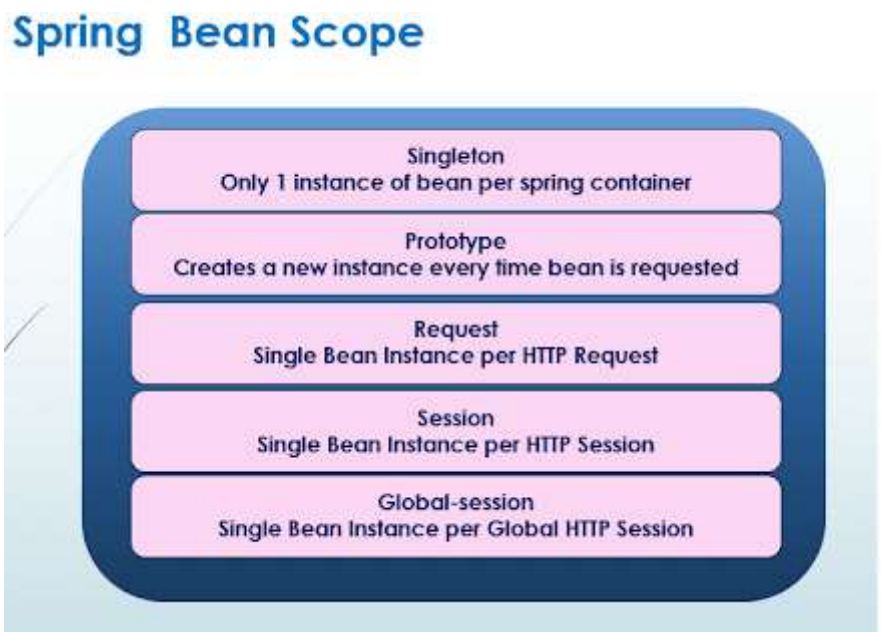
scopes and lifecycle phases

In Spring Framework, beans have different scopes and lifecycle phases that control their creation, management, and destruction. Understanding these concepts is essential for effectively managing beans in a Spring application. Here's a comprehensive overview:

1. **Bean Scopes**



Bean scope determines the lifecycle and visibility of a bean within the application context. The most common bean scopes in Spring are:

a. Singleton (Default Scope)

- **Definition**: A single instance of the bean is created and shared across the entire Spring container. All requests for this bean will return the same instance.
- **Usage**: Default scope if none is specified.
- **Configuration**: @Scope("singleton") (though this is the default, so it is optional).

```
@Component
@Scope("singleton")
```

```
public class SingletonBean {
    // Bean definition
}
```

b. Prototype

- **Definition**: A new instance of the bean is created every time it is requested from the container.
- **Usage**: Useful for stateful beans where each instance needs to maintain its own state.
- **Configuration**: @Scope("prototype").

```
@Component
@Scope("prototype")
public class PrototypeBean {
    // Bean definition
}
```

c. Request

- **Definition**: A new instance of the bean is created for each HTTP request. This scope is available only in a web-aware Spring ApplicationContext.
- **Usage**: Useful for web applications where each request needs a new instance of the bean.
- **Configuration**: @Scope("request").

```
@Component
@Scope("request")
public class RequestScopedBean {
    // Bean definition
}
```

d. Session

- **Definition**: A new instance of the bean is created for each HTTP session. This scope is available only in a web-aware Spring ApplicationContext.
- **Usage**: Useful for web applications where each session needs a new instance of the bean.
- **Configuration**: @Scope("session").

```
@Component
@Scope("session")
public class SessionScopedBean {
    // Bean definition
}
```

