Introduction to the Spring Framework

Frank P Moley III
@fpmoles

Course Materials

* https://github.com/fpmoles/talks-spring-intro-workshop

Who am I

* Software Architect with Garmin International
* LinkedIn Learning/lynda.com content author
* 15+ years of experience in all aspects of SDLC
* Spring 3.2 & 4.0 certified
* Co founder of local Java User Group & Spring User Group

Workshop requirements

* Exercise and solution files from github
* Java 1.8 on PATH
* Maven on PATH
* Tomcat 8.5
* Java IDE is preferred (I use Idea IntelliJ but STS or Eclipse works fine)

Workshop Agenda

* Part 1: The Application Context & Spring Lifecycle
* Part 2: Aspect Oriented Programming with Spring
* Part 3: Data Access with Spring
* Part 4: Web Applications with Spring
* Part 5: Spring Boot
* Part 6: Testing with Spring (As time permits)
* Part 7: Spring Security (As time permits)

Introduction to Spring

What is Spring?

* Premier framework for building Internet and Enterprise applications with Java
* Open Source
* Lightweight
* IoC container

Spring Framework

* J2EE abstractions for most common enterprise systems
* Reduces code duplication of consuming these systems
* Provides true repeatable patterns through abstractions
* Behavior support for common enterprise and internet tasks

Open Source!!!

* Spring is truly an open source framework
* Strong community back by some big players
* Apache 2 License
* Community driven, but corporately overseen

Lightweight

* Small libraries
* Broken up by function
* Doesn't require Application Server, but can leverage
* Can be leveraged without Spring imports in your code
* Supports OOP & DRY

Inversion of Control

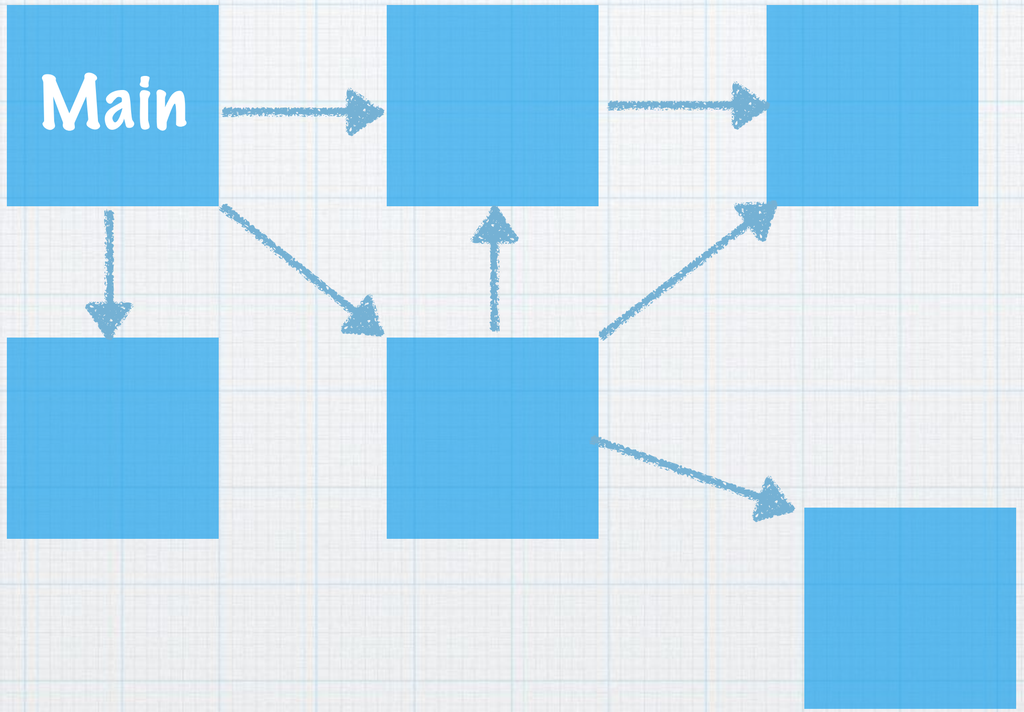
IoC

* Provides the application plumbing
* Handles code level dependencies
* Supports decoupling your objects
* Handles construction and maintenance of class instances

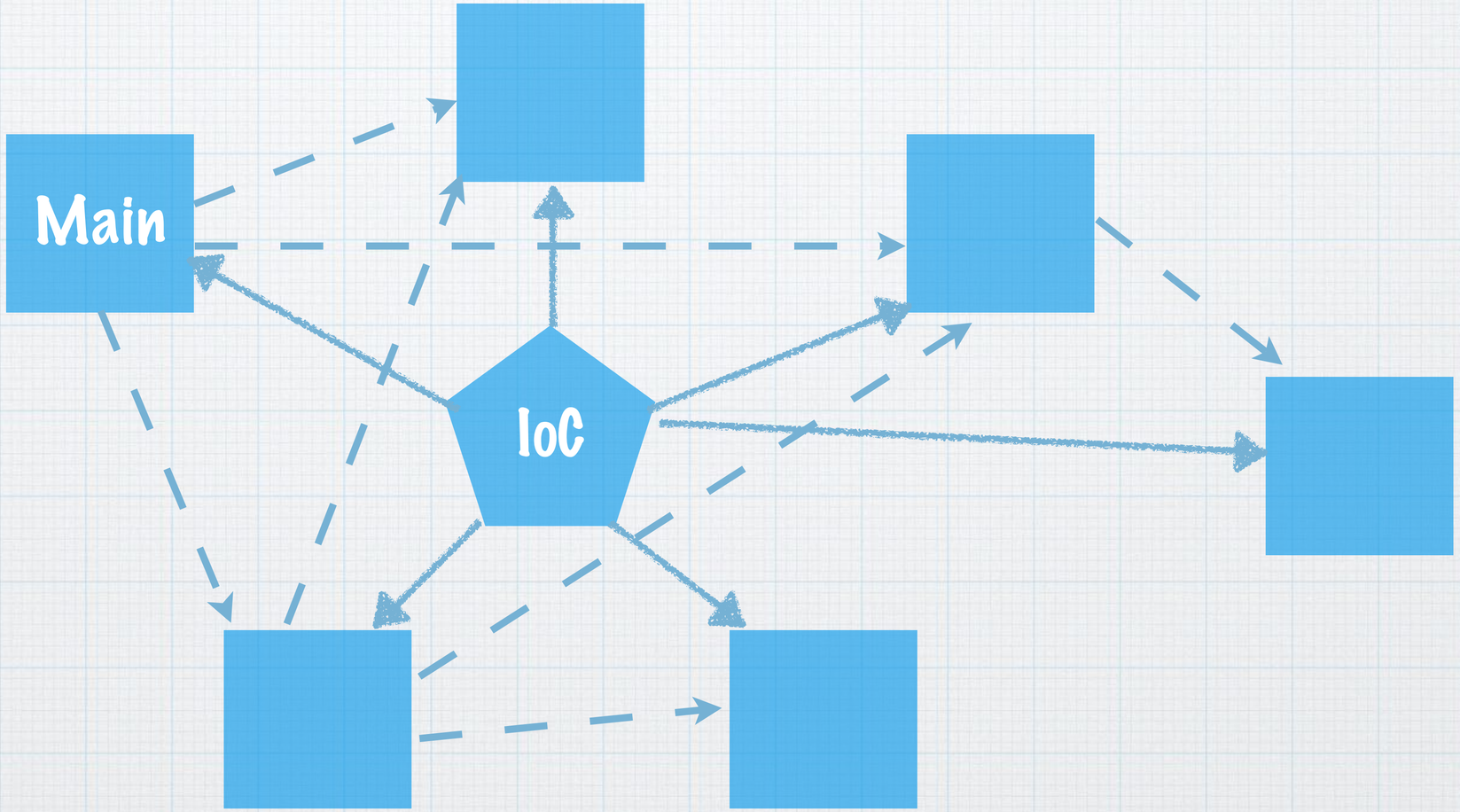
What is IoC?

