|  | **HashMap** | **ConcurrentHashMap** |
| --- | --- | --- |
| Synchronized | HashMap is not synchronized. | ConcurrentHashMap is synchronized. |
| Thread Safe | HashMap is not thread safe. | ConcurrentHashMap is thread safe. |
| Iterator type | HashMap iterator is fail-fast throws ConcurrentModificationException if concurrent modification happens during iteration. | ConcurrentHashMap is fail-safe and it will never throw ConcurrentModificationException during iteration. |
| Null values | HashMap allows key and value to be null. | ConcurrentHashMap does not allow null key/value. It will throw NullPointerException. |
| Performance | HashMap is faster. | ConcurrentHashMap is slower than HashMap. |
| Since Java Version | 1.2 | 1.5 |

### Why ConcurrentHashMap does not allow null keys and null values?

* ConcurrentHashMap does not allow null keys and null values
* The main reason that nulls aren't allowed in ConcurrentMaps because there will be ambiguities that may be just barely tolerable in non-concurrent maps can't be accommodated.
* **The main one is that if map.get(key) returns null, you can't detect whether the key explicitly maps to null vs the key isn't mapped.**
* In a non-concurrent map, you can check this via map.contains(key), but in a concurrent one, the map might have changed between calls.

The code is like this:

if (map.containsKey(k)) {

     return map.get(k);

} else {

     throw new KeyNotPresentException();

}

It might be possible that key k might be deleted in between the get(k) and containsKey(k) calls.

As a result, the code will return null as opposed to **KeyNotPresentException** (Expected Result if key is not present).

The Null key and value allowed in HashMap because **there is no Concurrent access**.

### Why can’t constructors be final, static, native, synchronized or abstract in Java?

### When you use a final keyword with a method or constructor it cannot be overridden. But, a constructor in Java cannot be overridden therefore; there is no need of using the final keyword with the constructor.

### Since you cannot override a constructor you cannot provide body to it if it is made abstract. Therefore, you cannot use abstract keyword with the constructor

### When you set a method as ‘static’, it means: “Method belong to class, not a particular object” but constructor implicitly called to initialize an object, so there is no purpose in having a static constructor.

### Can we override and overload static method

### No, you cannot override static method in Java, though you can declare method with same signature in sub class. It won't be overridden in exact sense, instead that is called method hiding.

### But at same time, you can overload static methods in Java, there is nothing wrong declaring static methods with same name, but different arguments

**Can we synchronized a static method**  🡪 yes. It will acquire the lock at class level.

### Methods in object class

1. **toString() :** toString() provides String representation of an Object and used to convert an object to String.

**2. hashCode() :** For every object, JVM generates a unique number which is hashcode. It returns distinct integers for distinct objects.

* Use of hashCode() method : Returns a hash value that is used to search object in a collection. JVM(Java Virtual Machine) uses hashcode method while saving objects into hashing related data structures like HashSet, HashMap, Hashtable etc. The main advantage of saving objects based on hash code is that searching becomes easy.
* Note : Override of hashCode() method needs to be done such that for every object we generate a unique number.
* hashCode() method of object class returns the memory reference of object in integer form. Definition of hashCode() method is public native hashCode(). It indicates the implementation of hashCode() is native because there is not any direct method in java to fetch the reference of object.

**3. equals(Object obj) :** Compares the given object to “this” object (the object on which the method is called)

* Note : It is generally necessary to override the hashCode() method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

**4. getClass() :** Returns the class object of “this” object and used to get actual runtime class of the object.

* It can also be used to get metadata of this class. The returned Class object is the object that is locked by static synchronized methods of the represented class. As it is final so we don’t override it.

**5. finalize():** This method is called just before an object is garbage collected.

* It is called by the Garbage Collector on an object when garbage collector determines that there are no more references to the object. We should override finalize() method to dispose system resources, perform clean-up activities and minimize memory leaks. For example before destroying Servlet objects web container, always called finalize method to perform clean-up activities of the session.
* Note :finalize method is called just once on an object even though that object is eligible for garbage collection multiple times.

**6. clone() :** It returns a new object that is exactly the same as this object.

**7. wait():** causes the current thread to wait until another thread invokes the **notify()**or **notifyAll()**methods for that object.

**8. notify():** wakes up a single thread that is waiting on that object’s monitor

**9. notifyAll():**wakes up all threads that are waiting on that object’s monitor

A thread waits on an object’s monitor by calling one of the **wait()**method. These methods can throw **IllegalMonitorStateException** if the current thread is not the owner of the object’s monitor.

### Static and dynamic binding

1. **Static Binding**: The binding which can be resolved at compile time by compiler is known as static or early binding. Binding of all the static, private and final methods is done at compile-time .
2. **Why binding of static, final and private methods is always a static binding?**

Static binding is better performance wise (no extra overhead is required). Compiler knows that all such methods cannot be overridden and will always be accessed by object of local class. Hence compiler doesn’t have any difficulty to determine object of class (local class for sure). That’s the reason binding for such methods is static.

1. **Dynamic Binding**: In Dynamic binding compiler doesn’t decide the method to be called. Overriding is a perfect example of dynamic binding. In overriding both parent and child classes have same method
2. Private, final and static members (methods and variables) use static binding while for virtual methods (In Java methods are virtual by default) binding is done during run time based upon run time object.
3. Static binding uses Type information for binding while Dynamic binding uses Objects to resolve binding.
4. Overloaded methods are resolved (deciding which method to be called when there are multiple methods with same name) using static binding while overridden methods using dynamic binding, i.e, at run time.

### Transient and volatile

* Transient is a variables modifier used in serialization. **At the time of serialization, if we don’t want to save value of a particular variable in a file, then we use transient keyword. When JVM comes across transient keyword, it ignores original value of the variable and save default value of that variable data type.**
* Transient keyword plays an important role to meet security constraints. There are various real-life examples where we don’t want to save private data in file. Another use of transient keyword is not to serialize the variable whose value can be calculated/derived using other serialized objects or system such as age of a person, current date, etc.
* Practically we serialized only those fields which represent a state of instance, after all serialization is all about to save state of an object to a file. It is good habit to use transient keyword with private confidential fields of a class during serialization
* **Transient and static:** Since static fields are not part of state of the object, there is no use/impact of using transient keyword with static variables. However there is no compilation error.
* **Transient and final:** final variables are directly serialized by their values, so there is no use/impact of declaring final variable as transient. There is no compile-time error though.
* **Volatile** in Java is used as an indicator to Java compiler and Thread that do not cache value of this variable and always read it from main memory.
* The Java volatile keyword cannot be used with method or class and it can only be used with a variable.

### Hash collision

The phenomenon when two keys have same hash code is called hash collision. If hashCode() method is not implemented properly, there will be higher number of hash collision and map entries will not be properly distributed causing slowness in the get and put operations(as hashcode of key is used to decide the bucket and if hashcode is same for most then data will be more on one bucket). **This is the reason for prime number usage in generating hash code so that map entries are properly distributed across all the buckets**.

### How to redirect output of System.out to file.

We must reassign the standard output by using the following method of System class:

System.setOut(PrintStream p);

PrintStream can be used for character output to a text file

// Java program to demonstrate redirection in System.out.println()

import java.io.\*;

public class SystemFact

{

    public static void main(String arr[]) throws FileNotFoundException

    {

        // Creating a File object that represents the disk file.

        PrintStream o = new PrintStream(new File("A.txt"));

        // Store current System.out before assigning a new value

        PrintStream console = System.out;

        // Assign o to output stream

        System.setOut(o);

        System.out.println("This will be written to the text file");

        // Use stored value for output stream

        System.setOut(console);

        System.out.println("This will be written on the console!");

    }

}

### Association, composition, aggregation

* **Association** is relation between two separate classes which establishes through their Objects.
* **Aggregation** implies a relationship where the child can exist independently of the parent.

For example, Bank and Employee, delete the Bank and the Employee still exist.

* **Composition** implies a relationship where the child cannot exist independent of the parent.

Example: Human and heart, heart don’t exist separate to a Human

* **Type of Relationship**: Aggregation relation is “has-a” and composition is “part-of” relation.
* **Type of association:** Composition is a strong Association whereas Aggregation is a weak Association.

