**IOC**

* Spring helps in creating loosely coupled application because of **Dependency Injection**.
* In spring objects define their associations (dependencies) and do not worry about how to get those **dependencies** ; now it is the responsibility of Spring to provide the required dependencies for creating objects.

**For example** : Suppose we have an object Employee and it has a dependency on object Address. So we define a bean corresponding to Employee where it will define its dependency on object Address. When Spring tries to create an Object Employee it sees that Employee has a dependency on object Address so first it will create the Address object (dependent object) and then inject this into the Employee Object.

* Inversion of Control **(IOC)** and Dependency Injection **(DI)** are used interchangeably. IOC is achieved through DI. DI is the process of providing the dependencies and IOC is the end result of DI (**Note:** DI is not the only way to achieve IOC, there are [other ways](https://en.wikipedia.org/wiki/Inversion_of_control#Implementation_techniques) as well).
* By DI the responsibility of creating objects is shifted from our application code to Spring container hence the phenomenon is called IOC.
* Dependency Injection can be done by setter injection, constructor injection.

**Bean scope**

All beans in the context are instantiated, injected and initialized when the context starts up. By the time the first bean has been retrieved from the context, all beans are ready for use.

There are two things that can prevent a bean being initialized at context start up:

* A bean has bean configured with a [different scope](http://static.springsource.org/spring/docs/3.0.x/reference/html/beans.html#beans-factory-scopes) (such as prototype, request or session), using the scope="xyz" attribute
* A bean has been marked with [lazy-init="true"](http://static.springsource.org/spring/docs/3.0.x/reference/html/beans.html#beans-factory-lazy-init), in which case it will only be instantiated when it's explicitly asked for, or if it's required as a dependency of some other bean.

What bean scopes does Spring support? Explain them.

The Spring Framework supports following five scopes, three of which are available only if the users use a web-aware Application Context.

**Singleton:** This scopes the bean definition to a single instance per Spring IoC container.

**Prototype:** This scopes a single bean definition to have any number of object instances.

**Request:** This scopes a bean [definition to an HTTP request](https://intellipaat.com/interview-question/mysql-interview-questions/). Only valid in the context of a web-aware Spring ApplicationContext