* Inversion of Control -> Dependency Injection
* Objects receive their dependencies fully constructed instead of constructing them
* Container keeps a handle on objects

Traditional Dependency Mgt



IoC Dependency Mgt



Benefits of IoC

* Reduces noise in your code
* Reduces object coupling
* Reduces defects that arise from incorrect construction
* Focus on the API contract

Application Context

Bean Factory

* Bean Factory maintains beans throughout the lifecycle of the application
* You interact with the Application Context, not the bean factory
* BeanFactory is the central component of the IOC container

Purpose of the Application Context

* Application Context provides a Read Only wrapper of the bean factory
* Technically an extension of BeanFactory
* Provides metadata for bean creation
* Provides mechanism for creation beans in the correct order

AC as IoC Container

* Provides all facilities for injection of beans at startup and runtime
* Most of utilizing Spring is actually configuring the IoC container
* Application context handles all singleton beans

A Note on Mx ACs

* A Spring application can have one or more Application Contexts
* Web containers always have multiple
* Parent context can interact with children, but not the other way around

Configuring the Application Context

* Java Config
* Component Scanning
* XML config

Part 1: Application Context

* Java Configuration
* Component Scanning
* XML Configuration
* Spring Bean Lifecycle

Benefits of Java Config

* Native language syntax
* Compile time checking of configuration
* Easier IDE integration

Java Config Syntax

* Java configuration stems from a class annotated with @Configuration
* Beans are created as methods of the class where method is annotated with @Bean
* Constant beans can be defined with @Value

@Configuration

* Specifies the class is a config class

```
@Configuration
public class ApplicationConfig {
```

@Bean

* Used on methods to signify a Bean
* Method name becomes bean id

```
@Bean
public InventoryService inventoryService(){

    return new InventoryService(itemRepository(), personRepository());
}
```

Configuration Encapsulation

* Configurations can be encapsulated to separate concerns
* Very useful in larger applications & shared libraries
* @Import or @ImportResource

```
@Configuration
@Import(SecurityConfig.class)
public class ApplicationConfig {
```

Leveraging the Environment

* Spring provides an abstraction to the environment
* Load properties from env, properties files, other sources
* Inject into attribute via @Value

```
@Value("${server.port}")
private int serverPort;
```

Loading Properties Files

* You can load properties from one or more files

```
@Configuration
@PropertySource("classpath:application.properties")
public class ApplicationConfig {
```

Profiles

* Configuration can be flexed based on a property
* @Profile annotation allows you to define beans based on a specific config
* spring.profiles.active environment variable turns a specific property on
* Very useful in multi-env deployments

Part 1: Application Context

* Java Configuration
* Component Scanning
* XML Configuration
* Spring Bean Lifecycle

@Component

* Indicates that a class should be loaded into the bean factory
* @Component and stereotypes
* Component Scanning scans a base package and loads configuration automatically for each bean it finds

DI with Component Scanning

* Dependency injection is achieved through the @Autowired annotation
* @Qualifier is used when multiple implementations of an interface are needed
* Properties injected with @Value

Configuring a Component Scan

* Component scan needs a starting point
  * XML
  * Java Config
* Application Context loads from the starting point then component scans the rest

Examples

```
@ComponentScan(basePackages = {"com.frankmoley.talks.spring.intro.config"})
public class ExampleConfig {
```

```
<context:component-scan base-package="com.frankmoley.talks.spring.intro"/>
```

Autowiring

* Can be applied to class attributes -> DON'T DO THIS
* Should be applied to setters for optional or changing dependancies
* Should be applied to constructors for required dependancies -> Most common

Part 1: Application Context

* Java Configuration
* Component Scanning
* XML Configuration
* Spring Bean Lifecycle

I heard it was dead

* Not dead, but dying, sort of
* Still fully supported
* Needed for legacy support
* Limited examples

Namespaces

* If you learn anything, it should be the namespaces
* Reduce configuration needs by providing baseline constructs
* Util, JEE, Context, JDBC most common

Example Config

```
<bean id="inventoryService" class="com.frankmoley.talks.spring.intro.service.InventoryService">
    <constructor-arg name="itemRepository" ref="itemRepository"/>
    <constructor-arg name="personRepository" ref="personRepository"/>
</bean>
```

Part 1: Application Context

* Java Configuration
* Component Scanning
* XML Configuration
* Spring Bean Lifecycle

Knowledge of Lifecycle

* Increases overall knowledge of Spring
* Improves extensibility of Framework
* Aides in troubleshooting

