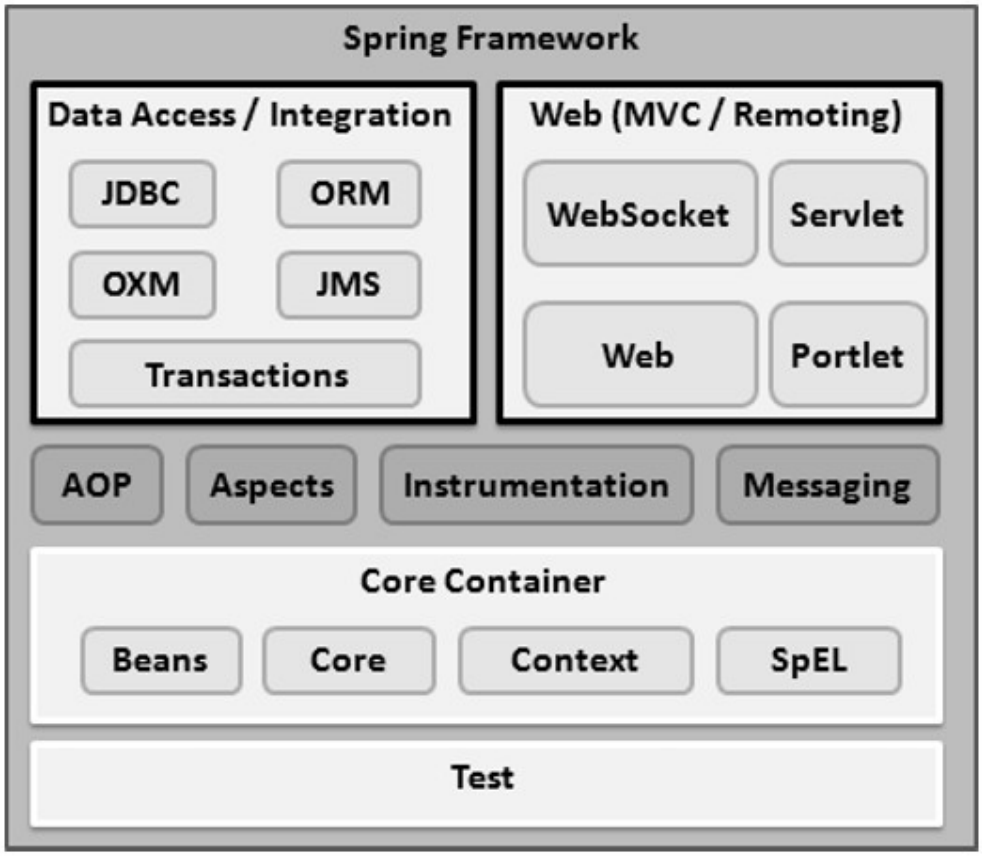
Spring Framework:

Spring framework is an open source Java platform that provides comprehensive infrastructure support for developing robust Java applications very easily and very rapidly. Spring is modular, allowing you to pick and choose which modules are applicable to you, without having to bring in the rest.The Spring Framework provides about 20 modules which can be used based on an application requirement.
Spring is a dependency Injection framework to make java application loosely coupled.

Key features of the Spring Framework include:

- **Inversion of Control (IoC):** Manages the lifecycle and dependencies of objects through Dependency Injection (DI).
- **Aspect-Oriented Programming (AOP):** Allows the separation of cross-cutting concerns like logging and transaction management.
- **Data Access**: Simplifies interaction with databases through JDBC and ORM frameworks like Hibernate.
- **Transaction Management:** Provides declarative and programmatic transaction management.



What is Inversion Of Control (IoC)?

Inversion Of Control (IoC) is a concept of software engineering, where the control of objects or a block of a program is transferred to a container or framework.

IoC allows a container to take control of the flow of a program and make calls to our custom code rather than making calls to a library.

To implement IoC, the Spring Dependency Injection concept is used.

What is Spring Container?

In the Spring framework, the interface 'ApplicationContext' represents the Spring container.

The Spring container takes responsibility of instantiating, configuring and assembling objects (Spring beans), and managing their life cycles as well.

In order to create beans, the Spring container takes help of configuration metadata, which can be in the form of either XML configuration or annotations.

What is a Spring Bean?

In the context of Spring Framework, an object which is managed (instantiated, configured & assembled) by the Spring container is known as Spring bean.

Dependency in Spring?

Consider a simple core Java class 'Invoice', its supporting class Vendor and interface TaxDetails as shown below.

```
public class Invoice {

  String name;
  Double amount;
  List<String> items;
  Vendor vendor;
  TaxDetails taxDetails;
}

class Vendor {

}

interface TaxDetails {

}
```

Invoice is a class which has various instance variables: name, amount, items, vendor, and taxDetails.