### Relation between hashcode and equals method

### 2 object if equals will have same hashcode but 2 objects with same hashcode may not be necessary that they are equal

### Top of exception hierarcy 🡪 Throwable

### Example of errors 🡪 stackoverflow or outofmemory

### How to sort external class in case we don’t have its implementation 🡪 comparator

### When to use comparable and comparator 🡪 if different sorting needed then comparator.

### Diff between throw and throws 🡪 throw in exception block and throws in method signature

### How to maintain an insertion order in map 🡪 linkedhashmap

**Can we use super () and this () within the same constructor? 🡪 No**

Both this**()** and **super()** are **constructor** calls. **Constructor** call must always be the first statement. So **we can**not have two statements as first statement, hence either **we can** call **super()** or **we can** call this**()** from the **constructor**, but not both

[**Static Methods**](https://docs.oracle.com/javase/tutorial/java/IandI/override.html)

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass *hides* the one in the superclass.

The distinction between hiding a static method and overriding an instance method has important implications:

* The version of the overridden instance method that gets invoked is the one in the subclass.
* The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

**Interface Methods**

[Default methods](https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html) and [abstract methods](https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html) in interfaces are inherited like instance methods. However, when the supertypes of a class or interface provide multiple default methods with the same signature, the Java compiler follows inheritance rules to resolve the name conflict. These rules are driven by the following two principles:

* **Instance methods are preferred over interface default methods.**

Consider the following classes and interfaces:

public class Horse {

public String identifyMyself() {

return "I am a horse.";

}

}

public interface Flyer {

default public String identifyMyself() {

return "I am able to fly.";

}

}

public interface Mythical {

default public String identifyMyself() {

return "I am a mythical creature.";

}

}

public class Pegasus extends Horse implements Flyer, Mythical {

public static void main(String... args) {

Pegasus myApp = new Pegasus();

System.out.println(myApp.identifyMyself());

}

}

The method Pegasus.identifyMyself returns the string I am a horse.

* **Methods that are already overridden by other candidates are ignored. This circumstance can arise when supertypes share a common ancestor.**

Consider the following interfaces and classes:

public interface Animal {

default public String identifyMyself() {

return "I am an animal.";

}

}

public interface EggLayer extends Animal {

default public String identifyMyself() {

return "I am able to lay eggs.";

}

}

public interface FireBreather extends Animal { }

public class Dragon implements EggLayer, FireBreather {

public static void main (String... args) {

Dragon myApp = new Dragon();

System.out.println(myApp.identifyMyself());

}

}

The method Dragon.identifyMyself returns the string I am able to lay eggs.

**If two or more independently defined default methods conflict, or a default method conflicts with an abstract method, then the Java compiler produces a compiler error. *You must explicitly override the supertype methods.***

Consider the example about computer-controlled cars that can now fly. You have two interfaces (OperateCar and FlyCar) that provide default implementations for the same method, (startEngine):

public interface OperateCar {

default public int startEngine(EncryptedKey key) {

// Implementation

}

}

public interface FlyCar {

default public int startEngine(EncryptedKey key) {

// Implementation

}

}

**A class that implements both OperateCar and FlyCar must override the method startEngine. *You could invoke any of the of the default implementations with the****super****keyword.***

public class FlyingCar implements OperateCar, FlyCar {

public int startEngine(EncryptedKey key) {

FlyCar.super.startEngine(key);

OperateCar.super.startEngine(key);

}

}

The name preceding super (in this example, FlyCar or OperateCar) must refer to a **direct superinterface that defines or inherits a default for the invoked method**. This form of method invocation is not restricted to differentiating between multiple implemented interfaces that contain default methods with the same signature. You can use the super keyword to invoke a default method in both classes and interfaces.

Inherited instance methods from classes can override abstract interface methods. Consider the following interfaces and classes:

public interface Mammal {

String identifyMyself();

}

public class Horse {

public String identifyMyself() {

return "I am a horse.";

}

}

public class Mustang extends Horse implements Mammal {

public static void main(String... args) {

Mustang myApp = new Mustang();

System.out.println(myApp.identifyMyself());

}

}

The method Mustang.identifyMyself returns the string I am a horse. The class Mustang inherits the method identifyMyself from the class Horse, which overrides the abstract method of the same name in the interface Mammal.

**Note**: Static methods in interfaces are never inherited.

[**Difference between super and super() in Java with Examples**](https://www.geeksforgeeks.org/difference-between-super-and-super-in-java-with-examples/) **🡪** super to refer variable or method and super() for constructor

**Why default methods in interface** 🡪 to allow the developers to add new methods to the interfaces without affecting the classes that implements these interfaces.

**How to handle hash collosion** 🡪 implement proper hash code

**how java do overloading/overriding internally 🡪** vtable and virtual function

double d = 10.0/0 and -10.0/0 🡪 Infinity and –Infinity **🡪** Java's float and double types, implement the IEEE 754 standard for floating point math, which mandates division by zero to return a special "infinity" value. Throwing an exception would actually violate that standard.

double d = 10/0 and -10/0 **🡪** ArithmeticException: / by zero

Integer arithmetic (implemented as two's complement representation by Java ) is different and has no special infinity or NaN values, thus throwing exceptions is a useful behaviour there.

**Can we use super in interface** 🡪yes. Usage is similar to default interface scenario

public class HelloWorld{

public static void main(String []args){

System.out.println("Hello World");

V v = new V();

v.drive();

}

}

class V implements Vehicle {}

interface Car{

public default void drive() {

System.out.println("Car is driving");

}

}

interface Jeep{

public default void drive() {

System.out.println("Jeep is driving");

}

}

interface Vehicle extends Car,Jeep {

public default void drive(){

Jeep.super.drive();

}

}

**Functional Interface? What all can we have in functional interface**

* Introduced in Java 8.
* Contains only a single abstract (unimplemented) method.
* can contain default and static methods which do have an implementation, in addition to the single unimplemented method
* can contain variable similar to normal interfaces (by default public static and final)
* **public**: for the accessibility across all the classes, just like the methods present in the interface
* **static**: as interface cannot have an object, the interfaceName.variableName can be used to reference it or directly the variableName in the class implementing it.
* **final**: to make them constants. If 2 classes implement the same interface and you give both of them the right to change the value, conflict will occur in the current value of the var, which is why only one time initialization is permitted.

**Static and Default methods in interface java 8**

**Usage**

default methods:

* It helps in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.
* It helps in extending interfaces without having the fear of breaking implementation classes.

static methods:

* They are part of interface, we can’t use it for implementation class objects.
* It helps in providing security by not allowing implementation classes to override them.

**Difference**

1) Default methods can be overridden in implementing class, while static cannot.

2) Static method belongs only to Interface class, so you can only invoke static method on Interface class, not on class implementing this Interface

3) Both class and interface can have static methods with same names, and neither overrides other!

**What if class has defined public static void main in super class and sub class inherit it 🡪** static method will not be overridden. If both classes in same file then the main in class declared as public will be executed.

**Usage of this keyword**

1. Used to refer the current class instance variable
2. Used to invoke current class default constructor
3. Used to call Current class methods
4. Can be used to pass current Java instance as parameter
5. Used to return current Java instance

**Usage of super keyword**

1. super() invokes the constructor of the parent class.
2. super.variable\_name refers to the variable in the parent class.
3. super.method\_name refers to the method of the parent class.

**What if array is passed to another method and changed its value will it get reflected in original array.**

public static void main(String []args){

String[] a = {"Original", "testing"};

System.out.println(a[0] + "\t"+a[1]); //Original testing

test(a);

System.out.println(a[0] + "\t"+a[1]); //changed testing

}

static void test(String[] a){

a[0] = "changed";

System.out.println(a[0] + "\t"+a[1]); //changed testing

}

public static void main(String []args){

String[] a = {"Original", "testing"};

System.out.println(a[0] + "\t"+a[1]); //Original testing

test(a);

System.out.println(a[0] + "\t"+a[1]); //Original testing

}

static void test(String[] a){

a = new String[2];

System.out.println(a[0] + "\t"+a[1]); //null null

}

Same is with new Object and Object.setXX();

**How to create only 11 instance of a class 🡪** singleton pattern **🡪** have a static variable in class and private constructor so variable cannot be instantiated and in getInstance method check if count > 11 then throw error else create a new instance

**How do we handle runtime exception 🡪** by following proper coding standard like null check and array size check

**How can we make sure main() is the last thread to finish in Java Program? 🡪** Thread join() method to make sure all the threads created by the program is dead before finishing the main function

**How to create daemon thread in Java? 🡪**  with Thread class setDaemon(true) We need to call this method before calling start() method else it will throw IllegalThreadStateException.