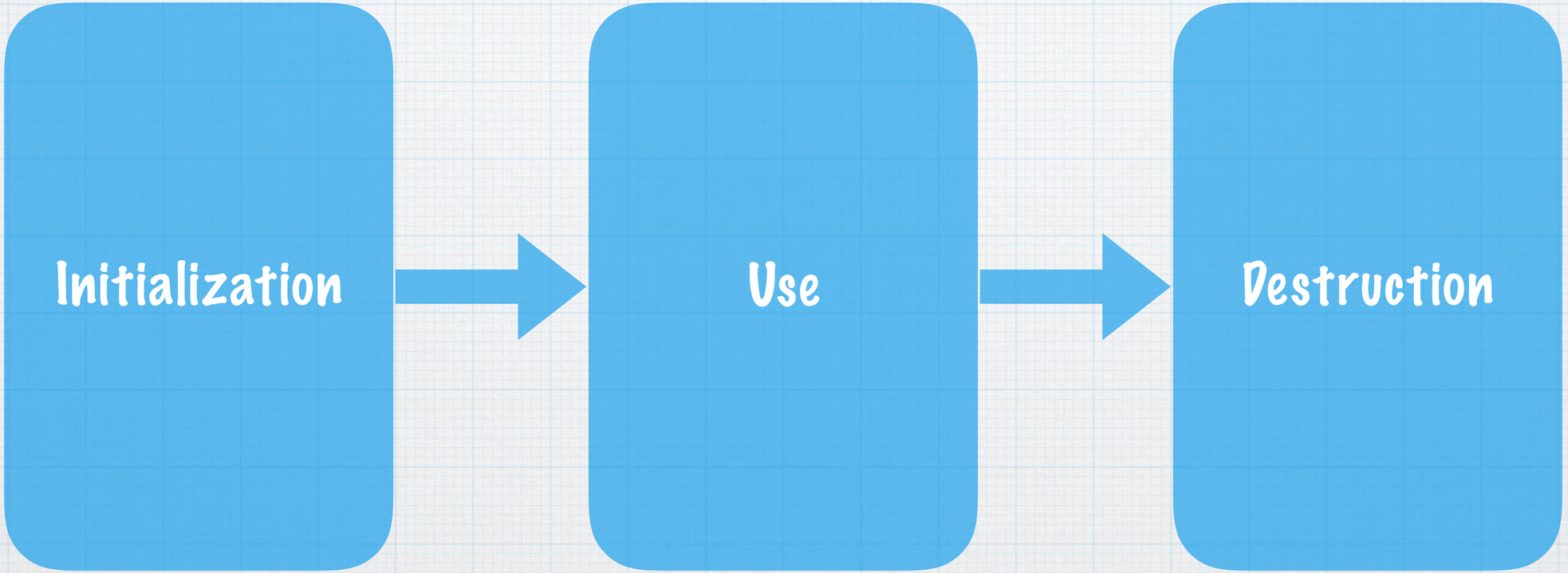
Professional Developer

* Deeper knowledge of the tools we use
* More educated discussions
* Better architectural decisions