**Session:** This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

**Global-session:** This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

**Transactions**

|  |
| --- |
| **Describe Global and Local transactions in Spring.** |
| Ans: Global transactions help to work with multiple transactional resources like relational database and message queue. Global transactions are managed through JTA and JNDI.  Local transactions are resource-specific like JDBC connection. Local Transactions can work with multiple transactional resources.  **Q2. What are the benefits of using Spring?**  Spring targets to make Java EE development easier. Here are the advantages of using it:   * **Lightweight:**there is a slight overhead of using the framework in development * **Inversion of Control (IoC):** Spring container takes care of wiring dependencies of various objects, instead of creating or looking for dependent objects * **Aspect Oriented Programming (AOP):**Spring supports AOP to separate business logic from system services * **IoC container:** it manages Spring Bean life cycle and project specific configurations * **MVC framework:** that is used to create web applications or RESTful web services, capable of returning XML/JSON responses * **Transaction management:** reduces the amount of boiler-plate code in JDBC operations, file uploading, etc., either by using Java annotations or by Spring Bean XML configuration file * **Exception Handling:**Spring provides a convenient API for translating technology-specific exceptions into unchecked exceptions   **Q3. What Spring sub-projects do you know? Describe them briefly.**   * **Core** – a key module that provides fundamental parts of the framework, like IoC or DI * **JDBC** – this module enables a JDBC-abstraction layer that removes the need to do JDBC coding for specific vendor databases * **ORM integration** – provides integration layers for popular object-relational mapping APIs, such as JPA, JDO, and Hibernate * **Web** – a web-oriented integration module, providing multipart file upload, Servlet listeners, and web-oriented application context functionalities * **MVC framework** – a web module implementing the Model View Controller design pattern * **AOP module** – aspect-oriented programming implementation allowing the definition of clean method-interceptors and pointcuts   **Q4. What is Dependency Injection?**  Dependency Injection, an aspect of Inversion of Control (IoC), is a general concept stating that you do not create your objects manually but instead describe how they should be created. An IoC container will instantiate required classes if needed.  For more details, please refer [here](http://www.baeldung.com/inversion-control-and-dependency-injection-in-spring).  **Q5. How can we inject beans in Spring?**  A few different options exist:   * Setter Injection * Constructor Injection * Field Injection   The configuration can be done using XML files or annotations.  For more details, check [this article](http://www.baeldung.com/inversion-control-and-dependency-injection-in-spring).  **Q6. Which is the best way of injecting beans and why?**  The recommended approach is to use constructor arguments for mandatory dependencies and setters for optional ones. Constructor injection allows injecting values to immutable fields and makes testing easier.  **Q7. What is the difference between *BeanFactory* and *ApplicationContext*?**  *BeanFactory* is an interface representing a container that provides and manages bean instances. The default implementation instantiates beans lazily when *getBean()* is called.  *ApplicationContext*is an interface representing a container holding all information, metadata, and beans in the application. It also extends the *BeanFactory* interface but the default implementation instantiates beans eagerly when the application starts. This behavior can be overridden for individual beans.  For all differences, please refer to [the reference](https://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html). Introduction to the Spring IoC container and beans This chapter covers the Spring Framework implementation of the Inversion of Control (IoC) [[1](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#_footnote_1)] principle. IoC is also known as dependency injection (DI). It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the Service Locator pattern.  The org.springframework.beans and org.springframework.context packages are the basis for Spring Framework’s IoC container. The [BeanFactory](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/beans/factory/BeanFactory.html) interface provides an advanced configuration mechanism capable of managing any type of object. [ApplicationContext](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/context/ApplicationContext.html) is a sub-interface of BeanFactory. It adds easier integration with Spring’s AOP features; message resource handling (for use in internationalization), event publication; and application-layer specific contexts such as the WebApplicationContext for use in web applications.  In short, the BeanFactory provides the configuration framework and basic functionality, and the ApplicationContext adds more enterprise-specific functionality. The ApplicationContext is a complete superset of the BeanFactory, and is used exclusively in this chapter in descriptions of Spring’s IoC container. For more information on using the BeanFactory instead of the ApplicationContext, refer to [The BeanFactory](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-beanfactory).  In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container. Dependency resolution process The container performs bean dependency resolution as follows:   * The ApplicationContext is created and initialized with configuration metadata that describes all the beans. Configuration metadata can be specified via XML, Java code, or annotations. * For each bean, its dependencies are expressed in the form of properties, constructor arguments, or arguments to the static-factory method if you are using that instead of a normal constructor. These dependencies are provided to the bean, when the bean is actually created. * Each property or constructor argument is an actual definition of the value to set, or a reference to another bean in the container. * Each property or constructor argument which is a value is converted from its specified format to the actual type of that property or constructor argument. By default Spring can convert a value supplied in string format to all built-in types, such as int, long, String, boolean, etc.   