**Spring Core Concepts**

1. **Dependency Injection:**

* Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application.
* Dependency Injection makes our programming code loosely coupled.
* There are **two different ways** of Dependency Injection Configurations.

1. **XML based Configuration** - Spring Framework helps us by providing a way to detect the relationship between the beans by reading the XML Configuration file or scanning the Java annotations when booting up the application
2. **Java Annotation-based Configurations** - With Java Annotation Configurations, developers wire beans by using the @Autowired annotation.

* Advantages:
* Achieve loosely coupled architecture by removing the tight coupling/dependency between a class & its dependence.
* loosely coupled helps the developers test the module by injecting the dependent Mock Objects.
* Removes unnecessary dependencies between the classes.
* Modules are independent of each other & can be injected, the scope of making the component reusable is very high.
* Disadvantages:
* Difficult to trace the code as the developer needs to refer to more files.
* Spring Framework takes care of the control rather than the developer, it will be difficult for the developers to understand how things work in the background & also to have customization.
* Dependency Injection allows loose coupling, increasing the number of Interfaces & classes.

Spring framework provides three ways to inject dependency

|  |  |  |
| --- | --- | --- |
| **Constructor-based DI** | **Setter-based DI** | **Field or property-based DI** |
| Constructor-based Dependency Injection refers to the use of the @Autowired annotation on top of a class **constructor**  @component  @RestController  public class CompanyDetail{  public class companyController {  private long Id;  private CompanyDetail cd;  private String name;  }  @Autowired  public companyController (CompanyDetail cd){  this.cd=cd;  }  } | Setter-based Dependency Injection refers to the use of the @Autowired annotation on top of the **setter** **method** of a class.  @component  @RestController  public class CompanyDetail{  public class companyController {  private long Id;  private CompanyDetail cd;  private String name;  }  @Autowired  public setcompanyController (CompanyDetail cd) {  //using setter  this.cd=cd;  }  } | Field-based Dependency Injection refers to the use of the @Autowired annotation on top of a field or property in a class.  @component  @RestController  public class CompanyDetail{  public class companyController {  private long Id;  private String name;  @Autowired  private CompanyDetail cd;  }  } |

1. **IOC and its memory management during project run.**

* IOC (Inversion of Control):
* The IoC container is responsible to instantiate, configure and assemble the objects.
* The IoC container gets information from the XML file and works accordingly.
* There are two types of IoC containers: **Bean Factory & Application Context**
* **Traditional Approach**: In traditional Java applications, objects often create and manage their own dependencies. This can lead to tight coupling and make the code less flexible and harder to test.
* **Spring's IoC**: Spring inverts this control. Instead of objects creating their dependencies, the Spring container (IoC container) is responsible for:
* **Instantiating objects (beans):** The container creates instances of your classes based on configuration (XML, annotations, or Java configuration).
* **Configuring objects:** The container sets properties and injects dependencies into your beans, wiring them together.
* **Managing the lifecycle of objects:** The container controls when beans are created, initialized, and destroyed.
* **Memory Management and the IoC Container**
* Bean Scopes**:** Spring provides different bean scopes that determine how many instances of a bean are created and how long they live:
* **Singleton** (default): One instance per Spring container. Efficient for stateless beans.
* **Prototype**: A new instance is created each time the bean is requested. Useful for stateful beans.
* **Request, session, application, web socket:** Scopes tied to the web environment's lifecycle.
* Lazy Initialization: By default, singleton beans are eagerly initialized when the container starts. You can use @Lazy to delay bean creation until its first requested, saving memory on startup.
* Garbage Collection: The JVM's garbage collector is responsible for reclaiming memory occupied by objects that are no longer referenced. Spring beans are subject to garbage collection like any other Java object.
* Circular Dependencies: The container can handle some circular dependencies between beans, but it's generally best to avoid them as they can complicate the code and impact startup time.
* Benefits of IoC
* Loose coupling: Easier to change and test components independently.
* Simplified configuration: Centralized management of bean dependencies and lifecycle.
* Reusability: Beans can be easily reused across different parts of the application.
* Testability: Easier to mock or stub dependencies during unit testing.

