**Spring IoC (Inversion of Control)**

Spring IoC (Inversion of Control) Container is the core of Spring Framework. It creates the objects, configures and assembles their dependencies, manages their entire life cycle. The Container uses Dependency Injection (DI) to manage the components that make up the application. It gets the information about the objects from a configuration file (XML) or Java Code or Java Annotations and Java POJO class. These objects are called Beans. Since the Controlling of Java objects and their lifecycle is not done by the developers, hence the name Inversion of Control. The followings are some of the main features of Spring IoC,

* Creating Object for us,
* Managing our objects,
* Helping our application to be configurable,
* Managing dependencies

*To read more on Spring IoC (Inversion of Control) please refer to this article:*[*Spring IoC (Inversion of Control)*](https://www.geeksforgeeks.org/spring-ioc-container/)

**Spring Dependency Injection**

Dependency Injection is the main functionality provided by Spring IOC(Inversion of Control). The Spring-Core module is responsible for injecting dependencies through either Constructor or Setter methods. The design principle of Inversion of Control emphasizes keeping the Java classes independent of each other and the container frees them from object creation and maintenance. These classes, managed by Spring, must adhere to the standard definition of Java-Bean. Dependency Injection in Spring also ensures loose coupling between the classes. There are two types of Spring Dependency Injection.

1. Setter Dependency Injection (SDI)
2. Constructor Dependency Injection (CDI)

**A. Setter Dependency Injection (SDI)**

Setter Injection is the simpler of the two  Dependency Injection methods. In this, the Dependency Injection will be injected with the help of setter and/or getter methods. Now to set the Dependency Injection as Setter Injection in the bean, it is done through the bean-configuration file For this, the property to be set with the Setter Injection is declared under the **<property>** tag in the bean-config file.

**B. Constructor Dependency Injection (CDI)**

In Constructor Injection, the Dependency Injection will be injected with the help of constructors. Now to set the Dependency Injection as Constructor Dependency Injection in bean, it is done through the bean-configuration file. For this, the property to be set with the CDI is declared under the **<constructor-arg>** tag in the bean-config file.

Let us finally come up with cut-throat differences between them depicted via the table given below to get a better understanding as there persists always a dilemma if not understood to great depth.

| Spring IoC (Inversion of Control) | Spring Dependency Injection |
| --- | --- |
| Spring IoC Container is the core of Spring Framework. It creates the objects, configures and assembles their dependencies, manages their entire life cycle. | Spring Dependency injection is a way to inject the dependency of a framework component by the following ways of spring: Constructor Injection and Setter Injection |
| Spring helps in creating objects, managing objects, configurations, etc. because of IoC (Inversion of Control). | Spring framework helps in the creation of loosely-coupled applications because of Dependency Injection. |
| Spring IoC is achieved through Dependency Injection. | Dependency Injection is the method of providing the dependencies and Inversion of Control is the end result of Dependency Injection. |
| IoC is a design principle where the control flow of the program is inverted. | Dependency Injection is one of the subtypes of the IOC principle. |
| [Aspect-Oriented Programing](https://www.geeksforgeeks.org/aspect-oriented-programming-and-aop-in-spring-framework/) is one way to implement Inversion of Control. | In case of any changes in business requirements, no code change is required. |

What does run method do in spring boot?

The class annotated with **@SpringBootApplication**. is the entry point of the spring boot application to start. It creates an appropriate ApplicationContext instance and load beans.

It also runs embedded Tomcat server in Spring web application. Once started the application can be accessed at localhost:8080

**@EnableAutoConfiguration:** It auto-configures the bean that is present in the classpath and configures it to run the methods. The use of this annotation is reduced in Spring Boot 1.2.0 release because developers provided an alternative of the annotation, i.e. **@SpringBootApplication**.

**@SpringBootApplication:** It is a combination of three annotations **@EnableAutoConfiguration, @ComponentScan,** and **@Configuration**.

@Autowired annotation to discover the beans automatically and inject collaborating beans (other associated dependent beans) into our bean

@**Repository** is a **Spring** annotation that indicates that the decorated class is a **repository**. A **repository** is a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects

@**Component** is an **annotation** that allows Spring to automatically detect our custom beans.

@ComponentScan enables component scanning and means controller classes and components you create can be discovered by the framework. It marks classes to be discovered with @Controller.

 @Configuration annotation which **indicates that the class has @Bean definition methods**. So Spring container can process the class and generate Spring Beans to be used in the application.

# Can we define an abstract class with no abstract methods in Java?

Yes, we can declare an abstract class with no abstract methods in Java.

The general contract for hashCode is:

* When it is invoked more than once during the execution of an application, the hashCode() method will consistently return the same hash code (integer value). Note that the object should not be modified.
* If the two objects are equal according to the equals() method, then invoking the hashCode() method on these two objects must produce the same integer value.
* It is not necessary that if the two objects are unequal according to equals() method, then invoking the hashCode() method on these two objects may produce distinct integer value. It means that it can produce the same hash code for both objects.

