static methods

Default methods (not explicitly called "default" but just normal methods with implementations that means can not use default keyword for methods.)

****Interface:****

i. ****Definition**:**

- An interface is a completely abstract class, which means it can only contain abstract methods and constants (public static final fields).

- All methods declared in an interface are implicitly public and abstract, and all fields are implicitly public, static, and final.

ii. ****Usage**:**

- Interfaces are used to define contracts that classes must adhere to.

- They allow multiple unrelated classes to implement the same set of methods, promoting code interoperability.

iii. ****Inheritance**:**

- A class can implement multiple interfaces (i.e., a single class can adhere to multiple contracts), enabling multiple inheritance of types.

iv. ****Access Modifiers**:**

- All members (methods and constants) of an interface are implicitly public.

v. **Multiple Inheritance**:

- Java allows multiple inheritance through interfaces because a single class can implement multiple interfaces.

vi. **Constructor**:

- Interfaces do not have constructors. You cannot create an instance of an interface.

Since Java 8, interfaces can have:

Default methods (using the default keyword)

Static methods

1. Default Methods – Avoid Breaking Existing Implementations

Before Java 8, if a new method was added to an interface, all implementing classes had to implement it. Default methods allow adding new methods without breaking old classes.

Example: Adding a Logging Method to an Interface

```
```java
interface PaymentGateway {
 void processPayment(double amount);

 default void logTransaction(String details) {
 System.out.println("Transaction logged: " + details);
 }
}
```

```

}

class PayPal implements PaymentGateway {

 public void processPayment(double amount) {
 System.out.println("Processing $" + amount + " via PayPal");
 }
}

public class Main {

 public static void main(String[] args) {
 PayPal paypal = new PayPal();
 paypal.processPayment(100);
 paypal.logTransaction("Paid $100 using PayPal");
 }
}
```

```

****Key Benefits:****

- Old implementations (like `PayPal`) do not need to change.
- The new method (`logTransaction`) is available without forcing updates to existing classes.

2. Static Methods – Replacing Utility Classes

Before Java 8, utility methods were typically placed in separate utility classes. Static methods in interfaces allow such methods to be placed within the interface itself, keeping related functionality in one place.

```
##### **Before Java 8: Using a Separate Utility Class**
```

```
```java
class AuthUtils {

 public static boolean isValidEmail(String email) {
 return email.contains("@");
 }
}

class GoogleAuth {

 public void authenticate(String user) {
 if (AuthUtils.isValidEmail(user)) {
 System.out.println("Google authentication successful for "
+ user);
 } else {
 System.out.println("Invalid email format.");
 }
 }
}
```

```

Here, the validation method is placed in a separate utility class (`AuthUtils`).

```
##### **After Java 8: Using a Static Method in an Interface**
```

```
```java
interface AuthenticationService {

 void authenticate(String user);
}
```

```

```

static boolean isValidEmail(String email) {
    return email.contains("@");
}

}

class GoogleAuth implements AuthenticationService {
    public void authenticate(String user) {
        if (AuthenticationService.isValidEmail(user)) {
            System.out.println("Google authentication successful for "
+ user);
        } else {
            System.out.println("Invalid email format.");
        }
    }
}
```

```

#### **\*\*Key Benefits:\*\***

- The static method is now **\*\*inside\*\*** the interface, keeping related logic together.
- There is **\*\*no need for a separate utility class\*\*** (`AuthUtils`).
- The static method can be accessed directly using `InterfaceName.methodName()`.

#### **### \*\*Final Summary\*\***

- **Default methods** allow adding new methods to interfaces without affecting existing implementations.

- **Static methods** help replace separate utility classes by providing helper methods directly within the interface.

## METHOD OVERLOADING AND METHOD OVERRIDING IN JAVA

---

Method overloading and method overriding are two important concepts in object-oriented programming, particularly in Java. They both involve defining multiple methods with the same name, but they serve different purposes and have distinct characteristics. Let's explore each concept:

### **Method Overloading:**

Method overloading, also known as compile-time polymorphism or static polymorphism, allows you to define multiple methods in a class with the same name but different parameters. The method signature, including the method name and the number or types of its parameters, must be different for overloaded methods. The compiler determines which method to call based on the number and types of arguments passed during a method invocation.

Key points about method overloading:

1. The return type of the method doesn't play a role in method overloading; it's not sufficient to differentiate overloaded methods.

2. Overloaded methods are invoked at compile time, based on the method's signature and the arguments passed.

3. Method overloading is a way to provide multiple ways to call a method with different argument lists for convenience and flexibility.

Here's an example of method overloading in Java:

```
```java
public class Calculator {

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

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

    public String add(String str1, String str2) {
        return str1 + str2;
    }
}

```

```

In the example above, the `add` method is overloaded with different parameter types.

**\*\*Method Overriding:\*\***

Method overriding, also known as runtime polymorphism or dynamic polymorphism, allows a subclass to provide a specific implementation of a method that is already defined in its superclass.

The overridden method in the subclass has the same name, return type, and parameters as the method in the superclass.

Method overriding is used to implement the "is-a" relationship and allows you to define behavior specific to the subclass.

Key points about method overriding:

1. The return type, method name, and parameter list must be the same in the overriding method as in the overridden method.

2. The `@Override` annotation is commonly used to indicate that a method is intended to override a method in the superclass. While not strictly required, it helps catch errors at compile time.

3. Overridden methods are invoked at runtime based on the actual object type, allowing polymorphic behavior.

Here's an example of method overriding in Java:

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

```

    }

}

class Dog extends Animal {

    @Override

    void makeSound() {

        System.out.println("The dog barks.");
    }

}

public class Main {

    public static void main(String[] args) {

        Animal animal = new Dog(); // Polymorphism

        animal.makeSound(); // Calls the makeSound method of the Dog
    }

}
```

```

In this example, the `makeSound` method in the `Dog` class overrides the method with the same name in the `Animal` class.

In summary, method overloading allows multiple methods with the same name but different parameters within the same class, while method overriding allows a subclass to provide a specific implementation for a method inherited from its superclass.

Method overriding is a key feature of polymorphism in object-oriented programming.

## COMPILE-TIME AND RUNTIME POLYMORPHISM

---

Compile-time polymorphism (also known as static polymorphism) and runtime polymorphism (also known as dynamic polymorphism) are two types of polymorphism in object-oriented programming languages like Java.

They occur when different methods with the same name are invoked but are determined at different stages of the program's lifecycle.

### **\*\*Compile-Time Polymorphism (Static Polymorphism):\*\***

1. **Method Overloading:** Compile-time polymorphism is mostly achieved through method overloading. It occurs when there are multiple methods with the same name in a class, but they have different parameter lists (i.e., a different number or types of parameters).
2. **Determined at Compile Time:** The method to be called is determined by the compiler based on the number and types of arguments passed during the method call.

3. **No Need for Inheritance**: Compile-time polymorphism can occur within a single class without the need for inheritance.

4. **Examples**: Method overloading is a common example of compile-time polymorphism. The specific method to be invoked is resolved during compilation.

Here's an example of compile-time polymorphism using method overloading:

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

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

```

**Runtime Polymorphism (Dynamic Polymorphism):**

1. **Method Overriding**: Runtime polymorphism is primarily achieved through method overriding. It occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

2. **\*\*Determined at Runtime\*\***: The method to be called is determined at runtime based on the actual object type, allowing for polymorphic behavior.

3. **\*\*Involves Inheritance\*\***: Runtime polymorphism typically involves inheritance, where a subclass extends a superclass and overrides one or more of its methods.

4. **\*\*Examples\*\***: Method overriding is a common example of runtime polymorphism. The specific method to be invoked depends on the actual runtime type of the object.

Here's an example of runtime polymorphism using method overriding:

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

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

public class Main {
```

```
public static void main(String[] args) {  
    Animal animal = new Dog(); // Polymorphism  
    animal.makeSound(); // Calls the makeSound method of the Dog  
}  
}  
***
```

In this example, the method to be invoked is determined at runtime based on the actual object type ('Dog'), demonstrating runtime polymorphism.

In summary, compile-time polymorphism is resolved by the compiler based on the method's signature, whereas runtime polymorphism is determined at runtime based on the actual object type, allowing for more dynamic and flexible behavior.

UPCASTING AND DOWNCASTING IN JAVA

Object TypeCasting

A process of converting one data type to another is known as Typecasting and Upcasting and Downcasting is the type of object typecasting.

In Java, the object can also be typecasted like the datatypes. Parent and Child objects are two types of objects.

So, there are two types of typecasting possible for an object, i.e., Parent to Child and Child to Parent or can say Upcasting and Downcasting.

Upcasting:

Upcasting is a type of object typecasting in which a child object is typecasted to a parent class object. By using the Upcasting, we can easily access the variables and methods of the parent class to the child class.

Here, we don't access all the variables and the method. We access only some specified variables and methods of the child class. Upcasting is also known as Generalization and Widening.

Key points about upcasting:

1. Upcasting is an implicit or automatic type conversion that is done by the compiler. No casting operator is needed.
2. It promotes code reusability and allows you to use a more general reference to an object of a specific subclass.
3. Upcasting allows you to access only the members (fields and methods) of the superclass from the reference.

Here's an example of upcasting in Java:

```
```java

class Animal {

 void makeSound() {

 System.out.println("The animal makes a sound.");
 }
}

class Dog extends Animal {

 void makeSound() {

 System.out.println("The dog barks.");
 }

 void fetch() {

 System.out.println("The dog fetches a ball.");
 }
}

public class Main {

 public static void main(String[] args) {

 Animal myAnimal = new Dog(); // Upcasting

 myAnimal.makeSound(); // Calls the makeSound method of the
Dog class
 }
}
```

```

In this example, the `Dog` object is upcast to the `Animal` reference, and you can access the overridden `makeSound` method.

Downcasting:

Downcasting is the opposite process of upcasting. It involves casting a reference from a superclass type to a subclass type.

Downcasting allows you to access specific members of the subclass that are not present in the superclass.

However, downcasting requires an explicit cast operator and is not always safe. It can lead to a `ClassCastException` if the actual object type is not compatible with the downcast.

Key points about downcasting:

1. Downcasting requires an explicit type cast operator, and it's subject to runtime checks to ensure type safety.
2. You can access members specific to the subclass after downcasting.
3. If the object is not an instance of the target subclass, a `ClassCastException` may occur.

Here's an example of downcasting in Java:

```
```java
```

```
public class Main {
 public static void main(String[] args) {
 Animal myAnimal = new Dog();
 if (myAnimal instanceof Dog) {
 Dog myDog = (Dog) myAnimal; // Downcasting
 myDog.fetch(); // Calls the fetch method of the Dog class
 }
 }
 ...
```

In this example, downcasting is performed after checking whether the object is an instance of the `Dog` class. If the check is successful, you can safely downcast and access the `fetch` method.

In Java, we rarely use Upcasting. We use it when we need to develop a code that deals with only the parent class. Downcasting is used when we need to develop a code that accesses behaviors of the child class.

### Type casting in JAVA

---

In Java, type casting is a method or process that converts a data type into another data type in both ways manually and automatically.

The automatic conversion is done by the compiler and manual conversion performed by the programmer.

### Widening Type Casting :

Converting a lower data type into a higher one is called widening type casting. It is also known as implicit conversion or casting down. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

Both data types must be compatible with each other.

The target type must be larger than the source type.

```
public class
WideningTypeCastingExample
{
 public static void
main(String[] args)
{
 int x = 7;

 //automatically converts the integer type into long type
 long y = x;

 //automatically converts the long type into float type
 float z = y;

 System.out.println("Before conversion, int value "+x);

 System.out.println("After conversion, long value "+y);

 System.out.println("After conversion, float value "+z);
```

```
}
```

```
}
```

### Narrowing Type Casting :

Converting a higher data type into a lower one is called narrowing type casting. It is also known as explicit conversion or casting up.

It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

```
public class
NarrowingTypeCastingExample
{
 public static void main(String
args[])
 {
 double d = 166.66;
 //converting double
 data type into long data type
 long l = (long)d;
 //converting long
 data type into int data type
 int i = (int)l;

 System.out.println("Before conversion: "+d);
 //fractional part lost
```

```
System.out.println("After conversion into long type: "+l);
 //fractional part lost

System.out.println("After conversion into int type: "+i);

}
```

## Wrapper classes in Java

---

The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.

Since J2SE 5.0, autoboxing and unboxing feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java : Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc.

1. **Wrapper Classes**: In Java, there are eight wrapper classes, each corresponding to a primitive data type:

- `Byte`: Represents a byte value.
- `Short`: Represents a short value.
- `Integer`: Represents an int value.
- `Long`: Represents a long value.
- `Float`: Represents a float value.
- `Double`: Represents a double value.
- `Character`: Represents a char value.
- `Boolean`: Represents a boolean value.

2. **Autoboxing and Unboxing**:

Autoboxing : The automatic conversion of primitive data type into its corresponding wrapper class(Object) is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the `valueOf()` method of wrapper classes to convert the primitive into objects.

//Java program to convert primitive  
into objects

//Autoboxing example of int to  
Integer

```
public class WrapperExample1{
```

```

 public static void main(String
args[]){
 //Converting int into
Integer
 int a=20;
 Integer
 i=Integer.valueOf(a);//converting int into Integer explicitly
 Integer
 j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
 System.out.println(a+" "+i+" "+j);
}
}

```

**Unboxing :** The automatic conversion of wrapper type(Object) into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

```

//Java program to convert object
into primitives
//Unboxing example of Integer to int
public class WrapperExample2{
 public static void main(String
args[]){
 //Converting Integer
to int
 Integer a=new
 Integer(3);
}
}

```

```
i=a.intValue(); //converting Integer to int explicitly
int j=a; //unboxing,
now compiler will write a.intValue() internally
```

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

3. **Parsing Methods**: Wrapper classes provide methods for parsing strings and converting them to primitive data types. These methods are especially useful when you need to convert user input or data from files into the appropriate data types.

- `Integer.parseInt(String)` : Parses a string and returns an `int`.
- `Double.parseDouble(String)` : Parses a string and returns a `double`.
- `Boolean.parseBoolean(String)` : Parses a string and returns a `boolean`.

Example of parsing methods:

```
```java  
String numberString = "123";  
int number = Integer.parseInt(numberString); // Converts the string "123" to an  
int  
```
```

Using wrapper classes, autoboxing, unboxing, and parsing methods, you can work with both primitive data types and objects more flexibly and conveniently in Java.

## Reading Inputs From USER or Sources

---

The `Scanner` class in Java is a part of the `java.util` package and is used for parsing primitive types and strings using regular expressions.

It is commonly used for reading input from various sources, such as keyboard input, files, or strings.

### ### Scanner Class

#### #### Common Methods of Scanner

Here are some of the most commonly used methods of the `Scanner` class:

1. **\*\*nextInt()\*\***: Reads an `int` value.
2. **\*\*nextLine()\*\***: Reads the entire line as a `String`.
3. **\*\*next()\*\***: Reads the next complete token as a `String`.
4. **\*\*nextDouble()\*\***: Reads a `double` value.
5. **\*\*nextBoolean()\*\***: Reads a `boolean` value.
6. **\*\*nextLong()\*\***: Reads a `long` value.
7. **\*\*nextFloat()\*\***: Reads a `float` value.
8. **\*\*close()\*\***: Closes the scanner.

#### #### Example of Using Scanner

Here's a simple example demonstrating how to use the `Scanner` class to read input from the keyboard:

```
```java
import java.util.Scanner;

public class ScannerExample {

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

        System.out.println("Enter your name: ");
        String name = scanner.nextLine();

        System.out.println("Enter your age: ");
        int age = scanner.nextInt();

        System.out.println("Enter your height: ");
        double height = scanner.nextDouble();

        System.out.printf("Name: %s, Age: %d, Height: %.2f\n", name, age,
height);

        scanner.close();
    }
}

```

```

### Alternative: BufferedReader and InputStreamReader

An alternative to `Scanner` for reading input is using `BufferedReader` in combination with `InputStreamReader`. This combination can be more efficient for reading large amounts of text.

#### #### Common Methods of BufferedReader

1. \*\*readLine()\*\*: Reads a line of text.
2. \*\*close()\*\*: Closes the stream and releases any system resources associated with it.

#### #### Example of Using BufferedReader

Here's how you can use `BufferedReader` and `InputStreamReader` to read input from the keyboard:

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

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

        try {
            System.out.println("Enter your name: ");
            String name = reader.readLine();

            System.out.println("Enter your age: ");
        }
    }
}
```

```

        int age = Integer.parseInt(reader.readLine());

        System.out.println("Enter your height: ");
        double height = Double.parseDouble(reader.readLine());

        System.out.printf("Name: %s, Age: %d, Height: %.2f\n", name,
age, height);

    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

### ### Comparison: Scanner vs BufferedReader

#### 1. \*\*Ease of Use\*\*:

- `Scanner` is more user-friendly and provides methods for parsing primitive types directly.
- `BufferedReader` requires more code for parsing different types of inputs since you need to manually parse the input string.

## 2. \*\*Performance\*\*:

- `BufferedReader` is generally faster than `Scanner` because it does not perform parsing and has a larger buffer size.
- `Scanner` can be slower, especially when performing a lot of input operations, due to its parsing overhead.

## 3. \*\*Flexibility\*\*:

- `Scanner` provides a wider range of input methods and is more flexible when dealing with different types of input formats.
- `BufferedReader` is more suited for reading large blocks of text efficiently.

Choosing between `Scanner` and `BufferedReader` depends on the specific needs of your application, such as ease of use versus performance requirements.

## Java Arrays

---

### \*\*Array:\*\*

1. Normally, an array is a collection of similar type of elements which has contiguous memory location.

2. Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location.

3. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

4. Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

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

 // creating an array and initialize with values - way 1
 int[] intArray = new int[5];

 intArray[0] = 3;
 intArray[1] = 4;
 intArray[2] = 6;
 intArray[3] = 7;
 intArray[4] = 8;

 // creating an array and initialize with values - way 2
 int[] intArray2 = {14, 15, 16, 17, 18};

 for (int i : intArray) {
 System.out.println(i);
 }
 System.out.println("\n");

 for (int i = 0; i < intArray2.length; i++) {
 System.out.println(intArray2[i]);
 }
 }
}
```

```
}

System.out.println("\n");

String[] stringArray = new String[5];

stringArray[0] = "Nani";

stringArray[1] = "Welcome";

stringArray[2] = "To";

stringArray[3] = "My world";

stringArray[4] = "Thank you!";

String[] stringArray2 = {"Lemon", "Orange", "Mango", "Grape", "Apple"};

for (String str : stringArray) {

 System.out.println(str);

}

System.out.println("\n");

for (int i = 0; i < stringArray2.length; i++) {

 System.out.println(stringArray2[i]);

}

System.out.println("\n");

char[] ch = new char[4];

ch[0] = 'N';

ch[1] = 'A';

ch[2] = 'N';
```

```
ch[3] = 'I';

for (char value : ch) {
 System.out.println(value);
}

System.out.println("\n");

char[] chars = {'P', 'A', 'L', 'L', 'A', 'P', 'U'};
for (int i = 0; i < chars.length; i++) {
 System.out.println(chars[i]);
}

/*
 * Multi Dimensional Array
 */
int[][] arr = new int[3][3];//3 row and 3 column
arr[0][0] = 1;
arr[0][1] = 2;
arr[0][2] = 3;
arr[1][0] = 4;
arr[1][1] = 5;
arr[1][2] = 6;
arr[2][0] = 7;
arr[2][1] = 8;
arr[2][2] = 9;
```

```

System.out.println("===== PRINTING MULTI DIMENSIONAL ARRAY
VALUES =====");
for (int i = 0; i < arr.length; i++) {
 for (int j = 0; j < arr.length; j++) {
 System.out.print(arr[i][j] + " ");
 }
 System.out.println();
}

int[][] arr2 = {{1, 2, 3}, {2, 4, 5}, {4, 4, 5}};
System.out.println("===== PRINTING ANOTHER MULTI DIMENSIONAL
ARRAY VALUES =====");
for (int[] value1 : arr2) {
 for (int value : value1) {
 System.out.print(value + " ");
 }
 System.out.println();
}
}

```

Java String

=====

In Java, string is basically an object that represents sequence of char values. The `java.lang.String` class is used to create a string object. An array of characters works same as Java string.

For example:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
```

```
String s=new String(ch);
```

is same as:

```
String s="javatpoint";
```

Java String class provides a lot of methods to perform operations on strings such as `compare()`, `concat()`, `equals()`, `split()`, `length()`, `replace()`, `compareTo()`, `intern()`, `substring()` etc.

The `java.lang.String` class implements `Serializable`, `Comparable` and `CharSequence` interfaces.

How to create a string object?

There are two ways to create String object:

In Java, strings can be created in two ways:

1. Using String Literals (`"")`)
2. Using the `new` Keyword (`new String()`)

Each method behaves differently in terms of memory allocation and object creation.

### 1. String Creation using String Literal (`""`)

- A string literal is a sequence of characters enclosed in double quotes (`""`).

- String literals are stored in the String Constant Pool (SCP) inside the Heap Memory.

- If the same string already exists in String Constant Pool, Java returns a reference to the existing object instead of creating a new one.

#### Example

```
```java  
String s1 = "Welcome";  
String s2 = "Welcome";  
```
```

#### Memory Allocation

- Since `"Welcome"` is first created, it is stored in the String Constant Pool (SCP).

- When `s2` is created, JVM checks String Constant Pool:

- If `"Welcome"` already exists, it does not create a new object.

- Instead, it returns the reference of the existing `"Welcome"` string.

#### Diagram Representation

```

Heap Memory

| String Constant Pool (SCP)

"Welcome" (Reference shared)

s1 ----->| (Single object)

s2 ----->|

Key Points

String s1 = "Welcome";

String s2 = "Welcome";

System.out.println(s1 == s2); // true (Same reference in String Constant Pool)

System.out.println(s1.equals(s2)); // true (Same content)

- Only one object is created in String Constant Pool.
- Both `s1` and `s2` refer to the same memory location.
- This approach saves memory because Java does not create multiple objects for the same string value.

2. String Creation using `new` Keyword

- When a string is created using `new`, Java always creates a new object in the Heap Memory (not String Constant Pool).
- Even if the same string already exists in String Constant Pool, a new object is created.

Example

```
```java
```

```
String s1 = new String("Welcome");
```

```
String s2 = new String("Welcome");
```

```
```
```

Memory Allocation

- `""Welcome"` is stored in String Constant Pool if not already present.
- `s1` and `s2` create separate objects in Heap Memory, even though the content is the same.

Diagram Representation

```
```
```

```
Heap Memory | String Constant Pool (SCP)
```

```
-----> | "Welcome"
```

```
Object1 ("Welcome") |
```

```
|
```

```
s2 -----> |
```

```
Object2 ("Welcome") |
```

```
```
```

Key Points

- Two separate objects are created in Heap Memory, even though the content is the same.
- `s1 == s2` will return `false` because they reference different objects.

- `s1.equals(s2)` will return `true` because their content is the same.

```
```java
```

```
System.out.println(s1 == s2); // false (Different objects)
```

```
System.out.println(s1.equals(s2)); // true (Same content)
```

```

```

```

```

## ## Comparison of Both Approaches

Feature	String Literal (`""`)	`new String()`
Memory Allocation	String Constant Pool (SCP)	Heap Memory
Object Creation	Only if not already present	Always creates a new object
Memory Efficiency	High	Low (creates duplicates)
`==` Comparison	True (same reference)	False (different objects)
`.equals()` Check	True	True

```

```

## ## When to Use Which Approach?

- Use string literals (`""`) when you want to save memory and reuse existing strings.
- Use `new String()` if you explicitly need a new string object (rare case).

In most cases, using string literals is the preferred choice because it improves performance and saves memory.

In Java, `String` objects are immutable, which means their state (the data they represent) cannot be changed after they are created. Once a `String` object is created, you cannot modify its contents. When you perform operations that seem to modify a `String`, you are actually creating a new `String` object rather than modifying the existing one. This design choice has several implications and advantages.

#### **\*\*Why Strings are Immutable:\*\***

#### **\*\*Immutability of Strings in Java:\*\***

##### **\*\*1. Stack and Heap Areas:\*\***

- In Java, the stack and heap are two memory areas used for storage.
  - **Stack:** It stores method call information, local variables, and reference variables. It's fast but limited in size.
  - **Heap:** It's a larger memory area where objects are stored, and it's managed by the garbage collector.

##### **\*\*2. String Constant Pool:\*\***

- The String Constant Pool is a special area in the heap that stores string literals (like "Hello"). It's a shared pool, and when you create a string literal, Java checks if it exists in the pool.

##### **\*\*3. Immutability:\*\***

- Strings in Java are immutable, meaning their content cannot change after creation.
- When you create a new string, it goes to the String Constant Pool if it's a literal. Otherwise, it's created in the heap.

- If another string with the same content is needed, Java reuses the existing one (if it's a literal).

#### \*\*How References are Saved and Accessed:\*\*

- Reference variables (like `String str;`) in the stack store memory addresses (references) pointing to objects in the heap.
- When you assign a new value to a reference, you change the memory address it points to.

#### \*\*Why StringBuilder and StringBuffer are Mutable:\*\*

- `StringBuilder` and `StringBuffer` are mutable string classes in Java, designed for efficient string manipulation.
- They use a dynamic array to store characters, allowing for in-place modifications.
- Unlike immutable strings, where a new object is created for each modification, mutable classes modify the existing object, reducing memory overhead.

#### \*\*Example:\*\*

```
```java
// Immutable String
String s1 = "Hello";
String s2 = s1.concat(" World"); // New string created
System.out.println(s1); // Output: Hello
System.out.println(s2); // Output: Hello World

// Mutable StringBuilder
StringBuilder sb1 = new StringBuilder("Hello");
sb1.append(" World"); // Modifies the existing StringBuilder
```

```
System.out.println(sb1); // Output: Hello World
```

```
---
```

****In Summary:****

- Strings are immutable in Java for thread safety, security, and simplicity.
- String literals go to the String Constant Pool, and dynamic strings are created in the heap.
- References in the stack point to objects in the heap.
- `StringBuilder` and `StringBuffer` are mutable for efficient string manipulation. They modify the existing object instead of creating new ones.

1. **Thread Safety:**

- Immutability makes `String` objects thread-safe. Since the content of a `String` cannot be changed, multiple threads can safely share and use the same `String` object without the need for synchronization. This simplifies concurrent programming.

2. **Security:**

- Because strings are used extensively in Java for representing textual data, immutability helps in building secure systems. For example, if a `String` representing a password is passed between methods or classes, immutability ensures that the password cannot be changed inadvertently.

3. **Caching and String Pool:**

- Java maintains a pool of string literals known as the "String pool." Immutability allows Java to reuse existing string literals, reducing memory consumption. When you create a new string with the same value, Java can check the pool and return the existing instance if available.

```
```java
String s1 = "Hello";

String s2 = "Hello"; // Reuses the same instance from the pool
```

```

4. **Hash Code Calculation:**

- Since the content of a `String` does not change, its hash code can be calculated once and cached. This is used in hash-based data structures like `HashMap` and `HashSet`.

Why Other Data Types are not Immutable:

While `String` is the most prominent example of immutability in Java, other fundamental data types, such as `Integer`, `Double`, and `Boolean`, are also immutable. This design choice brings similar benefits:

1. **Predictable and Safe Behavior:**

- Immutable objects behave predictably. Once created, their state cannot be changed, leading to safer and more reliable code. This predictability is crucial in scenarios where the value of an object should remain constant.

2. **Consistency:**

- Immutability promotes consistency in the language. If some data types were mutable while others were not, it could lead to confusion and unexpected behavior.

3. **Simplicity:**

- Immutable objects simplify the design and understanding of the code. Since the state of the object does not change, you can reason about the behavior of the code more easily.

However, it's important to note that not all data types in Java are immutable. For example, arrays, `StringBuilder`, and `ArrayList` are mutable. The choice between mutable and immutable objects depends on the specific requirements and use cases of the data type. Immutability is favored when predictability, thread safety, and ease of reasoning about code are critical. Mutable objects are preferred when dynamic modifications to the object's state are required.

for more explanation open this video : <https://youtu.be/5OXc-TsM3bg?si=LOaFCqlWqaBxyhaE>

Sure, here's a breakdown of immutable objects and immutable classes in Java, along with their uses, presented in a point-wise manner:

StringBuilder and StringBuffer Differences

Differences:

1. **Thread Safety**: `StringBuffer` is thread-safe due to synchronization, making it suitable for multi-threaded environments. `StringBuilder` is not thread-safe, making it more efficient in single-threaded scenarios but requiring external synchronization in multi-threaded environments.
2. **Performance**: `StringBuilder` is generally faster than `StringBuffer` because it lacks synchronization overhead. However, the difference may be negligible in many applications.

Immutable Objects:

=====

1. **Definition:** An immutable object is an object whose state cannot be modified after it is created. Once created, the values of the object's fields remain constant throughout its lifetime.

2. **Characteristics:**

- Immutable objects have all their fields marked as `final`, making them unchangeable.
- They don't have any setter methods to modify their state after construction.
- Any modification to an immutable object results in creating a new instance with the updated values.
- Default constructor(constructor without arguments) can not be created.

3. **Thread Safety:** Immutable objects are inherently thread-safe because their state cannot be changed after creation. This eliminates the need for synchronization in multithreaded environments.

4. **Benefits:**

- Simplicity: Immutable objects simplify code by ensuring that their state remains constant.
- Thread Safety: They eliminate race conditions and synchronization issues in concurrent programming.
- Ease of Caching: Immutable objects can be safely cached since their values never change.

Note :

String is immutable because its internal `char[]` array is `final`, meaning its reference cannot change.

However, not all methods and fields need to be `final` because Java ensures immutability by preventing modification rather than forcing everything to be `final`.

This also allows JVM optimizations.

5. **Example**:

```
```java

public final class ImmutablePerson {

 private final String name;

 private final int age;

 public ImmutablePerson(String name, int age) {

 this.name = name;

 this.age = age;

 }

 public String getName() {

 return name;

 }

 public int getAge() {

 return age;

 }

}

```

public class Main {

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

```
    ImmutablePerson person = new ImmutablePerson("John", 30); //  
Immutable Object  
  
    System.out.println("Name: " + person.getName());  
  
    System.out.println("Age: " + person.getAge());  
  
  
    // Attempting to modify the object's state will result in a compilation  
error  
  
    // person.setName("Alice"); // Compilation error  
  
}
```

- In this example, `ImmutablePerson` is an immutable class with two private final fields, `name` and `age`. It provides only getter methods to access the values of its fields, ensuring that its state remains constant after construction.

Immutable objects and classes offer simplicity, thread safety, and ease of reasoning about code. By enforcing immutability, Java developers can create more robust and maintainable software systems.

EXCEPTION HANDLING

****Exception**** : Exception is an abnormal condition. In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

****Exception-Handling**** : The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

Exception are two types : 1. Checked Exception 2. Unchecked Exception.

1. Checked Exceptions (Compile-Time Exceptions):

The classes that directly inherit the `Throwable` class except `RuntimeException` are known as checked exceptions.

Checked exceptions are exceptions that are checked at compile time by the Java compiler.

This means that if a method can throw a checked exception, the programmer is required to handle it using `try-catch` blocks or declare it using the `throws` clause in the method signature.

Examples of checked exceptions include:

- `'IOException'`: Occurs when there are problems with input and output operations.
- `'FileNotFoundException'`: Occurs when an attempt is made to access a file that does not exist.
- `'SQLException'`: Occurs when there are database-related errors.

Checked exceptions are typically related to external factors and resources.

Example of handling a checked exception:

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

```
public class Main {
 public static void main(String[] args) {
 try {
 // Code that may throw a checked exception
 } catch (IOException e) {
 // Handle the exception here
 }
 }
 ...
}
```

## \*\*2. Unchecked Exceptions (Runtime Exceptions):\*\*

The classes that inherit the `RuntimeException` are known as unchecked exceptions.

Unchecked exceptions, also known as runtime exceptions, are exceptions that are not checked at compile time. They typically represent programming errors, and it's not required to handle them explicitly using `try-catch` blocks.

Examples of unchecked exceptions include:

- `'NullPointerException'`: Occurs when trying to access an object or variable that is `'null'`.
- `'ArithmaticException'`: Occurs when there is an arithmetic error, such as division by zero.
- `'ArrayIndexOutOfBoundsException'`: Occurs when trying to access an array element with an index that is out of bounds.

Unchecked exceptions are usually the result of mistakes in the code, so the focus is on fixing the underlying issue rather than handling the exception.

Example of an unchecked exception:

```
```java
public class Main {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};
        int result = numbers[4]; // Throws
        ArrayIndexOutOfBoundsException
    }
}
```

```

"Throw" and "throws" are related concepts in Java used for exception handling, but they serve different purposes and are used in different contexts:

1. \*\*`throw`\*\*:

- **Definition**: `throw` is a keyword in Java used to manually throw an exception. When you use `throw`, you are explicitly raising an exception in your code, indicating that something exceptional has occurred.

- **Purpose**: It is used to throw exceptions, indicating that a specific error condition has been encountered. You can throw built-in exceptions or custom exceptions.

- **Example**:

```
```java
public void divide(int numerator, int denominator) {
    if (denominator == 0) {
        throw new ArithmeticException("Division by zero");
    }
    int result = numerator / denominator;
}
````
```

2. **`throws`**:

- **Definition**: `throws` is a modifier used in a method signature to declare that the method may throw one or more exceptions. It is used to specify which exceptions the method might throw, allowing the calling code to handle them.

- **Purpose**: It is used to declare that a method may throw exceptions, providing information to the caller about potential exceptions that need to be handled or propagated.

- **Example**:

```
```java
```

```
public int divide(int numerator, int denominator) throws  
ArithmeticException {  
  
    if (denominator == 0) {  
  
        throw new ArithmeticException("Division by zero");  
  
    }  
  
    return numerator / denominator;  
  
}  
  
...  
  
}
```

In the "throw" example, we manually throw an exception when a specific error condition is encountered. In the "throws" example, we declare in the method signature that the method may throw an exception, allowing the calling code to be aware of the potential exception and handle it.

It's important to note that "throw" is used within the method where the exception occurs, while "throws" is used in the method signature to declare exceptions that may be thrown by that method. These concepts work together to facilitate proper exception handling in Java.

3. `try`-`catch` Block:

- **Purpose**: Used for exception handling in Java.
- **Explanation**: The `try` block contains the code that might throw an exception. If an exception occurs, it is caught by the `catch` block, where you can handle the exception.
- **Why we use**: To gracefully handle exceptions and prevent program termination due to errors.

```java

```

public class TryCatchExample {

 public static void main(String[] args) {
 try {
 int result = divide(10, 0); // This may throw an
ArithmaticException

 } catch (ArithmaticException e) {
 System.out.println("Error: Division by zero");
 }
 }

 public static int divide(int numerator, int denominator) {
 return numerator / denominator;
 }
}
```

```

4. `try`-with-resources:

- **Purpose**: Used to automatically close resources after they are no longer needed.

- **Explanation**: Resources that implement the `AutoCloseable` interface (such as I/O streams) are declared within the `try` block. They are automatically closed when the `try` block exits, regardless of whether an exception occurs.

