Connecting spring boot to dual DB:

1. In application.properties give the two datasource properties
2. Configure two Data Config, like BookDBConfig and UserDBConfig.
3. Should have different packages for both UserRepo and BookRepo and model classes as well.

@Configuration

@EnableTransactionManagement

@EnableJpaRepositories(

        entityManagerFactoryRef = "entityManagerFactory",

        basePackages = {

                "com.ashokit.user.repository"

        }

)

public class UserDBConfig {

    @Primary

    @Bean(name = "dataSource")

    @ConfigurationProperties(prefix = "spring.user.datasource")

    public DataSource dataSource() {

        return DataSourceBuilder.create().build();

    }

    @Primary

    @Bean(name = "entityManagerFactory")

    public LocalContainerEntityManagerFactoryBean entityManagerFactory(EntityManagerFactoryBuilder builder,

            @Qualifier("dataSource") DataSource dataSource) {

        HashMap<String, Object> properties = new HashMap<>();

        properties.put("hibernate.hbm2ddl.auto", "update");

        return builder.dataSource(dataSource)

                      .properties(properties)

                      .packages("com.ashokit.user.model")

                      .persistenceUnit("User")

                      .build();

    }

    @Primary

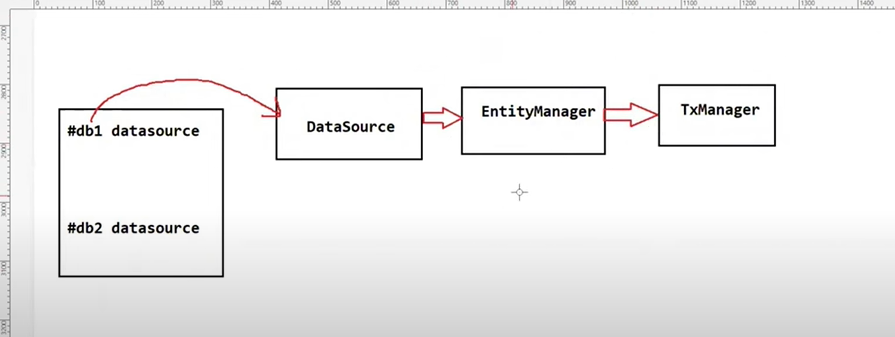
    @Bean(name = "transactionManager")

    public PlatformTransactionManager transactionManager(@Qualifier("entityManagerFactory") EntityManagerFactory entityManagerFactory) {

        return new JpaTransactionManager(entityManagerFactory);

    }

}



**Q) What is hot reloading and how does it work?\***

Hot reloading allows developers to make changes to their code, resources, or configuration files while an application is running, and see those changes immediately reflected without needing to **\*restart the application.\* using Dev Tools.**

**Handling multiple users at the same time:**

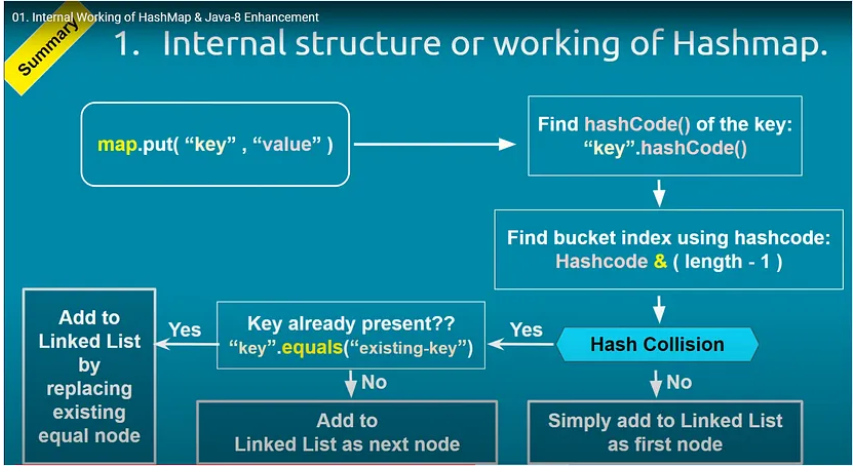
1. Thread pool and asynchronous : using @Async annotation on the method tells spring boot that it should run in a separate thread.
2. How to synchronous two java process: shared DB and share file system
3. JVM architecture: class Loader, Run time Data areas and execution engine

Class loader loads class file into JVM, Runtime Data areas keeps memory for variables and all.. And execution engine will execute the instructions.(wrie once and run anywhere)

1. Stack and heap memory: Heap is shared across threads while the stack is for each method having thread safe
2. Java makes its engine faster by using JIT compiler, byte code to native machine code..
3. Super calls the method in parent class even if we have overridden it.
4. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
5. **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.
6. **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.
7. **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.

**How does polymorphism help the Java collection framework?**

Letting it treated as sameWay, list interface we can use ArrayList or LinkedList



How HashCode and Equals work together in a Collection?

The Relationship Between hashCode() and equals() in Java Collections

In Java, the hashCode() and equals() methods are used in collections to store, retrieve, and identify objects efficiently. They work hand in hand to ensure fast access and accurate results in a collection.

Understanding hashCode():

The hashCode() method is a part of the Java Object class. It returns an integer representation of the object memory address. When an object is stored in a hashed data structure like a HashMap, HashSet, or Hashtable, Java uses the hashCode() method to determine where the object should be stored.

Understanding equals():

The equals() method is used to determine the equality of two objects. If equals() returns true for two objects, this means that they are equal in terms of their defined equality criteria (which can be overridden in a custom class to suit specific needs).

Working Together:

hashCode() and equals() methods work together in the following way:

1. When an object is inserted into a hashed collection, the hashCode() method is called first to determine the correct bucket location for storing the object.
2. If a collision occurs (two objects have the same hash code), the collection calls the equals() method to check if the objects are truly equal. If they are equal, the new object replaces the existing one. If they aren't, the collection resolves the collision and stores both objects.

Ensuring Consistency:

To ensure consistency and avoid misbehavior in collections, there are two important contracts:

1. If two objects are equal (as determined by the equals() method), then calling hashCode() on each of the two objects must produce the same result.
2. If two objects are unequal (as determined by the equals() method), it's not required that calling hashCode() on each of the two objects will produce distinct results. However, producing distinct results for unequal objects may improve the performance of hash tables.

In Summary:

hashCode() and equals() methods are fundamental to the correct functioning of hash-based collections in Java. They are used to determine the storage location of objects and to avoid storing duplicate objects. These two methods should be overridden together and behave consistently with each other to avoid unexpected results in collections.

**Using mutable object as key in HashMap?**

When mutable object is inserted in hashmap, it might change and hashcode might change then its difficult to retrieve the object.

**Can we build a server without using spring framework?**

We can build that using server socket connection, accepting requests from client and write code for all network connection,listening on port

**How to deploy a servlet web application without spring?**

Need to create a war file, configure web.xml file for servlet mapping and all in the end deploy and publis to tomcat server.

**AutoConfiguration:**

Add spring starter web

**Slow in production faster in Dev:**

Check logs, blue/green deployments , optimize query

**Circular Dependency:**

spring.main.allow-circular-references=true, using @Lazy annotation

SOLID Design Principles:

<https://medium.com/@javatechie/solid-design-principle-java-ae96a48db97>

==================================================================================

In Java, the throw and throws keywords are both used in exception handling, but they serve different purposes:

1. **throw**:
   * The throw keyword is used to **explicitly throw an exception** from a method or a block of code. It is typically used when you want to manually raise an exception.
   * It is followed by an instance of the Throwable class or its subclasses (like Exception or RuntimeException).
   * Example:

java

Copy code

public void checkAge(int age) {

if (age < 18) {

throw new IllegalArgumentException("Age must be 18 or above.");

}

}

1. **throws**:
   * The throws keyword is used in a method declaration to indicate that the method **might throw one or more exceptions**. It is a way of informing the caller of the method that they should be prepared to handle the specified exceptions.
   * It is followed by a list of exception types that the method might throw.
   * Example:

java

Copy code

public void readFile(String fileName) throws IOException {

// Code that might throw an IOException

}

**Summary:**

* throw is used to actually throw an exception.
* throws is used to declare that a method might throw exceptions, which must be handled by the method caller.

The equals and hashCode contract in Java is fundamental when dealing with objects, particularly in collections like HashMap, HashSet, and Hashtable. An analogy can help clarify this concept:

**Analogy: Identifying Books in a Library**

Imagine you are in a library with thousands of books. Each book has two important identifiers:

1. **Content (Equals):** This is the actual content of the book, including its title, author, and text. Two books are considered the same if their content is identical.
2. **Barcode (HashCode):** This is a unique code printed on each book that allows the library to quickly find and catalog the book. This barcode is generated based on the book’s content.

**Equals Method (Content Check)**

The equals method is like checking the content of two books to see if they are the same. If two books have the same title, author, and text, they are considered equal, regardless of their barcode.

* **Analogy:** Imagine you are comparing two books to see if they are identical. You carefully read through each one and determine they are exactly the same, even though their barcodes might differ.
* **In Java:** When you override the equals method, you define the logic that determines when two objects are considered equal based on their content.

**HashCode Method (Barcode Check)**

The hashCode method is like the barcode on a book. The barcode is generated based on the book’s content, so two identical books should ideally have the same barcode.

* **Analogy:** In the library, if two books have the same barcode, you assume they are the same book and treat them as such.
* **In Java:** When you override the hashCode method, you provide a way to generate an integer (barcode) that represents the object’s content. This allows collections like HashMap and HashSet to quickly locate objects.

**The Contract**

The equals and hashCode contract states:

1. **Consistent Equals and HashCode:** If two objects are equal (based on equals), they must have the same hashCode. In the library, if two books have the same content, they should have the same barcode.
2. **Different HashCodes for Different Objects (Not Always Required but Ideal):** If two objects have different content, their hashCode values should ideally be different. However, different books might still end up with the same barcode by coincidence (a hash collision), but this is rare and should be minimized.

