```
class Employee implements Comparable<Employee> {
    String name;
    int age;
    Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public int compareTo(Employee emp) {
        return this.name.compareTo(emp.name);
    }
    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age + "]";
    }
}
public class TestTreeMap {
    public static void main(String[] args) {

        TreeMap<Employee, Integer> map = new TreeMap<>();

        map.put(new Employee("Mike", 20), 100);
        map.put(new Employee("John", 10), 500);
        map.put(new Employee("Ryan", 15), 200);
        map.put(new Employee("Lisa", 20), 400);

        map.forEach((k,v) -> System.out.println(k + ":" + v));
    }
}
```

Here, we are sorting based on Employee name,

Output:

```
Employee [name=John, age=10]:500
Employee [name=Lisa, age=20]:400
Employee [name=Mike, age=20]:100
Employee [name=Ryan, age=15]:200
```

Let's look at a program where we pass a Comparator in the TreeMap constructor, and sort the Employee object's based on age in descending order:

Program 4:

```
class Employee {
    String name;
    int age;
    Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age + "]";
    }
}
```

```java
public class TestTreeMap {
    public static void main(String[] args) {
        TreeMap<Employee, Integer> map = new TreeMap<>(
                new Comparator<Employee>() {
                    @Override
                    public int compare(Employee e1, Employee e2) {
                        if(e1.age < e2.age){
                            return 1;
                        } else if(e1.age > e2.age) {
                            return -1;
                        }
                        return 0;
                    }
                }
            );

        map.put(new Employee("Mike", 20), 100);
        map.put(new Employee("John", 10), 500);
        map.put(new Employee("Ryan", 15), 200);
        map.put(new Employee("Lisa", 40), 400);

        map.forEach((k,v) -> System.out.println(k + ":" + v));
    }
}
```

Output:

```
Employee [name=Lisa, age=40]:400
Employee [name=Mike, age=20]:100
Employee [name=Ryan, age=15]:200
Employee [name=John, age=10]:500
```

Here, in Employee class, I have not implemented equals() and hashCode()

TreeMap's Javadoc:

```java
public class TreeMap<K,V>
    extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable, java.io.Serializable
{
    /**
     * The comparator used to maintain order in this tree map, or
     * null if it uses the natural ordering of its keys.
     *
     * @serial
     */
    private final Comparator<? super K> comparator;

    private transient Entry<K,V> root;
```

No-arg TreeMap constructor:

```java
public TreeMap() {
    comparator = null;
}
```

TreeMap constructor which takes comparator object:

```java
public TreeMap(Comparator<? super K> comparator) {
    this.comparator = comparator;
}
```

TreeMap.put() method excerpt:

```
public V put(K key, V value) {
    Entry<K,V> t = root;
    if (t == null) {
        compare(key, key); // type (and possibly null) check

        root = new Entry<>(key, value, null);
        size = 1;
        modCount++;
        return null;
    }
    int cmp;
    Entry<K,V> parent;
    // split comparator and comparable paths
    Comparator<? super K> cpr = comparator;
    if (cpr != null) {
        do {
            parent = t;
            cmp = cpr.compare(key, t.key);
            if (cmp < 0)
                t = t.left;
            else if (cmp > 0)
                t = t.right;
            else
                return t.setValue(value);
        } while (t != null);
    }
```

Question 110: Explain Java's TreeSet

Answer: TreeSet class is one of the implementation of Set interface

Some points to remember:

- TreeSet class contains unique elements just like HashSet
- TreeSet class does not allow null elements
- TreeSet class is not synchronized
- TreeSet class internally uses TreeMap, i.e. the value added in TreeSet is internally stored in the key of TreeMap
- TreeSet elements are ordered using their natural ordering or by a Comparator which can be provided at the set creation time
- TreeSet provides guaranteed log(n) time cost for the basic operations (add, remove and contains)
- TreeSet iterator is fail-fast in nature

TreeSet Javadoc:

```
public class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable


public TreeSet() {
    this(new TreeMap<E,Object>());
}


public TreeSet(Comparator<? super E> comparator) {
    this(new TreeMap<>(comparator));
}
```

Question 111: Difference between fail-safe and fail-fast iterators

Answer: Iterators in Java are used to iterate over the Collection objects.