- **Why we use**: Ensures proper resource management and prevents resource leaks by automatically closing resources.

```
```java
```

```
import java.io.BufferedReader;
```

```

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

public class TryWithResourcesExample {

 public static void main(String[] args) {
 try (FileReader reader = new FileReader("file.txt");
 BufferedReader bufferedReader = new
 BufferedReader(reader)) {
 String line = bufferedReader.readLine(); // Read from the
 file
 System.out.println("First line of file: " + line);

 boolean deleted = new File("file.txt").delete();
 System.out.println("Was file deleted? " + deleted);
 } catch (IOException e) {
 System.out.println("Error reading file: " + e.getMessage());
 }
 }
}
```

```

** ❌ Example Without br.close() (Resource Leak) **

```

public class ResourceLeakExample {

    public static void main(String[] args) {
        String path =
        "C://Users//nanip//Downloads//insurance_project_requirement.txt";

```

```
BufferedReader br = null;

try {

    br = new BufferedReader(new FileReader(path));

    System.out.println("File content: " + br.readLine());

    // ✘ Forgetting to close the stream
    // br.close(); // Oops! This is missing!

} catch (IOException e) {

    e.printStackTrace();

}

// Let's try to delete or open the file again (simulate another
operation)

try {

    boolean deleted = new File(path).delete();

    System.out.println("Was file deleted? " + deleted);

} catch (Exception e) {

    System.out.println("Error trying to delete file:");

    e.printStackTrace();

}

}
```

What Happened:

```
=====
```

File is still open by JVM

delete() fails because file handle wasn't released

On some OS (like Windows), you'll get an error like:

java.io.IOException: The process cannot access the file because it is being used by another process

5. `finally` Block:

- **Purpose**: Used to execute cleanup code that should always run, regardless of whether an exception occurs. usually we can use it close the resources.

- **Explanation**: The `finally` block is executed

creating a custom exception

Custom exceptions in Java are like special error messages you create to explain specific problems in your program.

They help you handle issues in a way that makes sense for your application, making your code easier to understand and maintain.

let's say you're building a banking application in Java. You might encounter a situation where an account balance is insufficient for a withdrawal.

Instead of using a generic `Exception`, you can create a custom exception to represent this specific scenario:

```
```java
public class InsufficientFundsException extends Exception {
 public InsufficientFundsException(String message) {
 super(message);
 }
}
```
``
```

Now, in your banking code, when a withdrawal attempt is made, you can use this custom exception:

```
```java
public class BankAccount {
 private double balance;

 public BankAccount(double initialBalance) {
 this.balance = initialBalance;
 }

 public void withdraw(double amount) throws
InsufficientFundsException {
 if (amount > balance) {
 throw new InsufficientFundsException("Insufficient
funds for withdrawal");
 }
 // Perform the withdrawal logic if there's enough balance
 }
}
```
``
```

```
        balance -= amount;  
  
        System.out.println("Withdrawal successful. Remaining  
balance: " + balance);  
  
    }  
  
}  
  
...  
  
Now, when someone tries to withdraw more money than the account  
balance, you get a clear and specific error message indicating an insufficient funds situation:
```

```
```java  
public class Main {

 public static void main(String[] args) {

 BankAccount account = new BankAccount(100.0);

 try {

 account.withdraw(150.0); // This will throw
InsufficientFundsException

 } catch (InsufficientFundsException e) {

 System.out.println("Error: " + e.getMessage());

 }

 }

...

This makes your code more readable and helps you handle specific issues
appropriately in your application.
```

In Java, creating a custom exception involves extending the `Exception` class or one of its subclasses. Here's a step-by-step guide on how to create and use a custom exception:

In this example, `CustomException` extends the `Exception` class. The constructor takes a `String` parameter that represents the error message associated with the exception.

#### #### Important Notes:

- Custom exceptions are typically used to handle specific error conditions in your application.
  - The `throws` keyword in the method signature indicates that the method might throw the specified exception.
  - The `try-catch` block is used to handle exceptions gracefully and provide a way to recover from errors.

Remember to adjust the class names and structures based on your specific application needs.

## Global Level Exception Handling

---

Global level exception handling in Java typically refers to handling exceptions that occur across the entire application, rather than handling exceptions on a per-method or per-class basis.

In a Spring application, global exception handling can be achieved using `@ControllerAdvice` or `@RestControllerAdvice` annotations combined with `@ExceptionHandler` methods.

Let's see an example of global exception handling and discuss bean scopes:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice
public class GlobalExceptionHandler {

 @ExceptionHandler(StudentNotFoundException.class)
 public ResponseEntity<String>
handleStudentNotFoundException(StudentNotFoundException ex) {
 return
 ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
}

 @ExceptionHandler(Exception.class)
 public ResponseEntity<String> handleGlobalException(Exception ex) {
 return
 ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Internal Server Error");
}
}
```

```
@RestController
public class StudentController {

 @GetMapping("student/{id}")
 public ResponseEntity<Student> getStudentById(@PathVariable int id) {
 if (id == 0) {
 throw new StudentNotFoundException("Student with id " + id + "
not found");
 }
 return ResponseEntity.ok(new Student(id, "John Doe"));
 }

}

public class StudentNotFoundException extends RuntimeException {
 public StudentNotFoundException(String message) {
 super(message);
 }
}

...

In this example:
```

- We have a `GlobalExceptionHandler` class annotated with `@ControllerAdvice`. This annotation indicates that this class provides centralized exception handling for all controllers in the application.

- The `handleException` method within `GlobalExceptionHandler` is annotated with `@ExceptionHandler(Exception.class)`. This annotation indicates that this method handles exceptions of type `Exception` and its subclasses.

- Inside the `handleException` method, we log the exception and return an HTTP response with an error message.

- We also have an `ExampleController` class with a method `exampleMethod`, which throws a `RuntimeException`. This simulates an exception occurring in the application.

- When an exception occurs in the `exampleMethod`, it will be caught by the `handleException` method in `GlobalExceptionHandler`, and an appropriate error response will be returned.

## HTTP STATUS CODES

---

- **200 OK**: The request has succeeded.
- **201 Created**: The request has been fulfilled and resulted in a new resource being created.
- **400 Bad Request**: The server cannot or will not process the request due to an apparent client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).
- **401 Unauthorized**: Similar to 403 Forbidden, but specifically for use when authentication is required and has failed or has not yet been provided.
- **403 Forbidden**: The client does not have access rights to the content, i.e., they are unauthorized, so the server is refusing to give the requested resource.
- **404 Not Found**: The server can't find the requested resource. In the context of REST APIs, this is often used for a resource that doesn't exist.
- **405 Method Not Allowed**: The method specified in the request-line is not allowed for the resource identified by the request-target.

- **\*\*500 Internal Server Error\*\***: A generic error message, given when an unexpected condition was encountered and no more specific message is suitable.
- **\*\*503 Service Unavailable\*\***: The server is currently unable to handle the request due to a temporary overload or maintenance of the server.

## INNER CLASS IN JAVA

---

In Java, an inner class is a class defined within another class. Inner classes provide a way to logically group classes within another class, and they have several use cases and benefits, such as encapsulation and code organization. There are several types of inner classes in Java:

### 1. **\*\*Non-static Inner Class (Member Inner Class):\*\***

- A non-static inner class is a class that is defined inside another class without the `static` keyword.
  - It has access to the instance variables and methods of the outer class, including private members.
  - To create an instance of a non-static inner class, you typically need to create an instance of the outer class first.

```
```java
public class OuterClass {
    private int outerField;

    public class InnerClass {
        public void display() {
```

```
        System.out.println("Outer field: " + outerField);

    }

}

```

```

## 2. \*\*Static Inner Class (Nested Class):\*\*

- A static inner class is a class defined inside another class with the `static` keyword.

- It does not have access to the instance variables and methods of the outer class directly, but it can access static members.

- You can create an instance of a static inner class without creating an instance of the outer class.

```
```java

public class OuterClass {

    private static int outerStaticField;

    public static class InnerClass {

        public void display() {
            System.out.println("Outer static field: " +
outerStaticField);
        }
    }
}

```

```

### 3. \*\*Local Inner Class:\*\*

- A local inner class is defined inside a method or block (e.g., a code block within a method).
  - It has local scope and can access final or effectively final local variables from the enclosing method.

```
```java
public class OuterClass {

    public void display() {

        final int localVar = 42;

        class LocalInnerClass {

            public void printLocalVar() {

                System.out.println("Local variable: " + localVar);

            }
        }

        LocalInnerClass inner = new LocalInnerClass();

        inner.printLocalVar();

    }
}

```

```

### 4. \*\*Anonymous Inner Class:\*\*

- An anonymous inner class is a class without a name defined inside a method's argument or within an expression.

- It is often used to implement an interface or extend a class and override its methods.

```
```java
public class OuterClass {
    public void displayMessage() {
        new Thread(new Runnable() {
            public void run() {
                System.out.println("Anonymous inner class
thread is running.");
            }
        }).start();
    }
}
```

```

Inner classes can be useful for implementing encapsulation, achieving better code organization, and enhancing code readability.

The choice of which type of inner class to use depends on the specific requirements of your program and the level of access needed to the outer class's members.

## JAVA 8 FEATURES:

---

Sure, here's a list of key **Java 8 features**:

1. **Lambda Expressions**

2. \*\*Functional Interfaces\*\*
3. \*\*Method References\*\*
4. \*\*Default & Static Methods in Interfaces\*\*
5. \*\*Streams API\*\*
6. \*\*Optional Class\*\*
7. \*\*Date and Time API (java.time package)\*\*
8. \*\*Collectors (java.util.stream.Collectors)\*\*
9. \*\*Parallel Streams\*\*
10. \*\*Base64 Encoding and Decoding (java.util.Base64)\*\*

## 1. Functional Interface

---

A functional interface in Java is an interface that has exactly one abstract method. Functional interfaces are also known as Single Abstract Method (SAM) interfaces.

These interfaces are a fundamental concept in Java's support for lambda expressions and the Java Stream API.

Functional interfaces are used to represent single behaviors, and they can be implemented as lambda expressions or method references, making code more concise and expressive.

Here are the key characteristics and rules for functional interfaces in Java:

### 1. \*\*Single Abstract Method (SAM)\*\*:

- A functional interface should have exactly one abstract (unimplemented) method. This method is referred to as the "functional method" or "SAM method."

- A functional interface can have additional default or static methods, but there must be only one abstract method.

## 2. \*\*@FunctionalInterface Annotation\*\*:

- While it's not strictly required, you can use the `@FunctionalInterface` annotation to explicitly declare an interface as a functional interface. This annotation helps provide clarity and allows the compiler to generate an error if there are multiple abstract methods.

## 3. \*\*Lambda Expressions\*\*:

- You can use lambda expressions to create instances of functional interfaces. The lambda expression provides the implementation of the single abstract method defined in the interface.

- Lambda expressions are a concise way to represent behavior or functionality.

Here's an example of a simple functional interface in Java:

```
```java
@FunctionalInterface
interface MyFunctionalInterface {
    void doSomething();
}

public class Main {
    public static void main(String[] args) {
        // Using a lambda expression to implement the functional
        interface
            MyFunctionalInterface functionalObj = () -> {
                System.out.println("Doing something...");};
    }
}
```

```
        functionalObj.doSomething();  
    }  
}  
...  
  
In this example, `MyFunctionalInterface` is a functional interface with a single abstract method, `doSomething()`. We use a lambda expression to provide the implementation for the `doSomething()` method.
```

Functional interfaces are widely used in Java, especially in the context of the Stream API, where they enable concise and expressive code for operations on collections. Java provides several built-in functional interfaces in the `java.util.function` package, such as `Predicate`, `Function`, `Consumer`, and `Supplier`, to cover common use cases for lambda expressions and the Stream API.

These functional interfaces simplify the development of functional-style programming in Java.

SOME IMPORTANT FUNCTIONAL INTERFACES :

Certainly! Here's an explanation of each functional interface along with examples:

1. **Supplier**:

- `Supplier` represents a supplier of results. It is functional interface. It does not accept any arguments but produces a result(T).
- T: denotes the type of the result. we can use Supplier for Lazy Loading.
- It has a single method `T get()` that returns a result.

```
Supplier<T> supplier = () -> "Hello, world!";

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

public class SupplierExample {
 public static void main(String[] args) {
 Supplier<String> supplier = () -> "Hello, world!";
 System.out.println(supplier.get()); // Output: Hello,
world!
 }
}
```

```

2. **Function**:

- `Function` represents a function that accepts one argument and produces a result.
- It has a single method `R apply(T t)`.
- T: denotes the type of the input argument
- R: denotes the return type of the function

```
Function<T, R> square = x -> x * x;
```

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

```

public class FunctionExample {

 public static void main(String[] args) {
 Function<Integer, Integer> square = x -> x * x;
 System.out.println(square.apply(5)); // Output: 25
 }
}
```

```

3. **BiFunction**:

- `BiFunction` represents a function that accepts two arguments and produces a result.
- It has a single method `R apply(T t, U u)`.

t– the first function argument

u– the second function argument

Returns: This method returns the function result which is of type R.

```

```java
import java.util.function.BiFunction;

public class BiFunctionExample {

 public static void main(String[] args) {
 BiFunction<Integer, Integer, Integer> add = (x, y) -> x + y;
 System.out.println(add.apply(3, 5)); // Output: 8
 }
}
```

```

```

#### 4. \*\*UnaryOperator\*\*:

- `UnaryOperator` represents an operation on a single operand that produces a result of the same type as its operand.
  - It extends `Function` and has a single method `T apply(T t)`.
  - T: denotes the type of the input argument to the operation and return type

```java

```
import java.util.function.UnaryOperator;

public class UnaryOperatorExample {

    public static void main(String[] args) {
        UnaryOperator<Integer> square = x -> x * x;
        System.out.println(square.apply(4)); // Output: 16
    }
}
```

```

#### 5. \*\*BinaryOperator\*\*:

- `BinaryOperator` represents an operation upon two operands of the same type, producing a result of the same type as the operands.
  - It extends `BiFunction` and has a single method `T apply(T t1, T t2)`.
  - T: denotes the type of the input argument to the operation and return type.

```
```java
import java.util.function.BinaryOperator;

public class BinaryOperatorExample {

    public static void main(String[] args) {
        BinaryOperator<Integer> add = (x, y) -> x + y;
        System.out.println(add.apply(3, 5)); // Output: 8
    }
}

```

```

#### 6. \*\*Predicate\*\*:

- `Predicate` represents a predicate (boolean-valued function) of one argument. It is functional interface.
- It has a single abstract method `boolean test(T t)` and other default and static methods.

```
```java
import java.util.function.Predicate;

public class PredicateExample {

    public static void main(String[] args) {
        Predicate<Integer> isEven = x -> x % 2 == 0;
        System.out.println(isEven.test(4)); // Output: true
    }
}
```

```
}
```

```
...
```

7. **Consumer**:

- `Consumer` represents an operation that accepts a single input argument and returns no result.

- It has a single method `void accept(T t)`.

- However these kind of functions don't return any value.

```
```java
```

```
import java.util.function.Consumer;
```

```
public class ConsumerExample {
 public static void main(String[] args) {
 Consumer<String> print = message ->
 System.out.println(message);
 print.accept("Hello, world!"); // Output: Hello, world!
 }
}
```

```
...
```

#### 8. \*\*BiConsumer\*\*:

- `BiConsumer` represents an operation that accepts two input arguments and returns no result.

- It has a single method `void accept(T t, U u)`.

```
```java
```

```
import java.util.function.BiConsumer;

public class BiConsumerExample {

    public static void main(String[] args) {
        BiConsumer<String, Integer> printLength = (str, length) ->
        System.out.println("Length of " + str + ": " + length);
        printLength.accept("Hello", 5); // Output: Length of
        Hello: 5
    }
}
```

```

These functional interfaces are extensively used in Java for functional programming and can be particularly handy when working with streams, lambda expressions, and method references.

These interfaces provide a way to pass behavior in a more functional style, especially useful when working with streams, lambdas, and functional programming paradigms in Java.

### One Example Program for Important Functional INTERFACES

---

```
package company.com.java8features;
```

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

```
public class SomeFunctionalInterfacesExample {

 public static void supplierExample() {

 Supplier<String> supplier = () -> "Nani Babu";

 Supplier<Integer> integerSupplier = () -> 26;

 System.out.println("I am " + supplier.get() + ". My Age is " +
integerSupplier.get());

 }

}
```

```
public static void functionExample() {

 Function<String, Integer> function = name ->
name.length();

 System.out.println("Length Of Given Name : " +
function.apply("Nani Babu"));

}
```

```
public static void biFunctionExample() {

 BiFunction<Integer, Double, Boolean> biFunction =
(value1, value2) -> value1 > value2;

 System.out.println(biFunction.apply(5,4.5));

}

public static void unaryOperatorExample(){

 UnaryOperator<Integer> unaryOperator = value -> value *
value;

 System.out.println(unaryOperator.apply(5));

}

public static void binaryOperator(){

 BinaryOperator<Integer> binaryOperator = (value1,
value2) -> value1 * value2;
```

```

 System.out.println(binaryOperator.apply(3, 2));

 }

public static void predicateExample(){

 Predicate<Integer> predicate = value -> value % 2 == 0; // checking even or odd number which returns boolean and takes input as Integer

 System.out.println(predicate.test(4));

 System.out.println(predicate.test(9));

}

public static void consumerExample(){

 Consumer<String> consumer = (name) ->
System.out.println("Hello "+name);

 consumer.accept("Nani Babu");

}

public static void biConsumerExample(){

 BiConsumer<String, Integer> biConsumer = (value1, value2) -> System.out.println("I am "+value1+". My age is : "+value2);

 biConsumer.accept("Nani Babu", 26);

}

public static void main(String[] args) {

 System.out.println("\n");

 supplierExample();

 System.out.println("\n");

 functionExample();

 System.out.println("\n");
}

```

```
 biFunctionExample();

 System.out.println("\n");

 unaryOperatorExample();

 System.out.println("\n");

 binaryOperator();

 System.out.println("\n");

 predicateExample();

 System.out.println("\n");

 consumerExample();

 System.out.println("\n");

 biConsumerExample();

 System.out.println("\n");

 }

}
```

## \*\*2. Lambda Expressions\*\*

---

Lambda expressions in Java provide a concise way to represent anonymous functions (functions that don't have a name) and are a part of the Java programming language since Java 8.

Lambda expressions are primarily used for writing code more briefly and with less boilerplate code, making your code more readable and expressive.

Here's a breakdown of lambda expressions in Java:

**\*\*Syntax\*\*:**

Lambda expressions have the following syntax:

```

(parameters) -> expression

```

- `parameters`: This represents the input parameters to the lambda expression.
- `->`: It is the lambda operator, which separates the parameters from the expression.
- `expression`: This is the body of the lambda expression, where you specify what the lambda should do. This can be a single expression or a block of code enclosed in curly braces `{}`.

**\*\*Usage\*\*:**

Lambda expressions are often used in places where functional interfaces are expected. A functional interface is an interface with a single abstract method. Java provides many built-in functional interfaces in the `java.util.function` package, such as `Predicate`, `Consumer`, and `Function`, which can be used with lambda expressions.

Here are some common examples of using lambda expressions:

1. **Functional Interfaces**:

```
```java
// Using a lambda expression to define a Predicate
Predicate<Integer> isEven = (num) -> num % 2 == 0;
```

```
// Using a lambda expression with a Consumer
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.forEach(name -> System.out.println("Hello, " + name));
```

```

2. **Threads**:

```
```java
// Using a lambda expression to define a Runnable for a new thread
Thread thread = new Thread(() -> {
    for (int i = 0; i < 10; i++) {
        System.out.println("Thread: " + i);
    }
});
thread.start();
```

```

3. **Sorting**:

```
```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.sort((name1, name2) -> name1.compareTo(name2));
```

```

Lambda expressions are a powerful feature that simplifies code, especially when working with collections, event handling, and functional programming constructs. They allow you to define behavior directly where it's needed, reducing the need for writing separate classes or anonymous inner classes.

### 3. Java Streams API

---

Java Streams are a powerful and versatile feature introduced in Java 8 for processing sequences of elements (e.g., collections) in a functional and declarative manner.

They provide a concise and expressive way to perform operations on data, making your code more readable and efficient.

Here are some key concepts and operations related to Java Streams:

#### 1. **Stream Creation**:

- You can create a Stream from various data sources, including collections, arrays, I/O channels, and more. For example:

```
```java
```

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

2. **Intermediate and Terminal Operations**:

- Streams consist of two types of operations: intermediate and terminal.
- Intermediate operations transform a Stream into another Stream and Methods are chained together. Examples include `filter`, `map`, `sorted`, `peek`, `skip`, and `limit` etc.
- Terminal operations produce a result or a side effect. Examples include `forEach`, `collect`, `reduce`, `count`, `min`, `max`, `anyMatch`, `noneMatch`, `findFirst`, `findAny` etc.

3. **Filtering**:

- You can use the `filter` method to selectively include elements in a Stream based on a given condition.

4. **Mapping**:

- The `map` method allows you to transform elements in a Stream to another type using a provided function.

5. **Sorting**:

- The `sorted` method orders the elements in a Stream based on a natural or custom order.

6. **Reduction**:

- The `reduce` operation combines elements in a Stream into a single result using a specified binary operator.

7. **Collecting**:

- The `collect` method is used to accumulate the elements of a Stream into a collection, such as a List or Set.

8. **ForEach**:

- The `forEach` operation performs an action on each element in the Stream.

9. **Grouping and Partitioning**:

- Streams support operations like `groupingBy` and `partitioningBy` for grouping elements into maps or partitions.

10. **Infinite Streams**:

- Streams can be infinite. You can create infinite Streams using operations like `generate` and `iterate`.

11. **Parallel Processing**:

- Streams support parallel processing, allowing for efficient use of multi-core processors. You can use the `parallel` and `parallelStream` methods to enable parallel execution.

12. **Lazy Evaluation**:

- Streams are lazily evaluated, which means intermediate operations are only executed when a terminal operation is called. This allows for efficient processing of large data sets.

Here's an example of a simple Java Stream pipeline that filters and transforms a list of integers:

```
```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sumOfSquares = numbers.stream()

 .filter(n -> n % 2 == 0) // Filter even numbers

 .map(n -> n * n) // Square each number

 .reduce(0, Integer::sum); // Sum the squared numbers

```
```

```

This example demonstrates how you can use Stream operations to perform data processing in a concise and readable way. Streams are a fundamental part of modern Java programming and are widely used in a variety of applications.

#### 4. Default and Static Methods in Interfaces

---

**\*\*Definition:\*\***

Java 8 allows interfaces to contain:

- **\*\*Default methods:\*\*** Provide a default implementation to avoid breaking existing classes.
- **\*\*Static methods:\*\*** Can be called without an object, directly using the interface name.

**\*\*Real-life Use Case:\*\*** Enhancing interfaces like `List`, `Map` without breaking older implementations.

#### #### ✅ Example: `UserService` Interface

Let's say we have a system that manages users. The interface provides:

- A `registerUser(String username)` method to be implemented.
- A \*\*default\*\* method to validate the username.
- A \*\*static\*\* method to show common rules.

---

#### ####💡 Java Code

```
```java
@FunctionalInterface
interface UserService {

    // Abstract method
    void registerUser(String username);

    // Default method with validation logic
    default boolean isValidUser(String username) {
        if (username == null || username.trim().isEmpty()) {
```

```

        System.out.println(" ✗ Username cannot be empty.");
        return false;
    }

    if (username.length() < 3) {
        System.out.println(" ✗ Username must be at least 3
characters.");
        return false;
    }

    return true;
}

// Static method to print common rules
static void printUserRules() {
    System.out.println(" ✎ Rules:");
    System.out.println("1. Username must not be empty.");
    System.out.println("2. Username must be at least 3 characters.");
}

}
```

```

### ### ✅ Implementation Class

```

```java
class UserServiceImpl implements UserService {
```

```
@Override
public void registerUser(String username) {
    if (isValidUser(username)) {
        System.out.println(" ✅ User '" + username + "' registered
successfully.");
    } else {
        System.out.println(" ⚠️ Registration failed for username: "
+ username);
    }
}
...
---
---
```

```
### 📄 `Main.java`


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

 // Call static method directly via interface
 UserService.printUserRules();

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

```
// Create object
UserService service = new UserServiceImpl();

// Valid registration
service.registerUser("nani");

// Invalid registration
service.registerUser("");
}
}


```

### ### Output

---

#### Rules:

1. Username must not be empty.
2. Username must be at least 3 characters.

 User 'nani' registered successfully.

 Username cannot be empty.

 Registration failed for username:

---

---

### ### Summary

Part	Used for
`abstract`	Main business logic → `registerUser()`
`default`	Shared reusable logic → `isValidUser()`
`static`	Utility info → `printUserRules()`

## 5. \*\*Method References\*\*

=====

### \*\*Definition:\*\*

A method reference is just a shorthand for a lambda expression that calls an existing method.

And lambda expressions can only be assigned to functional interfaces — i.e., interfaces with exactly one abstract method.

### \*\*Syntax:\*\*

```java

ClassName::methodName

```

**\*\*Types of Method References:\*\***

1. **Reference to a static method**
2. **Reference to an instance method of a particular object**
3. **Reference to an instance method of an arbitrary object of a particular type**
4. **Reference to a constructor**

---

**\*\* Example 1: Static Method Reference\*\***

```
```java

class Utils {

    public static void greet() {

        System.out.println("Hello from static method!");

    }

}

public class Demo {

    public static void main(String[] args) {

        Runnable r = Utils::greet; // Method reference

        r.run();

    }

}

```

```

\*\*  Example 2: Instance Method of a Particular Object\*\*

```
```java
class Printer {

    public void print(String message) {
        System.out.println("Message: " + message);
    }
}

public class Demo {

    public static void main(String[] args) {
        Printer printer = new Printer();
        Consumer<String> c = printer::print; // Method reference
        c.accept("Java is awesome!");
    }
}
```

```

\*\*  Example 3: Constructor Reference\*\*

```
```java
class Message {

    Message(String msg) {
        System.out.println("Message: " + msg);
    }
}
```

```
}

interface MessageCreator {

    Message create(String msg);

}

public class Demo {

    public static void main(String[] args) {

        MessageCreator creator = Message::new; // Constructor reference

        creator.create("Hello, World!");

    }

}

```

```

## 6. \*\*Optional Class\*\*

---

### \*\*Definition:\*\*

The `Optional<T>` class is a container object used to **\*\*avoid null checks\*\*** and **\*\*prevent NullPointerExceptions\*\***.

---

### \*\*💡 Example: Using Optional\*\*

```
```java
import java.util.Optional;

public class Demo {

    public static void main(String[] args) {
        Optional<String> name = Optional.of("Nani");

        // Check if value is present
        if (name.isPresent()) {
            System.out.println(name.get());
        }

        // orElse
        Optional<String> empty = Optional.empty();
        System.out.println(empty.orElse("Default"));

        // ifPresent
        name.ifPresent(n -> System.out.println("Hi " + n));
    }
}

```

```

## 7. \*\*Date and Time API (`java.time`)\*\*

=====

**\*\*Definition:\*\***

Java 8 introduced the `java.time` package to replace legacy `Date` and `Calendar`. It's **immutable**, **thread-safe**, and **easy to use**.

---

**\*\*💡 Example: Basic Usage\*\***

```
```java
```

```
import java.time.*;
```

```
public class Demo {
```

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

```
        LocalDate today = LocalDate.now();
```

```
        LocalTime time = LocalTime.now();
```

```
        LocalDateTime dateTime = LocalDateTime.now();
```

```
        System.out.println("Today: " + today);
```

```
        System.out.println("Time: " + time);
```

```
        System.out.println("DateTime: " + dateTime);
```

```
        LocalDate birthDate = LocalDate.of(1990, 5, 10);
```

```
        Period age = Period.between(birthDate, today);
```

```
        System.out.println("Age: " + age.getYears());
```

```
}
```

```
}
```

```
```
```

## 8. \*\*Collectors\*\*

---

### \*\*Definition:\*\*

`Collectors` are used to gather elements from a stream into a collection (List, Set, Map). It is used with ` `.collect()` terminal operation.

---

### \*\*💡 Example: Collectors Usage\*\*

```
```java
```

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

```
import java.util.stream.Collectors;
```

```
public class Demo {
```

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

```
        List<String> names = Arrays.asList("Nani", "Babu", "Pallapu");
```

```
        List<String> upper = names.stream()
```

```
            .map(String::toUpperCase)
```

```
            .collect(Collectors.toList());
```

```
        System.out.println(upper);
```

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

        System.out.println("Joined: " + joined);

    }

}

```

```

## 9. \*\*Parallel Streams\*\*

---

### \*\*Definition:\*\*

Parallel streams divide the source into multiple parts and process them **concurrently using multiple threads**.

```

```

### \*\*💡 Example: Normal vs Parallel Stream\*\*

```
```java

import java.util.Arrays;

public class Demo {

    public static void main(String[] args) {
        Arrays.asList("A", "B", "C", "D", "E")
            .stream()
```

```
        .forEach(s -> System.out.print(s + " ")); // Sequential      A B C D
E

System.out.println();

Arrays.asList("A", "B", "C", "D", "E")
    .parallelStream()
    .forEach(s -> System.out.print(s + " ")); // Parallel C E D B A

}

}

```

```

> \*\*Note:\*\* Use parallel stream when you have CPU-bound operations or large datasets.

## 10. \*\*Base64 Encoding and Decoding\*\*

---

### \*\*Definition:\*\*

Java 8 introduced `java.util.Base64` to encode and decode binary data in \*\*Base64 format\*\*.

### \*\*💡 Example: Base64 Usage\*\*

```
```java
import java.util.Base64;

public class Demo {
    public static void main(String[] args) {
        String original = "secret123";

        // Encoding
        String encoded =
Base64.getEncoder().encodeToString(original.getBytes());
        System.out.println("Encoded: " + encoded);

        // Decoding
        byte[] decodedBytes = Base64.getDecoder().decode(encoded);
        String decoded = new String(decodedBytes);
        System.out.println("Decoded: " + decoded);
    }
}
```

```

Hashing

=====

Hashing is a technique used to convert input data, such as passwords, into a fixed-size string of characters, called a hash value, using a hashing algorithm.

In Spring Boot projects, password hashing is crucial for securely storing passwords in databases.

It is a **\*\*one-way process\*\***, meaning you cannot convert the hash back to the original value.

BCrypt is a widely-used hashing algorithm specifically designed for securely hashing passwords.

It incorporates a salt (randomly generated data) and a cost factor (iteration count) to make the hashing process computationally expensive, which increases the difficulty of brute-force attacks.

Hashing is commonly used for:

- Password storage
- Data integrity checks
- Digital signatures

Characteristics of Hashing:

- One-way: cannot reverse a hash to get the original data
- Deterministic: same input always gives same output
- Fixed size: output length is constant
- Collision-resistant: different inputs should not result in the same hash
- Fast computation: hash is generated quickly

## Example in Java using BCrypt

---

Below is a simple Java example of hashing a password using BCrypt:

```
```java
```

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-core</artifactId>
</dependency>

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

public class HashingExample {

    public static void main(String[] args) {
        String originalPassword = "mySecret123";

        BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();

        // Hash the password
        String hashedPassword = encoder.encode(originalPassword);

        System.out.println("Original: " + originalPassword);
        System.out.println("Hashed: " + hashedPassword);
    }
}
```

```
// Verify password  
  
boolean isMatch = encoder.matches("mySecret123", hashedPassword);  
  
System.out.println("Password Match: " + isMatch);  
  
}  
  
}  
  
---
```

Sample Output:

```
-----  
Original: mySecret123  
Hashed: $2a$10$AfJ5ZL7RzOpI1qyt0xKs3e3XfJZZcNAIZ3dpG4F0vTfrMz4v8XwPO  
Password Match: true
```

Why Hashing is Important for Passwords:

- ```

- Even if someone gets access to the database, they can't see the real passwords.
- Helps protect users who use the same password across different sites.
- BCrypt automatically adds salt and increases the complexity of brute-force attacks.
```

---

HASHCODE

```
=====
```

In Java, the `hashCode()` method is defined in the \*\*`Object`\*\* class.

It returns an integer value (called the hash code), which is used to uniquely identify objects—especially when they're stored in **hash-based data structures** like:

- `HashMap`
- `HashSet`
- `Hashtable`

This hash code is a numerical representation of the object's memory address or some other unique identifier associated with the object.

The purpose of `hashCode()` is to make searching and storing objects faster and more efficient.

---

#### ### ◆ Example: Using `hashCode()` with a HashMap

```
```java
class Employee {

    private int id;

    public Employee(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }
}
```

```
@Override  
public int hashCode() {  
    return id;  
}  
}  
...  
  
### ◆ Using the `Employee` class in a `HashMap`
```

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

public class Main {
 public static void main(String[] args) {
 Map<Integer, Employee> employeeMap = new HashMap<>();

 Employee employee1 = new Employee(101);
 Employee employee2 = new Employee(102);

 // Storing employees using their IDs as keys
 employeeMap.put(employee1.getId(), employee1);
 employeeMap.put(employee2.getId(), employee2);

 // Searching for an employee by ID
```

```

 int searchId = 101;

 Employee foundEmployee = employeeMap.get(searchId);

 if (foundEmployee != null) {
 System.out.println("Employee found: " +
 foundEmployee.getId());
 } else {
 System.out.println("Employee not found with id: " +
 searchId);
 }
 }

 ...

```

---

#### #### ◆ What's Happening Here?

1. \*\*`hashCode()`\*\* returns the `id` of the employee.

This makes it a simple and unique identifier for each object.

2. \*\*The `HashMap` uses `id` as the key\*\*.

When we call `employeeMap.get(101)`, the `HashMap`:

- Calculates the hash using the key ('101')
- Quickly jumps to the bucket corresponding to that hash
- Uses `equals()` to confirm the match and return the correct `Employee`

---

### #### ◆ Why is `hashCode()` Important?

- It helps \*\*quickly locate the bucket\*\* where the object is stored.
- A well-designed `hashCode()` improves the \*\*performance of search operations\*\*.
- Without it, objects may not behave correctly in hash-based collections.

---

Collections in Java:

=====

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

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

Java Collection means a single unit of objects.

Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

### #### Difference Between Collection and Collections

- \*\*Collection\*\* :

→ is used to represent a single unit with a group of individual objects.  
→ It's the root interface for all collection types like \*\*List\*\*, \*\*Set\*\*, and \*\*Queue\*\*.

- \*\*Collections\*\* :

→ is used to operate on collections with several \*\*utility methods\*\*.  
→ It's a \*\*final class\*\* that contains static methods like `sort()`, `reverse()`, `min()`, `max()`, `shuffle()`, etc.

### Iterable Interface

---

The Iterable interface is the root interface for all the collection classes.

The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

`Iterator<T> iterator()`

### Iterator interface

---

Iterator interface provides the facility of iterating the elements in a forward direction only.

1      `public boolean hasNext()`      It returns true if the iterator has more elements otherwise it returns false.

2      `public Object next()`      It returns the element and moves the cursor pointer to the next element.

3      `public void remove()`      It removes the last elements returned by the iterator. It is less used.

There are various ways to traverse the collection elements:

- > By Iterator interface.
- > By for-each loop.
- > By ListIterator interface.
- > By for loop.
- > By forEach() method.
- > By forEachRemaining() method.

Note : if you want to iterate element in a forward direction or backward direction. You can use ListIterator

```
List<String> myList = new ArrayList<>();
// Add elements to myList
ListIterator<String> listIterator = myList.listIterator(myList.size());
while (listIterator.hasPrevious()) {
 String element = listIterator.previous();
 // Process element
}
```

Note : using forEachRemaining() method to iterate elements or print elements

```
Iterator<String> iterator = myList.iterator();
iterator.forEachRemaining(element -> {
 System.out.println(element);
});
```

## 1. List Interface

---

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

Null elements are not allowed.

To instantiate the List interface, we must use :

```
List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();
```

### i. ArrayList:

---

- 1) ArrayList internally uses a dynamic array(can grow or shrink in size) to store the elements.
- 2) Manipulation with ArrayList is slow because it internally uses an array.
- 3) An ArrayList class can act as a list only because it implements List only.
- 4) ArrayList is better for storing and accessing data.

5) The memory location for the elements of an ArrayList is contiguous(elements that are next to each other).

6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList.

7) To be precise, an ArrayList is a resizable array.

--> ArrayList and LinkedList both implement the List interface and maintain insertion order. Both are non-synchronized classes.

## ii. LinkedList

=====

Note : Each element in the list is represented by a Node object. Each Node contains a reference to the previous node (prev) and the next node (next), along with the actual data stored in the middle node.

null <- [Prev|Data|Next] <-> [Prev|Data|Next] <-> [Prev|Data|Next] -->  
null

1) LinkedList internally uses a doubly linked list to store the elements.

2) If any element is removed from the array, all the other elements are shifted in memory. Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.

3) LinkedList class can act as a list and queue both because it implements List and Deque interfaces.

4) LinkedList is better for manipulating data.

5) The location for the elements of a linked list is not contagious.

6) There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized.

7) LinkedList implements the doubly linked list of the list interface.

8) This allows forward and backward traversal.

