1) **What is Spring and what are it's advantages/why it is used?**

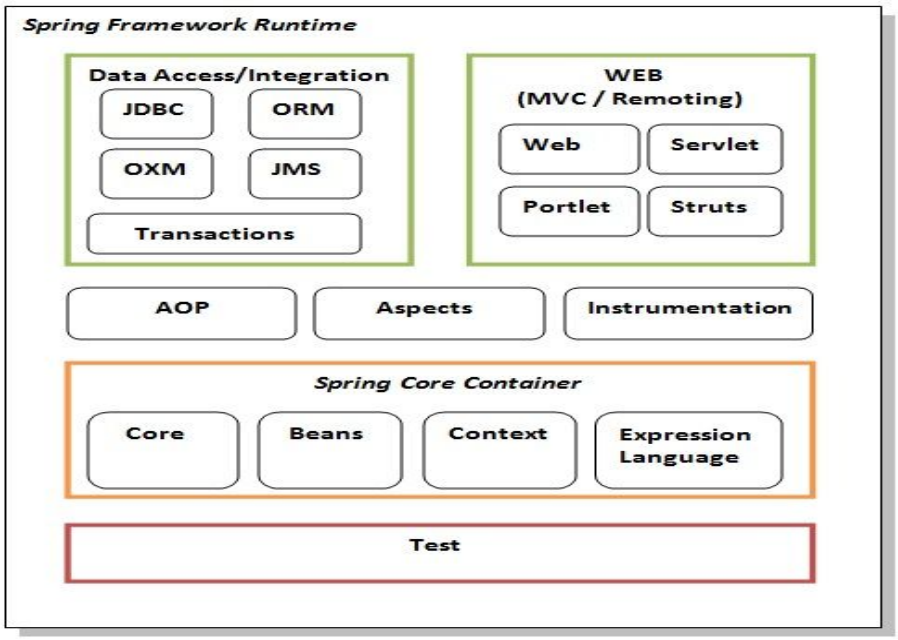
Spring is a framework which makes the development of JavaEE application easy. It was developed by Rod Johnson in 2003. Spring is a lightweight framework. It can be thought of as a framework of frameworks because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF etc.
The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc

Advantages

- **Predefined Templates**: Spring framework provides templates for JDBC, Hibernate, JPA etc. technologies. So there is no need to write too much code.
- **Loose Coupling**: The Spring applications are loosely coupled because of dependency injection.In such case, there is no need to modify the code if our logic is moved to new environment.
- **Easy to Test**
- **Lightweight:** Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface.
- **Fast Development**

2) What are different spring modules?

The Spring framework comprises of many modules such as core, beans, context, expression language, AOP, Aspects, Instrumentation, JDBC, ORM, OXM, JMS, Transaction, Web, Servlet, Struts etc. These modules are grouped into Test, Core Container, AOP, Aspects, Instrumentation, Data Access / Integration, Web (MVC / Remoting)



3. What is IoC?

As the name implies Inversion of control means now we have inverted the control of creating the object from our own using new operator to container or framework. Now it's the responsibility of container to create an object as required. We maintain one XML file where we configure our components, services, all the classes and their property. We just need to mention which service is needed by which component and container will create the object for us. This concept is known as dependency injection because all object dependency (resources) is injected into it by the framework.

Spring uses reflection to create instances from configuration file.

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:

- to instantiate the application class
- to configure the object
- to assemble the dependencies between the objects

4. Types of IoC containers and difference between them?

There are two types of IoC containers. They are:

1. BeanFactory
2. ApplicationContext

Difference between BeanFactory and the ApplicationContext

The org.springframework.beans.factory.BeanFactory and the org.springframework.context.ApplicationContextinterfaces acts as the IoC container. The ApplicationContext interface is built on top of the BeanFactory interface. It adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling, event propagation, application layer specific context (e.g. WebApplicationContext) for web application.

