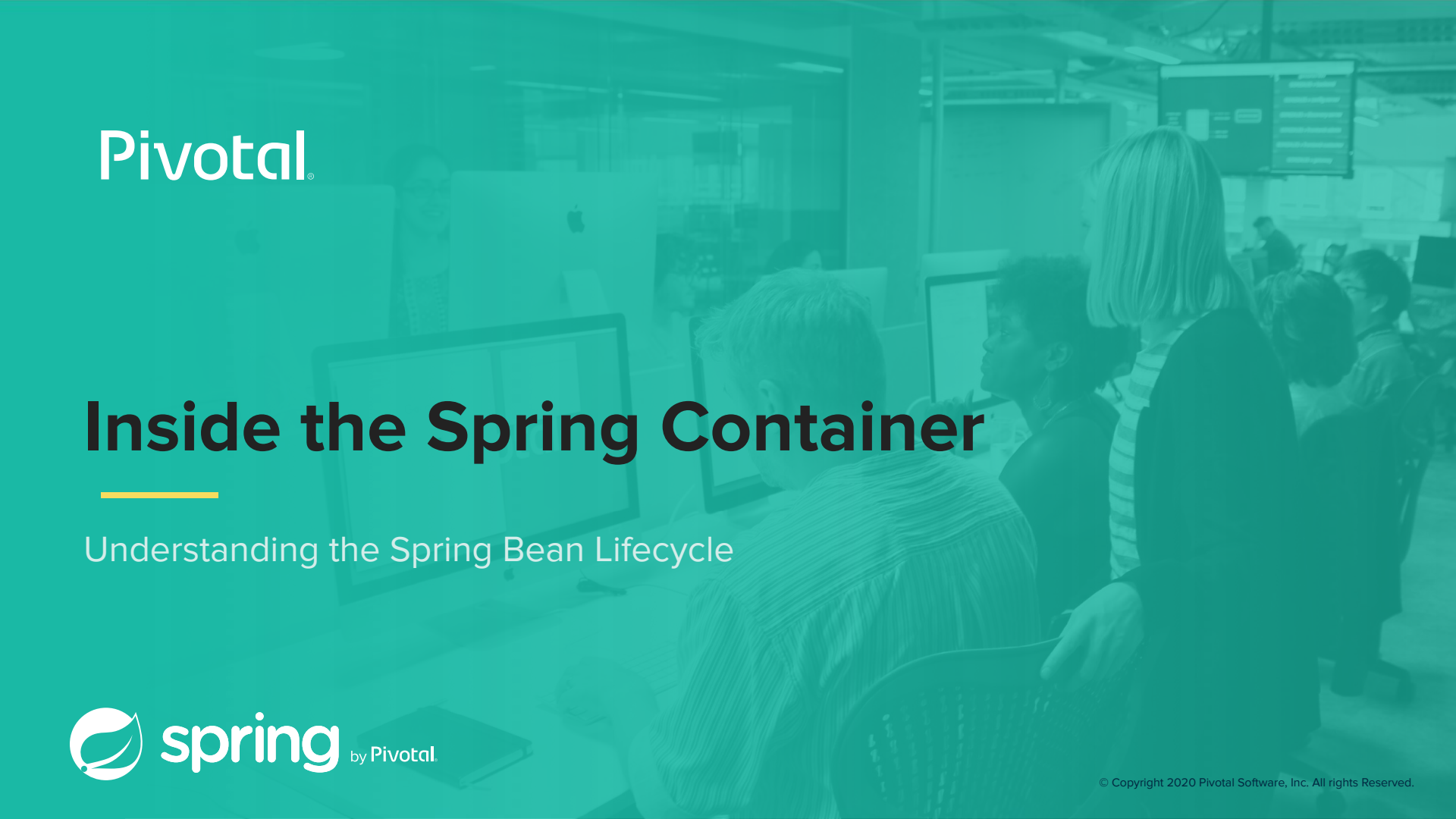


Pivotal

Inside the Spring Container

Understanding the Spring Bean Lifecycle

spring by Pivotal

Objectives

After completing this lesson, you should be able to

- Explain the Spring Bean Lifecycle
- Use a **`BeanFactoryPostProcessor`** and a **`BeanPostProcessor`**
- Explain how Spring proxies add behavior at runtime
- Describe how Spring determines bean creation order
- Avoid issues when Injecting beans by type

