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, its sub-classes and its sub-interfaces.)

***@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.*

**@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 : For writing custom business logic or for instantiating bean for 3rd party.

**@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 than @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;

**Internal Working of Hashmap**

A HashMap in Java is an implementation of a hash table data structure. It maps keys to values, allowing for efficient lookups based on the keys.

The internal working of a HashMap in Java can be understood by understanding the following components:

**Hash Function**: The hash function takes a key as input and returns an integer (the hash code) that is used to determine the index of the bucket in which the key-value pair will be stored. The hash code is calculated using the key's hashCode() method.

**Buckets**: The HashMap stores key-value pairs in "buckets". Each bucket is an array of Entry objects that store a key-value pair.

**Hash Code Collision**: If two keys have the same hash code, they are said to have collided. The HashMap handles collisions by chaining the entries in the same bucket using a linked list.

**Resizing**: The HashMap is implemented as an array of buckets, with the size of the array being a power of 2. The HashMap can be resized dynamically when the number of key-value pairs stored in the HashMap exceeds a certain load factor.

**get() and put() operations**: The get() operation takes a key as input and uses the hash function to calculate the hash code and find the index of the bucket. The linked list in the bucket is then searched for the key-value pair, and the value is returned if found. The put() operation is similar, but it also adds a new key-value pair to the HashMap if the key is not already present.

Overall, the HashMap in Java provides an efficient and flexible implementation of a hash table data structure, with the ability to handle hash code collisions and dynamic resizing.

Here's how the HashMap works in detail:

**Calculating the Hash Code**: When you put a key into the HashMap, the HashMap calculates the hash code of the key using the hashCode() method. The hash code is used to determine the index in the array where the value should be stored.

**Compression Function**: The hash code of the key is then passed through a compression function to determine the index in the array where the value should be stored. The compression function takes the hash code and computes a remainder when it's divided by the size of the array. This remainder is the index where the value should be stored.

**Handling Collisions**: In the unlikely event that two keys have the same hash code and therefore the same index in the array, a collision occurs. To handle collisions, the HashMap stores linked lists at each index in the array. When a collision occurs, the value is stored in the linked list at that index.

**Retrieving Values**: To retrieve a value from the HashMap, you provide the key. The HashMap calculates the hash code of the key and passes it through the compression function to determine the index in the array where the value is stored. It then searches the linked list at that index to find the value associated with the key.

In summary, the HashMap in Java uses a hash table to store key-value pairs, where the hash code of the key is used to determine the index in the array where the value should be stored. The compression function and linked lists are used to handle collisions and ensure efficient retrieval of values.

**@ExceptionHandler & @ControllerAdvice**

The @ExceptionHandler annotation in Spring Boot is used to define a method that should be called to handle exceptions that are thrown in your application. When an exception is thrown, Spring Boot will look for an @ExceptionHandler-annotated method that is capable of handling the exception type that was thrown. If it finds such a method, it will invoke that method to handle the exception.

@ControllerAdvice is a useful annotation in Spring Boot framework that allows you to define a single class to handle exceptions globally across all controllers in your application.  
It acts as a central exception handler for an entire application. We can use this annotation to provide a consistent way of handling exceptions and to write code that is clean, concise and reusable.

For example, you can define a class that extends ResponseEntityExceptionHandler and annotate it with @ControllerAdvice to handle exceptions in your controllers. In this class, you can define methods that will handle specific exceptions and return a custom response.

@ControllerAdvice  
public class CustomExceptionHandler extends ResponseEntityExceptionHandler {

@ExceptionHandler(value = {Exception.class})  
 public ResponseEntity<Object> handleAnyException(Exception ex, WebRequest request) {  
 // return custom response  
 }

@ExceptionHandler(value = {CustomException.class})  
 public ResponseEntity<Object> handleCustomException(CustomException ex, WebRequest request) {  
 // return custom response  
 }  
 }

**Java 8 Features**

a. Lambda Expressions: - similar to methods, but they do not need a name.

b. Stream API

c. Default Methods: - these are methods which are defined in interface and tagged as default. The implementation can be changed in child classes.

d. Static Methods: - these are methods which are defined in interface and keyword static. The implementation can't be changed in child classes.

e. Functional Interface

f. Optional Class

g. Method Referencing

**Functional Interfaces in Java**

They are the interfaces which have only one abstract method.   
It can have any number of static and default method.   
Example of functional Interface is Runnable, Comparable, Predicate, Consumer, Function, Supplier, etc.  
Functional interfaces were introduced in Java 8 in order to implement lambda expressions.