Open Source

* Better interactions with the community
* Contributing to the framework

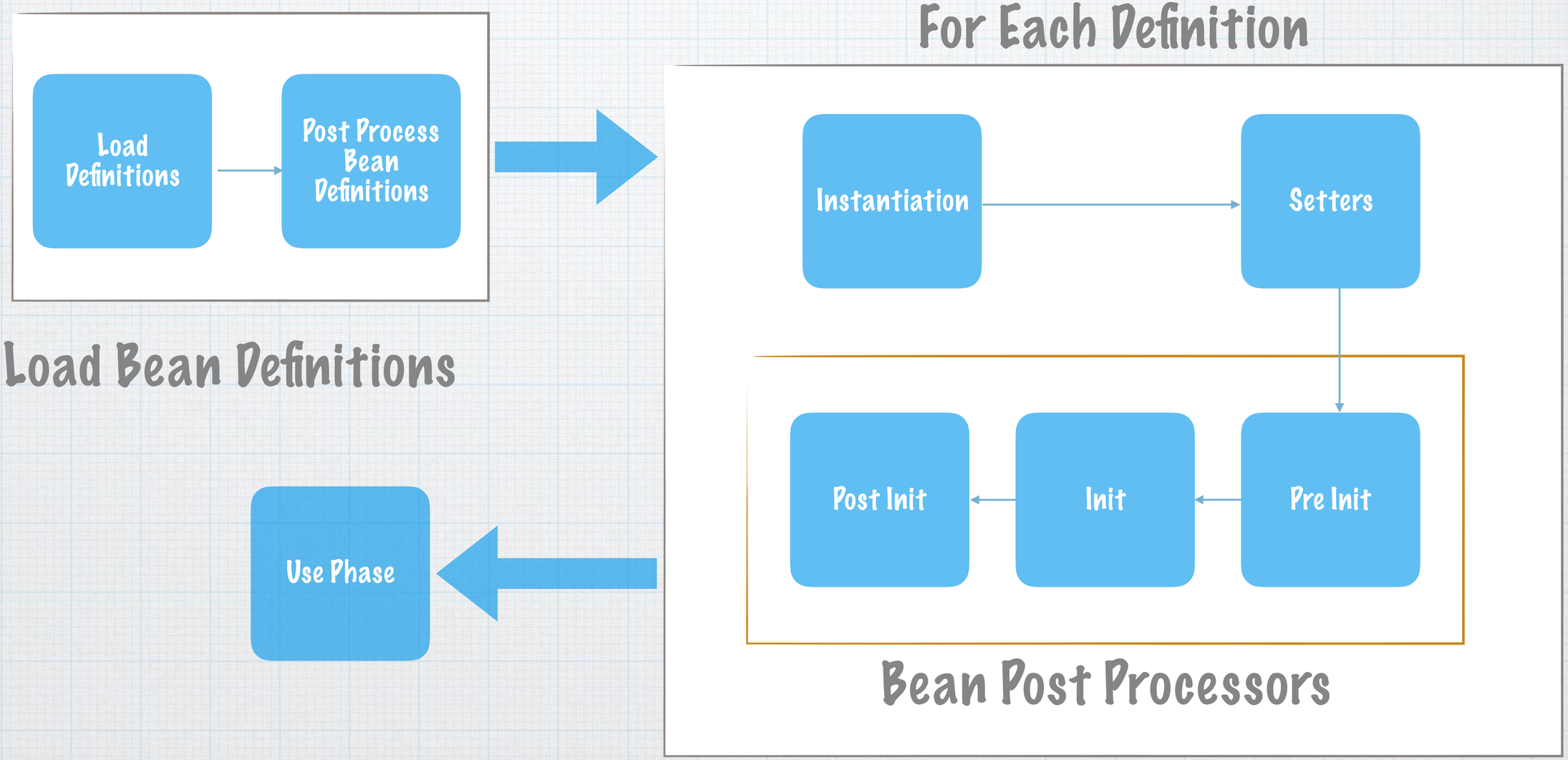
3 Main Phases



Initialization Phase Introduction

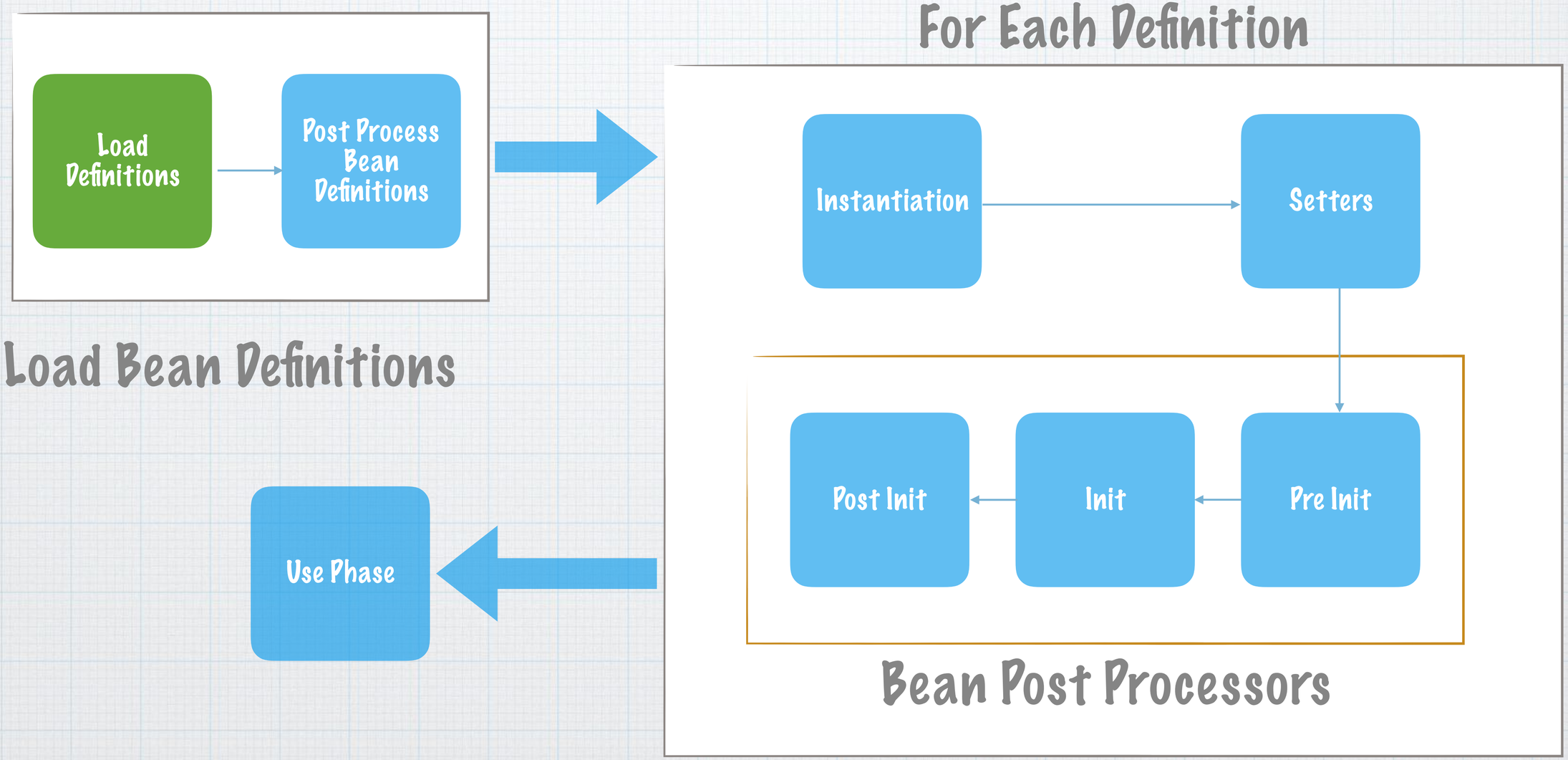
* Begins with creation of Application Context
* Bean Factory Initialization Phase
* Bean Initialization and Instantiation

Initialization Phase: Overview



Init: Load Definitions

Where we are


Load Definitions
Post Process Bean Definitions
Load Bean Definitions
For Each Definition
Instantiation
Setters
Post Init
Init
Pre Init
Bean Post Processors
Use Phase

Bean Definition Sources

* Java Config
* XML Config
* Component Scanning/ Auto config

Priming the Factory

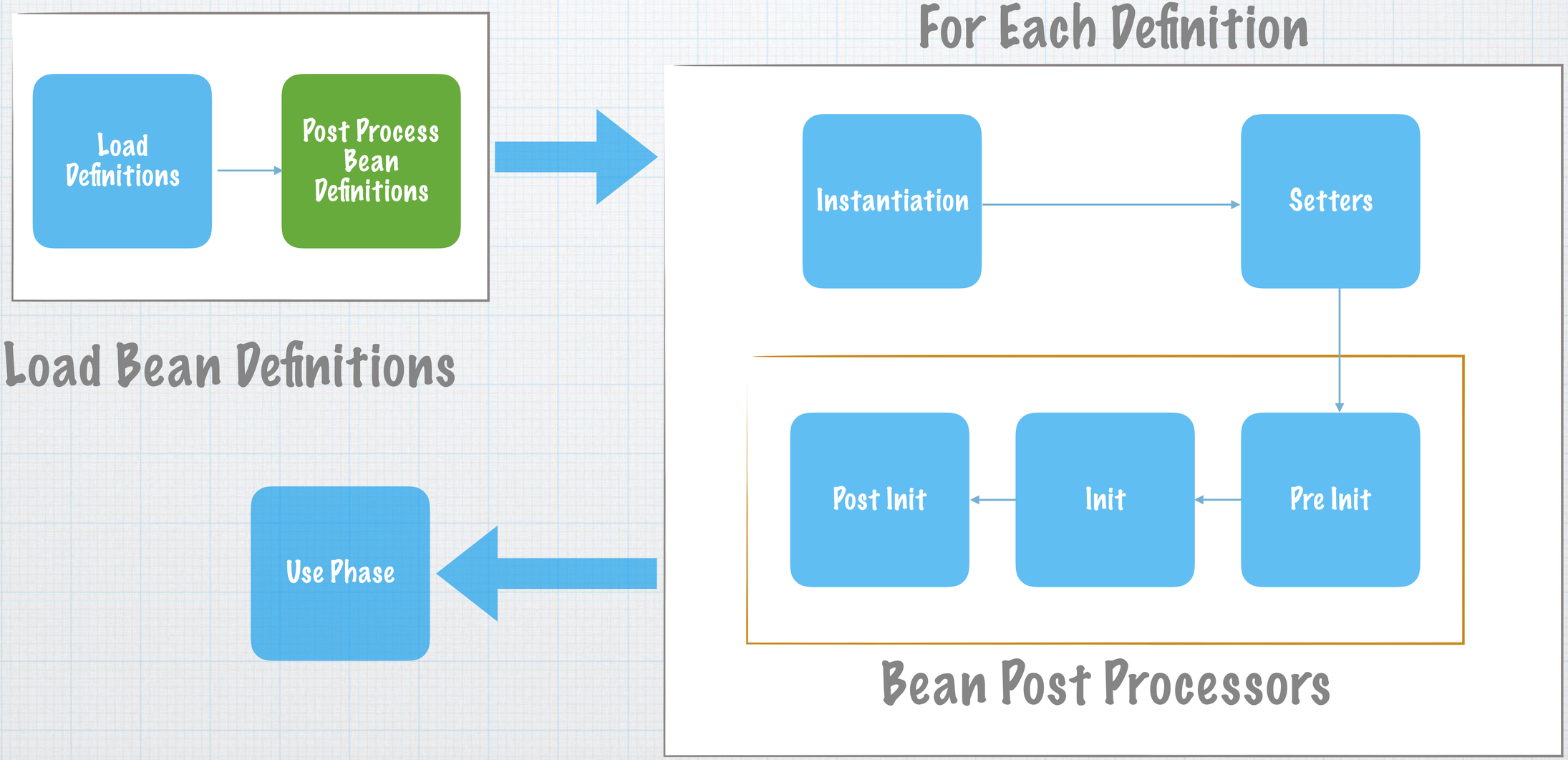
* The bean definitions are loaded into the Bean Factory from all sources
* Id is used to create the index for the factory
* Bean factory only contains references at this point