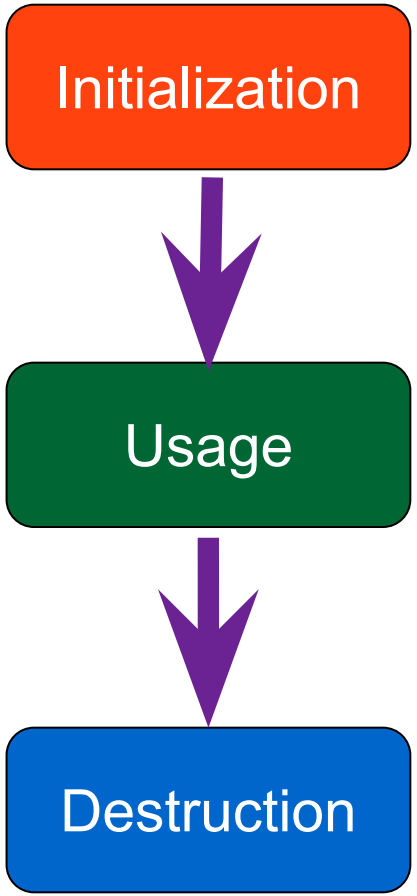
Agenda

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

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

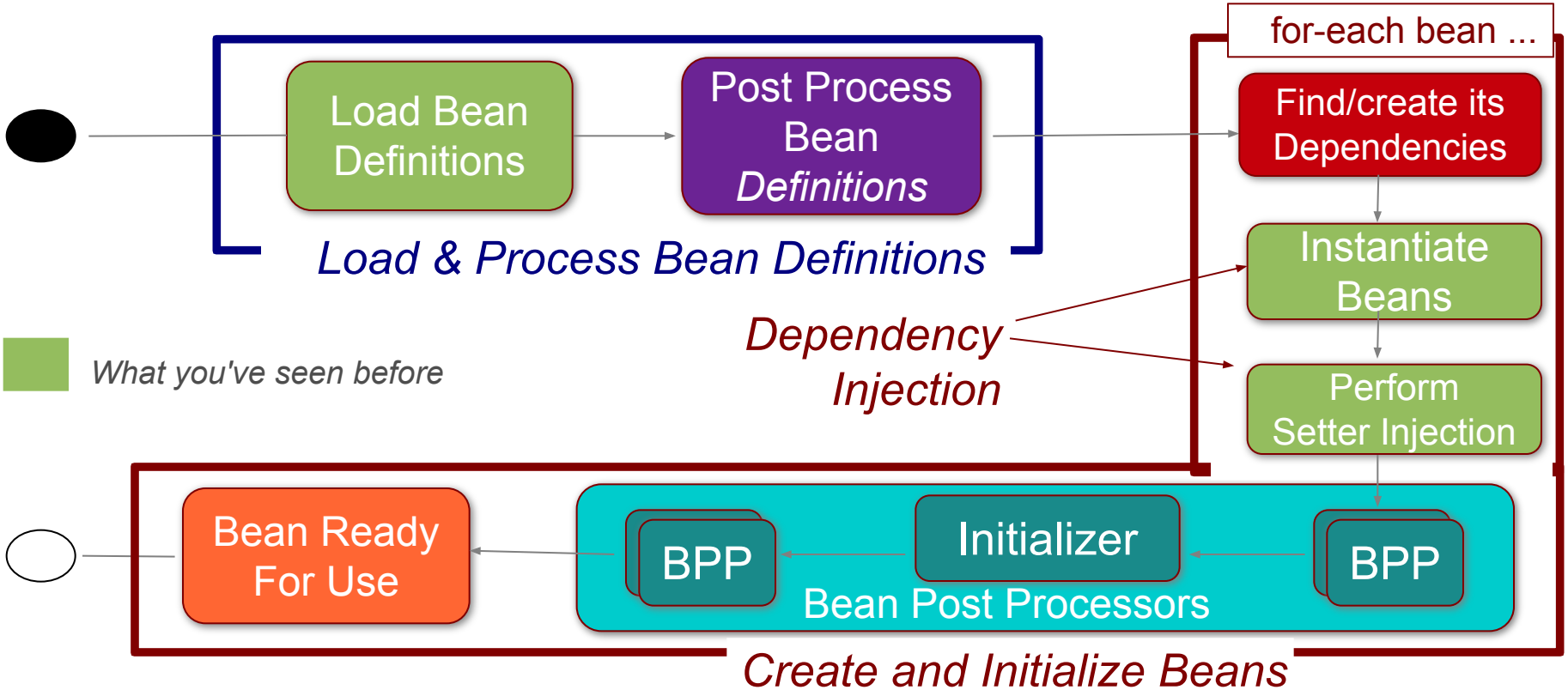
(1) The Initialization Phase

- When a context is created, the initialization phase completes

```
// Create application context from the configuration
ApplicationContext context = SpringApplication.run(AppConfig.class);
```

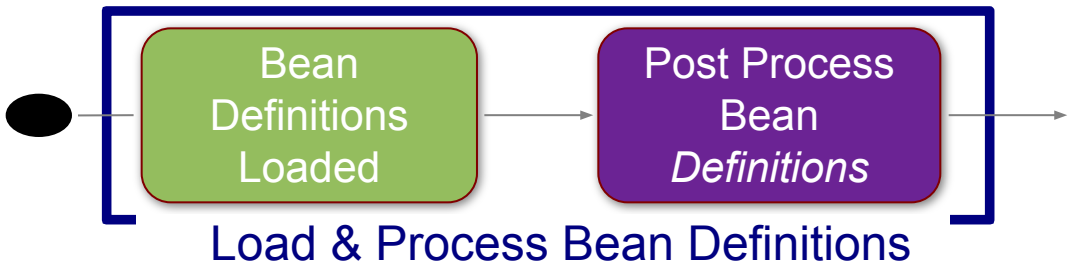
- But what exactly happens in this phase?
  - Two separate steps
    - **Step A:** Load & Process Bean Definitions
    - **Step B:** Perform Bean Creation