```

// Access elements

cities.getFirst();
cities.getLast();

// Add at beginning and end

cities.addFirst("Delhi");
cities.addLast("Mumbai");

```

---

### ### ✓ \*\*ArrayList vs LinkedList – Comparison Table\*\*

	Feature	**ArrayList**
**LinkedList**		
	**Underlying Structure**	Dynamic array (contiguous memory)
Doubly linked nodes (each has `prev` and `next` pointers)		
	**Access by Index**	<span style="color: green;">✓</span> Fast (`O(1)`) – Direct index access
<span style="color: red;">✗</span> Slow ( $O(n)$ ) – Traverses from head/tail		
	**Insertion at End**	<span style="color: green;">✓</span> Fast ( $O(1)$ usually, $O(n)$ if resizing)
<span style="color: green;">✓</span> Fast ( $O(1)$ )		
	**Insertion at Middle/Start**	<span style="color: red;">✗</span> Slow – Shifts elements ( $O(n)$ )
<span style="color: green;">✓</span> Fast ( $O(1)$ if node known, otherwise $O(n)$ )		

		<b>Deletion at Middle/Start</b>   <span style="color:red">X</span> Slow – Shifts elements (`O(n)`)
<span style="color:green">✓</span> Fast – Just update pointers (`O(1)` if node known)		
	<b>Memory Overhead</b>	<span style="color:green">✓</span> Low – Only stores values
<span style="color:red">X</span> High – Stores values + pointers to prev/next		
memory	<b>Iteration Performance</b>	<span style="color:green">✓</span> Faster – Data is in contiguous
		<span style="color:red">X</span> Slower – Data is scattered in heap
	<b>Use Case Example</b>	Best for <b>searching/filtering</b> , like viewing orders or claims list
		Best for <b>queues/workflows</b> , like processing order queues or Frequent Insert/Cancel in Middle or claim steps

---

### ###💡 Summary Tips:

- Use **`ArrayList`** when:

- You mostly read data.
- You need random access (`list.get(i)`).
- You filter/search frequently.

- Use **`LinkedList`** when:

- You frequently add/remove from start/middle.
- You're building a queue or stack (FIFO/LIFO).
- You don't need fast access by index.

=====

- 1) Vector is like the dynamic array which can grow or shrink its size.
- 2) Unlike array, we can store n-number of elements in it as there is no size limit.
- 3) It is a part of Java Collection framework since Java 1.2. It is found in the `java.util` package and implements the List interface, so we can use all the methods of List interface here.
- 4) It is recommended to use the Vector class in the thread-safe implementation only.
- 5) If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.
- 6) The Iterators returned by the Vector class are fail-fast.
- 7) In case of concurrent modification, it fails and throws the `ConcurrentModificationException`.
- 8) Maintains insertion order

It is similar to the ArrayList, but with two differences-

- a. Vector is synchronized.
- b. Java Vector contains many legacy methods that are not the part of a collections' framework.

#### What is Thread-Safe?

Thread-safe means that a piece of code or a class can be safely used by multiple threads at the same time without causing errors or data corruption.

#### Why is this important?

In a multi-threaded environment, multiple threads may try to access and modify shared resources (like a variable or collection) simultaneously.

If access is not managed properly, you may face issues like:

Inconsistent data

Race conditions

Program crashes

 Example (Non-thread-safe):

```
List<Integer> list = new ArrayList<>();
```

```
// Thread 1 adds 1
```

```
// Thread 2 adds 2 at the same time
```

// If both threads access the list at the same time without synchronization, it may corrupt the list.

 Example (Thread-safe):

```
Vector<Integer> vector = new Vector<>();
```

```
// Thread 1 adds 1
```

```
// Thread 2 adds 2
```

// Vector is synchronized internally, so it handles multiple threads accessing it at the same time safely.

#### iv. Stack

=====

1) The stack is a linear data structure that is used to store the collection of objects.

2) It is based on Last-In-First-Out (LIFO). Java collection framework provides many interfaces and classes to store the collection of objects.

3) One of them is the Stack class that provides different operations such as push, pop, search, etc.

4) In this section, we will discuss the Java Stack class, its methods, and implement the stack data structure in a Java program.

5) The stack data structure has the two most important operations that are push and pop.

6) The push operation inserts an element into the stack and pop operation removes an element from the top of the stack.

7) Maintains insertion order

Operation   Description
-------------------------

push()   Add an element on top of the stack
---------------------------------------------

pop()   Remove and return the top element
-------------------------------------------

peek()   Return the top element without removing it
-----------------------------------------------------

isEmpty()   Check if stack is empty
-------------------------------------

search()   Returns position of an element from the top (1-based)
------------------------------------------------------------------

## 2. Queue Interface

=====

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed.

There are various classes and interfaces like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated.

```
Queue<String> q1 = new PriorityQueue();
```

```
Queue<String> q2 = new ArrayDeque();
```

#### i. PriorityQueue

=====

1) PriorityQueue is also class that is defined in the collection framework that gives us a way for processing the objects on the basis of priority.

2) It is already described that the insertion and deletion of objects follows FIFO pattern in the Java queue.

3) However, sometimes the elements of the queue are needed to be processed according to the priority, that's where a PriorityQueue comes into action.

4) PriorityQueue uses offer() method to add objects and peek method to return high priority elements (usually lowest numbers have highest priority).

5) Null elements are not allowed.

use case :

Orders are given different priorities: HIGH, MEDIUM, LOW.

High-priority (rush) orders are processed first.

#### 3. Deque Interface

=====

The interface called Deque is present in java.util package.

It is the subtype of the interface queue. The Deque supports the addition as well as the removal of elements from both ends of the data structure.

Therefore, a deque can be used as a stack or a queue.

We know that the stack supports the Last In First Out (LIFO) operation, and the operation First In First Out is supported by a queue.

As a deque supports both, either of the mentioned operations can be performed on it. Deque is an acronym for "double ended queue".

#### i. ArrayDeque class

---

1) We know that it is not possible to create an object of an interface in Java.

2) Therefore, for instantiation, we need a class that implements the Deque interface, and that class is ArrayDeque.

3) It grows and shrinks as per usage. It also inherits the AbstractCollection class.

4) The important points about ArrayDeque class are:

a) Unlike Queue, we can add or remove elements from both sides.

b) Null elements are not allowed in the ArrayDeque.

c) ArrayDeque is not thread safe, in the absence of external synchronization.

d) ArrayDeque has no capacity restrictions.

e) ArrayDeque is faster than LinkedList and Stack.

Example :

---

```
ArrayDeque<Integer> deque = new ArrayDeque<>();
```

```
// Adding elements to the deque from both sides
```

```
deque.addFirst(1); // Adding element at the beginning
deque.addLast(2); // Adding element at the end
deque.addFirst(0); // Adding element at the beginning
deque.addLast(3); // Adding element at the end

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

// Removing elements from both sides
int firstElement = deque.removeFirst(); // Removing element from the
beginning
int lastElement = deque.removeLast(); // Removing element from the
end
```

real time use cases :

- a. Palindrome Checking :You can use a deque to check if a string is a palindrome by comparing characters from both ends.
- b. Undo Operations: Deques can be used to store previous states of an application for undo/redo functionality.

#### 4. Set Interface

=====

Set Interface in Java is present in `java.util` package.

It extends the `Collection` interface.

It represents the unordered set of elements which doesn't allow us to store the duplicate items.

We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

```
Set<data-type> s1 = new HashSet<data-type>();
Set<data-type> s2 = new LinkedHashSet<data-type>();
Set<data-type> s3 = new TreeSet<data-type>();
```

i. HashSet Class

=====

1) Java HashSet class is used to create a collection that uses a hash table for storage.

2) It inherits the AbstractSet class and implements Set interface.

3) The important points about Java HashSet class are:

a) HashSet stores the elements by using a mechanism called hashing.

b) HashSet contains unique elements only.

c) HashSet allows null value.

d) HashSet class is non synchronized.

e) HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashCode.

f) HashSet is the best approach for search operations.

g) The initial default capacity of HashSet is 16, and the load factor is 0.75.

## ii. LinkedHashSet Class

---

- 1) Java LinkedHashSet class is a Hashtable and Linked list implementation of the Set interface.
- 2) It inherits the HashSet class and implements the Set interface.
- 3) The important points about the Java LinkedHashSet class are:
  - a) Java LinkedHashSet class contains unique elements only like HashSet.
  - b) Java LinkedHashSet class provides all optional set operations and permits null elements.
  - c) Java LinkedHashSet class is non-synchronized.
  - d) Java LinkedHashSet class maintains insertion order.

## iii. TreeSet Class

---

- 1) Java TreeSet class implements the Set interface that uses a tree for storage.
- 2) It inherits AbstractSet class and implements the NavigableSet interface.
- 3) The objects of the TreeSet class are stored in ascending order.
- 4) The important points about the Java TreeSet class are:
  - a) Java TreeSet class contains unique elements only like HashSet.
  - b) Java TreeSet class access and retrieval times are quiet fast.
  - c) Java TreeSet class doesn't allow null element.
  - d) Java TreeSet class is non synchronized.
  - e) Java TreeSet class maintains ascending order.
  - f) Java TreeSet class contains unique elements only like HashSet.

#### iv. ConcurrentHashSet

---

1. Java ConcurrentHashSet is not a standard class in the Java Collections Framework. However, it can be implemented using `ConcurrentHashMap`.
2. It provides a thread-safe set implementation suitable for concurrent access from multiple threads.

#### 3. Key Features:

- a. It is based on the ConcurrentHashMap implementation.
- b. ConcurrentHashSet can be created by using ConcurrentHashMap's keySet() method.
- c. It inherits the concurrency characteristics of ConcurrentHashMap, allowing concurrent read and write access without external synchronization.
- d. Like ConcurrentHashMap, it supports high concurrency, scalability, and performance in multi-threaded environments.
- e. ConcurrentHashSet does not allow duplicate elements.
- f. It supports null elements.
- g. The behavior and characteristics of ConcurrentHashSet are similar to ConcurrentHashMap.

### 5 Map Interface

---

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry.

A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap.

A Map can't be traversed, so you need to convert it into Set using keySet() or entrySet() method.

Class	Description
-------	-------------

HashMap	is the implementation of Map, but it doesn't maintain any order.
---------	------------------------------------------------------------------

LinkedHashMap	is the implementation of Map. It inherits HashMap class. It maintains insertion order.
---------------	----------------------------------------------------------------------------------------

TreeMap	is the implementation of Map and SortedMap. It maintains ascending order.
---------	---------------------------------------------------------------------------

i. HashMap

---

1) Java HashMap class hierarchy

2) Java HashMap class implements the Map interface which allows us to store key and value pair, where keys should be unique.

3) If you try to insert the duplicate key, it will replace the element of the corresponding key.

4) It is easy to perform operations using the key index like updation, deletion, etc.

5) HashMap class is found in the java.util package.

6) HashMap in Java is like the legacy Hashtable class, but it is not synchronized.

7) It allows us to store the null elements as well, but there should be only one null key.

8) Since Java 5, it is denoted as `HashMap<K,V>`, where K stands for key and V for value.

9) It inherits the `AbstractMap` class and implements the `Map` interface.

a) Java `HashMap` contains values based on the key.

b) Java `HashMap` contains only unique keys.

c) Java `HashMap` may have one null key and multiple null values.

d) Java `HashMap` is non synchronized.

e) Java `HashMap` maintains no order.

f) The initial default capacity of Java `HashMap` class is 16 with a load factor of 0.75.

## ii. Hashtable

---

1. Java `Hashtable` class implements a hashtable, which maps keys to values.

2. It extends `Dictionary` class and implements the `Map` interface and `Cloneable`, `Serializable` interfaces.

a. A `Hashtable` is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the `hashCode()` method. A `Hashtable` contains values based on the key.

b. Java `Hashtable` class contains unique elements.

c. Java `Hashtable` class doesn't allow null key or value.

d. Java `Hashtable` class is synchronized.

e. The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

iii. LinkedHashMap class

=====

1) Java LinkedHashMap class hierarchy

2) Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

3) Points to remember:

- a) Java LinkedHashMap contains values based on the key.
- b) Java LinkedHashMap contains unique elements.
- c) Java LinkedHashMap may have one null key and multiple null values.
- d) Java LinkedHashMap is non synchronized.
- e) Java LinkedHashMap maintains insertion order.
- f) The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

```
public class LinkedHashMap<K,V> extends HashMap<K,V> implements
Map<K,V>
```

iv) TreeMap class

=====

Java TreeMap class hierarchy: TreeMap class implements SortedMap interface and SortedMap interface extends Map interface.

Java TreeMap class is a red-black tree based implementation.

It provides an efficient means of storing key-value pairs in sorted order.

The important points about Java TreeMap class are:

Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.

Java TreeMap contains only unique elements.

Java TreeMap cannot have a null key but can have multiple null values.

Java TreeMap is non synchronized.

Java TreeMap maintains ascending order.

#### iv. ConcurrentHashMap

---

1. Java ConcurrentHashMap class implements a concurrent hash table, which maps keys to values.

2. It is part of the `java.util.concurrent` package.

3. ConcurrentHashMap is designed for concurrent access from multiple threads without external synchronization.

4. It extends the AbstractMap class and implements the ConcurrentMap interface, which extends the Map interface.

#### 5. Key Features:

a. Internally, ConcurrentHashMap is divided into segments or buckets, each of which can be locked independently for updates.

b. It provides high concurrency for read and write operations.

c. Unlike Hashtable, it allows concurrent updates without blocking the entire map.

d. It supports high scalability by allowing concurrent updates to different segments.

- e. It provides better performance compared to Hashtable in multi-threaded scenarios.
- f. ConcurrentHashMap class doesn't allow null key or value.
- g. The default initial capacity of ConcurrentHashMap is 16, and the default concurrency level is 16.

## JDBC IN JAVA

---

JDBC, which stands for Java Database Connectivity, is a Java-based API (Application Programming Interface) that allows Java applications to interact with relational databases.

It provides a standard interface for connecting to databases, executing SQL queries, and retrieving and manipulating data.

JDBC is a crucial technology for database-driven Java applications and is part of the Java Standard Library.

Here are some key points about JDBC:

1. **Database Connectivity**: JDBC enables Java applications to establish connections to relational databases like MySQL, Oracle, PostgreSQL, SQL Server, and others. It provides a way for Java code to communicate with databases through drivers.

2. **Driver Types**:

- JDBC drivers are used to establish a connection between the Java application and the database. There are four types of JDBC drivers:

- Type-1 (JDBC-ODBC bridge)
- Type-2 (Native-API driver)
- Type-3 (Network Protocol driver)
- Type-4 (Thin driver, also known as a native-protocol driver)

- The most common and recommended driver type is Type-4 (Thin driver) because it doesn't require a native database client.

3. **Database Operations**: JDBC allows you to perform various database operations, including executing SQL queries, retrieving and updating data, and managing transactions.

#### 4. **Basic Steps**:

- To use JDBC, you typically follow these basic steps:
  1. Load the JDBC driver (if not using the Type-4 driver, you need to load the appropriate driver class).
  2. Establish a database connection using a connection URL, username, and password.
  3. Create a statement or prepared statement for executing SQL queries.
  4. Execute SQL queries to retrieve, insert, update, or delete data.
  5. Process the results, if any, from the database.
  6. Close the database connection when done.

5. **Exception Handling**: JDBC methods can throw various exceptions, including `SQLException`. Proper exception handling is important to deal with potential errors during database operations.

6. **Batch Processing**: JDBC supports batch processing, which allows you to execute multiple SQL statements in a single batch, reducing the overhead of repeated database communication.

7. **\*\*Transaction Management\*\***: You can use JDBC to manage database transactions, including committing or rolling back changes to ensure data consistency.

8. **\*\*Connection Pooling\*\***: In real-world applications, connection pooling is often used to efficiently manage database connections and improve performance.

Here's a basic example of how to use JDBC to connect to a database and execute a simple query in Java:

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

public class JDBCDemo {
    public static void main(String[] args) {
        String jdbcURL = "jdbc:mysql://localhost:3306/mydatabase";
        String username = "yourUsername";
        String password = "yourPassword";

        try {
            // Load the JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");
            // Establish a connection to the database

```

```
        Connection connection =
DriverManager.getConnection(jdbcURL, username, password);

        // Create a statement
        Statement statement = connection.createStatement();

        // Execute a query
        String sqlQuery = "SELECT * FROM employees";
        ResultSet resultSet = statement.executeQuery(sqlQuery);

        // Process the results
        while (resultSet.next()) {
            System.out.println("Name: " +
resultSet.getString("name"));
        }

        // Close resources
        resultSet.close();
        statement.close();
        connection.close();

    } catch (Exception e) {
        e.printStackTrace();
    }
}

}

```

```

JDBC provides a powerful and standardized way for Java applications to interact with databases, making it a fundamental technology for database-driven Java applications and enterprise systems.

## MULTITHREADING

---

**Thread :** A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

**Multithreading :** Multithreading in java is process of executing multiple threads simultaneously.

Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area.

They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

### 1) Process-based Multitasking (Multiprocessing) :

Each process has an address in memory. In other words, each process allocates a separate memory area.

A process is heavyweight.

Cost of communication between the process is high.

Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

### 2) Thread-based Multitasking (Multithreading) :

Threads share the same address space.

A thread is lightweight.

Cost of communication between the thread is low.

Java Thread class : Java provides Thread class to achieve thread programming.

Thread class provides constructors and methods to create and perform operations on a thread.

Thread class extends Object class and implements Runnable interface.

### Life cycle of a Thread (Thread States)

---

In Java, a thread always exists in any one of the following states. These states are:

In Java, the lifecycle of a thread can be represented in a similar manner to what you've outlined:

i. **\*\*New:\*\*** The thread has been instantiated using the `new` keyword, but it hasn't yet started its execution. At this stage, the thread object has been created, but the `start()` method hasn't been called yet.

ii. **\*\*Runnable/Active:\*\*** After calling the `start()` method on the thread object, it enters the runnable state. It's ready to run but the operating system or the JVM hasn't yet selected it to run or scheduled it for execution.

a. **Running :** When a thread is selected by the thread scheduler and is actively executing its tasks, it's said to be in the "running" state.

iii. **\*\*Blocked/Waiting:\*\*** Sometimes, a thread might need to pause its execution temporarily, often because it's waiting for a resource such as I/O operation to complete or for a lock on an object. When a thread is in this state, it's blocked or waiting.

iv. **\*\*Timed Waiting:\*\*** This state is similar to the blocked/waiting state, but the thread is waiting for a specified duration before resuming execution. This can happen, for example, when using methods like `Thread.sleep()` or `Object.wait(timeout)`.

v. **\*\*Terminated:\*\*** A thread enters the terminated state when it completes its task or is stopped forcefully. Once a thread is terminated, it cannot be started again. It's important to note that a thread can also terminate due to an uncaught exception.

Here's a Java code example illustrating these states:

```
```java
```

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

```
// 1. NEW State: Thread object created but  
not started
```

```
Thread thread = new Thread(() -> {  
    // 3. RUNNABLE / RUNNING State  
    System.out.println("3. Thread is  
executing...");
```

```
try {  
    Thread.sleep(2000); //  
    Simulate some work (TIMED_WAITING)  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}  
  
System.out.println("3. Thread  
execution completed.");  
});
```

```
System.out.println("1. Thread is in NEW  
state (created but not started).");
```

```
// 2. STARTED -> RUNNABLE State  
thread.start();  
System.out.println("2. Thread is in  
RUNNABLE state (started and eligible to run).");
```

```
try {
```

```

// 4. WAITING State: Main thread
will be waiting for 'thread' to finish

        thread.join();

    } catch (InterruptedException e) {
        e.printStackTrace();
    }

}

// 5. TERMINATED State

System.out.println("5. Thread is in
TERMINATED state (finished execution)."); // this only prints after thread is completed.

}

```

Output :

- 1.Thread is in NEW state (created but not started).
- 2.Thread is in RUNNABLE state (started and eligible to run).
- 3.Thread is executing...
- 4.Thread execution completed.
- 5.Thread is in TERMINATED state (finished execution).

In Java, there are two main ways to create a thread:

1. Extending the `Thread` class:

- You can create a new class that extends the `Thread` class and overrides its `run()` method, where you define the task that the thread will execute.

- Here's an example:

```
```java
class MyThread extends Thread {

 public void run() {
 // Define the task that this thread will execute
 System.out.println("Thread is running");
 }
}

public class Main {

 public static void main(String[] args) {
 // Create an instance of your custom thread class
 MyThread myThread = new MyThread();

 // Start the thread
 myThread.start();
 }
}
```

```

2. Implementing the `Runnable` interface:

- You can create a class that implements the `Runnable` interface and provides implementation for its `run()` method.
- Then, you can pass an instance of this class to the constructor of a `Thread` object.

- Here's an example:

```
```java

class MyRunnable implements Runnable {

 public void run() {

 // Define the task that this thread will execute

 System.out.println("Thread is running");

 }

}

public class Main {

 public static void main(String[] args) {

 // Create an instance of your custom runnable class

 MyRunnable myRunnable = new MyRunnable();

 // Create a thread with your runnable object

 Thread thread = new Thread(myRunnable);

 // Start the thread

 thread.start();

 }

}

```

```

1. Extending 'Thread' class creates a thread with the task defined in the subclass itself. Limited flexibility in class inheritance, as your class cannot extend any other class. Useful for quick, standalone thread implementations

2. Implementing `Runnable` interface separates the task from the thread, allowing flexibility in class hierarchy and promoting better design practices (Multiple Inheritance).

Certainly! Let's break down each concept related to threading in Java:

1. **Sleep**: `Thread.sleep()` is a method that pauses the execution of the current thread for a specified amount of time, allowing other threads to execute.

2. **Yield**: `Thread.yield()` method in Java is used to pause the execution of the currently running thread and give the opportunity for other threads of the same priority to run.

3. **Wait, Notify, and NotifyAll**:

- `wait()`: Causes the current thread to wait until another thread invokes the `notify()` or `notifyAll()` method for the same object.

- `notify()`: Wakes up a single thread that is waiting on the object's monitor.

- `notifyAll()`: Wakes up all threads that are waiting on the object's monitor.

- These methods are used for inter-thread communication and synchronization.

4. **Join**: The `join()` method is used to wait for a thread to die. It allows one thread to wait for the completion of another.

5. **Thread Pool**: A thread pool is a collection of pre-initialized threads that stand by, ready to be given work. This approach reduces thread creation overhead and manages the execution of tasks efficiently.

6. **Thread Group**: A thread group represents a collection of threads. It allows you to set properties such as the maximum priority of its threads and to control the threads within the group as a single unit.

7. **Synchronization**:

- **Method Synchronization**: Using the `synchronized` keyword on a method allows only one thread at a time to execute the synchronized method.
- **Block Synchronization**: Using synchronized blocks allows more granular control over the synchronization by locking on specific objects.
- **Static Synchronization**: Synchronizing static methods locks the class, preventing multiple threads from executing static synchronized methods simultaneously.

8. **Deadlock**: A deadlock occurs when two or more threads are blocked forever, each waiting for the other to release a resource. It's a common synchronization problem that needs to be carefully avoided.

9. **Inter-Thread Communication**: Inter-thread communication refers to communication between threads. It involves methods like `wait()`, `notify()`, and `notifyAll()`, which allow threads to coordinate their actions and share data safely.

Some Important concepts in Threads in java

1. Runnable Interface:

- **Purpose**: The `Runnable` interface is used to represent a task that can be executed asynchronously by a thread.
- **Why Use**: It provides a way to encapsulate code that needs to run concurrently without directly dealing with threads.
- **Methods**: It has a single method `run()` that contains the code to be executed by the thread.
- **Example**:

```
```java
public class MyRunnableTask implements Runnable {

 @Override
 public void run() {
 // Task logic goes here
 System.out.println("Runnable task executed");
 }
}
```

### ### 2. Callable Interface:

- **Purpose**: Similar to `Runnable`, but can return a result or throw an exception.
- **Why Use**: Useful when you need to get a result or handle exceptions from the task.
- **Methods**: It has a single method `call()` that contains the code to be executed by the thread.
- **Example**:

```
```java
import java.util.concurrent.Callable;

public class MyCallableTask implements Callable<String> {

    @Override
    public String call() throws Exception {
        // Task logic goes here
    }
}
```

```

        return "Callable task executed";
    }

}
```

```

### ### 3. Executors Class:

- **Purpose**: Provides utility methods for creating different types of thread pools.
- **Why Use**: Simplifies the process of managing threads and executing tasks asynchronously.
- **Methods**: Various factory methods for creating different types of thread pools, such as `newFixedThreadPool()`, `newSingleThreadExecutor()`, etc.
- **Example**:

```

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

ExecutorService executor = Executors.newFixedThreadPool(5);
executor.execute(new MyRunnableTask());
```

```

### ### 4. Future Interface:

- **Purpose**: Represents the result of an asynchronous computation.
- **Why Use**: Allows checking if the computation is complete, retrieving the result, or canceling the computation.
- **Methods**: Methods like `get()`, `isDone()`, `cancel()`, etc.

- **Example**:

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

ExecutorService executor = Executors.newSingleThreadExecutor();
Future<String> future = executor.submit(new MyCallableTask());
String result = future.get(); // Blocks until the task is complete
````
```

These concepts are fundamental in concurrent programming in Java and are extensively used when working with threads and asynchronous tasks. They provide flexibility, simplicity, and control over concurrent execution in Java applications.

## SOME IMPORTANT INTERFACES IN JAVA

---

### Comparable and Comparator interfaces in JAVA

---

Java provides two interfaces to sort objects using data members of the class:

1. Comparable

## 2. Comparator

### 1. Comparable interface

```
=====
```

A comparable object is capable of comparing itself with another object. The class itself must implements the `java.lang.Comparable` interface to compare its instances.

Consider a `Movie` class that has members like, rating, name, year. Suppose we wish to sort a list of `Movies` based on year of release.

We can implement the Comparable interface with the `Movie` class, and we override the method `compareTo()` of Comparable interface.

### example for Comparable

```
=====
```

```
package mypractice.com.comparable;
```

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

```
import java.util.Collections;
```

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

```
class Movie implements Comparable<Movie> {
```

```
 private String name;
```

```
 private String release;
```

```
 private String rating;
```

```
public Movie() {
}

public Movie(String name, String release, String rating) {

 this.name = name;
 this.release = release;
 this.rating = rating;

}

public void setName(String name) {

 this.name = name;

}

public String getName() {

 return name;

}

public void setRelease(String release) {

 this.release = release;
}
```

```
 }

 public String getRelease() {

 return release;
 }

 public void setRating(String rating) {

 this.rating = rating;
 }

 public String getRating() {

 return rating;
 }

 @Override

 public String toString() {

 return "Movie {" + "name = '" + name + "'" + ", release = '" + release + "'"
 + ", rating = '" + rating + "'" + '}';
 }

 @Override

 public int compareTo(Movie movie) {

 return Integer.compare(Integer.parseInt(this.release),
Integer.parseInt(movie.release));
 }
}
```

```
 }

}

public class Main {

 public static void main(String args[]) {

 List<Movie> list = new ArrayList<>();

 list.add(new Movie("Bahubali", "2015", "9.3"));

 list.add(new Movie("Saaho", "2019", "5.5"));

 list.add(new Movie("Varsham", "2005", "7.3"));

 list.add(new Movie("Saalar", "2023", "6.3"));

 list.add(new Movie("Darling", "2010", "8.3"));

 Collections.sort(list);

 list.forEach(movie -> System.out.println(movie.toString()));

 }
}
```

Comparator

=====

Unlike Comparable, Comparator is external to the element type we are comparing. It's a separate class. We create multiple separate classes (that implement Comparator) to compare by different members.

Collections class has a second sort() method and it takes Comparator. The sort() method invokes the compare() to sort objects.

To compare movies by Rating, we need to do 3 things:

1. Create a class that implements Comparator (and thus the compare() method that does the work previously done by compareTo()).
2. Make an instance of the Comparator class.
3. Call the overloaded sort() method, giving it both the list and the instance of the class that implements Comparator.

#### Example for Comparator

---

```
package mypractice.com.imp_interfaces_in_java.comparator;
```

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

```
/**
```

\* Unlike Comparable, Comparator is external to the element type we are comparing. It's a separate class.

- \* We create multiple separate classes (that implement Comparator) to compare by different members.

- \* Collections class has a second sort() method, and it takes Comparator.

- \* The sort() method invokes the compare() to sort objects.

- \*

- \* To compare movies by Rating, we need to do 3 things:

- \*

- \* 1. Create a class that implements Comparator (and thus the compare() method that does the work previously done by compareTo()).

- \* 2. Make an instance of the Comparator class.

- \* 3. Call the overloaded sort() method, giving it both the list and the instance of the class that implements Comparator.

- \*/

```
class Movie{
```

```
 private String name;
```

```
 private String release;
```

```
 private String rating;
```

```
 public Movie() {
```

```
 }
```

```
 public Movie(String name, String release, String rating) {
```

```
 this.name = name;
```

```
 this.release = release;
```

```
 this.rating = rating;
```

```
}

public void setName(String name) {

 this.name = name;

}

public String getName() {

 return name;

}

public void setRelease(String release) {

 this.release = release;

}

public String getRelease() {

 return release;

}
```

```
public void setRating(String rating) {
 this.rating = rating;
}

public String getRating() {
 return rating;
}

@Override
public String toString() {
 return "Movie{" +
 "name=\"" + name + '\"' +
 ", release=\"" + release + '\"' +
 ", rating=\"" + rating + '\"' +
 '}';
}
}

class RatingSort implements Comparator<Movie>{

 @Override
 public int compare(Movie movie1, Movie movie2){
 return Double.compare(Double.parseDouble(movie1.getRating()),
 Double.parseDouble(movie2.getRating()));
 }
}
```

```
class NameSort implements Comparator<Movie>{

 @Override

 public int compare(Movie movie1, Movie movie2){

 return movie1.getName().compareTo(movie2.getName());
 }
}

public class MainComparator {

 public static void main(String[] args){

 List<Movie> list = new ArrayList<>();

 list.add(new Movie("Bahubali", "2015", "9.3"));

 list.add(new Movie("Saaho", "2019", "5.5"));

 list.add(new Movie("Varsham", "2005", "7.3"));

 list.add(new Movie("Saalar", "2023", "6.3"));

 list.add(new Movie("Darling", "2010", "8.3"));

 System.out.println("===== SORTING BASED ON RATING =====");

 RatingSort ratingSort = new RatingSort();

 Collections.sort(list, ratingSort); // list.sort(ratingSort);

 list.forEach(movie -> System.out.println(movie.toString()));

 System.out.println("===== SORTING BASED ON NAME =====");

 NameSort nameSort = new NameSort();
 }
}
```

```
Collections.sort(list, nameSort);

list.forEach(movie -> System.out.println(movie.toString()));

}

}
```

NOTE :

=====

Comparable is meant for objects with natural ordering which means the object itself must know how it is to be ordered.

For example Roll Numbers of students. Whereas, Comparator interface sorting is done through a separate class.

Logically, Comparable interface compares “this” reference with the object specified and Comparator in Java compares two different class objects provided.

If any class implements Comparable interface in Java then collection of that object either List or Array can be sorted automatically by using Collections.sort() or Arrays.sort() method and objects will be sorted based on their natural order defined by compareTo method.

A basic differentiating feature is that using comparable we can use only one comparison. Whereas, we can write more than one custom comparators as you want for a given type, all using different interpretations of what sorting means. Like in the comparable example we could just sort by only one attribute, i.e., year but in the comparator, we were able to use different attributes like rating, name, and year as well.

To summarize, if sorting of objects needs to be based on natural order then use Comparable whereas if you sorting needs to be done on attributes of different objects, then use Comparator in Java.

## Serializable Interface

---

The Serializable interface is present in java.io package. It is a marker interface. A Marker Interface does not have any methods and fields. Thus classes implementing it do not have to implement any methods.

The Serializable interface in Java is a marker interface that indicates that the objects of the class implementing it can be serialized.

Serialization is the process of converting an object into a byte stream, which can be easily stored in a file or transmitted over a network. Later, this byte stream can be deserialized, reconstructing the original object.

(Serialization is a mechanism of converting the state of an object into a byte stream. Serialization is done using ObjectOutputStream.

Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory.

This mechanism is used to persist the object. Deserialization is done using ObjectInputStream.

Thus it can be used to make an eligible for saving its state into a file.)

👉 Simple Example: Imagine you are playing a video game and you save your progress (serialization). Later, when you reopen the game, you load the saved progress and continue playing (deserialization).

