



Inside the Spring Container

Understanding the Spring Bean Lifecycle

Objectives

After completing this lesson, you should be able to

- Explain the Spring Bean Lifecycle
- Use a BeanFactoryPostProcessor and a Bean PostProcessor
- Explain how Spring Proxies add Behavior at Runtime
- Describe how Spring determines Bean Creation Order
- Avoid issues when Injecting Beans by Type

Agenda

■ The Spring Bean Lifecycle

- Phase 1: Initialization
- Phase 2: Usage
- Phase 3: Destruction

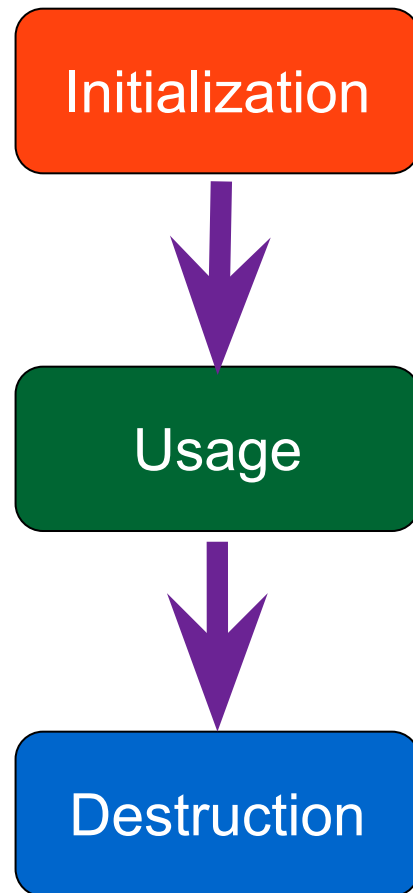
■ More on Bean Creation

The content of this chapter is a *much simplified* view of Spring's inner workings

Container Lifecycle

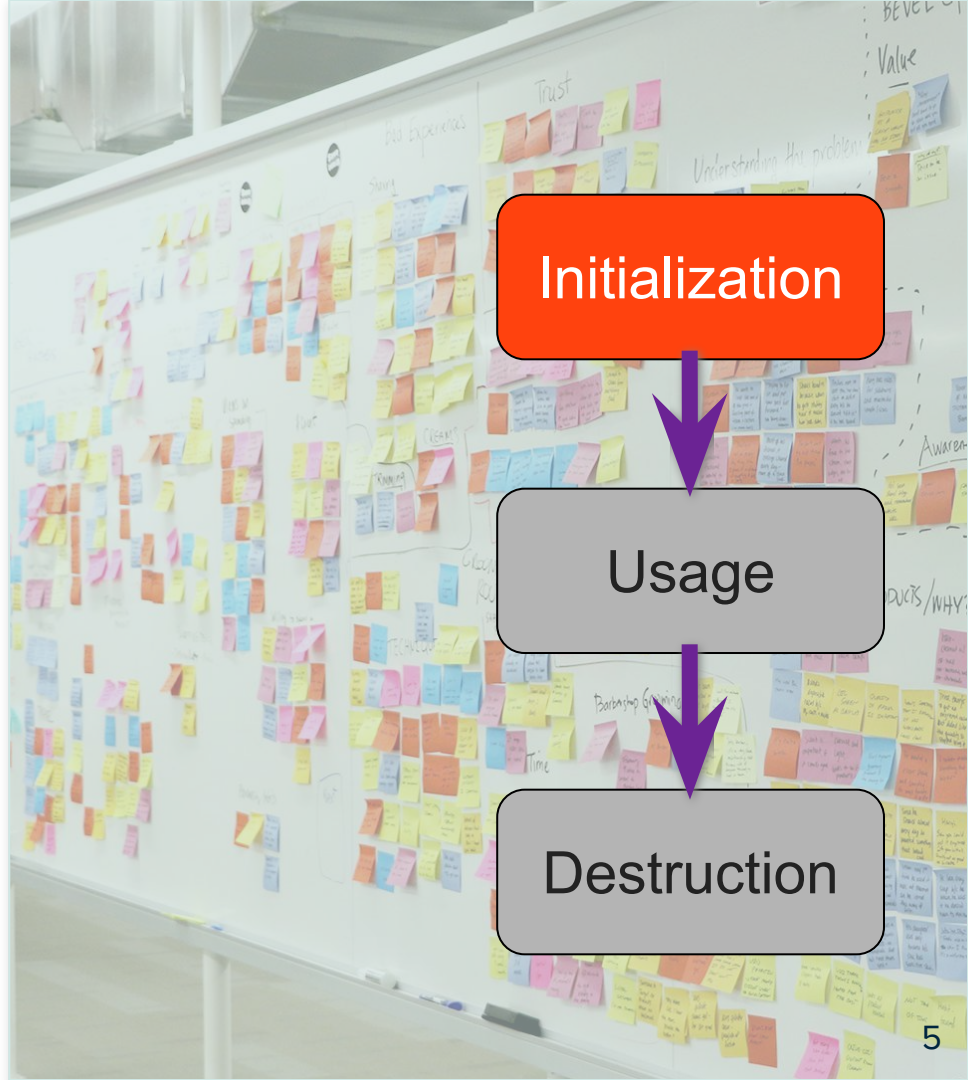
- Spring Bean container runs through three distinct phases
 - Initialization
 - Spring Beans are created
 - Dependency Injection occurs
 - Usage
 - Beans are available for use in the application
 - Destruction
 - Beans are released for Garbage Collection

Let's go deeper



Agenda

- **The Spring Bean Lifecycle**
 - Phase 1: Initialization
 - Phase 2: Usage
 - Phase 3: Destruction
- More on Bean Creation



Lifecycle of a Spring Application Context

(1) The Initialization Phase

- When a context is created the initialization phase completes

```
// Create the application from the configuration
```