The Spring container validates the configuration of each bean as the container is created. However, the bean properties themselves are not set until the bean is actually created. Beans that are singleton-scoped and set to be pre-instantiated (the default) are created when the container is created. Scopes are defined in [Bean scopes](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes). Otherwise, the bean is created only when it is requested. Creation of a bean potentially causes a graph of beans to be created, as the bean’s dependencies and its dependencies' dependencies (and so on) are created and assigned. Note that resolution mismatches among those dependencies may show up late, i.e. on first creation of the affected bean.  Circular dependencies  If you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.  For example: Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a BeanCurrentlyInCreationException.  One possible solution is to edit the source code of some classes to be configured by setters rather than constructors. Alternatively, avoid constructor injection and use setter injection only. In other words, although it is not recommended, you can configure circular dependencies with setter injection.  Unlike the *typical* case (with no circular dependencies), a circular dependency between bean A and bean B forces one of the beans to be injected into the other prior to being fully initialized itself (a classic chicken/egg scenario).  **Q8. What is a Spring Bean?**  The Spring Beans are Java Objects that are initialized by the Spring IoC container.  **Q9. What is the default bean scope in Spring framework?**  By default, a Spring Bean is initialized as a *singleton*.  **Q10. How to define the scope of a bean?**  To set Spring Bean’s scope, we can use *@Scope* annotation or “scope” attribute in XML configuration files. There are five supported scopes:   * **singleton** * **prototype** * **request** * **session** * **global-session**   For differences, please refer [here](https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch03s05.html).  **Q11. Are singleton beans thread-safe?**  No, singleton beans are not thread-safe, as thread safety is about execution, whereas the singleton is a design pattern focusing on creation. Thread safety depends only on the bean implementation itself.  No. The two concepts are not even related.  Singletons are about **creation**. This design pattern ensures that only one instance of a class is created.  Thread safety is about **execution**. To quote [Wikipedia](http://en.wikipedia.org/wiki/Thread_safety):  A piece of code is thread-safe if it only manipulates shared data structures in a manner that guarantees safe execution by multiple threads at the same time.  So eventually thread safety depends on the code and the code only. And this is the reason why Spring beans are not thread safe per se.  **Q12. What does the Spring bean lifecycle look like?**  First, a Spring bean needs to be instantiated, based on Java or XML bean definition. It may also be required to perform some initialization to get it into a usable state. After that, when the bean is no longer required, it will be removed from the IoC container.  The whole cycle with all initialization methods is shown on the image ([source](http://www.dineshonjava.com/2012/07/bean-lifecycle-and-callbacks.html)):    Sometimes we want to initialize resources in the bean classes, for example creating database connections or validating third party services at the time of initialization before any client request. [Spring framework](https://www.journaldev.com/16922/spring-framework) provide different ways through which we can provide post-initialization and pre-destroy methods in a spring bean life cycle.   1. By implementing **InitializingBean** and **DisposableBean** interfaces – Both these interfaces declare a single method where we can initialize/close resources in the bean. For post-initialization, we can implement InitializingBean interface and provide implementation of afterPropertiesSet() method. For pre-destroy, we can implement DisposableBean interface and provide implementation of destroy() method. These methods are the callback methods and similar to servlet listener implementations.   This approach is simple to use but it’s not recommended because it will create tight coupling with the Spring framework in our bean implementations.   1. Providing **init-method** and **destroy-method** attribute values for the bean in the spring bean configuration file. This is the recommended approach because of no direct dependency to spring framework and we can create our own methods.   Note that both post-init and pre-destroy methods should have no arguments but they can throw Exceptions. We would also require to get the bean instance from the spring application context for these methods invocation. ****Q13. What is the Spring Java-Based Configuration?**** It’s one of the ways of configuring Spring-based applications in a type-safe manner. It’s an alternative to the XML-based configuration.  Also, if you want to migrate your project from XML to Java config, please refer [to this article](http://www.baeldung.com/spring-xml-vs-java-config). ****Q14. Can we have multiple Spring configuration files in one project?**** Yes, in large projects, having multiple Spring configurations is recommended to increase maintainability and modularity.  You can load multiple Java-based configuration files:   |  |  | | --- | --- | | 1  2  3 | @Configuration  @Import({MainConfig.class, SchedulerConfig.class})  public class AppConfig { |   Or load one XML file that will contain all other configs:   |  |  | | --- | --- | | 1 | ApplicationContext context = new ClassPathXmlApplicationContext("spring-all.xml"); |   And inside this XML file you’ll have:   |  |  | | --- | --- | | 1  2 | <import resource="main.xml"/>  <import resource="scheduler.xml"/> |  ****Q15. What is Spring Security?**** Spring Security is a separate module of the Spring framework that focuses on providing authentication and authorization methods in Java applications. It also takes care of most of the common security vulnerabilities such as CSRF attacks.  