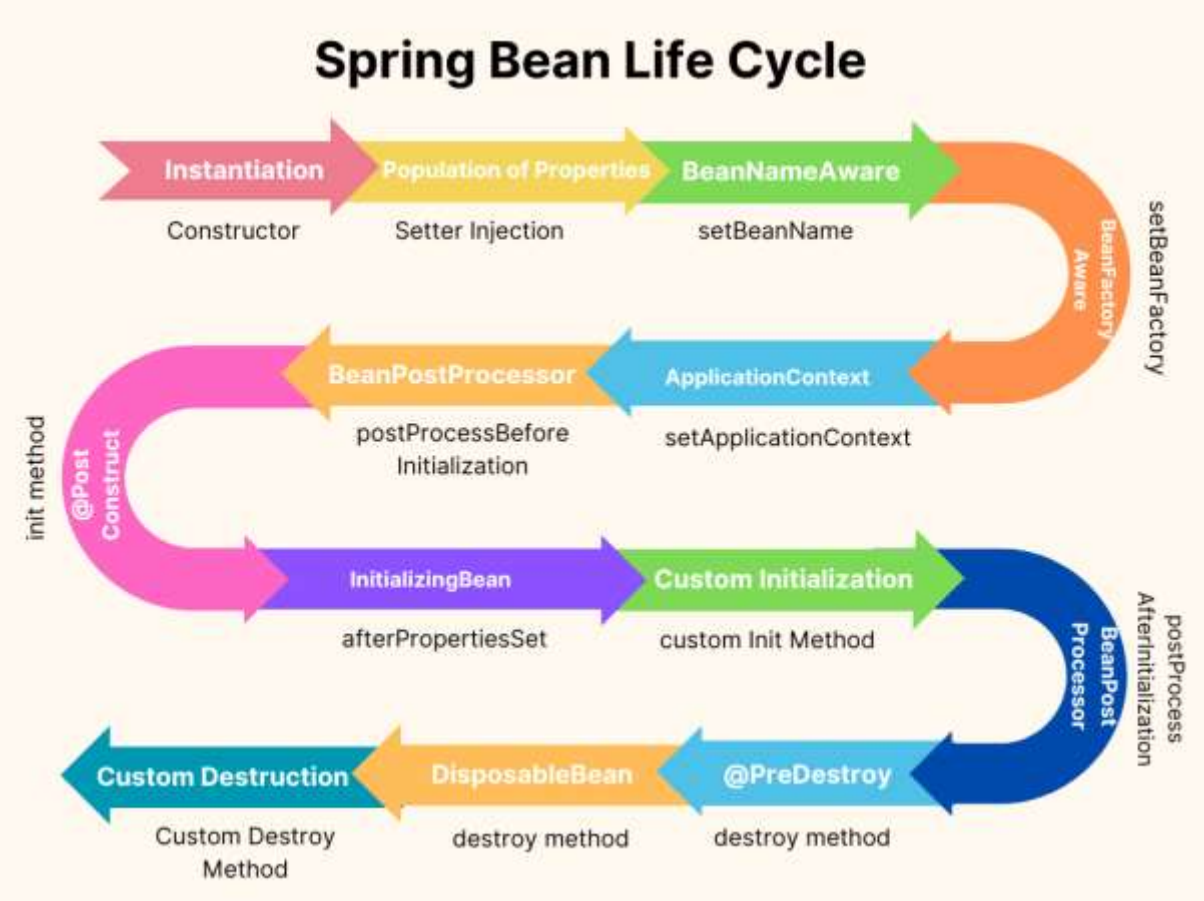
e. Application

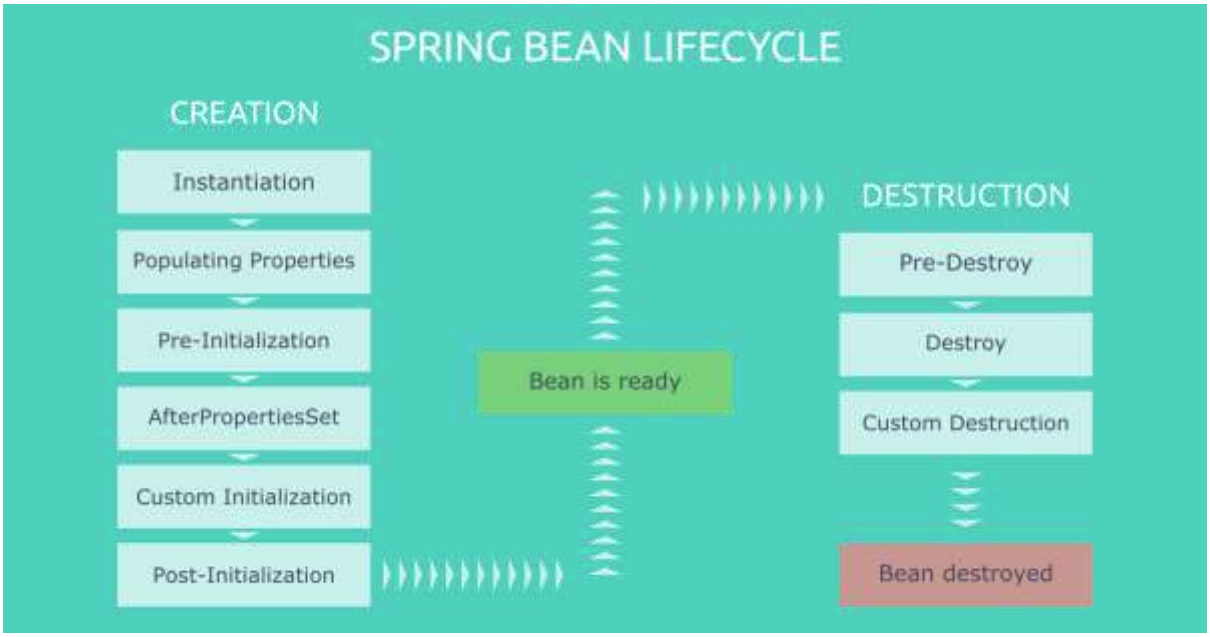
- **Definition**: A single instance of the bean is created for the entire servlet context. This scope is available only in a web-aware Spring ApplicationContext.
- **Usage**: Useful for beans that need to be shared across the entire web application.
- **Configuration**: @Scope("application").

```
@Component
@Scope("application")
public class ApplicationScopedBean {
    // Bean definition
}
```