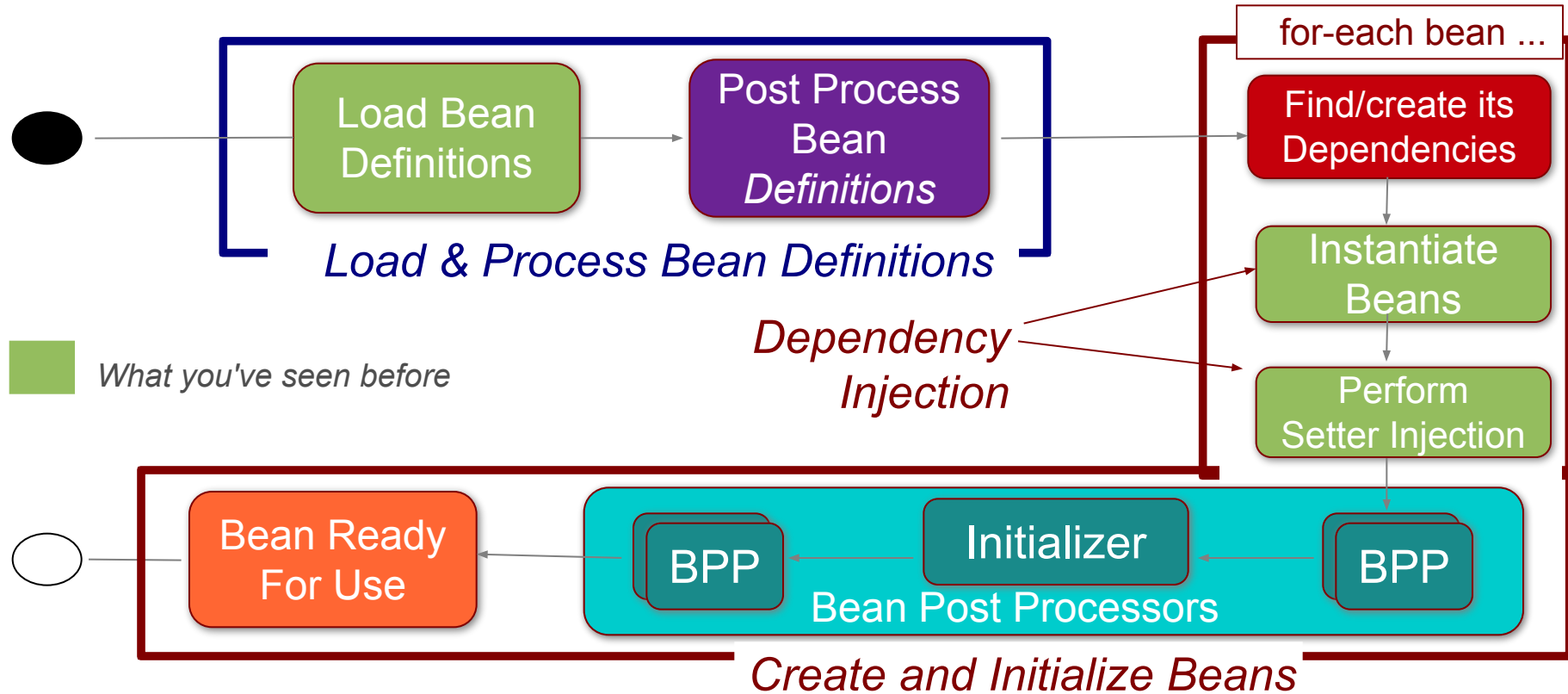
```
ApplicationContext context = SpringApplication.run(AppConfig.class);
```

- But what exactly happens in this phase?
 - Two separate steps
 - **Step A:** Load & Process Bean Definitions
 - **Step B:** Perform Bean Creation



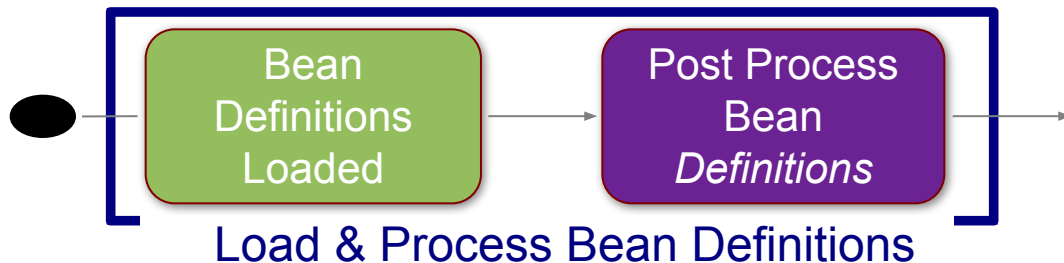
Initialization

Bean Initialization Steps



Step A. Load & Process Bean Definitions

- The `@Configuration` classes are processed
 - And/or `@Components` are scanned for
- Bean definitions added to a **BeanFactory**
 - Each indexed under its id and type
- Special **BeanFactoryPostProcessor** beans invoked
 - Can modify the *definition* of *any* bean



Load Bean Definitions

AppConfig.java

```
@Bean  
public TransferService transferService() { ... }  
@Bean  
public AccountRepository accountRepository() { ... }
```

TestInfrastructureConfig.java

```
@Bean  
public DataSource dataSource () { ... }
```

ApplicationContext is a
BeanFactory

transferService
accountRepository
dataSource

Can modify the *definition* of
any bean in the factory
before any objects are created

postProcessBeanFactory()

BeanFactoryPostProcessors

BeanFactoryPostProcessor

Internal Extension Point



- Applies transformations to bean *definitions*
 - Before objects are actually created
- Several useful implementations provided in Spring
 - Reading properties, registering a custom scope ...
- You can write your own (not common)
 - Implement **BeanFactoryPostProcessor** interface

```
public interface BeanFactoryPostProcessor {  
    public void postProcessBeanFactory  
                (ConfigurableListableBeanFactory beanFactory);  
}
```