To use Spring Security in web applications, you can get started with a simple annotation: @EnableWebSecurity.  You can find the whole series of articles related to [security on Baeldung](http://www.baeldung.com/security-spring). ****Q16. What is Spring Boot?**** Spring Boot is a project that provides a pre-configured set of frameworks to reduce boilerplate configuration so that you can have a Spring application up and running with the smallest amount of code. ****Q17. Name some of the Design Patterns used in the Spring Framework?****  * **Singleton Pattern:** Singleton-scoped beans * **Factory Pattern:** Bean Factory classes * **Prototype Pattern:** Prototype-scoped beans * **Adapter Pattern:** Spring Web and Spring MVC * **Proxy Pattern:** Spring Aspect Oriented Programming support * **Template Method Pattern:** JdbcTemplate, HibernateTemplate, etc. * **Front Controller:** Spring MVC DispatcherServlet * **Data Access Object:** Spring DAO support * **Model View Controller:**Spring MVC  ****Q18. How does the scope**** Prototype****work?**** Scope prototype means that every time you call for an instance of the Bean, Spring will create a new instance and return it. This differs from the default singleton scope, where a single object instance is instantiated once per Spring IoC container. **3. Spring MVC******Q19. How to Get**** ServletContext ****and**** ServletConfig****Objects in a Spring Bean?**** You can do either by:   1. Implementing Spring-aware interfaces. The complete list is available [here](http://www.buggybread.com/2015/03/spring-framework-list-of-aware.html). 2. Using @Autowired annotation on those beans:  |  |  | | --- | --- | | 1  2  3  4  5 | @Autowired  ServletContext servletContext;    @Autowired  ServletConfig servletConfig; |  ****Q20. What is the role of the**** @Required ****annotation?**** The @Required annotation is used on setter methods, and it indicates that the bean property that has this annotation must be populated at configuration time. Otherwise, the Spring container will throw a BeanInitializationException exception.  Also, @Required differs from @Autowired – as it is limited to a setter, whereas @Autowired is not. @Autowired can be used to wire with a constructor and a field as well, while @Required only checks if the property is set.  Let’s see an example:   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8 | public class Person {      private String name;        @Required      public void setName(String name) {          this.name = name;      }  } |   Now, the name of the Person bean needs to be set in XML config like this:   |  |  | | --- | --- | | 1  2  3 | <bean id="person" class="com.baeldung.Person">      <property name="name" value="Joe" />  </bean> |   Please note that @Required doesn’t work with Java based @Configuration classes by default. If you need to make sure that all your properties are set, you can do so when you create the bean in the @Bean annotated methods. ****Q21. What is the role of the**** @Autowired****annotation?**** The @Autowired annotation can be used with fields or methods for injecting a bean by type. This annotation allows Spring to resolve and inject collaborating beans into your bean.  For more details, please refer [to this tutorial](http://www.baeldung.com/spring-autowire). ****Q22. What is the Role of the**** @Qualifier****Annotation?**** It is used simultaneously with the @Autowired annotation to avoid confusion when multiple instances of a bean type are present.  Let’s see an example. We declared two similar beans in XML config:   |  |  | | --- | --- | | 1  2  3  4  5  6 | <bean id="person1" class="com.baeldung.Person" >      <property name="name" value="Joe" />  </bean>  <bean id="person2" class="com.baeldung.Person" >      <property name="name" value="Doe" />  </bean> |   When we try to wire the bean, we’ll get an org.springframework.beans.factory.NoSuchBeanDefinitionException. To fix it, we need to use @Qualifier to tell Spring about which bean should be wired:   |  |  | | --- | --- | | 1  2  3 | @Autowired  @Qualifier("person1")  private Person person; |  ****Q23. How to handle exceptions in Spring MVC environment?**** There are three ways to handle exceptions in Spring MVC:   1. **Using @ExceptionHandler at controller level** – this approach has a major feature – the @ExceptionHandler annotated method is only active for that particular controller, not globally for the entire application 2. **Using HandlerExceptionResolver** – this will resolve any exception thrown by the application 3. **Using @ControllerAdvice**– Spring 3.2 brings support for a global @ExceptionHandler with the @ControllerAdvice annotation, which enables a mechanism that breaks away from the older MVC model and makes use of ResponseEntity along with the type safety and flexibility of @ExceptionHandler   For more detailed information on this topic, go through [this writeup](http://www.baeldung.com/exception-handling-for-rest-with-spring). ****Q24. How to validate if the bean was initialized using valid values?**** Spring supports [JSR-303](http://beanvalidation.org/1.0/spec/) annotation-based validations. JSR-303 is a specification of the Java API for bean validation, part of JavaEE and JavaSE, which ensures that properties of a bean meet specific criteria, using annotations such as @NotNull, @Min, and @Max. The article regarding JSR-303 is available [here](http://www.baeldung.com/javax-validation).  What’s more, Spring provides the Validator interface for creating custom validators. For example, you can have a look [here](http://www.baeldung.com/spring-mvc-custom-validator). ****Q25. What is Spring MVC Interceptor and how to use it?**** Spring MVC Interceptors allow us to intercept a client request and process it at three places – before handling, after handling, or after completion (when the view is rendered) of a request.  The interceptor can be used for cross-cutting concerns and to avoid repetitive handler code like logging, changing globally used parameters in Spring model, etc.  For details and various implementations, take a look [at this series](http://www.baeldung.com/spring-mvc-handlerinterceptor). ****Q26. What is a Controller in Spring MVC?**** Simply put, all the requests processed by the DispatcherServlet are directed to classes annotated with @Controller. Each controller class maps one or more requests to methods that process and execute the requests with provided inputs.  