To make a class serializable, you simply need to implement the Serializable interface. Here's a basic example:

```
import java.io.Serializable;

public class Student implements Serializable{

 int id;
 String name;

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

## Serialization

---

```
import java.io.*;

class Persist{

 public static void main(String args[]){
 try{
 //Creating the object
 Student s1 =new Student(211,"ravi");
 //Creating stream and writing the object
 FileOutputStream fout=new FileOutputStream("f.txt");
 ObjectOutputStream out=new ObjectOutputStream(fout);
 out.writeObject(s1);
 }
 }
}
```

```
out.flush();

//closing the stream

out.close();

System.out.println("success");

}catch(Exception e){System.out.println(e);}

}

}
```

### Deserialization

---

```
import java.io.*;

class Depersist{

public static void main(String args[]){

try{

//Creating stream to read the object

ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));

Student s=(Student)in.readObject();

//printing the data of the serialized object

System.out.println(s.id+" "+s.name);

//closing the stream

in.close();

}catch(Exception e){System.out.println(e);}

}

}
```

Key points about serialization and the Serializable interface:

1. **\*Serialization Process:\*** When an object of a serializable class is serialized, the Java runtime converts the object's state (fields) into a byte stream. This byte stream can then be saved to a file, sent over a network, etc.

2. **\*No Methods in Serializable:\*** Serializable is a marker interface, meaning it does not have any methods that need to be implemented. It acts as a signal to the Java runtime that instances of the class can be serialized.

3. **\*Versioning Control:\*** It's a good practice to include a serialVersionUID field (as shown in the example) to control versioning. This helps to manage compatibility between the serialized form of the class and its later versions.

Remember that not all objects can or should be serialized. Some classes may contain non-serializable fields or sensitive information that should not be exposed in serialized form. In such cases, appropriate steps must be taken to handle serialization accordingly.

SerialVersionUID :

The serialization process at runtime associates an id with each Serializable class which is known as SerialVersionUID.

It is used to verify the sender and receiver of the serialized object. The sender and receiver must be the same. To verify it, SerialVersionUID is used.

The sender and receiver must have the same SerialVersionUID, otherwise, InvalidClassException will be thrown when you deserialize the object.

We can also declare our own SerialVersionUID in the Serializable class. To do so, you need to create a field SerialVersionUID and assign a value to it.

It must be of the long type with static and final. It is suggested to explicitly declare the serialVersionUID field in the class and have it private also. For example:

```
private static final long serialVersionUID=1L;
```

Real Use Case: Version Compatibility :

1. Imagine a Banking App:

Server Side (v1.0): Saves Account object to a file.

Later... App Updated to v2.0: Adds email field to Account class.

You try to read the old file into the new version —  boom! Exception.

 If you define serialVersionUID manually, this won't be a problem.

2. Imagine you're using Kafka or Redis cache:

You serialize and send an object from Microservice A.

Microservice B (with updated code) deserializes it.

 If the classes differ and serialVersionUID isn't fixed — crash during deserialization!

### Singleton Class:

=====

\*Explanation:\*

A Singleton class is a design pattern that restricts the instantiation of a class to a single instance and provides a global point of access to that instance.

It ensures that there is only one instance of the class in the application, and it provides a mechanism to access that instance.

\*Uses:\*

- **\*Resource Management:\*** When you want to control access to a shared resource, such as a database connection or a configuration manager.
- **\*Logging:\*** In scenarios where a single logging instance needs to manage logs across the application.
- **\*Caching:\*** To maintain a single cache manager instance throughout the application.

**\*Example Code:\***

Certainly! Here's an example of a singleton class for resource management in MySQL and logging:

```
java

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.logging.Logger;

public class ResourceManager {

 private static ResourceManager instance;
 private Connection databaseConnection;
 private Logger logger;

 private ResourceManager() {
 // Private constructor to prevent instantiation from outside the class.
 initializeDatabaseConnection();
 initializeLogger();
 }

}
```

```
public static synchronized ResourceManager getInstance() {
 if (instance == null) {
 instance = new ResourceManager();
 }
 return instance;
}

private void initializeDatabaseConnection() {
 // Initialize your MySQL database connection here.
 String url = "jdbc:mysql://localhost:3306/your_database";
 String user = "your_username";
 String password = "your_password";

 try {
 databaseConnection = DriverManager.getConnection(url, user,
password);
 } catch (SQLException e) {
 e.printStackTrace();
 // Log the exception using the logger.
 logger.severe("Failed to initialize database connection: " +
e.getMessage());
 }
}

private void initializeLogger() {
 // Initialize your logger here.
 logger = Logger.getLogger(ResourceManager.class.getName());
```

```
 }

 public Connection getDatabaseConnection() {
 return databaseConnection;
 }

 public void performDatabaseOperation() {
 // Example method for performing a database operation.
 // Use this method in other classes to interact with the database.
 // ...
 }

 public void logMessage(String message) {
 // Example method for logging messages.
 // Use this method in other classes to log messages.
 logger.info(message);
 }
}
```

In this example, the ResourceManager class is a singleton responsible for managing the MySQL database connection and logging. Other classes can obtain an instance of this class using the getInstance() method. They can then use methods like getDatabaseConnection() to get the database connection or logMessage() to log messages.

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

```

// Get an instance of ResourceManager.

ResourceManager resourceManager = ResourceManager.getInstance();

// Use the database connection obtained from ResourceManager.

Connection databaseConnection =
resourceManager.getDatabaseConnection();

// Perform a database operation using the obtained connection.

try (Statement statement = databaseConnection.createStatement()) {

 ResultSet resultSet = statement.executeQuery("SELECT * FROM
your_table");

 // Process the resultSet or perform other database operations...

} catch (SQLException e) {

 e.printStackTrace();

 // Log the exception using ResourceManager's logger.

 resourceManager.logMessage("Error in DatabaseOperationClass: "
+ e.getMessage());

}

}

```

In this example:

1. The ResourceManager instance is obtained using ResourceManager.getInstance().

2. The database connection is obtained through `resourceManager.getDatabaseConnection()`.
3. A database operation is performed using the obtained connection. Any exceptions are logged using `resourceManager.logMessage()`.

Similarly, other classes can use the `ResourceManager` singleton to log messages:

java

```
public class LoggingClass {
 public static void main(String[] args) {
 // Get an instance of ResourceManager.
 ResourceManager resourceManager = ResourceManager.getInstance();

 // Log a message using ResourceManager's logger.
 resourceManager.logMessage("This is a log message from
LoggingClass.");
 }
}
```

This way, the `ResourceManager` encapsulates the details of resource management and logging, providing a centralized and consistent way for other classes to access these functionalities.

Please note that this is a simplified example, and you may need to adapt it based on your specific requirements and the details of your MySQL setup. Additionally, consider using connection pooling for more efficient resource management in a production environment

#### #### Object Cloning:

=====

##### \*Explanation:\*

Object cloning is the process of creating an exact copy of an object. In Java, the Cloneable interface is used, and the `clone()` method is overridden to achieve object cloning.

Cloning can be shallow or deep, depending on whether it creates copies of referenced objects.

##### \*Uses:\*

- \*Prototype Pattern:\* Creating new objects by copying an existing object (prototype) rather than creating them from scratch.
- \*Immutable Classes:\* Cloning is often used in classes where you want to create copies without risking modification of the original instance.
- \*Copying Collections:\* Creating independent copies of collections to avoid shared references.

##### \*Example Code:\*

```
java
// Person.java

public class Person implements Cloneable {
 private String name;
 private int age;

 // Constructors, getters, setters...
```

```
@Override
public Object clone() throws CloneNotSupportedException {
 return super.clone();
}

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

Usage:

```
java
// CloningUsage.java
public class CloningUsage {
 public static void main(String[] args) throws CloneNotSupportedException {
 Person person1 = new Person("John", 30);
 Person person2 = (Person) person1.clone();

 System.out.println("Original: " + person1);
```

```
 System.out.println("Cloned: " + person2);

 }

}
```

In the Person class example, the clone method creates a shallow copy. If your class contains references to other objects, and you need to create a deep copy, you may need to override the clone method accordingly.

Remember to handle `CloneNotSupportedException` or use the `Cloneable` interface carefully, considering alternatives like copy constructors or serialization based on your specific requirements.

\*Note:\* The `Cloneable` interface is a marker interface, and it's recommended to override the `clone` method when implementing it.

Remember that cloning has some complexities, especially with deep cloning and handling mutable objects within the cloned object. It's often considered in conjunction with the `clone` method, and sometimes custom serialization and deserialization approaches are preferred.

Deep cloning and shallow cloning are two concepts related to the copying of objects in Java. Let's explore the differences between them and provide examples with code.

### Shallow Cloning:

**\*\*Shallow cloning\*\*** creates a new object but does not create copies of the objects referenced by the original object. Instead, it copies references to the objects. As a result, changes made to the internal state of the cloned object may affect the original object and vice versa if they share mutable objects.

Here's an example using the `clone()` method for shallow cloning:

```
```java
class Person implements Cloneable {

    String name;

    Address address;

    public Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }

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

class Address {
    String city;
```

```

public Address(String city) {
    this.city = city;
}

}

public class ShallowCloneExample {
    public static void main(String[] args) throws CloneNotSupportedException {
        Address address = new Address("City A");
        Person originalPerson = new Person("John", address);

        // Shallow clone
        Person clonedPerson = (Person) originalPerson.clone();

        // Modify the cloned object
        clonedPerson.name = "Jane";
        clonedPerson.address.city = "City B"; // Shallow cloning won't create a
new Address object

        // Original object is also affected
        System.out.println(originalPerson.name);      // Output: John
        System.out.println(originalPerson.address.city); // Output: City B
    }
}
```

```

### Deep Cloning:

**\*\*Deep cloning\*\*** creates a new object and also creates copies of the objects referenced by the original object. It ensures that changes made to the internal state of the cloned object do not affect the original object, and vice versa.

In Java, achieving deep cloning often involves implementing a custom clone method that explicitly creates copies of the internal objects.

```
```java
class Person implements Cloneable {

    String name;

    Address address;

    public Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    // Deep clone
    @Override
    protected Object clone() throws CloneNotSupportedException {
        Person clonedPerson = (Person) super.clone();
        // Create a new Address object for deep cloning
        clonedPerson.address = (Address) address.clone();
        return clonedPerson;
    }
}
```

```
class Address implements Cloneable {  
    String city;  
  
    public Address(String city) {  
        this.city = city;  
    }  
  
    // Deep clone for Address  
    @Override  
    protected Object clone() throws CloneNotSupportedException {  
        return super.clone();  
    }  
}  
  
public class DeepCloneExample {  
    public static void main(String[] args) throws CloneNotSupportedException {  
        Address address = new Address("City A");  
        Person originalPerson = new Person("John", address);  
  
        // Deep clone  
        Person clonedPerson = (Person) originalPerson.clone();  
  
        // Modify the cloned object  
        clonedPerson.name = "Jane";
```

```

        clonedPerson.address.city = "City B"; // Original Address object remains
unchanged

// Original object is not affected

System.out.println(originalPerson.name);      // Output: John

System.out.println(originalPerson.address.city); // Output: City A

}

}

```

```

In the deep cloning example, a custom `clone` method is implemented for both `Person` and `Address` classes to ensure that a new `Address` object is created during cloning. This way, changes to the `Address` object in the cloned `Person` do not affect the original `Person`.

## IMPORTANTS

---

Using `List<Student> students = new ArrayList<>();` instead of `ArrayList<Student> students = new ArrayList<>();` provides more flexibility and follows the principle of programming to interfaces. Here are the reasons and benefits:

### 1. \*Flexibility and Abstraction:\*

- Using `List<Student>` allows you to switch the implementation later without changing the rest of your code. You can easily switch to a different List implementation like `LinkedList` or any other class that implements the `List` interface.

```
java
```

```
List<Student> students = new LinkedList<>();
```

## 2. \*Programming to Interfaces:\*

- It's a good practice to program to interfaces or superclasses rather than concrete classes. This promotes flexibility and helps in adhering to the Dependency Inversion Principle of SOLID.

```
java
```

```
// Preferred way - Programming to the interface
```

```
List<Student> students = new ArrayList<>();
```

```
// Avoid - Concrete implementation
```

```
ArrayList<Student> students = new ArrayList<>();
```

## 3. \*Easier Testing:\*

- When using interfaces, it's easier to mock objects for testing, leading to more effective unit testing.

### ### Examples:

#### #### Programming to Interface:

```
java
```

```
// Programming to Interface
```

```
List<Student> students = new ArrayList<>();

students.add(new Student("John"));
students.add(new Student("Alice"));

for (Student student : students) {
 System.out.println(student.getName());
}
```

#### #### Programming to Concrete Class:

```
java
// Avoid - Programming to Concrete Class
ArrayList<Student> students = new ArrayList<>();

students.add(new Student("John"));
students.add(new Student("Alice"));

for (Student student : students) {
 System.out.println(student.getName());
}
```

In the first example, if you decide later to change the ArrayList to a different implementation, you only need to change the right side of the assignment. In the second example, if you used ArrayList throughout your code and later decided to switch to LinkedList,

you would need to modify every line where you instantiated or used the ArrayList, which can be error-prone and time-consuming.

Programming to interfaces allows for more maintainable, modular, and testable code. It is a key principle in object-oriented design, promoting loose coupling between components and making your code more adaptable to changes.

Certainly! Let's dive into explanations of Spring Boot Security and OAuth, along with their uses, advantages, disadvantages, and some examples.

## HIBERNATE

---

Hibernate is an open-source Java framework that provides an object-relational mapping (ORM) solution.

It simplifies database access for Java applications by mapping Java objects to database tables,

enabling developers to work with databases using Java objects rather than writing raw SQL queries.

Here are some key aspects of Hibernate:

**\*\*Features of Hibernate\*\*:**

1. **Object-Relational Mapping (ORM)**: Hibernate allows you to map Java objects to database tables, making it easier to work with relational databases.

2. **Database Independence**: Hibernate abstracts the underlying database, allowing you to write database-agnostic code. You can switch between different databases without changing your application's code.

3. **Automatic Table Generation**: Hibernate can automatically generate database tables based on your Java entities.

4. **Caching**: Hibernate provides caching mechanisms to improve performance, reducing the number of database queries.

5. **Query Language (HQL)**: Hibernate offers its query language called HQL (Hibernate Query Language) for querying the database. HQL is similar to SQL but operates on Java objects.

6. **Lazy Loading**: Hibernate supports lazy loading, which means that it loads related data from the database only when it is explicitly requested.

7. **Transaction Management**: It provides built-in support for managing database transactions.

8. **Association Mapping**: Hibernate supports various types of associations between entities, such as one-to-one, one-to-many, many-to-one, and many-to-many.

**Differences between Hibernate and JDBC**:

1. **Abstraction Level**: Hibernate provides a higher-level abstraction for database access, while JDBC is a lower-level API for interacting with databases. With Hibernate, you work with Java objects, whereas with JDBC, you write SQL queries and work with result sets directly.
2. **SQL vs. HQL**: With JDBC, you write SQL queries explicitly. In Hibernate, you can use HQL, which is a higher-level query language that operates on Java objects.
3. **Mapping**: In Hibernate, you define entity classes to map to database tables, while in JDBC, you need to write code to map Java objects to SQL and vice versa.
4. **Portability**: Hibernate provides database independence, allowing you to switch between different databases with minimal code changes. JDBC code may be tightly coupled to a specific database, making it less portable.

**Examples:**

**JDBC Example:**

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

public class JDBCDemo {
    public static void main(String[] args) {
        try {
            // Load the JDBC driver

```

```
Class.forName("com.mysql.jdbc.Driver");

// Establish a connection to the database
Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "username", "password");

// Create a SQL statement
Statement statement = connection.createStatement();

// Execute a query
ResultSet resultSet = statement.executeQuery("SELECT * FROM
employees");

// Process the result set
while (resultSet.next()) {
    System.out.println(resultSet.getString("name"));
}

// Close resources
resultSet.close();
statement.close();
connection.close();

} catch (Exception e) {
    e.printStackTrace();
}

}
```

```

**\*\*Hibernate Example\*\*:**

```
```java
```

```
@Entity  
@Table(name = "employees")  
public class Employee {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private int id;  
  
    private String name;  
  
    // Getters and setters  
}
```

```
public class HibernateDemo {  
  
    public static void main(String[] args) {  
  
        SessionFactory sessionFactory = new  
        Configuration().configure().buildSessionFactory();  
  
        Session session = sessionFactory.openSession();  
  
        // Create a new employee  
        Employee employee = new Employee();  
        employee.setName("John Doe");
```

```
// Save the employee to the database
session.beginTransaction();
session.save(employee);
session.getTransaction().commit();

// Retrieve an employee
Employee retrievedEmployee = session.get(Employee.class, 1);
System.out.println(retrievedEmployee.getName());

session.close();
sessionFactory.close();

}

}

```

```

In the Hibernate example, you define an `Employee` entity, and Hibernate takes care of the database interaction for you. This code is more abstract and portable compared to the low-level JDBC code.

=====

Caching

=====

Caching in Hibernate is a powerful feature that can significantly improve the performance of your application by reducing the number of database queries.

It helps in avoiding repetitive database access for frequently accessed data by storing the data in memory. Hibernate supports different levels of caching, which can be categorized as first-level and second-level caches.

#### ### 1. First-Level Cache

The first-level cache is associated with the session (or entity manager) in Hibernate. It is enabled by default and cannot be disabled. Here's how it works:

- **Scope**: The first-level cache is session-scoped, meaning it is valid only within a single session.
- **Operation**: When you perform operations such as `find`, `get`, or `load`, Hibernate first checks the session cache to see if the entity is already loaded. If it is, it returns the entity from the cache instead of querying the database.
- **Lifecycle**: The cache is cleared when the session is closed or when explicitly cleared using methods like `session.clear()` or `session.evict()`.

#### #### Example

```
```java
```

```
Session session = sessionFactory.openSession();
```

```
// The first query will hit the database
```

```
Employee employee1 = session.get(Employee.class, 1L);
```

```
// The second query will fetch the entity from the first-level cache
```

```
Employee employee2 = session.get(Employee.class, 1L);
```

```
session.close();
```

```
```
```

### ### 2. Second-Level Cache

The second-level cache is session-factory-scoped and can be shared across sessions. It is optional and needs to be explicitly configured. Hibernate integrates with several caching providers such as EHCache, Infinispan, and others.

- **Scope**: The second-level cache is shared across sessions and is associated with the session factory.
- **Configuration**: It must be enabled and configured in the Hibernate configuration files (e.g., `hibernate.cfg.xml` or `application.properties`).

#### #### Configuration Example

**hibernate.cfg.xml**:

```
```xml
<property name="hibernate.cache.use_second_level_cache">true</property>
<property
name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegionFact
ory</property>
<property
name="hibernate.cache.provider_configuration_file_resource_path">/ehcache.xml</property>
```

```

**ehcache.xml**:

```
```xml
<ehcache>
    <cache name="com.example.Employee"
        maxEntriesLocalHeap="1000" // number of entities it can store.
        timeToLiveSeconds="3600"/> // expire time for each cache entity.
        (3600-1 hour)
</ehcache>
```
```

```

Example Usage

```
```java
Session session1 = sessionFactory.openSession();
Employee employee1 = session1.get(Employee.class, 1L);
session1.close(); // Employee data is cached in the second-level cache

Session session2 = sessionFactory.openSession();
Employee employee2 = session2.get(Employee.class, 1L); // This will hit the second-level
cache
session2.close();
```
```

```

#### ### 3. Query Cache

Hibernate also provides a query cache, which caches the results of queries. This cache works in conjunction with the second-level cache. To use the query cache, it must be explicitly enabled.

#### #### Configuration Example

\*\*hibernate.cfg.xml\*\*:

```xml

```
<property name="hibernate.cache.use_query_cache">true</property>
```

```

#### #### Example Usage

```java

```
Session session = sessionFactory.openSession();
```

```
// Enable query caching for this query
```

```
List<Employee> employees = session.createQuery("FROM Employee")
```

```
.setCacheable(true)
```

```
.list();
```

```
session.close();
```

```

#### ### Benefits of Caching

- **Performance Improvement**: By reducing the number of database queries, caching can significantly improve the performance and scalability of the application.
- **Reduced Latency**: Fetching data from the cache is much faster than retrieving it from the database, reducing the latency for data access.
- **Scalability**: Offloading read operations from the database can help the system handle a larger number of requests and users.

#### Example with Simple Scenario use of Caching

---

1. **When you hit the URL with the same product ID multiple times within an hour**:
  - The first time you access the product details, it may hit the database to fetch the data.
  - Subsequent requests with the same product ID will retrieve the product details from the cache instead of hitting the database. This is because the data is stored in the cache after the first retrieval.
2. **After 1 hour**:
  - If you hit the URL with the same product ID after the cache expiry time (1 hour), the cache for that specific product ID will expire.
  - When the cache expires, subsequent requests with the same product ID will not find the data in the cache, so they will hit the database again to fetch the fresh data.

#### ASSOCIATE MAPPING :

---

Certainly! Let's explain each type of association mapping using `Employee` and `Address` entities.

### 1. \*\*One-to-One Relationship\*\*:

In a one-to-one relationship, each entity instance is associated with exactly one instance of another entity. In our example, each `Employee` instance will have exactly one associated `Address` instance, and vice versa.

```
```java
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToOne(mappedBy = "employee", cascade = CascadeType.ALL)
    private Address address;

    // Getters and setters
}

@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```

    private String street;
    private String city;
    private String country;

    @OneToOne
    @JoinColumn(name = "employee_id")
    private Employee employee;

    // Getters and setters
}

...

```

2. **One-to-Many Relationship**:

In a one-to-many relationship, each instance of the owning entity (source) is associated with multiple instances of the target entity (destination). In our example, each `Employee` instance can have multiple associated `Address` instances.

```

```java
@Entity
public class Employee {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;
}

```

```
 @OneToMany(mappedBy = "employee", cascade = CascadeType.ALL)
 private List<Address> addresses = new ArrayList<>();

 // Getters and setters
}

@Entity
public class Address {
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String street;
 private String city;
 private String country;

 @ManyToOne
 @JoinColumn(name = "employee_id")
 private Employee employee;

 // Getters and setters
}

```

```

3. **Many-to-One Relationship**:

In a many-to-one relationship, multiple instances of the owning entity (source) are associated with exactly one instance of the target entity (destination). In our example, multiple `Employee` instances can be associated with exactly one `Address` instance.

```
```java
@Entity
public class Employee {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 @ManyToOne
 @JoinColumn(name = "address_id")
 private Address address;

 // Getters and setters
}

@Entity
public class Address {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String street;
```

```
 private String city;

 private String country;

 // Getters and setters
}

...
```

#### 4. **Many-to-Many Relationship**:

In a many-to-many relationship, multiple instances of each entity can be associated with multiple instances of the other entity. In our example, multiple `Employee` instances can be associated with multiple `Address` instances, and vice versa.

```
```java  
  
@Entity  
  
public class Employee {  
  
    @Id  
  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
  
    private Long id;  
  
    private String name;  
  
    @ManyToMany  
  
    @JoinTable(name = "employee_address",  
              joinColumns = @JoinColumn(name = "employee_id"),  
              inverseJoinColumns = @JoinColumn(name =  
"address_id"))
```

```
private List<Address> addresses = new ArrayList<>();  
  
    // Getters and setters  
}  
  
  
@Entity  
public class Address {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    private String street;  
    private String city;  
    private String country;  
  
    @ManyToMany(mappedBy = "addresses")  
    private List<Employee> employees = new ArrayList<>();  
  
    // Getters and setters  
}  
...  
  
=====
```

These examples demonstrate how different types of association mappings can be implemented using Hibernate annotations in Spring Boot applications.

CASCADE TYPES

The `cascade` attribute in JPA (Java Persistence API) specifies the set of operations that should be propagated from a parent entity to its associated entities. When `cascade = CascadeType.ALL` is used, it means that all the operations performed on the parent entity (such as persist, merge, remove, refresh, detach) will also be performed on the associated entities.

JPA defines several types of cascade operations:

1. **`CascadeType.PERSIST`**: When the parent entity is persisted (saved), the associated entity is also persisted.
2. **`CascadeType.MERGE`**: When the parent entity is merged, the associated entity is also merged. This operation synchronizes the state of the detached entity with the current state in the database.
3. **`CascadeType.REMOVE`**: When the parent entity is removed (deleted), the associated entity is also removed.
4. **`CascadeType.REFRESH`**: When the parent entity is refreshed, the state of the associated entity is also refreshed from the database.
5. **`CascadeType.DETACH`**: When the parent entity is detached from the persistence context, the associated entity is also detached.
6. **`CascadeType.ALL`**: This is a shorthand for specifying all of the above cascade types. It means that persist, merge, remove, refresh, and detach operations will all be cascaded to the associated entity.

Usage

When you use `cascade = CascadeType.ALL`, it means that any operation you perform on the parent entity will also be automatically applied to the associated entities. For example:

```
```java
```

```
@OneToOne(mappedBy = "employee", cascade = CascadeType.ALL)
private Address address;
```
```

This configuration implies that:

- **Persist**: If you save an `Employee`, the `Address` will be saved as well.
- **Merge**: If you update an `Employee`, the changes will also be applied to the `Address`.
- **Remove**: If you delete an `Employee`, the `Address` will be deleted as well.
- **Refresh**: If you refresh an `Employee`, the `Address` will also be refreshed.
- **Detach**: If you detach an `Employee`, the `Address` will also be detached from the persistence context.

Example Without Cascade

If you don't use cascade, each entity must be managed (persisted, merged, removed, etc.) individually. For example:

```
```java  
@OneToOne(mappedBy = "employee")
private Address address;
```
```

With this configuration:

- If you persist an `Employee`, the associated `Address` will not be automatically persisted. You must explicitly persist the `Address`.

- If you remove an `Employee`, the associated `Address` will not be automatically removed. You must explicitly remove the `Address`.

Example with Cascade

Consider the following example:

```
```java
```

```
Employee employee = new Employee();
employee.setName("John Doe");
```

```
Address address = new Address();
address.setStreet("123 Main St");
address.setCity("Anytown");
address.setCountry("Country");
employee.setAddress(address);
address.setEmployee(employee);
```

```
// Persisting the employee will also persist the address
entityManager.persist(employee);
```

```
// Removing the employee will also remove the address
entityManager.remove(employee);
```
```

In this example, because `cascade = CascadeType.ALL` is used, when you persist the `Employee`, the `Address` is also persisted automatically. Similarly, when you remove the `Employee`, the `Address` is also removed automatically.

Summary

- **Using `cascade = CascadeType.ALL`**: Ensures that all operations (persist, merge, remove, refresh, detach) on the parent entity are propagated to the associated entities. This can simplify code by eliminating the need to manually manage associated entities.

- **Not Using Cascade**: Requires explicit management of each entity. You need to persist, merge, remove, etc., each entity individually, providing more control but requiring more code and careful handling to maintain consistency.

Choosing whether to use cascade operations depends on the specific requirements of your application and how you want to manage the lifecycle of your entities.

LAZY LOADING IN HIBERNATE :

=====

Lazy loading in Hibernate is a technique used to defer the loading of associated entities or collections until they are explicitly accessed. This helps in reducing unnecessary database queries and improves performance by loading only the required data when needed.

There are two main types of lazy loading in Hibernate:

1. **Lazy Loading for Associations**: This is when Hibernate delays the loading of associated entities until they are accessed for the first time.

2. ****Lazy Loading for Collections****: This is when Hibernate delays the loading of collections of associated entities until they are accessed for the first time.

Let's differentiate between the two types with examples:

1. Lazy Loading for Associations

Consider a scenario where you have two entities: `Employee` and `Department`, with a many-to-one association from `Employee` to `Department`.

Employee Entity:

```
```java
@Entity
public class Employee {
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
 private String name;

 @ManyToOne(fetch = FetchType.LAZY) // Lazy loading for association
 private Department department;

 // Getters and setters
}
```

```

Department Entity:

```
```java
@Entity
public class Department {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 // Getters and setters
}

```
```

```

In this example, the `department` association in the `Employee` entity is configured for lazy loading (`FetchType.LAZY`). This means that when you fetch an `Employee`, the associated `Department` will not be loaded from the database until you explicitly access it.

### 2. Lazy Loading for Collections

Consider a scenario where you have an entity `Order` that has a one-to-many association with `OrderItem`. The `Order` entity contains a collection of `OrderItem` objects.

#### Order Entity:

```
```java
@Entity
public class Order {
```

```
public class Order {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String orderNumber;  
  
    @OneToMany(mappedBy = "order", fetch = FetchType.LAZY) // Lazy loading for  
    collection  
    private List<OrderItem> orderItems = new ArrayList<>();  
  
    // Getters and setters  
}  
...  
//
```

OrderItem Entity:

```
```java  
@Entity
public class OrderItem {
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
 private String product;
 private int quantity;

 @ManyToOne
```

```
 @JoinColumn(name = "order_id")

 private Order order;

 // Getters and setters
}

...
```

In this example, the `orderItems` collection in the `Order` entity is configured for lazy loading (`FetchType.LAZY`). This means that when you fetch an `Order`, the associated `OrderItem` objects will not be loaded from the database until you explicitly access them.

#### ### Summary

- **Lazy Loading for Associations**: Delays the loading of associated entities until they are accessed for the first time.
- **Lazy Loading for Collections**: Delays the loading of collections of associated entities until they are accessed for the first time.

These lazy loading configurations help optimize database access and improve performance by loading only the necessary data when needed.

I see you've provided two identical `EmployeeController` classes. Let's assume you meant to provide two different scenarios: one where the `department` field is not explicitly accessed, and one where it is explicitly accessed.

#### ### Scenario 1: Department Not Explicitly Accessed

```
```java
@RestController
@RequestMapping("/employees")
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    @GetMapping("/{id}")
    public ResponseEntity<Employee> getEmployee(@PathVariable Long id) {
        Employee employee = employeeService.findEmployeeById(id);
        // At this point, the department associated with the employee is not
        loaded
        // It will only be loaded from the database when explicitly accessed
        return ResponseEntity.ok(employee);
    }
}
```

```

Output:

```
```json
{
    "id": 1,
    "name": "John Doe",
    "department": null
}
```

```

### ### Scenario 2: Department Explicitly Accessed

```
```java
```

```
@RestController  
@RequestMapping("/employees")  
public class EmployeeController {  
  
    @Autowired  
    private EmployeeService employeeService;  
  
    @GetMapping("/{id}")  
    public ResponseEntity<Employee> getEmployee(@PathVariable Long id) {  
        Employee employee = employeeService.findEmployeeById(id);  
  
        // Explicitly accessing the department field  
        Department department = employee.getDepartment();  
  
        // At this point, Hibernate will execute another SQL query to fetch the  
        // associated Department from the database  
  
        return ResponseEntity.ok(employee);  
    }  
}
```

Output:

```
```json
{
 "id": 1,
 "name": "John Doe",
 "department": {
 "id": 1,
 "name": "Engineering"
 }
}
```

```

In scenario 1, since the department field is not explicitly accessed, the department associated with the employee is 'null'.

In scenario 2, the department field is explicitly accessed using `employee.getDepartment()`, triggering Hibernate to execute another SQL query to fetch the associated department from the database. Therefore, the department details are included in the output JSON response.

SPRING

```
=====
```

Spring is a widely used framework for building enterprise-level Java applications. It provides comprehensive infrastructure support for developing Java-based applications, making it easier to create robust, maintainable, and scalable software.

Spring offers various modules for different purposes, such as dependency injection, aspect-oriented programming, data access, messaging, and more. In a Spring application, you can develop components as POJOs (Plain Old Java Objects) and use Spring to manage these components, their dependencies, and other aspects of the application.

Here's a simplified example of a traditional Spring application (non-Spring Boot) to help you understand the fundamental concepts. We'll create a simple Spring application that manages a list of users.

Advantages of Spring Framework

There are many advantages of Spring Framework. They are as follows:

1) Predefined Templates

Spring framework provides templates for JDBC, Hibernate, JPA etc. technologies. So there is no need to write too much code. It hides the basic steps of these technologies.

Let's take the example of JdbcTemplate, you don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. You need to write the code of executing query only. Thus, it save a lot of JDBC code.

2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

3) Easy to test

The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.

4) Lightweight

Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

5) Fast Development

The Dependency Injection feature of Spring Framework and it support to various frameworks makes the easy development of JavaEE application.

6) Powerful abstraction

It provides powerful abstraction to JavaEE specifications such as JMS, JDBC, JPA and JTA.

7) Declarative support

It provides declarative support for caching, validation, transactions and formatting.

Step 1: Set Up the Project

Create a Maven project and add the necessary dependencies to your `pom.xml`.

pom.xml:

```xml

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/maven-4.0.0.xsd">

 <modelVersion>4.0.0</modelVersion>
 <groupId>com.example</groupId>
 <artifactId>spring-user-app</artifactId>
 <version>1.0-SNAPSHOT</version>
 <dependencies>
 <dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-context</artifactId>
 <version>5.3.10.RELEASE</version>
 </dependency>
 </dependencies>
</project>
```

```

Step 2: Create a User Entity

Create a `User` entity class to represent users:

```
```java
public class User {
 private long id;
 private String username;
 private String email;
```

```
// Getters and setters
}
...

...
...
...
...
...
```

#### \*\*Step 3: Create a UserRepository\*\*

Create a repository class to manage user data:

```
```java  
public class UserRepository {  
  
    private List<User> users = new ArrayList<>();  
  
    public void addUser(User user) {  
        users.add(user);  
    }  
  
    public List<User> getUsers() {  
        return users;  
    }  
}  
  
...  
...  
...
```

Step 4: Create a Controller

Create a controller class to handle user-related operations:

```
```java
public class UserController {

 private UserRepository userRepository;

 public UserController(UserRepository userRepository) {
 this.userRepository = userRepository;
 }

 public void addUser(User user) {
 userRepository.addUser(user);
 }

 public List<User> getUsers() {
 return userRepository.getUsers();
 }
}

```

```

Step 5: Create the Spring Configuration

Create a Spring configuration XML file to define Spring beans and wire them together.
Let's call it `applicationContext.xml`:

applicationContext.xml:

```
```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

 <bean id="userRepository" class="com.example.UserRepository" />
 <bean id="userController" class="com.example.UserController">
 <constructor-arg ref="userRepository" />
 </bean>
</beans>
```

```

Step 6: Create the Main Application

Create a Java class to load the Spring application context and use the defined beans:

```
```java
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Application {
 public static void main(String[] args) {
 ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
 }
}
```

```
UserController userController = context.getBean(UserController.class);

// Use userController to handle user operations

}

}

```

```

In this example, we manually configure Spring beans and their dependencies using an XML configuration file. This demonstrates the basics of a traditional Spring application. You can expand this foundation to build more complex applications.

SPRING AND SPRINGBOOT

Spring and **Spring Boot** are two frameworks in the Spring ecosystem for building Java applications. Here's an overview of each and the key differences between them:

Spring:

1. **Core Container:** The Spring Framework, often referred to as "Spring," provides a comprehensive programming and configuration model for building enterprise applications. It

includes modules for managing application configuration, handling dependency injection, and providing core features like AOP (Aspect-Oriented Programming) and transactions.

2. **XML Configuration:** Spring applications are often configured using XML files or Java-based configurations. Configuration details, such as bean definitions and dependency injection, are explicitly defined in these configuration files.

3. **Fine-Grained Control:** Spring offers a high degree of customization and fine-grained control over application components and their behavior. Developers have the flexibility to configure and define application components in a highly customizable way.

4. **Requires Boilerplate Code:** Developing Spring applications typically involves writing more boilerplate code, such as XML configuration or Java-based configuration classes, to set up the application context and configure beans.

5. **Flexible for Complex Applications:** Spring is well-suited for complex enterprise applications where custom configuration and extensive control are required.

Spring Boot:

1. **Opinionated Framework:** Spring Boot is an opinionated framework built on top of the Spring Framework. It simplifies the setup and development of Spring applications by providing a set of defaults and conventions.

2. **Auto-Configuration:** Spring Boot introduces auto-configuration, which automatically configures many aspects of the application based on classpath dependencies. Developers can override these defaults when needed.

3. **Annotations and Defaults:** Spring Boot encourages the use of annotations and sensible defaults. This leads to reduced configuration and less boilerplate code.

4. **Microservices-Ready:** Spring Boot is popular for building microservices due to its ease of development and deployment. It's optimized for creating stand-alone, production-ready Spring-based applications with minimal effort.

