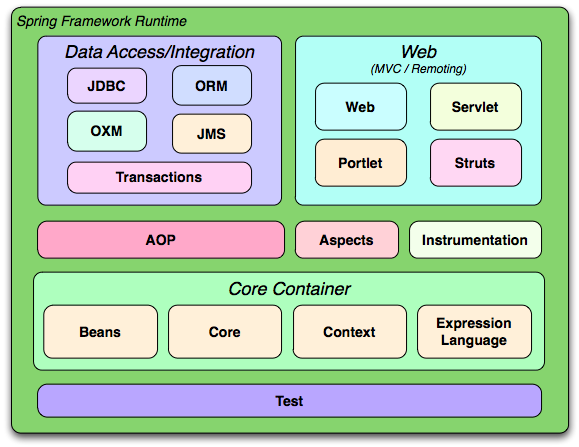
* What is spring framework?

Spring framework is a powerful lightweight application development framework used for Enterprise Java(J2EE).

The basic version of spring framework is around 2MB, written by Rod Johnson, released in 2003 under apache liscence.

The spring framework consists of feature organised into about 20 modules. The modules are groupsed into Core container, Data access and Integration, AOP, Web and Test.



Core Continer:

The Core Container consists of Beans, Core, Context and Expression language.

The core and bean provides the fundamental parts of the spring framework including the dependency injection and Inversion of control.

* What is Spring inversion of control (IOC) and dependency Injection (DI)?

<https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch03s04.html>

Inversion of control is a programming technique in which the object couping is bound at run time (“Loosly coupled”) and typically not known at compile time. This binding process is achieved through dependency injection.

The act of connecting objects with other objects, or “injecting” objects into other objects, is done by an assembler rather than by the objects themselves.

Dependency Injection in Spring can be done through constructors, setters or fields.

1. ***Constructor-Based Dependency Injection***.
2. **public** **class** Foo {
3. **private** Bar bar;
4. **private** Baz baz;
6. **public** Foo(Bar bar, Baz baz) {
7. **this**.bar = bar;
8. **this**.baz = baz;
9. }
11. **public** Bar getBar() {
12. **return** bar;
13. }
14. **public** **void** setBar(Bar bar) {
15. **this**.bar = bar;
16. }
17. **public** Baz getBaz() {
18. **return** baz;
19. }
20. **public** **void** setBaz(Baz baz) {
21. **this**.baz = baz;
22. }
23. }

No potential ambiguity exists, assuming that Bar and Baz classes are not related by inheritance. Thus the following configuration works fine, and you do not need to specify the constructor argument indexes and/or types explicitly in the <constructor-arg/> element.

1. **<beans>**
2. **<bean** id="foo" class="x.y.Foo"**>**
3. **<constructor-arg** ref="bar"**/>**
4. **<constructor-arg** ref="baz"**/>**
5. **</bean>**
7. **<bean** id="bar" class="x.y.Bar"**/>**
8. **<bean** id="baz" class="x.y.Baz"**/>**
10. **</beans>**

**Constructor argument type matching**

1. **public** **class** ExampleBean {
2. **private** **int** years;
3. **private** String ultimateAnswer;
5. **public** ExampleBean(**int** years, String ultimateAnswer) {
6. **this**.years = years;
7. **this**.ultimateAnswer = ultimateAnswer;
8. }
9. **public** **int** getYears() {
10. **return** years;
11. }
12. **public** **void** setYears(**int** years) {
13. **this**.years = years;
14. }
15. **public** String getUltimateAnswer() {
16. **return** ultimateAnswer;
17. }
18. **public** **void** setUltimateAnswer(String ultimateAnswer) {
19. **this**.ultimateAnswer = ultimateAnswer;
20. }
21. }
22. **<bean** id="exampleBean" class="examples.ExampleBean"**>**
23. **<constructor-arg** type="int" value="7500000"**/>**
24. **<constructor-arg** type="java.lang.String" value="42"**/>**
25. **</bean>**

**Constructor argument index**

* **<bean** id="exampleBean" class="examples.ExampleBean"**>**
* **<constructor-arg** index="0" value="7500000"**/>**
* **<constructor-arg** index="1" value="42"**/>**
* **</bean>**

1. ***Setter-based dependency injection***

Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

1. **public** **class** TextEditor {
2. **private** SpellChecker spellChecker;
4. // a setter method to inject the dependency.
5. **public** **void** setSpellChecker(SpellChecker spellChecker) {
6. System.out.println("Inside setSpellChecker." );
7. **this**.spellChecker = spellChecker;
8. }
9. // a getter method to return spellChecker
10. **public** SpellChecker getSpellChecker() {
11. **return** spellChecker;
12. }
13. **public** **void** spellCheck() {
14. spellChecker.checkSpelling();
15. }
16. }
17. **public** **class** SpellChecker {
18. **public** SpellChecker(){
19. System.out.println("Inside SpellChecker constructor." );
20. }
21. **public** **void** checkSpelling() {
22. System.out.println("Inside checkSpelling." );
23. }
24. }

XML based setter injection.