If you need to take a step back, we recommend having a look at the concept of the [Front Controller in the typical Spring MVC architecture](http://www.baeldung.com/spring-controllers). **4. Spring Web******Q27. How does the**** @RequestMapping ****annotation work?**** The @RequestMapping annotation is used to map web requests to Spring Controller methods. In addition to simple use cases, we can use it for mapping of HTTP headers, binding parts of the URI with @PathVariable, and working with URI parameters and the @RequestParam annotation.  More details on @RequestMapping are available [here](http://www.baeldung.com/spring-requestmapping). ****Q28. What’s the Difference Between**** @Controller****,**** @Component****,**** @Repository,****and**** @Service****Annotations in Spring?**** According to the official Spring documentation, @Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.  Let’s take a look at specific use cases of last three:   * **@Controller** – indicates that the class serves the role of a controller, and detects @RequestMapping annotations within the class * **@Service** – indicates that the class holds business logic and calls methods in the repository layer * **@Repository** – indicates that the class defines a data repository; its job is to catch platform-specific exceptions and re-throw them as one of Spring’s unified unchecked exceptions  ****Q29. What are**** DispatcherServlet ****and**** ContextLoaderListener****?**** Simply put, in the Front Controller design pattern, a single controller is responsible for directing incoming HttpRequests to all of an application’s other controllers and handlers.  **Spring’s DispatcherServlet implements this pattern and is, therefore, responsible for correctly coordinating the HttpRequests to the right handlers.**  On the other hand, ContextLoaderListener starts up and shuts down Spring’s root WebApplicationContext. It ties the lifecycle of ApplicationContext to the lifecycle of the ServletContext. We can use it to define shared beans working across different Spring contexts.  For more details on DispatcherServler, please refer [to this tutorial](http://www.baeldung.com/spring-dispatcherservlet). ****Q30. What is**** ViewResolver ****in Spring?**** The ViewResolver enables an application to render models in the browser – without tying the implementation to a specific view technology – by mapping view names to actual views.  For a guide to the ViewResolver, have a look [here](http://www.baeldung.com/spring-mvc-view-resolver-tutorial). ****Q31. What is a**** MultipartResolver ****and when is it used?**** The MultipartResolver interface is used for uploading files. The Spring framework provides one MultipartResolver implementation for use with Commons FileUpload and another for use with Servlet 3.0 multipart request parsing.  Using these, we can support file uploads in our web applications. **5. Spring Data Access******Q32. What is Spring**** JDBCTemplate ****class and how to use it?**** The Spring JDBC template is the primary API through which we can access database operations logic that we’re interested in:   * creation and closing of connections * executing statements and stored procedure calls * iterating over the ResultSet and returning results   To use it, we’ll need to define the simple configuration of DataSource:   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @Configuration  @ComponentScan("org.baeldung.jdbc")  public class SpringJdbcConfig {      @Bean      public DataSource mysqlDataSource() {          DriverManagerDataSource dataSource = new DriverManagerDataSource();          dataSource.setDriverClassName("com.mysql.jdbc.Driver");          dataSource.setUrl("jdbc:<mysql://localhost:3306/springjdbc>");          dataSource.setUsername("guest\_user");          dataSource.setPassword("guest\_password");            return dataSource;      }  } |   For further explanation, you can go through [this quick article](http://www.baeldung.com/spring-jdbc-jdbctemplate). ****Q33. How would you enable t****ransactions ****in Spring and what are their benefits?**** There are two distinct ways to configure Transactions – with annotations or by using Aspect Oriented Programming (AOP) – each with their advantages.  The benefits of using Spring Transactions, according to the [official docs](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/transaction.html), are:   * Provide a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO * Support declarative transaction management * Provide a simpler API for programmatic transaction management than some complex transaction APIs such as JTA * Integrate very well with Spring’s various data access abstractions  ****Q34. What is Spring DAO?**** Spring Data Access Object is Spring’s support provided to work with data access technologies like JDBC, Hibernate, and JPA in a consistent and easy way.  You can, of course, go more in-depth on persistence, with the [entire series](http://www.baeldung.com/persistence-with-spring-series/) discussing persistence in Spring. **6. Spring Aspect-Oriented Programming (AOP)******Q35. What is Aspect-Oriented Programming?**** Aspects enable the modularization of cross-cutting concerns such as transaction management that span multiple types and objects by adding extra behavior to already existing code without modifying affected classes.  Here is the example of [aspect-based execution time logging](http://www.baeldung.com/spring-aop-annotation). ****Q36. What are****Aspect****,**** Advice****,**** Pointcut, ****and****JoinPoint****in AOP?****  * ***Aspect***: a class that implements cross-cutting concerns, such as transaction management * **Advice**: the methods that get executed when a specific JoinPoint with matching Pointcut is reached in the application * ***Pointcut***: a set of regular expressions that are matched with JoinPoint to determine whether Advice needs to be executed or not * ***JoinPoint***: a point during the execution of a program, such as the execution of a method or the handling of an exception  ****Q37. What is**** Weaving****?**** According to the [official docs](https://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html), weaving is a process that links aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime. |