Fail-fast iterators: immediately throw *ConcurrentModificationException,* if the collection is modified while iterating over it. Iterator of ArrayList and HashMap are fail-fast iterators.

All the collections internally maintain some sort of array to store the elements, Fail-fast iterators fetch the elements from this array. Whenever, we modify the collection, an internal field called modCount is updated. This modCount is used by Fail-safe iterators to know whether the collection is structurally modified or not. Every time when the Iterator's next() method is called, it checks the modCount. If it finds that modCount has been updated after the Iterator has been created, it throws ConcurrentModificationException.

Program 1:

```
public class FailFastIteratorTest {
    public static void main(String[] args) {

        ArrayList<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);

        Iterator<Integer> itr = list.iterator();
        while(itr.hasNext()) {
            System.out.println(itr.next());
            list.add(4);
        }
    }
}
```

Output:

```
1Exception in thread "main"
java.util.ConcurrentModificationException
        at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)
        at java.util.ArrayList$Itr.next(ArrayList.java:859)
        at com.iterator.demo.FailFastIteratorTest.main(FailFastIteratorTest.java:16)
```

But they don't throw the exception, if the collection is modified by Iterator's remove() method.

Program 2:

```
public class FailFastIteratorTest {
    public static void main(String[] args) {

        ArrayList<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        System.out.println("List: " + list);
        Iterator<Integer> itr = list.iterator();
        while(itr.hasNext()) {
            System.out.println(itr.next());
            itr.remove();
        }
        System.out.println("List: " + list);
    }
}
```

Output:

```
List: [1, 2, 3]
1
2
3
List: []
```

Javadoc:

arrayList.iterator() method:

```
public Iterator<E> iterator() {
    return new Itr();
}
```

Itr is a private nested class in ArrayList:

```
private class Itr implements Iterator<E> {
    int cursor;     // index of next element to return
    int lastRet = -1; // index of last element returned; -1 if no such
    int expectedModCount = modCount;

    Itr() {}

    public boolean hasNext() {
        return cursor != size;
    }
```

Itr.next() method:

```
@SuppressWarnings("unchecked")
public E next() {
    checkForComodification();
    int i = cursor;
    if (i >= size)
        throw new NoSuchElementException();
    Object[] elementData = ArrayList.this.elementData;
    if (i >= elementData.length)
        throw new ConcurrentModificationException();
    cursor = i + 1;
    return (E) elementData[lastRet = i];
}
```

See the first statement is a call to checkForComodification():

```
final void checkForComodification() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
}
```

On the other hand, ***Fail-safe iterators*** does not throw *ConcurrentModificationException*, because they operate on the clone of the collection, not the actual collection. This also means that any modification done on the actual collection goes unnoticed by these iterators. The last statement is not always true though, sometimes it can happen that the iterator may reflect modifications to the collection after the iterator is created. But there is no guarantee of it. CopyOnWriteArrayList, ConcurrentHashMap are the examples of fail-safe iterators.

Program 1: ConcurrentHashMap example

```java
public class FailSafeIteratorTest {
    public static void main(String[] args) {
        ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();
        map.put(1, "Mike");
        map.put(2, "John");
        map.put(3, "Lisa");

        Iterator<Integer> itr = map.keySet().iterator();
        while(itr.hasNext()) {
            Integer key = itr.next();
            System.out.println(key + " : " + map.get(key));
            map.put(4, "Ryan");
        }
        System.out.println("Map: " + map);
    }
}
```

Output:

```
1 : Mike
2 : John
3 : Lisa
4 : Ryan
Map: {1=Mike, 2=John, 3=Lisa, 4=Ryan}
```

Here, iterator is reflecting the element which was added during the iteration operation.