**How to handle ERROR in java 🡪** An Error usually shouldn't be caught, as it indicates an abnormal condition that should never occur.

**Diff ways of creating object in java without new keyword**

1. Class.newInstance() method 🡪 ClassName obj= ClassName.class.newInstance();
2. newInstance() method of Constructor class (refelction) **🡪**

Constructor<ClassName> obj =ClassName.class.getConstructor(); ClassName obj1 = obj.newInstance();

1. Object.clone() method **🡪** ClassName obj1 = new ClassName(); ClassName obj2 = (ClassName) obj1.clone();
2. Object Serialization and Deserialization
3. ClassLoader loadClass() **🡪** instance.getClass().getClassLoader().loadClass("NewClass").newInstance()

**What is the difference between Serializable and Externalizable interface in Java? 🡪** Externalizable provides us writeExternal() and readExternal() method which gives us flexibility to control java serialization mechanism instead of relying on Java's default serialization.

**SerialVersionUID** is used for version control of object and throws java.io.InvalidClassException in case of serialVersionUID mismatch

**Following is the list of changes which are compatible to a serializable class:**

1. Add fields
2. Change a field from static to non-static
3. Change a field from transient to non-transient
4. Add classes to the object tree

**List of incompatible changes:**

1. Delete fields
2. Change class hierarchy
3. Change non-static to static
4. Change non-transient to transient
5. Change type of a primitive field

So, if no suid is present, in spite of making compatible changes, jvm generates new suid thus resulting in an exception if prior release version object is used.

**While serializing you want some of the members not to serialize? How do you achieve it? 🡪**  Declare it either static or transient based on your need

**What will happen if one of the members in the class doesn't implement Serializable interface? 🡪** ‘NotSerializableException’ will be thrown at runtime

**What happens if the object to be serialized includes the references to other serializable objects? 🡪** The whole object graph of the object to be serialized will be saved

**What will be the value of transient variable after de-serialization? 🡪** It’s default value. For.Ex 0 in case of int

**Does the order in which the value of the transient variables and the state of the object using the defaultWriteObject() method are saved during serialization matter? 🡪** Yes, while restoring the object’s state the transient variables and the serializable variables that are stored must be restored in the same order in which they were saved.

**If a class is serializable but its superclass in not, what will be the state of the instance variables inherited from super class after deserialization? 🡪** The values of the instance variables inherited from superclass will be reset to the values they were given during the original construction of the object as the non-serializable super-class constructor will run

**Static in serialisation 🡪** does not get serialized as they are class variables

**Suppose super class of a new class implement Serializable interface, how can you avoid new class to being serialized?**

To avoid Java serialization you need to implement writeObject() and readObject() method in your Class and need to throw NotSerializableException from those method. This is another benefit of customizing java serialization process.

**Does constructor get invoked when class is deserialized? 🡪** No. Constructor is invoked only in case the super class is not serializable

**How to Serialize a collection in java? 🡪** All standard implementations of collections List, Set and Map interface already implement java.io.Serializable. This means you do not really need to write anything specific to serialize collection objects. However you should keep following things in mind before you serialize a collection object - Make sure all the objects added in collection are Serializable.

**Class level lock:** Every class in Java has a unique lock which is nothing but a class level lock. If a thread wants to execute a static synchronized method, then thread requires a class level lock

**Object level lock:** Every object in Java has a unique lock. If a thread wants to execute a synchronized method on a given object, first it has to get a lock of that object.

**wait**(long **timeout**) causes current thread to **wait** until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of **time** has elapsed. when the timeout expires the thread wakes up and tries to re-acquire the synchronization lock. This is for hang notification, for low-power polling, etc

**Can we start a thread twice in Java? 🡪** No,Doing so will throw an illegalThreadStateException

**What Are the Available Implementations of Executorservice in the Standard Library?**

The ExecutorService interface has three standard implementations:

1. **ThreadPoolExecutor** — for executing tasks using a pool of threads. Once a thread is finished executing the task, it goes back into the pool. If all threads in the pool are busy, then the task has to wait for its turn.
2. **ScheduledThreadPoolExecutor** allows to schedule task execution instead of running it immediately when a thread is available. It can also schedule tasks with fixed rate or fixed delay.
3. **ForkJoinPool** is a special ExecutorService for dealing with recursive algorithms tasks. If you use a regular ThreadPoolExecutor for a recursive algorithm, you will quickly find all your threads are busy waiting for the lower levels of recursion to finish. The ForkJoinPool implements the so-called work-stealing algorithm that allows it to use available threads more efficiently.

|  |  |
| --- | --- |
| **Runnable** | **Callable** |
| run() method | call() method |
| No return value | Returns a generic value V |
| Cannot throw exception | can throw checked exception |
| Use execute to put in task queue | Use submit to put in task queue |

**What is ClassLoader in Java**

ClassLoader in Java is a class which is used to load class files in Java. Java code is compiled into class file by javac compiler and JVM executes Java program, by executing byte codes written in class file. ClassLoader is responsible for loading class files from file system, network or any other source. **There are three default class loader used in Java, Bootstrap , Extension and System or Application class loader.**

Location from which Bootstrap, Extension and Application ClassLoader load Class files.

1. Bootstrap ClassLoader - JRE/lib/rt.jar
2. Extension ClassLoader - JRE/lib/ext or any directory denoted by java.ext.dirs
3. Application ClassLoader - CLASSPATH environment variable, -classpath or -cp option, Class-Path attribute of Manifest inside JAR file.

**Encapsulation:**

* Encapsulation is an approach that joins data members(variables) and implementation details into a single unit called the class that implies class is formed with variables and methods present inside it.
* Encapsulation is a protection mechanism for data members present inside the class i.e data members are not accessible by end users.
* In encapsulation, the data members(variables) of a class will be not accessible by other classes and can be accessed only through the methods of their current class.
* Encapsulation is implemented using a private and protected access modifier.
* Encapsulation is used for reducing the complexity of a web application

**Abstraction:**

* The structure of representing necessary features without including the background details is specified as an abstraction.
* Abstraction is the technique of covering the implementation details from the end-user and only displaying the functionality to the users.
* Abstraction is utilized to execute polymorphic ideas with classes i.e With abstraction we can only declare the methods inside a class. Usually, another derive class is used to describe the same methods.
* An abstract class cannot be instantiated so end-users can't directly access it.
* It is used to reduce the complexity of the application and make the code reusable in an application.

**What is bounded and unbounded wildcards in Generics Java?** 🡪 <? extends T> and <? super T> represent bounded wildcards while <?> represent an unbounded wildcard in generics

**Covariant method overriding** helps to remove [type casting](http://javarevisited.blogspot.sg/2012/12/what-is-type-casting-in-java-class-interface-example.html) on client side, by allowing you **to return subtype** of actually return type of overridden method.