Phase Complete

* All beans have been loaded into the Bean Factory
* Beans are just references and metadata at this point
* No object instantiation of your code has occurred

Init: Post Process Bean Definitions

Where we are


Load Definitions
Post Process Bean Definitions
Load Bean Definitions
For Each Definition
Instantiation
Setters
Post Init
Init
Pre Init
Bean Post Processors
Use Phase

Bean Factory Post Processors

* Perform work on the entire bean factory
* Can modify or transform any bean in the factory prior to instantiation
* Most familiar example is the PropertySourcePlaceholderConfigurer

BeanFactoryPostProcessor Interface

* First extension point in the lifecycle
* Allows you to write custom components to impact the bean factory
* Not very common to write your own, very common to use existing ones (registering scope, properties)

A Note about Java Config

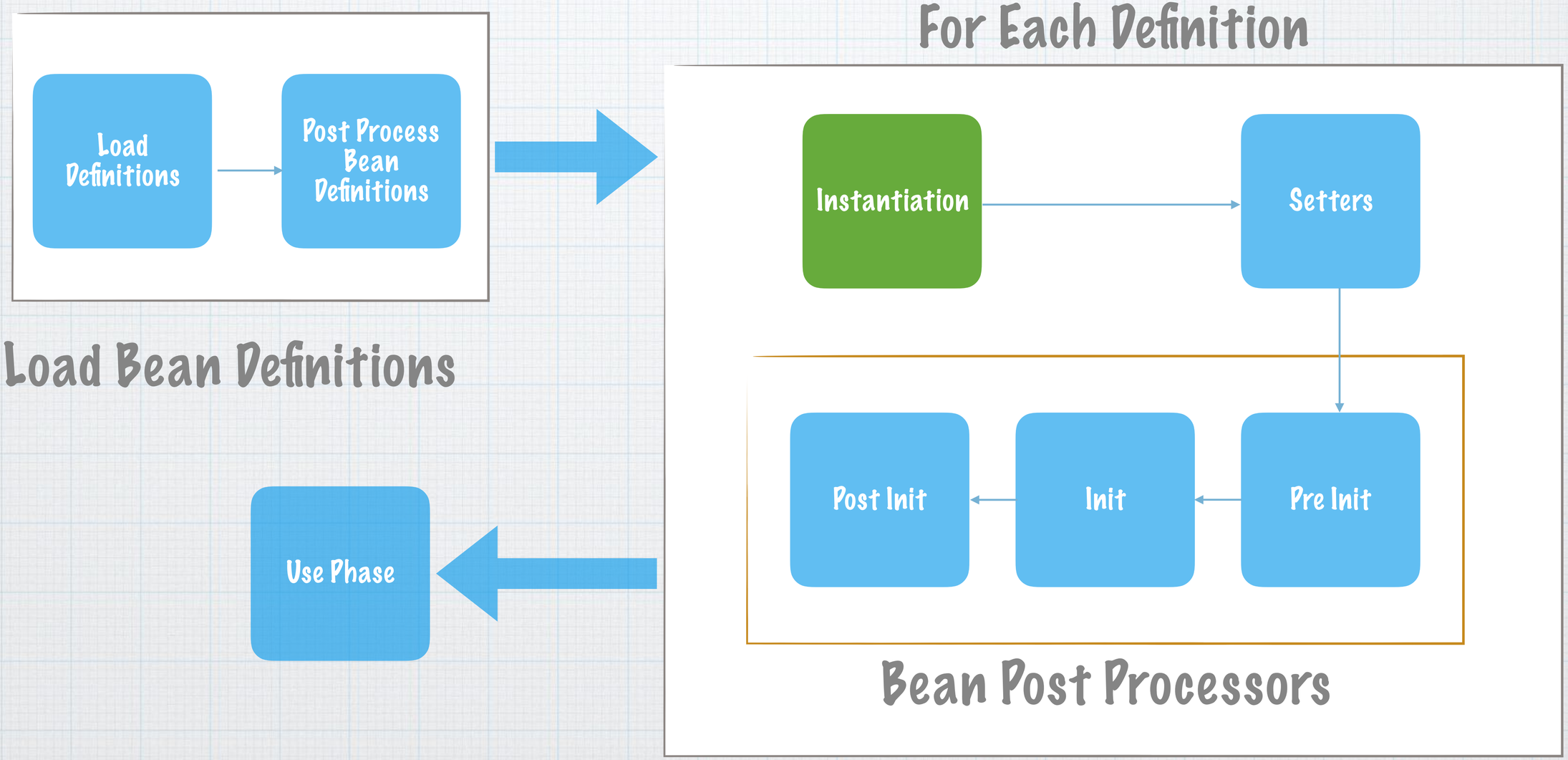
* When extending the BeanFactoryPostProcessor interface
  * Bean definitions must be static
* Removes risks of side effects of dynamic instances

Phase Complete

* Bean Factory is loaded with references
* Bean Factory and all beans in it are configured
* All "system level" work is completed in Spring at this point

Init: Instantiation

Where we are


Load Definitions
Post Process Bean Definitions
Load Bean Definitions
For Each Definition
Instantiation
Setters
Post Init
Init
Pre Init
Bean Post Processors
Use Phase

Construction

* Beans are instantiated in the factory using the constructors
* Done in the correct order to ensure dependencies are created first
* Handle to class instance remains in the bean factory for the lifecycle of the application for singletons