BeanFactoryPostProcessor

Most Common Example



- Recall `@Value` and `${..}` variables

```
@Configuration
@PropertySource ( "classpath:/config/app.properties" )
public class ApplicationConfig {

    @Value("${max.retries}")
    int maxRetries;

    ...
}
```

```
# Maximum retry count
max.retries=3
app.properties
```

- Use a `PropertySourcesPlaceholderConfigurer` to evaluate them

BeanFactoryPostProcessor Declaration



- Simply create as a bean in the usual way
 - Define using `@Bean` method

```
@Bean
public static BeanFactoryPostProcessor myConfigurer() {
    return new MyConfigurationCustomizer();
}
```

```
public class MyConfigurationCustomizer implements BeanFactoryPostProcessor {
    // Perform customization of the configuration such as the placeholder syntax
}
```

BeanFactoryPostProcessor Considerations




- **BeanFactoryPostProcessor** is an *internal* bean invoked by Spring (not your code)
- It needs to run *before* any beans are created
 - Use of *static* **@Bean** method is recommended

```
@Bean
public static DeprecatedBeanWarner deprecatedBeanChecker() {
    return new DeprecatedBeanWarner();
}
```

Another example: used to check if any Spring Bean is being created from a deprecated class.

PropertySourcesPlaceholderConfigurer Considerations



```
graph LR; A[PropertySourcesPlaceholderConfigurer] --> B[Post Process Bean Definitions]; B --> C[ ];
```

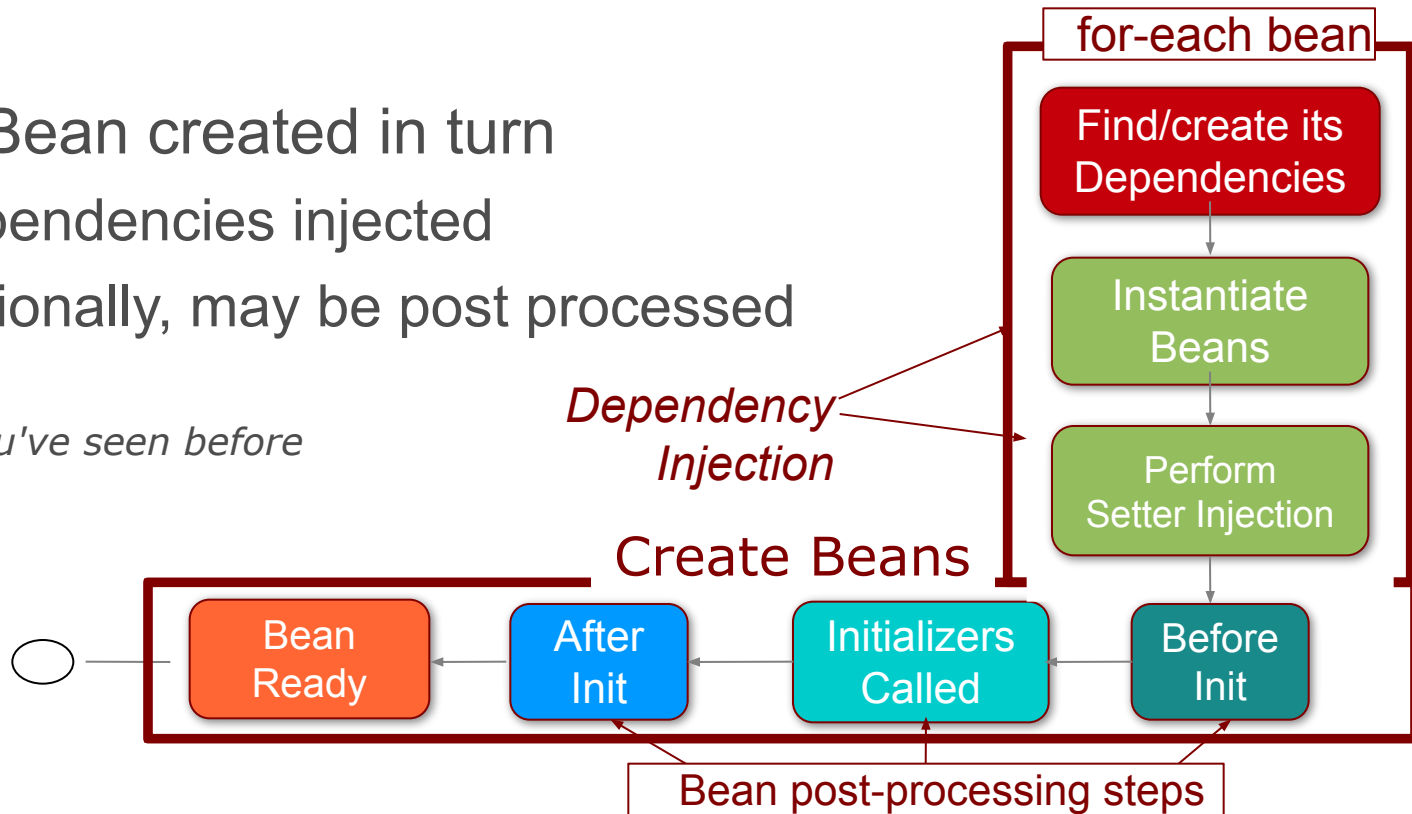
Post Process
Bean
Definitions

- Typically you do not need to setup this bean yourself
 - Spring Boot sets it up for you automatically
 - Spring from 4.3 sets up a basic value-resolver for you
 - If no **PropertySourcesPlaceholderConfigurer** bean exists
- When to create one manually?
 - Still using Spring 4.2 or earlier
 - You wish to configure how it works
 - Can ignore System Environment and/or System Variables