5. **Quick Start:** Spring Boot provides a quick and easy way to start new projects. Developers can create a new Spring Boot application with minimal configuration and immediately start building features.

****Key Differences:****

1. **Configuration:** Spring applications typically require explicit configuration through XML files or Java-based configuration classes, while Spring Boot encourages convention over configuration and relies on auto-configuration.

2. **Boilerplate Code:** Spring applications often involve writing more boilerplate code for setting up the application context and defining beans. Spring Boot reduces the need for boilerplate code.

3. **Complexity:** Spring offers a high degree of customization and is suitable for complex, custom enterprise applications. Spring Boot is designed to simplify development and is well-suited for microservices and rapid application development.

4. **Default Settings:** Spring Boot provides default settings and opinions to get started quickly, while Spring leaves many configuration decisions to the developers.

In summary, Spring is a comprehensive framework that offers fine-grained control and flexibility, while Spring Boot is an opinionated framework that simplifies Spring application development by providing defaults and conventions. The choice between Spring and Spring Boot depends on the specific project requirements and the development team's preferences. Spring Boot is often preferred for new projects, prototypes, and microservices.

SPRING-BOOT

=====

application.properties file

=====

#changing port number

server.port=8081

spring.jpa.hibernate.ddl-auto=update

spring.datasource.url=jdbc:mysql://localhost:3306/practice

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

#Logging the sql queries

logging.level.org.hibernate.SQL=DEBUG

logging.level.org.hibernate.type=TRACE

spring.jpa.show-sql=true

#Write a sql query beautiful on console

spring.jpa.properties.hibernate.format_sql=true

WHAT IS SPRING-BOOT

Spring Boot is a Java-based framework that simplifies the development of standalone, production-ready applications.

It is part of the larger Spring ecosystem, but it focuses on making it easy to create Spring applications with minimal configuration.

Spring Boot provides a set of conventions and defaults that enable developers to build applications quickly and efficiently.

1. Simple web application.

```
String data = "Welcome To Plant %s!";

@GetMapping("/")
public String message(@RequestParam(value = "name", defaultValue = "Nani")
String name){

    return String.format(data,name);
}
```

--> @RequestParam : it is telling Spring to expect a name value in the request, but if it's not there, it will use the word "Nani" by default.

<http://localhost:8081/?name=Rajesh>

output: Welcome To Plant Rajesh!

2. Spring Boot Restful Services:

1. Spring Boot is often used to create RESTful web services, which are web services that use HTTP methods (GET, POST, PUT, DELETE) to perform CRUD (Create, Read, Update, Delete) operations on resources.
2. Spring Boot simplifies the creation of RESTful services by providing annotations like `@RestController` to define REST endpoints, and it includes libraries to handle HTTP requests and responses.
3. You can easily build RESTful APIs to expose data or services over HTTP.
4. These services are typically used for communication between systems, mobile apps, web applications, and other clients

3. SpringBoot Data JPA

Spring Data JPA is a part of the larger Spring Data project.

It is a framework that simplifies data access in Java applications that use the Java Persistence API (JPA) to interact with relational databases.

JPA is a Java specification for working with databases in an object-oriented way, allowing developers to work with databases using Java classes and objects instead of SQL queries.

1. Automatic Repository Creation: Spring Boot Data JPA can automatically create repository interfaces for your domain models, which significantly reduces the amount of boilerplate code you need to write for basic CRUD operations.

2. Query Methods: It allows you to create custom query methods by simply defining method names following a specific naming convention. Spring Data JPA translates these method names into SQL queries automatically.

3. Pagination and Sorting: Built-in support for paging and sorting of query results.

4. Transactional Support: It simplifies transaction management, ensuring that database operations are executed within a single transaction context.

5. Custom Queries: You can write native SQL queries, JPQL (Java Persistence Query Language) queries, or use the Criteria API to create complex queries.

```
@RequestBody for List<User>
```

```
=====
```

```
[
```

```
{
```

```
    "firstName":"NaniBabu",
    "lastName":"Pallapu",
    "loginName":"NPallapu",
    "password":"Hyderabad@369",
    "email":"nanipallapu369@mail.com",
    "phone":"9392590089"
```

```
,
```

```
{
```

```
        "firstName":"Priyanka",
        "lastName":"Bandi",
        "loginName":"PBandi",
        "password":"Priya@369",
        "email":"pryanka369@mail.com",
        "phone":"8008142536"
    },
]
```

@RequestBody for Single User Object

```
=====
{
    "firstName":"NaniBabu",
    "lastName":"Pallapu",
    "loginName":"NPallapu",
    "password":"Hyderabad@369",
    "email":"nanipallapu369@mail.com",
    "phone":"9392590089"
}
```

Spring Boot MVC (Model-View-Controller)

Spring Boot MVC is an architectural pattern and framework for building web applications in Spring Boot. It provides a structured approach to handling web requests and responses, separating an application into three interconnected components:

1. Model: Represents the application's data and business logic. It contains the information that the application operates on.
2. View: Represents the presentation layer of the application, responsible for rendering the user interface and displaying data from the Model.
3. Controller: Acts as an intermediary between the Model and View. It processes incoming HTTP requests, interacts with the Model to retrieve data, and selects the appropriate View for rendering the response.

Uses of Spring Boot MVC:

Spring Boot MVC is widely used for building web applications, both simple and complex. Some common use cases include:

1. Web Applications: Building web applications with dynamic content and user interaction.
2. RESTful APIs: Creating RESTful web services to expose data and functionality to clients.
3. Authentication and Authorization: Implementing user authentication and authorization for web applications.

4. Form Handling: Handling forms and user input in web applications.

5. View Templating: Rendering HTML or other view templates to generate dynamic web pages.

Here's an example of a simple Spring Boot MVC application to demonstrate the use of the Model, View, and Controller components:

Model: User.java

```
```java
public class User {

 private String username;
 private String email;

 // Getters and setters
}
```

````

View: userForm.html

```
```html
<!DOCTYPE html>
<html>
<head>
 <title>User Registration</title>
</head>
<body>
```

```
<h1>User Registration</h1>
<form method="post" action="/register">
 <label for="username">Username:</label>
 <input type="text" id="username" name="username" required>

 <label for="email">Email:</label>
 <input type="email" id="email" name="email" required>

 <input type="submit" value="Register">
</form>
</body>
</html>
```

```

Controller: UserController.java

```
```java
@Controller
public class UserController {

 @GetMapping("/registration")
 public String showRegistrationForm(Model model) {
 model.addAttribute("user", new User());
 return "userForm";
 }
}
```

```
@PostMapping("/register")
public String registerUser(@ModelAttribute("user") User user) {
 // Process and store the user data
}
```

```
 return "registrationSuccess";
 }
}
...

In this example:
```

- `User` represents the Model, holding user data.
- The `userForm.html` file is the View, displaying a user registration form.
- `UserController` acts as the Controller, handling the HTTP requests. The `showRegistrationForm` method displays the registration form, and the `registerUser` method processes the submitted form.

This is a simplified example, but it demonstrates the essential components of a Spring Boot MVC application. When a user accesses the `/registration` URL, the registration form is displayed. When the form is submitted, the `registerUser` method processes the user's input, and the response is rendered using a view template.

Spring Boot simplifies the setup and configuration of such applications, making it easier to develop web-based systems.

dependency injection (DI) :

---

dependency injection (DI) in Spring Boot is a design pattern where objects are provided with their dependencies rather than creating them internally.

Imagine you have a class that needs to interact with a database. Instead of creating a database connection inside the class, you pass the database connection to the class from outside.

This way, the class doesn't need to worry about how the database connection is created or managed; it just uses it.

Dependency injection promotes loose coupling between classes, making your code easier to maintain, test, and extend.

It also allows for better modularization and reuse of components within your application.

(OR)

In Spring Boot, Dependency Injection is a design pattern and framework feature that helps manage the dependencies between various components in a Spring application.

Dependency Injection (DI) in Spring Boot, in simple terms, is a way for components within your application to receive the objects they depend on, rather than creating them directly.

Think of it as a mechanism for providing necessary dependencies to an object, rather than the object having to create them itself.

The primary goal of DI is to achieve the Inversion of Control (IoC) principle, where the control of creating and managing objects is shifted from the components themselves to an external entity or container.

In the context of Spring Boot, this container is the Spring IoC container.

Spring framework provides three ways to inject dependency

1. By Constructor Injection
2. By Setter method Injection
3. By Field Injection

1. **Constructor Injection**: Dependencies are provided via the class constructor, often resulting in immutable objects and ensuring that all required dependencies are available when the object is created.

```
```java
    private final MyDependency dependency;

    @Autowired
    public MyClass(MyDependency dependency) {
        this.dependency = dependency;
    }
    ...
```

```

2. **Setter Method Injection**: Dependencies are injected via setter methods, allowing for flexibility in setting dependencies after object creation.

```
```java
    private MyDependency dependency;

    @Autowired
    public void setDependency(MyDependency dependency) {
```

```

```
 this.dependency = dependency;
 }
 ...
```

3. **\*\*Field Injection\*\***: Dependencies are directly injected into fields, potentially leading to less readable code and making it harder to mock dependencies for testing.

```
```java  
@Autowired  
private MyDependency dependency;  
...``
```

The main difference lies in how dependencies are provided to the class: constructor injection ensures that dependencies are available upon object creation, setter injection allows for more flexible injection after object creation, and field injection directly injects dependencies into class fields, potentially leading to readability and testability issues.

Without Dependency Injection:

In this example, the Car class creates its own Engine:

```
class Engine {  
    public void start() {  
        System.out.println("Engine started");  
    }
```

```
}
```

```
class Car {
```

```
    private Engine engine;
```

```
    public Car() {
```

```
        engine = new Engine(); // Creating the engine internally
```

```
}
```

```
    public void start() {
```

```
        engine.start();
```

```
        System.out.println("Car started");
```

```
}
```

```
}
```

```
class Bike {
```

```
    private final Engine engine;
```

```
    public Bike() {
```

```
        this.engine = new Engine();
```

```
}
```

```
    public void start() {
```

```
        engine.start();
```

```
        System.out.println("Bike Started...");
```

```
}
```

```
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car(); // creates its own engine
        car.start();
        Bike bike = new Bike(); //creates its own engine
        bike.start();
    }
}
```

With Dependency Injection:

With Dependency Injection, you can provide the Engine to the Car from the outside:

```
class Engine {
    public void start() {
        System.out.println("Engine started");
    }
}
```

```
class Car {
    private final Engine engine;
```

```
public Car(Engine engine) {  
    this.engine = engine; // Injecting the engine from the outside  
}  
  
public void start() {  
    engine.start();  
    System.out.println("Car started");  
}  
}  
  
class Bike {  
    private final Engine engine;  
  
    public Bike(Engine engine) {  
        this.engine = engine; // Injecting the engine from the outside  
    }  
  
    public void start() {  
        engine.start();  
        System.out.println("Bike Started...");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Engine engine = new Engine(); // Engine created externally
```

```
        Car car = new Car(engine); // Car receives the Engine through injection  
        Bike bike = new Bike(engine); // Car receives the Engine through injection  
  
        car.start();  
  
        bike.start();  
  
    }  
  
}
```

From Above Example :

=====

Without Dependency Injection:

- Tightly Coupled:
 - Car and Bike create their own Engine inside the class.
 - Hard to change or test the Engine easily.

With Dependency Injection:

- Loosely Coupled:
 - Car and Bike receive the Engine from outside (injected).
 - Makes it easier to change the Engine or test the classes.

Code Example:

- Without DI:
 - Car and Bike create an Engine themselves.
- With DI:
 - Engine is created outside and passed to Car and Bike.

Benefits of DI:

- Easier Testing: You can easily replace Engine with a mock.
- More Flexibility: You can switch between different Engine types.
- Better Code: Classes are cleaner and more reusable.

Here's why Dependency Injection is used and its advantages:

1. Loose Coupling:** DI promotes loose coupling between components, making it easier to maintain, extend, and test the application. Components don't need to know how their dependencies are created or configured; they simply rely on the provided interfaces.

2. Reusability:** Components can be reused across different parts of the application, as they are not tightly bound to specific implementations of their dependencies.

3. Testability:** DI allows for easier unit testing. You can inject mock or stub dependencies during testing to isolate and verify the behavior of individual components.

4. Configurability:** DI enables you to configure the application's components and dependencies, often through configuration files or annotations, without changing the code. This makes the application more adaptable to different environments or configurations.

Here's a simple example of Dependency Injection in a Spring Boot application:

UserService.java:

```java

```
@Service
public class UserService {
 private final UserRepository userRepository;

 @Autowired
 public UserService(UserRepository userRepository) {
 this.userRepository = userRepository;
 }

 public User getUserById(Long id) {
 return userRepository.findById(id).orElse(null);
 }
}

```

```
UserRepository.java:
```java  
public interface UserRepository extends JpaRepository<User, Long> {  
}  
***
```

In this example:

- `UserService` is a Spring service class that depends on `UserRepository` for fetching user data.

- ` UserRepository` is an interface that extends `JpaRepository`. Spring Data JPA will provide the actual implementation of this interface.

- Constructor injection is used in `UserService` to receive an instance of `UserRepository`. The `@Autowired` annotation tells Spring to inject the dependency.

- By using DI, `UserService` is loosely coupled with the concrete implementation of `UserRepository`. The Spring IoC container takes care of creating and providing instances of `UserRepository` to `UserService`.

In your Spring Boot application, you can configure and define beans in various ways, such as through annotations or XML configuration files. Spring Boot's auto-configuration and component scanning capabilities help simplify the DI process, allowing you to focus on your application's logic rather than the details of object creation and wiring.

Spring IOC Container :

The Spring IOC (Inversion of Control) container creates objects of classes and manages their lifecycle, dependencies, and configurations.

It also controls the instantiation, wiring, and lifecycle of these objects, allowing them to be loosely coupled and promoting modular, maintainable, and scalable application development.

There are two main types of Spring containers: the **BeanFactory** (a more basic container with lazy-loading of beans) and the more feature-rich **ApplicationContext** (a superset of BeanFactory with additional functionalities and eager-loading of beans).

Bean Factory :

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

public class BeanFactoryExample {
    public static void main(String[] args) {
        BeanFactory factory = new XmlBeanFactory(new
ClassPathResource("beans.xml"));

        // Retrieving a bean from BeanFactory
        MyBean bean = (MyBean) factory.getBean("myBean");

        // Using the bean
        bean.doSomething();
    }
}
```

ApplicationContext :

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
public class ApplicationContextExample {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
```

```
// Retrieving a bean from ApplicationContext  
  
MyBean bean = (MyBean) context.getBean("myBean");  
  
// Using the bean  
  
bean.doSomething();  
  
}  
  
}
```

NOTE : @SpringBootApplication is typically used in Spring Boot applications. When you annotate your main application class with @SpringBootApplication, it implicitly creates an ApplicationContext. Internally, Spring Boot uses ApplicationContext rather than BeanFactory for managing beans and providing various application services.

Annotations in simple terms :

=====

In Spring Boot, annotation refers to a special type of metadata that we can add to our code to provide additional information or instructions to the Spring framework. Annotations are used to simplify and streamline the development process by reducing the amount of configuration code we need to write.

Annotations in Spring Boot serve several purposes. Firstly, they help in defining the behavior of our application components, such as controllers, services, and repositories. By adding annotations to these components, we can specify their roles and responsibilities within the application.

Secondly, annotations enable Spring Boot to automatically configure and manage various aspects of our application. For example, by using the @Autowired annotation, we can let Spring Boot handle the dependency injection of required dependencies into our classes.

1. @Component :

- @Component indicates that an annotated class is a "Spring Bean / Component".
 - @Component tells the spring container to automatically create the spring bean.
 - Spring IOC container creates the object of the class and manage that object. You do not need to use the `new` keyword to create an object.
 - Spring Container will give the name of the bean as Class name with first letter is small.(StudentComponent --> studentComponent) we can give our name also.

example:

```
@Component("studentComponent")  
class StudentComponent{  
}
```

2. @Autowired :

- The @Autowired annotation is used for automatic dependency injection
- The @Autowired annotation is used to inject the bean automatically.
- The @Autowired annotation is used in field injection, constructor injection and setter injection.

```
// 1. Field Injection
```

```
@Autowired
```

```
private PaymentService paymentService;
```

```
// 2. Constructor Injection
```

```
private final PaymentService paymentService;
```

```
@Autowired
```

```
public OrderServiceConstructor(PaymentService paymentService) {
```

```
    this.paymentService = paymentService;
```

```
}
```

```
// 3. Setter Injection
```

```
private PaymentService paymentService;
```

```
@Autowired
```

```
public void setPaymentService(PaymentService paymentService) {
```

```
    this.paymentService = paymentService;
```

```
}
```

3. @Qualifier :

- @Qualifier annotation is used in conjunction with @Autowired to avoid confusion when we have two or more beans configured for same type.

Example :

```
public interface MessageService {
```

```
    void sendMessage(String message);

}

@Component
@Qualifier("email")
public class EmailService implements MessageService {

    @Override
    public void sendMessage(String message) {
        System.out.println("Sending email: " + message);
    }
}

@Component
@Qualifier("sms")
public class SMSService implements MessageService {

    @Override
    public void sendMessage(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

@Component
public class NotificationService {
    private final MessageService messageService;
```

```

// Constructor injection with @Qualifier
@Autowired
public NotificationService(@Qualifier("email") MessageService
messageService) {
    this.messageService = messageService;
}

public void notifyUser(String message) {
    messageService.sendMessage(message);
}

```

4. @Primary :

- We use @Primary Annotation to give higher preference to a bean when there are multiple beans of the same type.

- based on above example almost same : use @Primary instead of using @Qualifier. EmailService bean is having a higher preference of it's same type(MessageService).

```

@Component
@Primary
public class EmailService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println("Sending email: " + message);
    }
}

```

5. @Bean :

- `@Bean` annotation indicates that a method produces a bean to be managed by the Spring Container.

- Then `@Bean` annotation is usually declared in Configuration class to create Spring Bean definitions.

- By default, the bean name is as method name. we can specify bean name using `@Bean(name="beanName")`.

- `@Bean` annotation provides `initMethod` and `destroyMethod` attributes to perform certain actions after bean initialization or before bean destruction by a container.

- when you declare a class as a bean in Spring Boot, you're essentially telling Spring, "Hey, I want you to manage this object for me, and make it available for use throughout my application."

- This allows Spring to handle things like dependency injection, lifecycle management, and more, making your application easier to develop and maintain.

Note : in simple words -> "Bean" in Spring Boot is just a regular Java object managed by the Spring framework. It's called a "bean" because it's like a building block that Spring uses to construct and manage your application.

Example :

```
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class AppConfig {  
  
    @Bean(name = "userService")  
    public MessageService userService() {  
        return new UserService();  
    }  
}
```

```
@Bean(name = "emailService")
public MessageService emailService() {
    return new EmailService();
}

}
```

// we can use initMethod and destroyMethod like below example
//for example1, you want to insert records in database table and delete
the records from database table.

```
@Bean(initMethod="init", destroyMethod="destroy")
public StudentController studentController(){
    return new StudentController();
}
```

StudentController {

 public void init(){
 // you can write the logic to insert the records while application is
 being started.
 }
}

```
    public void destroy(){  
        // you can write the logic to delete the records while applicatino is  
        being shutdown.  
    }
}
```

```
}
```

When to Use @Bean:

When you can't modify the source code (e.g. external library example : RestTemplate, WebClient).

When using complex object creation/configuration.

When conditional bean creation is needed.

6. @Lazy :

- By default , Spring creates all singleton bean eagerly at the startup/bootstraping of the application context. However, sometimes you might want to defer the initialization of a bean until it's actually needed, which can help improve startup time and reduce memory usage.

- You can load the Spring beans lazily(on-demand) using @Lazy annotation.

- that mean, at application startup , spring container will not create a bean for the particlur class which was annotated with @Component and @Lazy. Spring Container will create bean only on-demand.

- @Lazy annotation is used with @Configuration , @Component and @Bean.

- Eager initialization is recommended : to avoid and detect all possible errors immediately rather than at runtime.

7. @Scope :

- The scope of a bean defines the life cycle and visibility of that bean in the contexts we use it.

- @Scope annotation is used to define the scope of the bean.

- We use `@Scope` to define the scope of `@Component` class or a `@Bean` definition.

Scope OF Beans In Spring

The latest version of the Spring framework defines 6 types of scopes:

1. singleton - only one instance of the bean is created and shared across the entire application. This is default scope. `@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)` or `@Scope(value="singleton")` - It will give you only one bean with same hashcode if you try to create/call the beans and print multiple times.

2. prototype - new Instance of the bean is created everytime it is requested. `@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)` or `@Scope(value = "prototype")` - It will give you different beans with different hashcodes if you try to create/call the beans and print multiple times.

3. request : A new instance of the bean is created for each HTTP request. This scope is only available in a web-aware application. Beans with request scope are destroyed at the end of each HTTP request.`@Scope(value = WebApplicationContext.SCOPE_REQUEST)`

4. session: A single instance of the bean is created for each HTTP session. This scope is only available in a web-aware application. Beans with session scope persist throughout the duration of the HTTP session. `@Scope(value = WebApplicationContext.SCOPE_SESSION)`

5. application : A single instance of the bean is created for the entire lifecycle of the application context. This scope is not tied to any particular HTTP request or session and is available across the entire application. It is equivalent to the "singleton" scope, but with a broader application context.`@Scope(value = WebApplicationContext.SCOPE_APPLICATION)`

6. websocket : A single instance of the bean is created for each WebSocket connection. This scope is only available in applications that use WebSocket communication. Beans with websocket scope persist for the duration of the WebSocket connection.`@Scope(value = "websocket", proxyMode = ScopedProxyMode.TARGET_CLASS)`

The last four scopes mentioned, request, session, application and websocket, are only available in a web-aware application.

8. @Value :

- @Value annotation is used to assign default values to variables and method arguments or constructor arguments or setters.
- @Value annotation is used to get value for specific property key from the property file.
- We can read spring environment variables as well as system variables using @Value annotation.

ex:

```
@Value("nani")  
private String;
```

```
@Value("${email}") // the value of email is provided in application.properties file.  
private String;
```

9. @SpringBootApplication :

- This annotation is used to mark the main class of a Spring Boot application.
- It combines three annotations: `@Configuration` or `@SpringBootConfiguration`, `@EnableAutoConfiguration`, and `@ComponentScan`

(i) @Configuration :

- @Configuration is used to indicate that a class defines one or more bean definition methods.

- These methods are used to configure the Spring application context and define beans.

- When Spring Boot scans the classpath for bean definitions, it looks for classes annotated with `@Configuration` and processes their bean definitions.

(ii) `@EnableAutoConfiguration` :

- `@EnableAutoConfiguration` tells Spring Boot to automatically configure the Spring application based on the dependencies and the environment.

- It scans the classpath for configuration classes and applies automatic configuration based on the dependencies present.

- It helps reduce the amount of configuration required in Spring Boot applications by automatically configuring beans, components, and other settings based on sensible defaults.

(iii) `@ComponentScan`:

- `@ComponentScan` is used to specify the base package(s) to scan for Spring-managed components, such as `@Component`, `@Service`, `@Repository`, and `@Controller`.

- It tells Spring where to look for components to create beans, allowing Spring to find and register them in the application context.

- If not specified, `@ComponentScan` defaults to scanning the package of the class with this annotation

10. `@ConfigurationProperties(prefix = "application", ignoreUnknownFields = false)` - Properties are configured in the `application.properties` file

11. **`@Transactional`**:

- `@Transactional` is used to mark methods or classes as transactional, meaning that database operations within the annotated method or class will be executed within a transaction boundary.

- It ensures that either all database operations within the annotated method or class complete successfully or none of them do. If an exception occurs, the transaction is rolled back.

- It's typically used at the service layer to define transactional boundaries around service methods.

Commit: When a transaction is committed, all changes made within the transaction are permanently saved to the database. The changes become visible to other transactions.

Rollback: If an error occurs or an exception is thrown within the transaction, the changes made within the transaction are rolled back, meaning that the database is restored to its state before the transaction began.

Commit - With `@Transactional(noRollbackFor = Exception.class)`, you're telling the system to continue with the transaction even if there are minor issues, ensuring that your data is saved even in case of non-critical errors. Commit by default: If no exceptions occur during the execution of the method annotated with `@Transactional`, the transaction will be committed by default.

Rollback (default) - With `@Transactional(rollbackFor = Exception.class)`, you're ensuring that if any problem occurs, the entire transaction will be canceled to maintain data integrity, ensuring that no partial or incorrect data is saved. Rollback for unchecked exceptions: If an unchecked exception (subclass of `RuntimeException` or `Error`) is thrown during the execution of the method annotated with `@Transactional`, the transaction will be rolled back by default.

`@Transactional(readOnly = true)` : when the method only needs to read data from the database and should not perform any write operations. This can improve performance and reduce the risk of unintended data modifications.

- Only read operations (e.g., SELECT queries) are allowed.
- Write operations (e.g., INSERT, UPDATE, DELETE queries) are not allowed.

`@Transactional(readOnly = false)`: (or simply `@Transactional`) when the method needs to perform both read and write operations on the database.

12. **`@Modifying`**:

- `@Modifying` is used in conjunction with `@Query JPQL` (Java Persistence Query Language) queries to indicate that the query will modify the database state.
 - It's used when executing UPDATE, DELETE, or INSERT queries using JPQL.
 - This annotation is required to instruct JPA that the query being executed will perform database modifications, enabling it to properly manage transactional behavior.

Example :

```
@Modifying  
 @Query("UPDATE Customer C SET C.currentStatus = :status WHERE C.customerId  
 = :customerId")  
 void updateCurrentStatus(@Param("customerId") Integer customerId,  
 @Param("status") String status);
```

13. **`@Query`**:

- `@Query` is used to declare JPQL queries directly within repository methods.

- It allows developers to define custom queries using JPQL syntax directly within the repository interface or class.
- It provides more flexibility compared to query methods derived from method names.

14. **nativeQuery**:

- `nativeQuery` is used to execute native SQL queries directly within repository methods.
- It allows developers to execute native SQL queries (SQL written in the database-specific dialect) within the repository interface or class.
- It's useful when JPQL is not sufficient or when you need to execute database-specific queries.

Example :

```

@Query(value = "SELECT distinct U.user_id, U.sales_person_code, U.first_name,
U.last_name" +
    " FROM [User] U " +
    " JOIN viewable_team VT ON VT.sales_person_code =
U.sales_person_code AND VT.team_code in (select value from string_split(:teamCodes, ','))" +
    " AND VT.is_primary_ae = 'true'" +
    " GROUP BY U.user_id, U.sales_person_code, U.first_name, U.last_name",
nativeQuery = true)

List<Object[]> findAEUsers(@Param("teamCodes") String teamCodes);

```

15. **@EnableWebSecurity**:

- is an annotation in Spring Security that activates web security features in a Spring Boot application.

- It's usually paired with a custom configuration class extending `WebSecurityConfigurerAdapter`, where you can define security rules like access control, login, and logout settings to secure your web application

16. `@RestController`: This annotation is used to define a RESTful controller class. It combines the `@Controller` and `@ResponseBody` annotations, making it easier to handle HTTP requests and responses.

17. `@RequestMapping`: This annotation is used to map HTTP requests to specific methods in a controller class. It allows you to define the URL path and HTTP method for each request mapping.

18. `@Autowired`: This annotation is used for automatic dependency injection. It allows Spring to automatically wire dependencies into your beans, reducing the need for manual configuration.

19. `@Component`: This annotation is a generic stereotype for any Spring-managed component. It is used to mark a class as a Spring bean, allowing it to be automatically detected and instantiated by the Spring container.

20. `@Service`: This annotation is used to mark a class as a service component. It is typically used to define business logic or service layer components.

21. `@Repository`: This annotation is used to mark a class as a repository component. It is typically used to define data access objects (DAOs) or repositories for database operations.

22. `@Transient`: This is an annotation used in conjunction with JPA (Java Persistence API) entities to indicate that a field should not be persisted to the database.

This annotation is typically used when you have fields in your entity class that should not be stored in the database, either because they are derived/calculated fields, transient data, or simply not relevant for persistence.

Example :

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Transient;

@Entity
@Table
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    // This field will not be persisted(stored) to the database
    @Transient
    private transient String temporaryData;

    // Getters and setters
}
```

23. `@Profile`: annotation is used to specify which beans or configurations should be active or inactive based on the environment or profile in which the application is running.

Profiles are a way to segregate parts of your application's configuration and code based on different environments (such as development, testing, production) or specific use cases.

Here's how you can use `@Profile` in Spring Boot:

a. ****Annotate Beans****: You can annotate Spring beans with `@Profile` to specify that they should only be created when a certain profile is active. For example:

```
```java
@Component
@Profile("development")
public class DevelopmentDataSource implements DataSource {
 // Development-specific configuration
}

@Component
@Profile("production")
public class ProductionDataSource implements DataSource {
 // Production-specific configuration
}
...```

```

b. **Specify Active Profiles**: You can specify active profiles using the `spring.profiles.active` property in your `application.properties` or `application.yml` file. For example:

```
```properties
spring.profiles.active=development
```
```

```

You can also set active profiles programmatically or through command-line arguments.

c. **Conditional Configuration**: You can use `@ConditionalOnProperty` along with `@Profile` to conditionally enable or disable certain configurations based on properties or profiles.

```
```java
@Configuration
@Profile("production")
@ConditionalOnProperty(name = "custom.property", havingValue =
"true")
public class ProductionConfig {
 // Configuration for production environment when
 // custom.property is set to true
}
```
```

```

d. **Spring Boot Profiles**: Spring Boot provides several predefined profiles such as `default`, `dev`, `test`, and `prod`. The `default` profile is active when no other profiles are specified.

Using `@Profile`, you can create environment-specific configurations, beans, or components, making it easier to manage and customize your Spring Boot application for different deployment environments.

24. `@Async`: annotation in Spring Boot allows methods to be executed asynchronously in separate threads, improving performance and responsiveness by parallelizing non-blocking tasks such as I/O operations or long-running computations.

Use it when you need to offload tasks that don't depend on each other's results and can be executed concurrently.

Certainly! Here's a simple code example demonstrating the usage of `@Async` annotation in a Spring Boot application:

```
```java
import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Service;

@Service
public class MyAsyncService {

    @Async
    public void asyncMethod() {
        // Simulate a long-running task
        try {
            Thread.sleep(5000); // Sleep for 5
seconds
        }
    }
}
```

```

        System.out.println("Async method
completed");

    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();

        System.err.println("Async method
interrupted");

    }

}

...

```

In this example:

- We have a `MyAsyncService` class annotated with `@Service`, indicating that it's a Spring-managed bean.
 - The `asyncMethod()` is annotated with `@Async`, indicating that it should be executed asynchronously in a separate thread.
 - Inside the `asyncMethod()`, we simulate a long-running task by sleeping the thread for 5 seconds.
 - When calling `asyncMethod()`, it will return immediately, and the actual execution will be performed asynchronously in a separate thread.

To enable asynchronous processing in your Spring Boot application, ensure that you have `@EnableAsync` annotation at the configuration level, usually in your main application class:

```

```java
import org.springframework.boot.SpringApplication;

```

```
import
org.springframework.boot.autoconfigure.SpringBootApplication;

import
org.springframework.scheduling.annotation.EnableAsync;

@SpringBootApplication
@EnableAsync
public class MyApplication {

 public static void main(String[] args) {

 SpringApplication.run(MyApplication.class,
args);

 }

}

...

With this setup, you can call `asyncMethod()` from any
other component or service in your application, and it will be executed asynchronously, allowing
the calling thread to continue without waiting for the method to complete.
```

Note : One Important Class

What is CompletableFuture?

=====

A CompletableFuture is used for asynchronous programming.  
Asynchronous programming means writing non-blocking code.

It runs a task on a separate thread than the main application thread and notifies the main thread about its progress, completion or failure.

In this way, the main thread does not block or wait for the completion of the task. Other tasks execute in parallel. Parallelism improves the performance of the program.

A CompletableFuture is a class in Java. It belongs to `java.util.concurrent` package. It implements `CompletionStage` and `Future` interface.

To create an instance of `CompletableFuture`, we can use the static method `supplyAsync` provided by `CompletableFuture` class which takes `Supplier` as an argument. `Supplier` is a Functional Interface that takes no value and returns a result.

`supplyAsync()`: It complete its job asynchronously. The result of supplier is run by a task from `ForkJoinPool.commonPool()` as default. The `supplyAsync()` method returns `CompletableFuture` on which we can apply other methods.

`thenApply()`: The method accepts function as an arguments. It returns a new `CompletableFuture` when this stage completes normally. The new stage use as the argument to the supplied function.

`join()`: the method returns the result value when complete. It also throws a `CompletionException` (unchecked exception) if completed exceptionally.

```
list.stream()
 .map(num -> CompletableFuture.supplyAsync(() -> num *
num))
 .map(CompletableFuture ->
CompletableFuture.thenApply(num -> num * num))
 .map(value -> value.join())
 .forEach(someData -> System.out.println(someData));
```

## Spring-Boot Actuator

---

Developing and Managing an application are the two most important aspects of the application's life cycle. It is very crucial to know what's going on beneath the application.

Also when we push the application into production, managing it gradually becomes critically important. Therefore, it is always recommended to monitor the application both while at the development phase and at the production phase.

For the same use case, Spring Boot provides an actuator dependency that can be used to monitor and manage your Spring Boot application, By /actuator and /actuator/health endpoints you can achieve the purpose of monitoring.

With the help of Spring Boot, we can achieve the above objectives.

Spring Boot's 'Actuator' dependency is used to monitor and manage the Spring web application.

We can use it to monitor and manage the application with the help of HTTP endpoints or with the JMX.

```
<dependencies>
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-actuator</artifactId>
 </dependency>
</dependencies>
```

```
management.endpoints.web.exposure.include=*
```

IDs : beans, caches , details, health, mappings etc.

example : <http://localhost:9999/actuator> or <http://localhost:9999/actuator/mappings>

### Spring-Boot DevTools Dependencies

---

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-devtools</artifactId>
 <scope>runtime</scope>
 <optional>true</optional>
</dependency>
```

#### Advantages:

- Automatic restart for faster development iterations.
- LiveReload support for automatic browser refresh on static resource changes.
- Enhanced error handling with detailed messages and stack traces.
- Remote development support for debugging and code swapping.
- Additional utilities like property defaults and template caching.

#### Disadvantages:

- Potential performance overhead, especially in larger projects.

- Increased complexity in the development environment.
- Possible compatibility issues with third-party libraries.
- Not suitable for production use due to overhead and risks.
- Adds to project dependency footprint, potentially complicating dependency management.

Note : **\*\*Changes May Not Always be Detected\*\***: DevTools primarily monitors changes in the classpath, such as modifications to Java classes and resources.

However, changes outside of the classpath, like alterations to configuration files or dependencies, may not trigger an automatic restart.

This limitation exists because DevTools focuses on the classpath for efficiency and may not have visibility into changes beyond it.

#### DIFFERENT TYPES OF QUERIES :

---

In Spring Boot, you can write various types of queries to interact with your database. Here are some common types of queries along with examples:

##### 1. **\*\*JPQL Queries:\*\***

- JPQL (Java Persistence Query Language) queries are object-oriented queries defined in terms of entity objects.