Bean Initialization Steps



Step A: Load & Process Bean Definitions

- The **`@Configuration`** classes are processed
  - And/or **`@Component`** annotated classes 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



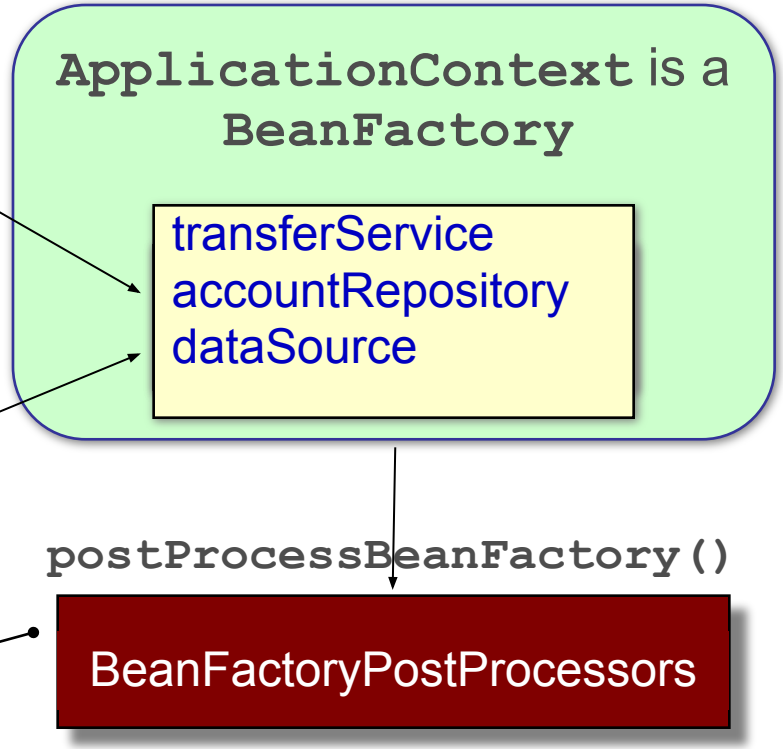
Load Bean Definitions

AppConfig.java

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

TestInfrastructureConfig.java

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



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

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
  - *This is* a `BeanFactoryPostProcessor`