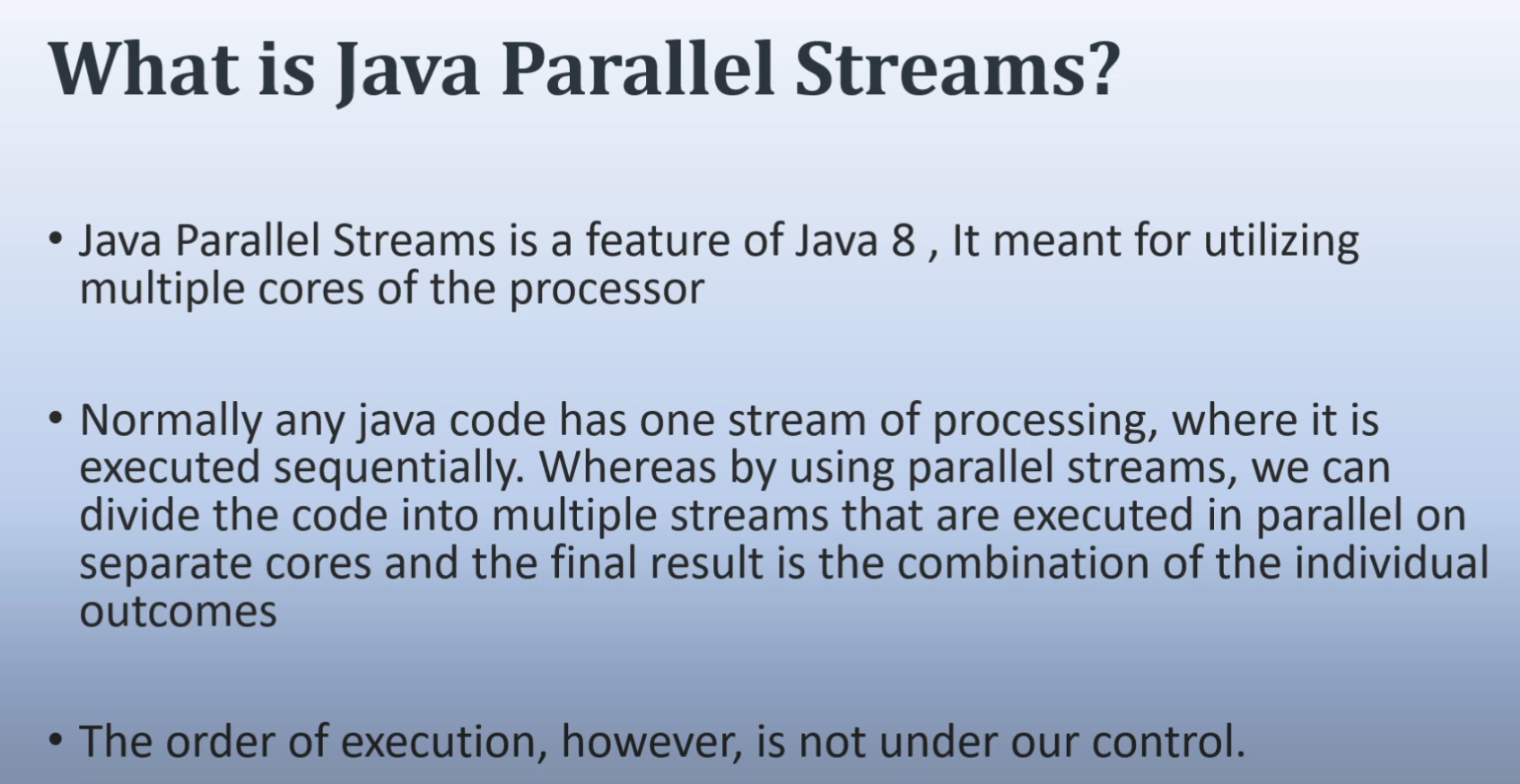
**Violation of the Contract**

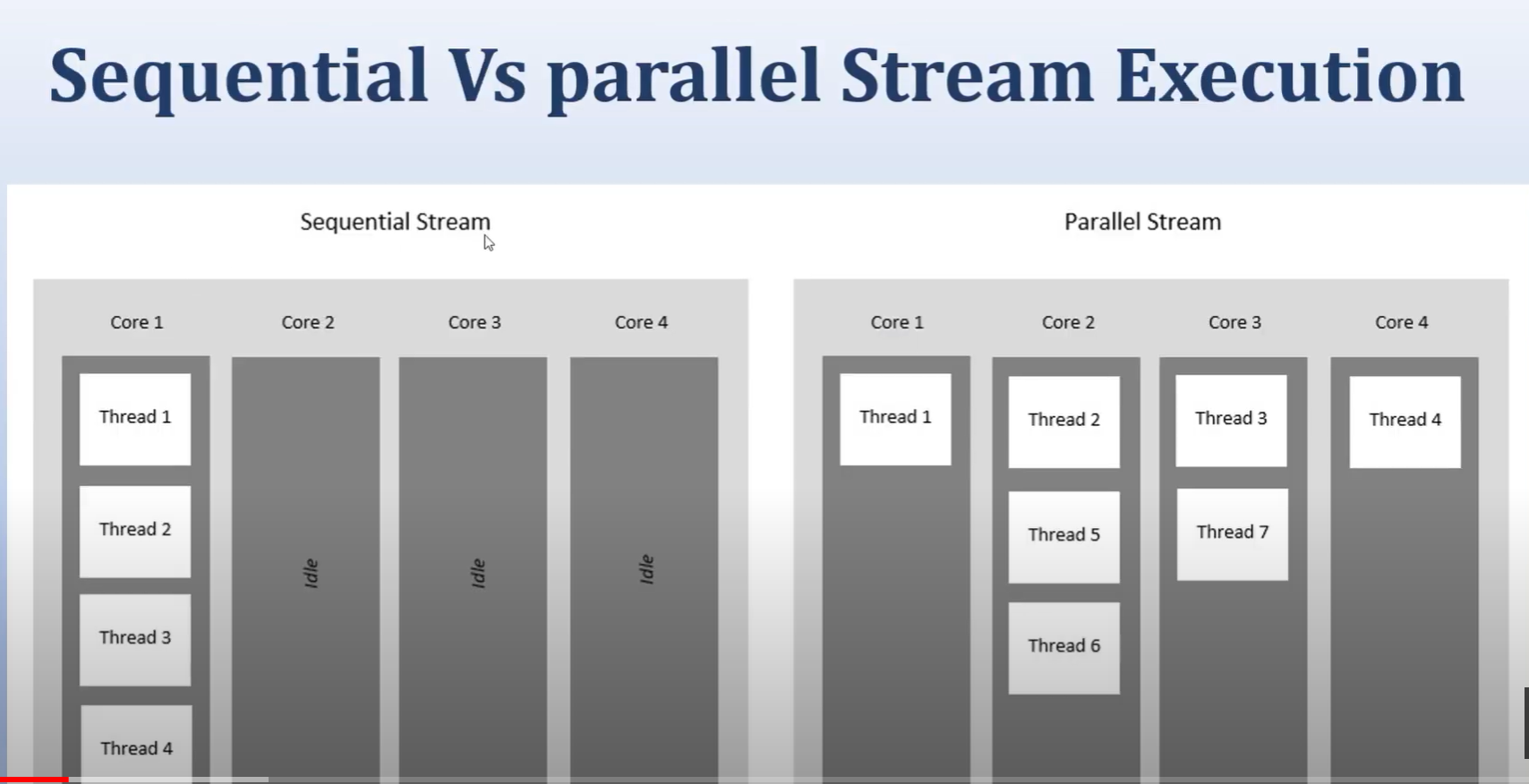
* **Same Content, Different Barcodes:** If two books have the same content but different barcodes, the library might store them in different places, causing confusion. In Java, if two equal objects have different hash codes, it can cause issues in hash-based collections like HashMap.
* **Different Content, Same Barcode:** If two different books have the same barcode, the library might mistakenly treat them as the same book. In Java, this is a hash collision, and while it’s not a violation of the contract, it can degrade performance.

**Conclusion**

In Java, following the equals and hashCode contract is crucial for ensuring that objects behave correctly in collections that rely on hashing. Just like in a library, where consistent content (equals) and barcode (hashCode) management is key to efficient operation, in Java, adhering to this contract is essential for avoiding bugs and ensuring efficient data retrieval.

**How does parallel stream work in java?**





Parallel streams in Java are part of the Stream API, introduced in Java 8. They enable parallel processing of data streams, allowing operations on the stream to be executed concurrently, leveraging multi-core processors. Here's how they work internally:

**1. Splitting the Data**

* **Spliterator:** A parallel stream internally uses a Spliterator, which is a special kind of iterator that supports parallelism. The Spliterator recursively splits the data source into smaller parts, which can be processed independently. This splitting happens until the data chunks are small enough to be processed efficiently by individual threads.

**2. Creating the ForkJoinPool**

* **ForkJoinPool:** Parallel streams use a ForkJoinPool under the hood, which is a special kind of thread pool designed for parallel decomposition and processing tasks. The default pool used is the common ForkJoinPool, which has a number of threads equal to the number of available processors (Runtime.getRuntime().availableProcessors()).

**3. Distributing the Tasks**

* Once the data is split, each chunk is assigned to a separate thread from the ForkJoinPool. The threads work on their assigned tasks concurrently. The framework uses the *work-stealing* algorithm to dynamically balance the workload. If a thread finishes its task and others are still working, it will "steal" work from them, ensuring efficient utilization of resources.

**4. Processing the Data**

* The operations defined on the stream (like map, filter, reduce, etc.) are applied to the data chunks in parallel. Intermediate operations are lazily evaluated, just like in a sequential stream, but now they are done in parallel.

**5. Merging the Results**

* After all chunks are processed, the results are merged back together. This happens automatically in the case of reduction operations, where the partial results from different threads are combined.

**6. Final Result**

* The final result of the parallel stream operation is then collected and returned as a single result, similar to how it would be in a sequential stream, but the processing is done in parallel, potentially offering significant performance benefits, especially with large datasets.

The distinct() method in Java 8's Stream API is used to filter out duplicate elements from a stream, ensuring that only unique elements are present in the resulting stream. Here's how distinct() works internally:

**1. HashSet for Uniqueness**

* Internally, distinct() uses a HashSet to track seen elements. As the stream is processed, each element is checked against this set.
* If the element is not already in the set, it is added to the set and passed on to the resulting stream.
* If the element is already in the set, it is considered a duplicate and is not included in the resulting stream.

**2. Stateful Intermediate Operation**

* distinct() is a stateful intermediate operation. This means it needs to remember the state (in this case, the elements it has already seen) across the entire processing of the stream.
* Because of this statefulness, distinct() may have some impact on performance, particularly with large streams or parallel streams, as it needs to maintain and check the HashSet during processing.

**3. Effect on Parallel Streams**

* When used in a parallel stream, distinct() still works correctly, but it may require additional synchronization to manage the shared state of the HashSet across multiple threads. This can introduce some overhead, potentially reducing the benefits of parallelism.
* The order of elements may be affected if the stream is unordered. If the stream is ordered, distinct() will preserve the order of the first occurrence of each element.