```
```java
@Query("SELECT e FROM Employee e WHERE e.department = ?1")
List<Employee> findByDepartment(Department department);
````
```

## 2. \*\*Native SQL Queries:\*\*

- Native SQL queries are SQL queries written in the native database dialect.

```
```java
```

```
@Query(value = "SELECT * FROM employees WHERE department_id = ?1",  
nativeQuery = true)
```

```
List<Employee> findByDepartmentIdNative(Long departmentId);
```

```
```
```

## 3. \*\*Named Queries:\*\*

- Named queries are predefined queries with a name specified in the entity class.

```
```java
```

```
@NamedQuery(name = "Employee.findByLastName", query = "SELECT e FROM  
Employee e WHERE e.lastName = :lastName")
```

```
List<Employee> findByLastName(@Param("lastName") String lastName);
```

```
```
```

## 4. \*\*Criteria Queries:\*\*

- Criteria API provides a type-safe way to define queries programmatically without using any query strings.

```
```java
```

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();
```

```
CriteriaQuery<Employee> criteria = builder.createQuery(Employee.class);
```

```
Root<Employee> root = criteria.from(Employee.class);
```

```
criteria.select(root).where(builder.equal(root.get("department"), department));
```

```
List<Employee> employees = entityManager.createQuery(criteria).getResultList();
```

```
```
```

## 5. \*\*Query by Example (QBE):\*\*

- Query by Example allows creating queries based on the example entity provided.

```
```java
```

```
Employee exampleEmployee = new Employee();
exampleEmployee.setLastName("Doe");
Example<Employee> example = Example.of(exampleEmployee);
List<Employee> employees = employeeRepository.findAll(example);
```

```

## 6. \*\*Derived Query Methods:\*\*

- Spring Data JPA provides the ability to derive query methods from the method name.

```
```java
```

```
List<Employee> findByLastName(String lastName);
```

```

SQL :

-----

Example SQL query to retrieve all employees from an "Employees" table

```
SELECT * FROM Employees;
```

HQL :

-----

```
// Example HQL query to retrieve all employees using Hibernate
Query query = session.createQuery("FROM Employee");
List<Employee> employees = query.list();
```

JPQL :

=====

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
 @Query("SELECT e FROM Employee e WHERE e.name LIKE %:name%")
 List<Employee> findByNameContaining(@Param("name") String name);
}
```

These are some common types of queries you can write in Spring Boot applications to interact with your database. The choice of query type depends on factors like complexity, performance, and maintainability of the application.

SpringB0ot Application Flow :

=====

Here's a simplified explanation of what happens when you run a Spring Boot application:

1. The `main` method of the main application class is executed.
2. Spring Boot initializes the application context and sets up the Spring IoC container.
3. Component scanning is performed to identify and register beans annotated with `@Component`, `@Service`, `@Repository`, and other stereotype annotations.
4. Spring Boot enables auto-configuration based on the dependencies and configurations present in the classpath.
5. Configuration classes annotated with `@Configuration` are processed, and beans defined in these classes are registered with the application context.

6. The embedded web server (e.g., Tomcat, Jetty) is started if the application includes web dependencies.

7. Request mapping and endpoint handling are configured based on controllers and REST endpoints defined in the application.

8. The application is ready to handle incoming requests and perform business logic as defined in the application components.

## Synchronous Communication & ASynchronous Communication

---

### 1. \*\*Synchronous Communication\*\*:

- In synchronous communication, the sender waits until the receiver processes the message and sends back a response before continuing further execution.

- This means that the sender and receiver are synchronized in their communication, and each operation is blocking until completion.

- In a Java Spring Boot application, synchronous communication often occurs through traditional HTTP request-response mechanisms, where the client sends a request to the server and waits for the server to process the request and send back a response.

- In an Angular application, synchronous communication can involve making HTTP requests using Angular's HttpClient module, where the application waits for the response before proceeding with further actions.

### 2. \*\*Asynchronous Communication\*\*:

- In asynchronous communication, the sender does not wait for the receiver to process the message. Instead, it continues its execution, and the receiver processes the message whenever it's ready.

- Asynchronous communication is non-blocking, allowing systems to perform other tasks while waiting for a response.

- In a Java Spring Boot application, asynchronous communication can be achieved using asynchronous programming techniques such as CompletableFuture or reactive programming frameworks like Spring WebFlux.

- In an Angular application, asynchronous communication is commonly implemented using asynchronous operations like Promises or Observables. This allows the application to continue functioning while waiting for responses from external services or APIs.

In summary, synchronous communication blocks execution until a response is received, while asynchronous communication allows for non-blocking operation, enabling better utilization of resources and potentially improved performance in applications. Both approaches have their use cases, and the choice between them depends on factors such as performance requirements, scalability, and the nature of the application's tasks.

#### Microservices :

---

**\*\*Microservices\*\*** is an architectural approach where a software application is divided into a collection of small, independent, and loosely coupled services, each responsible for specific functionality.

Spring Boot, a popular Java framework, is often used to build microservices due to its ease of development and the tools it provides.

#### Importants in Microservices

---

### API Gateway, Load Balancing, and Discovery Service using Spring Cloud Eureka:

1. **\*\*API Gateway\*\***: An API Gateway is a server that acts as an entry point for all client requests. It provides a single point of entry to the system, enabling various functions such as authentication, routing, load balancing, etc. Spring Cloud provides a module called Spring Cloud Gateway for implementing API Gateways.

in simple terms : It's like a traffic policeman at a busy intersection, directing cars (requests) to different lanes (services) based on their destination, and ensuring smooth flow of traffic (data).

steps to create API-Gateway :

1.     @EnableEurekaClient is used to make the API gateway service act as a client that registers itself with the Discovery Service (Eureka Client).
2.     Define your gateway routes in the application.properties or application.yml file. For example:

```
Product-Service Route

"lb://product-service" uses Spring Cloud LoadBalancer to lookup
instances of the "product-service" registered with Eureka.

#spring.cloud.gateway.routes[0].uri=lb://product-service configures the
URI for the "product-service" route to use load-balanced instances retrieved from Eureka

spring.cloud.gateway.routes[0].id=product-service

spring.cloud.gateway.routes[0].uri=lb://product-service

spring.cloud.gateway.routes[0].predicates[0]=Path=/api/product/**
```

2. \*\*Load Balancing\*\*: Load balancing distributes incoming client requests across multiple backend servers to ensure optimal resource utilization, reliability, and scalability. In Spring Cloud, load balancing can be achieved using client-side load balancing or server-side load balancing.

in simple terms : Imagine you're at a buffet with multiple food stations. Instead of everyone crowding at one station, people spread out evenly to different stations, ensuring no station runs out of food and everyone gets served efficiently.

3. **Discovery Service**: A discovery service is responsible for maintaining a registry of available services and their instances. Spring Cloud Eureka is a popular service discovery solution that allows services to register themselves and discover other services.

in simple terms : Think of a library where books are stored in different sections, but you're not sure where to find the book you need. The discovery service acts like a librarian who knows where every book is located. Instead of searching through each aisle yourself, you can ask the librarian for the location of the book you want. The librarian then directs you to the right section, saving you time and effort.

`@EnableEurekaServer` is used to make the service act as a Discovery Service (Eureka Server).

`@EnableEurekaClient` is used to make the service act as a client that registers itself with the Discovery Service (Eureka Client).

#### #### Server-side Load Balancing vs Client-side Load Balancing:

##### Server-side Load Balancer:

---

- Server-side load balancers handle the distribution of incoming requests among multiple backend servers.
- They are responsible for routing client requests to appropriate backend servers based on predefined algorithms.
- Spring Cloud and Eureka Discovery Service falls under the category of Server-side Load Balancer

##### Example :

```
import org.springframework.cloud.gateway.route.RouteLocator;
import org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class GatewayConfig {

 @Bean
 public RouteLocator customRouteLocator(RouteLocatorBuilder
builder) {
 return builder.routes()
 .route("service1", r -> r.path("/service1/**"))
 .uri("lb://SERVICE-NAME"))

 // Replace SERVICE-NAME with the name of your service
 .route("service2", r -> r.path("/service2/**"))
 .uri("lb://SERVICE-NAME"))

 // Replace SERVICE-NAME with the name of your service
 .build();
 }
}

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.cloud.client.discovery.EnableDiscoveryClient;
```

```
@SpringBootApplication
 @EnableDiscoveryClient
 public class GatewayApplication {
 public static void main(String[] args) {
 SpringApplication.run(GatewayApplication.class, args);
 }
 }
```

#### #### Client-side Load Balancing:

---

- Client-side load balancing involves distributing requests among multiple backend servers directly from the client side, typically using a load balancing library or framework.
- In client-side load balancing, the client application is responsible for selecting the appropriate backend server to send the request to.
- The client retrieves the list of available servers from a service registry (such as Eureka) and applies a load balancing algorithm locally.
- This approach reduces the load on server-side load balancers and enables more flexible load balancing strategies.

When you annotate a `WebClient.Builder` or `RestTemplate` bean with `@LoadBalanced`, it enables client-side load balancing using Ribbon, which is part of Spring Cloud Netflix.

Spring Cloud Netflix Ribbon performs client-side load balancing by choosing an instance of the target service from the available instances registered with the Eureka server.

Therefore, by using `@LoadBalanced` with `WebClient.Builder` or `RestTemplate`, you are enabling client-side load balancing, where the client makes decisions about which service instance to call.

```
@Bean
@LoadBalanced
public WebClient.Builder webClientBuilder(){
 return WebClient.builder();
}

@Bean
@LoadBalanced
public RestTemplate restTemplate(){
 return new RestTemplate();
}

@SpringBootApplication
@EnableEurekaClient
public class ApiGatewayApplication {
 public static void main(String[] args) {
 SpringApplication.run(ApiGatewayApplication.class, args);
 }
}
```

### RestTemplate vs WebClient:

```
=====
```

```
@RestController
public class ServiceBController {

 @GetMapping("/hello")
 public String hello() {
 return "Hello from Service B!";
 }
}
```

- **RestTemplate**: RestTemplate is a synchronous HTTP client provided by Spring for making RESTful API calls. It's been around since Spring 3 and is widely used in Spring-based applications. It's built on top of the Java URLConnection framework and provides a convenient way to interact with RESTful services.

```
=====
```

Example :

```
@Autowired
private RestTemplate restTemplate;

@GetMapping("/call-service-b")
public String callServiceB() {
 String serviceBUrl = "http://service-b:8081";
 return restTemplate.getForObject(serviceBUrl + "/hello", String.class);
```

```
}
```

- **WebClient**: WebClient is a non-blocking, reactive HTTP client introduced in Spring WebFlux, which is part of Spring 5. It's designed for asynchronous, event-driven applications and supports both synchronous and asynchronous communication. WebClient is ideal for building reactive microservices and handling high concurrency.

```
=====
```

Example :

```
@Autowired
```

```
private WebClient.Builder webClientBuilder;
```

```
@GetMapping("/call-service-b")
public Mono<String> callServiceB() {
 return webClientBuilder.build()
 .get()
 .uri("http://service-b:8081/hello")
 .retrieve()
 .bodyToMono(String.class);
}
```

Here's your complete version of `RestTemplate`, `WebClient`, and `FeignClient` usage — with proper HTTP method comments, endpoint URLs, and return types stored in variables.

---

## ## ● \*\*1. RestTemplate – All Methods With Comments\*\*

### ✅ `GET` → `/hello`

```java

// GET Request

```
String url = "http://localhost:8081/hello";
```

```
String response = restTemplate.getForObject(url, String.class);
```

```
return response;
```

```

### ✅ `GET` with ResponseEntity → `/hello`

```java

// GET Request

```
String url = "http://localhost:8081/hello";
```

```
ResponseEntity<String> response = restTemplate.getEntity(url, String.class);
```

```
return response.getBody();
```

```

### ✅ `POST` → `/post`

```java

// POST Request

```
String url = "http://localhost:8081/post";
```

```
MyRequest request = new MyRequest("test");
```

```
String response = restTemplate.postForObject(url, request, String.class);
```

```
    return response;
```

```
    ...
```

```
#### ✅ `POST` with ResponseEntity → `/post`
```

```
```java
```

```
// POST Request
```

```
String url = "http://localhost:8081/post";
```

```
ResponseEntity<String> response = restTemplate.postForEntity(url, request,
String.class);
```

```
return response.getBody();
```

```
...
```

```
✅ `PUT` → `/update`
```

```
```java
```

```
// PUT Request
```

```
String url = "http://localhost:8081/update";
```

```
restTemplate.put(url, request);
```

```
return "Put executed";
```

```
...
```

```
#### ✅ `DELETE` → `/delete/1`
```

```
```java
```

```
// DELETE Request
```

```
String url = "http://localhost:8081/delete/1";
```

```
restTemplate.delete(url);
```

```
return "Deleted successfully";
```

---

### ✅ `exchange()` with Header → `/hello`

```java

// GET Request with Headers

```
String url = "http://localhost:8081/hello";  
HttpHeaders headers = new HttpHeaders();  
headers.set("Authorization", "Bearer token");  
HttpEntity<Void> entity = new HttpEntity<>(headers);
```

```
ResponseEntity<String> response = restTemplate.exchange(url, HttpMethod.GET, entity,  
String.class);
```

return response.getBody();

● **2. WebClient – Full Methods With URL and HTTP Methods**

✅ Setup

```java

```
WebClient webClient = WebClient.builder().build();
```

---

### ✅ `GET` → `/hello`

```java

```
// GET Request

Mono<String> response = webClient.get()

    .uri("http://localhost:8081/hello")

    .retrieve()

    .bodyToMono(String.class);

return response;

```
```

### ✅ `POST` → `/post`

```
```java

// POST Request

MyRequest request = new MyRequest("test");
```

```
Mono<String> response = webClient.post()

    .uri("http://localhost:8081/post")

    .bodyValue(request)

    .retrieve()

    .bodyToMono(String.class);

return response;

```
```

### ✅ `PUT` → `/update`

```
```java

// PUT Request

Mono<String> response = webClient.put()

    .uri("http://localhost:8081/update")
```

```
.bodyValue(request)
.retrieve()
.bodyToMono(String.class);

return response;
```

```

####  `DELETE` → `/delete/1`

```
```java
// DELETE Request

Mono<Void> response = webClient.delete()

    .uri("http://localhost:8081/delete/1")
    .retrieve()
    .bodyToMono(Void.class);

return response;
```

```

####  `GET` with Header → `/hello`

```
```java
// GET Request with Authorization Header

Mono<String> response = webClient.get()

    .uri("http://localhost:8081/hello")
    .header("Authorization", "Bearer token")
    .retrieve()
    .bodyToMono(String.class);

return response;
```

```

---

### ## ● \*\*3. FeignClient – Declarative with URLs and HTTP Methods\*\*

#### ### ✓ Feign Client Interface

```java

```
@FeignClient(name = "demo-client", url = "http://localhost:8081")  
public interface DemoFeignClient {  
  
    // GET Request  
    @GetMapping("/hello")  
    String getHello();  
  
    // POST Request  
    @PostMapping("/post")  
    String createSomething(@RequestBody MyRequest request);  
  
    // PUT Request  
    @PutMapping("/update")  
    void updateSomething(@RequestBody MyRequest request);  
  
    // DELETE Request  
    @DeleteMapping("/delete/{id}")  
    void deleteSomething(@PathVariable("id") Long id);  
}
```

``

✅ Controller using FeignClient

```java

@RestController

public class FeignClientController {

    @.Autowired

    private DemoFeignClient demoFeignClient;

    // GET Request

    @GetMapping("/feign/hello")

    public String callHello() {

        String response = demoFeignClient.getHello();

        return response;

    }

    // POST Request

    @PostMapping("/feign/create")

    public String create() {

        MyRequest request = new MyRequest("data");

        String response = demoFeignClient.createSomething(request);

        return response;

    }

    // PUT Request

```
@PutMapping("/feign/update")
public String update() {
 MyRequest request = new MyRequest("updated");
 demoFeignClient.updateSomething(request);
 return "Updated";
}

// DELETE Request
@DeleteMapping("/feign/delete/{id}")
public String delete(@PathVariable Long id) {
 demoFeignClient.deleteSomething(id);
 return "Deleted";
}

```
---
```

 Ready for ZIP?

Would you like me to generate a **complete Spring Boot project ZIP file** with:

- `RestTemplateController.java`
- `WebClientController.java`
- `FeignClientController.java`
- DTO: `MyRequest.java`

- Full setup (`pom.xml`, application files)

Let me know and I'll generate it for you!

In summary, RestTemplate is a synchronous client suitable for traditional blocking applications, while WebClient is an asynchronous client suitable for reactive, non-blocking applications.

Certainly! Let's break down the provided code and configuration for a microservices architecture using Spring Cloud Gateway, Eureka Server, and Eureka Clients:

1. Discovery Services Application (`DiscoveryServicesApplication`):

- `@EnableEurekaServer`: Enables this application as a Eureka server for service registration and discovery.
 - `eureka.client.register-with-eureka=false` and `eureka.client.fetch-registry=false`: These properties are set to false because this application itself acts as the Eureka server and does not need to register with another Eureka server or fetch registry information.
 - `spring.application.name=discovery-services`: Specifies the name of this application.

2. Product Service Application (`ProductServiceApplication`):

- `@EnableEurekaClient`: Indicates that this application is a Eureka client and should register itself with the Eureka server.
- **Database and Eureka Configurations**: Configures database connection details and specifies the Eureka server URL for service registration and discovery.

3. API Gateway Application (`ApiGatewayApplication`):

- **`@EnableEurekaClient`**: Indicates that this application is a Eureka client and should register itself with the Eureka server.
- **Gateway Routes Configuration**: Configures routes for forwarding requests to other microservices based on their paths.
- **Eureka Client Configuration**: Specifies the Eureka server URL for service registration and discovery.

Flow of Microservices Architecture:

1. **Service Registration**: Each microservice (e.g., Product Service, Order Service) registers itself with the Eureka-Server (Discovery-Service) upon startup using the `@EnableEurekaClient` annotation.
2. **Service Discovery**: The API Gateway queries the Eureka server to discover the available microservices using the `spring.cloud.gateway.discovery.locator.enabled=true` property.
3. **Routing and Load Balancing**: The API Gateway routes incoming requests to the appropriate microservices based on their paths and uses Spring Cloud LoadBalancer (`lb://`) to load balance requests across multiple instances of each microservice retrieved from Eureka.
4. **Centralized Configuration**: Eureka server acts as a centralized registry for all microservices, allowing them to dynamically discover and communicate with each other without hardcoding service locations or configurations.

Overall, this architecture provides a scalable, resilient, and flexible solution for building microservices-based applications, enabling dynamic service registration, discovery, and routing.

Microservices Flow:

1. ****User Interaction (Angular Frontend):****

- Angular frontend provides the user interface.
- Users interact with the frontend to perform actions and view data.

2. ****API Gateway (E.g. Netflix Zuul or spring-cloud-starter-gateway):****

- Acts as a single entry point for client requests.
- Routes requests to appropriate microservices.
- Performs tasks like load balancing, authentication, and rate limiting.

Note : Load Balancing : Load balancing is like a traffic cop directing cars to different lanes on a highway to avoid congestion and keep traffic flowing smoothly.

It distributes incoming requests across multiple servers, preventing any one server from becoming overloaded and ensuring faster, more reliable access to websites and applications.

3. ****Discovery Service (E.g., Netflix Eureka):****

- Helps microservices locate and communicate with each other.
- Microservices register themselves upon startup.
- Maintains a registry of available microservices and their network locations.

4. ****Backend Microservices (Spring Boot):****

- Individual microservices handle specific business functions.
- Expose RESTful APIs for CRUD operations and other tasks.
- Communicate with each other via HTTP calls.

5. ****RestTemplate and WebClient:****

- Spring Boot's HTTP clients for making HTTP requests to other services.

- RestTemplate: synchronous and blocking, suitable for traditional I/O operations.
- WebClient: asynchronous and non-blocking, suitable for reactive programming.

Sure, here's an example of using both RestTemplate and WebClient in a Spring Boot application:

1. **Using RestTemplate (Synchronous):**

```
```java
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class MyService {

 private final String API_URL = "https://api.example.com/data";

 private final RestTemplate restTemplate;

 public MyService(RestTemplate restTemplate) {
 this.restTemplate = restTemplate;
 }

 public String fetchData() {
 // Synchronous call using RestTemplate
 String response = restTemplate.getForObject(API_URL,
String.class);
 return response;
 }
}
```

```
}
```

```
...
```

## 2. \*\*Using WebClient (Asynchronous):\*\*

```
```java
```

```
import org.springframework.stereotype.Service;  
import org.springframework.web.reactive.function.client.WebClient;  
import reactor.core.publisher.Mono;
```

```
@Service
```

```
public class MyReactiveService {
```

```
    private final String API_URL = "https://api.example.com/data";
```

```
    private final WebClient webClient;
```

```
    public MyReactiveService(WebClient.Builder webClientBuilder) {
```

```
        this.webClient =
```

```
        webClientBuilder.baseUrl(API_URL).build();
```

```
}
```

```
    public Mono<String> fetchData() {
```

```
        // Asynchronous call using WebClient
```

```
        Mono<String> responseMono = webClient.get()
```

```
            .retrieve()
```

```
            .bodyToMono(String.class);
```

```
    return responseMono;
```

```
    }  
}  
...  
  
}
```

With `RestTemplate`, the execution thread is blocked until the response is received, while with `WebClient`, the execution is non-blocking, allowing the thread to be free to handle other tasks while waiting for the response. This makes `WebClient` suitable for applications requiring high concurrency and responsiveness, such as reactive applications.

6. **Authentication and Authorization:**

- Handled at the frontend using mechanisms like JWT.
- Upon successful authentication, frontend includes JWT token in HTTP requests.
- Backend verifies JWT token to ensure user identity and grants access based on roles.

Certainly! Below are examples of how to use each of the mentioned `RestTemplate` methods:

1. `getForObject`

```
```java  
// Example: Fetching a user resource by ID
User user = restTemplate.getForObject("http://api.example.com/users/{id}", User.class,
userId);
...

```
```

2. `getForEntity`

```
```java
// Example: Fetching a user resource by ID and retrieving the entire response entity

 ResponseEntity<User> response =
restTemplate.getForEntity("http://api.example.com/users/{id}", User.class, userId);

User user = response.getBody();

HttpStatus statusCode = response.getStatusCode();

HttpHeaders headers = response.getHeaders();

```

```

3. `postForObject`

```
```java
// Example: Creating a new user by sending user data as request body

User newUser = ...; // Assuming newUser is an instance of User class with user data

User createdUser = restTemplate.postForObject("http://api.example.com/users",
newUser, User.class);

```

```

4. `postForEntity`

```
```java
// Example: Creating a new user and retrieving the entire response entity

User newUser = ...; // Assuming newUser is an instance of User class with user data

 ResponseEntity<User> response =
restTemplate.postForEntity("http://api.example.com/users", newUser, User.class);

User createdUser = response.getBody();

HttpStatus statusCode = response.getStatusCode();
```

```
HttpHeaders headers = response.getHeaders();
```

```
...
```

```
5. `exchange`
```

```
```java
```

```
// Example: Sending a PUT request to update user data and retrieving the entire  
response entity
```

```
User updatedUser = ...; // Assuming updatedUser is an instance of User class with  
updated data
```

```
HttpHeaders headers = new HttpHeaders();
```

```
headers.setContentType(MediaType.APPLICATION_JSON);
```

```
HttpEntity<User> requestEntity = new HttpEntity<>(updatedUser, headers);
```

```
ResponseEntity<User> response =  
restTemplate.exchange("http://api.example.com/users/{id}", HttpMethod.PUT, requestEntity,  
User.class, userId);
```

```
User updatedUserResponse = response.getBody();
```

```
...
```

```
### 6. `delete`
```

```
```java
```

```
// Example: Deleting a user resource by ID
```

```
restTemplate.delete("http://api.example.com/users/{id}", userId);
```

```
...
```

These examples demonstrate how to use various `RestTemplate` methods to interact with RESTful services for common CRUD operations (Create, Read, Update, Delete). Each method provides different levels of flexibility and control over the HTTP request and response handling.

**\*\*Normal Spring Boot Monolithic Application:\*\***

---

In a monolithic Spring Boot application, all functionalities are bundled together in a single application. Here's an example of a simplified monolithic application for a basic e-commerce platform:

**\*\*Microservice Architecture with Spring Boot:\*\***

---

**\*\*Microservices\*\*** is an architectural approach where a software application is divided into a collection of small, independent, and loosely coupled services, each responsible for specific functionality. Spring Boot, a popular Java framework, is often used to build microservices due to its ease of development and the tools it provides.

**\*\*Microservice in Spring Boot:\*\***

A **microservice in Spring Boot** is a standalone, independently deployable application that performs a specific function within a larger software system. Each microservice typically has its own database and communicates with other microservices through well-defined APIs (often using HTTP/REST).

**\*\*Uses of Microservices with Spring Boot:\*\***

- **Scalability:** Microservices can be independently scaled, allowing you to allocate more resources to specific services that need it, while leaving others unaffected.

- **Maintainability:** Smaller codebases are easier to manage and update. Changes in one service don't affect others, making maintenance more manageable.

- **Faster Development:** Smaller teams can develop and deploy microservices independently, accelerating development cycles.

- **Resilience:** Isolating services reduces the impact of failures. If one service fails, it doesn't bring down the entire system.

- **Technology Diversity:** Different microservices can use different technologies and databases to choose the right tools for each task.

#### **Example of a Microservice in Spring Boot:**

Consider an e-commerce platform. You might have separate microservices for user authentication, product catalog, order processing, and payment handling.

=====

BUILDING APPLICATION :

=====

1. We will build simple ONLINE SHOPPING APPLICATION.

2. We will cover below topics:

- i. Service Discovery
- ii. Centralized Configuration.
- iii. Distributed Tracing.
- iv. Event Driven Architecture.
- v. Centralized Logging.
- vi. Circuit Breaker.
- vii. Secure Microservice Using Keycloak.

Services We are going to build :

---

1. Product Service : Create and View Products, acts as Product Catalog.

2. Order Service : Can order products.

3. Inventory Service : Can check if product is in stock or not.

4. Notification Service : Can send notifications, after order is placed.

5. Order Service , Inventory Service and Notification Service are going to interact with each other.

6. Synchronous and Asynchronous communication.

1. After creating product-service, order-service and inventory-service, We use WebClient to make Synchronous request from Order-Service to the Inventory-Service, then Inventory-Service will respond with required data and WebClient will take that required data and give it to Order-Service.

2. Discovery-Service is nothing but a server which will store all the information about services(Inventory-Service) like service-name, IP address.

3. When we are using the Discovery-Service , Our microservices will register in the Discovery-Service by making the request at the starting of the application.

4. Whenever services(Inventory-Service) are making request , Discovery-Service will add the entries of the services into it's local copy. we call it registry, that's why we call it Service Registry.

5. Once all the information about the services(Inventory-Service) is present in Discovery-Service, When our Order-Service wants to call the Inventory-Service, First Order-Service will call Discovery-Service by asking where I can find the Inventory-Service. Then Discovery-Service will respond with particular IP address to call Inventory-Service. Then Order-Service will call to Inventory-Service.

6. In this way, we can avoid the hardcoding URL of the Inventory-Service by making use of Discovery-Service.

7. When we are making initial call to the Discovery-Service, Discovery-Service will sends its registry as the response to the client , the client will store the local copy of the registry in a separate location.

8. In Some reasons, If the Discovery-Service is not available, first Discovery-Service will check the local copy because it already has information about Inventory-Service. It will call Inventory-Service IP address, If the first instance of the Inventory-Service is not available, it will check the next entry of Inventory-Service likewise it goes through all entries of its registry. If all the instances are down, Discovery-Service will fail by saying that Inventory-Service is not available.

#### ### Spring Boot Security:

=====

##### \*Explanation:\*

Spring Boot Security is a powerful authentication and access control framework for Java applications.

It's built on top of the Spring Security framework and simplifies the configuration and integration of security features into Spring Boot applications.

##### \*Uses:\*

- \*User Authentication and Authorization:\* Secure your application by authenticating users and controlling access to different resources.
- \*Form-based Login:\* Easily set up login pages with minimal configuration.
- \*Method-Level Security:\* Restrict access to specific methods or services based on user roles.

##### \*Advantages:\*

- **\*Ease of Integration:\*** Seamless integration with Spring Boot applications.
- **\*Customization:\*** Highly customizable to suit various authentication and authorization requirements.
- **\*Community Support:\*** Extensive community support and documentation.

**\*Disadvantages:\***

- **\*Learning Curve:\*** It might have a learning curve for beginners, especially when dealing with complex configurations.
- **\*Overhead:\*** In some cases, the comprehensive features might introduce more complexity than needed for simple applications.

**\*Example Code:\***

Here's a basic example of a Spring Boot Security configuration class:

```
java
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
 @Override
 protected void configure(HttpSecurity http) throws Exception {
 http
 .authorizeRequests()
 .antMatchers("/public/**").permitAll()
 .anyRequest().authenticated()
 .and()
 .formLogin();
 }
}
```

different types of security authentication on springboot security?

---

Spring Security supports various types of authentication mechanisms to secure applications. Here are some of the common types of authentication supported by Spring Security:

1. **Form-Based Authentication:**

- **Description:** Users enter their credentials (username and password) in a login form.
- **Usage:** Suitable for web applications where users interact through a web-based user interface.

2. **HTTP Basic Authentication:**

- **Description:** Users provide their credentials in the form of a username and password in the HTTP headers.
- **Usage:** Often used for simple authentication in web services where the communication is stateless.

3. **HTTP Digest Authentication:**

- **Description:** Similar to HTTP Basic Authentication, but the password is hashed before being sent.
- **Usage:** Offers a more secure alternative to HTTP Basic Authentication, especially in scenarios where credentials might be intercepted.

#### 4. \*\*JWT (JSON Web Token) Authentication:\*\*

- \*\*Description:\*\* Authentication is performed using tokens, and the server does not need to keep track of the user's state.
- \*\*Usage:\*\* Suitable for stateless architectures, microservices, and modern web applications.

#### 5. \*\*OAuth 2.0 Authentication:\*\*

- \*\*Description:\*\* Allows third-party applications to obtain limited access to an HTTP service.
- \*\*Usage:\*\* Commonly used for authorization and delegation scenarios in web and mobile applications.

#### 6. \*\*SAML (Security Assertion Markup Language) Authentication:\*\*

- \*\*Description:\*\* Supports single sign-on (SSO) using XML-based tokens (SAML assertions).
- \*\*Usage:\*\* Ideal for scenarios where users need to authenticate once and access multiple services without re-authenticating.

#### 7. \*\*LDAP (Lightweight Directory Access Protocol) Authentication:\*\*

- \*\*Description:\*\* Authenticates users against an LDAP directory service.
- \*\*Usage:\*\* Commonly used in enterprise environments where user information is stored in LDAP directories.

#### 8. \*\*JDBC Authentication:\*\*

- \*\*Description:\*\* Authenticates users against a database using JDBC (Java Database Connectivity).
- \*\*Usage:\*\* Useful when user credentials are stored in a relational database.

## 9. \*\*Custom Authentication Providers:\*\*

- \*\*Description:\*\* Allows developers to implement custom authentication logic by extending `AuthenticationProvider`.

- \*\*Usage:\*\* When none of the built-in authentication mechanisms meets specific requirements, developers can create their own authentication logic.

## 10. \*\*Remember-Me Authentication:\*\*

- \*\*Description:\*\* Provides a "remember me" functionality allowing users to be remembered across sessions.

- \*\*Usage:\*\* Useful for improving user experience by keeping users authenticated between sessions.

These authentication mechanisms can be configured and customized based on the specific security requirements of your Spring Boot application. The choice of authentication depends on factors such as the application's architecture, the type of users, and the desired level of security.

## PLEASE OPEN THE LINK FOR SPRINGBOOT SECURITY TUTORIAL

---

[https://www.tutorialspoint.com/spring\\_security/index.htm](https://www.tutorialspoint.com/spring_security/index.htm)

## Stateful and Stateless Applications using SPRING SECURITY

---

Stateful and stateless applications refer to how they manage user session data and user state.

### 1. \*\*Stateful Application\*\*:

- In a stateful application, the server keeps track of the state of the client session.
- It means that the server maintains session information, such as user authentication details, in its memory or a database across multiple requests from the same client.
- This usually requires more server resources and can make scaling more complex. However, it can provide more flexibility and control over the user session.

### 2. \*\*Stateless Application\*\*:

- In a stateless application, the server does not store any client session state.
- Each request from a client must contain all the information necessary to understand and fulfill that request.
- Typically, this information is included in the request headers.
- Stateless applications are generally easier to scale and can be more resilient, but they may require more data to be transmitted with each request.

Here's a basic example of implementing JWT authentication in a Spring Boot application, demonstrating both stateful and stateless approaches:

#### ### Stateful Approach:

In the stateful approach, you'll use session management to keep track of user authentication.

```
```java
```

```
@RestController
```

```
public class AuthController {
```

```

    @PostMapping("/login")
    public ResponseEntity<String> login(@RequestParam String username,
    @RequestParam String password, HttpSession session) {
        // Authenticate user (omitted for simplicity)
        session.setAttribute("username", username);
        String token = generateJWT(username);
        return ResponseEntity.ok(token);
    }

    @GetMapping("/secureData")
    public ResponseEntity<String> getSecureData(HttpSession session) {
        String username = (String) session.getAttribute("username");
        // Retrieve secure data based on the username
        return ResponseEntity.ok("Welcome, " + username + "!");
    }
}

...