Step B: Perform Bean Creation

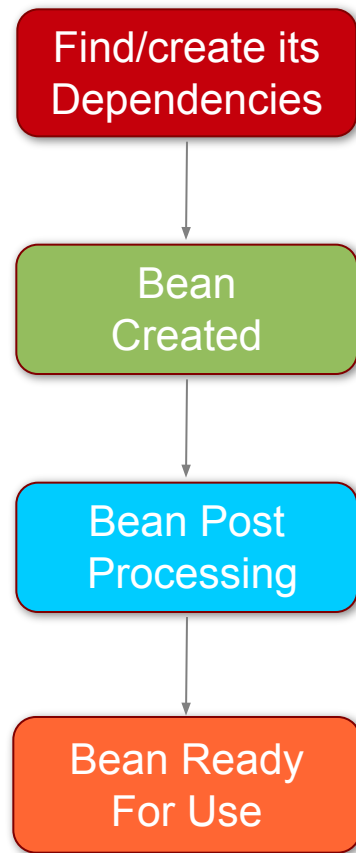
- Each Bean created in turn
 - Dependencies injected
 - Optionally, may be post processed

 *What you've seen before*



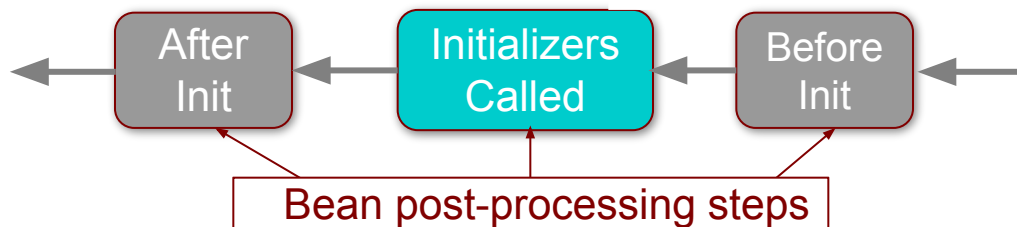
Bean Creation Sequence of Events - Singleton

- Bean creation
 - Created with dependencies injected
 - Each *singleton* bean *eagerly* instantiated
 - Unless marked as lazy
- Next each bean goes through a *post-processing* phase
 - *BeanPostProcessors*
- Now bean is fully initialized & ready to use
 - Tracked by id until the context is destroyed



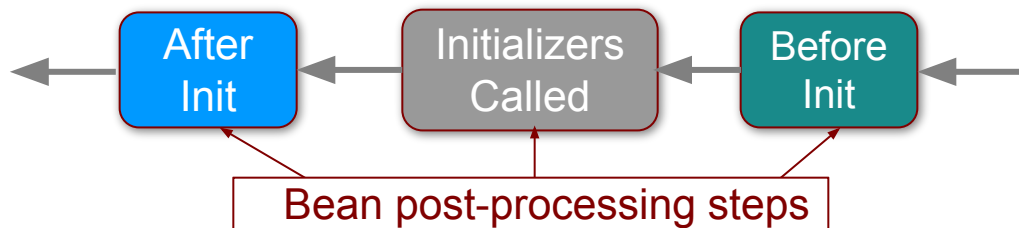
The Initializer Extension Point

- Special case of a bean post-processing
 - Causes initialization methods to be called
 - Such as `@PostConstruct`, `init-method`
- Internally Spring uses several initializer BPPs
 - *Example: `CommonAnnotationBeanPostProcessor` enables `@PostConstruct`, `@Resource` ...*



BeanPostProcessor Extension Point

- Important extension point in Spring
 - Can modify bean *instances* in any way
 - *Powerful* enabling feature
 - Will run against *every* bean
 - Can modify a bean before and/or after Initialization
 - **BeforeInit** runs *before* the initializer
 - **AfterInit** runs *after* the initializer



BeanPostProcessor Interface

*Course will show
several BPPs*

- Bean Post Processors implement a known interface
 - Spring provides several implementations
 - You can write your own (not common)
 - Typically implement the after initialization method

```
public interface BeanPostProcessor {  
    public Object postProcessBeforeInitialization(Object bean, String beanName);  
    public Object postProcessAfterInitialization(Object bean, String beanName);  
}
```