**4. Example Usage**

java

Copy code

List<String> words = Arrays.asList("apple", "banana", "apple", "orange", "banana", "grape");

List<String> distinctWords = words.stream()

.distinct()

.collect(Collectors.toList());

System.out.println(distinctWords); // Output: [apple, banana, orange, grape]

In this example, distinct() filters out the duplicate "apple" and "banana", resulting in a list of unique words.

**5. Custom Objects**

* When working with custom objects, the behavior of distinct() depends on the equals() and hashCode() methods of the objects. For distinct() to work correctly, these methods must be properly overridden to reflect what it means for two objects to be considered equal.
* If equals() and hashCode() are not overridden, distinct() may not correctly identify duplicates, as it relies on these methods to determine uniqueness.

**6. Complexity Considerations**

* The time complexity of distinct() is O(n) for a sequential stream, where n is the number of elements in the stream. This is because each element must be checked against the HashSet and possibly added to it. The space complexity is also O(n) due to the storage required for the HashSet.

**Summary**

The distinct() method in Java 8 streams provides a simple and efficient way to filter out duplicate elements by leveraging a HashSet internally. While it works well for most cases, understanding its reliance on equals() and hashCode(), especially in custom objects and parallel streams, is crucial for ensuring correct behavior and performance.

* **Multiple Inheritance:** Allows a class to inherit from multiple parent classes (not directly supported in Java for classes, but possible with interfaces).
* **Multilevel Inheritance:** Involves a chain of inheritance, where one class inherits from another, which in turn inherits from another class, creating a multi-level hierarchy.

**Concurrency** and **parallelism** are two concepts often discussed in the context of computer science and software engineering, particularly when it comes to multitasking and performance optimization. Here are the key differences between them:

**Concurrency**

1. **Definition**:
   * Concurrency is the ability of a system to handle multiple tasks or operations simultaneously by interleaving them. It doesn't necessarily mean that tasks are running at the same time, but rather that the system can switch between tasks quickly to give the appearance of simultaneous execution.
2. **Execution**:
   * In a single-core CPU, concurrent tasks are not actually running at the same time but are rapidly switched between, thanks to the operating system's scheduling.
3. **Use Case**:
   * Concurrency is useful when tasks involve waiting for external resources (e.g., I/O operations, network requests), allowing the CPU to work on other tasks during these wait times.

**Parallelism**

1. **Definition**:
   * Parallelism is the simultaneous execution of multiple tasks or operations. This can only be achieved on a multi-core or multi-processor system where different tasks run on different processors or cores at the same time.
2. **Execution**:
   * Parallel tasks are truly running at the same time on separate CPU cores, taking advantage of hardware capabilities to improve performance.
3. **Use Case**:
   * Parallelism is beneficial for computationally intensive tasks that can be broken down into smaller, independent tasks that can run simultaneously, such as large data processing or scientific computations.

**Key Differences**

* **Concurrency** involves managing multiple tasks by interleaving their execution, often used to improve responsiveness and resource utilization.
* **Parallelism** involves executing multiple tasks at the same time, leveraging multi-core processors to improve computation speed.

In summary, concurrency is about dealing with lots of things at once, while parallelism is about doing lots of things at once.

**Default Methods** and **Static Methods** were introduced in Java 8 to enhance interfaces' flexibility:

1. **Default Methods**:
   * These are methods in an interface that have a default implementation.
   * They allow you to add new methods to interfaces without breaking the existing implementations of those interfaces.
   * A class implementing the interface can either use the default method as-is or override it to provide a specific implementation.

*Example:*

java

Copy code

interface MyInterface {

default void defaultMethod() {

System.out.println("This is a default method.");

}

}

1. **Static Methods**:
   * These are methods in an interface that belong to the interface itself rather than any instance of a class implementing the interface.
   * Static methods can be called directly from the interface without needing an instance of a class that implements it.
   * They are commonly used for utility or helper methods related to the interface.

*Example:*

java

Copy code

interface MyInterface {

static void staticMethod() {

System.out.println("This is a static method.");

}

}

**Common Java Functional Interfaces**

Java’s standard library includes a range of predefined functional interfaces within the java.util.function package. Here are some of the most commonly used ones:

* **Predicate<T>:** Accepts an argument of type T and returns a boolean. Commonly used for filtering data.
  + boolean test(T t)
* **Consumer<T>:** Accepts an argument of type T and returns no result (void). Useful for performing actions on objects.
  + void accept(T t)
* **Function<T, R>:** Accepts an argument of type T and returns a result of type R. This is useful for mapping one value to another.
  + R apply(T t)
* **Supplier<T>:** Represents a supplier of results (of type T) with no input.
  + T get()
* **UnaryOperator<T>:** A specialization of Function where the input and output are of the same type.
  + T apply(T t)
* **BinaryOperator<T>:** Like UnaryOperator but takes two parameters of the same type and returns a result of the same type. Often used for reductions and aggregate functions.
  + T apply(T t1, T t2)

Callable and Runnable are both interfaces in Java used to represent tasks that can be executed by a thread, but they have some key differences in terms of functionality:

**1. Return Type**

* **Runnable**: The Runnable interface does not return a result. Its run() method has a void return type, meaning it cannot return a value when the thread completes its execution.

java

Copy code

@FunctionalInterface

public interface Runnable {

void run();

}

Example:

java

Copy code

Runnable task = () -> System.out.println("Task executed");

new Thread(task).start();

* **Callable**: The Callable interface is designed to return a result. Its call() method returns a value of type V, which can be retrieved after the thread completes its execution.

java

Copy code

@FunctionalInterface

public interface Callable<V> {

V call() throws Exception;

}

Example:

java

Copy code

Callable<String> task = () -> "Task completed";

String result = task.call();

System.out.println(result); // Task completed

**2. Exception Handling**

* **Runnable**: The run() method of Runnable cannot throw a checked exception. If you need to handle exceptions, you must do so within the run() method itself.

java

Copy code

Runnable task = () -> {

try {

// Task code

} catch (Exception e) {

e.printStackTrace();

}

};

* **Callable**: The call() method of Callable can throw checked exceptions, allowing you to propagate exceptions that occur during the execution of the task.

java

Copy code

Callable<String> task = () -> {

if (someConditionFails) {

throw new Exception("Something went wrong");

}

return "Task completed";

};

**3. Usage with Executors**

* **Runnable**: Typically used with the Thread class or the ExecutorService for executing tasks. When using ExecutorService, the Runnable tasks can be submitted via submit() or execute(). However, if you submit a Runnable to an ExecutorService, you won't be able to get a return value directly.

java

Copy code

ExecutorService executor = Executors.newFixedThreadPool(2);

executor.submit(() -> System.out.println("Runnable task executed"));

executor.shutdown();

* **Callable**: Commonly used with the ExecutorService and can be submitted via the submit() method, which returns a Future object. This allows you to retrieve the result of the computation or check if the task has completed.

java

Copy code

ExecutorService executor = Executors.newFixedThreadPool(2);

Future<String> future = executor.submit(() -> "Callable task executed");

String result = future.get(); // This blocks until the result is available

System.out.println(result); // Callable task executed

executor.shutdown();

**4. Functional Interface**

* **Runnable**: Being a functional interface, Runnable has only one method run(), making it compatible with lambda expressions and method references.
* **Callable**: Similarly, Callable is also a functional interface with a single method call(), which can also be used with lambda expressions.

**Summary of Differences**

| **Feature** | **Runnable** | **Callable** |
| --- | --- | --- |
| Return Type | void | Generic type V |
| Method Name | run() | call() |
| Exception Handling | Cannot throw checked exceptions | Can throw checked exceptions |
| Usage with Executors | submit() or execute() | submit() with Future |

**When to Use Which**

* **Use Runnable** when you don't need to return a result or throw a checked exception from the task.
* **Use Callable** when you need to return a result or may need to throw a checked exception during the execution of the task.

Functional interfaces in Java are interfaces that are designed to have exactly one abstract method, often referred to as a **Single Abstract Method** (SAM). This is a fundamental requirement for them to be used with lambda expressions and method references in functional programming. Here's why functional interfaces can have only one SAM:

**1. Lambda Expression Compatibility**

Lambda expressions in Java are a way to provide a concise implementation of a functional interface. The syntax and behavior of a lambda expression are directly tied to the presence of a single abstract method in the interface.

* **Syntax Simplicity**: If a functional interface had more than one abstract method, the lambda expression would not be able to unambiguously determine which method it is implementing. By having only one abstract method, the lambda expression can be directly mapped to that method.

**Example:**

java

Copy code

@FunctionalInterface

interface MyFunction {

int apply(int x);

}

MyFunction square = x -> x \* x;

System.out.println(square.apply(5)); // 25

In this case, the lambda expression x -> x \* x clearly maps to the single abstract method apply(int x).

**2. Type Inference**

Type inference in lambda expressions relies on the target type, which is determined by the functional interface's single abstract method. The Java compiler uses the SAM to infer the types of the parameters and return values in the lambda expression.

* **Ambiguity Avoidance**: If an interface had more than one abstract method, the compiler would be unable to infer the correct method to implement, leading to ambiguity and potential errors.

**Example of Ambiguity:**

java

Copy code

interface InvalidFunction {

int apply(int x);

int subtract(int y);

}

// This would be ambiguous because the lambda could match either method

// InvalidFunction func = x -> x \* x;

The ambiguity above demonstrates why an interface with multiple abstract methods cannot be used as a functional interface.

**3. Functional Programming Principle**

In functional programming, functions are first-class citizens and are often passed around as arguments, returned from other functions, or assigned to variables. A functional interface represents such a function. By having only one abstract method, a functional interface effectively represents a single action or behavior, aligning with the principles of functional programming.

* **Single Responsibility**: Each functional interface should represent a single responsibility or action, which is why it should have only one abstract method.

**4. @FunctionalInterface Annotation**

Java provides the @FunctionalInterface annotation to enforce the SAM rule. This annotation is optional but when used, it tells the compiler to ensure that the interface contains exactly one abstract method. If you try to add more than one abstract method to an interface annotated with @FunctionalInterface, the compiler will throw an error.

* **Code Safety and Clarity**: The @FunctionalInterface annotation provides a way to ensure that an interface is indeed a functional interface, preventing accidental addition of extra abstract methods.

java

Copy code

@FunctionalInterface

interface MyFunction {

int apply(int x);

// Uncommenting the below line would cause a compilation error

// int anotherMethod(int y);

}

**5. Default and Static Methods**

While a functional interface can have only one abstract method, it can still have multiple **default** and **static** methods. These methods are not abstract and provide a concrete implementation, so they do not violate the SAM rule.

