Spring is a Dependency Injection framework.

Dependency: If a class is dependent on another class. All the dependents of a class, that is whatever the class needs to run are called dependencies.
Tight coupling: Not considered to be good. Tight coupling is we are hard-coding the dependencies rather than providing the abstraction. Line SortAlgorithm sort = new BubbleSortAlgorithm() is hard-coding the instance. If we need to change BubbleSort to

Quicksort then we need to change the instantiation. Instead of that, can be done even better by directly sending the dependencies?

Common Spring Terminology:

1. Beans
2. Autowiring
3. Dependency Injection
4. Inversion of Control
5. IOC Container
6. Application Context

1. Beans: Different objects maintained by the spring framework
2. Autowiring: Find the correct dependency
3. Dependency Injection: identified dependency is injected
4. Inversion of Control: Change the control from class to spring framework.
5. IOC Container: An application context is an IOC Container. It is where all the beans are created and managed. A most important part of the spring framework. Whole main functionalities of spring happen here.

Spring injection can happen in 2 ways:

1. Constructor injection
    a. When to use: All mandatory dependencies are autowired using constructor injection - earlier version of spring
2. Setter injection - private setter needs to be present and Autowired declared on top
    a. Or if the Autowired is declared on top of a private variable, that should be enuf
    b. Optional dependencies: setter injection - earlier version of springs
    c. If you put Autowired, but there is no bean of Autowired present at all, then the context will not inject the bean.
    d. So there is not a need for constructor injection.
    e. Simple declare @Autowired on top of an injectable variable
    f. Lines are very thin between constructor vs setter injection

Spring is built very modularly. Core Container module contains all the core components.

1. Data Access/Integration - JDBC, ORM, OXM, JMS
    a. JDBC - spring jdbc is much easier
    b. ORM - Object Relational Mapping - Hibernate and MyBetis
    c. JMS - Talk other apps using queue
    d. OXM - object to XML
    e. Transaction - Transaction management is also taken care of
2. Web framework support:
    a. Struts
    b. Spring MVC
    c. Websocket
    d. Servlet
3. Cross-cutting concerns:
    a. Things applicable for more than one layer
    b. Using unit testing we can do cross-cutting tests.

Spring modules:

1. Spring Boot: Quickly build applications.
2. Spring Cloud: build cloud-native apps.
3. Spring Data: Consistent Data Access - for connecting to SQL, NoSql DBs.
4. Spring Integration: Application Integration.
5. Spring Batch: Batch applications
6. Spring Security: Protect your applications
7. Spring HATEOS: develop HATEOAS comparable services
    a. In rest, we need to pass more data to the end-user
    b. That is HATEOS

Greate features with Spring:

1. Easy way to write unit tests - by using JUnit or mockito
2. No plumbing code - unnecessary code needed by the application
3. Architectural flexibility - Even spring has MVC, it supports other MVC, such as struts
    a. You can integrate other MVC, JDBC or any other framework with Spring
4. Staying Current - kept on upgrading till date

Spring in-depth:

1. Autowiring types and Qualifiers
2. Bean Scope and Life Cycle
3. IOC Container and Application Context
4. XML & Java Application Contexts
5. Component Scan
6. External Properties
7. Container and Dependency Injection(CDI)

- **Autowiring types and Qualifiers**
    - Autowire can happen with name and type
        - @Autowired private SortAlgorithm bubbleSortAlgorithm;
        - The above code will autowire bubble sort instead of quick sort
        - @Primary @Component would declare a class as primary
        - If two classes are declared as Primary then clash and error
        - Let's say @Primary @Component on BubbleSortAlgorithm and @Autowired private SortAlgorithm quickSortAlgorithm; which one is preferred? - Ans: BubbleSortAlgorithm. @Primary is given more importance than the quick sort algorithm.
    - Autowiring with @Qualifier
        - Declaring a qualifier:
            - @Component @Qualifier("quick") public class QuickSortAlgorithm
            - @Component @Qualifier("bubble") public class BubbleSortAlgorithm
        - Using a qualifier
            - @Autowired @Qualifier("bubble") private SortAlgorithm sortAlgorithm;

- **Bean Scope and Life Cycle**
    - Singleton beans: By default any bean in context is singleton. One instance per serving context.
    - Prototype beans: New bean whenever requested
    - Request: One bean per HTTP request. Used in web app. One flow of a request from browser to server and response returned to browser. When we have scope of request, one per request.

- Session: one bean per HTTP session. Used in web app. What is HTTP session?