Post-processed bean

Original bean

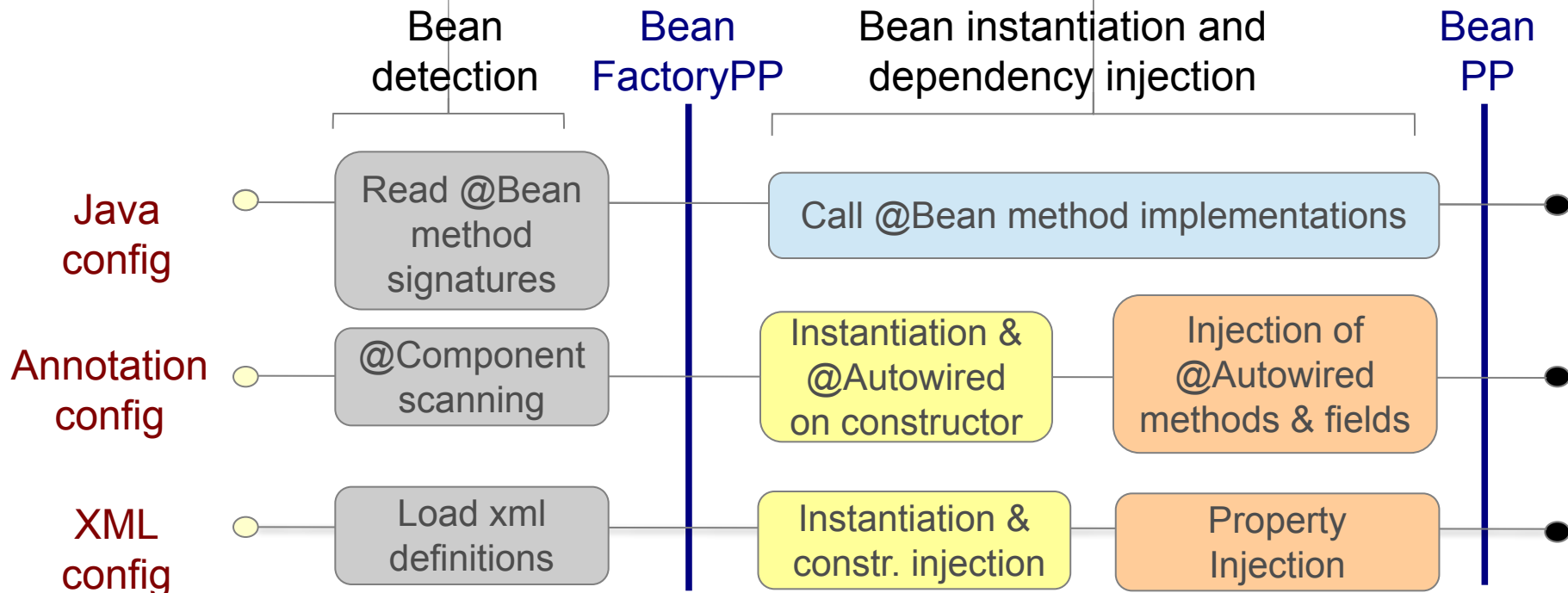
Example: CustomBeanPostProcessor

@Component ←

Can be found by component-scanner, like any other bean

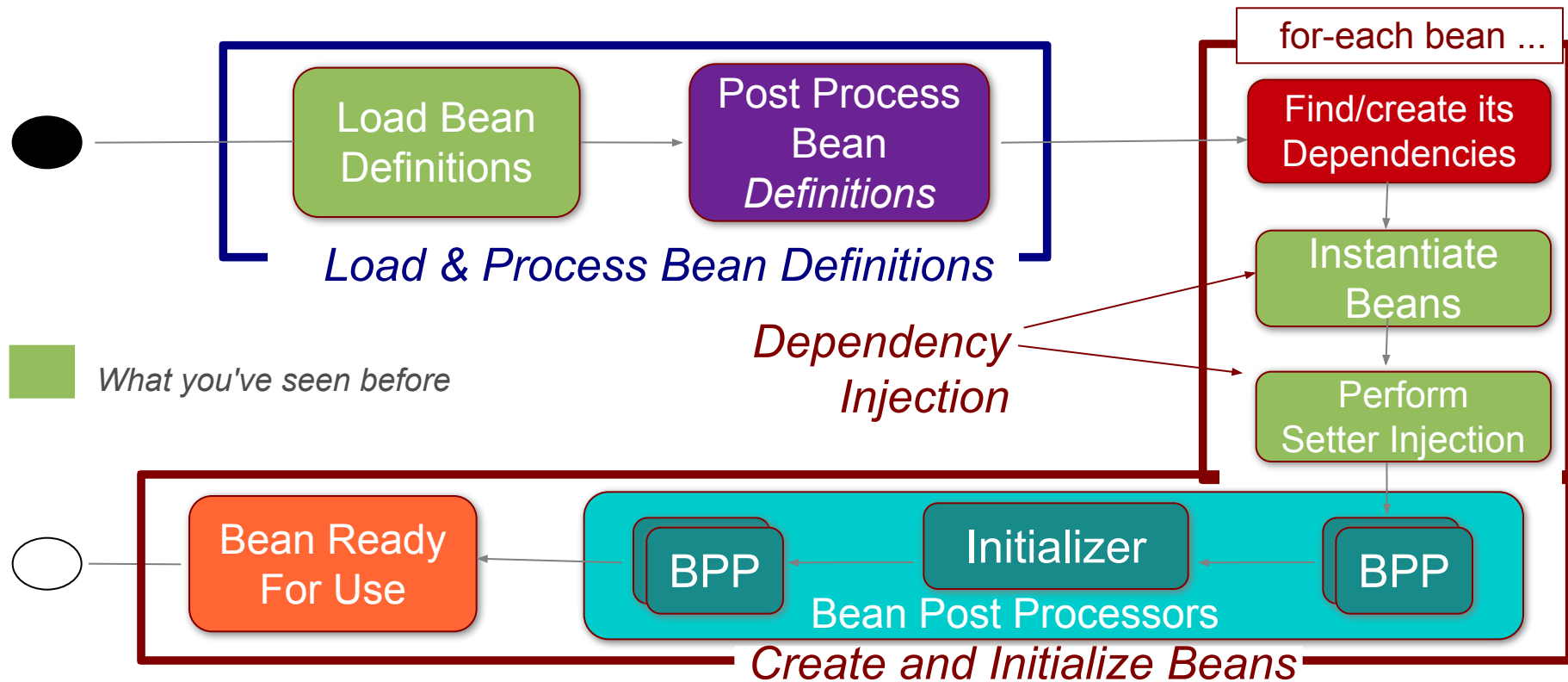
```
public class CustomBeanPostProcessor implements BeanPostProcessor {  
  
    public Object postProcessBeforeInitialization(Object bean, String beanName) {  
        // Some code  
        return bean;    // Remember to return your bean or you'll lose it!  
    }  
  
    public Object postProcessAfterInitialization(Object bean, String beanName) {  
        // Some code  
        return bean;    // Remember to return your bean or you'll lose it!  
    }  
}
```

Configuration Lifecycle



BeanFactoryPP → BeanFactoryPostProcessor
BeanPP → BeanPostProcessor

Bean Initialization Steps



Agenda

- **The Spring Bean Lifecycle**

- Phase 1: Initialization

- **Phase 2: Usage**

- Phase 3: Destruction

- **More on Bean Creation**



Lifecycle of a Spring Application Context

(2) The Use Phase



- When you invoke a bean obtained from the context

```
ApplicationContext context = // get it from somewhere
```

```
// Lookup the entry point into the application
```

```
TransferService service = context.getBean("transferService", TransferService.class);
```

```
// Use it!
```

```
service.transfer(new MonetaryAmount("50.00"), "1", "2");
```

- But exactly what happens in this phase?