1. **Spring Beans and Bean lifecycle:**

* Spring Beans
* **Core of Spring Applications**: Beans are the fundamental building blocks managed by the Spring IoC container. They represent the objects that form the backbone of your application, such as services, repositories, components, and controllers.
* **Managed by the Container**: The Spring container is responsible for creating, configuring, and assembling these beans based on your provided configuration (XML, annotations, or Java configuration). This allows you to focus on the core logic of your beans, while the container handles their dependencies and lifecycle.
* Benefits of Using Beans:
* **Loose Coupling**: Promotes modularity and testability by allowing you to easily swap implementations or mock dependencies.
* **Reusability**: Beans can be easily shared and reused across different parts of your application.
* **Lifecycle Management**: The container provides hooks into the bean lifecycle, allowing you to perform custom initialization and cleanup tasks.
* Spring Bean Lifecycle

The lifecycle of a Spring bean consists of several distinct stages:

* **Instantiation**: The container creates a new instance of the bean class using its constructor.
* **Dependency Injection (DI):** The container injects any required dependencies into the bean, satisfying its collaborators.
* **Initialization**: The bean performs any necessary initialization logic, such as opening database connections or loading configuration files. This can be done through:
* Implementing the InitializingBean interface and overriding the afterPropertiesSet() method.
* Specifying an init-method in the bean configuration.
* Using the @PostConstruct annotation on a method.
* **Usage**: The bean is ready for use and can be accessed by other parts of the application.
* **Destruction**: When the application context is closed, the bean performs any necessary cleanup, such as closing resources or releasing connections. This can be done through:
* Implementing the DisposableBean interface and overriding the destroy() method.
* Specifying a destroy-method in the bean configuration.
* Using the @PreDestroy annotation on a method.

1. **Spring AOP (Aspect-Oriented Programming)**

* Spring AOP is a powerful programming paradigm that allows you to modularize cross-cutting concerns in your Spring applications.
* **Cross-cutting concerns**: These are functionalities that span multiple parts of your application, such as logging, transaction management, security, and exception handling. Traditionally, these concerns are scattered throughout the codebase, leading to code tangling and maintenance difficulties.
* **AOP Solution:** AOP addresses this by providing a way to encapsulate these cross-cutting concerns into separate modules called aspects. Aspects can then be applied to specific points in your code, known as join points, without modifying the core business logic.
* Key Terminology in Spring AOP:
* **Aspect**: A modular unit that encapsulates a cross-cutting concern. It defines the advice and pointcuts that determine where and how the concern should be applied.
* **Advice**: The actual code that is executed at a specific join point. Spring AOP provides several types of advice:
* Before advice: Executes before a join point.
* After returning advice: Executes after a join point successfully completes.
* After throwing advice: Executes if a join point throws an exception.
* After (finally) advice: Executes regardless of whether a join point completes successfully or throws an exception.
* Around advice: Surrounds a join point, allowing you to control its execution flow.
* Join Point: A specific point in the execution of your program where an aspect can be applied, such as a method call, field access, or exception handling.
* Pointcut: An expression that defines the set of join points where an aspect's advice should be applied. Pointcuts use patterns to match specific methods, classes, or annotations.
* Target Object: The object being advised by one or more aspects.
* Weaving: The process of linking aspects with the target objects to create the advised objects. Spring AOP uses runtime weaving through proxies.
* Benefits of Spring AOP
* **Modularity**: Encapsulates cross-cutting concerns into separate modules, improving code organization and maintainability.
* **Reusability**: Aspects can be easily applied to multiple parts of your application, promoting code reuse.
* **Separation of Concerns**: Decouples core business logic from cross-cutting concerns, making your code cleaner and easier to understand.
* **Flexibility**: Allows you to dynamically add or remove aspects without modifying the target code.
* Common Use Cases
* **Logging**: Track method calls, parameters, and execution times for debugging and performance analysis.
* **Transaction Management**: Declaratively manage transactions around method calls, ensuring data consistency.
* **Security**: Enforce authentication and authorization checks before executing sensitive operations.
* **Exception Handling:** Centralize error handling and logging across your application.
* **Caching**: Improve performance by transparently caching method results.
* Implementation in Spring

Spring AOP provides two ways to implement aspects:

* **XML Configuration**: Define aspects and pointcuts using XML-based configuration.
* **Annotations**: Use annotations like @Aspect, @Before, @AfterReturning, etc., to declaratively define aspects and advice.