Using BeanFactory

```
Resource resource=new ClassPathResource("applicationContext.xml");
BeanFactory factory=new XmlBeanFactory(resource);
```

Using ApplicationContext

```
        ApplicationContext context =      new
ClassPathXmlApplicationContext("applicationContext.xml");
```

5. What is dependency Injection and it's types?

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application. Dependency Injection makes our programming code loosely coupled.

It removes the dependency of the programs. In such case we provide the information from the external source such as XML file. It makes our code loosely coupled and easier for testing.
In such case, instance of class is provided by external source such as XML file either by constructor or setter method.
Two ways to perform Dependency Injection in Spring framework
Spring framework provides two ways to inject dependency

- **By Constructor**
- **By Setter method**

6. Difference between setter and constructor Injection

No.	Constructor Injection	Setter Injection
1)	No Partial Injection*	Partial Injection*

2)	Doesn't override the setter property	Overrides the constructor property if both are defined.
3)	Creates new instance if any modification occurs	Doesn't create new instance if you change the property value
4)	Better for too many properties	Better for few properties.

Suppose we have three dependencies say String, int and long and we provide values only for two dependencies i.e. for String and int. So if the container automatically satisfies the third dependency then it is known as Partial Injection.

6. Give a simple example of DI with constructor-arg?

We can inject the dependency by constructor. The <constructor-arg>subelement of <bean> is used for constructor injection

Let's see the simple example to inject primitive and string-based values. We have created three files here:

1. Employee.java
2. applicationContext.xml
3. Test.java

Employee.Java

```
public class Employee {
private int id;
private String name;

public Employee() {System.out.println("def cons");}

public Employee(int id) {this.id = id;}

public Employee(String name) {  this.name = name;}

public Employee(int id, String name) {
   this.id = id;
   this.name = name;
}

void show(){
   System.out.println(id+" "+name);
```

```
}

}
```

applicationContext.xml

We are providing the information into the bean by this file. The constructor-arg element invokes the constructor. In such case, parameterized constructor of int type will be invoked. The value attribute of constructor-arg element will assign the specified value. The type attribute specifies that int parameter constructor will be invoked.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
   xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="e" class="Employee">
<constructor-arg value="10" type="int"></constructor-arg>
</bean>
 </beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the show method.

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.*;

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

      Resource r=new ClassPathResource("applicationContext.xml");
      BeanFactory factory=new XmlBeanFactory(r);

      Employee s=(Employee)factory.getBean("e");
      s.show();
   }
}
```

Output:10 null

7. Give one simple example of DI with Setter.

We can inject the dependency by setter method also. The <property>subelement of <bean> is used for setter injection
Let's see the simple example to inject primitive and string-based values by setter method. We have created three files here:

1. Employee.java
2. applicationContext.xml
3. Test.java

It is a simple class containing three fields id, name and city with its setters and getters and a method to display these informations.
Employee.java

```
public class Employee {
private int id;
private String name;
private String city;

public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}

public String getCity() {
    return city;
}
public void setCity(String city) {
    this.city = city;
}
void display(){
    System.out.println(id+" "+name+" "+city);
}

}
```

applicationContext.xml

We are providing the information into the bean by this file. The property element invokes the setter method. The value subelement of property will assign the specified value.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
   xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="obj" class="Employee">
<property name="id">
<value>20</value>
</property>
<property name="name">
<value>Arun</value>
</property>
<property name="city">
<value>ghaziabad</value>
</property>

</bean>

</beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the display method.

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.*;

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

      Resource r=new ClassPathResource("applicationContext.xml");
      BeanFactory factory=new XmlBeanFactory(r);

      Employee e=(Employee)factory.getBean("obj");
      s.display();
```

```
    }
}
```
Output:20 Arun ghaziabad

7. What is autowiring?

Autowiring feature of spring framework enables you to inject the object dependency implicitly(of child/aggregated objects). It internally uses setter or constructor injection.
Autowiring can't be used to inject primitive and string values. It works with reference only.