**Type Erasure:** compiler does all its verification on generic code and strips the type information out of the class bytecode. At runtime, ALL collection code both legacy & generics looks like pre-generic version of collection. No type information exits at runtime i.e List<String> l = new ArrayList<String>()🡪 at runtime : List list = new ArrayList();

**Generics and Collection**

* Parent[] myArray = new Child[3] **🡪** allowed **🡪** ArrayStoreException at runtime for arrays **🡪** if you add dog in a cat array (referred as animal array)
* List<Parent> p = new Arraylist<child> is this possible? **🡪** No compile time error (generics) **🡪** No equivalent exception for generics due to type erasure
* List<?> or List<? extends Object> **🡪** any type but only for access **🡪** <Dog>,<Integer> ,<Object>
* List <Object> **🡪** only List<Object> **🡪** No List<Integer>,<Dog>,etc
* List<?> list = new ArrayList<Dog>() **🡪** allowed
* List<? extends Animal> list = new ArrayList<Dog>() **🡪** allowed **🡪** error will adding to list. list.add(new Dog) **🡪** not allowed
* List<?> foo = new ArrayList<? Extends Animal>() **🡪** Not allowed **🡪** cannot use wildcard notation in object creation
* List<? super Dog> list = new ArrayList<Animal>() **🡪** allowed **🡪**allowed to add in list
* List<? super Animal> list = new ArrayList<Dog>() **🡪** not allowed **🡪** Dog is subclass of Animal

**Singleton pattern vs. Spring Singleton Scope**

* **Singleton pattern** is scoped by per Java class.
* **Singleton bean scope** is scoped by per spring container.
* In singleton pattern, Java considers a class as singleton if it cannot create more than one instance of that class within a **given class loader** whereas spring considers a bean class as singleton scope if it cannot create more than one instance of bean class within a given **Applicationcontext(container).**
* Very important point to discuss here is what happens if there are multiple containers and same class loader.Lets create one example to understand this.

public static void main(String[] args)

{

ApplicationContext factory = new ClassPathXmlApplicationContext(new String[] { "Applicationcontext.xml"});

Student student1 = (Student) factory.getBean("student");

student1.setName("Shikha");

 System.out.println("Bean name 1 : " + student1.getName());

ApplicationContext factory2 = new ClassPathXmlApplicationContext(new String[] { "Applicationcontext.xml"});

      Student student2= (Student) factory2.getBean("student");

      System.out.println("Bean name in case of new applicationcontext: " + student2.getName());   //returns null as new student beand gets created

      System.out.println("context classloader: "+factory.getClassLoader()); //same for both this and below line

       System.out.println("newContext classloader: "+factory2.getClassLoader());

 }

**equals and hashcode method override, how to restrict subclass from using super class equals method.** 🡪 Make super class equals as final

### Java SE 8 New Features?

### 1. Functional Interface : Each functional interface has a single abstract method, called the functional method, implementation can be provided using the lambda expressions. 2. Lambda Expressions : It is a feature derived from the functional programming. It is a function that does not belong to any class. 3. Optional : Instead of using null values Optional class is used for representing Optional values. 4. Stream api 5. Spliterator  6. Method References 7. New Date and Time API.

* Linked list in java internally uses doubly linked list
* **How to prevent overriding in java** ...3 ways --> private , static , final

### What is Lambda Expression?

Lambda Expression is an anonymous function which accepts a set of input parameters and returns results.

Lambda Expression is a block of code without any name, with or without parameters and with or without results. This block of code is executed on demand.

### What are the three parts of a Lambda Expression? What is the type of Lambda Expression?

A Lambda Expression contains 3 parts:

* Parameter List 🡪 A Lambda Expression can contain zero or one or more parameters. It is optional.
* Lambda Arrow Operator 🡪 “->” is known as Lambda Arrow operator. It separates parameters list and body.
* Lambda Expression Body

What is the type of a Lambda Expression? 🡪 The Type of a Lambda Expression is a [Functional Interface](https://www.journaldev.com/2763/java-8-functional-interfaces).

### Explain Differences between Collection API and Stream API?

|  |  |
| --- | --- |
| Collection API | Stream API |
| It’s available since Java 1.2 | It is introduced in Java SE8 |
| It is used to store Data(A set of Objects). | It is used to compute data(Computation on a set of Objects). |
| We can use both Spliterator and Iterator to iterate elements. We can use [forEach](https://www.journaldev.com/13941/java-foreach-java-8-foreach) to performs an action for each element of this stream. | We can’t use Spliterator or Iterator to iterate elements. |
| It is used to store limited number of Elements. | It is used to store either Limited or Infinite Number of Elements. |
| Typically, it uses External Iteration concept to iterate Elements such as Iterator. | Stream API uses internal iteration to iterate Elements, using forEach methods. |
| Collection Object is constructed Eagerly. | Stream Object is constructed Lazily. |
| We add elements to Collection object only after it is computed completely. | We can add elements to Stream Object without any prior computation. That means Stream objects are computed on-demand. |
| We can iterate and consume elements from a Collection Object at any number of times. | We can iterate and consume elements from a Stream Object only once. |

### What is Optional in Java 8? What is the use of Optional?Advantages of Java 8 Optional?

* Optional is a final Class introduced as part of Java SE 8. It is defined in java.util package.
* It is used to represent optional values that is either exist or not exist.
* It can contain either one value or zero value. If it contains a value, we can get it. Otherwise, we get nothing.It is a bounded collection that is it contains at most one element only. It is an alternative to “null” value.

**Main Advantage of Optional is:**

* It is used to avoid null checks.
* It is used to avoid “NullPointerException”.

[**What is method reference in Java 8?**](https://www.bestinterviewquestion.com/question/what-is-method-reference-in-java-8-xou0d1067)

Method references support in pointing to the methods by their names. A method reference is denoted by using "::" symbol. A method reference is used to indicate the following methods

* Static methods
* Instance methods
* Constructors using new operator (TreeSet::new)

### What is Spliterator in Java SE 8?Differences between Iterator and Spliterator in Java SE 8?

Spliterator stands for Splitable Iterator. It is newly introduced by Oracle Corporation as part Java SE 8.  
Like Iterator and ListIterator, It is also one of the Iterator interface.

|  |  |
| --- | --- |
| Spliterator | Iterator |
| It is introduced in Java SE 8. | It is available since Java 1.2. |
| Splitable Iterator | Non-Splitable Iterator |
| It is used in Stream API. | It is used for Collection API. |
| It uses Internal Iteration concept to iterate Streams. | It uses External Iteration concept to iterate Collections. |
| We can use Spliterator to iterate Streams in Parallel and Sequential order. | We can use Iterator to iterate Collections only in Sequential order. |
| We can get Spliterator by calling spliterator() method on Stream Object. | We can get Iterator by calling iterator() method on Collection Object. |
| Important Method: tryAdvance() | Important Methods: next(), hasNext() |

* **Spring beans** are the objects which are created and managed completely by **spring** container.

**Bean Scope**

1. **singleton**: This bean scope is **default** and it enforces the container to have only one instance per spring container irrespective of how much time you request for its instance. This singleton behavior is maintained by bean factory itself.
2. **prototype**: produces a new instance each and every time a bean is requested.
3. **request**: a new bean instance will be created for each web request made by client. As soon as request completes, bean will be out of scope and garbage collected.
4. **session**: Just like request scope, this ensures one instance of bean per user session. As soon as user ends its session, bean is out of scope.
5. **global-session**: global-session is something which is connected to Portlet applications. When your application works in Portlet container it is built of some amount of portlets. Each portlet has its own session, but if your want to store variables global for all portlets in your application than you should store them in global-session. This scope doesn’t have any special effect different from session scope in Servlet based applications.

**@Component** is a class level annotation and its purpose it to make the class as spring managed component and auto detectable bean for classpath scanning feature.

**@Bean** is used to explicitly declare and register a bean (as a configuration bean) in Spring IOC container that is returned from a method. **@Bean** is a method level annotation and it is used within a class that is annotated with **@Configuration.** Simply,**@Bean** annotation is used to register the bean returned by a method as a spring configuration bean in IOC Container.  **@Bean** is only a method level annotation and it cannot be used with classes and object declaration. by default, name of the method serves as the bean name. **@Bean** annotation indicates that a method produces a bean that should be managed by the Spring container.

**Basic stater depencdency required for spring boot application 🡪** typically Maven POM file inherits from the spring-boot-starter-parent project and declares dependencies to one or more “Starters" for ex : spring-boot-starter-web,spring-boot-starter-actuator,spring-boot-starter-data-jpa,spring-boot-starter-activemq,etc

**Spring bean default lazy or not 🡪** No. By default its always eager instantiation

## @SpringBootApplication

Spring boot is mostly about auto-configuration. This auto-configuration is done by **component scanning** i.e. finding all classes in classspath for [@Component](https://howtodoinjava.com/spring-core/how-to-use-spring-component-repository-service-and-controller-annotations/) annotation. It also involve scanning of @Configuration annotation and initialize some extra beans.