1. <!-- Definition for textEditor bean -->
2. **<bean** id = "textEditor" class = "com.tutorialspoint.TextEditor"**>**
3. **<property** name = "spellChecker" ref = "spellChecker"**/>**
4. **</bean>**
6. <!-- Definition for spellChecker bean -->
7. **<bean** id = "spellChecker" class = "com.tutorialspoint.SpellChecker"**></bean>**
8. ***Autowiring collaborators***

The Spring container can autowire relationships between collaborating beans. You can allow Spring to resolve collaborators (other beans) automatically for your bean by inspecting the contents of the ApplicationContext. Autowiring has the following advantages:

* Autowiring can significantly reduce the need to specify properties or constructor arguments.
* Autowiring can update a configuration as your objects evolve.

Autowiring modes

| **Mode** | **Explanation** |
| --- | --- |
| no | (Default) No autowiring. Bean references must be defined via a ref element. Changing the default setting is not  recommended for larger deployments, because specifying collaborators explicitly gives greater control and  clarity. To some extent, it documents the structure of a system. |
| byName | Autowiring by property name. Spring looks for a bean with the same name as the property that needs to  be autowired. For example, if a bean definition is set to autowire by name, and it contains a *master*  property (that is, it has a *setMaster(..)* method), Spring looks for a bean definition named master, and uses  it to set the property. |
| byType | Allows a property to be autowired if exactly one bean of the property type exists in the container.  If more than one exists, a fatal exception is thrown, which indicates that you may not use *byType* autowiring  for that bean. If there are no matching beans, nothing happens; the property is not set. If this is not desirable,  setting the dependency-check="objects" attribute value specifies that an error should be thrown in this case. |
| constructor | Analogous to *byType*, but applies to constructor arguments. If there is not exactly one bean of the constructor  argument type in the container, a fatal error is raised. |
| autodetect | Chooses *constructor* or *byType* through introspection of the bean class. If a default constructor is found,  the *byType* mode is applied. |

Consider the limitations and disadvantages of autowiring:

* Explicit dependencies in property and constructor-arg settings always override autowiring. You cannot autowire so-called simple properties such as primitives, Strings, and Classes (and arrays of such simple properties). This limitation is by-design.
* Autowiring is less exact than explicit wiring. Although, as noted in the above table, Spring is careful to avoid guessing in case of ambiguity that might have unexpected results, the relationships between your Spring-managed objects are no longer documented explicitly.
* Wiring information may not be available to tools that may generate documentation from a Spring container.
* Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired. For arrays, collections, or Maps, this is not necessarily a problem. However for dependencies that expect a single value, this ambiguity is not arbitrarily resolved. If no unique bean definition is available, an exception is thrown.

1. Autowiring (no)
2. **<bean** id="department" class="guru.springframework.autowiringdemo.Department"**>**
3. **<property** name="deptName" value="Information Technology" **/>**
4. **</bean>**
5. **<bean** id="employee" class="guru.springframework.autowiringdemo.Employee"**></bean>**
6. Autowiring ‘byName’:
7. **<bean** id="department" class="guru.springframework.autowiringdemo.Department"**>**
8. **<property** name="deptName" value="Information Technology" **/>**
9. **</bean>**
10. **<bean** id="employee" class="guru.springframework.autowiringdemo.Employee" autowire="byName"**></bean>**
11. Autowiring ‘byType’:
12. **<bean** id="department" class="guru.springframework.autowiringdemo.Department"**>**
13. **<property** name="deptName" value="Information Technology" **/>**
14. **</bean>**
15. **<bean** id="employee" class="guru.springframework.autowiringdemo.Employee" autowire="byType"**></bean>**
16. Autowiring ‘constructor’
17. **<bean** id="department" class="guru.springframework.autowiringdemo.Department"**>**
18. **<property** name="deptName" value="Information Technology" **/>**
19. **</bean>**
20. **<bean** id="employee" class="guru.springframework.autowiringdemo.Employee" autowire="constructor"**></bean>**

5. Autowiring ‘autodetect’

1. **<bean** id="department" class="guru.springframework.autowiringdemo.Department"**>**
2. **<property** name="deptName" value="Information Technology" **/>**
3. **</bean>**
4. **<bean** id="employee" class="guru.springframework.autowiringdemo.Employee" autowire="autodetect"**></bean>**

**@Autowired Annotation**

In Spring, you can use @Autowired annotation to auto-wire bean on the setter method, constructor, or a field. Moreover, it can autowire the property in a particular bean. We must first enable the annotation using below configuration in the configuration file.