No.	Mode	Description
1)	no	this is the default mode, it means autowiring is not enabled.
2)	byName	injects the bean based on the property name. It uses setter method.
3)	byType	injects the bean based on the property type. It uses setter method.
4)	constructor	It injects the bean using constructor

B.java
This class contains a constructor and method only.
```
package org.sssit;
public class B {
B(){System.out.println("b is created");}
void print(){System.out.println("hello b");}
}
```
A.java
This class contains reference of B class and constructor and method.
```
package org.sssit;
public class A {
B b;
A(){System.out.println("a is created");}
public B getB() {
    return b;
}
```

```
public void setB(B b) {
    this.b = b;
}
void print(){System.out.println("hello a");}
void display(){
    print();
    b.print();
}
}
```

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="b" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="byName"></bean>

</beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the display method.

```
package org.sssit;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Test {
public static void main(String[] args) {
    ApplicationContext context=new ClassPathXmlApplicationContext("applicationContext.xml");
    A a=context.getBean("a",A.class);
    a.display();
}
}
```

Output:

```
b is created
a is created
hello a
hello b
```

---

1) byName autowiring mode

In case of byName autowiring mode, bean id and reference name must be same.
It internally uses setter injection.

```
<bean id="b" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="byName"></bean>
```

But, if you change the name of bean, it will not inject the dependency.
Let's see the code where we are changing the name of the bean from b to b1.

```
<bean id="b1" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="byName"></bean>
```

---

2) byType autowiring mode

In case of byType autowiring mode, bean id and reference name may be different. But there must be only one bean of a type.
It internally uses setter injection.

```
<bean id="b1" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="byType"></bean>
```

In this case, it works fine because you have created an instance of B type. It doesn't matter that you have different bean name than reference name.
But, if you have multiple bean of one type, it will not work and throw exception.
Let's see the code where are many bean of type B.

```
<bean id="b1" class="org.sssit.B"></bean>
<bean id="b2" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="byName"></bean>
```

In such case, it will throw exception.

---

3) constructor autowiring mode

In case of constructor autowiring mode, spring container injects the dependency by highest parameterized constructor.
If you have 3 constructors in a class, zero-arg, one-arg and two-arg then injection will be performed by calling the two-arg constructor.

```
<bean id="b" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="constructor"></bean>
```

8. What is spring bean?

The objects that form the backbone of Spring application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and managed by a Spring IoC container. These beans are created with the configuration metadata that we supply to the container, for example, in the form of XML definitions.

Central to the Spring Framework is its inversion of control container, which provides a way of configuring and managing Java objects. The container is responsible for managing object lifecycles of specific objects: creating these objects, calling their initialization methods, and configuring these objects by wiring them together.
Objects created by the container are **also called managed objects or beans**. The container can be configured by loading XML files or detecting specific Java annotations on configuration classes. These data sources contain the bean definitions which provide the information required to create the beans.

Objects can be obtained by means of either dependency lookup or dependency injection. Dependency lookup is a pattern where a caller asks the container object for an object with a specific name or of a specific type. ***Dependency injection is a pattern where the container passes objects by name to other objects, via either constructors, properties, or factory methods.***

9. What is the difference between singleton and prototype bean?

A bean has scopes which define their existence on the application

Singleton: means single bean definition to a single object instance per Spring IOC container.The bean instance will be only once and same instance will be returned by the IOC container.

```
<bean id = "helloWorld" class = "HelloWorld" scope = "singleton">
   </bean>
```

Prototype: means a single bean definition to any number of object instances. There could be multiple object instances.The bean instance will be created each time when requested.

```
<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld" scope = "prototype">
   </bean>
```

Whatever beans we defined in spring framework are singleton beans. There is an attribute in bean tag named ‘singleton’ if specified true then bean becomes singleton and if set to false then the bean becomes a prototype bean. By default, it is set to true. So, all the beans in spring framework are by default singleton beans.

```
<bean id="employee"     class="Employee" singleton=”false”>
     <property name="newBid"/>
</bean>
```

10. What are different types of Spring bean/scopes of it?

1. Singleton(default*)
Scopes a single bean definition to a single object instance per Spring IoC container.The bean instance will be only once and same instance will be returned by the IOC container. It is the default scope.
2. Prototype
Scopes a single bean definition to any number of object instances.The bean instance will be created each time when requested.
3. Request
It scopes a single bean definition to the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
The bean instance will be created per HTTP request.

