- **What is Spring framework?**
  Answer: Spring framework is a dependency injection system which is created with the primary goal of preventing objects to manage their dependencies' lifecycle and rather depend on the container (Inversion of Control) inject the dependencies during runtime. It helps build decoupled systems.
  Spring framework allows to wire up the classes with their dependencies using configuration defined in XML file or using annotations approach.
  The following is the code without using Spring Framework:

```
public class Laptop {
  private Processor processor;
  private RAM ram;
  private OperatingSystem operatingSystem;

  public Laptop(String processorType) {
      this.processor = ProcessorFactory.getProcessor(processorType);
      this.ram = new RAM();
      this.operatingSystem = new OperatingSystem();
  }
}
```

- With Spring Framework, the above code would look like this:

```
@Component
public class Laptop {
  private Processor processor;
  private RAM ram;
  private OperatingSystem operatingSystem;

  public Laptop(@Autowired Processor processor, @Autowired RAM ram, @Autowired
OperatingSystem operatingSystem) {
      this.processor = processor;
      this.ram = ram;
      this.operatingSystem = operatingSystem;
  }
}
```

- **What is dependency injection? What are benefits of dependency injection containers?**
  Answer: Dependency injection is about injecting objects dependency during runtime thereby preventing the object to manage the dependent objects' lifecycle. The dependency can be injected either by another object or dependency injector system. This dependency injection system is what forms the core of Spring framework.
  The following is code where object manages the dependent objects' lifecycle.

```
public class Car {

  private Engine engine;

  public Car() {
      this.engine = new Engine();
  }
}
```

- The following is the code where object dependencies are injected:

```
@Component(name="car")
public class Car {

  private Engine engine;

  public Car(@Autowired Engine engine) {
      this.engine = engine;
  }
}
```

- The above style of dependency injection is termed as injection via the constructor. Above code enables the Engine object to be injected (Autowired) during Runtime while creating a bean named car of a class type as Car. Dependencies can also be injected via setter methods. This post can be read for greater details.
- **What are different forms of dependency injection?**
  The following are different forms of dependency injection widely used in Spring:
    - Constructor injection: Constructor injection uses a constructor to inject appropriate implementations into the class. The following is a sample code:

      ```
      public class Car {
        private Engine engine;
        public Car(@Autowired Engine engine) {
            this.engine = engine;
        }
      }
      ```

    - Setter injection: Setter injection advocates injecting dependencies using setter methods. The following is the sample code:

      ```
      public class Car {
        @Autowired private Engine engine;
        public void setEngine(Engine engine) {
            this.engine = engine;
        }
      }
      ```

- **What is @Autowired annotation? Is there any difference between @Inject and @Autowired in Spring framework?**
  Answer: Both of them under the hood does the same thing. @Inject is part of the Java CDI (Contexts and Dependency Injection) standard introduced in Java EE 6 (JSR-299). The CDI defines a standard for dependency injection. @Autowired is Spring's own (legacy) annotation. Here is a good read on the difference between @Inject and @Autowired;
- **Explain the concept of ApplicationContext and WebApplicationContext?**
  Answer: ApplicationContext represents the root context configuration of the web application. In other words, it represents the application context of the whole application. The WebApplicationContext is an extension of the plain ApplicationContext that has some extra features necessary for web

applications. In a Spring MVC web application, there is one root WebApplicationContext which contains all the infrastructure beans which are shared between other contexts and Servlet instances. Then, each DispatcherServlet consists of its own WebApplicationContext which inherits all the beans from root WebApplicationContext. The greater details on WebApplicationContext can be found on the page, Spring – The Web. Some more information can be found on this page that provides a good read on the difference between applicationContext and WebApplicationContext.

- **What is the difference between BeanFactory and ApplicationContext? When to use which one of them?**
  BeanFactory is the root interface for accessing a Spring bean container. The interface is implemented by objects that hold a number of bean definitions, each uniquely identified by a String name. A BeanFactory will load bean definitions stored in a configuration source (such as an XML document). The details can be found on this page, BeanFactory. ApplicationContext is an interface to provide configuration for an application. Of many interfaces that it extends, one of them is BeanFactory. ApplicationContext provides methods for accessing application components, load file resources, publish events to registered listeners, resolve messages, supporting externalization. Further details could be found on this page, ApplicationContext.
- **Why is it suggested to** autowire **the interface and not the implemented class?**
  Answer: Primary reason being the fact that different implementations could be injected at the runtime to meet/fulfill a different kind of requirements. It is also a good practice to follow – **Design by interface and not by implementation**. Provides a good read on why to autowire the interface and not to the implementation.
- **How can Spring be integrated with Java-based web framework?**
  Answer: The following is one of the ways:
  - Define following in web.xml file. Pay attention to the fact the ContextLoaderListener is declared and a path representing contextConfigLocation is defined.

```
<context-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>/WEB-INF/applicationContext*.xml</param-value>
</context-param>
<listener>
 <listener-class>
 org.springframework.web.context.ContextLoaderListener
 </listener-class>
</listener>
```

  - Later in the application, retrieve Bean objects using following code. In the code below, an instance of WebApplicationContext is obtained from which the bean instance is retrieved. Get the details on WebApplicationContextUtils from Spring Doc page on WebApplicationContextUtils.

```
 WebApplicationContext webAppContext =
WebApplicationContextUtils.getRequiredWebApplicationContext(servlet.getSe
SomeBean someBean = (SomeBean) webAppContext.getBean("someBean");
```

  - Different techniques are listed on this page.

- **What are differences between Spring Boot and Spring MVC?**
  Answer: Spring MVC is an HTTP oriented MVC framework managed by the Spring Framework and based in Servlets. Spring boot is a Spring utility for quickly setting up applications while offering an out of the box configuration in order to build Spring-powered applications. A very nice article related to the difference between Spring Boot and Spring MVC can be found here.
- **What's the difference between @Component, @Repository & @Service annotations in Spring?**
  Answer: The @Component annotation marks a java class as a bean so the Spring framework picks it up for component-scanning mechanism and pulls such classes into the application context.
  @Service annotation is a specialization of @Component annotation and is recommended to be used to annotate a class as a service if it is meant to be a service. It enhances the readability and thus, usability of the class.
  @Repository annotation is again a specialization of @Component annotation and is used to represent DAOs (data access objects). As per domain-driven design, a class annotated with @Repository is expected to provide mechanisms for encapsulating storage, retrieval, and search behavior which emulates a collection of objects.