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();
}
```

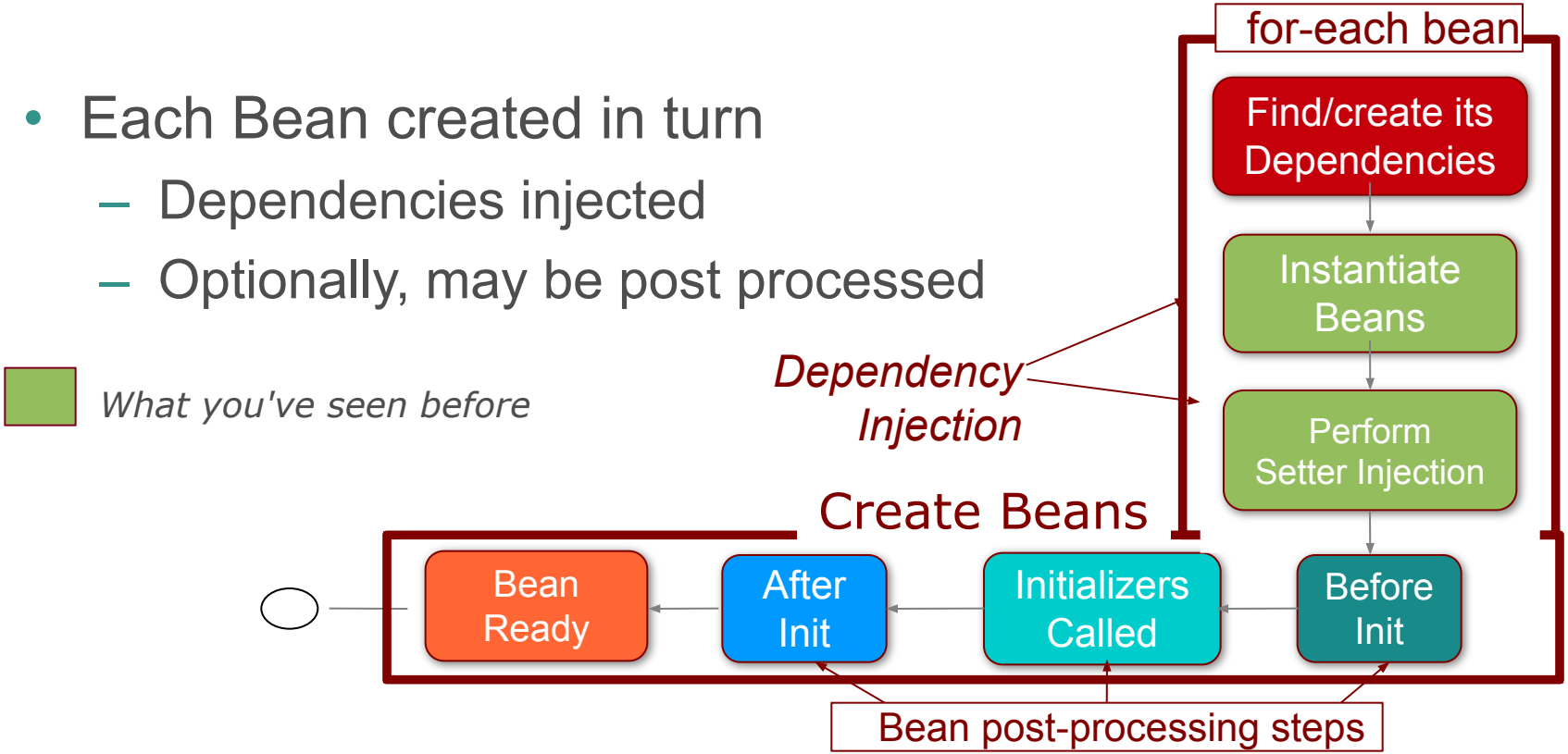
Another example: used to check if any Spring Bean is being created from a deprecated class.

`PropertySourcesPlaceholderConfigurer` *Considerations*

- 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

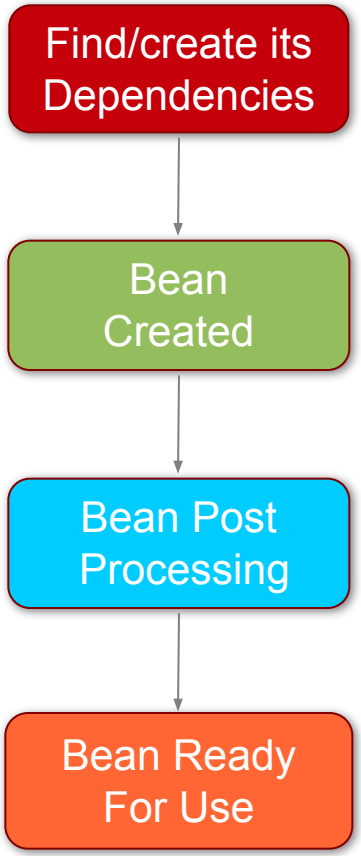
Step B: Perform Bean Creation

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



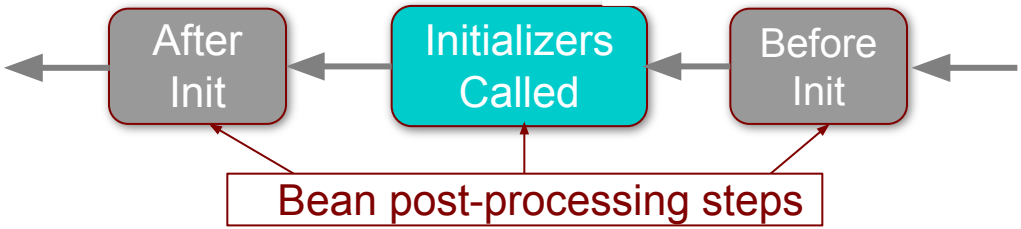