Eager vs Lazy

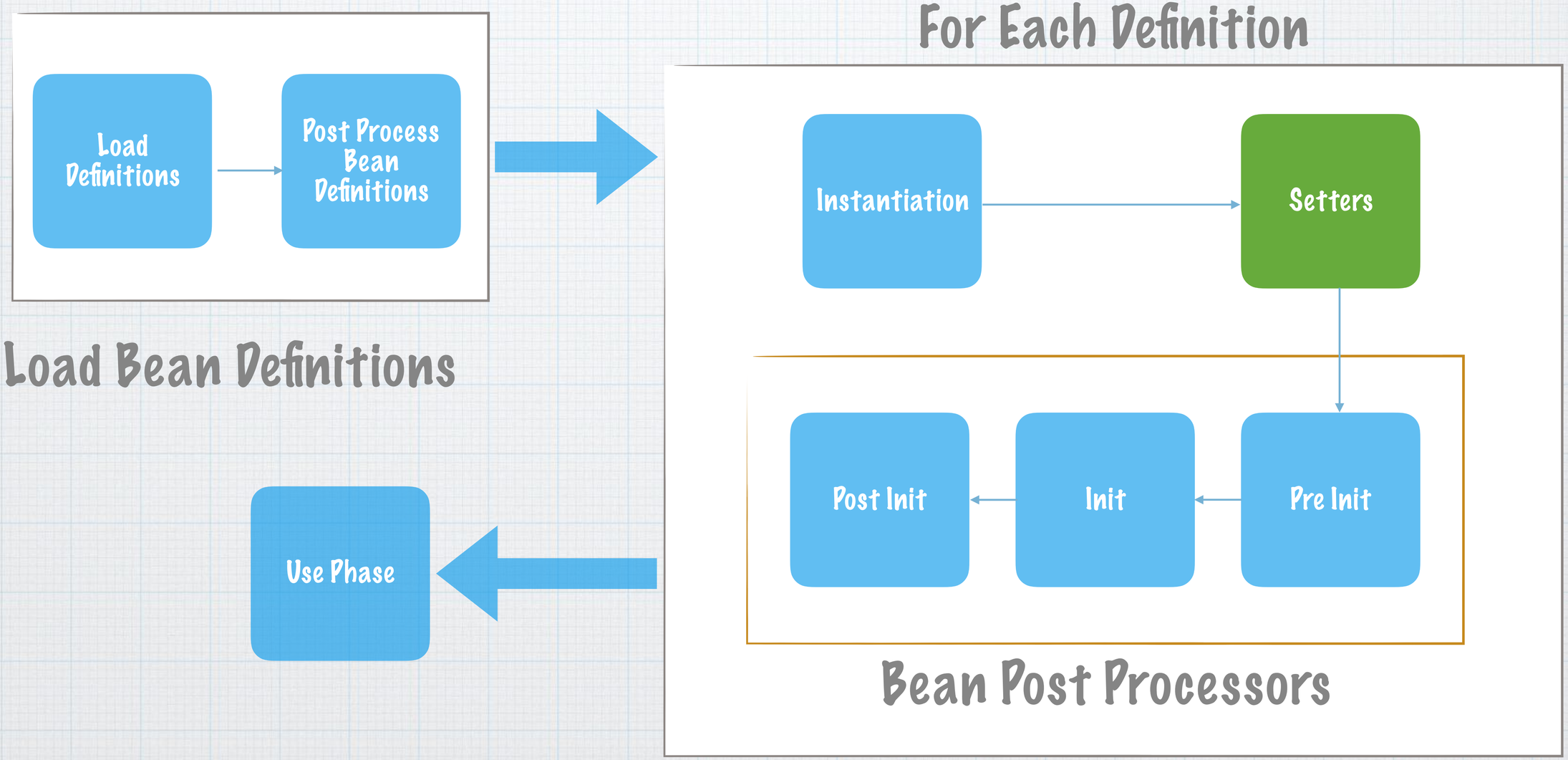
* By default all beans are instantiated eagerly
* To truly be lazy, there can be nothing that depends on them
* Lazy beans can be specified as lazy, but the Application Context reserves the right to ignore

Phase Complete

* Bean pointer is referenced in Bean Factory still
* Objects have been constructed
* Not available for use yet

Init: Setters

Where we are


Load Definitions
Post Process Bean Definitions
Load Bean Definitions
For Each Definition
Instantiation
Setters
Post Init
Init
Pre Init
Bean Post Processors
Use Phase

Post Initialization Dependency Inj

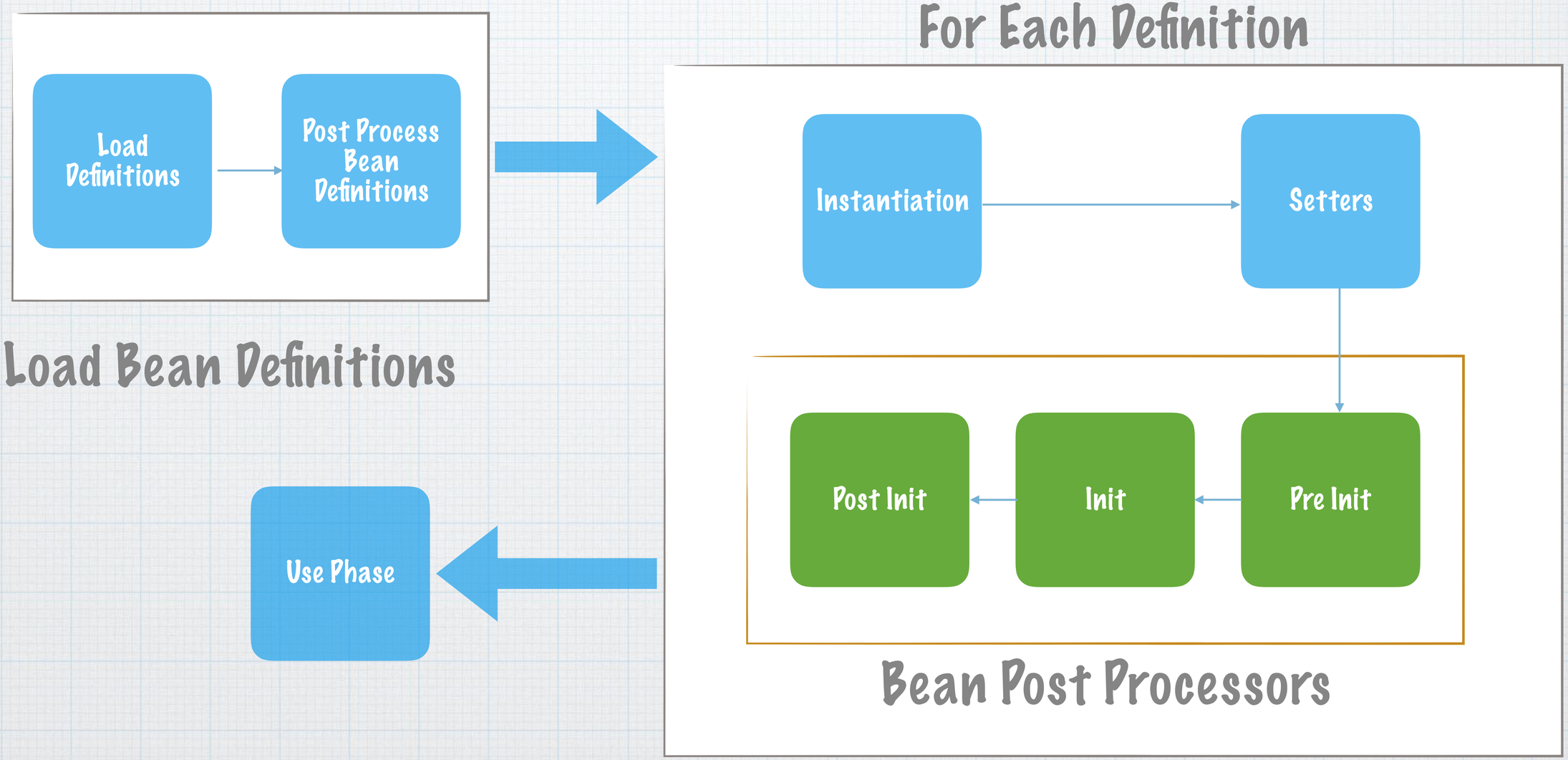
* After all beans have been instantiated
* Setters are called
* Autowiring occurs (non constructor based)
* Java Config behaves differently