If we want to create an object of Invoice, we need all instance variables.

In other words, instantiation of class Invoice depends on all the instance variables inside the class. In the context of Spring, our Invoice class/bean has six dependencies: name, amount, items, vendor, and taxDetails. Now, let's see different types of dependencies that Spring Container can inject.

What is a dependent bean in Spring?

In the previous example of Invoice class. We defined the Invoice class as a bean in spring, and it has helper or supporting classes(dependencies) as 'Vendor' and 'TaxDetails'. Here, 'Invoice' is a target bean and 'Vendor' is the dependent bean.

A class/spring bean that takes support of other class/spring bean is called **target class** or main class.
A class/spring bean that acts as a helper class to other class/spring bean is called the **dependent class**.

What is Injection in the term Spring Dependency Injection?

As the word 'injection' suggests, inserting something into another thing.

Injection is the process of inserting values to variables while creating an object.

Dependency Injection is a creational pattern where an object receives its dependencies from an external source rather than creating them itself.

In Spring, this is achieved through

1. XML configuration
2. Annotations
3. Java-based configuration.

Benefits of Dependency Injection Pattern in Spring:

- **Loose Coupling**: Objects are less dependent on each other, making the system more modular.
- **Easier Testing**: Dependencies can be easily mocked or stubbed out.
- **Enhanced Maintainability**: Changes to a dependency do not require changes to the dependent class.

What are Design Patterns?

Design patterns are standardized solutions to common problems in software design. They are templates designed to help developers overcome recurring issues and improve code efficiency and readability.

These patterns are categorized into three main types:

- **Creational Design Patterns**: Deal with object creation mechanisms. Examples include Singleton, Factory Method, and Dependency Injection. These patterns aim to create objects in a manner suitable for the given situation.
- **Structural Design Patterns**: Focus on object composition or the way relationships between entities are realized. Examples include Adapter, Composite, and Proxy.
- **Behavioral Design Patterns**: Address communication between objects, defining how objects interact and distribute responsibility. Examples include Observer, Strategy, and Template Method.

What is the Factory Design Pattern?

The Factory design pattern in Spring is a creational pattern that provides a way to create objects without specifying the exact class of the object being created. Instead of directly calling a constructor, the factory pattern relies on a factory class or method to handle the object creation, promoting loose coupling and making the code more extensible.

Key Concepts of the Factory Pattern:

- **Abstraction:**

  The factory pattern abstracts the object creation process, allowing subclasses to alter the type of objects that will be created.

- **Loose Coupling:**

  It reduces dependencies between the object creation logic and its usage, leading to more modular and maintainable code.

- **Flexibility:**

  The factory pattern allows for easy swapping of implementations without affecting the code that uses the objects.

Example :
Ordinary Method :

```
interface Animal{}

public class Cat implements Animal {}
public class Dog implements Animal {}
public class Cow implements Animal {}

class AnimalFactory {
   public static Animal createAnimal(String type) {
      switch (type) {
         case "cat":
            return new Cat();
         case "dog":
            return new Dog();
         case "cow":
            return new Cow();
         default:
            throw new IllegalArgumentException("Invalid Animal type");
      }
   }
}
//The client code can use this factory method to create different animal types:
Animal cat= AnimalFactory.createAnimal("cat");
Animal dog= AnimalFactory.createAnimal("dog");
Animal cow = AnimalFactory.createAnimal("cow");
```

Factory Method: This pattern provides an **interface for creating objects**, but **allows subclasses to alter the type of objects** that will be created.

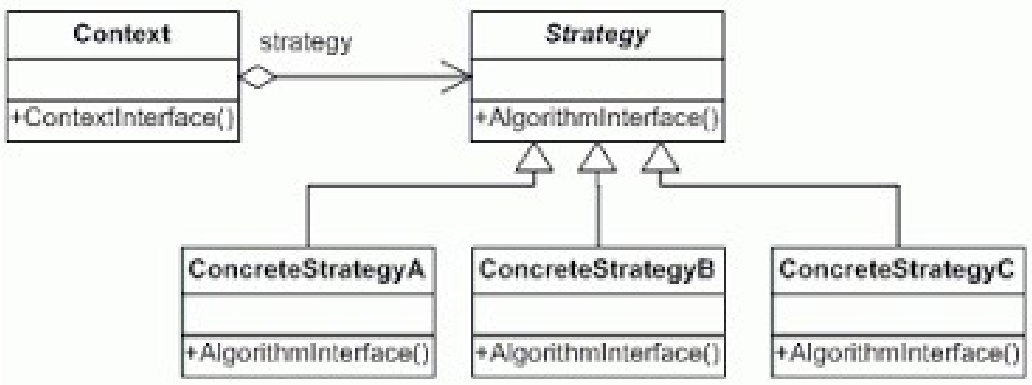
```
interface AnimalFactory {
   Animal createAnimal();
}

class CatFactory implements AnimalFactory {
   public Animal createAnimal() {
      return new Cat();
   }
}

class DogFactory implements AnimalFactory {
   public Animal createAnimal() {
      return new Dog();
   }
}

class CowFactory implements AnimalFactory {
   public Animal createAnimal() 007B
      return new Cow();
   }
}

//The client code can use the factory classes to create different animal types:
 AnimalFactory catFactory = new CatFactory();
 Cat cat = (Cat) catFactory.createAnimal();

 AnimalFactory dogFactory = new DogFactory();
 Dog dog = (Dog) dogFactory.createAnimal();

 AnimalFactory cowFactory = new CowFactory();
 Cow cow = (Cow) cowFactory.createAnimal();
```