* **Enhancements**: Default and static methods allow functional interfaces to be extended with additional methods without breaking their functional nature or compatibility with lambda expressions.

java

Copy code

@FunctionalInterface

interface MyFunction {

int apply(int x);

// Default method with implementation

default int addTen(int x) {

return x + 10;

}

// Static method with implementation

static int multiply(int x, int y) {

return x \* y;

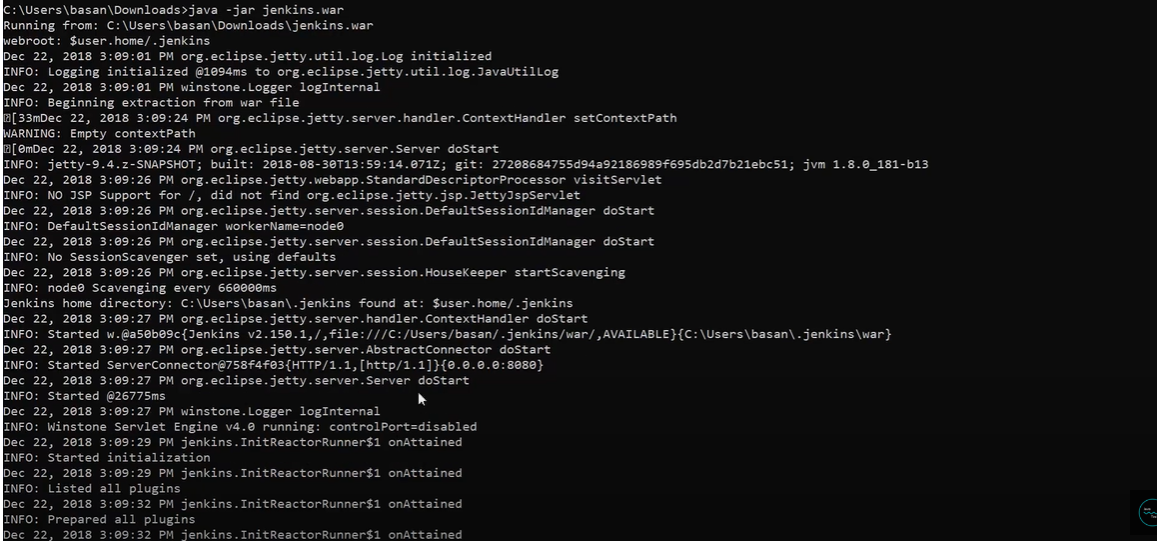
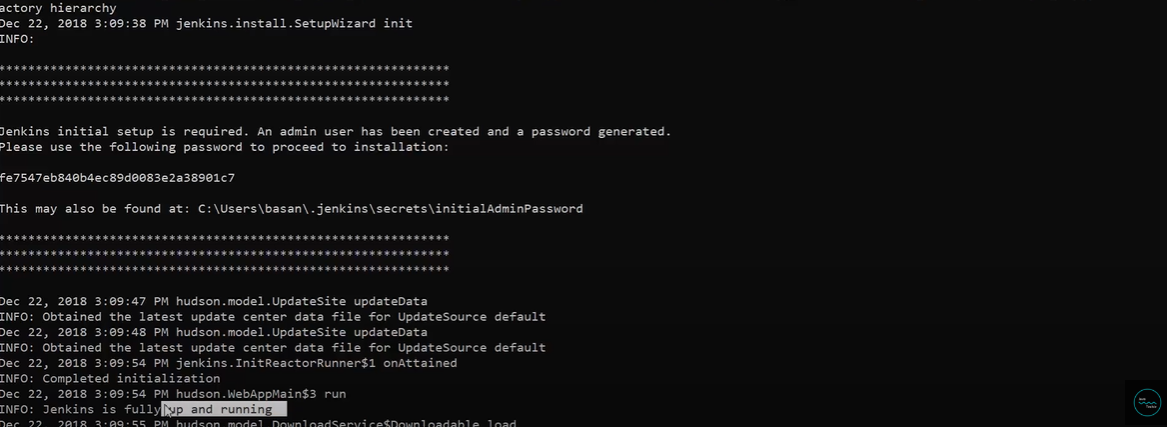
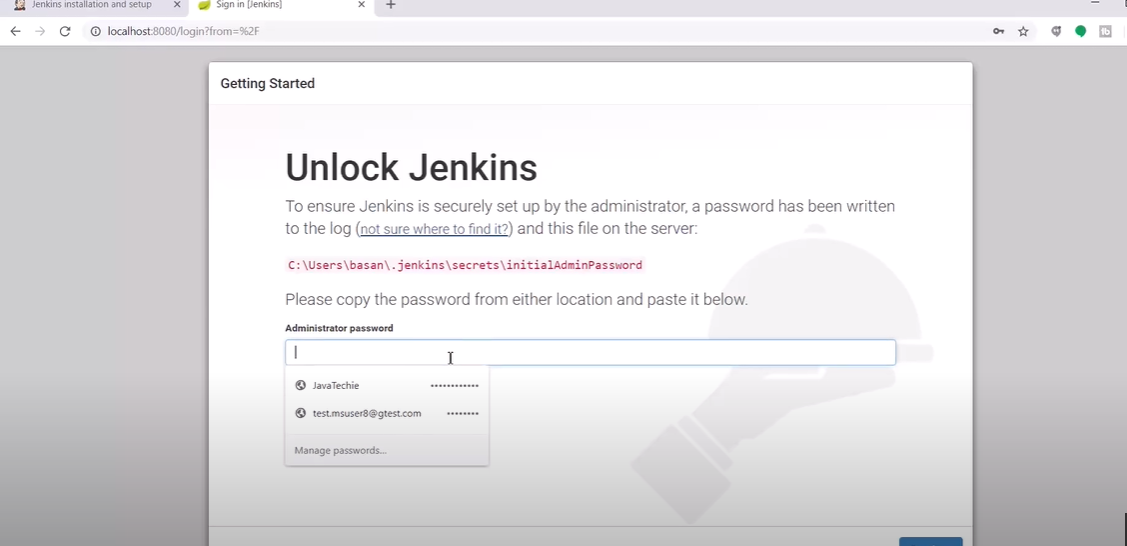
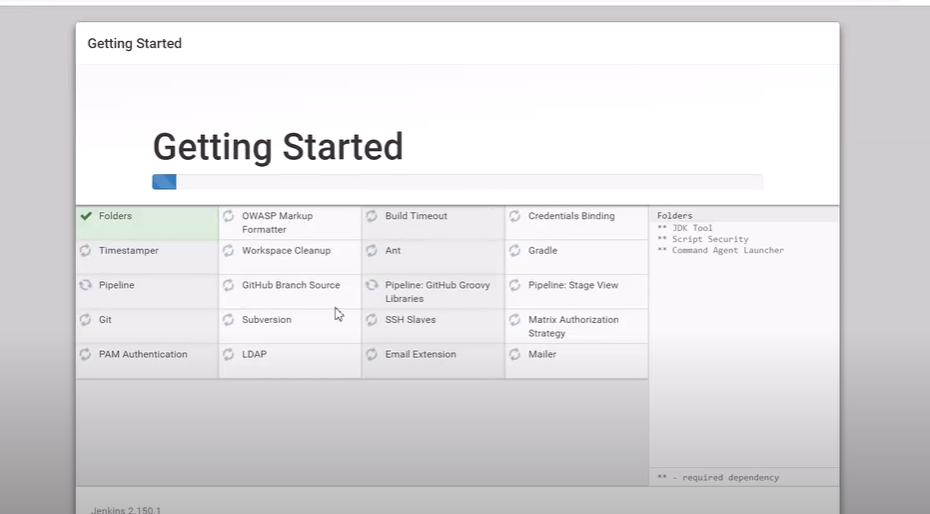
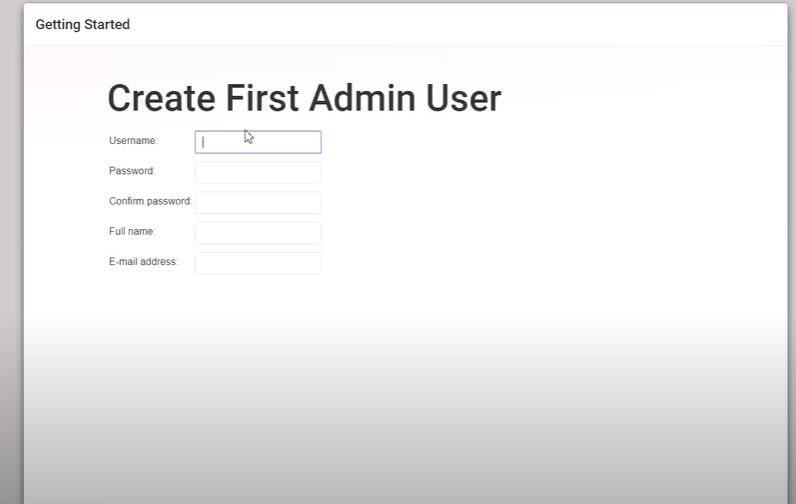
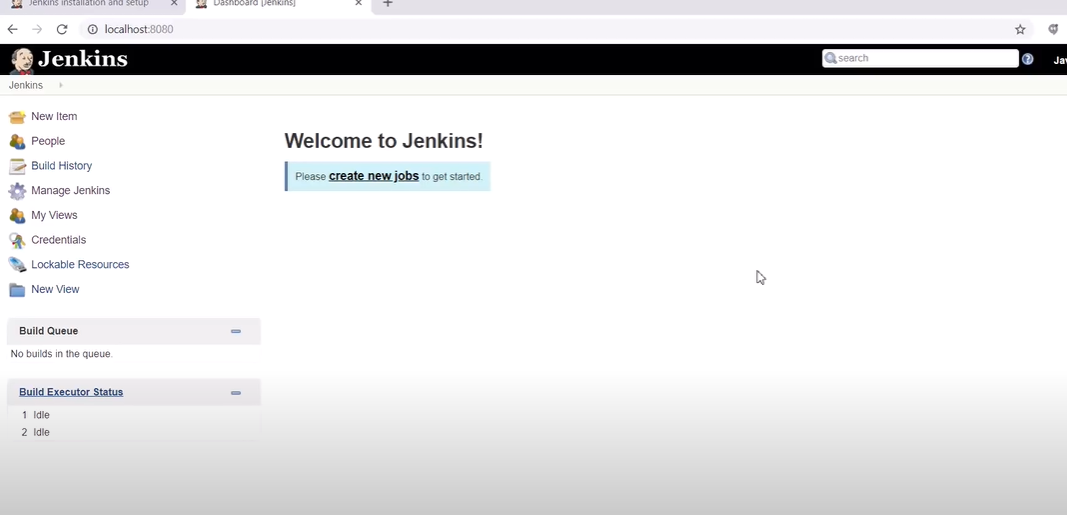
}