Program 2: CopyOnWriteArrayList example

```java
public class FailSafeIteratorTest {
    public static void main(String[] args) {
        CopyOnWriteArrayList<Integer> list = new CopyOnWriteArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);

        Iterator<Integer> itr = list.iterator();
        while(itr.hasNext()) {
            System.out.println(itr.next());
            list.add(4);
        }
        System.out.println("List: " + list);
    }
}
```

Output:

```
1
2
3
List: [1, 2, 3, 4, 4, 4]
```

Question 112: Difference between Iterator and ListIterator

Answer:

- Iterator can traverse the collection only in one direction i.e. forward direction but ListIterator can traverse the list in both directions, forward as well as backward, using previous() and next() method

- Iterator cannot add element to a collection while iterating over it, but ListIterator can add elements while iterating over the list
- Iterator cannot modify an element while iterating over a collection, but ListIterator has set(E e) method which can be used to modify the element
- Iterator can be used with List, Set or Map, but ListIterator only works with List
- Iterator has no method to obtain an index of the collection elements but ListIterator has methods like previousIndex() and nextIndex() which can be used to obtain the index

Question 113: Difference between Iterator.remove and Collection.remove()

Answer: Iterator.remove() does not throw ConcurrentModificationException while iterating over a collection but Collection.remove() method will throw ConcurrentModificationException.

Java Collection framework is very important topic when preparing for the interviews, apart from the above questions, you can read about Stack, Queue topics also, but if you are short on time, what we have discussed so-far should be enough for the interview.

Question 114: What is the difference between a Monolith and Micro-service architecture?

Answer: In monolithic architecture, applications are built as one large system, whereas in micro-service architecture we divide the application into modular components which are independent of each other.

Monolithic architecture has some advantages:

- Development is quite simple
- Testing a monolith is also simple, just start the application and do the end-to-end testing, Selenium can be used to do the automation testing
- These applications are easier to deploy, as only one single jar/war needs to be deployed
- Scaling is simple when the application size is small, we just have to deploy one more instance of our monolith and distribute the traffic using a load balancer
- Network latency is very low/none because of one single codebase

However, there are various disadvantages of monolith architecture as well:

- Rolling out a new version means redeploying the entire application
- Scaling a monolith application becomes difficult once the application size increases. It also becomes difficult to manage
- The size of the monolith can slow down the application start-up and deployment time
- Continuous deployment becomes difficult
- A bug in any module can bring down the entire application
- It is very difficult to adopt any new technology in a monolith application, as it affects the whole application, both in terms of time and cost

Micro-service architecture gives following advantages:

- Micro-services are easier to manage as they are relatively smaller in size
- Scalability is a major advantage of using a micro-service architecture, each micro-service can be scaled independently
- Rolling out a new version of micro-service means redeploying only that micro-service
- A bug in micro-service will affect only that micro-service and its consumers, not the entire application
- Micro-service architecture is quite flexible. We can choose different technologies for different micro-services according to business requirements
- It is not that difficult to upgrade to newer technology versions or adopt a newer technology as the codebase is quite smaller (this point is debatable in case the micro-service becomes very large)
- Continuous deployment becomes easier as we only have to re-deploy that micro-service

Despite all these advantages, Micro-services also comes with various disadvantages:

- As micro-services are distributed, it becomes complex compared to monolith. And this complexity increases with the increase in micro-services
- As micro-services will need to communicate with each other, it increases the network latency. Also, extra efforts are needed for a secure communication between micro-services
- Debugging becomes very difficult as one micro-service will be calling other micro-services and to figure out which micro-service is causing the error, becomes a difficult task in itself

- Deploying a micro-service application is also a complex task, as each service will have multiple instances and each instance will need to be configured, deployed, scaled and monitored
- Breaking down a large application into individual components as micro-services is also a difficult task

We have discussed both advantages and disadvantages of monolith and micro-services, you can easily figure out the differences between them, however, I am also stating them below.