What is the Strategy Design Pattern?

The Strategy Pattern in Spring, a behavioral design pattern, allows you to choose an algorithm or behavior at runtime. It defines a family of algorithms, encapsulates each one, and makes them interchangeable, enabling you to switch between them without altering the core logic of your application

Key Components:

- **Context:** The main class that uses the Strategy.
- **Strategy Interface:** A common interface for all algorithms.
- **Concrete Strategies:** Implementations of the Strategy interface, each representing a specific algorithm.
-



Example:

Consider a sorting application where we need to sort a list of integers. However, the sorting algorithm to be used may vary depending on factors such as the size of the list and the desired performance characteristics.

Imagine a scenario where you need to implement different payment methods in an e-commerce application. You can define a PaymentStrategy interface and then create concrete strategies for credit card payments, PayPal, etc. The e-commerce application can then choose the appropriate payment method at runtime based on user input.

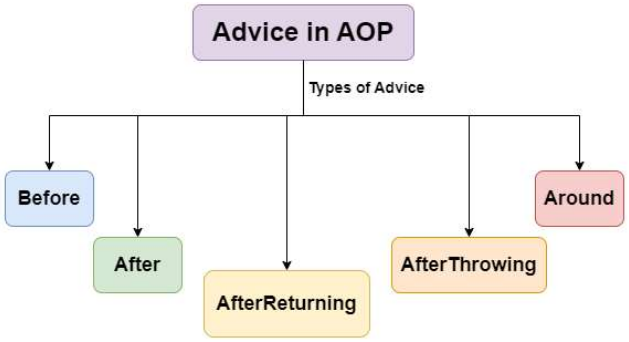
Aspect-Oriented Programming (AOP)

Aspect-Oriented Programming (AOP) is a powerful programming paradigm that enhances modularity by allowing you to separate cross-cutting concerns from your core business logic. These cross-cutting concerns, like logging, security, or transaction management, are handled separately, making your code more organized and easier to maintain.

Key Concepts:

- **Aspects:** These are modular units of code that encapsulate cross-cutting concerns. For example, a logging aspect can be applied across various methods in different classes..
- **Pointcuts:** These are expressions that specify where in your code the aspect's behavior should be applied.
- **Advice:** This is the actual code that the aspect executes at the pointcuts. There are five types of advice:
    - **Before**: Executed before the method call.
    - **After**: Executed after the method call, regardless of its outcome.
    - **AfterReturning**: Executed after the method returns a result, but not if an exception occurs.
    - **Around:** Surrounds the method execution, allowing you to control the method execution and its result.
    - **AfterThrowing**: Executed if the method throws an exception.

- **Join Points:** These are specific points in your application's execution flow where advice can be applied.
- **Weaving:** This is the process of dynamically applying the advice to the join points, typically during runtime.



- **Join Point:** A specific point in the execution of a program, such as method execution or exception handling, where an aspect can be applied.
- **Pointcut**: A Pointcut is a predicate that defines where advice should be applied. It matches join points using expressions.
- **Weaving**: This is the process of linking aspects with the target object. Spring AOP only supports runtime weaving using proxy-based mechanisms.

Working:

1. **1. Define Aspects:**

   You define aspects that contain the cross-cutting logic (e.g., logging before and after method execution).

2. **2. Specify Pointcuts:**

   You use pointcuts to pinpoint where in your code the aspect's behavior should be applied.

3. **3. Spring AOP automatically weaves the advice:**
   Spring AOP uses a proxy-based approach to automatically weave the advice into the target methods at runtime, without modifying the original code.

Bean scopes:

Bean scopes in Spring define the lifecycle and visibility of beans within the Spring IoC container. They determine how many instances of a bean are created and how those instances are shared. Here are the primary bean scopes:

1. Singleton:

- A single instance of the bean is created per Spring IoC container.
- This instance is shared across all requests for that bean.
- This is the default scope if no scope is specified.

