Java By Amit Sir

1. @RestController :- Rest Controller returning JSON/XML
2. @PostMapping :- Shorthand for request method.
3. @GetMapping :- Shorthand for request method.
4. @DeleteMapping :- Shorthand for request method.
5. @PathVariable :- Build URL template Variable.
6. @RequestBody :- Blind http request body (JSON) to object.
7. @Entity :- Mark model class as database entity.
8. @Id :- Primary key and auto ID generation.
9. @Override :- it is used to show this method is override method
10. @Autowired :- Auto-injection dependencies by type
11. @Service :- Business logic layer bean
12. JpaRepository – **Java Persistence API Repository.**
13. Interface inherit with interface that time we used Extends keyword.

Packages :-

Some important packages when we start our spring boot project in STS Ide.

i. **Controller :** Controller is used to create API's.
ii. **Services :** We used services to write a business Logic / logical class.
iii. **Entity :** It is used to create all entities required for the project. An entity is used to create a table, where the entity class name is treated as the table name, and the variables inside the class are considered as column names.
iv. **Repository :** Repository package is used to create Interface class which is extends with JpaRepository, JpaRepositroy content all method which is help to perform postrgres or database operation.
v. **Factory :**Instate of we create the object to the class factory will create the object is known as factory design pattern.

It is used to create a factory class, where we use this class to create all objects inside it. If we create objects directly in the controller using the **new** keyword, it results in a tightly coupled pipeline. To avoid this problem, we create a factory class (also referred to as a repository in this context) and convert it into a loosely coupled pipeline.

Encapsulation :-

It is the process of binding or storing data into a single object. To achieve this, we need to make all variables private. Using getter and setter methods, we can achieve encapsulation in our programming.

Getter method is used to get the data from the private field.

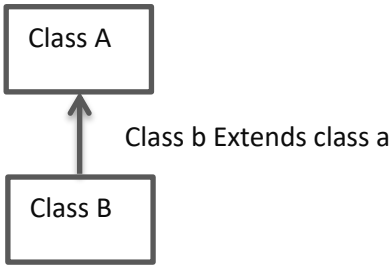
Setter method is used to set the data to the private variable.

Inheritance :-

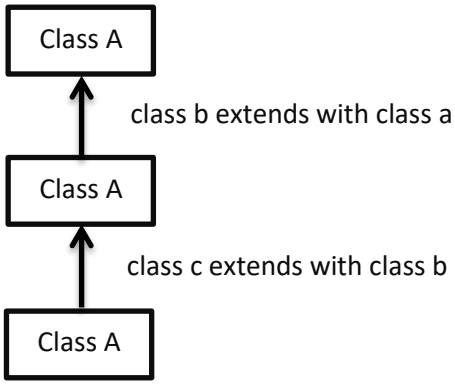
It is the process of inherit or acquiring a data from super class to sub class. To perform inheritance we use extends or implement keyword. Main purpose of inheritance for code reusability.

There are three types of Inheritance

i. Single level inheritance



ii. Multi-level inheritance



iii. Multiple inheritance (not used in java class, used in interface)
In this type, a single child class inherits properties from multiple parent classes. Multiple inheritances are **not allowed with classes** in Java to avoid ambiguity. However, it can be **achieved using interfaces** with the implements keyword.

In Java, classes do not support multiple inheritance because when we perform multiple inheritance using classes, it causes ambiguity due to the diamond problem.

For single level inheritance and multi-level inheritance we used extends keyword and for multiple inheritance we used implement keyword because of **Diamond problem**. For that reason java class is not support multiple inheritance.

Polymorphism :-

It is mean same methods with different input or arguments.

Polymorphism classified into two types :-

1. Run-time Polymorphisms (**Method Overriding)**
2. Compile time polymorphism.(**Method Overloading)**

i. **Method Overriding**
If we can't satisfy with parent method that time we create another method in child class. That time we used Method Overriding. It is
In method overriding it is compulsory to same name, same input, same or co-wallet return type and same or greater access modifier.

ii. **Method Overloading**
Same name but different formal argument or input, return type and access modifier will be anything.

Constructor :-

A constructor is similar to a method but without a return type. The constructor name is the same as the class name. Every class has one default constructor.