Bean Creation Sequence of Events - Singleton

- 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



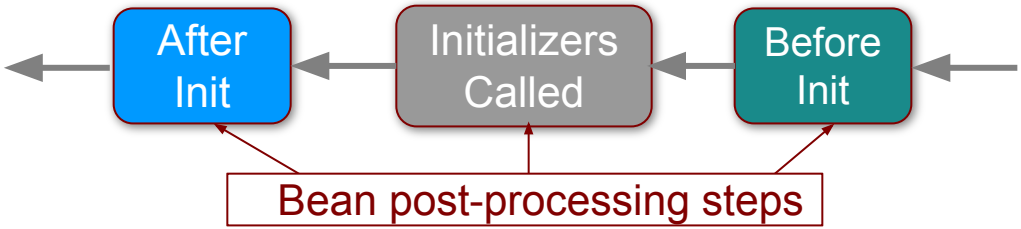
The Initializer Extension Point

- 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

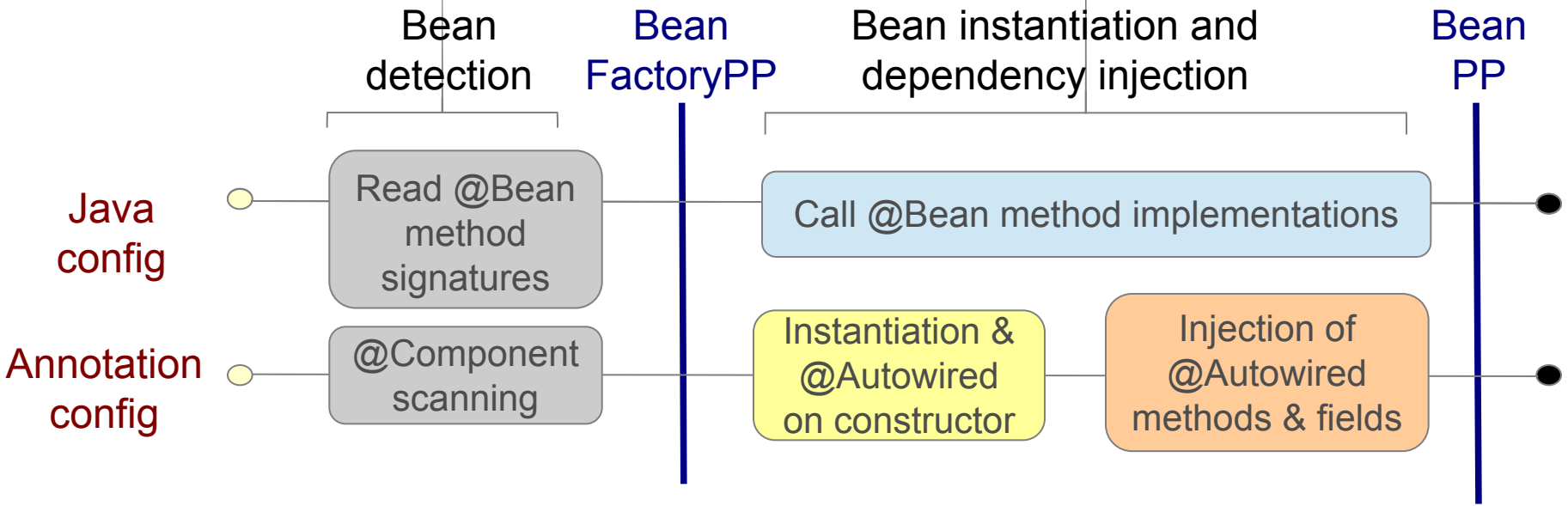
Original bean

Example: `CustomBeanPostProcessor`

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

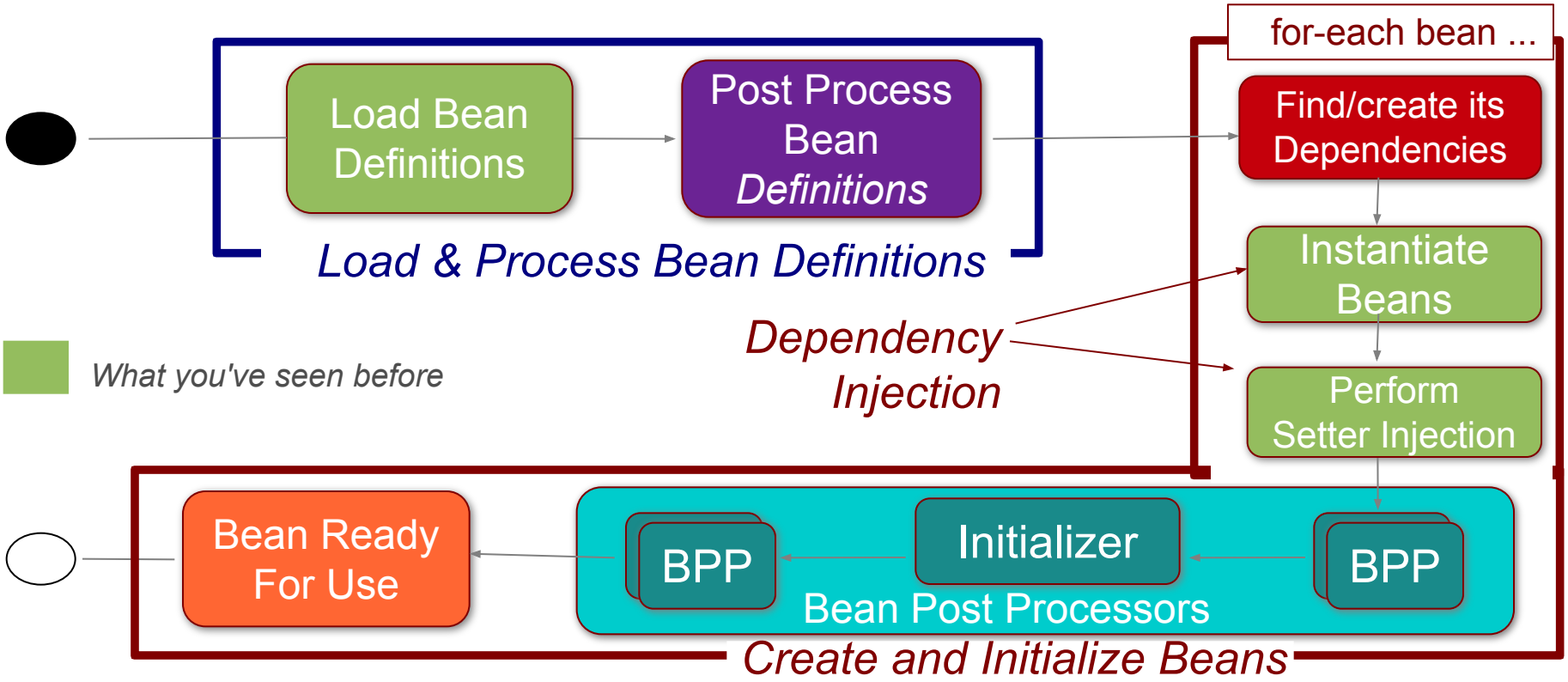
```
@Component
public class CustomBeanPostProcessor implements BeanPostProcessor {