4. Session
Scopes a single bean definition to the lifecycle of a HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.The bean instance will be created per HTTP session.

5. Global session
Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet* context. Only valid in the context of a web-aware Spring ApplicationContext.The bean instance will be created per HTTP global session. It can be used in portlet context only.
***Portlets** are pluggable user interface software components that are managed and displayed in a web portal.*

10. Define Spring bean life cycle?

Spring framework is based on IOC so we call it as IOC container also, So Spring beans reside inside the IOC container. Spring beans are nothing but Plain old java object (POJO).

Stages of Life Cycle of bean

1. The container will look the bean definition inside configuration file (e.g. bean.xml).
2 using reflection container will create the object and if any property is defined inside the bean definition then it will also be set.
3. If the bean implements the BeanNameAware interface, the factory calls setBeanName() passing the bean's ID.
4. If the bean implements the BeanFactoryAware interface, the factory calls setBeanFactory(), passing an instance of itself.

5. If there are any BeanPostProcessors associated with the bean, their post-ProcessBeforeInitialization() methods will be called before the properties for the Bean are set.
6. If an init() method is specified for the bean, it will be called.
7. If the Bean class implements the DisposableBean interface, then the method destroy() will be called when the Application no longer needs the bean reference.
8. If the Bean definition in the Configuration file contains a 'destroy-method' attribute, then the corresponding method definition in the Bean class will be called.

12. What is spring AOP and why it is used/ what problem it solves?

Aspect Oriented Programming (AOP) compliments OOPs in the sense that it also provides modularity. But the key unit of modularity is aspect than class.
AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by **cross-cutting concerns.**
A cross-cutting concern is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.
It provides the pluggable way to dynamically add the additional concern before, after or around the actual logic.

Problem without AOP We can call methods (that maintains log and sends notification) from the methods starting with m. In such scenario, we need to write the code in all the 5 methods.
But, if client says in future, I don't have to send notification, you need to change all the methods.
It leads to the maintenance problem.

Solution with AOP We don't have to call methods from the method. Now we can define the additional concern like maintaining log, sending notification etc. in the method of a class.
In future, if client says to remove the notifier functionality, we need to change code only once,
So, maintenance is easy in AOP.

13. Defined all following concepts in AOP

- **Join point** Join point is any point in your program such as method execution, exception handling, field access etc. Spring supports only method execution join point. We pass an object of Joinpoint to advice-method to get information of the method.

- **Pointcut** It is the expression which tells the type of joinpoint about which we have concern; a predicate that matches join points.It is something that defines at what join-points an advice should be applied.

- **Aspect** It is a class that contains advices, joinpoints etc., It's basically the concern which we want to implement; example we want to do logging; so we create a commonLogging class.

- **Advice** It is the method code which includes the actual part which we want to run; example it will contain sys out statemtns to print currently executing method name.Advice represents an action taken by an aspect at a particular join point.
  Eg. Before ; before join point
  After ; after join point
- **Weaving** It is the process of linking aspect with other application types or objects to create an advised object. Weaving can be done at compile time, load time or runtime. Spring AOP performs weaving at runtime.

#Apache AspectJ is the implementation API for AOP.

Example of AOP with Before

```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class TrackOperation{

   @Before("execution(* Operation.*(..))")//applying pointcut on before advice
   public void myadvice(JoinPoint jp)//it is advice (before advice)
   {
      System.out.println("additional concern");
      //System.out.println("Method Signature: "  + jp.getSignature());
   }
}
```