Case I: You Bean is Just a Bean

- The bean is just your raw object
 - Simply invoked directly (nothing special)

`transfer("$50", "1", "2")`



`TransferServiceImpl`

- *Nothing new here!*

Case II: Your Bean is a Proxy

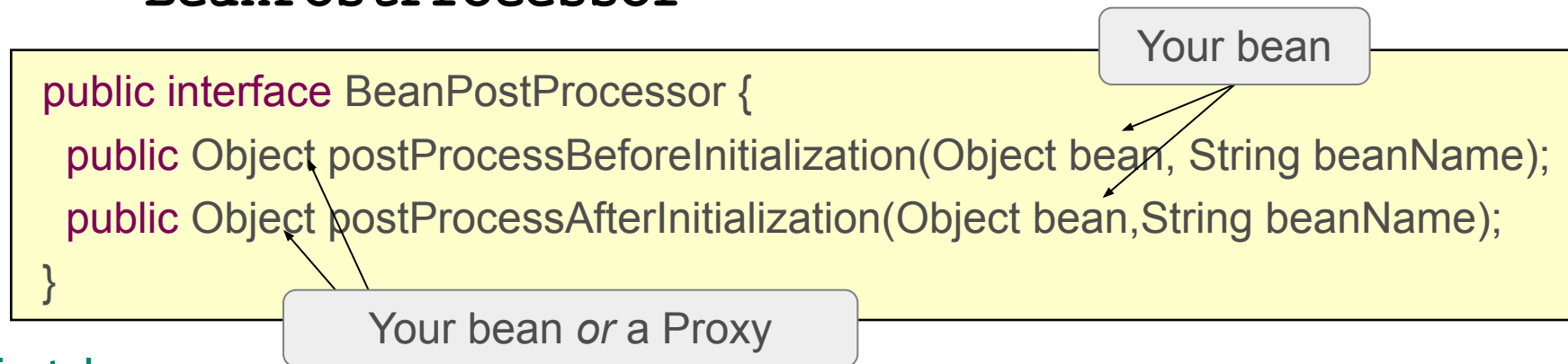
- Your bean is wrapped in a *proxy*

`transfer("$50", "1", "2")`



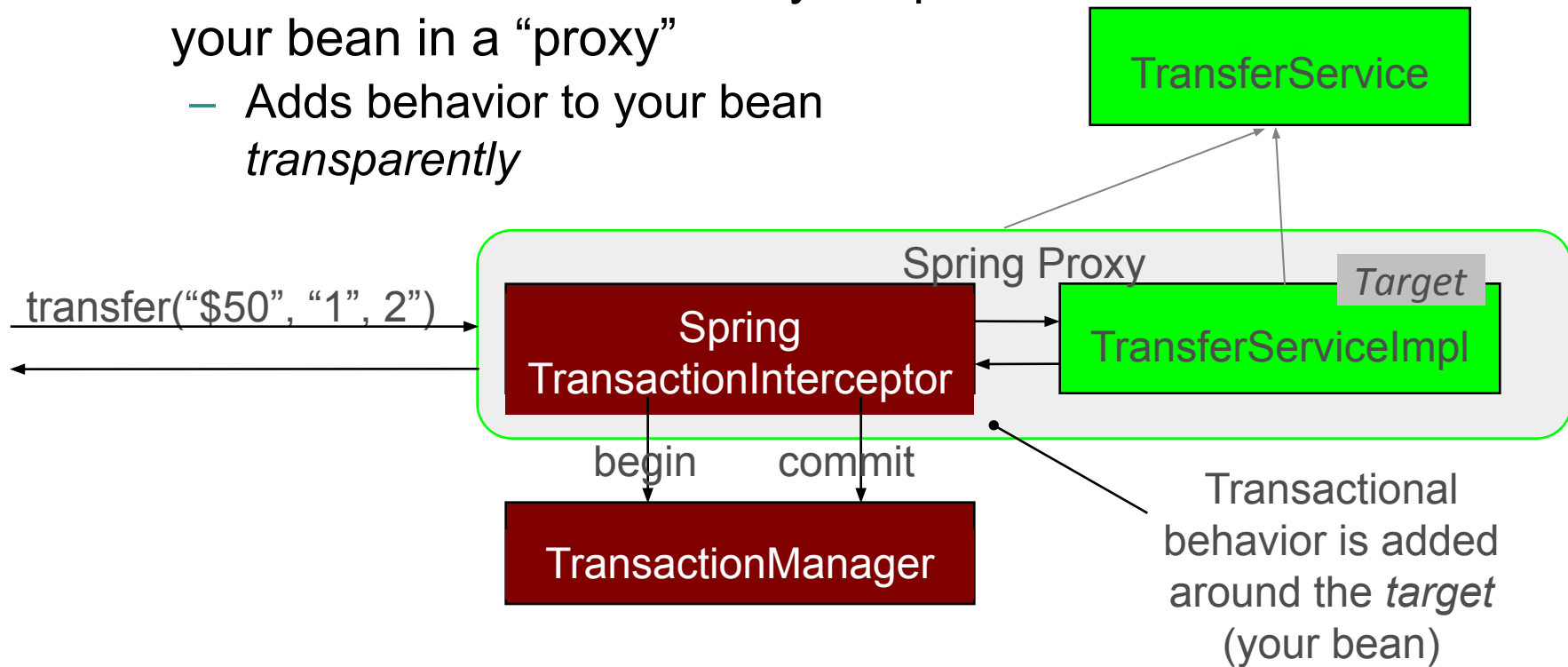
The diagram shows a horizontal arrow pointing from the code `transfer("$50", "1", "2")` to a blue rectangular box labeled "Spring Proxy". Inside this box is a smaller green rectangular box labeled "TransferServiceImpl".