2. Bean Lifecycle

The bean lifecycle consists of several phases from creation to destruction. Key lifecycle methods and events include:





a. Instantiation

- **Definition**: Spring creates a new instance of the bean.

- **Method**: Constructor injection or factory method.

b. Populating Properties

- **Definition**: Spring injects dependencies into the bean.
- **Method**: Setter injection or field injection.

c. Post-Process Initialization

- **Definition**: Beans may be processed by BeanPostProcessors before and after initialization methods.
- **Methods**: postProcessBeforeInitialization and postProcessAfterInitialization.

```
@Component
public class CustomBeanPostProcessor implements
BeanPostProcessor {
    @Override
    public Object postProcessBeforeInitialization(Object bean, String
beanName) throws BeansException {
        // Code to execute before bean initialization
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String
beanName) throws BeansException {
        // Code to execute after bean initialization
        return bean;
    }
}
```

d. Initializing Bean

- **Definition**: Spring calls the afterPropertiesSet method if the bean implements InitializingBean or a custom init-method if specified.
- **Methods**: afterPropertiesSet() method or init-method attribute.

```
@Component
public class InitializingBeanExample implements InitializingBean {
    @Override
    public void afterPropertiesSet() throws Exception {
        // Custom initialization logic
    }
}
```

```
@Component
public class CustomInitMethodBean {
    @PostConstruct
    public void init() {
        // Custom initialization logic
    }
}
```

e. Bean Usage

- **Definition**: The bean is used by the application.

f. Destruction

- **Definition**: Spring calls the destroy method if the bean implements DisposableBean or a custom destroy-method if specified.
- **Methods**: destroy() method or destroy-method attribute.

```
@Component
public class DisposableBeanExample implements DisposableBean {
    @Override
    public void destroy() throws Exception {
        // Custom destruction logic
    }
```

```
}


@Component
public class CustomDestroyMethodBean {
    @PreDestroy
    public void cleanup() {
        // Custom cleanup logic
    }
}
```

Summary

- **Scopes**: Singleton, Prototype, Request, Session, and Application.
- **Lifecycle**: Instantiation, Property Injection, Post-Processing, Initialization, Usage, and Destruction.

Understanding and managing bean scopes and lifecycle helps in designing efficient and properly scoped beans within your Spring application, ensuring the correct behavior and performance of your application.

In the Spring Framework, a **bean** is an object that is managed by the Spring IoC (Inversion of Control) container. Beans are the backbone of a Spring application, and they are created, configured, and assembled by the container based on the configuration provided.

Key Characteristics of a Spring Bean

1. **Managed by Spring Container**: Beans are instantiated, configured, and managed by the Spring IoC container. The container is responsible for their lifecycle, including creation, dependency injection, and destruction.
2. **Configuration**: Beans are defined in configuration files or classes. In XML-based configuration, beans are defined using <bean> elements. In -based configuration, beans are defined using @Bean methods within @Configuration classes.

3. **Lifecycle**: Beans go through various lifecycle phases such as instantiation, property setting, initialization, and destruction. The Spring container manages these phases.
4. **Dependency Injection**: Beans can have dependencies that are injected by the Spring container. This is achieved through constructor injection, setter injection, or field injection.
5. **Scope**: Beans can have different scopes that define their lifecycle and visibility within the application context, such as singleton, prototype, request, session, and application.