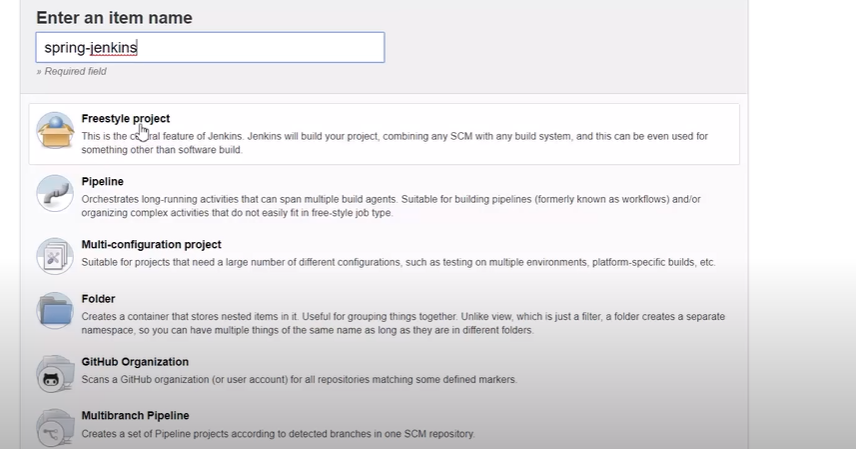
}

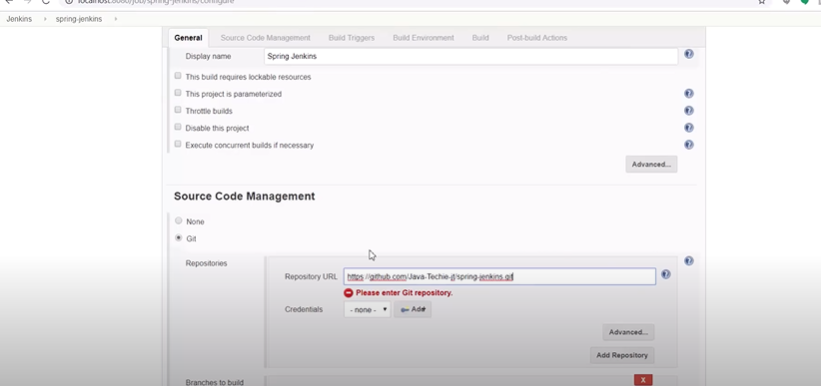
**Conclusion**

The restriction of having only one abstract method in a functional interface ensures that lambda expressions and method references can be used unambiguously and effectively. This design aligns with the principles of functional programming and allows for clean, concise, and type-safe code. The @FunctionalInterface annotation further reinforces this design, ensuring that developers adhere to the intended use of these interfaces.

Setting up CI/CD Jenkins Pipeline.

* Download and Install Jenkins pipeline.
* Open CMD on the downloaded path and run java -jar Jenkins.war
* 
* 
* 
* Install suggested plugins.
* 
* 
* 
* **Continuous Integration with github**
* Create a new job and mention the job name



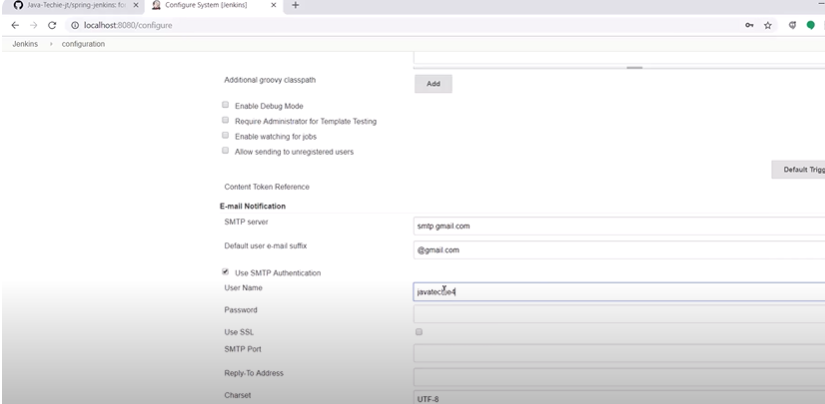
* Choose freestyle project and click ok.
* Choose GitHub project and give project URL(git hub URL)
* In source Code management choose git and give repository URL
* 
* Give Jenkins credentials
* Poll SCM… schedule \*\*\*\*\* cron job expression.
* Add build step🡪 invoke top level maven targets🡪 mvn clean install
* Post build actions 🡪 Email notifications🡪 add recipients

Got to Jenkins and click build now 🡪 we can check the console output.

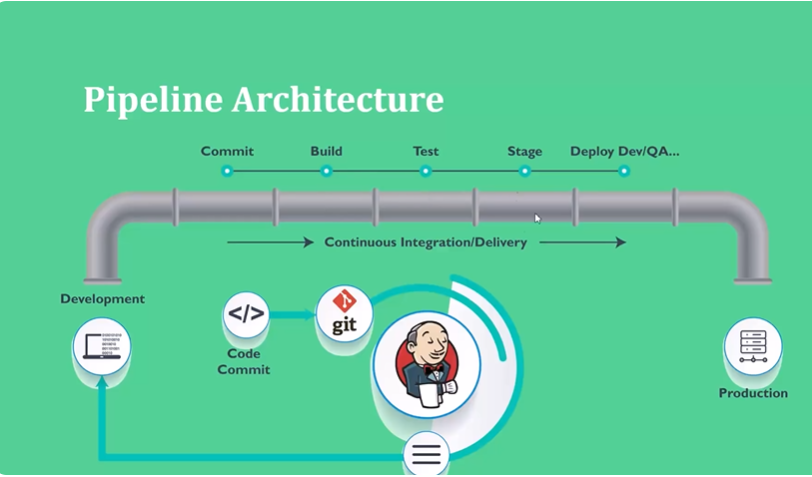


Manage Jenkins🡪 configure system🡪 EmailNotification🡪smtp.gmail.com

User e-mail suffix🡪 @gmail.com



Click on Apply and save it.



Build plugin pipeline , Declarative pipeline, scripted pipeline.

If App have less job to execute then go for build plugin pipeline

If your app have multiple job like development, code review, unit test, coverageTest, LoadTest, Integration Test, Packaging, Build Etc.,, then go for declarative or scripted pipeline.  
  
**Building Docker Images using Jenkins step by step| DevOps Integration**

* Click on manage plugins🡪cloudBees Docker Build and Publish plugin.
* Add Build step 🡪 build publish docker image or docker build and publish
* In the docker build and publish give the repository name: Tag, docker registry credentials

When an IAM (Identity and Access Management) role is assumed by a service in AWS, several steps occur behind the scenes to ensure that the service receives the appropriate permissions to perform actions on AWS resources. Here's a breakdown of what happens:

**1. Role Creation and Trust Policy**

* **IAM Role Creation:** An IAM role is created with a defined set of permissions. These permissions are specified in the role's policy, which dictates what actions the role can perform on which resources.
* **Trust Policy:** Along with the permissions policy, a trust policy is attached to the role. The trust policy specifies which entities (like AWS services, users, or other roles) are allowed to assume the role.

**2. Role Assumption Request**

* **Service Request:** When a service (like an EC2 instance, Lambda function, or another AWS service) needs to perform actions using an IAM role, it makes an API call to the sts:AssumeRole action.
* **Credentials Exchange:** AWS Security Token Service (STS) processes this request. The service must be authorized to assume the role based on the trust policy.

**3. Temporary Security Credentials**

* **Temporary Credentials Issuance:** If the request is authorized, STS generates temporary security credentials for the service. These credentials consist of:
  + An **Access Key ID**.
  + A **Secret Access Key**.
  + A **Session Token**.
* **Session Duration:** These credentials are valid for a limited duration, typically up to one hour by default, but it can be configured to last longer depending on the role's settings.

**4. Service Usage of Temporary Credentials**

* **API Requests:** The service uses these temporary credentials to make API calls to other AWS services. Each API request includes the Access Key ID, Secret Access Key, and Session Token to authenticate and authorize the request.
* **Permissions Check:** AWS verifies the permissions associated with the role before allowing the service to perform the requested action.

**5. Action Execution**

* **Execution of Requests:** The service can now perform the actions defined in the IAM role's permissions policy, such as accessing S3 buckets, launching EC2 instances, or invoking other AWS services.

**6. Credentials Expiry**

* **Expiration:** Once the temporary credentials expire, the service must assume the role again to get a new set of credentials if it needs to continue making API requests.

**7. Monitoring and Logging**

* **CloudTrail Logging:** AWS CloudTrail logs the role assumption event, including details about the entity that assumed the role, the actions performed, and the time of the assumption. This is crucial for auditing and monitoring purposes.

**Example Scenario: EC2 Instance Assuming a Role**

If an EC2 instance is configured to assume an IAM role, here's how it would work:

1. The EC2 instance has a role attached during launch or through an instance profile.
2. When the instance needs to interact with AWS services (e.g., access an S3 bucket), it uses the IAM role's temporary credentials.
3. The role's trust policy allows the EC2 service to assume the role.
4. STS provides temporary credentials, which the instance uses to make authorized requests to S3.

This mechanism ensures that the service can operate securely with the least privilege necessary, reducing the risk of unauthorized access or misuse of resources.

In Java, both Comparable and Comparator are used for sorting objects, but they serve different purposes and are used in different scenarios.

**Comparable Interface:**

* **Usage:** Use Comparable when you want to define a natural ordering for the objects of a class. This is typically used when the class itself has a clear, single way to compare its instances (e.g., alphabetical order for strings, numerical order for integers).
* **Implementation:** A class that implements Comparable must override the compareTo() method. The compareTo() method defines how objects of the class should be compared to one another.
* **Example:**

java

Copy code

public class Employee implements Comparable<Employee> {

private int id;

private String name;

@Override

public int compareTo(Employee other) {

return Integer.compare(this.id, other.id); // Sort by id

}

}

* **Limitations:** A class can only have one compareTo method, which means it can only be sorted in one way using Comparable.