[@SpringBootApplication](https://howtodoinjava.com/spring-boot/springbootapplication-auto-configuration/) annotation enable all able things in one step. It enables the three features:

1. @EnableAutoConfiguration : enable auto-configuration mechanism
2. [@ComponentScan](https://howtodoinjava.com/spring-mvc/spring-mvc-difference-between-contextannotation-config-vs-contextcomponent-scan/) : enable @Component scan
3. @SpringBootConfiguration : register extra beans in the context

**The java class annotated with @SpringBootApplication is the main class of a Spring Boot application and application starts from here.**

|  |
| --- |
| import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;   @SpringBootApplication  public class Application {       public static void main(String[] args) {          SpringApplication.run(Application.class, args);      }   } |

## @EnableAutoConfiguration

This annotation enables auto-configuration of the Spring Application Context, attempting to guess and configure beans that we are likely to need based on the presence of predefined classes in classpath.

For example, if we have tomcat-embedded.jar on the classpath, we are likely to want a TomcatServletWebServerFactory.

Auto-configuration classes are regular Spring Configuration beans. They are located using the SpringFactoriesLoader mechanism (keyed against this class). Generally auto-configuration beans are @Conditional beans (most often using @ConditionalOnClass and @ConditionalOnMissingBean annotations).

## @SpringBootConfiguration

It indicates that a class provides Spring Boot application configuration. It can be used as an alternative to the Spring’s standard @Configuration annotation so that configuration can be found automatically.

Application should only ever include one @SpringBootConfiguration and most idiomatic Spring Boot applications will inherit it from @SpringBootApplication.

The main difference is both annotations is that @SpringBootConfiguration allows configuration to be automatically located. This can be especially useful for unit or integration tests.

## @ImportAutoConfiguration

It import and apply only the specified auto-configuration classes. The difference between @ImportAutoConfiguration and @EnableAutoConfiguration is that later attempts to configure beans that are found in the classpath during scanning, whereas @ImportAutoConfiguration only runs the configuration classes that we provide in the annotation.

We should use @ImportAutoConfiguration when we don’t want to enable the default auto-configuration.

|  |
| --- |
| **@ImportAutoConfiguration example** |
| @ComponentScan("path.to.your.controllers")  @ImportAutoConfiguration({WebMvcAutoConfiguration.class, DispatcherServletAutoConfiguration.class      ,EmbeddedServletContainerAutoConfiguration.class, ServerPropertiesAutoConfiguration.class      ,HttpMessageConvertersAutoConfiguration.class})  public class App  {      public static void main(String[] args)      {          SpringApplication.run(App.class, args);      }  } |

## @AutoConfigureBefore, @AutoConfigureAfter, @AutoConfigureOrder

We can use the @AutoConfigureAfter or @AutoConfigureBefore annotations if our configuration needs to be applied in a specific order (before of after).

If we want to order certain auto-configurations that should not have any direct knowledge of each other, we can also use @AutoConfigureOrder. That annotation has the same semantic as the regular @Order annotation but provides a dedicated order for auto-configuration classes.

|  |
| --- |
| @AutoConfigureAfter Example |
| @Configuration  @AutoConfigureAfter(CacheAutoConfiguration.class)  @ConditionalOnBean(CacheManager.class)  @ConditionalOnClass(CacheStatisticsProvider.class)  public class RedissonCacheStatisticsAutoConfiguration  {      @Bean      public RedissonCacheStatisticsProvider redissonCacheStatisticsProvider(){          return new RedissonCacheStatisticsProvider();      }  } |

## Condition Annotations

All auto-configuration classes generally have one or more @Conditional annotations. It allow to register a bean only when the condition meets. Following are some useful conditional annotations to use.

#### 5.1. @ConditionalOnBean and @ConditionalOnMissingBean

These annotations let a bean be included based on the presence or absence of specific beans.

It’s value attribute is used to specify beans **by type** or by name. Also the search attribute lets us limit the ApplicationContext hierarchy that should be considered when searching for beans.

Using these annotations at the class level prevents registration of the @Configuration class as a bean if the condition does not match.

In below example, bean JpaTransactionManager will only be loaded if a bean of type JpaTransactionManager is not already defined in the application context.

|  |
| --- |
| @Bean  @ConditionalOnMissingBean(type = "JpaTransactionManager")  JpaTransactionManager transactionManager(EntityManagerFactory entityManagerFactory)  {      JpaTransactionManager transactionManager = new JpaTransactionManager();      transactionManager.setEntityManagerFactory(entityManagerFactory);      return transactionManager;  } |

#### 5.2. @ConditionalOnClass and @ConditionalOnMissingClass

These annotations let configuration classes be included based on the presence or absence of specific classes. Notice that annotation metadata is parsed by using spring ASM module, and even if a class might not be present in runtime – you can still refer to the class in annotation.

We can also use value attribute to refer the real class or the name attribute to specify the class name by using a String value.

Below configuration will create EmbeddedAcmeService only if this class is available in runtime and no other bean with same name is present in application context.

|  |
| --- |
| @Configuration  @ConditionalOnClass(EmbeddedAcmeService.class)  static class EmbeddedConfiguration  {      @Bean      @ConditionalOnMissingBean      public EmbeddedAcmeService embeddedAcmeService() { ... }  } |

#### 5.3. @ConditionalOnNotWebApplication and @ConditionalOnWebApplication

These annotations let configuration be included depending on whether the application is a “web application” or not. In Spring, a web application is one which meets at least one of below three requirements:

1. uses a Spring WebApplicationContext
2. defines a session scope
3. has a StandardServletEnvironment

#### 5.4. @ConditionalOnProperty

This annotation lets configuration be included based on the presence and value of a Spring Environment property.

For example, if we have different datasource definitions for different environments, we can use this annotation.

|  |
| --- |
| @Bean  @ConditionalOnProperty(name = "env", havingValue = "local")  DataSource dataSource()  {      // ...  }  @Bean  @ConditionalOnProperty(name = "env", havingValue = "prod")  DataSource dataSource()  {      // ...  } |

#### 5.5. @ConditionalOnResource

This annotation lets configuration be included only when a specific resource is present in the classpath. Resources can be specified by using the usual Spring conventions.

|  |
| --- |
| @ConditionalOnResource(resources = "classpath:vendor.properties")  Properties additionalProperties()  {  } |

#### 5.6. @ConditionalOnExpression

This annotation lets configuration be included based on the result of a [SpEL expression](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions). Use this annotation when condition to evaluate is complex one and shall be evaluated as one condition.

|  |
| --- |
| @Bean  @ConditionalOnExpression("${env} && ${havingValue == 'local'}")  DataSource dataSource()  {  } |

#### 5.7. @ConditionalOnCloudPlatform

This annotation lets configuration be included when the specified cloud platform is active.

|  |
| --- |
| @Configuration  @ConditionalOnCloudPlatform(CloudPlatform.CLOUD\_FOUNDRY)  public class CloudConfigurationExample  {    @Bean    public MyBean myBean(MyProperties properties)    {      return new MyBean(properties.getParam);    }  } |

**@Value annotation** is used to assign default values to variables and method arguments. We can read spring environment variables as well as system variables using @Value annotation.

[@Value – Default Value](https://www.journaldev.com/21448/spring-value-annotation#spring-value-8211-default-value) : @Value annotation argument can be a string only, but spring tries to convert it to the specified type.

@Value("10")

private int defaultInt;

@Value – Spring Environment Property/ System Environment : @Value("${mongodb.url:127.0.0.1}")

@Value – SpEL : @Value("#{systemProperties['java.home']}") or @Value("#{config['mongodb.url']?:'127.0.0.1'}") <util:properties id="config" location="classpath:config.properties"/>

@Value with methods : When the @Value annotation is found on a method, Spring context will invoke it when all the spring configurations and beans are getting loaded. If the method has multiple arguments, then every argument value is mapped from the method annotation. If we want different values for different arguments then we can use @Value annotation directly with the argument.