- Proxy created during initialization phase by a **BeanPostProcessor**



Proxy Power Example: Transactions

- **BeanPostProcessor** may wrap your bean in a “proxy”
 - Adds behavior to your bean *transparently*



Kinds of Proxies

- Spring supports both *JDK* or *CGLib* proxies

JDK Proxy

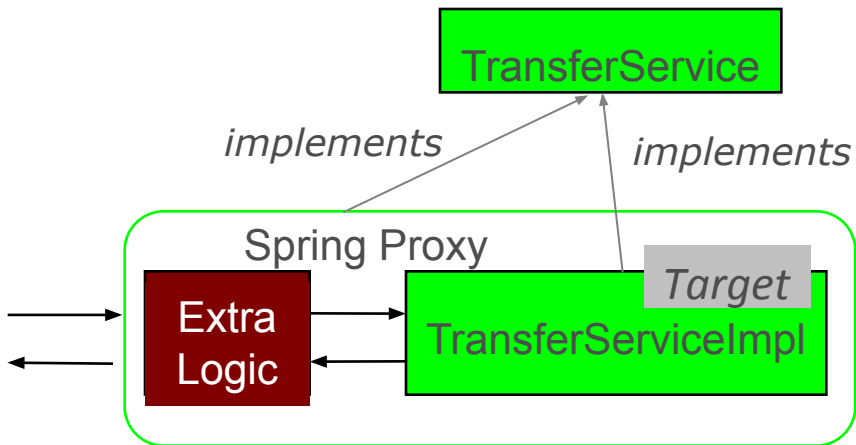
- Also called *dynamic* proxies
- API is built into the JDK
- Requirements: Java interface(s)
- *All* interfaces proxied

CGLib Proxy

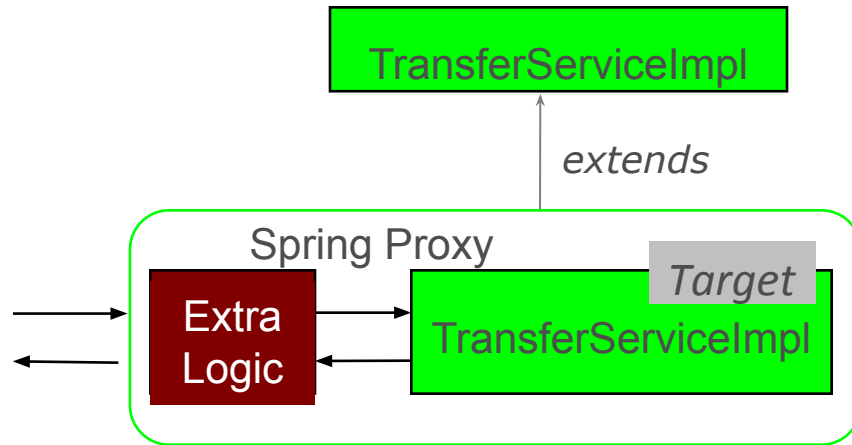
- NOT built into JDK
- Included in Spring jars
- Used when interface not available
- Cannot be applied to final classes or methods

JDK vs CGLib Proxies

- JDK Proxy
 - Interface based



- CGLib Proxy
 - subclass based



Agenda

■ The Spring Bean Lifecycle

— Phase 1: Initialization

— Phase 2: Usage

— Phase 3: Destruction

■ More on Bean Creation



Lifecycle of a Spring Application Context

(3) *The Destruction Phase*

- The context is closed (or shutdown hook invoked)

```
ConfigurableApplicationContext context = // get it from somewhere  
... // Do something
```

```
// Shutdown  
context.close();
```

- But exactly what happens in this phase?