The differences are:

- In a monolithic architecture, if any fault occurs, it might bring down the entire application as everything is tightly coupled, however, in case of micro-service architecture, a fault affects only that micro-service and its consumers
- Each micro-service can be scaled independently according to requirement. For example, if you see that one of your micro-service is taking more traffic, then you can deploy another instance of that micro-service and then distribute the traffic between them. Now, with the help of cloud computing services such as AWS, the applications can be scaled up and down automatically. However, in case of monolith, even if we want to scale one service within the monolith, we will have to scale the entire monolith
- In case of monolith, the entire technology stack is fixed at the start. It will be very difficult to change the technology at a later stage in time. However, as micro-services are independent of each other, they can be coded in any language, taking the advantage of different technologies according to the use-case. So micro-services gives you the freedom to choose different technologies, frameworks etc.
- Deploying a new version of a service in monolith requires more time and it increases the application downtime, however, micro-services entails comparatively lesser downtime

Question 115: What is Dependency Injection in Spring?

Answer: Dependency Injection is the most important feature of Spring framework. Dependency Injection is a design pattern where the dependencies of a class are injected from outside, like from an xml file. It ensures loose-coupling between classes.

In a Spring MVC application, the controller class has dependency of service layer classes and the service layer classes have dependencies of DAO layer classes.

Suppose class A is dependent on class B. In normal coding, you will create an object of class B using 'new' keyword and call the required method of class B. However, what if you can tell someone to pass the object of class B in class A? Dependency injection does this. You can tell Spring, that class A needs class B object and Spring will create the instance of class B and provide it in class A.

In the above example, we can see that we are passing the control of objects to Spring framework, this is called Inversion of Control (IOC) and Dependency injection is one of the principles that enforce IOC.

Question 116: What are the different types of Dependency Injection?

Answer: Spring framework provides 2 ways to inject dependencies:

- By Constructor
- By Setter method

Constructor-based DI: when the required dependencies are provided as arguments to the constructor, then it is known as constructor-based dependency injection, see the examples below:

Using XML based configuration:

Injecting a dependency is done through the bean-configuration file, for this <constructor-arg> xml tag is used:

```
<bean id="classB" class="com.dependency.demo.B" />

<bean id="classA" class="com.dependency.demo.A">
    <constructor-arg ref="classB" />
</bean>
```

In case of more than 1 dependency, the order sequence of constructor arguments should be followed to inject the dependencies.

Java Class A:

```
package com.dependency.demo;

public class A {

    B b;

    public A (B b) {
        this.b = b;
    }

}
```

Java Class B:

```
package com.dependency.demo;

public class B {

}
```

Using Java Based Configuration:

When using Java based configuration, the constructor needs to be annotated with @Autowired annotation to inject the dependencies,

Our classes A and B will be annotated with @Component (or any other stereotype annotation), so that they will be managed by Spring.

```java
package com.dependency.demo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class A {

    B b;

    @Autowired
    public A (B b) {
        this.b = b;
    }

}
```

```java
package com.dependency.demo;
import org.springframework.stereotype.Component;

@Component
public class B {

}
```

Before Spring version 4.3, @Autowired annotation was needed for constructor dependency injection, however, in newer Spring versions, @Autowired is optional, if the class has only one constructor.

But, if the class has multiple constructors, we need to explicitly add @Autowired to one of the constructors so that Spring knows which constructor to use for injecting the dependencies.

Setter-method injection: in this, the required dependencies are provided as the field parameters to the class and the values are set using setter methods of those properties. See the examples below.

Using XML based configuration:

Injecting a dependency is done through the bean configuration file and <property> xml tag is used where 'name' attribute defines the name of the field of java class.

```xml
<bean id="classB" class="com.dependency.demo.B" />

<bean id="classA" class="com.dependency.demo.A">
    <property name="b">
        <ref bean="classB" />
    </property>
</bean>
```

Java class A:

```java
package com.dependency.demo;
public class A {

    B b;

    public void setB (B b) {
        this.b = b;
    }

}
```

Java class B:

```java
package com.dependency.demo;

public class B {

}
```

Using Java based configuration:

The setter method needs to be annotated with @Autowired annotation.

```java
package com.dependency.demo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class A {

    B b;

    @Autowired
    public void setB (B b) {
        this.b = b;
    }

}
```

```java
package com.dependency.demo;
import org.springframework.stereotype.Component;

@Component
public class B {

}
```

There is also a Field injection, where Spring injects the required dependencies directly into the fields when those fields are annotated with @Autowired annotation.