**Comparator Interface:**

* **Usage:** Use Comparator when you need to sort objects in a way that might differ from their natural order, or if the class does not implement Comparable. It is also useful when you want to sort the same objects in different ways.
* **Implementation:** A class that implements Comparator must override the compare() method. This method compares two objects of the class and returns a comparison result.
* **Example:**

java

Copy code

public class EmployeeByNameComparator implements Comparator<Employee> {

@Override

public int compare(Employee e1, Employee e2) {

return e1.getName().compareTo(e2.getName()); // Sort by name

}

}

* **Flexibility:** You can create multiple Comparator implementations to sort objects in different ways (e.g., by name, by age, by salary, etc.).

**When to Use:**

* **Use Comparable:**
  + When a class has a natural default ordering.
  + When you expect the objects to be sorted primarily in that natural order across your application.
* **Use Comparator:**
  + When you need multiple ways to sort objects.
  + When the class doesn’t have a natural ordering.
  + When you don't want to modify the class itself to implement Comparable.

**Example Use Case:**

Suppose you have an Employee class, and the natural order should be by id. However, you also need to sort employees by name or salary in some parts of your application. You would:

* Implement Comparable to define the natural order by id.
* Implement different Comparator classes to handle sorting by name or salary.

Understanding how Comparable and Comparator work internally in Java requires diving into how sorting algorithms like Arrays.sort() or Collections.sort() utilize these interfaces.

**1. Comparable Interface**

When a class implements Comparable, it provides a natural ordering of its instances. The compareTo() method is the key to this ordering.

**How it works internally:**

* **compareTo() Method:**
  + When you implement Comparable, you must override the compareTo() method. This method compares the current object with the specified object and returns:
    - A negative integer if the current object is less than the specified object.
    - Zero if the current object is equal to the specified object.
    - A positive integer if the current object is greater than the specified object.
  + For example, if sorting by ID:

java

Copy code

public class Employee implements Comparable<Employee> {

private int id;

@Override

public int compareTo(Employee other) {

return Integer.compare(this.id, other.id);

}

}

* **Sorting Mechanism:**
  + When you call Collections.sort() or Arrays.sort() on a list or array of objects that implement Comparable, the sorting algorithm (like Merge Sort or TimSort) internally calls the compareTo() method to determine the order of objects.
  + The sorting algorithm will compare pairs of objects using compareTo() and arrange them based on the returned value.

**2. Comparator Interface**

The Comparator interface provides a way to define multiple custom orders for objects without modifying the object's class. You can create multiple comparators for different sorting criteria.

**How it works internally:**

* **compare() Method:**
  + When you implement Comparator, you must override the compare() method. This method compares two objects of the type and returns:
    - A negative integer if the first object is less than the second.
    - Zero if the first object is equal to the second.
    - A positive integer if the first object is greater than the second.
  + For example, if sorting by name:

java

Copy code

public class EmployeeByNameComparator implements Comparator<Employee> {

@Override

public int compare(Employee e1, Employee e2) {

return e1.getName().compareTo(e2.getName());

}

}

* **Sorting Mechanism:**
  + When you call Collections.sort() or Arrays.sort() and pass a Comparator as an argument, the sorting algorithm uses the compare() method of the Comparator to determine the order of the objects.
  + The sorting algorithm works the same way as with Comparable, but instead of using the natural ordering defined in compareTo(), it uses the custom order defined in the compare() method of the Comparator.

**Internal Sorting Algorithms**

* **TimSort:** Java's default sorting algorithm for Collections.sort() and Arrays.sort() is a hybrid sorting algorithm derived from merge sort and insertion sort. TimSort works efficiently on many real-world data sets and uses the compareTo() or compare() methods to determine the order of elements.
* **Efficiency:** TimSort is stable and has a time complexity of O(n log n) in the worst case. The internal working of TimSort involves splitting the array or list into small sections (runs), sorting these sections, and then merging them. During the merge phase, the compareTo() or compare() methods are invoked to decide the order.

**Summary of Internal Flow**

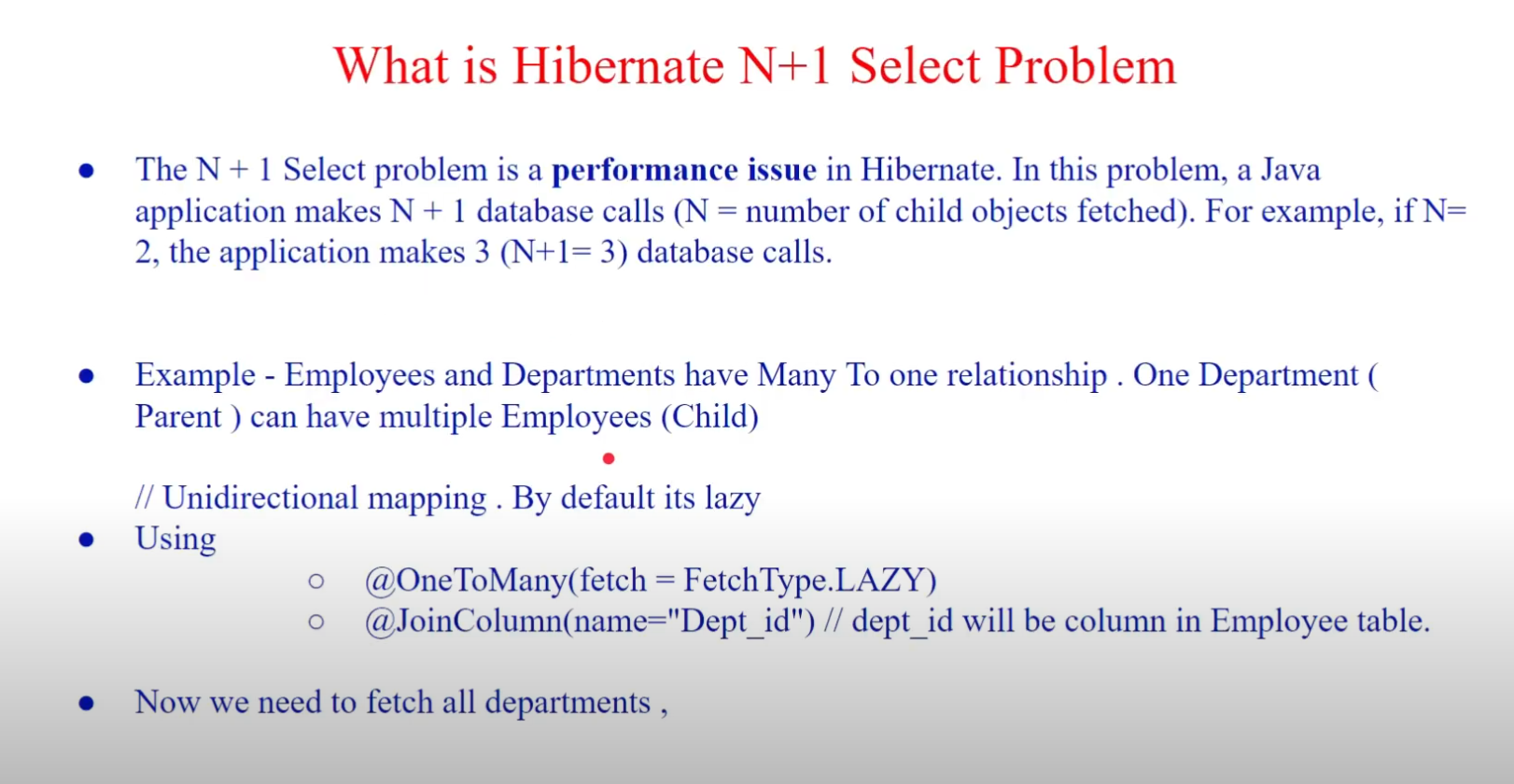
1. **Comparable:**
   * Sorting algorithms call the compareTo() method repeatedly during sorting to decide the order of objects.
   * Objects are compared based on their natural order defined by the compareTo() method.
2. **Comparator:**
   * Sorting algorithms call the compare() method of the provided Comparator object during sorting to decide the order of objects.
   * Objects are compared based on the custom order defined by the compare() method.

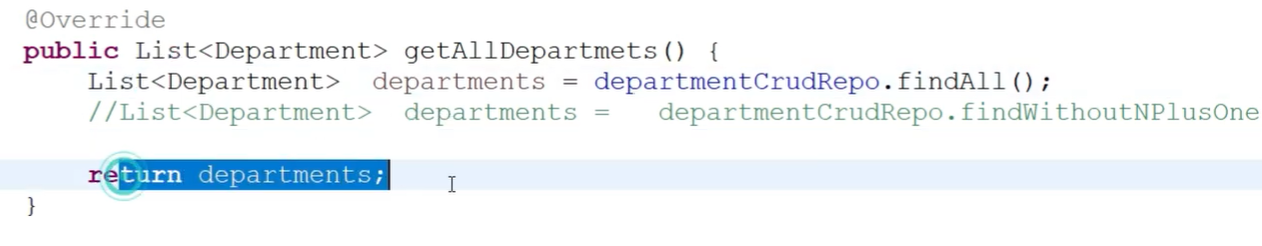
These sorting mechanisms are foundational to many operations in Java, enabling flexible and efficient sorting based on different criteria.

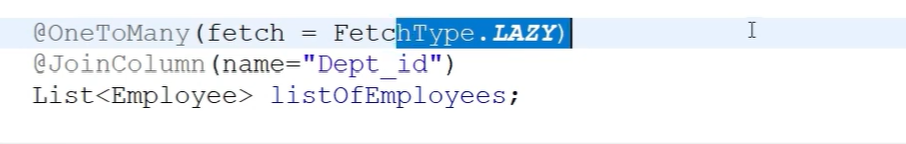
<https://medium.com/@reetesh043/difference-between-completablefuture-and-future-in-java-4f7e00bcdb56#:~:text=Future%3A%20Does%20not%20inherently%20support,and%20variants%20with%20custom%20Executor>.