In general, a Java memory leak happens **when an application unintentionally (due to logical errors in code) holds on to object references that are no longer required**. These unintentional object references prevent the built-in Java garbage collection mechanism from freeing up the memory consumed by these objects.

| Iterator | ListIterator |
| --- | --- |
| Can traverse elements present in Collection only in the forward direction. | Can traverse elements present in Collection both in forward and backward directions. |
| Helps to traverse Map, List and Set. | Can only traverse List and not the other two. |
| Indexes cannot be obtained by using Iterator. | It has methods like nextIndex() and previousIndex() to obtain indexes of elements at any time while traversing List. |
| Cannot modify or replace elements present in Collection | We can modify or replace elements with the help of set(E e) |
| Cannot add elements and it throws ConcurrentModificationException. | Can easily add elements to a collection at any time. |
| Certain methods of Iterator are next(), remove() and hasNext(). | Certain methods of ListIterator are next(), previous(), hasNext(), hasPrevious(), add(E e). |

## 1) Java static variable

If you declare any variable as static, it is known as a static variable.

* The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
* The static variable gets memory only once in the class area at the time of class loading.

### Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

# Java String intern()

The **Java String class** **intern()** method returns the interned string. It returns the canonical representation of string.

It can be used to return string from memory if it is created by a new keyword. It creates an exact copy of the heap string object in the String Constant Pool.

### Signature

The signature of the intern() method is given below:

1. **public** String intern()

### Returns

interned string

1. **public** **class** InternExample{
2. **public** **static** **void** main(String args[]){
3. String s1=**new** String("hello");
4. String s2="hello";
5. String s3=s1.intern();//returns string from pool, now it will be same as s2
6. System.out.println(s1==s2);//false because reference variables are pointing to different instance
7. System.out.println(s2==s3);//true because reference variables are pointing to same instance
8. }}

**Daemon thread in Java** is a service provider thread that provides services to the user thread. when all the user threads dies, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

The word SOLID acronym for:

* Single Responsibility Principle (SRP)

The single responsibility principle states that **every Java class must perform a single functionality**. Implementation of multiple functionalities in a single class mashup the code and if any modification is required may affect the whole class. It precise the code and the code can be easily maintained.

* Open-Closed Principle (OCP)

The application or module entities the methods, functions, variables, etc. The open-closed principle states that according to new requirements **the module should be open for extension but closed for modification.** The extension allows us to implement new functionality to the module.

* Liskov Substitution Principle (LSP)

It applies to inheritance in such a way that the **derived classes must be completely substitutable for their base classes**. In other words, if class A is a subtype of class B, then we should be able to replace B with A without interrupting the behavior of the program.

* Interface Segregation Principle (ISP)

The principle states that the larger interfaces split into smaller ones. Because the implementation classes use only the methods that are required. We should not force the client to use the methods that they do not want to use.

* Dependency Inversion Principle (DIP)

we must use abstraction (abstract classes and interfaces) instead of concrete implementations. High-level modules should not depend on the low-level module but both should depend on the abstraction. Because the abstraction does not depend on detail but the detail depends on abstraction. It decouples the software.

## Why should we use SOLID principles?

* It reduces the dependencies so that a block of code can be changed without affecting the other code blocks.
* The principles intended to make design easier, understandable.
* By using the principles, the system is maintainable, testable, scalable, and reusable.
* It avoids the bad design of the software.

Why future is used?

java.util.concurrent.The callable object can return the computed result done by a thread in contrast to a runnable interface which can only run the thread. The Callable object returns a Future object which provides methods to monitor the progress of a task being executed by a thread. The future object can be used to check the status of a Callable and then retrieve the result from the Callable once the thread is done. It also provides timeout functionality.

| **ConcurrentHashMap** | **SynchronizedMap** | **HashTable** |
| --- | --- | --- |
| We will get thread safety without locking the total map object just with a bucket level lock. | We will get thread safety by locking the whole map object. | We will get thread safety by locking the whole map object |
| At a time multiple threads are allowed to operate on map objects safely. | At a time only one thread is allowed to perform any operation on a map object. | At a time one thread is allowed to operate on a map object. |
| Read operation can be performed without lock but write operation can be performed with bucket level lock. | Every read and write operations required total map object | Every read and write operations required total map object |
| While one thread iterating map objects the other thread is allowed to modify the map and won’t get ConcurrentModificationException. | While one thread iterating map object the other threads are not allowed to modify the map otherwise we will get ConcurrentModificationException | While one thread iterating map object the other threads are not allowed to modify the map otherwise we will get ConcurrentModificationException |
| Iterator of ConcurrentHashMap is fail-safe and won’t raise ConcurrentModificationException | Iterator of SynchronizedMap is fail-fast and it will raise ConcurrentModificationException | Iterator of HashTable is fail-fast and it will raise ConcurrentModificationException |
| Null is not allowed for both keys and values. | Null is allowed for both keys and values | Null is not allowed for both keys and values. |
| Introduce in java 1.5version | Introduce in java 1.2 version | Introduce in java 1.0version |

Horizontal Scaling means modifying the compute resources of an existing cluster, for example, by adding new nodes to it or by adding new pods by increasing the replica count of pods (Horizontal Pod Autoscaler). Vertical Scaling means to modify the attributed resources (like CPU or RAM) of each node in the cluster.

What is interceptor?

How to change bean scope in spring?

How to use cross origin