Question 117: Difference between Constructor and Setter injection

Answer: The differences are:

- Partial dependency is not possible with Constructor based injection, but it is possible with Setter based injection. Suppose there are 4 properties in a class and the class has setter methods and a constructor with 4 parameters. In this case, if you want to inject only one/two property, then it is only possible with setter methods (unless you can define a new parametrized constructor with the needed properties)
- Cyclic dependency is also not possible with Constructor based injection. Suppose class A has dependency on class B and class B has dependency on class A and we are using constructor based injection, then when Spring tries to create object of class A, it sees that it needs class B object, then it tries to resolve that dependency first. But when it tries to create object of class B, it finds that it needs class A object, which is still under construction. Here Spring recognizes that a circular reference may have occurred and you will get an error in this case. This problem can easily be solved by using Setter based injection because dependencies are not injected at the object creation time
- While using Constructor injection, you will have to remember the order of parameters in a constructor when the number of constructor parameters increases. This is not the case with Setter injection
- Constructor injection helps in creating immutable objects, because a bean object is created using constructor and once the object is created, its dependencies cannot be altered anymore. Whereas with Setter injection, it's possible to inject dependency after object creation which leads to mutable objects.

Use constructor-based injection, when you want your class to not even be instantiated if the class dependencies are not resolved because Spring container will ensure that all the required dependencies are passed to the constructor.

Question 118: What is @Autowired annotation?

Answer: @Autowired is a way of telling Spring that auto-wiring is required. It can be applied to field, constructor and methods.

Question 119: What is the difference between BeanFactory and ApplicationContext?

Answer: The differences are:

- BeanFactory is the most basic version of IOC containers which should be preferred when memory consumption is critical whereas ApplicationContext extends BeanFactory, so you get all the benefits of BeanFactory plus some advanced features for enterprise applications
- BeanFactory instantiates beans on-demand i.e. when the method getBean(beanName) is called, it is also called Lazy initializer whereas

ApplicationContext instantiates beans at the time of creating the container where bean scope is Singleton, so it is an Eager initializer

- BeanFactory only supports 2 bean scopes, *singleton and prototype* whereas ApplicationContext supports all bean scopes
- ApplicationContext automatically registers BeanFactoryPostProcessor and BeanPostProcessor at startup, whereas BeanFactory does not register these interfaces automatically
- Annotation based dependency injection is not supported by BeanFactory whereas ApplicationContext supports it
- If you are using plain BeanFactory, features like transactions and AOP will not take effect (not without some extra steps), even if nothing is wrong with the configuration whereas in ApplicationContext, it will work
- ApplicationContext provides additional features like MessageSource access (i18n or Internationalization) and Event Publication

Use an *ApplicationContext* unless you have a really good reason for not doing so.

Question 120: Explain the life-cycle of a Spring Bean

Answer: Spring beans are java classes that are managed by Spring container and the bean life-cycle is also managed by Spring container.

The bean life-cycle has below steps:

- Bean instantiated by container
- Required dependencies of this bean are injected by container
- Custom Post initialization code to be executed (if required)
- Bean methods are used
- Custom Pre destruction code to be executed (if required)

When you want to execute some custom code that should be executed before the bean is in usable state, you can specify an *init() method* and if some custom code needs to be executed before the bean is destroyed, then a *destroy() method* can be specified.