**N+1 query problem**

Assume there are one to many relationship between employee and department

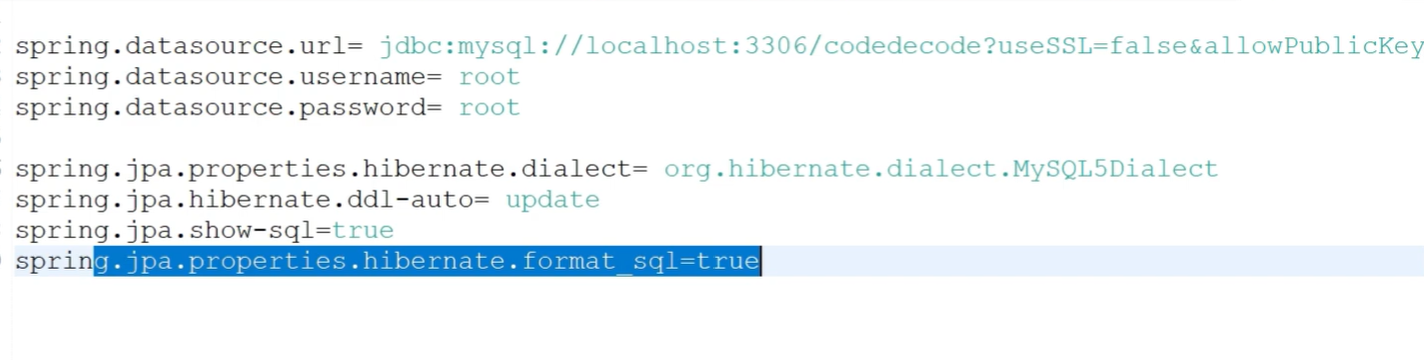


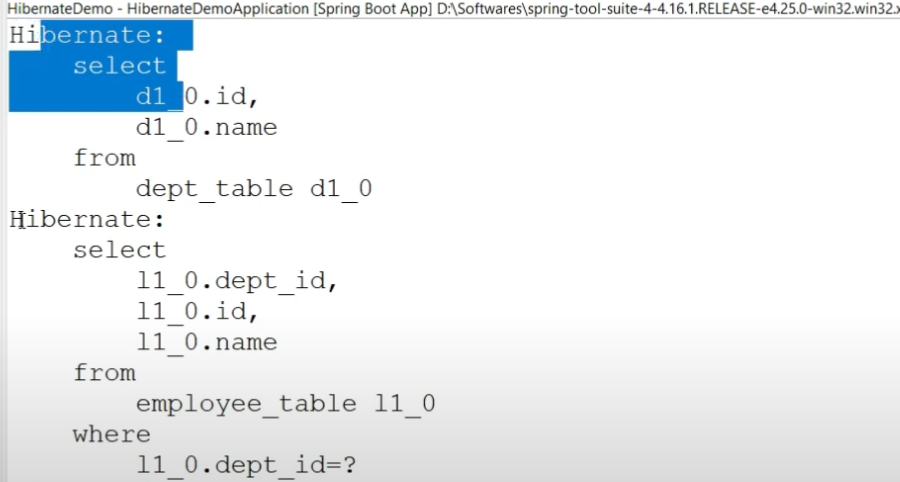


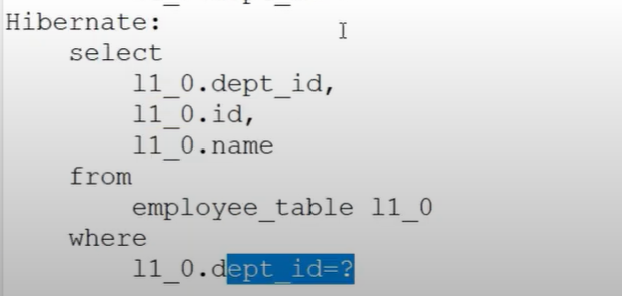


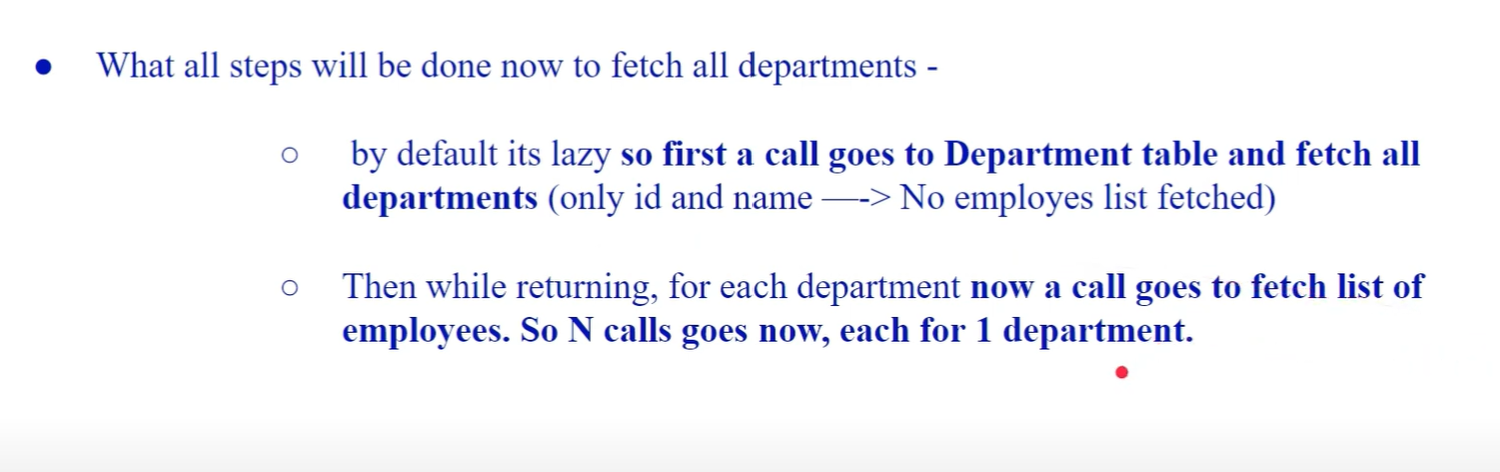
At departmentCrudRepo.findAll() it fetches all the departments without any employees 1 query.

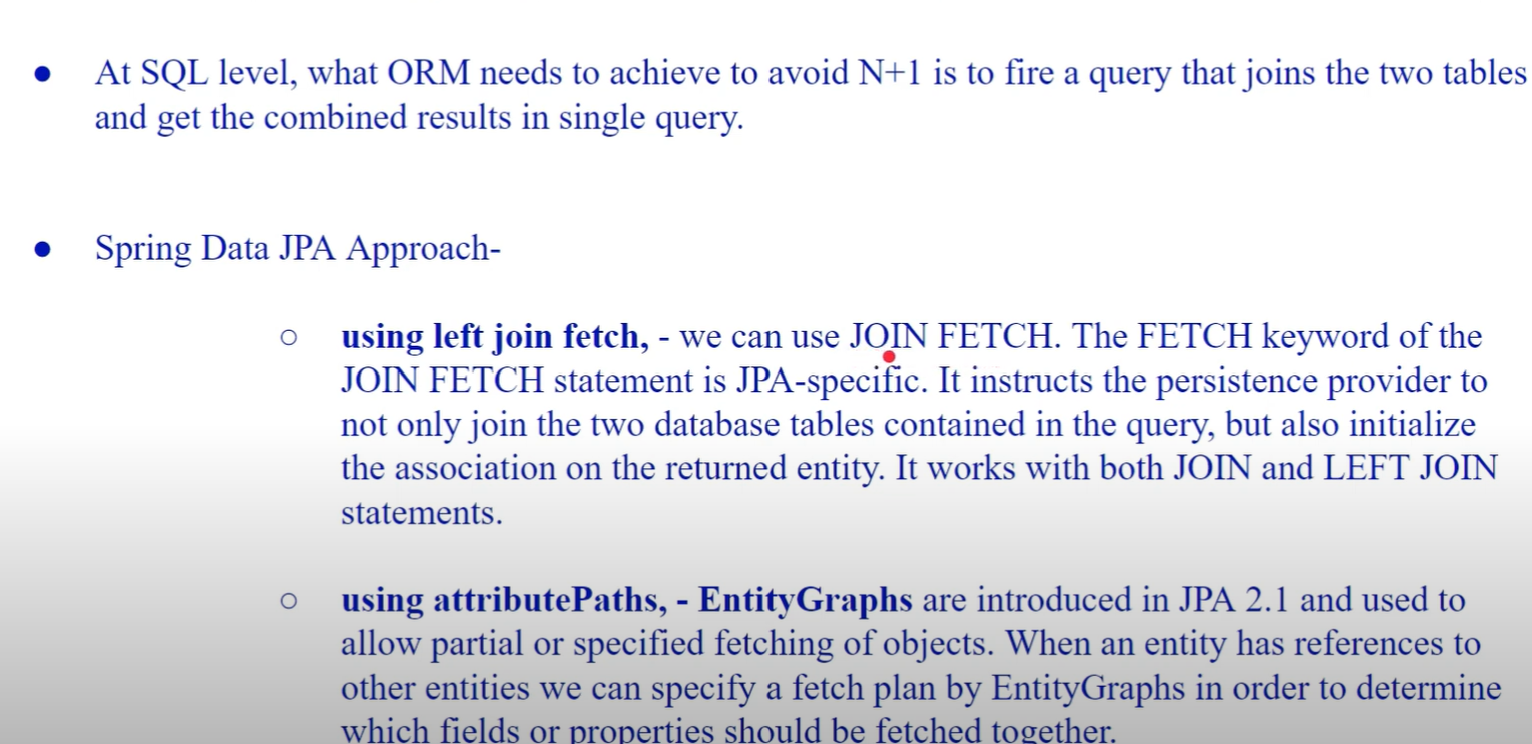
But at line return departments, it will fetch all the employees for the departmentIds fetched which is N queries.