@Value("Test")

public void printValues(String s, String v){} //both 's' and 'v' values will be 'Test'

@Value("Test")

public void printValues(String s, @Value("Data") String v){}// s=Test, v=Data

**version of spring you used**

**Spring 3.0** - released in 2009. It made full-fledged use of improvements in Java5 and also provided support to JEE6.

**recently Spring 4.0** - This version was released in 2013. This was the first version to provide full support to Java 8.

**What are the different features of Spring Framework?**

1. **Lightweight**: Spring is lightweight when it comes to size and transparency.
2. **Inversion of control (IOC):** The objects give their dependencies instead of creating or looking for dependent objects. This is called Inversion Of Control.
3. **Aspect oriented Programming (AOP):** Aspect oriented programming in Spring supports cohesive development by separating application business logic from system services.
4. **Container:** Spring Framework creates and manages the life cycle and configuration of the application objects.
5. **MVC Framework:** Spring Framework’s MVC web application framework is highly configurable. Other frameworks can also be used easily instead of Spring MVC Framework.
6. **Transaction Management:** Generic abstraction layer for transaction management is provided by the Spring Framework. Spring’s transaction support can be also used in container less environments.
7. **JDBC Exception Handling:** The JDBC abstraction layer of the Spring offers an exception hierarchy, which simplifies the error handling strategy

**Dependency Injection:** In Dependency Injection, you do not have to create your objects but have to describe how they should be created. You don’t connect your components and services together in the code directly, but describe which services are needed by which components in the configuration file. The IoC container will wire them up together

**DI in Spring Framework: only constructor and setter injections are used.**

|  |  |
| --- | --- |
| **Constructor Injection** | **Setter Injection** |
| There is no partial injection. | There can be partial injection. |
| It doesn’t override the setter property. | It overrides the constructor property. |
| It will create a new instance if any modification is done. | It will not create new instance if any modification is done. |
| It works better for many properties. | It works better for few properties. |
| public class Bar {  private Foo foo;  public Bar(Foo foo){  this.foo = foo;  }  }  <bean id="BarBean" class="com.javacodegeeks.Bar">  <**constructor-arg** ref="fooBean" />  </bean> | public class Bar {  private Foo foo;  public void setFoo(Foo foo){  this.foo = foo;  }  }  <bean id="barBean" class="com.javacodegeeks.Bar">  <**property name**="foo">  <ref bean="fooBean" />  </property>  </bean> |

**Can you inject null and empty string values in spring? 🡪** Yes

**Spring Bean AutoWiring**

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

Autowiring can't be used to inject primitive and string values. It works with reference only.

**Advantages:** It requires the less code because we don't need to write the code to inject the dependency explicitly.

**Disadvantage:** No control of programmer. It can't be used for primitive and string values.

**Autowiring Modes**

|  |  |  |
| --- | --- | --- |
| **No** | **Mode** | **Description** |
| 1 | No/default | It is the default autowiring mode. By default spring bean autowiring is turned off. Spring bean autowire default value is “default” that means no autowiring is to be performed. autowire value “no” also have the same behavior. |
| 2 | byname | The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method. |
| 3 | byType | The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method. |
| 4 | constructor | The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters. |
| 5 | autodetect | It is deprecated since Spring 3. |
| 6 | @Autowired annotation | We can use Spring @Autowired annotation for spring bean autowiring. @Autowired annotation can be applied on variables and methods for autowiring byType.  @Autowired  private Employee employee;    @Autowired  public void setEmployee(Employee emp){  this.employee=emp;  }  We can also use @Autowired annotation on constructor for constructor based spring autowiring. |

For @Autowired annotation to work, we also need to enable annotation based configuration in spring bean configuration file. This can be done **by context:annotation-config** element or by defining a bean of type **org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor**.

**Inversion of Control (IoC)**

* IoC is the mechanism to achieve loose-coupling between Objects dependencies. To achieve loose coupling and dynamic binding of the objects at runtime, objects dependencies are injected by other assembler objects
* **At the core of the Spring Framework, lies the Spring container. The container creates the object, wires them together, configures them and manages their complete life cycle. The Spring container makes use of Dependency Injection to manage the components that make up an application. The container receives instructions for which objects to instantiate, configure, and assemble by reading the configuration metadata provided. This metadata can be provided either by XML, Java annotations or Java code.**
* **Types of IOC containers are there in spring? 🡪 BeanFactory and ApplicationContext**

|  |  |
| --- | --- |
| **BeanFactory** | **ApplicationContext** |
| It is an interface defined in org.springframework.beans.factory.**BeanFactory** | It is an interface defined in org.springframework.context.**ApplicationContext** |
| It uses Lazy initialization | It uses Eager/ Aggressive initialization |
| It explicitly provides a resource object using the syntax | It creates and manages resource objects on its own |
| It doesn’t supports internationalization | It supports internationalization |
| It doesn’t supports annotation based dependency | It supports annotation based dependency |

Some of the useful ApplicationContext implementations that we use are

* **AnnotationConfigApplicationContext:** If we are using Spring in standalone java applications and using annotations for Configuration, then we can use this to initialize the container and get the bean objects.
* **ClassPathXmlApplicationContext**: If we have spring bean configuration xml file in standalone application, then we can use this class to load the file and get the container object.
* **FileSystemXmlApplicationContext**: This is similar to ClassPathXmlApplicationContext except that the xml configuration file can be loaded from anywhere in the file system.
* **AnnotationConfigWebApplicationContext** and **XmlWebApplicationContext** for web applications.

Usually, if you are working on Spring MVC application and your application is configured to use Spring Framework, Spring IoC container gets initialized when the application starts and when a bean is requested, dependencies are injected automatically. However, for a standalone application, you need to initialize the container somewhere in the application and then use it to get the spring beans.

**Spring Bean Configuration: three ways to configure beans to be used in the application.**

* 1. **Annotation Based Configuration:** By using @Service or @Component annotations. Scope details can be provided with @Scope annotation.
  2. **XML Based Configuration:** By creating Spring Configuration XML file to configure the beans. If you are using Spring MVC framework, the xml based configuration can be loaded automatically by writing some boiler plate code in web.xml file.
  3. **Java Based Configuration:** Starting from Spring 3.0, we can configure Spring beans using java programs. Some important annotations used for java based configuration are @Configuration, @ComponentScan and @Bean.

**What do you understand by auto-configuration in Spring Boot and how to disable the auto-configuration?**

* Auto-configuration is used to automatically configure the required configuration for the application. For example, if you have a data source bean present in the classpath of the application, then it automatically configures the JDBC template. With the help of auto-configuration, you can create a Java application in an easy way, as it automatically configures the required beans, controllers, etc.
* To disable the auto-configuration property, you have to exclude attribute of @EnableAutoConfiguration, in the scenario where you do not want it to be applied.

**@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})**

* If the class is not on the classpath, then to exclude the auto-configuration, you have to mention the following code:

@EnableAutoConfiguration(excludeName={Sample.class})

* Apart from this, Spring Boot also provides the facility to exclude list of auto-configuration classes by using the spring.autoconfigure.exclude property. You can go forward, and add it either in the application.properties or add multiple classes with comma-separated.

**What are the differences between @SpringBootApplication and @EnableAutoConfiguration annotation?**

|  |  |
| --- | --- |
| **@SpringBootApplication** | **@EnableAutoConfiguration** |
| Used in the main class or bootstrap class | Used to enable auto-configuration  and component scanning in your project |
| It is a combination of @Configuration, @ComponentScan and @EnableAutoConfiguration annotations. | It is a combination of @Configuration and @ComponentScan annotations |

**What are the steps to deploy Spring Boot web applications as JAR and WAR files?**

* To deploy a Spring Boot web application, you just have to add the following plugin in the pom.xml file:

|  |  |
| --- | --- |
|  | <plugin>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-maven-plugin</artifactId>  </plugin> |

* By using the above plugin, you will get a JAR executing the package phase. This JAR will contain all the necessary libraries and dependencies required. It will also contain an embedded server. So, you can basically run the application like an ordinary JAR file.  
  **Note:** The packaging element in the pom.xml file must be set to **jar** to build a JAR file as below:

|  |  |
| --- | --- |
|  | <packaging>jar</packaging> |

* Similarly, if you want to build a WAR file, then you will mention

|  |  |
| --- | --- |
|  | <packaging>war</packaging> |

**Can you explain how to deploy to a different server with Spring Boot? 🡪** Generate a war and deploy to another server

**What is the best way to expose custom application configuration with Spring Boot?**

One way to expose the custom application configuration in Spring Boot is by using the @Value annotation. But, the only problem with this annotation is that all the configuration values will be distributed throughout the application. Instead, you can use a centralized approach.