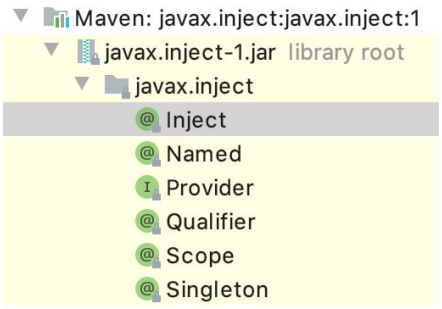
- **Complex Scope Scenarios of a Spring Bean - Mix prototype and singleton**
  - Scenario: Bean is of scope singleton, but the dependency is of the scope prototype. What happens in this scenario?
    - Since the bean in the outer scope is a singleton, even though the dependency bean is of type prototype, only one bean gets created and only one instance of dependency bean exists.
    - If we really want a different instance of the dependency(let's say JDBC connection) and not use the same instance again(as happening in this scenario), then we need to configure something called as proxy.
      - Add additional attribute to the annotation
      - @Scope(value=ConfigurableBeanFactory.SCOPE_PROTOTYPE , proxyMode = ScopedProxyMode.TARGET_CLASS) - add this on the outer class so that the inner dependencies can be gathered based on their individual scope. The proxy makes sure you get a new dependency object if it is a prototype.
      - The number of objects created should be kept at a minimum. Not very ideal to create more number of objects as it could be a problem for garbage collection/
  - Scenario2: Bean is of scope prototype, but the dependency is of the scope singleton. What happens in this scenario?
    - New bean gets created whenever asked for, but the internal bean which is singleton stays as such.

- **Difference between spring singleton and GOF singleton(design pattern singleton)**
  - According to Spring's definition of singleton - there will be one instance per spring context. There can be multiple contexts in spring JVM. so each context can have one singleton object. But GOF defines that in such instances there should be only one instance of that object.

- **Using ComponentScan to scan for beans**
  - @SpringBootApplication - by defaults picks up all the classes in the package in which the annotation is declared and also classes from the sub-packages within the package.

- So spring won't be able to find components in other packages. To avoid that we use ComponentScan
- **@SpringBootApplication @ComponentScan("com.other.package.name") solves it**

- **<u>The lifecycle of a bean - @PostConstruct and @PreDestroy</u>**
  - **@PostConstruct - As soon as the bean is created and initialized with the dependencies, the post construct method is called.**
  - If you want to do something to the value of a bean as soon as it is constructed, we can use the PostConstruct method.
  - **@PreDestroy -** Just before a bean is removed from the context.

- **<u>Contexts and Dependency Injection(CDI)</u>**
  - CDI tries to standardize, by creating an interface, of the annotations
    - @Inject - @Autowired
    - @Named - @Component & @Qualifier
    - @Singleton - Defines the scope of Singleton
  - CDI implemented frameworks would provide the functionality
  - Whether you use the above annotations(on the left) or Springs annotations(on the right), either way, it is going to create the bean.
  - A comparable standardization is JPA and Hibernate. JPA is an interface, and hibernate provides the implementation.
  - 

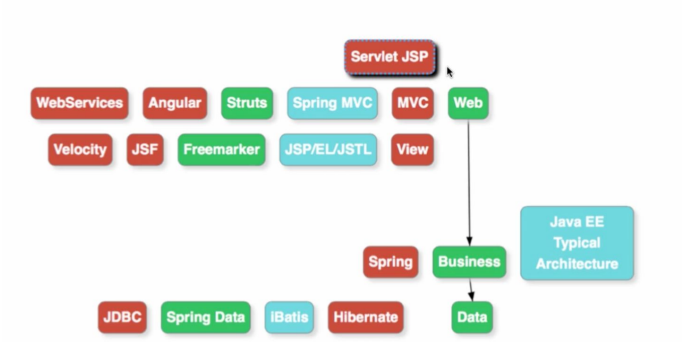
  - You can either use CDI or Spring annotations is up to you.

- **<u>Spring vs Spring Boot</u>**
  - Removing **@SpringBootApplication** and the dependency `spring-boot-starter-pack` from the pom.xml, will remove the spring boot dependencies.
  - Replace the annotation @SpringBootApplication with 2 annotations
    - @Configuration and @ComponentScan
  - We also need to create application context

    - And also slf4j is removed when Springboot is removed.
    - We need to add **`spring-core`, `spring-context`, and `slf4j-api`** dependencies after removing spring-boot-starter dependency

- **Add logback and close Spring Context**
    - Spring context implements closeable, so we can directly use try block to initialize the context, in-case the try block fails, the context is automatically closed.

- **IOC Container vs Application Context vs Bean Factory - are all the different terms used to describe Application context**
    - IOC Container - The control is shifting from an object about object creation to the framework which is injecting dependency. Inject all the dependencies it needs and inject them. The IOC container controls all the beans. IOC container is a generic concept. It is not framework specific. Whichever part does the IOC part is called the IOC Container. There are 2 implementations of the IOC container.
        - 2. Application Context
        - 1. Bean Factory - this is the main.
    - Spring recommends using Application Context 90% of the time.
    - Application Context = BeanFactory++
        - **Application context = BF + Spring AOP features + I18n capabilities + WebApplicationContext for web applications etc**
    - Welcome Service -

- **@Component @Service @Controller @Repository**
    - 

    - Component Annotations - All are equivalent
        - @Component - It is very generic. Generic Component
        - @Repository - encapsulating storage, retrieval, and search behavior typically from a relational database

        - @Service - Business service facade
        - @Controller - Use it as a component at the web layer. A controller in MVC pattern.

- **<u>Read values from an external properties file:</u>**
    - Create application.properties in the service
    - Add the property to application.properties
    - some.property.value=10
    - application.properties is automatically loaded by spring-boot
    - Otherwise, use annotation below @ComponentScan to load the property file
        - @PropertySource(“classpath:app.properties”)
        - The class into which this property is injected also needs to be a bean