    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



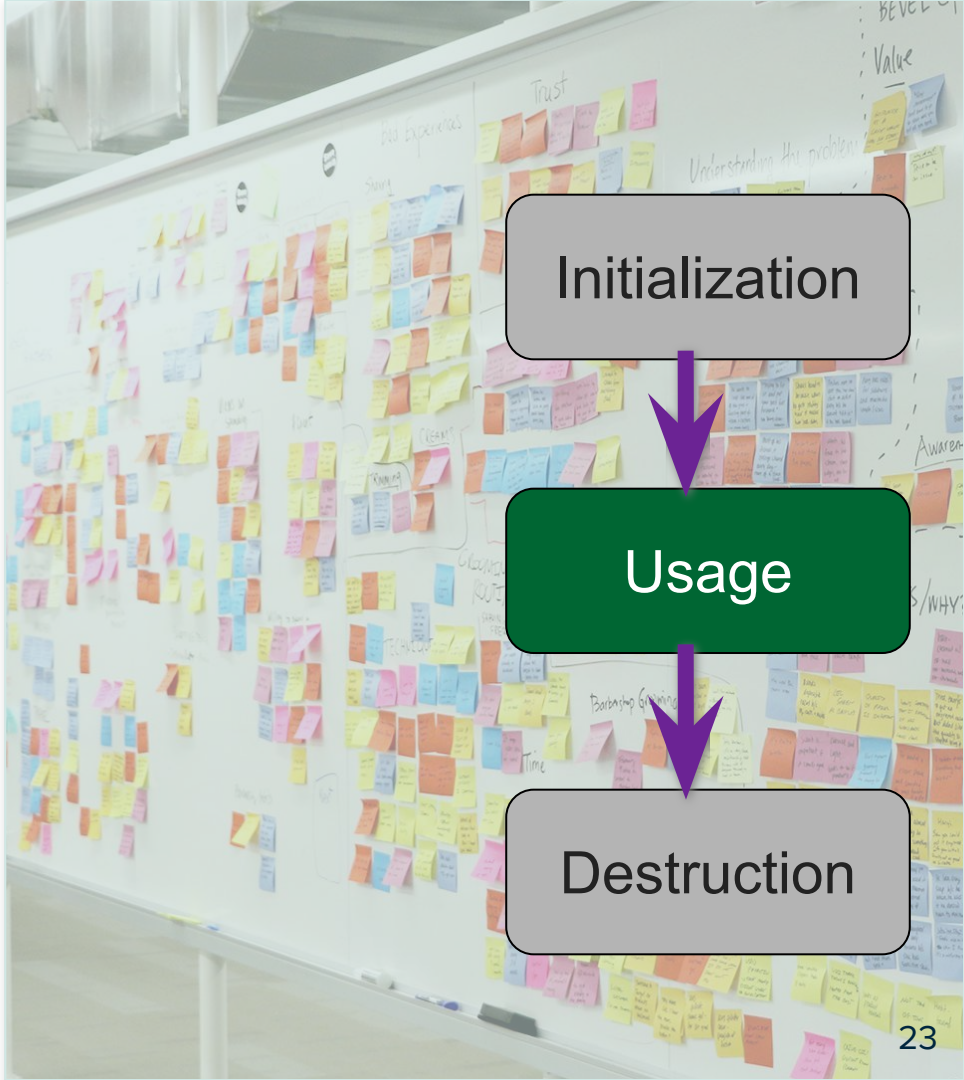
BeanFactoryPP → BeanFactoryPostProcessor
BeanPP → BeanPostProcessor

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

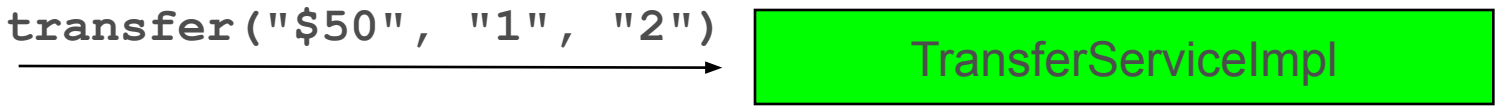
- When you invoke a bean obtained from the context

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

// Retrieve a bean through application context. (Typically you will use @Autowired
// in order to get a reference to a bean.)
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

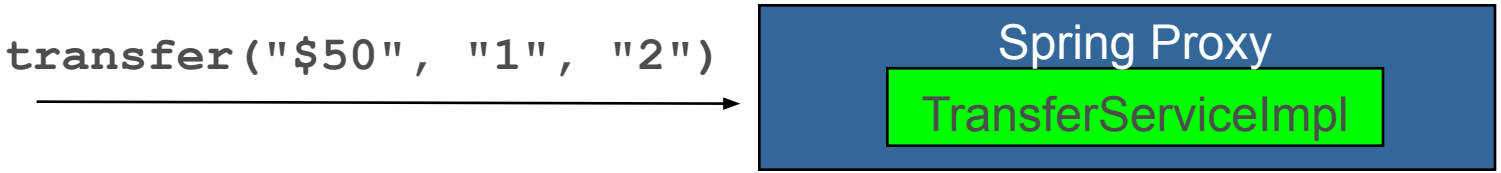
- The bean is just your raw object
  - Simply invoked directly (nothing special)