By centralized approach, I mean that you can define a configuration component using the @**ConfigurationProperties** as follows:

@Component

@ConfigurationProperties("example")

public class SampleConfiguration {

private int number;

private boolean value;

private String message;

According to the above snippet, the values configured in application.properties will be as follows:

example.number: 100

example.value: true

example.message: Dynamic Message

**What do you think is the need for Profiles?**

Profiles are used to provide a way to segregate the different parts of the application configuration and make it available for various environments. So, basically, any @Component or a @Configuration can be marked with @Profile to limit as it is loaded.

Since it is a known fact that a Profile is nothing but a key to identify an environment let’s consider the following two profiles in the example

* Dev
* Prod
* Consider the following properties present in the application properties file:

example.number: 100  
example.value: true  
example.message: Dynamic Message

Now, say you want to customize the application.properties for dev profile, then you need to create a file with name application-dev.properties and override the properties that you want to customize. You can mention the following code:

example.message: Dynamic Message in Dev

Similarly, if you want to customize the application.properties for prod profile, then you can mention the following code snippet:

example.message: Dynamic Message in Prod

Once you are done with the profile-specific configuration, you have to set the active profile in an environment. To do that, either you can

* Use -Dspring.profiles.active=prod in  arguments
* Use spring.profiles.active=prod in application.properties file

**How to instruct an auto-configuration to back off when a bean exists? 🡪** @ConditionalOnMissingBean annotation

**What do you understand by Spring Boot supports relaxed binding?**

Relaxed binding, is a way in which, **the property name does not need to match the key of the environment property**. In Spring Boot, relaxed binding is applicable to the type-safe binding of the configuration properties. For example, if a property in a bean class with the **@ConfigurationProperties annotation** is used sampleProp, then it can be bounded to any of the following environment properties:

* sampleProp
* sample-Prop
* sample\_Prop
* SAMPLE\_PROP

**Can you control logging with Spring Boot? How?**  
Yes, we can control logging with Spring Boot by specifying log levels on application.properties file. Spring Boot loads this file when it exists in the [classpath](http://www.java67.com/2012/08/what-is-path-and-classpath-in-java-difference.html) and it can be used to configure both Spring Boot and application code.  
Spring Boot uses Commons Logging for all internal logging and you can change log levels by adding following lines in the application.properties file:  
logging.level.org.springframework=DEBUG  
logging.level.com.demo=INFO

**What are the ways in which SpringBoot implements hot deployment?**

* 1. Spring Loaded
  2. Spring-boot-devtools

### What is “Transitively Dependency Resolution Management”?

### “Transitively Dependency Resolution Management” means: If we define an “A” dependency in build scripts, “A” is dependent on “B” and “B” is dependent on “C”, That means “A” is also dependent on “C”. Then Build Tools will download and add all Three Jar files “A”, “B” and “C” to our application classpath.

### What are the core configuration files for Spring Boot? What is the difference between them?

### The core configuration files for Spring Boot are the application and bootstrap configuration files.

### The application configuration file is easy to understand and is primarily used for automated configuration of Spring Boot projects.

### The bootstrap configuration file has the following application scenarios.

### When using Spring Cloud Config to configure the hub, you need to add the configuration properties of the connection to the configuration center in the bootstrap configuration file to load the configuration information of the external configuration center.

### Some fixed attributes that cannot be overridden;

### Some encryption/decryption scenarios;

### What is JavaConfig?

### Spring JavaConfig is a product of the Spring community that provides pure Java methods for configuring Spring IoC containers. So it helps to avoid using XML configuration. The advantages of using JavaConfig are:

1. **Object-oriented configuration**. Since the configuration is defined as a class in JavaConfig, users can take advantage of the object-oriented features in Java. A configuration class can inherit from another, override its @Bean method, and so on.
2. **Reduce or eliminate XML configuration**. The benefits of externalization configuration based on the principle of dependency injection have been proven. However, many developers don’t want to switch back and forth between XML and Java. JavaConfig provides developers with a pure Java way to configure Spring containers similar to the XML configuration concept. From a technical point of view, it is possible to configure the container using only the JavaConfig configuration class, but in fact many people think that it is ideal to mix JavaConfig and XML.
3. **Type safety and refactoring friendly**. JavaConfig provides a type-safe way to configure the Spring container. Thanks to Java 5.0 support for generics, it is now possible to retrieve beans by type rather than by name, without any cast or string-based lookups.

### What are the formats of Spring Boot configuration files? What is the difference between them?

### The main difference between them is the writing format.

### In addition, .yml format does not support @PropertySourceannotations import configuration.

**Which is the core annotation of Spring Boot? Which annotations is it mainly composed of?**

### The annotation above the startup class is @SpringBootApplication, which is also a core annotation for Spring Boot. The main combination contains the following three annotations:

### @SpringBootConfiguration: Combines the @Configuration annotation to implement the configuration file.

### @EnableAutoConfiguration: Turns on automatic configuration. You can also turn off an automatic configuration option, such as turning off data source auto-configuration: @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class }).

### @ComponentScan: Spring component scan.

**What are the ways to enable Spring Boot features?**

### Inherit the spring-boot-starter-parent project

### Import spring-boot-dependencies project dependencies

**How do I run some specific code when Spring Boot starts?**

### You can implement the interface ApplicationRunner or CommandLineRunner. These two interfaces are implemented in the same way. They only provide a run method.

**Can Spring Boot be compatible with older Spring projects? How do I do this?**

### Compatible, use @ImportResourceannotations to import old Spring project configuration file.

**[How Might You Implement Spring Security In Spring Boot Application?](https://www.onlineinterviewquestions.com/spring-boot-interview-questions/" \l "collapseUnfiled15)**

### Usage of spring security in Spring boot application requires quite a little configuration. You have to include spring-boot-starter-security starter in pom.xml. You need to make spring config class, which will expand WebSecurity Configure Adapter and override expected strategy to accomplish security in Spring boot application.

**DI internally used Java Reflection API**

**How to get ServletContext and ServletConfig object in a Spring Bean?**

### Implementing Spring \*Aware interfaces, for these ServletContextAware and ServletConfigAware interfaces, for complete example of these aware interfaces, please read Spring Aware Interfaces

### Using @Autowired annotation with bean variable of type ServletContext and ServletConfig. They will work only in servlet container specific environment only though.

### @Autowired

### ServletContext servletContext;

### *How to handle exceptions in Spring MVC Framework?*

### *Spring MVC Framework provides the following ways to help us achieving robust exception handling.*

### *Controller Based – We can define exception handler methods in our controller classes. All we need is to annotate these methods with @ExceptionHandler annotation.*

### *Global Exception Handler – Exception Handling is a cross-cutting concern and Spring provides @ControllerAdvice annotation that we can use with any class to define our global exception handler.*

### *HandlerExceptionResolver implementation – For generic exceptions, most of the times we serve static pages. Spring Framework provides HandlerExceptionResolver interface that we can implement to create global exception handler. The reason behind this additional way to define global exception handler is that Spring framework also provides default implementation classes that we can define in our spring bean configuration file to get spring framework exception handling benefits.*

### *What are the minimum configurations needed to create Spring MVC application?*

### *Add spring-context and spring-webmvc dependencies in the project.*

### *Configure DispatcherServlet in the web.xml file to handle requests through spring container.*

### *Spring bean configuration file to define beans, if using annotations then it has to be configured here. Also we need to configure view resolver for view pages.*