There are various ways to define these init() and destroy() method for a bean:

By using xml file,

<bean> tag has 2 attributes that can be used to specify its init and destroy methods,

```
<bean id="testClass" class="com.example.demo.Test"
        init-method="init" destroy-method="destroy" />
```

You can give any name to your initialization and destroy methods, and here is our Test class

```
package com.example.demo;

public class Test {
    public void init() throws Exception{
        //do some initialization task
        System.out.println("init method");
    }

    public void destroy() throws Exception{
        //do some cleanup task
        System.out.println("destroy method");
    }
}
```

By implementing InitializingBean and DisposableBean interfaces

InitializingBean interface has afterPropertiesSet() method which can be used to execute some initialization task for a bean and DisposableBean interface has a destroy() method which can be used to execute some cleanup task.

Here is our Test class,

```
package com.example.demo;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class Test implements InitializingBean, DisposableBean {
    @Override
    public void afterPropertiesSet() throws Exception{
        //do some initialization task
        System.out.println("init method");
    }

    @Override
    public void destroy() throws Exception{
        //do some cleanup task
        System.out.println("destroy method");
    }
}
```

And, in the xml file:

```
<bean id="testClass" class="com.example.demo.Test" />
```

By using @PostConstruct and @PreDestroy annotations

```
package com.example.demo;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class Test {
    @PostConstruct
    public void init() throws Exception{
        //do some initialization task
        System.out.println("init method");
    }

    @PreDestroy
    public void destroy() throws Exception{
        //do some cleanup task
        System.out.println("destroy method");
    }
}
```

And, in the xml file:

```
<bean id="testClass" class="com.example.demo.Test" />
```

Question 121: What are the different scopes of a Bean?

Answer: Spring framework supports 5 scopes,

singleton – only one bean instance per Spring IOC container

prototype – it produces a new instance each and every time a bean is requested

request – a single instance will be created and made available during complete life-cycle of an HTTP request

session – a single instance will be created and made available during complete life-cycle of an HTTP session

global session – a single instance will be created during the life-cycle of a *ServletContext*

@Scope annotation or *scope attribute* of *bean tag* can be used to define bean scopes in Spring.

Question 122: What is the Default scope of a bean?

Answer: Default scope of a bean is ***Singleton*** that means only one instance per context.

Question 123: What happens when we inject a prototype scope bean in a singleton scope bean?

Answer: When you define a bean scope to be singleton, that means only one instance will be created and whenever we request for that bean, that same instance will be returned by the Spring container, however, a prototype scoped bean returns a new instance every time it is requested.

Spring framework gets only one chance to inject the dependencies, so if you try to inject a prototyped scoped bean inside a singleton scoped bean, Spring will instantiate the singleton bean and will inject one instance of prototyped scoped bean. This one instance of prototyped scoped bean is the only instance that is ever supplied to the singleton scoped bean.

So here, whenever the singleton bean is requested, you will get the same instance of prototyped scoped bean.

Question 124: How to inject a prototype scope bean in a singleton scope bean?

Answer: We have discussed in the previous question that when a prototyped scoped bean is injected in a singleton scoped bean, then on each request of singleton bean, we will get the same instance of prototype scoped bean, but there are certain ways where we can get a new instance of prototyped scoped bean also.

The solutions are:

- Injecting an ApplicationContext in Singleton bean and then getting the new instance of prototyped scoped bean from this ApplicationContext
- Lookup method injection using @Lookup
- Using scoped proxy

Injecting ApplicationContext:

To inject the ApplicationContext in Singleton bean, we can either use @Autowired annotation or we can implement ApplicationContextAware interface,

```java
import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.stereotype.Component;

@Component
public class SingletonBean implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext)
            throws BeansException {
        this.applicationContext = applicationContext;
    }

    public PrototypeBean getPrototypeBean() {
        return applicationContext.getBean(PrototypeBean.class);
    }
}
```

Here, whenever the getPrototypeBean() method is called, it will return a new instance of PrototypeBean.

But this approach contradicts with Spring IOC (Inversion of Control), as we are requesting the dependencies directly from the container.