Destruction

Bean Clean Up



Destruction

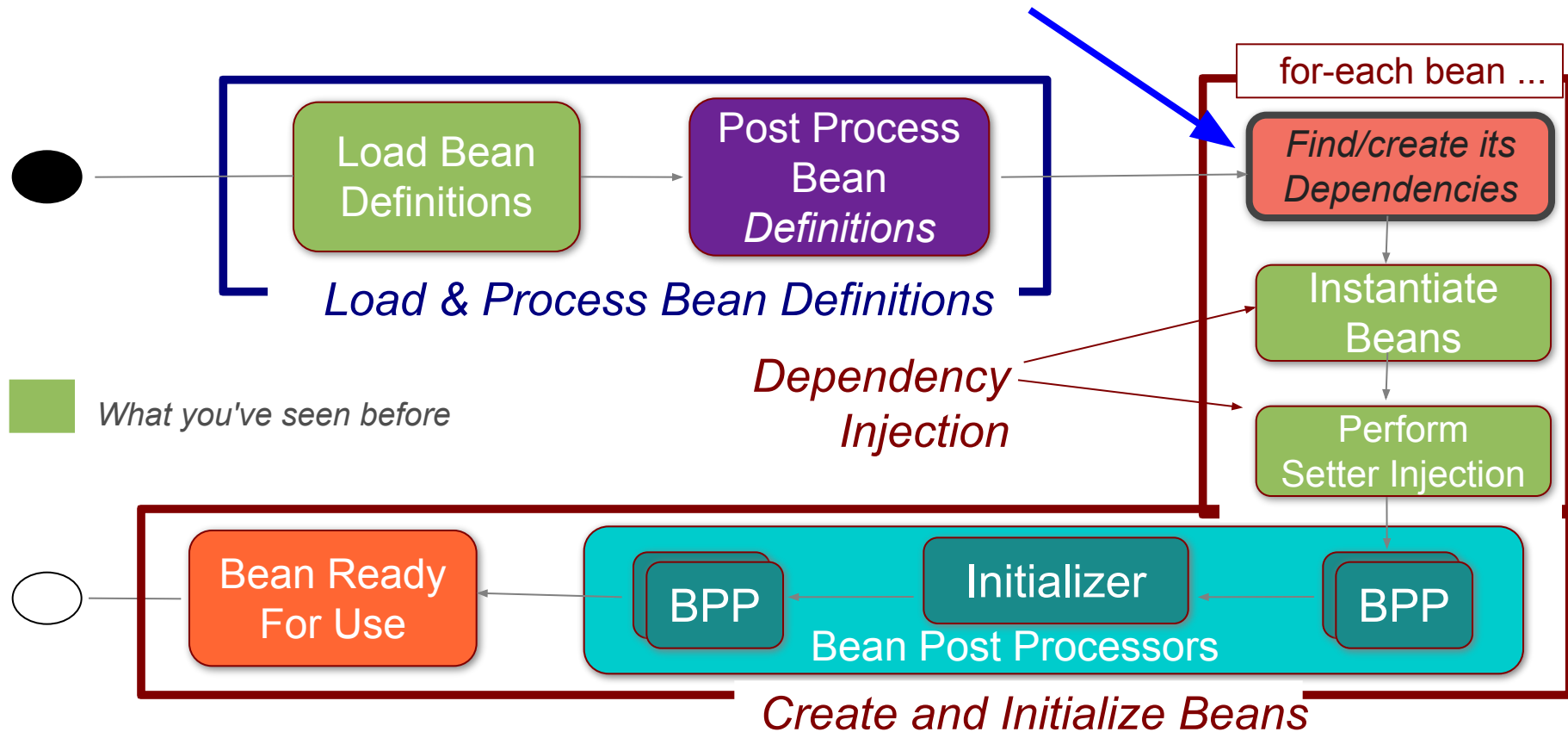
- All beans are cleaned up
 - Any registered `@PreDestroy` methods are invoked
 - Beans released for the Garbage Collector to destroy
- Also happens when any bean goes out of scope
 - Except Prototype scoped beans
- **Note:** Only happens if application shuts down gracefully
 - Not if it is killed or fails

Agenda

- The Spring Bean Lifecycle
- **More on Bean Creation**



Bean Initialization – Determining Dependencies



Creating Dependencies

- Beans have to be created in the right order
 - Beans must be created *after* their dependencies
- Two steps
 - Evaluate dependencies for each bean
 - Get each dependency needed
 - Create any if need be
 - This is recursive



How Does Spring Know What to Inject?

- Implementing *Dependency Injection*
 - Spring defaults to looking for a bean of correct type
 - May use bean name: `@Qualifier`, `@Resource`

You can force dependency order

```
@Component
@DependsOn("accountService")
class TransferService {
    ...
}
```

```
@Bean
@DependsOn("accountService")
public TransferService transferService() {
    return ...
}
```

Create `TransferService` *after* bean called `accountService`

Determining Bean Name & Type

Definition	Name	Type
1. Java config	From Annotation Or from method name	From method return type
2. Annotation-based Config	From Annotation Or derived from classname	Directly from annotated class
3. XML	From id or name attribute Or derived from classname	From class attribute

- 1 typically returns an *interface*
- 2 and 3 provide *actual* implementation class

Why Won't This Work?

```
@Configuration  
class Config {
```

```
    @Bean  
    public TransferService transferService(AccountRepository repo) {  
        return new BankTransferServiceImpl( repo );  
    }
```

```
    @Bean  
    public BankingClient bankingService(BankService svc) {  
        return new BankingClient( svc );  
    }  
    ...
```

```
// TransferService does not extend BankService  
class BankTransferServiceImpl  
    implements TransferService, BankService {  
    ...  
}
```

??



No @Bean method exists returning a BankService

Solution 1: Return *Actual* Type

```
@Configuration  
class Config {
```

```
    @Bean
```

```
    public BankTransferServiceImpl transferService(AccountRepository repo) {  
        return new BankTransferServiceImpl( repo );  
    }
```

```
    @Bean
```

```
    public BankingClient bankingService(BankService svc) {  
        return new BankingClient( svc );  
    }
```

```
    ...
```

```
// BankTransferServiceImpl is a BankService  
class BankTransferServiceImpl  
    implements TransferService, BankService {  
    ...  
}
```

Can determine **BankTransferServiceImpl** implements
both **TransferService** *and* **BankService**



Solution 2: Return Composite Interface

```
// BankTransferServiceImpl is a BankService  
interface BankTransferService  
    extends TransferService, BankService {  
}
```

```
class BankTransferServiceImpl  
    implements BankTransferService { ... }
```

```
@Configuration  
class Config {
```

```
    @Bean
```

```
    public BankTransferService transferService(AccountRepository repo) {  
        return new BankTransferServiceImpl( repo );  
    }
```

```
    @Bean
```

```
    public BankingClient bankingService(BankService svc) {  
        return new BankingClient( svc );  
    }
```

```
    ...
```

Can determine **BankTransferService** extends
both **TransferService** and **BankService**

Defining Spring Beans – Best Practice



- Aim to be “*sufficiently expressive*”
 - Return interfaces *except*
 - Where *multiple* interfaces exist
 - *AND* they are needed for dependency injection
 - Writing to interfaces is *good practice*
- **Warning:** Even if you return actual types
 - Still dependency inject *interfaces*
 - Injecting actual types is brittle
 - Dependency injection may *fail* if the bean is proxied or a different implementation returned



Topics Covered

- Spring Bean Lifecycle
 - Three phases: *initialize, use, destroy*
 - **BeanFactoryPostProcessor**
 - Processes bean *definitions* (**no** beans yet)
 - Allocate using *static* **@Bean** method
 - **BeanPostProcessor**
 - Processes Beans
 - Performs initialization, creates proxies, ...
- Care with **@Bean** Definitions
 - When to consider your return types