  - *Nothing new here!*

Case II: Your Bean is a Proxy

- Your bean is wrapped in a *proxy*



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

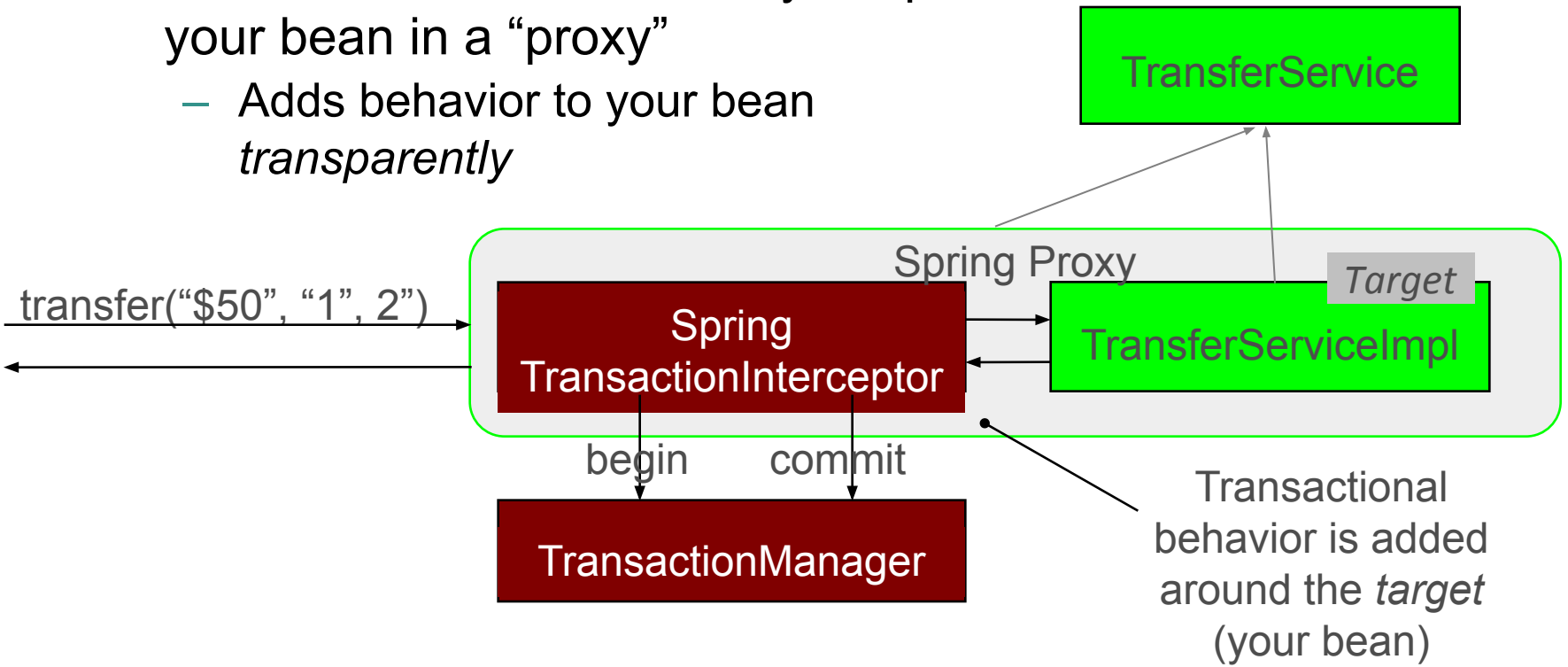
```
public interface BeanPostProcessor {
  public Object postProcessBeforeInitialization(Object bean, String beanName);
  public Object postProcessAfterInitialization(Object bean,String beanName);
}
```

Your bean

Your bean *or* a Proxy

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

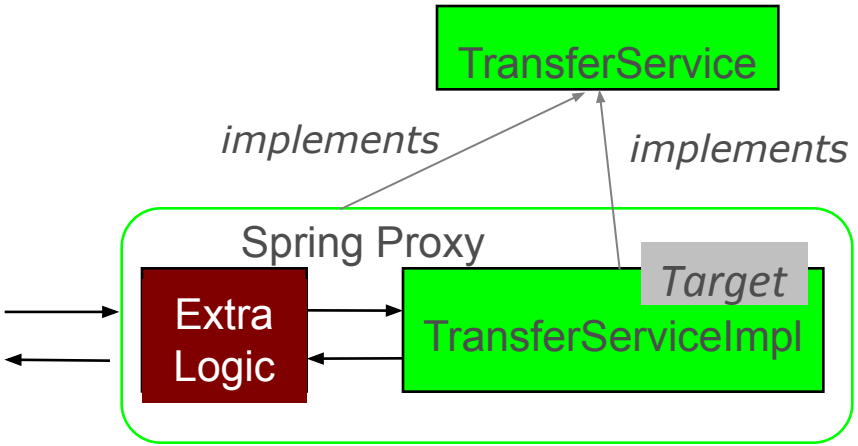
- Also called *dynamic* proxies
- API is built into the JDK
- Requirements: Java interface(s)
- *All* interfaces proxied