1. @Configuration
2. @ComponentScan("guru.springframework.autowiringdemo")
3. **public** **class** AppConfig {}

As an alternative, we can use below XML-based configuration in Spring:

**<context:annotation-config** **/>**

**@Autowired on Properties**

In the below example, when the annotation is directly used on properties, Spring looks for and injects **Department** when **Employee** is created. This is how it eliminates the need for getters and setters.

1. **import** org.springframework.stereotype.Component;
3. @Component
4. **public** **class** Department {
5. **private** String deptName;
7. **public** String getDeptName() {
8. **return** deptName;
9. }
11. **public** **void** setDeptName(String deptName) {
12. **this**.deptName = deptName;
13. }
14. }
15. **import** org.springframework.beans.factory.annotation.Autowired;
16. **public** **class** Employee {
17. **private** **int** eid;
18. **private** String ename;
19. @Autowired   // Autowired annotation is used.
20. **private** Department department;
21. **public** **int** getEid() {
22. **return** eid;
23. }
24. **public** **void** setEid(**int** eid) {
25. **this**.eid = eid;
26. }
27. **public** String getEname() {
28. **return** ename;
29. }
30. **public** **void** setEname(String ename) {
31. **this**.ename = ename;
32. }
33. **public** **void** showEployeeDetails(){
34. System.out.println("Employee Id : " + eid);
35. System.out.println("Employee Name : " + ename);
36. department.setDeptName("Information Technology");
37. System.out.println("Department : " + department.getDeptName());
38. }
39. }

***@Autowired on Setters***

1. **import** org.springframework.beans.factory.annotation.Autowired;
2. **import** org.springframework.stereotype.Component;
3. @Component
4. **public** **class** Employee {
5. **private** **int** eid;
6. **private** String ename;
7. **private** Department department;
8. **public** **int** getEid() {
9. **return** eid;
10. }
11. **public** **void** setEid(**int** eid) {
12. **this**.eid = eid;
13. }
14. **public** String getEname() {
15. **return** ename;
16. }
17. **public** **void** setEname(String ename) {
18. **this**.ename = ename;
19. }
20. **public** Department getDepartment() {
21. **return** department;
22. }
23. @Autowired
24. **public** **void** setDepartment(Department department) {
25. **this**.department = department;
26. }
27. **public** **void** showEployeeDetails(){
28. System.out.println("Employee Id : " + eid);
29. System.out.println("Employee Name : " + ename);
30. department.setDeptName("Information Technology");
31. System.out.println("Department : " + department.getDeptName());
32. }
33. }

***@Autowired on Constructors***

1. **import** org.springframework.beans.factory.annotation.Autowired;
2. **import** org.springframework.stereotype.Component;
3. @Component
4. **public** **class** Employee {
5. **private** **int** eid;
6. **private** String ename;
7. **private** Department department;
8. @Autowired
9. **public** EmployeeBean(DepartmentBean deptBean) {
10. System.out.println("\*\*\* Autowiring by using @Autowire annotation on constructor \*\*\*");
11. **this**.deptBean = deptBean;
12. }
13. **public** **int** getEid() {
14. **return** eid;
15. }
16. **public** **void** setEid(**int** eid) {
17. **this**.eid = eid;
18. }
19. **public** String getEname() {
20. **return** ename;
21. }
22. **public** **void** setEname(String ename) {
23. **this**.ename = ename;
24. }
25. **public** Department getDepartment() {
26. **return** department;
27. }
28. **public** **void** setDepartment(Department department) {
29. **this**.department = department;
30. }
31. **public** **void** showEployeeDetails(){
32. System.out.println("Employee Id : " + eid);
33. System.out.println("Employee Name : " + ename);
34. department.setDeptName("Information Technology");
35. System.out.println("Department : " + department.getDeptName());
36. }
37. }

* What is Bean 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. These beans are created with the configuration metadata that we supply to the container.