### *Controller class with request mappings defined to handle the client requests*

### *How can we use Spring to create Restful Web Service returning JSON response? --> @ResponseBody*

### *How to use Tomcat JNDI DataSource in Spring Web Application?*

### *For using servlet container configured JNDI DataSource, we need to configure it in the spring bean configuration file and then inject it to spring beans as dependencies. Then we can use it with JdbcTemplate to perform database operations.*

### *Sample configuration would be:*

### *<beans:bean id="dbDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">*

### *<beans:property name="jndiName" value="java:comp/env/jdbc/MyLocalDB"/>*

### *</beans:bean>*

### *How would you achieve Transaction Management in Spring?*

### *Spring framework provides transaction management support through Declarative Transaction Management as well as programmatic transaction management. Declarative transaction management is most widely used because it’s easy to use and works in most of the cases.*

### *We use annotate a method with @Transactional annotation for Declarative transaction management. We need to configure the transaction manager for the DataSource in the spring bean configuration file.*

### *<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">*

### *<property name="dataSource" ref="dataSource" />*

### *</bean>*

### *How to inject a java.util.Properties into a Spring Bean?*

### *We need to define propertyConfigurer bean that will load the properties from the given property file. Then we can use Spring EL support to inject properties into other bean dependencies. For example;*

### *<bean id="propertyConfigurer" class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">*

### *<property name="location" value="/WEB-INF/application.properties" />*

### *</bean>*

### *<bean class="com.journaldev.spring.EmployeeDaoImpl">*

### *<property name="maxReadResults" value="${results.read.max}"/>*

### *</bean>*

### *If you are using annotation to configure the* *spring bean, then you can inject property like below.*

*@Value("${maxReadResults}") private int maxReadResults;*

**If any starter has its own configuration class , i dont want it and instead write own configuration class how to do 🡪** *@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class}) and @configuration*

**Exclude depencdency in spring boot**

*<dependency>*

*<groupId>org.springframework.boot</groupId>*

*<artifactId>spring-boot-starter-web</artifactId>*

*<exclusions>*

*<exclusion>*

*<groupId>org.springframework.boot</groupId>*

*<artifactId>spring-boot-starter-tomcat</artifactId>*

*</exclusion>*

*</exclusions>*

*</dependency>*

**How to know which bean is loaded?**

*1) Use ApplicationContext.getBeanDefinitionNames() to find the name of all loaded beans*

*2) Use ApplicationContext.getBean(beanName) to get bean including its runtime type information.*

@SpringBootApplication

public class SpringBootWebApplication extends SpringBootServletInitializer implements CommandLineRunner {

@Override

protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {

return application.sources(SpringBootWebApplication.class);

}

public static void main(String[] args) throws Exception {

SpringApplication.run(SpringBootWebApplication.class, args);

}

@Autowired

private ApplicationContext appContext;

@Override

public void run(String... args) throws Exception

{

String[] beans = appContext.getBeanDefinitionNames();

Arrays.sort(beans);

for (String bean : beans)

{

System.out.println(bean + " of Type :: " + appContext.getBean(bean).getClass());

}

}

}

**@Resource annotation in spring**

The @Resource annotation in spring performs the autowiring functionality. This annotation follows the autowire=byName semantics in the XML based configuration i.e. it takes the name attribute for the injection. Below snippet shows how to use this annotation.

import javax.annotation.Resource;

public class Employee {

private String id;

private String name;

@Resource(name="mycompany")

private Company company;

}

This annotation takes an optional name argument. In case no name attribute is specified with this annotation, the default name is interpreted from the field-name or the setter method (i.e. the bean property name). Always remember that if the @Resource annotation doesn’t find the bean with the name it will automatically switch it’s autowiring technique to autowire=byType (i.e. @Autowired annotation).

**Spring circular dependency problem.**

* It happens when a bean A depends on another bean B, and the bean B depends on the bean A as well:

Bean A → Bean B → Bean C → Bean A

* When Spring context is loading all the beans, it tries to create beans in the order needed for them to work completely. For instance, if we didn’t have a circular dependency, like the following case:

Bean A → Bean B → Bean C

Spring will create bean C, then create bean B (and inject bean C into it), then create bean A (and inject bean B into it)

* Spring throws BeanCurrentlyInCreationException in that situation

**Fixing circular dependencies :**

1. setter injection : This way Spring creates the beans, but the dependencies are not injected until they are needed.
2. @Lazy at constructor injection point : instead of fully initializing the bean, it will create a proxy to inject it into the other bean. The injected bean will only be fully created when it’s first needed. BeanA(@Lazy BeanB b) {..}
3. Implement ApplicationContextAware and InitializingBean
4. Use @PostConstruct and @Autowired

**Spring Boot CommandLineRunner and ApplicationRunner**

Whenever you need, to perform a specific task once after all the Spring Beans are created and the Application Context has been created then you can implement ApplicationRunner or CommandLineRunner interface. Both the interfaces works the same way and has a single run() method. A similar implementation is already there in Spring and many people would have known it JobLauncherCommandLineRunner which is used for running jobs in Spring Batch.

**Difference between CommandLineRunner and ApplicationRunner**

Both of them provides the same functionality and the only difference between CommandLineRunner and ApplicationRunner is CommandLineRunner.run() accepts String array[] whereas ApplicationRunner.run() accepts ApplicationArguments as argument.

**@Order Annotation Spring Boot**

Whenever you have more than one class implementing the CommandLineRunner / ApplicationRunner, then you can use the @Order annotation mention which run() method has to be executed first

@Order(1)

@Order(2)

**Where do you need @EnableWebMVC?**

The @EnableWebMvc annotation is required to enable Spring MVC when Java configuration is used to configure Spring MVC instead of XML. It is equivalent to <mvc: annotation-driven> in an XML configuration.It enables support for the @Controller-annotated classes that use @RequestMapping to map incoming requests to handler methods

**Swap Column values (say gender) from F to M and from M to F**

UPDATE tbl SET Gender = (CASE WHEN Gender = 'M' THEN 'F' ELSE 'M' END) where Gender in ('F','M')

**Find Nth maximum using single database query.**

*SELECT name, salary FROM #Employee e1 WHERE N-1 = (*

*SELECT COUNT(DISTINCT salary) FROM #Employee e2 WHERE e2.salary > e1.salary)*

**Get the nth highest record example**

For example, if you want to get the second most expensive product (n = 2) in the products table, you use the following query:

*SELECT productCode, productName, buyPrice FROM products ORDER BY buyPrice DESC LIMIT 1 , 1;*

**Insert into temp Select 1 union select 2 union select 3 (postgress)** *The UNION operator combines result sets of two or more SELECT statements into a single result set. The UNION operator removes all duplicate rows unless the UNION ALL is used.*

***How many object will be created in 3 class inheritance*** *🡪 only 1*

***Select manager name and sum of salary from employees 🡪*** *SELECT manager\_name, SUM(salary) FROM employees GROUP BY manager\_name;*

***How to write custom scope in spring 🡪*** *by implementing scope interface*

***What are the types of the transaction management that is supported by spring?***

1. *Programmatic transaction management:*
   1. *Using the TransactionTemplate (Recommended by Spring Team)*
   2. *Using a PlatformTransactionManager implementation directly:*
2. *Declarative Transaction (Usually used almost in all scenarios of any web application) : we used this in NTP*

***Explain the RowCallbackHandler in Spring?***

*In order to navigate through the records we generally go for ResultSet. But spring provides an interface that handles this entire burden and leaves the user to decide what to do with each row. The interface provided by spring is RowCallbackHandler. There is a method processRow() which needs to be implemented so that it is applicable for each and every row.*

*void processRow(java.sql.ResultSet rs);*

*public static void main(String args[]) throws Exception {*

*ApplicationContext ac = new ClassPathXmlApplicationContext("context.xml", Main.class);*

*DataSource dataSource = (DataSource) ac.getBean("dataSource");*

*// DataSource mysqlDataSource = (DataSource) ac.getBean("mysqlDataSource");*

*JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);*

*jdbcTemplate.query("select first\_name from customer", new RowCallbackHandler() {*

*public void processRow(ResultSet resultSet) throws SQLException {*

*while (resultSet.next()) {*

*// send email to resultSet.getString(1)*

*} }*

*});*

*}*