ApplicationContext.xml

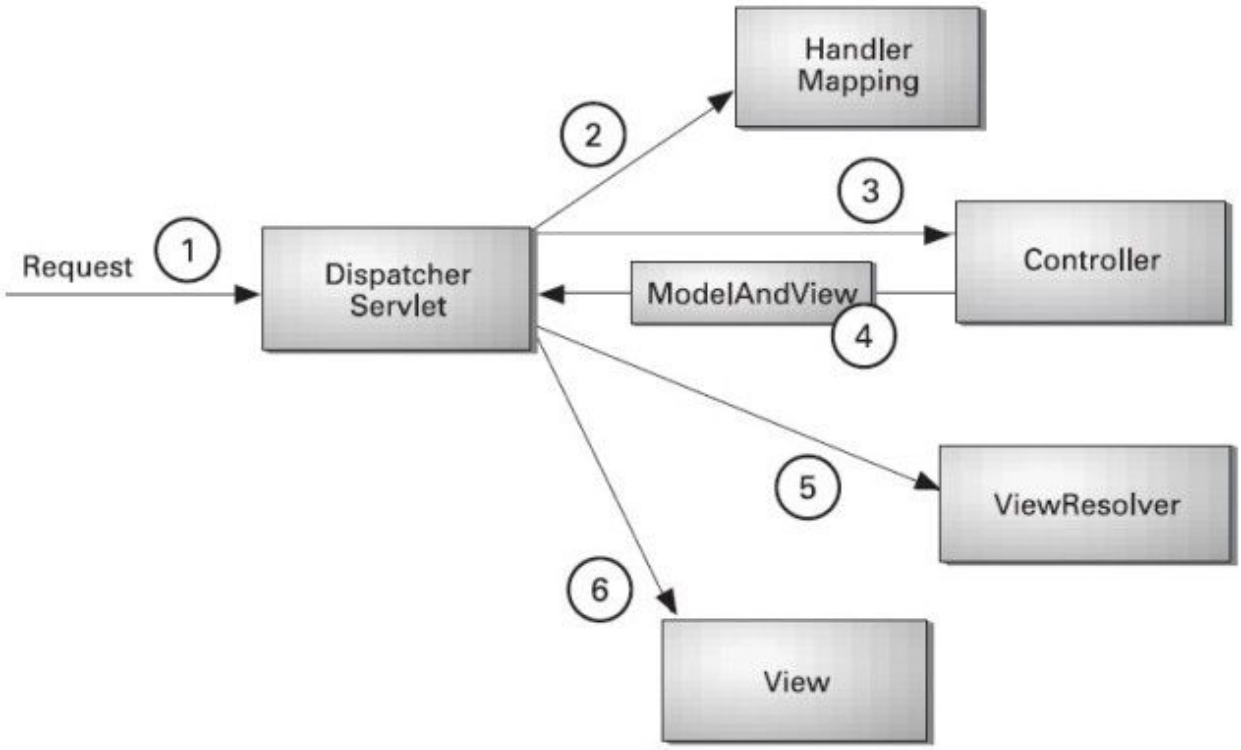
```
 <bean id="opBean" class="Operation">   </bean>
   <bean id="trackMyBean" class="TrackOperation"></bean>

   <bean
class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator"
></bean>
</beans>
```

14. What is spring MVC

The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

- The Model encapsulates the application data and in general they will consist of POJO.
- The View is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
- The Controller is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.

In Spring Web MVC, DispatcherServlet class works as the front controller. It is responsible to manage the flow of the spring mvc application.
The @Controller annotation is used to mark the class as the controller (since Spring 3).
The @RequestMapping annotation is used to map the request url. It is applied on the method.



15. What is the role of dispatcherServlet in MVC

The DispatcherServlet is very important from Spring MVC perspective, it acts as a FrontController i.e. all requests pass through it. It is responsible for routing the request to

controller and view resolution before sending the response to the client. When Controller returns a Model or View object, it consults all the view resolvers registered to find the correct type of ViewResolver which can render the response for clients.

16. What is the difference between @Controller and @RestController in Spring MVC?

Even though both are used to indicate that a Spring bean is a Controller in Spring MVC setup, @RestController is better when you are developing RESTful web services using Spring MVC framework. It's a combination of @Controller + @ResponseBody annotation which allows the controller to directly write the response and bypassing the view resolution process, which is not required for RESTful web service.

17. Type of transaction management spring supports

Two types of transaction management is supported by spring:
1. Programmatic transaction management
2. Declarative transaction management.