Lookup Method Injection using @Lookup:

```java
import org.springframework.beans.factory.annotation.Lookup;
import org.springframework.stereotype.Component;

@Component
public class SingletonBean {

    @Lookup
    public PrototypeBean getPrototypeBean() {
        return null;
    }

}
```

Here, Spring will dynamically overrides getPrototypeBean() method annotated with @Lookup and it will look up the bean which is the return type of this method. Spring uses CGLIB library to do this.

Using Scoped Proxy

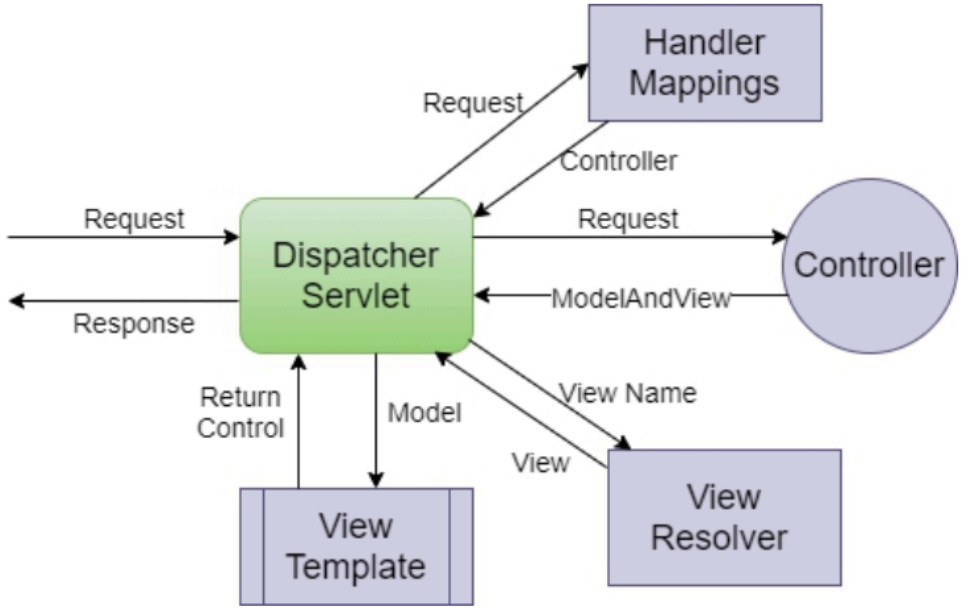
```java
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Scope;
import org.springframework.context.annotation.ScopedProxyMode;
import org.springframework.stereotype.Component;

@Component
public class SingletonBean {

    @Bean
    @Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE,
            proxyMode = ScopedProxyMode.TARGET_CLASS)
    public PrototypeBean getPrototypeBean() {
        return new PrototypeBean();
    }

}
```

Spring uses CGLIB to create the proxy object and the proxy object delegates method calls to the real object. In the above example, we are using *ScopedProxyMode.TARGET_CLASS* which causes an AOP proxy to be injected at the target injection point. The default Proxy mode is *ScopedProxyMode.NO*.

To avoid CGLIB usage, configure the proxy mode with *ScopedProxyMode.INTERFACES* and it will use JDK dynamic proxy.

Question 125: Explain Spring MVC flow

Answer:



In Spring, DispatcherServlet acts as the front Controller. When a request comes in Spring MVC application, below steps get executed,

- the request is first received by the DispatcherServlet
- DispatcherServlet will take the help of HandlerMapping and it will get to know the specific Controller that is associated with this request using @RequestMapping's
- Now, the request gets transferred to its associated Controller, the Controller will process this request by executing appropriate methods and returns the ModelAndView object back to the DispatcherServlet
- The DispatcherServlet will transfer this object to ViewResolver to get the actual view page

- Finally, DispatcherServlet passes the Model object to the View page which displays the result

Remember, in developing REST services, the Controller's request mapping methods are annotated with @ResponseBody annotations, so they don't return a logical view name to DispatcherServlet, instead it writes the output directly into the HTTP response body.

Question 126: What is the difference between <context:annotation-config> and <context:component-scan>?