Uses of Constructor :-

1. We use a constructor to create an object.
2. To assign value to the private variable.

There are three types of constructor

I. Default constructor :- Default constructor is already present in class.

II. No argument Constructor :- Constructor without formal argument.

Example:-

```
Demo() {
    System.out.println("Default Constructor");
}
```

III. Parameterized Constructor :- contain some formal argument.

Example :-

```
ClassName(int value) {
    x = value;
}
```

This Keyword and Super Keyword :-

1. This keyword :-

- The this keyword refers to the **current object** of the class.
- It is used to **access variables, methods, or constructors** of the current class.
- Commonly used to **differentiate between instance variables and parameters** with the same name.
- It can be used to **call another constructor** in the same class using this().

2. Super keyword :-

- The super keyword refers to the **parent (superclass) object**.
- It is used to **access parent class variables, methods, and constructors**.
- By default, every constructor calls super() as the **first statement** to invoke the parent class's no-argument constructor.
- You can explicitly call a **parent class constructor with arguments** using super(arguments).

Interface:-

Interface is a keyword, where we can used interface keyword to create interface. Interface content only abstract method we can't define concrete method in interface. In interface every method is public abstract method and very variable is public static final.

We can't create object of a interface because when we create object of any class that time constructor is help us, but in interface there is no constructor for that reason we can't create object of an interface. To inherit the interface to any class we need to used implements keyword, but when we want to inherit interface to interface that time we need to used extends keyword.

Abstract Class:-

An abstract is a keyword, where we can used abstract to create an abstract class. Abstract class content abstract method as well as concrete method.

Abstract class content constructor but we can't create object of abstract class, because abstract class is not a complete class for that reason we can't create object. If we want to inherit abstract class that time compulsory to need to override the abstract method in normal class. And after that we create normal class object in main class and using this object we called methods from abstract class.

Question:- Then why abstract class content constructor in abstract class?

In abstract class constructor is used to assign the values to the variable which is declared in abstract class.

Static Keyword:- Static keyword we used to declared static method, once we declare any method as static then as a good programmer we can't called this method using object variable, we directly called using class name.

Final Keyword :- We use the final keyword at the class level, method level, and variable level in Java:

- If a **class** is declared as final, it **cannot be inherited** by any other class.
- If a **method** is declared as final, it **cannot be overridden** by subclasses.
- If a **variable** is declared as final, it **cannot be re-initialized** (its value can't be changed once assigned).

Automatic Promotion :-

In Java, automatic promotion means when we perform operations between different data types, the smaller data type is automatically converted into a bigger data type. This is done to avoid errors and make the operation work smoothly.

Access Modifier :

In Java, access modifiers are used to define the access level of variables, methods, constructors, or classes. It controls how other classes can access them.

There are four types of access modifiers in Java:

1. **private** – Accessible **only within the same class**
2. **default** (no modifier) – Accessible **within the same package only**
3. **protected** – Accessible **within the same package** and **subclasses in other packages**
4. **public** – Accessible **from anywhere**

Greater → Lower

Public → Protected → Default → Private

Design Pattern :-

1. Factory :-

Factory is a design pattern, with the help of factory design pattern we make our application loosely coupled. Instead of we create object using new keyword we take help of factory design pattern. This is because when we create object using new keyword leads to tight coupling. To avoid this problem we used factory design pattern.

Tight coupling:- When we create object using new keyword, is known as Tightly coupling.

Loosely coupling:- Instate of we creating object spring is create object for us, is known as Loosely coupling.

Dependency Pulling :-

Dependency pulling: To make our application loosely coupled we use dependency pulling. In the dependency pulling we can pull the dependency from spring factory (ApplicationContex <---@interface hai ye). ApplicationContex is a factory provided by spring to maintain the bean lifecycle.

Note: Here We need to do down casting because spring factory will return Object(class) type object.

2. Template Design pattern.

In Spring, the Template Design Pattern is used for code reusability. We can create multiple common methods inside an abstract class, and to call all of them at once, we define one new method known as the template method which calls those methods in order.