```

Stateless Approach:

In the stateless approach, you'll use JWT for authentication. No session management is needed on the server side.

```

```java
@RestController
public class AuthController {

```

```

 @Autowired
 private JwtUtil jwtUtil;

 @PostMapping("/login")
 public ResponseEntity<String> login(@RequestParam String username,
 @RequestParam String password) {
 // Authenticate user (omitted for simplicity)
 String token = jwtUtil.generateToken(username);
 return ResponseEntity.ok(token);
 }

 @GetMapping("/secureData")
 public ResponseEntity<String> getSecureData(@RequestHeader("Authorization")
String token) {
 String username = jwtUtil.extractUsername(token);
 // Retrieve secure data based on the username
 return ResponseEntity.ok("Welcome, " + username + "!");
 }
}

```

```

In the stateless approach, `JwtUtil` is a utility class that handles JWT generation and validation.

```

```java
@Component
public class JwtUtil {

```

```
 @Value("${jwt.secret}")

 private String secret;

 public String generateToken(String username) {
 return Jwts.builder()
 .setSubject(username)
 .setIssuedAt(new Date())
 .setExpiration(new Date(System.currentTimeMillis() + 1000
* 60 * 60)) // 1 hour
 .signWith(SignatureAlgorithm.HS256, secret)
 .compact();
 }

 public String extractUsername(String token) {
 return
Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody().getSubject();
 }
 ...
}
```

In this example, the `JwtUtil` class provides methods for generating and parsing JWTs. The `@Value` annotation is used to inject the secret key from configuration. Ensure that you set a strong secret key and manage it securely.

### ### OAuth 2.0: SPRING SECURITY

---

#### \*Explanation:\*

OAuth 2.0 is an open standard for authorization that enables secure access to resources on behalf of a user without exposing their credentials.

It's commonly used for third-party authentication and authorization.

#### \*Uses:\*

- \*Third-Party Authentication:\* Allows users to log in using credentials from another platform (e.g., Google, Facebook).
- \*Secure API Access:\* Enables secure access to APIs on behalf of a user without exposing their credentials.
- \*Single Sign-On (SSO):\* Users can authenticate once and access multiple applications.

#### \*Advantages:\*

- \*Secure Authorization:\* Enhances security by not exposing user credentials to third-party applications.
- \*Single Sign-On:\* Simplifies user experience by allowing a single login for multiple services.
- \*Standardization:\* A widely adopted standard with broad industry support.

#### \*Disadvantages:\*

- \*Complexity:\* Implementing OAuth 2.0 can be complex, especially for beginners.
- \*Token Management:\* Requires careful handling of access tokens to prevent security issues.

\*Example Code:\*

For a Spring Boot OAuth 2.0 example, consider using Spring Security's OAuth 2.0 client. Here's a simplified example:

properties

```
application.properties

spring.security.oauth2.client.registration.google.client-id=<your-client-id>
spring.security.oauth2.client.registration.google.client-secret=<your-client-secret>
```

java

```
@SpringBootApplication
@EnableOAuth2Client
public class MyApplication {

 public static void main(String[] args) {
 SpringApplication.run(MyApplication.class, args);
 }
}
```

This example configures Google as an OAuth 2.0 provider for authentication.

These are simplified examples, and real-world implementations may involve additional configurations and considerations.

Always refer to official documentation for comprehensive details and best practices.

Certainly! I can guide you through a brief tutorial on Spring Boot Security and OAuth 2.0. Keep in mind that this is a high-level overview, and you may want to refer to official documentation or more detailed tutorials for a comprehensive understanding.

#### #### Spring Boot Security Tutorial:

##### 1. \*Dependencies:\*

- Add the following dependencies to your pom.xml or build.gradle:

xml

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

##### 2. \*Security Configuration:\*

- Create a class extending WebSecurityConfigurerAdapter to customize security configurations.

java

```
@EnableWebSecurity
```

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {
```

```
 @Override
```

```
 protected void configure(HttpSecurity http) throws Exception {
```

```
 http.authorizeRequests().antMatchers("/public/**").permitAll()
```

```
 .anyRequest().authenticated().and().formLogin();
```

```
 }
}

}
```

### 3. \*User Authentication:\*

- Configure in-memory authentication or use a custom user service:

```
java
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
 auth.inMemoryAuthentication()
 .withUser("user").password("{noop}password").roles("USER");
}
```

### #### OAuth 2.0 Tutorial:

---

#### 1. \*Add Dependencies:\*

- Include the following dependencies:

```
xml
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

#### 2. \*Configure OAuth 2.0:\*

- In application.properties or application.yml, specify OAuth 2.0 settings:

```
properties
spring.security.oauth2.client.registration.google.client-id=<your-client-id>
spring.security.oauth2.client.registration.google.client-secret=<your-client-
secret>
```

### 3. \*Secure Endpoints:\*

- Use `@EnableOAuth2Client` on your `@SpringBootApplication` class.
- Secure specific endpoints using `@PreAuthorize` or configure in `SecurityConfig`.

### 4. \*Customize Login Page:\*

- Customize the login page by providing your own HTML template in `resources/templates/login.html`.

Remember to replace placeholder values with your actual configurations. This is a simplified guide, and you should refer to the official Spring Boot and OAuth 2.0 documentation for detailed explanations and best practices.

## OAuth 2.0

---

OAuth 2.0, which stands for "Open Authorization", is a standard designed to allow a website or application to access resources hosted by other web apps on behalf of a user.

\* OAuth 2.0 is an authorization protocol and NOT an authentication protocol.

\* It is designed primarily as a means of granting access to a set of resources, for example, remote APIs or user's data.

\* OAuth 2.0 uses Access Tokens.

## OAuth 2.0 Roles

---

\* Resource Owner

\* Client

\* Authorization Server

\* Resource Server

Resource Owner : The user or system that owns the protected resources and can grant access to them.

Client : The client is the system that requires access to the protected resources.

To access resources, the Client must hold the appropriate Access Token.

Authorization Server : This server receives requests from the Client for Access Token and issues them upon successful authentication and consent by the Resource Owner.

Resource Server : A server that protects the user's resource and receives access requests from the Client.

It accepts and validates an Access Token from the Client and returns the appropriate resources to it.

## OAuth 2.0 Scopes

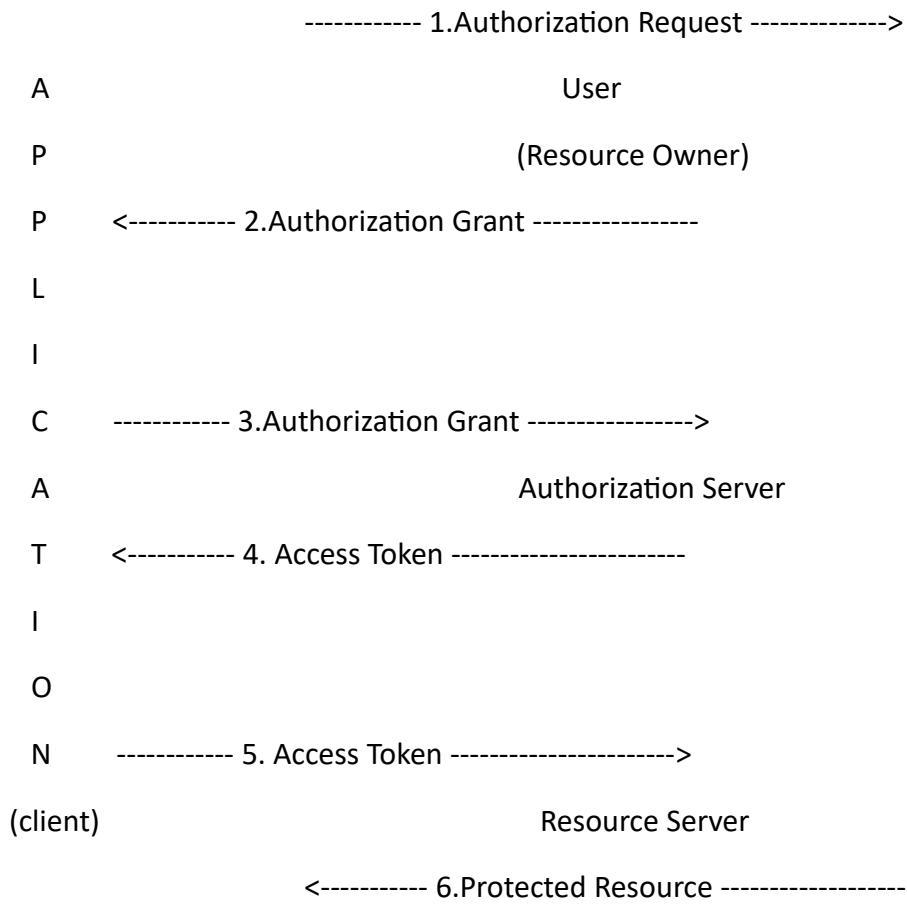
=====

Scopes are an important concept in OAuth 2.0.

They are used to specify exactly the reason for which access to resource may be granted.

## Abstract Protocol Flow

=====



## SERVICE API

### JWT Spring Boot Security

---

JWT (JSON Web Token) is a widely used standard for securely transmitting information between parties as a JSON object. In the context of Spring Boot security, JWT is often used for authentication and authorization purposes.

In Spring Boot security, JWT can be implemented using the Spring Security framework along with additional libraries such as jjwt. Here's a brief explanation of how JWT works in Spring Boot security:

1. Authentication: When a user logs in or provides their credentials, the server verifies the credentials and generates a JWT token. This token contains encoded information about the user, such as their username and roles.
2. Token Generation: The server signs the JWT token using a secret key known only to the server. This ensures that the token cannot be tampered with or modified by unauthorized parties.

3. Token Transmission: The server sends the JWT token back to the client, typically as a response to a successful login request. The client then stores this token, usually in local storage or a cookie.

4. Authorization: For subsequent requests, the client includes the JWT token in the request headers. The server validates the token's signature and extracts the user information from it. Based on the extracted information, the server can then authorize or deny access to protected resources.

5. Token Expiration: JWT tokens can have an expiration time, which is typically included in the token payload. Once a token expires, the client needs to obtain a new token by re-authenticating with the server.

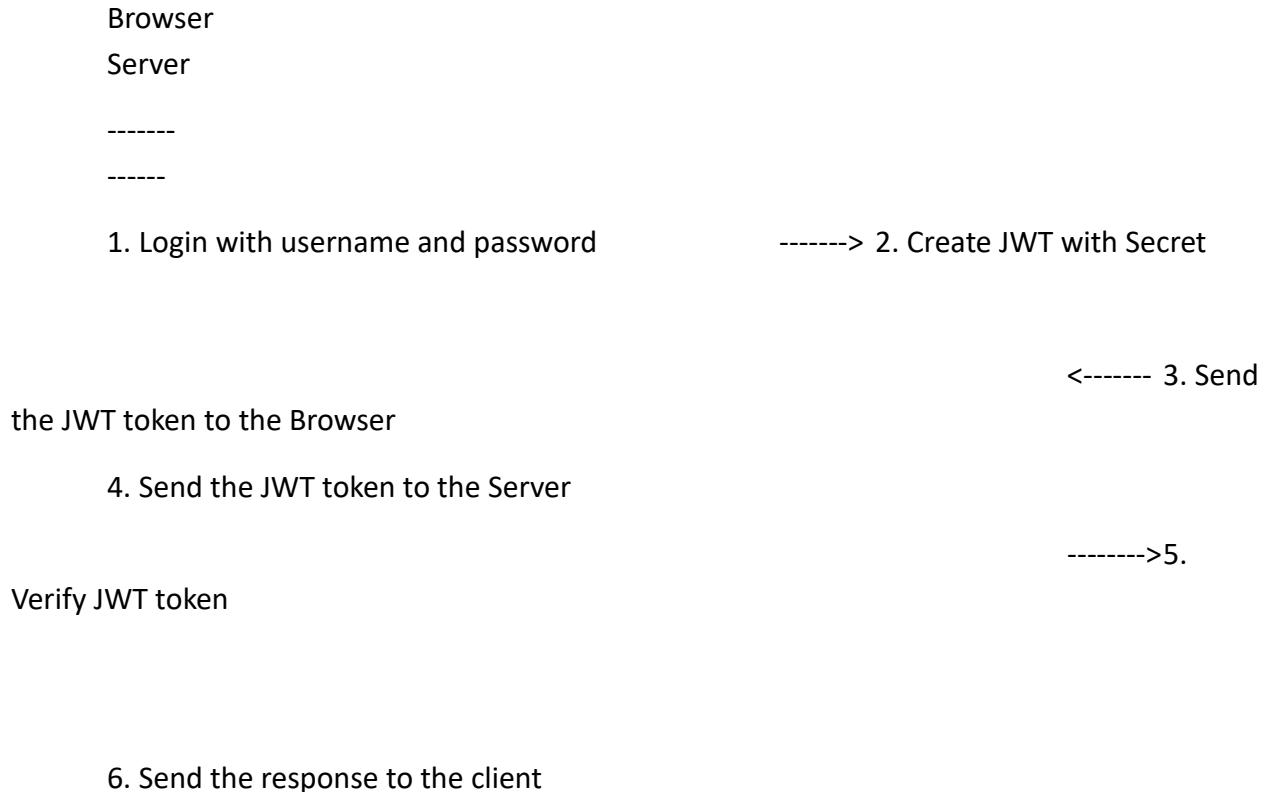
By using JWT in Spring Boot security, developers can implement stateless authentication and authorization mechanisms, as the server does not need to store session information. This makes JWT a scalable and efficient solution for securing web applications.

It's important to note that while JWT provides a secure way to transmit information, it does not encrypt the data within the token. Therefore, sensitive information should not be included in the token payload to maintain security.

<https://www.geeksforgeeks.org/spring-boot-3-0-jwt-authentication-with-spring-security-using-mysql-database/>

JWT Authentication - Spring Security

=====



Sure, here's an explanation of the JWT (JSON Web Token) authentication process in your application:

#### 1. \*\*Client Authentication Request\*\*:

- When a client (such as a web browser or mobile app) sends a request to your server to authenticate a user, it typically includes the user's credentials (e.g., username and password) in the request body.

#### 2. \*\*JWT Token Creation\*\*:

- Upon receiving the authentication request, your server's `JwtResource` controller handles the request.

- The `createJwtToken` method in the `JwtService` class is invoked. This method generates a JWT token based on the provided user credentials.

- The `createJwtToken` method utilizes Spring Security's `AuthenticationManager` to authenticate the user's credentials.

### 3. \*\*User Authentication\*\*:

- The `JwtService` class uses Spring Security's `AuthenticationManager` to authenticate the user's credentials.
- The `authenticate` method in `JwtService` performs the authentication process using Spring Security's authentication mechanisms.
- If the provided credentials are valid, authentication succeeds, and the user is considered authenticated.

### 4. \*\*JWT Token Generation\*\*:

- After successful authentication, the `createJwtToken` method in the `JwtService` class generates a JWT token.
- The JWT token includes claims such as the user's identity (username), issuance time, expiration time, and possibly other custom claims.

### 5. \*\*Token Issuance\*\*:

- The generated JWT token is returned to the client as part of the response from the `createJwtToken` method in the `JwtResource` controller.
- The client receives the JWT token and typically stores it securely (e.g., in local storage or a cookie) for future authenticated requests.

### 6. \*\*Subsequent Requests with JWT\*\*:

- For subsequent requests that require authentication, the client includes the JWT token in the request headers, typically in the `Authorization` header with the value `Bearer <token>`.
- The client sends the JWT token along with the request to access protected resources on the server.

## 7. \*\*JWT Token Validation\*\*:

- Upon receiving a request with a JWT token, the server's `JwtRequestFilter` intercepts the request.
- The filter extracts the JWT token from the request headers and validates its integrity and expiration using the `JwtUtil` class.
- If the token is valid, the filter allows the request to proceed; otherwise, it may reject the request with an unauthorized response.

## 8. \*\*Access to Protected Resources\*\*:

- If the JWT token is valid and the user is authenticated, the server grants access to the requested resources.
- Spring Security's authorization mechanisms may further control access based on user roles and permissions.

## 9. \*\*Token Expiry and Renewal\*\*:

- When a JWT token expires, the client needs to obtain a new token by re-authenticating with the server.
- The client can initiate the re-authentication process by sending another authentication request to obtain a fresh JWT token.

In summary, the JWT authentication process involves generating a token upon successful user authentication, issuing the token to the client, validating the token on subsequent requests, and granting access to protected resources if the token is valid and the user is authenticated.

## DATA STRUCTURES

---

$\Rightarrow O(1)$  -> Order One or Constant Time.

$\Rightarrow O(n)$  -> Order n or Linear Time.

O notation, also known as Big O notation, is a mathematical notation used in computer science to describe the behavior of algorithms and functions in terms of their time complexity or space complexity as the input size grows. It provides an upper bound on the time or space required by an algorithm or function.

The notation is expressed using the letter "O" followed by a function. This function represents the upper bound of the algorithm's time complexity or space complexity, typically in terms of the input size, denoted as "n."

Here are a few common examples of O notation:

-  $O(1)$  - Constant time complexity: The time or space required by the algorithm remains constant regardless of the input size.

-  $O(\log n)$  - Logarithmic time complexity: The time or space required by the algorithm grows logarithmically as the input size grows.

-  $O(n)$  - Linear time complexity: The time or space required by the algorithm grows linearly with the input size.

-  $O(n^2)$  - Quadratic time complexity: The time or space required by the algorithm grows quadratically with the input size.

-  $O(2^n)$  - Exponential time complexity: The time or space required by the algorithm grows exponentially with the input size.

These notations help in analyzing and comparing the efficiency of algorithms and understanding how they scale with different input sizes. By knowing the time or space complexity of an algorithm, developers can make informed decisions about algorithm selection and optimization.

Time complexity and space complexity are measures used to analyze algorithms based on their performance characteristics.

1. **Time Complexity**: It is a measure of the amount of time an algorithm takes to run as a function of the length of the input. It gives an upper bound on the running time in terms of the input size.
2. **Space Complexity**: It measures the amount of memory space an algorithm consumes as a function of the input size. It includes both the space used by the algorithm itself and any auxiliary data structures it uses.

Certainly!  $O(1)$  and  $O(1)$  refer to constant time complexity, but they apply to different operations.

1.  **$O(1)$  Time Complexity**:
  - $O(1)$  stands for "order one" or "constant time."
  - It means that the time taken by an algorithm or operation remains constant, regardless of the size of the input data.
  - Regardless of the size of the data structure, the operation completes in the same amount of time.
  - Examples:
    - Accessing an element in an array by index (assuming random access): Regardless of the size of the array, accessing any element takes the same amount of time because arrays provide constant-time access based on the index.
    - Adding or removing an element at the beginning or end of a `LinkedList` or `ArrayList` when there's no resizing or shifting involved: In `LinkedLists`, adding or removing elements at the beginning or end typically involves updating references, which takes constant

time. In ArrayLists, adding elements at the end when there's available capacity can also be a constant-time operation.

## 2. **\*\*O(1) Space Complexity\*\*:**

- O(1) space complexity means that the amount of additional memory required by an algorithm or operation remains constant, regardless of the size of the input data.

- It implies that the algorithm or operation uses a constant amount of memory that doesn't grow with the size of the input.

### - Examples:

- Storing a fixed number of variables: Regardless of the size of the input data, if the algorithm only requires a fixed number of variables or a fixed-size data structure, its space complexity is O(1).

- Storing a single pointer or reference: If an algorithm only needs to store a single pointer or reference, the space complexity is O(1) because the memory required for the pointer remains constant, regardless of the size of the input.

In summary, O(1) time complexity means that an operation executes in constant time, while O(1) space complexity means that the amount of additional memory used remains constant. These complexities are desirable because they indicate that an algorithm or operation is efficient and does not degrade in performance as the input size grows.

Certainly! O(n) refers to linear time complexity. Let's delve into it:

## **\*\*O(n) Time Complexity\*\*:**

- O(n) stands for "order n" or "linear time."

- It means that the time taken by an algorithm or operation grows linearly with the size of the input data.

- As the size of the input increases, the time taken to complete the operation increases proportionally.

### - Examples:

- Linear search: When searching for an element in an unsorted array sequentially, the time taken is  $O(n)$ , as the algorithm may need to traverse the entire array to find the element.

- Traversing a LinkedList: If you need to iterate over all elements in a LinkedList, the time taken would be  $O(n)$  because you need to visit each node in the list once.

- Summing the elements of an array: If you sum all elements in an array sequentially, the time taken would be  $O(n)$  because you need to visit each element once to calculate the sum.

#### \*\* $O(n)$ Space Complexity\*\*:

- $O(n)$  space complexity means that the amount of additional memory required by an algorithm or operation grows linearly with the size of the input data.

- As the size of the input increases, the amount of memory used also increases proportionally.

- Examples:

- Creating a new array to store the result of an operation that depends on the size of the input array: If you need to create a new array to store the result of doubling each element in the input array, the space complexity would be  $O(n)$ , as the size of the new array depends on the size of the input array.

- Storing the elements of a LinkedList: If you store  $n$  elements in a LinkedList, the space complexity would be  $O(n)$  because you need to allocate memory for each node in the list.

In summary,  $O(n)$  time complexity indicates that the time taken by an algorithm grows linearly with the size of the input, while  $O(n)$  space complexity indicates that the amount of additional memory used also grows linearly with the size of the input.

Let's demonstrate these concepts with a simple example of finding the maximum element in an array:

```
```java
```

```
public class MaxElement {

    // Function to find the maximum element in an array
    public static int findMax(int[] arr) {
        int max = arr[0]; // Initialize max with the first element

        // Traverse the array to find the maximum element
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] > max) {
                max = arr[i];
            }
        }

        return max;
    }

    public static void main(String[] args) {
        int[] array = {3, 7, 2, 8, 5}; // Sample input array
        int maxElement = findMax(array);
        System.out.println("Maximum element: " + maxElement);
    }
}
```

In this program, the `findMax` function traverses the input array once to find the maximum element. Let's analyze its time and space complexity:

- **Time Complexity**: The function has a time complexity of $O(n)$, where n is the size of the input array. This is because it needs to traverse the entire array once, regardless of the input values.

- **Space Complexity**: The function has a space complexity of $O(1)$, which means it uses a constant amount of memory regardless of the size of the input array. This is because it only needs a few variables (`max`, `i`) to perform the operation, and the amount of memory they occupy does not depend on the size of the input array.

Understanding time and space complexity helps in evaluating the efficiency of algorithms and choosing the most appropriate one for a given problem.

IN SIMPLE TERMS

Time Complexity: Think of it as how long an algorithm takes to run as the size of the input grows. It tells you how many operations an algorithm needs relative to the size of its input. For example, if you have a list of numbers, how many comparisons or steps does it take to find the biggest number?

Space Complexity: This measures how much memory an algorithm needs as the size of the input grows. It includes both the memory used by the algorithm itself and any additional data structures it requires. For instance, if you're sorting a list of numbers, how much additional memory do you need to store intermediate values or pointers?

In simpler terms, time complexity is about speed - how fast does the algorithm run as the input gets bigger, while space complexity is about memory - how much extra memory does the algorithm use as the input gets bigger.

In Data Structures and Algorithms (DSA), various types of sorting algorithms, search algorithms, and data structures like stacks, queues, and lists are commonly used. Here's a brief overview of each:

Sorting Algorithms:

1. **Bubble Sort:**

- Compares adjacent elements and swaps them if they are in the wrong order.
- Repeats this process until the entire list is sorted.

2. **Selection Sort:**

- Divides the input list into two parts: a sorted and an unsorted region.
- In each iteration, finds the smallest element in the unsorted region and swaps it with the first element of the unsorted region.

3. **Insertion Sort:**

- Builds the final sorted array one item at a time.
- Takes each element from the unsorted list and inserts it into its correct position in the sorted list.

4. **Merge Sort:**

- Divides the array into two halves, recursively sorts them, and then merges them.
- Uses a divide-and-conquer strategy.

5. **Quick Sort:**

- Selects a pivot element and partitions the array into two sub-arrays: elements less than the pivot and elements greater than the pivot.

- Recursively applies the same process to the sub-arrays.

6. **Heap Sort:**

- Builds a binary heap and repeatedly extracts the maximum element, placing it in the sorted portion of the array.

- Utilizes a binary heap data structure.

Searching Algorithms:

1. **Linear Search:**

- Checks each element in the list until a match is found or the end of the list is reached.

2. **Binary Search:**

- Applies to a sorted list and repeatedly divides the search interval in half.

- Compares the target value to the middle element and narrows down the search.

Data Structures:

1. **Arrays:**

- A collection of elements, each identified by an index or a key.

2. **Linked List:**

- A linear data structure where elements are stored in nodes, and each node points to the next node in the sequence.

3. **Stack:**

- Follows the Last In, First Out (LIFO) principle.
- Operations include push (add an item) and pop (remove the last-added item).

4. **Queue:**

- Follows the First In, First Out (FIFO) principle.
- Operations include enqueue (add an item) and dequeue (remove the front item).

5. **Hash Table:**

- Uses a hash function to map keys to indices in an array.
- Provides constant-time average complexity for basic operations like insert and search.

6. **Binary Tree:**

- A hierarchical tree structure where each node has at most two children.
- Useful for efficient searching, insertion, and deletion.

7. **Graph:**

- A collection of nodes (vertices) and edges that connect pairs of nodes.
- Used to represent relationships between entities.

8. **Heap:**

- A specialized tree-based data structure.
- Used to efficiently find the maximum or minimum element in a collection.

These are fundamental concepts in DSA, and there are variations and advanced versions of these algorithms and data structures. The choice of algorithm or data structure depends on the specific requirements of a problem and the characteristics of the data being processed.

Certainly! Here are the Java programs for each of the sorting algorithms you mentioned:

1. Bubble Sort:

```
```java
public class BubbleSort {

 public static void main(String[] args) {
 int[] arr = {64, 34, 25, 12, 22, 11, 90};

 int n = arr.length;

 for (int i = 0; i < n-1; i++) {
 for (int j = 0; j < n-i-1; j++) {
 if (arr[j] > arr[j+1]) {
 // swap temp and arr[i]
 int temp = arr[j];
 arr[j] = arr[j+1];
 arr[j+1] = temp;
 }
 }

 System.out.println("Sorted array using Bubble Sort: " +
 Arrays.toString(arr));
 }
 }
}
```

```

2. Selection Sort:

```java

```
public class SelectionSort {
 public static void main(String[] args) {
 int[] arr = {64, 34, 25, 12, 22, 11, 90};
 int n = arr.length;

 for (int i = 0; i < n-1; i++) {
 int minIndex = i;
 for (int j = i+1; j < n; j++)
 if (arr[j] < arr[minIndex])
 minIndex = j;

 // Swap the found minimum element with the first element
 int temp = arr[minIndex];
 arr[minIndex] = arr[i];
 arr[i] = temp;
 }

 System.out.println("Sorted array using Selection Sort: " +
 Arrays.toString(arr));
 }
}
```

### ### 3. Insertion Sort:

```java

```
public class InsertionSort {
```

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

```
int[] arr = {64, 34, 25, 12, 22, 11, 90};
```

```
int n = arr.length;
```

```
for (int i = 1; i < n; i++) {
```

```
int key = arr[i];
```

```
int j = i - 1;
```

```
// Move elements of arr[0..i-1] that are greater than key to one position ahead of their current position
```

```
while (j >= 0 && arr[j] > key) {
```

`arr[j + 1] = arr[i];`

j = j - 1;

}

```
arr[j + 1] = key;
```

}

```
System.out.println("Sorted array using Insertion Sort: " +  
Arrays.toString(arr));
```

}

}

三

```
#### 4. Merge Sort:

```java

import java.util.Arrays;

public class MergeSort {

 public static void main(String[] args) {

 int[] arr = {64, 34, 25, 12, 22, 11, 90};

 mergeSort(arr, 0, arr.length - 1);

 System.out.println("Sorted array using Merge Sort: " +
 Arrays.toString(arr));

 }

 private static void mergeSort(int[] arr, int left, int right) {

 if (left < right) {

 int middle = left + (right - left) / 2;

 // Sort first and second halves
 mergeSort(arr, left, middle);
 mergeSort(arr, middle + 1, right);

 // Merge the sorted halves
 merge(arr, left, middle, right);

 }

 }

}
```

```
private static void merge(int[] arr, int left, int middle, int right) {
 int n1 = middle - left + 1;
 int n2 = right - middle;

 int[] L = new int[n1];
 int[] R = new int[n2];

 // Copy data to temporary arrays L[] and R[]
 for (int i = 0; i < n1; ++i)
 L[i] = arr[left + i];
 for (int j = 0; j < n2; ++j)
 R[j] = arr[middle + 1 + j];

 // Merge the temporary arrays
 int i = 0, j = 0;
 int k = left;
 while (i < n1 && j < n2) {
 if (L[i] <= R[j]) {
 arr[k] = L[i];
 i++;
 } else {
 arr[k] = R[j];
 j++;
 }
 k++;
 }
}
```

```

// Copy remaining elements of L[] if any
while (i < n1) {
 arr[k] = L[i];
 i++;
 k++;
}

// Copy remaining elements of R[] if any
while (j < n2) {
 arr[k] = R[j];
 j++;
 k++;
}

}
```

```

5. Quick Sort:

```

```java
import java.util.Arrays;

public class QuickSort {
 public static void main(String[] args) {
 int[] arr = {64, 34, 25, 12, 22, 11, 90};
 quickSort(arr, 0, arr.length - 1);
 }
}
```

```
 System.out.println("Sorted array using Quick Sort: " +
Arrays.toString(arr));
 }

private static void quickSort(int[] arr, int low, int high) {
 if (low < high) {
 int partitionIndex = partition(arr, low
 , high);

 // Recursively sort elements before and after partition
 quickSort(arr, low, partitionIndex - 1);
 quickSort(arr, partitionIndex + 1, high);
 }
}

private static int partition(int[] arr, int low, int high) {
 int pivot = arr[high];
 int i = (low - 1);

 for (int j = low; j < high; j++) {
 if (arr[j] < pivot) {
 i++;
 // Swap arr[i] and arr[j]
 }
 }
}
```

```

 int temp = arr[i];
 arr[i] = arr[j];
 arr[j] = temp;
 }

}

// Swap arr[i+1] and arr[high] (or pivot)
int temp = arr[i + 1];
arr[i + 1] = arr[high];
arr[high] = temp;

return i + 1;
}
```

```

6. Heap Sort:

```

```java
import java.util.Arrays;

public class HeapSort {

 public static void main(String[] args) {
 int[] arr = {64, 34, 25, 12, 22, 11, 90};
 heapSort(arr);

 System.out.println("Sorted array using Heap Sort: " + Arrays.toString(arr));
 }
}
```

```

```
}
```

```
private static void heapSort(int[] arr) {  
    int n = arr.length;  
  
    // Build max heap  
    for (int i = n / 2 - 1; i >= 0; i--)  
        heapify(arr, n, i);  
  
    // One by one extract an element from the heap  
    for (int i = n - 1; i >= 0; i--) {  
        // Move the current root to the end  
        int temp = arr[0];  
        arr[0] = arr[i];  
        arr[i] = temp;  
  
        // Call max heapify on the reduced heap  
        heapify(arr, i, 0);  
    }  
}
```

```
private static void heapify(int[] arr, int n, int i) {  
    int largest = i;  
    int left = 2 * i + 1;  
    int right = 2 * i + 2;
```

```
if (left < n && arr[left] > arr[largest])  
    largest = left;  
  
if (right < n && arr[right] > arr[largest])  
    largest = right;  
  
if (largest != i) {  
    int swap = arr[i];  
    arr[i] = arr[largest];  
    arr[largest] = swap;  
  
    heapify(arr, n, largest);  
}  
}  
}  
...  
  
These are basic implementations, and depending on your use case, you may want to  
handle edge cases, optimize, or adapt these algorithms.
```

Certainly! Here are Java programs for Linear Search and Binary Search:

1. Linear Search:

```
```java
public class LinearSearch {

 public static void main(String[] args) {
 int[] arr = {4, 2, 7, 1, 9, 5, 8};
 int target = 5;

 int result = linearSearch(arr, target);

 if (result != -1) {
 System.out.println("Element " + target + " found at index " +
result);
 } else {
 System.out.println("Element " + target + " not found in the array");
 }
 }

 private static int linearSearch(int[] arr, int target) {
 for (int i = 0; i < arr.length; i++) {
 if (arr[i] == target) {
 return i; // Element found, return index
 }
 }
 }
}
```

```
 }

 return -1; // Element not found

}

```
``
```

2. Binary Search:

```
```java

import java.util.Arrays;

public class BinarySearch {

 public static void main(String[] args) {

 int[] arr = {1, 2, 4, 5, 7, 8, 9};

 int target = 5;

 int result = binarySearch(arr, target);

 if (result != -1) {

 System.out.println("Element " + target + " found at index " +
result);

 } else {

 System.out.println("Element " + target + " not found in the array");

 }

 }

 private static int binarySearch(int[] arr, int target) {
```

```
int left = 0, right = arr.length - 1;

while (left <= right) {
 int mid = left + (right - left) / 2;

 if (arr[mid] == target) {
 return mid; // Element found, return index
 } else if (arr[mid] < target) {
 left = mid + 1;
 } else {
 right = mid - 1;
 }
}

return -1; // Element not found
}
```

These programs demonstrate the basic implementations of Linear Search and Binary Search in Java. Note that Binary Search assumes the array is sorted.

Certainly! Here are Java programs for each of the mentioned data structures:

### 1. Arrays:

```
```java
public class ArrayExample {

    public static void main(String[] args) {
        // Declaration and initialization of an array
        int[] numbers = {1, 2, 3, 4, 5};

        // Accessing elements by index
        System.out.println("Element at index 2: " + numbers[2]);

        // Modifying an element
        numbers[3] = 8;

        // Iterating through the array
        System.out.print("Array elements: ");
        for (int number : numbers) {
            System.out.print(number + " ");
        }
    }
}
```

```

### 2. Linked List:

```
```java

```

```
class Node {  
    int data;  
    Node next;  
  
    Node(int data) {  
        this.data = data;  
        this.next = null;  
    }  
}  
  
public class LinkedListExample {  
    public static void main(String[] args) {  
        // Creating nodes  
        Node head = new Node(1);  
        Node second = new Node(2);  
        Node third = new Node(3);  
  
        // Connecting nodes to form a linked list  
        head.next = second;  
        second.next = third;  
  
        // Traversing the linked list  
        System.out.print("Linked List elements: ");  
        Node current = head;  
        while (current != null) {  
            System.out.print(current.data + " ");  
        }  
    }  
}
```

```
        current = current.next;  
    }  
}  
}  
```
```

#### #### 3. Stack:

```
'''java
```

```
import java.util.Stack;
```

```
public class StackExample {
```

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

```
 // Creating a stack
```

```
 Stack<Integer> stack = new Stack<>();
```

```
 // Pushing elements onto the stack
```

```
 stack.push(1);
```

```
 stack.push(2);
```

```
 stack.push(3);
```

```
 // Popping elements from the stack
```

```
 System.out.println("Popped element: " + stack.pop());
```

```
 // Displaying the stack
```

```
 System.out.println("Stack elements: " + stack);
```

```
}
```

```
}
```

```
...
```

```
4. Queue:
```

```
```java
```

```
import java.util.LinkedList;
```

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

```
public class QueueExample {
```

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

```
        // Creating a queue
```

```
        Queue<Integer> queue = new LinkedList<>();
```

```
        // Enqueuing elements
```

```
        queue.add(1);
```

```
        queue.add(2);
```

```
        queue.add(3);
```

```
        // Dequeueing elements
```

```
        System.out.println("Dequeued element: " + queue.poll());
```

```
        // Displaying the queue
```

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

```
}
```

```
}
```

```
...
```

5. Hash Table:

```java

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

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

```
public class HashTableExample {
```

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

```
 // Creating a hash table
```

```
 Map<String, Integer> hashTable = new HashMap<>();
```

```
 // Inserting key-value pairs
```

```
 hashTable.put("One", 1);
```

```
 hashTable.put("Two", 2);
```

```
 hashTable.put("Three", 3);
```

```
 // Retrieving value for a key
```

```
 System.out.println("Value for key 'Two': " + hashTable.get("Two"));
```

```
 // Displaying the hash table
```

```
 System.out.println("Hash table: " + hashTable);
```

```
}
```

```
}
```

```
```
```

6. Binary Tree:

```
```java

class TreeNode {

 int data;

 TreeNode left, right;

 TreeNode(int data) {
 this.data = data;
 this.left = this.right = null;
 }
}

public class BinaryTreeNodeExample {

 public static void main(String[] args) {
 // Creating a binary tree
 TreeNode root = new TreeNode(1);
 root.left = new TreeNode(2);
 root.right = new TreeNode(3);
 root.left.left = new TreeNode(4);
 root.left.right = new TreeNode(5);

 // Traversing the binary tree
 System.out.println("Inorder traversal:");
 inorderTraversal(root);
 }

 // Inorder traversal of a binary tree
}
```

```
private static void inorderTraversal(TreeNode root) {
 if (root != null) {
 inorderTraversal(root.left);
 System.out.print(root.data + " ");
 inorderTraversal(root.right);
 }
}
...
}
```

### ### 7. Graph:

```
```java  
import java.util.LinkedList;  
import java.util.List;  
  
class Graph {  
    int vertices;  
    List<List<Integer>> adjacencyList;  
  
    Graph(int vertices) {  
        this.vertices = vertices;  
        this.adjacencyList = new LinkedList<>();  
        for (int i = 0; i < vertices; i++) {  
            this.adjacencyList.add(new LinkedList<>());  
        }  
    }
```

```
// Function to add an edge to the graph
void addEdge(int source, int destination) {
    this.adjacencyList.get(source).add(destination);
    this.adjacencyList.get(destination).add(source);
}

}

public class GraphExample {
    public static void main(String[] args) {
        // Creating a graph with 5 vertices
        Graph graph = new Graph(5);

        // Adding edges to the graph
        graph.addEdge(0, 1);
        graph.addEdge(0, 4);
        graph.addEdge(1, 2);
        graph.addEdge(1, 3);
        graph.addEdge(1, 4);
        graph.addEdge(2, 3);
        graph.addEdge(3, 4);
    }
}

```
```

```
8. Heap:
```java  
import java.util.PriorityQueue;  
  
public class HeapExample {  
    public static void main(String[] args) {  
        // Creating a max heap (PriorityQueue in Java is a min heap by default)  
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);  
  
        // Adding elements to the max heap  
        maxHeap.add(5);  
        maxHeap.add(2);  
        maxHeap.add(8);  
        maxHeap.add(1);  
  
        // Polling elements from the max heap  
        System.out.println("Max element: " + maxHeap.poll());  
  
        // Displaying the max heap  
        System.out.println("Max heap elements: " + maxHeap);  
    }  
}```
```

These programs demonstrate the basic functionality of each data structure. Let me know if you have any specific questions or if you'd like further explanations!