Answer: The differences are:

<context:annotation-config> is used to activate annotations in beans that are already registered in the application context. So for example, if you have a class A that is already registered in the context and you have @Autowired, @Qualifier annotations in the class A, then <context:annotation-config> resolves these annotations.

<context:component-scan> can also do what <context:annotation-config> does, but *component-scan* also scans the packages for registering the beans to application context. If you are using, *component-scan*, then there is no need to use *annotation-config*.

Question 127: What is the difference between Spring and SpringBoot?

Answer: SpringBoot makes it easy to work with Spring framework. When using Spring framework, we have to take care of all the configuration ourselves like, for making a web application, DispatcherServlet, ViewResolver etc configurations are needed. SpringBoot solves this problem through a combination of Auto Configuration and Starter projects.

Question 128: What is auto-configuration in SpringBoot?

Answer: Spring applications have a lot of XML or Java Bean Configurations. Spring Boot Auto configuration looks at the jars available on the CLASSPATH and configuration provided by us in the application.properties file and it tries to auto-configure those classes.

For example, if Spring MVC jar is on the classpath, then DispatcherServlet will be automatically configured and if Hibernate jar is on the classpath, then a DataSource will be configured (Of course, we will have to provide datasource url, username and password).

Question 129: What are SpringBoot starters?

Answer: Think of SpringBoot starters as a set of related jars that we can use for our application, we don't have to go out and add the dependencies one by one and worry about which version will be compatible with the spring boot version that you are using, starters take care of all that.

For example, when you want to make a web application, you simply will add 'spring-boot-starter-web' as a dependency and you will have all the jars that are needed for a web application, like, DispatcherServlet, ViewResolver, Embedded Tomcat etc. Similarly, 'spring-boot-starter-data-jpa' dependency can be used when you want to work with Spring and JPA.

Question 130: What is @SpringBootApplication Annotation?

Answer: @SpringBootApplication is a combination of 3 different annotations:

@Configuration: This annotation marks a class as a Configuration class in Java-based configuration, it allows to register extra beans in the context or import additional configuration classes

@ComponentScan: to enable component scanning

@EnableAutoConfiguration: to enable Spring Boot's auto-configuration feature

These 3 annotations are frequently used together, so SpringBoot designers bundled them into one single @SpringBootApplication, now instead of 3 annotations you just need to specify only one

annotation on the Main class. However, if you don't need one of these annotation depending on your requirement, then you will have to use them separately.

Question 131: Where does a Spring Boot application start from?

Answer: The *SpringApplication* class provides a convenient way to bootstrap a Spring application that is started from a main() method. In many situations, you can delegate to the static SpringApplication.run method, as shown in the following example:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

}
```

Question 132: How to remove certain classes from getting auto-configured in SpringBoot?

Answer: @SpringBootApplication annotation accepts below parameters that can be used to remove certain classes from taking part in auto-configuration.

exclude: Exclude the list of classes from the auto-configuration

excludeNames: Exclude the list of class names from the auto-configuration

Question 133: How to autowire a class which is in a package other than SpringBoot application class's package or any of its sub-packages

Answer: When you specify @ComponentScan annotation on a class, then it starts scanning for the components from that package and its sub-packages, so if your class is in a different package altogether, then you will have to explicitly tell Spring to look into your package,

@ComponentScan annotation has below parameters:

scanBasePackages: Base packages to scan for annotated components

scanBasePackageClasses: Type-safe alternative to scanBasePackages() for specifying the packages to scan for annotated components. The package of each class specified will be scanned.

Question 134: What is application.properties file in a SpringBoot application?

Answer: SpringBoot allows us to configure our application configuration using application.properties file. In this file, we can define the configurations that will be needed to configure a datasource, like

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=adminUser
spring.datasource.password=adminPass
```

We can define the logging level of certain packages, using

```
logging.level.somePackagePath=INFO
logging.level.someOtherPackagePath=DEBUG
```

We can also define the port number that our embedded server will run on, using

```
server.port=9000
```

We can have different application.properties file for different environments like dev, stage and prod.