1. **Inversion Of Control**: Giving control of Object Creation to Spring so that Object can be created and injected into another class. For Example, Instead of creating Address object using new keyword Spring IOC Container creates object of address type and injects into student object. So here we can see that the control is inverted from you creating the object to Spring creating the object.
2. **Dependency Injection**: It is a technique in which an object receives other object that it depends on.  
   For example: In a Student class we have a dependency on address, so whenever we try to create object of Student class, the address should be created and should be injected in student object.  
   There are 2 ways in which a dependency can be injected.  
   a. Setter Injection (use setter method for a field)  
   b. Constructor Injection (use Constructor to set values for a field)
3. **Spring IOC Container**:

Spring IoC container performs:-

1. It creates the object.
2. Hold the Object in memory.
3. And Inject them in another object (Dependency Injection)

It must know following things before performing its task.  
a. Beans or POJO classes it has to manage.  
b. Configuration Files. Generally XML Configurations. In configuration files we tell that which bean is dependent on other things.

1. **Application Context:** It is an interface which represents Spring IoC Container. It also implements BeanFactory. Since it is an interface, we need to create sub class of an object. Some of the important subclasses are :-
2. ClasspathXMLApplicationContext
3. AnnotationConfigApplicationContext
4. FileSystemXMLApplicationContext

ClasspathXMLApplicationContext searches for XML configuration using JAVA classpath.  
AnnotationConfigApplicationContext is used in annotations. Searches for desired Bean for the used annotation.  
FileSystemXMLApplication searches for config file from file system.  


1. **Bean Factory v/s application Context**

|  |  |  |
| --- | --- | --- |
|  | Bean Factory | Application Context |
| Type Of Applications | Suitable to build stand alone applications | Suitable to build web applications integrated with AOP & ORM. |
| Functionality | Fundamental container that provides basic functionality | Advanced container that extends Bean Factory that provides basic + advanced functionality. |
| Support for Annotation | Does not support annotation. In Bean Autowiring, we need to configure the properties in XML file only. | It supports Annotation based configuration in Bean Autowiring. |
| Memory | Requires less memory as it provides basic features | Requires more memory. |

Bean Factory is deprecated from Spring 3.0.

1. **LifeCycle Of Bean**

Bean life cycle is managed by the spring container. When we run the program then, 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. And finally, the bean is destroyed when the spring container is closed. Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the init() method and the destroy() method.

1. public void **init()** : used for initializing code, Loading config, connecting db etc.
2. public void **destroy() :** used for writing code clean up.

  
Configuration Techniques  
a. XML (init-method attribute & destroy-method attribute)  
b. Spring Interface(Initialization Bean Interface & Disposable Bean Interface)  
c. Annotation(@PreConstruct & @PreDestroy) -> @PreConstruct will provide init functionality & @PreDestroy will provide destroy functionality.

@EnableAutoConfiguration

1. Enables the auto configurations of Application Context by scanning classpath components.

@SpringBootConfiguration

1. It is class level annotation
2. It is implementation of @Configuration
3. Main difference b/w @Configuration & @SpringBootConfiguration is that, @SBC allow configurations to be loaded automatically.

@ComponentScan

1. Searches/Scans for components such as @Controller, @Service & @Repository.

**POJO v/s Java Bean v/s Spring Bean**

**POJO** = Plain Old Java Object. Object of any class is POJO. E.g. ABC abc = new ABC();  
Here ‘abc’ is POJO.

**Java Bean** = Also called Enterprise Java Bean. EJB is a POJO with few restrictions.

1. Restriction 1: Class should have default/ no-arg constructor.
2. Restriction 2: Every variable must have a getter & setter.
3. Restriction 3: The class must implement serializable interface.

**Spring Bean** = Any POJO maintained by Spring IOC container is called Spring Bean or Simply Bean. Application Context is implementation of Spring IOC Container. (All code of Application Context is written inside Application Context, it’s sub-classes and it’s sub-interfaces.)

**@BEAN**

It tells that method will produce a Bean. It is used in conjunction with @Configuration. E.g.

public class MyCustomBean{

@override  
 public String toString() {return “MyCustomBean”;}

}

@Configuration   
public class MyApp {

@Bean  
 public MyCustomBean getCustomerBean() {return new MyCustomerBean();}

}  
Use case for @Bean : If we have lot of custom business logic or If we want to instantiate bean for 3rd party

**@Configuration** = Indicates that a class has some methods in which beans are defined a.k.a Bean Definition Methods. So Spring Container can process the class & generate Spring Beans to be used in the application.

**@Primary v/s @Qualifier**

**@Qualifier** -> Autowire a specific bean among same type of Bean.

**@Primary ->** Used to give high preference to the specific beanamong multiple beans of same type to inject to a bean.

@Qualifier has higher priority then @Primary

@Component @Primary  
class QuickSort implements SortingAlogorithm {}

@Component  
class BubbleSort implements SortingAlgorithm {}

@Component @Qualifier(“RadixSortQualifier”)  
class RadixSort implements SortingAlgorithm {}

Just give me preferred algorithm (marked with @Primary)

@Component  
class ComplexAlgorithm   
@Autowired  
private SortingAlgorithm algorithm;

@Component  
class AnotherComplexAlgorithm  
@Autowired @Qualifier(“RadixSortQualifier”)  
private SortingAlgorithm iWantToUseRadixSortOnly;

**REST API**

@RestController

@RequestMapping("/college")

public class StudentController {  
@GetMapping("/student")

public Student getStudent() {

return new Student("Ramesh", "Fadatare");

}

}

**Rest API using Query Parameters**

Sample url = <http://localhost:8080/query?firstName=Shivam&lastName=Khandelwal>

@GetMapping("/student/query")

public Student studentQueryParam(

@RequestParam(name = "firstName") String firstName,

@RequestParam(name = "lastName") String lastName) {

return new Student(firstName, lastName);

}

**Rest API using path variables**

Sample url = <http://localhost:8080/student/shivam/khandelwal/>

@GetMapping("/student/{firstName}/{lastName}/")

public Student studentPathVariable(

@PathVariable("firstName") String firstName,

@PathVariable("lastName") String lastName) {

return new Student(firstName, lastName);}

}

**Rest API returning ResponseEntity**

@GetMapping("/getAllUsers")

public ResponseEntity<List<User>> getAllUsers() {

List <User> user;

user = userService.getAllUser();

return new ResponseEntity<>(user,HttpStatus.OK);

}

**Adding Swagger**

1. Getting Swagger 2 Spring Dependency  
2. Enabling swagger in our code.  
3. Configuring Swagger.  
4. Adding details as annotations to API.

**Getting Swagger 2 Spring Dependency**Add springfox-swagger2 & springfox-swagger-ui maven dependency.

**Enabling swagger in our code.**  
We need to add @EnableSwagger2 annotation in Main class.