Q. What is IOC.

IoC means Inversion of Control. When we start application IOC container scan all the classed which have stereotype Annotations. It will create object of those classes. This object store in logical memory is known is IOC container.

Instead of creating objects using the new keyword and making our application tightly coupled, Spring creates the objects for us and makes the application loosely coupled. which helps us make our application loosely coupled

SOLID Principle:-

S – Single class single responsibility

O – Open for extension closed for modification

L –

I –

D – Dependency Injection

Stereotype annotations.

i. @Service :-

We use the **@Service** annotation to tell Spring that this class should be created and managed as a service component, and we use the **@Autowired** annotation to automatically inject that service into other classes where it is needed.

ii. @Controller :-

We use the **@Controller** annotation (or more commonly **@RestController**) in Spring to create and handle REST APIs that respond to client requests.

iii. @Repository :-

@Repository is a **stereotype annotation.** It tells Spring that this class **handles database operations.**

iv. @Component :-

@Component is a generic stereotype annotation in Spring. It tells Spring to detect and manage this class as a bean automatically during component scanning

Dependency Injection :-

Dependency injection is a core feature of spring, which is help us to make our application loosely coupled, instead of we creating object using new keyword spring is create object for us. For that factory will to spring, spring factory will used applicationcontex and getBean to make our application loosely coupled.

Dependency injection is classified into 4 different types.

i. **Field Injection :-**
In **field injection**, Spring injects the dependency **directly into the field (variable)** using the @Autowired annotation. In this type of injection, there is **no need to create constructor or setter methods** — Spring automatically injects the object

ii. **Setter injection:-**
In setter injection, Spring uses setter methods to inject dependencies after the bean is constructed. Setter injection is not mandatory by default. Using the @Autowired annotation, we can make this injection optional by writing: @Autowired(required = false).

iii. **Constructor Injection :-**

In **constructor injection**, dependencies are passed to a class through its **constructor**. Spring uses the constructor to create the bean and automatically injects the required dependencies.

iv. **Lookup Injection**

@getMapping :- It is used to get the data or fetch the data from database or from the server.

@postMapping :- It is used to post and set the data into database or server.

@putMapping :- It is used to update the data stored in database.

@deleteMapping :- It is used to delete the data from database.

@ **Qualifier :-** When there are **multiple objects of the same type**, Spring gets **confused** about which one to inject. To solve this, we use @Qualifier to **tell Spring exactly which bean to use** **or**

@Qualifier :- is used to **specify the class name** so that Spring can inject the correct one when there are multiple choices.