@Bean

@Bean is a method-level annotation and a direct analog of the XML <bean/> element. The annotation supports most of the attributes offered by <bean/>, such as: init-method, destroy-method, autowiring, lazy-init, dependency-check, depends-on and scope.

To declare a bean, simply annotate a method with the @Bean annotation. When JavaConfig encounters such a method, it will execute that method and register the return value as a bean within a BeanFactory. By default, the bean name will be the same as the method name. The following is a simple example of a @Bean method declaration:

1. @Configuration
2. **public** **class** AppConfig {
3. @Bean
4. **public** TransferService transferService() {
5. **return** **new** TransferServiceImpl();
6. }
7. }

For comparison sake, the configuration above is exactly equivalent to the following Spring XML:

1. **<beans>**
2. **<bean** name="transferService" class="com.acme.TransferServiceImpl"**/>**
3. **</beans>**

Using @Bean initMethodName / destroyMethodName attributes

1. **public** **class** Foo {
2. **public** **void** init() {
3. // initialization logic
4. }
5. }
6. **public** **class** Bar {
7. **public** **void** cleanup() {
8. // destruction logic
9. }
10. }
11. @Configuration
12. **public** **class** AppConfig {
13. @Bean(initMethodName="init")
14. **public** Foo foo() {
15. **return** **new** Foo();
16. }
17. @Bean(destroyMethodName="cleanup")
18. **public** Bar bar() {
19. **return** **new** Bar();
20. }
21. }

* What are different types of bean scope in Spring?

There are 5 types of bean scopes supported :

1. singleton – Return a single bean instance per Spring IoC container
2. prototype – Return a new bean instance each time when requested
3. request – Return a single bean instance per HTTP request. \*
4. session – Return a single bean instance per HTTP session. \*
5. globalSession – Return a single bean instance per global HTTP session. \*

***Singleton vs Prototype***

Singleton example

1. **import** org.springframework.context.ApplicationContext;
2. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
4. **import** com.mkyong.customer.services.CustomerService;
6. **public** **class** App
7. {
8. **public** **static** **void** main( String[] args )
9. {
10. ApplicationContext context =
11. **new** ClassPathXmlApplicationContext(**new** String[] {"Spring-Customer.xml"});
13. CustomerService custA = (CustomerService)context.getBean("customerService");
14. custA.setMessage("Message by custA");
15. System.out.println("Message : " + custA.getMessage());
17. //retrieve it again
18. CustomerService custB = (CustomerService)context.getBean("customerService");
19. System.out.println("Message : " + custB.getMessage());
20. }
21. }
22. **<bean** id="customerService"
23. class="com.mkyong.customer.services.CustomerService" **/>**

Output:

1. Message : Message by custA
2. Message : Message by custA

2. Prototype example: If you want a new ‘customerService’ bean instance, every time you call it, use prototype instead.

1. **<bean** id="customerService" class="com.mkyong.customer.services.CustomerService"
2. scope="prototype"**/>**

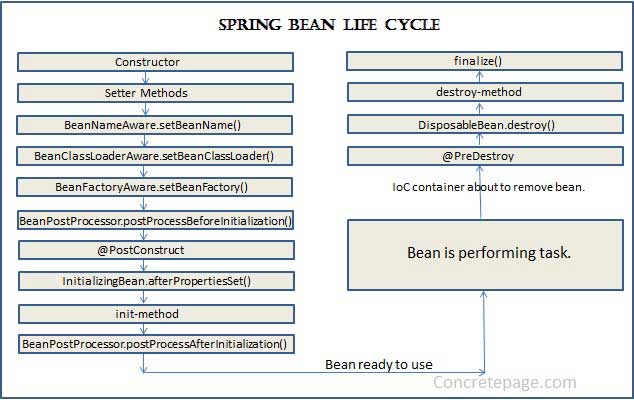
Output:

1. Message : Message by custA
2. Message : null

* What is spring bean life cycle in java?

<https://dzone.com/articles/spring-bean-lifecycle>

The Spring IoC (Inversion of Control) container manages Spring beans. A “Spring bean” is just a Spring-managed instantiation of a Java class.



The Spring IoC container is responsible for instantiating, initializing, and wiring beans. The container also manages the life cycle of beans.

As you can see in Part 1 of the preceding figure, the container instantiates a bean by calling its constructor and then populates its properties.

This is followed by several calls to the bean until the bean is in the ready state.

Similarly, as shown in Part 2, when the container shuts down, the container calls the bean to enable it to perform any required tasks before the bean is destroyed.