**We can create our own functional interface.**

To create our own functional interface, we need to follow the following steps:-

a. Create an interface.  
b. Annotate it with @FunctionalInterface  
c. define only one abstract method. (Abstract method does not have a body)  
d. (Optional) Create any static or default methods.

Example of Functional Interface:-

@FunctionalInterface  
public interface FunctionalInterface {

void singleAbstractMethod(); // Abstract Method  
 default void defaultMethod(){  
 System.out.println("This is default Method"); // This is optional

}

}

**Method Referencing In Java 8**

It is a special type of lambda expression in which we reference existing methods. It uses ::(double colon)

Types Of method reference

1. Static Method Reference.  
    If a lambda expression just call a static method of a class

(args) -> Class.staticMethod(args)

Shorthand -> if a lambda expression just call a static method of a class

Class::staticMethod

1. Instance Method Reference of a particular object

If a lambda expression just call a default method of an object

(args) -> obj.instanceMethod(args)

Shorthand -> if a lambda expression just call a default method of an object

obj::instanceMethod.

1. Instance Method Reference of an arbitrary object of a particular type.
2. Constructor Reference.

**Predicates**

It is pre-defined functional interface.  
Abstract Method -> public Boolean test(T t); Return type is always Boolean;  
Predicate Joining:- It is used when we want to combine multiple predicates to find single results.  
There are 3 ways to join a predicate.

1. And:- Predicate1.and(Predicate2).test("the value which we want to test")
2. Or:- Predicate1.or(Predicate2).test("the value which we want to test")
3. Negate:- Predicate1.negate().test("the value which we want to test")

**Functions**It is pre-defined functional interface.  
Abstract Method -> apply(T t);  
Working of functions is such a way that "If we give some input & perform some output. This output can be anything (not necessarily a Boolean value)".  
In Functions return type is not fixed. Hence we have to declare both input type and return type.  
Just like predicate joining, we have functional chaining. We can combine / chain multiple functions together with “andThen” and “compose”.  
 f1.andThen(f2).apply(Input); - first f1 then f2 // from left to right  
 f1.compose(f2).apply(Input) - first f2 then f1 // from right to left

Multiple functions can be chained together like :

f1.andThen(f2).andThen(f3).andThen(f4).apply(Inputs);

**Consumer**

It is predefined Functional Interface which never return anything (never supply), they just consume.

Abstract method: - accept.  
Example :- c1.andThen(c2).accept (Input); - first c1 then c2.

Multiple consumers can be chained together like: c1.andThen(c2).andThen(c3).andThen(c4).apply(Inputs);

**Supplier**

It is functional Interface which does not take any input but it will always supply an object.  
There is no chaining as no input is given. It is just opposite of Consumer.

Abstract method: - get(T t);

Example: current date.

**Stream API**

It is special Iterator class that allow processing of collection of Objects in a functional manner.

In simple words: Stream is sequence of Objects upon which we perform variety of operations to get desired output. Stream API is used to process these collections of Objects.

There are 2 types of streams:-

1. Parallel Stream(runs on multiple core & multiple thread; O/P is not predictable; High Performan)
2. Sequential Streams(Runs on single core & one thread; O/P is predictable; Poor performance)

**Optional in java 8**

Every Java Programmer is familiar with NullPointerException. It can crash your code. And it is very hard to avoid it without using too many null checks. So, to overcome this, Java 8 has introduced a new class Optional in java.util package. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable because the facts which were hidden are now visible to the developer.  
Some important methods in Optional Class are

a. *isPresent()* - Return true if there is a value present, otherwise false.

b. *get()* - If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.

c. *ofNullable(T value)* - if non-null, Returns an Optional describing the specified value otherwise returns an empty Optional.

**REST API**

@RestController

@RequestMapping("/college")

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

public Student getStudent() {

return new Student("Shivam", "Khandelwal");

}

}

**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);

}

***Add Post mapping as well.***

**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.