JUNIT

=====

It is an open-source testing framework for java programmers. The java programmer can create test cases and test his/her own code.

It is one of the unit testing framework. Current version is junit 4.

To perform unit testing, we need to create test cases. The unit test case is a code which ensures that the program logic works as expected.

The org.junit package contains many interfaces and classes for junit testing such as Assert, Test, Before, After etc.

1) Manual Testing

If you execute the test cases manually without any tool support, it is known as manual testing. It is time consuming and less reliable.

2) Automated Testing

If you execute the test cases by tool support, it is known as automated testing. It is fast and more reliable. script example : Selenium WebDriver

Annotations for Junit testing

@Test annotation specifies that method is the test method.

@Test(timeout=1000) annotation specifies that method will be failed if it takes longer than 1000 milliseconds (1 second).

@BeforeClass annotation specifies that method will be invoked only once, before starting all the tests.

@Before annotation specifies that method will be invoked before each test.

@After annotation specifies that method will be invoked after each test.

@AfterClass annotation specifies that method will be invoked only once, after finishing all the tests.

Assert class

The org.junit.Assert class provides methods to assert the program logic.

The common methods of Assert class are as follows:

`void assertEquals(boolean expected,boolean actual)`: checks that two primitives/objects are equal. It is overloaded.

`void assertTrue(boolean condition)`: checks that a condition is true.

`void assertFalse(boolean condition)`: checks that a condition is false.

`void assertNull(Object obj)`: checks that object is null.

`void assertNotNull(Object obj)`: checks that object is not null.

MOCKITO TESTING

What is Mocking ?

Mocking is a process of developing the objects that act as the mock or clone of the real objects.

In other words, mocking is a testing technique where mock objects are used instead of real objects for testing purposes.

Mock objects provide a specific (dummy) output for a particular (dummy) input passed to it.

The mocking technique is not only used in Java but also used in any object-oriented programming language.

There are many frameworks available in Java for mocking, but Mockito is the most popular framework among them.

What is Mockito ?

Mockito is a Java-based mocking framework used for unit testing of Java application. Mockito plays a crucial role in developing testable applications.

Mockito was released as an open-source testing framework under the MIT (Massachusetts Institute of Technology) License.

It internally uses the Java Reflection API to generate mock objects for a specific interface.

Mock objects are referred to as the dummy or proxy objects used for actual implementations

Methods in MOCKITO

`mock()` : It is used to create mock objects of a given class or interface. Mockito contains five `mock()` methods with different arguments.

When we didn't assign anything to mocks, they will return default values. All five methods perform the same function of mocking the objects.

Ex : Calculate calculate = mock(Calculate.class);

`when()` : It enables stubbing methods. It should be used when we want to mock to return specific values when particular methods are called.

In simple terms, "When the XYZ() method is called, then return ABC." It is mostly used when there is some condition to execute.

Ex : `when(mock.someCode ()).thenReturn(5);`

`verify()`: The `verify()` method is used to check whether some specified methods are called or not.

In simple terms, it validates the certain behavior that happened once in a test. It is used at the bottom of the testing code to assure that the defined methods are called.

`spy()` : Mockito provides a method to partially mock an object, which is known as the spy method.

When using the spy method, there exists a real object, and spies or stubs are created of that real object.

`spy()` method: It creates a spy of the real object. The spy method calls the real methods unless they are stubbed.

We should use the real spies carefully and occasionally, for example, when dealing with the legacy code.

Syntax: `<T> spy(T object)`

`spy()` method with Class: It creates a spy object based on class instead of an object. The `spy(Class<T> classToSpy)` method is particularly useful for spying abstract classes because they cannot be instantiated.

Syntax: `<T> spy(Class<T> classToSpy)`

```
List spyArrayList = spy(ArrayList.class);
```

`reset()` : The Mockito `reset()` method is used to reset the mocks. It is mainly used for working with the container injected mocks.

Usually, the `reset()` method results in a lengthy code and poor tests.

It's better to create new mocks rather than using `reset()` method. That is why the `reset()` method is rarely used in testing

1. We use the `@ExtendWith(MockitoExtension.class)` annotation at the class level to tell JUnit 5 to initialize mocks and prepare them for injection.

2. The `@Mock` annotation creates a mock implementation of the `EmployeeRepository` interface.

3. `@InjectMocks` is used to inject mock fields into the service instance.

4. In the test method `testGetEmployeeDetails`, we define the behavior of the mock (`when(repository.findEmployeeById(1L)).thenReturn(emp);`) and then execute the service method to be tested.

Additionally

=====

1. **Convert JSON to Object and Object to JSON**:

=====

- **Jackson Dependency**: Jackson is a popular Java library for handling JSON data. You can use the `'jackson-databind'` dependency to convert JSON strings to Java objects and vice versa.

- **Conversion Process**:

- To convert JSON to an object, you can use the `'ObjectMapper'` class from Jackson. The `'readValue()'` method of `'ObjectMapper'` is used to deserialize JSON data into a Java object.

- To convert an object to JSON, you can use the `'writeValueAsString()'` method of `'ObjectMapper'`. This method serializes a Java object into a JSON string.

- **Example**:

```

```java

ObjectMapper objectMapper = new ObjectMapper();

// Convert JSON to Object

MyObject myObject = objectMapper.readValue(jsonString, MyObject.class);

// Convert Object to JSON

String jsonOutput = objectMapper.writeValueAsString(myObject);

```

```

2. **Convert XML to Object and Object to XML**:

- **JAXB Dependency**: JAXB (Java Architecture for XML Binding) is a Java API for mapping XML to Java objects and vice versa. You can use the `jaxb-api` dependency for XML data binding.

- **Conversion Process**:

- To convert XML to an object, you need to create a JAXB context for your class and then use the `Unmarshaller` to deserialize XML data into a Java object.

- To convert an object to XML, you need to create a JAXB context for your class and then use the `Marshaller` to serialize the object into XML.

- **Example**:

```

```java

JAXBContext jaxbContext = JAXBContext.newInstance(MyObject.class);

Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();

Marshaller marshaller = jaxbContext.createMarshaller();


```

```
// Convert XML to Object

MyObject myObject = (MyObject) unmarshaller.unmarshal(xmlFile);

// Convert Object to XML

marshaller.marshal(myObject, xmlOutputFile);
...

...
```

These are the general steps and concepts involved in converting JSON and XML data to Java objects and vice versa using Jackson for JSON and JAXB for XML in a Spring Boot application. Depending on your specific use case and requirements, you may need to customize the conversion process or handle exceptions appropriately.

#### INTERVIEW QUESTION:

---

Sure, let's break down each of these concepts:

##### ### 1. Composite Key in SQL:

- **Definition**: A composite key is a combination of two or more columns that uniquely identify a record in a database table.
- **Example**: In a table representing orders, a composite key might be a combination of `order\_id` and `customer\_id`.
- **Usage**: It's used when a single column cannot uniquely identify records, requiring a combination of columns for uniqueness.

### ### 2. @Qualifier Annotation:

- **Definition**: `@Qualifier` is used in Spring Framework for resolving bean ambiguity by specifying the bean to be injected by its name.

- **Example**: In case multiple beans of the same type are available, `@Qualifier("beanName")` specifies which one to inject.

- **Usage**: When there are multiple beans of the same type and Spring needs clarification on which one to inject.

### ### 3. @Required Annotation:

- **Definition**: `@Required` annotation indicates that the annotated property must be populated during bean initialization in Spring.

- **Example**: Applied to setter methods, ensuring the corresponding properties are set before the bean is used.

- **Usage**: Ensures that mandatory properties are set, preventing configuration errors.

### ### 4. SingletonList:

- **Definition**: `SingletonList` is an immutable list containing a single specified element, useful when you need an immutable list with only one element.

- **Example**: ``Collections.singletonList("example")`` creates a list containing only the string "example".

- **Usage**: When you need to represent a list with a single element and ensure its immutability.

### ### 5. Usage of String for Passwords:

- **Explanation**: Strings are immutable, making them safer for storing sensitive information like passwords, as they cannot be modified in memory.

- **Example**: ``String password = "secret";``

- **Usage**: Prevents security vulnerabilities by avoiding accidental modification or exposure of sensitive information.

### ### 6. Difference between InnerJoin and OuterJoin:

- **Inner Join**: Retrieves records that have matching values in both tables based on a specified condition.

- **Outer Join**: Retrieves all records from one table and only the matching records from the other table, based on a specified condition.

- **Example**: Inner Join: `SELECT \* FROM table1 INNER JOIN table2 ON table1.id = table2.id;` Outer Join: `SELECT \* FROM table1 LEFT JOIN table2 ON table1.id = table2.id;`

- **Usage**: Inner Join when you need only matching records, Outer Join when you need all records from one table regardless of matches in the other.

These explanations provide a concise understanding of each concept along with simple examples and usage scenarios.

1. difference between hashmap and hashtable?

2. difference between Runnable and Callable interfaces in Thread?

3. Operators in java ? mainly Unary, Shift, Bitwise and Assingment?

4. Aggregations in SQL?

5. Stored Procedures in Java and MySQL?

6. Union and Union All in SQL?

7. Self Join in SQL ?

## IaaS, SaaS, PaaS, and IDaaS architectures and concepts

---

Sure, let's break down each of these concepts:

### 1. \*\*IaaS (Infrastructure as a Service):\*\*

- IaaS provides virtualized computing resources over the internet. It includes virtualized computing resources such as virtual machines, storage, and networking. With IaaS, users can rent these resources on a pay-as-you-go basis rather than purchasing and maintaining physical hardware. Examples of IaaS providers include Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP).

### 2. \*\*SaaS (Software as a Service):\*\*

- SaaS delivers software applications over the internet on a subscription basis. Instead of installing and maintaining software on individual computers, users can access the software via a web browser or API. SaaS providers handle all the maintenance, including updates, patches, and security. Examples of SaaS applications include Google Workspace (formerly G Suite), Microsoft Office 365, Salesforce, and Dropbox.

### 3. \*\*PaaS (Platform as a Service):\*\*

- PaaS provides a platform allowing customers to develop, run, and manage applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app. PaaS typically includes development tools, database management systems, middleware, and other services. It allows developers to focus on writing code and deploying applications without worrying about underlying infrastructure.

Examples of PaaS providers include Heroku, Google App Engine, and Microsoft Azure App Service.

#### 4. \*\*IDaaS (Identity as a Service):\*\*

- IDaaS is a cloud-based service that provides identity and access management capabilities to organizations. It allows companies to manage user identities, access privileges, and authentication methods centrally. IDaaS typically offers features such as single sign-on (SSO), multi-factor authentication (MFA), user provisioning, and directory services. Examples of IDaaS providers include Okta, Azure Active Directory, and Ping Identity.

When should we use HyperVisor and Docker ?

=====

#### \*\*Hypervisor:\*\*

- \*\*Description:\*\* A hypervisor(VM) is a software layer that allows multiple operating systems to run concurrently on a single physical machine. It creates and manages virtual machines (VMs), each with its own OS, enabling efficient resource sharing and isolation.

- \*\*Use Case:\*\* Ideal for running a single application.
- \*\*Description:\*\* Requires a single hypervisor and a single operating system (OS).
- \*\*Resource Usage:\*\* Each application runs in its own virtual machine (VM) with its own OS, leading to potential resource waste.
- \*\*Cost Implications:\*\* May require purchasing licenses for multiple OS instances.
- \*\*Summary:\*\* Suitable for scenarios where only one application needs to be isolated and run independently.

- **Example:** Deploy your monolithic Spring Boot application within the VM. You create a VM named "EcommerceAppVM" with Ubuntu Server installed. Within this VM, you deploy your monolithic Spring Boot application for the entire e-commerce platform, including order management, inventory management, product catalog, and user authentication.

#### **\*\*Docker:\*\***

- **Description:** Docker is a platform that simplifies the development, deployment, and management of applications using containerization technology. It packages applications and their dependencies into lightweight, portable containers, allowing them to run consistently across different environments without the overhead of traditional virtualization.

- **Use Case:** Suitable for deploying and running multiple services or applications simultaneously.

- **Description:** Utilizes containers, which package each application with its dependencies into a single unit.

- **Resource Usage:** Containers share the host OS kernel, reducing resource overhead compared to VMs.

- **Efficiency:** Offers better resource utilization and efficiency by avoiding the need for separate OS instances for each application.

- **Cost Savings:** Minimizes resource wastage and eliminates the need to purchase licenses for multiple OS instances.

- **Summary:** Preferred for scenarios where multiple applications need to be deployed and managed efficiently on the same hardware.

- **Example:** Deploy and Run your microservices on Docker Containers. You create Docker images for the order service, inventory service, product service, and authentication service. Then, you run multiple instances of these Docker images as containers on a Docker host. Each container represents a separate microservice, allowing you to independently scale and update components of the e-commerce platform.

## DOCKER

=====

Docker is a containerization platform that allows you to package your application and its dependencies into a standardized unit called a container. Containers are lightweight, portable, and isolated environments that run on top of the host operating system. Docker provides tools and APIs to build, ship, and run containers consistently across different environments.

#### ### Why Use Docker?

1. **Consistency**: Docker ensures consistency between development, testing, and production environments. You can package your application and its dependencies into a container image, guaranteeing that it runs the same way everywhere.
2. **Isolation**: Containers provide isolation for your application, preventing conflicts with other applications or dependencies on the host system.
3. **Portability**: Containers are portable across different platforms and cloud providers. You can build your application once and run it anywhere that supports Docker.
4. **Efficiency**: Docker containers are lightweight and share the host operating system's kernel, resulting in faster startup times and efficient resource utilization.
5. **Scalability**: Docker enables easy scaling of your application by running multiple instances of containers, either manually or through orchestration tools like Kubernetes.

#### ### Docker Concepts:

1. **Docker Images**: An image is a read-only template with instructions for creating a Docker container. It contains the application code, runtime, libraries, and other dependencies needed to run the application.

2. **Docker Containers**: A container is a runnable instance of a Docker image. It encapsulates the application and its runtime environment, including the filesystem, network, and processes.

3. **Dockerfile**: A Dockerfile is a text file that contains instructions for building a Docker image. It defines the base image, dependencies, environment variables, and commands needed to set up the container environment.

4. **Docker Registry**: A registry is a repository for storing and sharing Docker images. The Docker Hub is a public registry, but you can also set up private registries for internal use.

5. **Docker Engine**: Docker Engine is the runtime environment for running Docker containers. It includes the Docker daemon, API, and command-line interface (CLI) tools for managing containers and images.

#### With Docker vs. Without Docker:

##### With Docker:

1. You package your Spring Boot microservices application along with its dependencies into Docker images.
2. You define Dockerfiles for each microservice to specify the container environment and build instructions.
3. You use Docker Compose or Kubernetes to manage and orchestrate your Dockerized microservices application.
4. Docker ensures consistency, isolation, and portability of your application across different environments.

#### Without Docker:

1. You manually install and configure dependencies (Java, Spring Boot, etc.) on each server or environment where you deploy your microservices application.
2. You need to ensure that each server has the correct versions of dependencies, leading to potential inconsistencies and conflicts.
3. Managing dependencies and ensuring consistency becomes challenging, especially in large-scale deployments.
4. Deployment and scaling are more complex and time-consuming without the lightweight, portable nature of Docker containers.

In summary, Docker simplifies the development, deployment, and management of microservices applications by providing a consistent, isolated, and portable environment for running your application and its dependencies.

Certainly! Below are some common Docker commands and instructions you can use for building and running a Java-Spring Boot-Angular microservices application:

### Docker Commands:

=====

1. \*\*Build Docker Image\*\*:

```

`docker build -t myapp .`

```

This command builds a Docker image named `myapp` using the Dockerfile in the current directory (`.`).

2. **Run Docker Container**:

```

```
docker run -p 8080:8080 myapp
```

```

This command runs a Docker container based on the `myapp` image, forwarding port 8080 from the container to port 8080 on the host.

3. **List Running Containers**:

```

```
docker ps
```

```

This command lists all running Docker containers.

4. **Stop Container**:

```

```
docker stop <container_id>
```

```

Replace `<container\_id>` with the ID of the container you want to stop.

5. **Remove Container**:

```

```
docker rm <container_id>
```

```

This command removes a stopped container. Use `-f` flag to force removal of a running container.

6. **Remove Image**:

```

```
docker rmi <image_id>
```

```

This command removes a Docker image. Use `-f` flag to force removal of an image in use by a container.

## 7. \*\*View Logs\*\*:

```

```
docker logs <container_id>
```

```

This command displays the logs of a specific container.

## ### Dockerfile Instructions for Spring Boot Application:

---

```Dockerfile

```
# Use a base image with Java installed
```

```
FROM openjdk:11-jdk-slim
```

```
# Set working directory in the container
```

```
WORKDIR /app
```

```
# Copy the packaged Spring Boot application JAR file into the container
```

```
COPY target/myapp.jar /app
```

```
# Expose the port that the Spring Boot application will run on
```

```
EXPOSE 8080
```

```
# Command to run the Spring Boot application when the container starts  
CMD ["java", "-jar", "myapp.jar"]  
```
```

#### Dockerfile Instructions for Angular Application:

```
```Dockerfile
```

```
# Use a base image with Node.js and NPM installed  
FROM node:14 as build
```

```
# Set working directory in the container
```

```
WORKDIR /app
```

```
# Copy the Angular application source code into the container
```

```
COPY ..
```

```
# Install dependencies and build the Angular application
```

```
RUN npm install && npm run build
```

```
# Use a lightweight base image to serve the Angular application
```

```
FROM nginx:alpine
```

```
# Copy the built Angular application files into the NGINX web server directory
```

```
COPY --from=build /app/dist/myapp /usr/share/nginx/html
```

```
# Expose port 80 to access the Angular application
```

```
EXPOSE 80
```

```
---
```

These are basic examples. You may need to adjust them based on your project structure and requirements. Let me know if you need further explanation or assistance with any specific command or instruction!

```
---
```

```
## 🚀 FULL DEPLOYMENT FLOW OF SPRING BOOT APP
```

```
=====
```

```
---
```

```
#### ✅ **STEP 1: Maven Build & Docker Image (Local Setup)**
```

```
**Already done by you!**
```

```
```pom.xml
```

```
<finalName>spring-with-docker</finalName>
```

```

```

```
```Dockerfile
```

```
FROM openjdk:17
```

```
EXPOSE 9191
```

```
ADD target/spring-with-docker.jar spring-with-docker.jar
```

```
ENTRYPOINT ["java", "-jar", "spring-with-docker.jar"]
```

```
---
```

```
```bash
```

```
mvn clean install
```

```
docker build -t spring-with-docker .
```

```
docker run -p 9192:9191 spring-with-docker
```

```

```

✓ \*\*You can test app locally at:\*\* `http://localhost:9192/message`

```

```

## 🛡 STEP 2: Push Code to \*\*GitLab\*\*

### Why?

To enable CI/CD and store your source code remotely for automated builds.

### How?

1. Create a \*\*GitLab repo\*\*

2. Run:

```
```bash
```

```
git init
```

```
git remote add origin https://gitlab.com/your-username/your-repo.git
```

```
git add .  
git commit -m "initial commit"  
git push -u origin main  
...  
---
```

🚀 STEP 3: Push Docker Image to **Docker Hub**

Why?

Docker Hub stores your image, so Kubernetes/AWS can pull and run it.

How?

1. **Login** to Docker Hub:

```
```bash
```

```
docker login
```

```
...

...
```

#### 2. \*\*Tag\*\* your image:

```
```bash
```

```
docker tag spring-with-docker your-dockerhub-username/spring-with-docker
```

```
...  
  
...
```

3. **Push** the image:

```
```bash
```

```
docker push your-dockerhub-username/spring-with-docker
```

```

STEP 4: Automate Build & Push with **GitLab CI/CD**

Why?

To automatically build and push Docker images when you push code.

How?

Create ``.gitlab-ci.yml` file in root directory:

```
```yaml
```

```
image: docker:latest
```

```
services:
```

```
 - docker:dind
```

```
stages:
```

```
 - build
```

```
variables:
```

```
 DOCKER_DRIVER: overlay2
```

```
before_script:
```

```
 - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD"
```

```
build:
 stage: build
 script:
 - docker build -t your-dockerhub-username/spring-with-docker .
 - docker push your-dockerhub-username/spring-with-docker

```

 This automates Docker image creation + pushing on every GitLab push.

```

 STEP 5: Deploy to **Kubernetes (Minikube or Cloud)**
```

#### Why?

To scale, manage, and auto-restart containers.

```

5.1 Create Deployment YAML:
```

```
```yaml  
# deployment.yaml  
apiVersion: apps/v1  
kind: Deployment  
metadata:
```

```
name: spring-docker-app

spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-app
  template:
    metadata:
      labels:
        app: spring-app
    spec:
      containers:
        - name: spring-app
          image: your-dockerhub-username/spring-with-docker
        ports:
          - containerPort: 9191
```

```

#### ### 5.2 Create Service YAML:

```
```yaml
# service.yaml
apiVersion: v1
kind: Service
```

```
metadata:  
  name: spring-docker-service  
  
spec:  
  type: NodePort  
  
  selector:  
    app: spring-app  
  
  ports:  
    - port: 9191  
      targetPort: 9191  
      nodePort: 30091  
  
  ...  
  
---
```

5.3 Apply to Kubernetes:

```
```bash  
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml

...

```

 Access via: `http://<NodeIP>:30091/message`

##  STEP 6: Deploy to \*\*AWS\*\*

---

### ### 6.1 Using \*\*EC2 Instance + Docker\*\*

1. Launch EC2 (Amazon Linux or Ubuntu)

2. SSH into EC2:

```
```bash
```

```
ssh -i "your-key.pem" ec2-user@<your-ec2-ip>
```

```
```
```

3. Install Docker and run your app:

```
```bash
```

```
sudo yum update -y
```

```
sudo yum install docker -y
```

```
sudo service docker start
```

```
docker pull your-dockerhub-username/spring-with-docker
```

```
docker run -d -p 80:9191 your-dockerhub-username/spring-with-docker
```

```
```
```

 Open EC2 IP in browser: `http://<ec2-ip>/message`

---

### ### 6.2 Using \*\*AWS EKS (Managed Kubernetes)\*\*

> EKS = AWS Kubernetes. Better for scaling in production.

1. Create EKS Cluster (via AWS Console or Terraform)
2. Install `eksctl`, `kubectl`, and `aws-cli`
3. Deploy same `deployment.yaml` & `service.yaml`
4. Use `kubectl apply` to deploy

---

### ## 🎉 Summary Table:

| Step | Tool           | Purpose                      |
|------|----------------|------------------------------|
| 1    | Maven + Docker | Build locally                |
| 2    | GitLab         | Source version control       |
| 3    | Docker Hub     | Store Docker image           |
| 4    | GitLab CI/CD   | Automate Docker build & push |
| 5    | Kubernetes     | Scale/manage containers      |
| 6    | AWS (EC2/EKS)  | Run app in the cloud         |

---

Would you like a \*\*PDF/Word version\*\* of this full flow with diagrams?

Or want to explore \*\*CI/CD with Kubernetes\*\*, \*\*AWS Terraform\*\*, or \*\*Monitoring with Prometheus\*\* next?

## KUBERNATES :

=====

Kubernetes (often abbreviated as K8s) is an open-source container orchestration platform for automating the deployment, scaling, and management of containerized applications. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF). Kubernetes provides a robust framework for managing distributed systems and microservices at scale.

### ### Why Use Kubernetes?

1. **Orchestration**: Kubernetes automates the deployment, scaling, and management of containerized applications, freeing developers from manual intervention and reducing operational overhead.

2. **Scalability**: Kubernetes enables horizontal scaling of applications by automatically adding or removing containers based on resource utilization and user-defined scaling policies.

3. **High Availability**: Kubernetes ensures high availability of applications by automatically restarting containers that fail, scheduling containers on healthy nodes, and providing self-healing capabilities.

4. **Resource Utilization**: Kubernetes optimizes resource utilization by intelligently scheduling containers on nodes with available resources and efficiently packing multiple containers on the same node.

5. **Rolling Updates**: Kubernetes supports rolling updates for applications, allowing you to deploy new versions of your application without downtime by gradually replacing old containers with new ones.

#### #### Kubernetes Concepts:

1. **Pods**: Pods are the smallest deployable units in Kubernetes. A pod can contain one or more containers that share the same network namespace and storage volumes.

2. **Deployments**: Deployments manage the lifecycle of pods and provide features like rolling updates and scaling. They ensure that a specified number of replicas of a pod are running at any given time.

3. **Services**: Services provide a stable network endpoint for accessing a set of pods. They enable load balancing, service discovery, and internal communication between different parts of an application.

4. **ReplicaSets**: ReplicaSets are responsible for maintaining a specified number of replicas of a pod. They ensure that the desired number of pods are running and automatically create or delete pods as needed.

5. **Nodes**: Nodes are individual machines (physical or virtual) in a Kubernetes cluster where containers are deployed. Each node runs Kubernetes services and hosts one or more pods.

#### ### With Kubernetes vs. Without Kubernetes:

##### #### With Kubernetes:

- You deploy your Spring Boot microservices application as containers in a Kubernetes cluster.
- Kubernetes automatically manages the deployment, scaling, and networking of your application, ensuring high availability and efficient resource utilization.
- You can use Kubernetes features like rolling updates, auto-scaling, and service discovery to simplify application management and improve reliability.

##### #### Without Kubernetes:

- You manually deploy and manage your microservices application on individual servers or virtual machines.
- Managing deployment, scaling, networking, and reliability requires manual intervention and custom scripting.
- Ensuring high availability and efficient resource utilization can be challenging and time-consuming, especially in large-scale deployments.

In summary, Kubernetes provides a powerful platform for automating and managing containerized applications, simplifying deployment, scaling, and management tasks, and improving application reliability and efficiency.

Certainly! Kubernetes provides a rich set of commands (kubectl) and features for managing containerized applications within a cluster. Below are some common commands and restrictions you may encounter when working with Kubernetes:

#### #### Common kubectl Commands:

---

##### 1. \*\*Creating Resources\*\*:

- `kubectl create`: Create a resource from a file, URL, or stdin.
- `kubectl apply`: Apply a configuration to a resource.

##### 2. \*\*Managing Resources\*\*:

- `kubectl get`: Retrieve information about resources.
- `kubectl describe`: Show details about a specific resource.
- `kubectl edit`: Edit a resource in your preferred editor.

##### 3. \*\*Scaling\*\*:

- `kubectl scale`: Scale the number of replicas of a resource (e.g., Deployment, ReplicaSet).

##### 4. \*\*Deleting Resources\*\*:

- `kubectl delete`: Delete one or more resources.
- `kubectl delete -f <filename>`: Delete resources defined in a YAML file.

##### 5. \*\*Viewing Logs\*\*:

- `kubectl logs`: Print the logs of a container in a pod.

## 6. \*\*Port Forwarding\*\*:

- `kubectl port-forward`: Forward one or more local ports to a pod.

## 7. \*\*Executing Commands\*\*:

- `kubectl exec`: Execute a command in a container in a pod.

## 8. \*\*Troubleshooting\*\*:

- `kubectl describe pod <pod\_name>`: Describe the state of a pod, including events and conditions.

- `kubectl get events`: View cluster events.

## #### Restrictions in Kubernetes:

### 1. \*\*Resource Limits\*\*:

- Kubernetes imposes resource limits on pods to prevent over-utilization of cluster resources. You may need to specify resource requests and limits in your pod manifests.

### 2. \*\*Security Policies\*\*:

- Kubernetes enforces security policies to restrict access and prevent unauthorized actions. You may encounter restrictions related to role-based access control (RBAC), network policies, and pod security policies.

### 3. \*\*Namespaces\*\*:

- Kubernetes uses namespaces to organize resources and provide isolation between different environments or teams. Access to resources may be restricted based on namespace permissions.

4. **\*\*Cluster Quotas\*\*:**

- Kubernetes administrators can define quotas to limit the amount of compute resources or the number of objects (e.g., pods, services) that can be created within a namespace.

5. **\*\*Networking\*\*:**

- Kubernetes networking policies and plugins may impose restrictions on network communication between pods and external services. Network policies can control traffic flow and enforce security rules.

6. **\*\*Persistent Storage\*\*:**

- Kubernetes provides persistent volume (PV) and persistent volume claim (PVC) objects for managing storage resources. Access to persistent storage may be restricted based on policies defined by the cluster administrator.

Understanding these commands and restrictions is essential for effectively managing and troubleshooting applications deployed on Kubernetes. Additionally, familiarizing yourself with Kubernetes documentation and best practices can help you navigate these complexities and optimize your application deployment process.

KAFKA Tutorial

=====

Apache Kafka

-----

Apache Kafka is a software platform which is based on a distributed streaming process.

It is a publish-subscribe messaging system which let exchanging of data between applications, servers, and processors as well.

## What is a messaging system

---

A messaging system is a simple exchange of messages between two or more persons, devices, etc. A publish-subscribe messaging system allows a sender to send/write the message and a receiver to read that message.

In Apache Kafka, a sender is known as a producer who publishes messages, and a receiver is known as a consumer who consumes that message by subscribing it.

## What is Streaming process

---

A streaming process is the processing of data in parallelly connected systems.

This process allows different applications to limit the parallel execution of the data, where one record executes without waiting for the output of the previous record.

Therefore, a distributed streaming platform enables the user to simplify the task of the streaming process and parallel execution.

Therefore, a streaming platform in Kafka has the following key capabilities:

1. As soon as the streams of records occur, it processes it.
2. It works similar to an enterprise messaging system where it publishes and subscribes streams of records.
3. It stores the streams of records in a fault-tolerant durable way.

Producer API: This API allows/permits an application to publish streams of records to one or more topics. (discussed in later section)

Consumer API: This API allows an application to subscribe one or more topics and process the stream of records produced to them.

**Streams API:** This API allows an application to effectively transform the input streams to the output streams. It permits an application to act as a stream processor which consumes an input stream from one or more topics, and produce an output stream to one or more output topics.

**Connector API:** This API executes the reusable producer and consumer APIs with the existing data systems or applications.

### Why Apache Kafka

---

Apache Kafka is a software platform that has the following reasons which best describes the need of Apache Kafka.

1. Apache Kafka is capable of handling millions of data or messages per second.
2. Apache Kafka works as a mediator between the source system and the target system.

Thus, the source system (producer) data is sent to the Apache Kafka, where it decouples the data, and the target system (consumer) consumes the data from Kafka.

3. Apache Kafka is having extremely high performance, i.e., it has really low latency value less than 10ms which proves it as a well-versed software.

4. Apache Kafka has a resilient architecture which has resolved unusual complications in data sharing.

5. Organizations such as NETFLIX, UBER, Walmart, etc. and over thousands of such firms make use of Apache Kafka.

6. Apache Kafka is able to maintain the fault-tolerance. Fault-tolerance means that sometimes a consumer successfully consumes the message that was delivered by the producer.

But, the consumer fails to process the message back due to backend database failure, or due to presence of a bug in the consumer code.

In such a situation, the consumer is unable to consume the message again. Consequently, Apache Kafka has resolved the problem by reprocessing the data

### Kafka key Concepts

---

Producer: A producer is an application or service that sends data (messages or events) to a Kafka topic.

Consumer: A consumer is an application or service that reads data from a Kafka topic.

Broker: A broker is a Kafka server that stores messages and serves them to consumers. Kafka clusters usually have multiple brokers.

Topic: A topic is a category or feed name where messages are published by producers and read by consumers. Topics help organize messages.

Partition: Each topic can be divided into partitions. Partitions allow Kafka to scale horizontally and process data in parallel.

Offset: An offset is a unique identifier assigned to each message in a partition. It helps keep track of the read position for consumers.

**Consumer Group:** A consumer group is a group of consumers that work together to consume messages from a topic. Each message is delivered to only one consumer in the group.

**ZooKeeper:** ZooKeeper was used for managing Kafka's cluster metadata and coordination. Starting from Kafka version 3.0, it is optional and can be replaced with Kafka's built-in KRaft mode.

Here are some practical example use cases for using **Kafka in an Employee Management System**:

---

---

### ### 1. Audit Logging

Whenever an employee is created, updated, or deleted, send an event to Kafka like:

- "Employee ID 101 created by Admin"
- "Employee ID 205 updated: address changed"

Kafka stores this data in a topic, which can later be consumed by a logging or auditing service.

---

### ### 2. Asynchronous Email or SMS Notifications

Instead of sending emails directly after an employee is onboarded, send a message to a Kafka topic:

- Topic: `employee-notifications`
- Message: "Welcome email for employee ID 102"

A separate notification microservice consumes this topic and sends the email/SMS, making the main process faster and non-blocking.

---

### ### 3. Real-Time Dashboard Updates

For real-time analytics, stream data through Kafka:

- When new employees join, send the data to a topic.
- The dashboard service listens to that topic and updates the UI live.

---

### ### 4. Sync Data Across Microservices

If you have microservices like:

- employee-service
- payroll-service
- attendance-service

You can use Kafka to sync data changes across services. For example, when an employee's department changes, publish an event to Kafka. The payroll and attendance services consume it and update their local data.

---

#### ### 5. Trigger Background Jobs

When an employee is deactivated or leaves the company, Kafka can trigger background tasks:

- Revoke system access
- Archive documents
- Remove from mailing lists

Each task is handled by a different service that listens to Kafka topics.

---

#### ### 6. Integrate with External Systems

Suppose HR data needs to be sent to an external insurance provider. You can publish employee events to Kafka, and an external connector service consumes and pushes the data outside your system.

---