When a bean is initialized it might require to perform some action before it can come into a usable state(State in which application can use it) and when a bean is getting destroyed, there might be some cleanup activity required for the given bean. These activities are known as bean Lifecycle.

Standard bean lifecycle interfaces & the standard order of execution are given below.

1- IoC container will look for the configuration metadata of given Bean.

2- Once found, the container will create the instance of Bean(Using reflection API).

3- After instance, creation dependency will be injected(DI).

**If Bean Class implements any of the below-highlighted interfaces then the corresponding method will be invoked in below order(Point 4 – 13).**

4- **setBeanName** method of BeanNameAware Interface. It sets the name of the bean in the bean factory that created this bean.

5- **setBeanClassLoader** method of BeanClassLoaderAware Interface. Callback that supplies the bean to a bean instance.

6- **setBeanFactory** method of BeanFactoryAware Interface. Callback that supplies the owning factory to a bean instance.

**Below method execution will be applicable when running in an application context. (Points 7 – 11)**

7- **setResourceLoader** method of ResourceLoaderAware Interface. It set the ResourceLoader that this object runs in.

8- **setApplicationEventPublisher** method of ApplicationEventPublisherAware Interface. Set the ApplicationEventPublisher that this object runs in.

9- **setMessageSource** method of MessageSourceAware Interface. Set the MessageSource that this object runs in.

10- **setApplicationContext** method of ApplicationContextAware Interface. Set the ApplicationContext that this object runs in.

11- **setServletContext** method of ServletContextAware Interface. Set the ServletContext that this object runs in.

12- **postProcessBeforeInitialization** method of BeanPostProcessor Interface. Apply this BeanPostProcessor to the given new bean instance before any bean initialization callbacks.

13- **afterPropertiesSet** method of InitializingBean Interface. Invoked by a BeanFactory after it has set all bean properties supplied.

In case Bean class has custom init method defined(via init-method attribute)

14- **Custom** init method will be invoked.

15- **postProcessAfterInitialization** methods of BeanPostProcessors. Apply this BeanPostProcessor to the given new bean instance after any bean initialization callbacks

When Bean Factory is getting shut down following lifecycle methods will be executed.

1- DisposableBean’s destroy method. Invoked by a BeanFactory on the destruction of a singleton.

2- Custome destroy method will be executed if there is any defined via destroy-method attributes

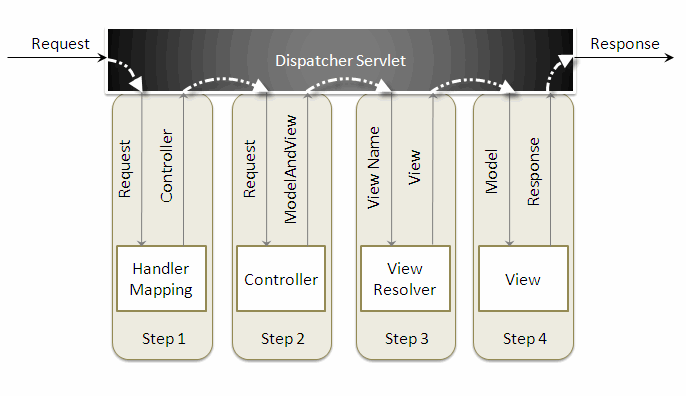
* What is spring MVC.

<https://dzone.com/tutorials/java/spring/spring-mvc-tutorial-1.html>

Spring MVC helps in building flexible and loosely coupled web applications. The Model-view-controller design pattern helps in seperating the **business logic, presentation logic, and navigation logic**.

* 1. Models are responsible for encapsulating the application data.
  2. The Views render a response to the user with the help of the model object.
  3. Controllers are responsible for receiving the request from the user and calling the back-end services.

The figure below shows the flow of requests in the Spring MVC Framework.



When a request is sent to the Spring MVC Framework the following sequence of events happen.

1. The **DispatcherServlet** first receives the request.
2. The DispatcherServlet consults the **HandlerMapping** and invokes the **Controller** associated with the request.
3. The **Controller** processes the request by calling the appropriate service methods and returns a **ModelAndView** object to the DispatcherServlet. The ModelAndView object contains the **model data** and the **view name**.
4. The DispatcherServlet sends the view name to a **ViewResolver** to find the actual View to invoke.
5. Now, the DispatcherServlet will pass the model object to the View to render the result.
6. The View, with the help of the model data, will render the result back to the user.

* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.