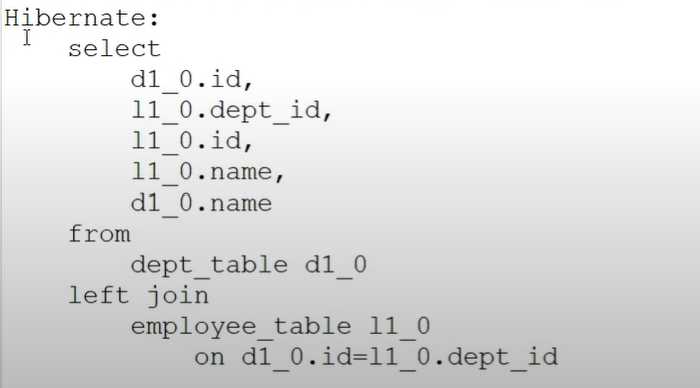


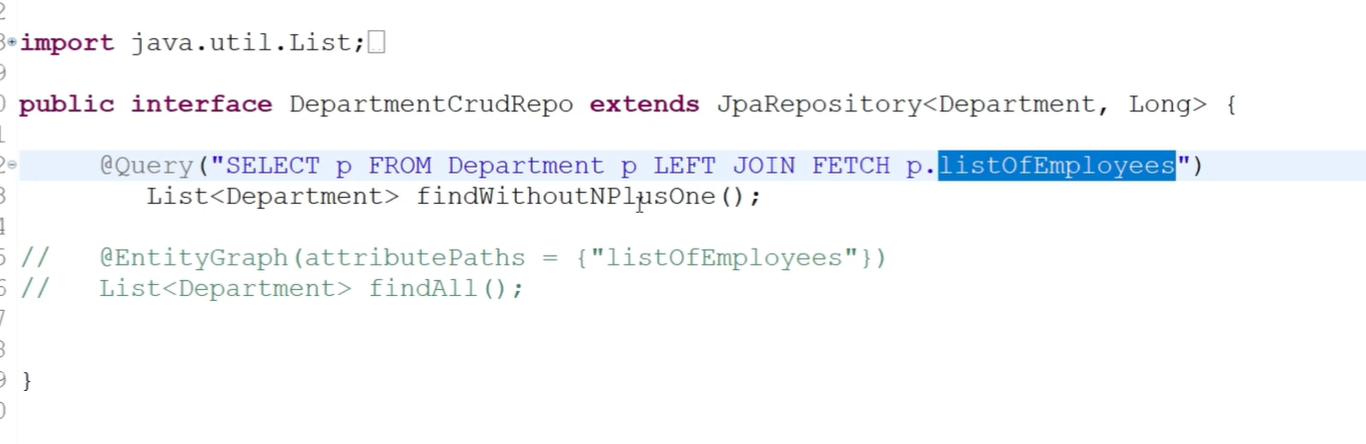












**Method References**

In Java, method references are a shorthand notation of a lambda expression to call a method. They provide a way to pass the reference of a method as an argument to a functional interface. There are four main types of method references:

1. **Reference to a Static Method**  
   This type refers to a static method in a class. The syntax is ClassName::staticMethodName.

**Example:**

java

Copy code

Function<String, Integer> parseInt = Integer::parseInt;

1. **Reference to an Instance Method of a Particular Object**  
   This type refers to an instance method of a particular object. The syntax is instance::instanceMethodName.

**Example:**

java

Copy code

String str = "Hello";

Supplier<Integer> lengthSupplier = str::length;

1. **Reference to an Instance Method of an Arbitrary Object of a Particular Type**  
   This type refers to an instance method of an arbitrary object of a particular type. The syntax is ClassName::instanceMethodName.

**Example:**

java

Copy code

BiFunction<String, String, Boolean> equalsFunction = String::equalsIgnoreCase;

1. **Reference to a Constructor**  
   This type refers to a constructor. The syntax is ClassName::new.

**Example:**

java

Copy code

Supplier<List<String>> listSupplier = ArrayList::new;

These method references make the code more concise and readable, especially when a lambda expression simply calls an existing method.

The **diamond problem** is a common problem in multiple inheritance where a class inherits from two classes that both inherit from a common superclass, leading to ambiguity. Java addresses the diamond problem by avoiding multiple inheritance of classes but allows multiple inheritance through interfaces with specific rules.

**Diamond Problem Explanation:**

Consider this example in a language that supports multiple inheritance:

plaintext

Copy code

A

/ \

B C

\ /

D

* Class B and class C both inherit from class A.
* Class D inherits from both B and C.

If both B and C have overridden a method from A, then class D may have ambiguity about which method to inherit—B's or C's.

**Java's Approach:**

Java avoids the diamond problem with the following design decisions:

1. **No Multiple Inheritance of Classes**:  
   Java does not allow a class to inherit from more than one class. This means the diamond problem does not occur with classes in Java.

java

Copy code

class A {}

class B extends A {}

class C extends A {}

// class D extends B, C {} // Not allowed in Java

1. **Multiple Inheritance with Interfaces**:  
   Java allows a class to implement multiple interfaces. If these interfaces have methods with the same signature, Java uses specific rules to resolve conflicts.

**Example with Interfaces:**

java

Copy code

interface A {

void doSomething();

}

interface B extends A {

default void doSomething() {

System.out.println("B's implementation");

}

}

interface C extends A {

default void doSomething() {

System.out.println("C's implementation");

}

}

class D implements B, C {

@Override

public void doSomething() {

// Resolve the conflict explicitly

B.super.doSomething(); // or C.super.doSomething();

}

}

In the example above, class D implements both B and C. Since both B and C have default implementations of doSomething(), Java requires class D to explicitly choose which method to use, avoiding the ambiguity that would occur in a language with unrestricted multiple inheritance.

**Key Points:**

* Java avoids the diamond problem with classes by disallowing multiple inheritance.
* Java supports multiple inheritance with interfaces but requires explicit resolution of conflicts when there are multiple default methods with the same signature.
* You can resolve these conflicts using InterfaceName.super.methodName() in the implementing class.

**Loose coupling** and **high cohesion** are fundamental design principles in software engineering that contribute to creating well-structured, maintainable, and flexible code. They are often discussed together because they complement each other in promoting good software design.

**Loose Coupling**

**Loose coupling** refers to the degree of independence between software modules. When modules are loosely coupled, changes in one module have minimal impact on others. This makes the system more flexible, easier to maintain, and less prone to bugs when changes are made.

* **Low Dependency**: Modules don't rely heavily on the internal details of other modules. Instead, they interact through well-defined interfaces.
* **Easier Maintenance**: Changes in one module, like bug fixes or updates, require minimal or no changes in other modules.
* **Improved Reusability**: Loosely coupled modules can be reused in different contexts or applications with little or no modification.

**Example:**

* Consider two classes: OrderService and PaymentService. If OrderService directly interacts with PaymentService through its concrete implementation, they are tightly coupled. If you later decide to switch to a different payment service, you’ll have to modify the OrderService class.
* By using an interface, PaymentProcessor, that both PaymentService and any other payment services implement, OrderService becomes loosely coupled. It interacts with PaymentProcessor without needing to know the details of the specific payment service.

**High Cohesion**

**High cohesion** refers to how closely related and focused the responsibilities of a single module (or class) are. A highly cohesive module has a well-defined purpose and contains only functionality that is strongly related to that purpose.

* **Single Responsibility**: A module or class should focus on a single task or a group of related tasks.
* **Better Understandability**: Highly cohesive modules are easier to understand, test, and maintain because they encapsulate related behaviors.
* **Improved Reusability**: Since cohesive modules have clear, focused responsibilities, they can be reused in different parts of the system without unnecessary dependencies.

**Example:**

* A class that handles both order processing and user authentication is low in cohesion because it has multiple unrelated responsibilities. If you split this into two classes—OrderProcessor and UserAuthenticator—each class now has a single, well-defined responsibility, leading to high cohesion.

**Combining Loose Coupling and High Cohesion**

* **Loose Coupling + High Cohesion**: Ideally, software design should aim for both loose coupling and high cohesion. This means each module or class should do one thing well (high cohesion) and interact with other modules through well-defined, minimal interfaces (loose coupling).

**Example in Practice:**

* Imagine a system where UserService handles all user-related operations. If UserService has methods for handling authentication, profile management, and data storage, and it directly interacts with data access objects (DAOs) and third-party services, the service is likely to be tightly coupled and low in cohesion.
* To improve the design, you can separate concerns: create AuthenticationService for authentication, ProfileService for user profiles, and UserRepository for data access. These services interact with each other through interfaces, making the system loosely coupled and each service highly cohesive.

In summary, **loose coupling** makes the system more flexible and easier to maintain, while **high cohesion** ensures that each module or class has a clear, focused purpose, making the system more understandable and easier to work with.

In Java, functional interfaces are interfaces with a single abstract method, designed to be used with lambda expressions or method references. However, functional interfaces can also include default and static methods without losing their status as functional interfaces. These methods provide additional functionality while keeping the interface's primary purpose intact.

**Default Methods in Functional Interfaces**

**Default methods** allow you to add new methods to interfaces without breaking the classes that implement these interfaces. The method is implemented directly in the interface using the default keyword.

* **Purpose**: To provide a default implementation that can be used by all implementing classes. It helps in evolving interfaces by adding new methods without forcing all implementing classes to provide an implementation.
* **Usage**: Implementing classes can either use the default implementation or override it to provide a custom behavior.

**Example:**

java

Copy code

@FunctionalInterface

interface MyFunctionalInterface {

void abstractMethod(); // Single abstract method

default void defaultMethod() {

System.out.println("Default method in functional interface.");

}

}

class MyClass implements MyFunctionalInterface {

@Override

public void abstractMethod() {

System.out.println("Implementation of abstract method.");

}

// Optionally, override the default method

@Override

public void defaultMethod() {

System.out.println("Overridden default method.");

}

}

public class Main {

public static void main(String[] args) {

MyClass obj = new MyClass();

obj.abstractMethod(); // Output: Implementation of abstract method.

obj.defaultMethod(); // Output: Overridden default method.

}

}

**Static Methods in Functional Interfaces**

**Static methods** in interfaces are methods that belong to the interface itself, rather than to any instance of a class that implements the interface. These methods cannot be overridden by implementing classes and are invoked using the interface name.

* **Purpose**: To provide utility methods related to the interface, which do not depend on the state of any instance. These methods are similar to static methods in classes.
* **Usage**: They are called directly using the interface name, making them useful for utility or factory methods.

**Example:**

java

Copy code

@FunctionalInterface

interface MyFunctionalInterface {

void abstractMethod(); // Single abstract method

static void staticMethod() {

System.out.println("Static method in functional interface.");

}

}

public class Main {

public static void main(String[] args) {

// Call the static method using the interface name

MyFunctionalInterface.staticMethod(); // Output: Static method in functional interface.

}

}

**Key Points:**

* **Default Methods**:
  + Allow interfaces to provide a default implementation.
  + Can be overridden by implementing classes.
  + Useful for extending interfaces without breaking existing implementations.
* **Static Methods**:
  + Belong to the interface itself, not to instances of implementing classes.
  + Cannot be overridden.
  + Provide utility methods related to the interface.

**Functional Interface Definition:**

A functional interface can still include multiple default and static methods without violating its "functional" nature, as long as it has exactly one abstract method. The @FunctionalInterface annotation can be used to indicate that the interface is intended to be a functional interface, although it's not required. If the annotation is used, the compiler will enforce the single abstract method rule.

**Example with Both Default and Static Methods:**

java

Copy code

@FunctionalInterface

interface MyFunctionalInterface {

void abstractMethod(); // Single abstract method

default void defaultMethod() {

System.out.println("Default method in functional interface.");

}

static void staticMethod() {

System.out.println("Static method in functional interface.");

}

}

In summary, default and static methods in functional interfaces add flexibility and utility without compromising the functional interface's core purpose.

<https://medium.com/techmonks/12-factor-app-principles-and-cloud-native-microservices-a383f6abc97f>

Top of Form

Bottom of Form