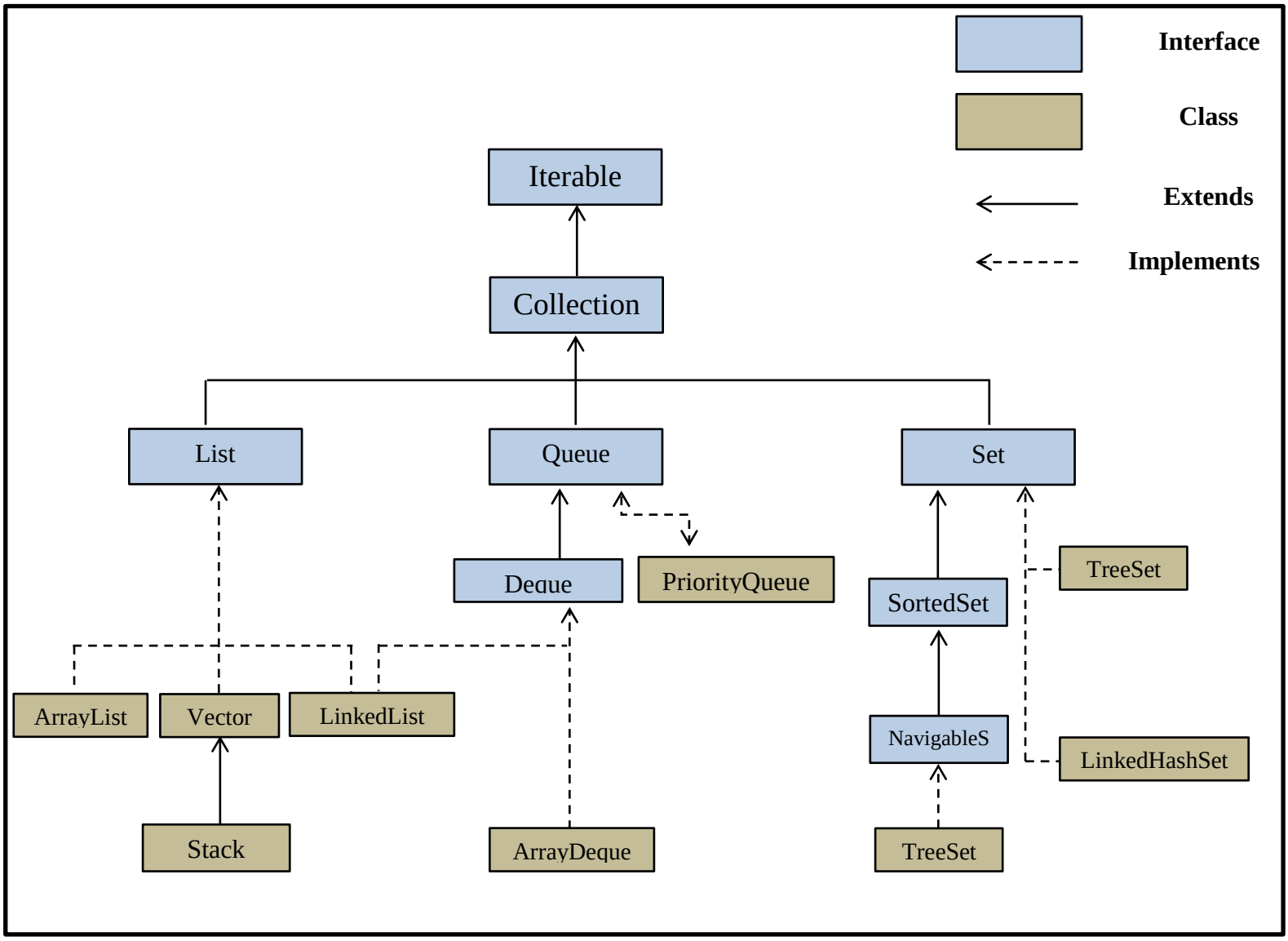
@Primary :- If you don't want to use @Qualifier every time, you can use @Primary to **mark one bean as the default**. Spring will use the @Primary bean **automatically**, unless you specifically ask for another one with @Qualifier.

Collection notes

Arrays :-

1. An array is a data structure used to store multiple values of the same type.
2. It creates a block of memory to store a fixed number of elements of a homogeneous (same) type.
3. An array is fixed in size — once we declare its size, we cannot change it.
4. Array elements can be accessed quickly using their index (starting from 0).
5. We can declared array as follows :
    i. Datatype [ ] variable = new Datatype[Size];
    ii. int a[ ] = new int[size];
    iii. int [ ] b = {1,2,3,4,5,6,7,8};

Collection Hierarchy



I. Iterable :

Iterable is an interface in Java. It is the parent interface of the Collection framework. All child interfaces like **List, Set, and Queue** implement Iterable, and all their implementing classes also inherit it.

II. Collection :

The **Collection** interface which is represents a group of objects. It allows storing and processing **different types of** objects. The Collection interface has **three main child interfaces**:

- **List** – Stores elements in an ordered manner and allows duplicates.
- **Queue** – Follows FIFO (First-In-First-Out) order, mainly used for processing elements.
- **Set** – Does not allow duplicate elements.

1. **List :**
    - i. **ArrayList**
        1. **ArrayList** implements List interface.
        2. **ArraysList** are good for reading data because accessing elements by index is fast.
        3. However, when inserting data in the middle, all elements must be shifted, which make our operation time-consuming.
        4. For that reason, we can say that **ArrayList** are good for reading and bad for insertion.
    - ii. **LinkedList**
        1. **LinkedList** implements List Interface.
        2. It is good for adding and removing items, especially in the middle of the list.
        3. It stores data in "nodes", and each node is connected to the next one and last one.
        4. But getting an item by its position is slower because we have to go through the list one by one.
    - iii. **Vector**
        1. **Stack**