Phase Complete

* Beans are fully initialized
* All initial dependencies are injected
* Beans still not ready for use

Init: Bean Post Processing

Where we are


Load Definitions
Post Process Bean Definitions
Load Bean Definitions
For Each Definition
Instantiation
Setters
Post Init
Init
Pre Init
Bean Post Processors
Use Phase

Bean Post Processing

* Final point of configuration manipulation
* Each bean may have additional behaviors added
* 2 types of extensible and generic processing, before and after initializer

Initializer

* Second Bean Post Processor Action
* Special case
* @PostConstruct methods called here
* Framework provides many of these

Lifecycle Methods

* @PostConstruct & @PreDestroy
* void methods
* @PostConstruct executes during the init phase during BPP -> Spring is available
* @PreDestroy occurs when the ApplicationContext closes

Bean Post Processor Interface

* The Bean Post Processor Interface allows you to inject common behavior to a class of beans
* Still operates on specific beans
* Before and After types
* Framework leverages lots of these

Phase Complete

* Beans have been instantiated and initialized
* Dependencies have been injected
* Beans are finally ready for use

Config Differences in Init

* Java Config merges Instantiation & Setter calling
* Component Scanning
  * Autowired constructors during Instantiation
  * Autowired Setters and Fields* during Setter
* XML no changes

Use Phase

Use Phase

* Most of the time is spent in this phase
* Application Context serves proxies to the original class
* Application Context maintains handle to each bean (singleton)

Context Aware Beans

* Spring provides interface ApplicationContextAware
* Gives your class a handle to the ApplicationContext
* Not a very common interface to use, but is available during the use phase

Destruction Phase

Destruction

* Begins when close is called on ApplicationContext
* Any @PreDestroy method is called
* Beans are not destroyed

Caveats

* Context cannot be reused again
* Prototype beans are not impacted
* Only Garbage Collection actually destroys bean instances

Exercise 1

Aspect Oriented Programming with Spring

What are Aspects?

* Aspects are reusable blocks of code that are injected into your application at runtime
* Aspects are powerful tools for adding behavior
* Solve cross cutting concerns in one place

Cross Cutting Concern?

* Evaluate business requirements and look for words like every or always
* Look for system requirements that apply to multiple business requirements
* Examples: Logging, Security, Transactions, Caching

Why Leverage Aspects?

* Code duplication/DRY
* Mixing of concerns
* Maintain application logic

Spring Based AOP

* Leverages AspectJ for aspecting
* Byte code modification (run time interweaving)
* Dynamic Proxy based

Core Concepts

* Join Point
* Pointcut
* Advice
* Aspect

The Pointcut

Pointcut

* Used to define the JoinPoint where the advice will be weaved into
* Follows a standard syntax
* In Spring, can be defined as a distinct element or embedded in the advice

Pointcut Syntax

* designator("r p.c.m(arg)")
* R is return type
* P is package, C is class, m is method
* Arg is args

Common Designators

* execution - expression for matching method execution
* within - expressions for matching within certain types
* target - expressions for matching a specific type
* @annotation - expressions for matching a specific annotation

Examples

```
@Pointcut("execution(* com.frankmoley.talks.spring.intro.service.*Service.get*(..))")
public void executeGetLogging(){}
```

```
@Pointcut("@annotation(Loggable)")
public void executeLogging(){}
```

The Advice

3* Flavors

* Before Advice
* After Advice (After Returning, After Throwing, After)
* Around Advice

Before Advice

* Executes advice operation before the JoinPoint
* Exceptions in the advice cause the JoinPoint to not be executed
* Useful for call count metrics
* Leverages the @Before annotation

Before Example

```
@Before(value="executeLogging()")
public void adviseLogging(JoinPoint joinPoint){
    //...
}
```

AfterReturning Advice

* Executes advise operation after the JoinPoint
* JoinPoint must return normally, exception flows will not execute this advice code
* Leverages the @AfterReturning annotation
* You get a handle to the return value

AfterReturning Example

```
@AfterReturning(value="executeLogging()", returning = "returnValue")
public void adviseLogging(JoinPoint joinPoint, Object returnValue){
    //...
}
```

AfterThrowing Advice

* Executes the advice code only if the JoinPoint throws an exception
* Can be filtered to a specific exception type
* Useful for alerting mechanisms when certain conditions exist
* @AfterThrowing annotation

AfterThrowing Example

```
@AfterThrowing(value="executeLogging()", throwing = "exception")
public void adviseLogging(JoinPoint joinPoint, RuntimeException exception){
    //...
}
```

After Advice

* The advice is executed regardless of exception or normal return
* No handle of return value
* @After annotation

After Example

```
@After(value="executeLogging()")
public void adviseLogging(JoinPoint joinPoint){
    //...
}
```

Around Advice

* Most useful in many cases
* You control the execution before the JoinPoint and After the JoinPoint
* Must return an Object to propagate up the call stack
* @Around annotation

Around Example

```
@Around(value="executeLogging()")
public Object adviseLogging(ProceedingJoinPoint joinPoint) {
    //... Before
    Object returnValue = null;
    try {
        returnValue = joinPoint.proceed();
        //... After
    } catch (Throwable e) {
        //... handle any exceptions
    }
    return returnValue;
}
```

Exercise 2

Data Access with Spring

A Note about Exceptions

* All abstractions of Spring Data leverage RuntimeExceptions
* Not working with Runtime? Be sure to extend as needed

Part 3: Data Access with Spring

* Spring JDBC
* Transaction Management
* Spring Data
* Spring Data JPA

Spring JDBC

* Most rudimentary version of data access using Spring JDBC
* JdbcTemplate handles the connection open, commit, rollback, and closing
* User controls queries **I like this part**
* Also handles ResultSet mapping to objects

Benefits

* Less error prone (copy/paste) than traditional JDBC code
* Cleaner exception handling
* Very powerful when using highly optimized queries
* Team performance!