2. Prototype:

- A new instance of the bean is created every time it is requested from the container.
- Each request receives a unique instance.
- Spring does not manage the complete lifecycle of prototype beans, the client is responsible for cleanup.

3. Request:

- A new instance of the bean is created for each HTTP request.
- Only valid in web-aware Spring applications.
- The bean is destroyed when the HTTP request completes.

4. Session:

- A new instance of the bean is created for each HTTP session.
- Only valid in web-aware Spring applications.
- The bean persists as long as the user's HTTP session is active.

5. Application:

- A single instance of the bean is created per ServletContext.
- Similar to singleton but specific to web applications.
- Shared across all requests and sessions within the ServletContext.

6. WebSocket:

- Creates a new instance per WebSocket session.
- Used for managing beans within WebSocket contexts.

7. Custom Scopes:

- Spring allows defining custom scopes for unique business requirements.
- Can be implemented to manage beans based on specific criteria like tenant or user.

How to Define Bean Scopes:

- Using the @Scope annotation on the class level with @Component or @Bean annotations.

- Using specialized annotations like @RequestScope, @SessionScope, or @ApplicationScope.
  Key Considerations:
- Choosing the correct scope is essential for managing state and behavior of beans.
- Singleton beans are suitable for stateless services.
- Prototype beans are used for stateful objects that should not be shared.
- Web-specific scopes (request, session) are for web applications.
- Application scope is for application-wide data.

What is Autowiring?

Autowiring is a feature in the Spring Framework that automatically injects dependencies into components, eliminating the need for manual configuration. It simplifies the process of wiring beans together by matching the types of the properties, constructor arguments, or method arguments.

Spring Container detects the relationship between the beans either by reading the XML Configuration file or by scanning the Java annotations at the time of booting up the application. Further, it will create the objects & wire the dependencies. Since Spring Container does this process automatically, it is referred to as Autowiring

Developer only needs to provide @Autowired annotation at a specific place.

Annotations in Spring

Annotations are a form of metadata that provides information about the code. They are used to configure Spring applications and provide instructions to the Spring container.

Common Spring Annotations:

- **@Component**: Marks a class as a Spring-managed component.
- **@Service**: Marks a class as a service component.
- **@Repository**: Marks a class as a repository component.
- **@Controller**: Marks a class as a controller component.
- **@Autowired**: Used for automatic dependency injection.
- **@Bean**: Declares a method that returns a bean to be managed by the Spring container.
- **@Configuration**: Marks a class as a configuration class.
- **@SpringBootApplication**: Combines @Configuration, @EnableAutoConfiguration, and @ComponentScan.

Lifecycle callbacks in Spring

The **lifecycle of a bean in Spring** refers to the sequence of events that occur from the moment a bean is instantiated until it is destroyed. Bean life cycle is managed by the spring container. When we run the program, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request, and then dependencies are injected. Finally, the bean is destroyed when the spring container is closed.

Understanding this lifecycle is important for managing resources effectively and ensuring that beans are properly initialized and cleaned up.

Phases of Bean Life Cycle :

- **Container Started:** The Spring IoC container is initialized.
- **Bean Instantiated:** The container creates an instance of the bean.
- **Dependencies Injected:** The container injects the dependencies into the bean.
- **Custom init() method:** If the bean implements InitializingBean or has a custom initialization method specified via @PostConstruct or init-method.
- **Bean is Ready:** The bean is now fully initialized and ready to be used.
- **Custom utility method:** This could be any custom method you have defined in your bean.
- **Custom destroy() method:** If the bean implements DisposableBean or has a custom destruction method specified via @PreDestroy or destroy-method, it is called when the container is shutting down.

Bean Configuration styles:

Spring provides three ways to implement the life cycle of a bean

1. XML Configuration:

- Beans are defined in an XML file using the <bean> tag.
- Properties and dependencies are configured using XML attributes and nested elements.
- This was the traditional way of configuring Spring beans.

2. Annotation-Based Configuration:

- Beans are defined using annotations directly in your Java classes.
- @Component, @Service, @Repository, and @Controller are used to mark classes as beans.
- @Autowired is used for dependency injection.
- @Configuration and @Bean are used for more explicit bean definitions.
- This style is more concise and type-safe compared to XML.

3. Java-Based Configuration:

- Beans are defined within Java classes annotated with @Configuration.
- Methods annotated with @Bean are used to create and configure beans.
- This style provides full control over bean creation and configuration.
- It's often used for more complex scenarios or when XML or annotation-based approaches are insufficient.

Spring: Spring Core Basics-Spring Dependency Injection concepts, Introduction to Design patterns, Factory Design Pattern, Strategy Design pattern, Spring Inversion of Control, AOP, Bean Scopes,Singleton, Prototype, Request, Session, Application, WebSocket, Auto wiring, Annotations, Life Cycle Call backs, Bean Configuration styles