CGLib Proxy

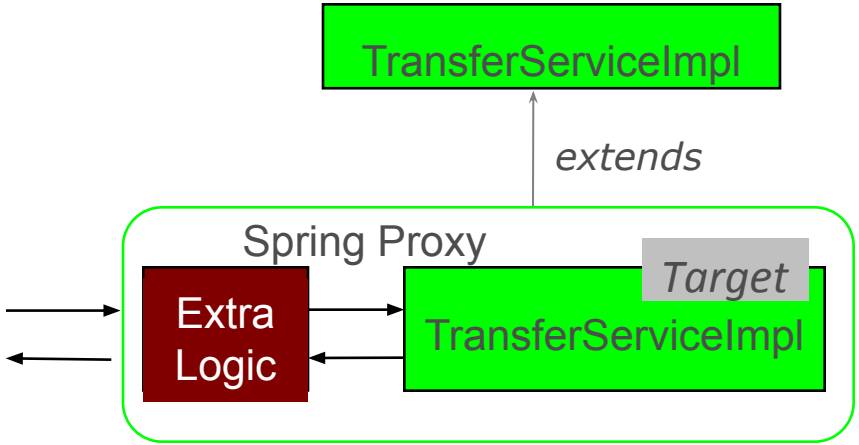
- NOT built into JDK
- Included in Spring jars
- Used when interface not available
- Cannot be applied to final classes or methods

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?

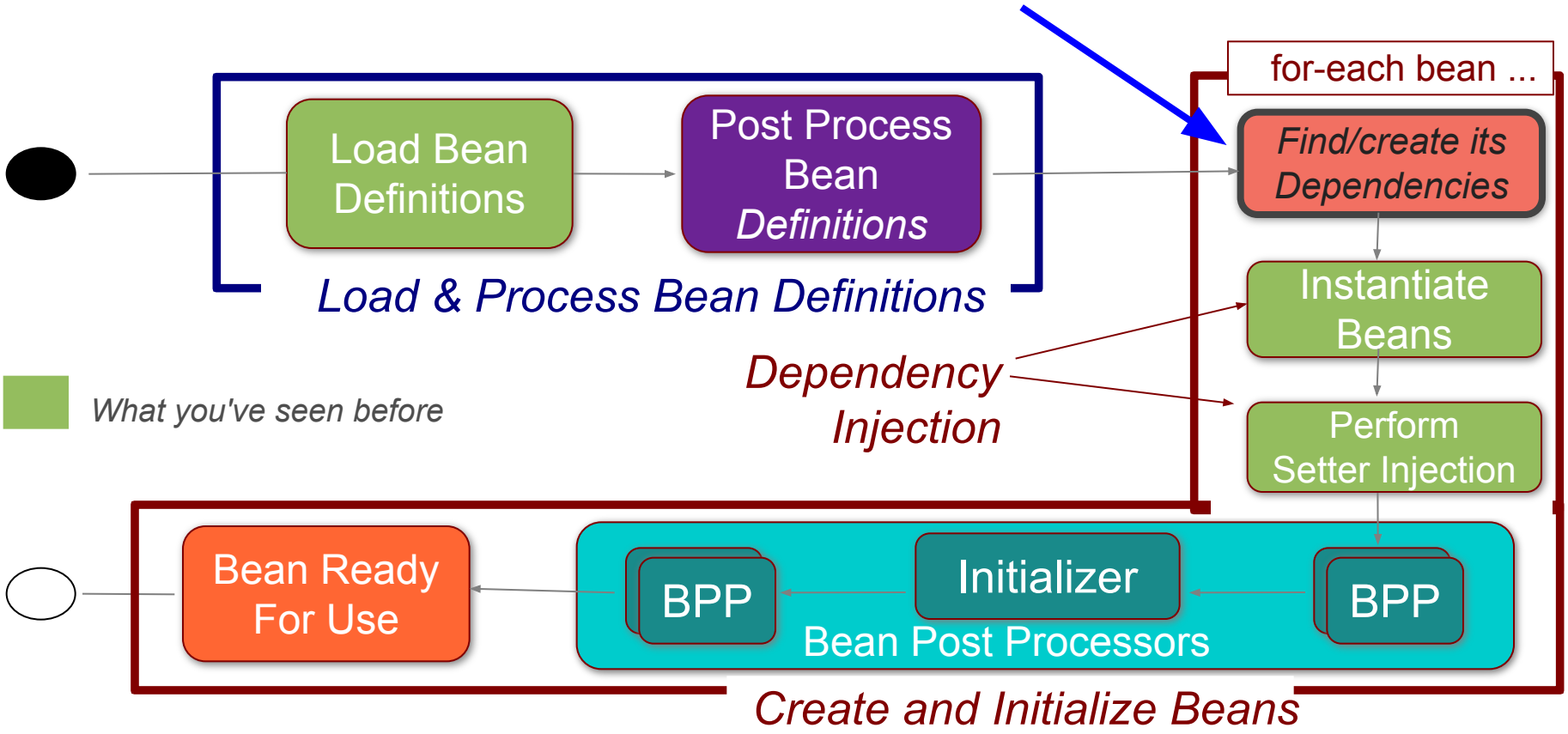
Bean Clean Up

- 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

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 @Bean name/value attr Or from method name	From method return type
3. Annotation-based Config	From Annotation value attr Or derived from class name	From annotated class

- 1 typically returns an *interface*
- 2 provides *actual* implementation class

Why Won't This Work?

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

```
@Configuration
class Config {

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

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

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

Summary

- 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