How Beans Work in Spring

1. **Definition**: Beans are defined in the Spring configuration. For XML-based configuration, you define beans using <bean> tags. For -based configuration, you define beans using @Bean annotations inside @Configuration classes.

```
<!-- XML Configuration -->
<bean id="myBean" class="com.example.MyBean" />
```

```
//  Configuration
@Configuration
public class AppConfig {
    @Bean
    public MyBean myBean() {
        return new MyBean();
    }
}
```

2. **Instantiation**: The Spring container creates instances of the beans based on the configuration.
3. **Configuration**: The container sets properties and dependencies on the bean as defined in the configuration.

4. **Injection**: Dependencies are injected into the bean. This can be done through constructor parameters, setter methods, or field injection.

```
@Component
public class MyBean {
    private Dependency dependency;

    @Autowired
    public MyBean(Dependency dependency) {
        this.dependency = dependency;
    }

    // Getter and setter for dependency
}
```

5. **Lifecycle Management**: The container manages the lifecycle of beans, including initialization and destruction. You can specify custom initialization and destruction methods if needed.

```
@Component
public class MyBean {
    @PostConstruct
    public void init() {
        // Initialization logic
    }

    @PreDestroy
    public void cleanup() {
        // Cleanup logic
    }
}
```

6. **Accessing Beans**: Beans are accessed from the application context. You can retrieve beans by their type or by their name.

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
MyBean myBean = context.getBean(MyBean.class);
```

Summary

A Spring bean is essentially a object that the Spring container manages. It is created, configured, and assembled by the container, which handles its entire lifecycle. Beans can have dependencies that are injected by the container, and they can be configured with various scopes and lifecycle callbacks. Understanding beans is fundamental to working effectively with the Spring Framework.

In Spring, **bean lifecycle callbacks** allow you to execute custom code at various points in the lifecycle of a bean. These callbacks are used for tasks such as initialization, cleanup, and other custom actions that need to occur when the bean is created or destroyed. Here's a detailed look at the different types of lifecycle callbacks you can use in Spring:

Types of Bean Lifecycle Callbacks

1. **Initialization Callbacks**: These methods are called after the bean's properties have been set but before the bean is used. They are used for initialization tasks.
2. **Destruction Callbacks**: These methods are called before the bean is destroyed. They are used for cleanup tasks.

Methods for Bean Lifecycle Callbacks

1. **InitializingBean Interface**: Implementing the InitializingBean interface allows you to define the afterPropertiesSet() method, which will be called by the Spring container after the bean's properties have been set.

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

public class MyBean implements InitializingBean {
    @Override
    public void afterPropertiesSet() throws Exception {
        // Initialization logic here
    }
}
```

2. **DisposableBean Interface**: Implementing the DisposableBean interface allows you to define the destroy() method, which will be called by the Spring container before the bean is destroyed.

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

public class MyBean implements DisposableBean {
    @Override
    public void destroy() throws Exception {
        // Cleanup logic here
    }
}
```

3. **Custom Initialization and Destruction Methods**: You can specify custom initialization and destruction methods in the bean definition using the init-method and destroy-method attributes in XML configuration or with annotations in configuration.

   XML Configuration Example:

   xml

```
<bean id="myBean" class="com.example.MyBean" init-method="init" destroy-method="cleanup"/>
```

Configuration Example:

```
@Configuration
public class AppConfig {
    @Bean(initMethod = "init", destroyMethod = "cleanup")
    public MyBean myBean() {
        return new MyBean();
    }
}
```

Bean Class Example:

```
public class MyBean {
    public void init() {
        // Custom initialization logic
    }

    public void cleanup() {
        // Custom cleanup logic
    }
}
```

4. **@PostConstruct and @PreDestroy Annotations**: These annotations provide a convenient way to specify initialization and destruction methods without having to implement specific interfaces or configure XML.

```
import x.annotation.PostConstruct;
import x.annotation.PreDestroy;

@Component
public class MyBean {
```

```
    @PostConstruct
    public void init() {
        // Initialization logic here
    }

    @PreDestroy
    public void cleanup() {
        // Cleanup logic here
    }
}
```

Summary

- **Initialization Callbacks**: Methods that run after the bean's properties have been set, such as afterPropertiesSet(), custom init methods, or @PostConstruct.
- **Destruction Callbacks**: Methods that run before the bean is destroyed, such as destroy(), custom cleanup methods, or @PreDestroy.

By using these lifecycle callbacks, you can ensure that your beans are properly initialized and cleaned up, helping to manage resources and perform necessary setup and teardown tasks.