Answer the following:

1. **Explain the concept of dependency injection and its types.**

Ans –

Dependency Injection (DI):

- **Definition**: Dependency Injection is a design pattern used in Spring to achieve **Inversion of Control (IoC)**. It allows the Spring Framework to manage and provide the required dependencies to an object instead of the object creating them itself.
- **Purpose**: Promotes loose coupling, modularity, and testability.

How It Works:

- Dependencies are **injected** into an object by the framework during runtime.
- Spring provides the required objects based on configuration, either via XML, annotations, or Java-based configuration.

Types of Dependency Injection:

1. **Constructor Injection**:
   - Dependencies are injected through the class constructor.
   - Recommended for mandatory dependencies.

Example:

```
import org.springframework.stereotype.Component;


@Component
public class UserService {
   private final UserRepository userRepository;


   // Constructor Injection
   public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

2. Setter Injection:

- Dependencies are injected via setter methods.
- Useful for optional dependencies.

Example:

```
import org.springframework.stereotype.Component;

@Component
public class UserService {
    private UserRepository userRepository;

    // Setter Injection
    public void setUserRepository(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

3. Field Injection:

- Dependencies are injected directly into fields using @Autowired.
- The simplest method but not recommended for required dependencies, as it bypasses immutability.

Example :

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class UserService {
    @Autowired
    private UserRepository userRepository;
}
```

2. What is the concept of IoC? Explain with examples.

Ans –

Definition:

- IoC is a design principle where the control of object creation and their lifecycle is transferred to the Spring Framework instead of being handled manually by the developer.

How IoC Works:

1. The developer specifies the object's configuration and dependencies.
2. The Spring IoC container manages the lifecycle and dependency resolution.

Spring IoC Container:

- **BeanFactory**: Basic IoC container that supports lazy loading.
- **ApplicationContext**: Advanced IoC container with additional features like event propagation, AOP, etc.

Example of IoC:

1. **XML Configuration**:

```
<!-- applicationContext.xml -->
<beans xmlns="http://www.springframework.org/schema/beans">
   <bean id="userRepository" class="com.example.UserRepository" />
   <bean id="userService" class="com.example.UserService">
      <constructor-arg ref="userRepository" />
   </bean>
</beans>
```

Example 1: Without IoC (Tightly Coupled Code)

```
public class Car {
    private Engine engine;
        public Car() {
            this.engine = new Engine(); // Car class creates an instance of
            Engine directly
  }
}
```

In this code, the Car class is tightly coupled with the Engine class, making it difficult to change the Engine implementation without modifying the Car class.

Example 2: With IoC (Using Dependency Injection)

```
public class Car {
private Engine engine;
 // Engine is injected into the Car class through the constructor
            public Car(Engine engine) {
                    this.engine = engine;
        }
 }
```

In this version, the Car class does not create the Engine object; instead, it receives the Engine instance from an external source (e.g., a framework like Spring).

Example with Spring Framework:

```
 @Component
public class Engine {
    public void start() {
         System.out.println("Engine started.");
   }
}
 @Component
 public class Car {
     private Engine engine;
     @Autowired // Dependency injection by Spring
     public Car(Engine engine) {
         this.engine = engine;
    }
      public void drive() {
          engine.start();
         System.out.println("Car is driving.");
     }
}
```

In the Spring Framework, the @Autowired annotation is used for dependency injection. The IoC container (Spring) is responsible for creating and injecting the Engine instance into the Car class. This reduces the coupling between Car and Engine and improves flexibility, as you can easily swap Engine with a different implementation without modifying Car.

3. **Explain the use of the @Bean annotation with examples.**

Ans –

Purpose:

- Used in Spring to define a method that creates and returns a bean to be managed by the Spring IoC container.
- It replaces XML <bean> definitions.

Key Features:

1. **Scope**:
    - By default, beans created with @Bean are **singleton**.
    - You can specify other scopes, such as prototype, using @Scope.
2. **Dependency Injection**:
    - Method parameters of @Bean annotated methods are automatically resolved by Spring.

Example:

1. **Basic Example**:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {

    @Bean
    public UserRepository userRepository() {
        return new UserRepository();
    }

    @Bean
    public UserService userService() {

        return new UserService(userRepository());

    }

}
```

2. **With Dependency Injection**:

```
@Configuration
public class AppConfig {

    @Bean
    public UserRepository userRepository() {
        return new UserRepository();
    }

    @Bean
    public UserService userService(UserRepository userRepository) {
        return new UserService(userRepository);
    }
}
```

3. **Prototype Scope**:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Scope;

@Configuration
public class AppConfig {

    @Bean
    @Scope("prototype")
    public UserService userService() {
        return new UserService();
    }
}
```