The JdbcTemplate

* SOAPBOX TIME: Use the NamedParameterJdbcTemplate
* Bind variables on queries
* Template pattern is common throughout Spring
* Thread safe after construction -> i.e. use a single instance

The Entity

```
public class User {

    private long id;
    private String username;
    private String name;
```

The RowMapper

```
public class UserRowMapper implements RowMapper<User> {

    @Override
    public User mapRow(ResultSet resultSet, int rowNum) throws SQLException {
        User user = new User();
        user.setId(resultSet.getLong( columnLabel: "USER_ID"));
        user.setName(resultSet.getString( columnLabel: "NAME"));
        user.setUsername(resultSet.getString( columnLabel: "USERNAME"));
        return user;
    }
}
```

The Query

```
private static final String GET_USER_QUERRY =
        "SELECT USER_ID, USERNAME, NAME FROM USER WHERE USER_ID = :userId";
```

```
public User getUser(long userId){
    return this.jdbcTemplate.queryForObject(GET_USER_QUERRY,
            Collections.singletonMap("userId", userId),userRowMapper);
}
```

Part 3: Data Access with Spring

* Spring JDBC
* Transaction Management
* Spring Data
* Spring Data JPA

ACID

* Spring provides a Transaction Management System
* Provides
  * Atomic
  * Consistent
  * Isolated
  * Durable

Boundaries

* Seldom should a repository method be wrapped
* Need to look at true units of work
* Note that transactions are proxies -> no private methods

@Transactional

* @EnableTransactionManagement on configuration class
* Use @Transaction on method (or class)
* timeout settings can be leveraged
* Isolation levels

Isolation Levels

* READ_UNCOMMITTED
* READ_COMMITTED
* REPEATABLE_READ
* SERIALIZABLE

Propagation of Tx

* Mandatory
* Nested
* Never
* Not Supported
* Required
* Requires_New
* Supported

Rollback

* By default, rollbacks occur if RuntimeException is thrown
* Can modify to rollback any specific exception
* Can modify to not rollback for specific exception

Part 3: Data Access with Spring

* Spring JDBC
* Transaction Management
* Spring Data
* Spring Data JPA

Spring Data?

* Provides common interface for most data access technologies
* Allows the developer to swap out data sources with limited code changes
* Focus on the business logic, not on the data access technology

Entity

* Domain object
* Often maps to datasource element (table, document, region)
* Often translated to aggregate objects for serving to higher level code

Repository

* Core object of Spring Data is the Repository
* Interface based on the Repository design pattern
  * Get a piece of data, come back and get the next piece
  * No massive join/aggregation paths in complex DAO methods

Extending a Repository

* You can use standard language to add dynamic functionality to your repository
* Uses reflection based on bean notation to create queries
* Supported syntax is based on datastore technology, but all follows same syntax

Repository Language

* Convention based language
* findBy{attribute} i.e. findByUsername
* Can include multiple operations findByUsernameOrName
* Can include orderBy, distinct, first, last, etc
    * findFirst10UsersByUsernameIgnoreCaseOrderByNameDesc

Table 2.3. Supported keywords inside method names

Keyword	Sample	JPQL snippet
`And`	`findByLastnameAndFirstname`	`… where x.lastname = ?1 and x.firstname = ?2`
`Or`	`findByLastnameOrFirstname`	`… where x.lastname = ?1 or x.firstname = ?2`
`Between`	`findByStartDateBetween`	`… where x.startDate between 1? and ?2`
`LessThan`	`findByAgeLessThan`	`… where x.age < ?1`
`GreaterThan`	`findByAgeGreaterThan`	`… where x.age > ?1`
`After`	`findByStartDateAfter`	`… where x.startDate > ?1`
`Before`	`findByStartDateBefore`	`… where x.startDate < ?1`
`IsNull`	`findByAgeIsNull`	`… where x.age is null`
`IsNotNull,NotNull`	`findByAge(Is)NotNull`	`… where x.age not null`
`Like`	`findByFirstnameLike`	`… where x.firstname like ?1`
`NotLike`	`findByFirstnameNotLike`	`… where x.firstname not like ?1`
`StartingWith`	`findByFirstnameStartingWith`	`… where x.firstname like ?1` (parameter bound with appended `%`)
`EndingWith`	`findByFirstnameEndingWith`	`… where x.firstname like ?1` (parameter bound with prepended `%`)
`Containing`	`findByFirstnameContaining`	`… where x.firstname like ?1` (parameter bound wrapped in `%`)
`OrderBy`	`findByAgeOrderByLastnameDesc`	`… where x.age = ?1 order by x.lastname desc`
`Not`	`findByLastnameNot`	`… where x.lastname <> ?1`
`In`	`findByAgeIn(Collection<Age> ages)`	`… where x.age in ?1`
`NotIn`	`findByAgeNotIn(Collection<Age> age)`	`… where x.age not in ?1`
`True`	`findByActiveTrue()`	`… where x.active = true`
`False`	`findByActiveFalse()`	`… where x.active = false`

https://docs.spring.io/spring-data/jpa/docs/1.4.3.RELEASE/reference/html/jpa.repositories.html

Part 3: Data Access with Spring

* Spring JDBC
* Transaction Management
* Spring Data
* Spring Data JPA

ORM Abstractions

* Builds off of the JPA model to handle all aspects of EntityManager
* Transactional boundaries still managed in your code (or preferred through transactional proxies)
* No need to interact with Hibernate or any other JPA implementation, Spring handles the dependencies

First Class Entity Support

* Support of relationship models: @OneToOne, @OneToMany etc
* EntityManager injection through Spring allows for the container to manage the EntityManager as well as associated caches
* Full transaction support through EntityManager and Transactional Proxies

CrudRepository

* The Crud Repository is the primary "implementation" of the Repository Pattern in Spring Data - it is an interface that Spring proxies with an implementation
* Extends the Repository marker interface
* Provides the expected operations: save, findOne, delete. Also supports findAll, exists, save (in batch mode), count, and delete (in batch mode)

PagingAndSortingRepository

* Extension of CrudRepository adds the ability to sort or page results from the findAll method
* Very useful in Restful Repository based micro services among other locations
* Not available in all databases -> requires a cursor

Entity Example

```java
@Entity
public class User {

    @Id
    private long id;
    private String username;
    private String name;
```

Repository Example

```java
@Repository
public interface UserRepository extends CrudRepository<User, Long>{

    User findByUsername(String username);
    User findByNameIgnoreCase(String name);
    List<User> findFirst10ByUsernameLike(String username);

}
```

Exercise 3

Web Applications with Spring

Part 4: Web Applications with Spring

* The Web Container with Spring
* MVC for Dynamic Web Pages
* MVC for ReSTful Services
* Consuming ReSTful Services with Spring