Course Introduction

Main Objective

Learn how to build Java applications with

Spring and Hibernate

Course Benefits

Nearly every job posting asks for skills in Spring and Hibernate!

Course Road Map

- Getting Started with Spring and Hibernate: Dev Environment
- Apply Spring Inversion of Control / Dependency Injection
- Perform object/relational mapping with Hibernate
- Leverage the Hibernate API to develop CRUD apps
- and much more …

Source Code

All source code is available for download

Questions / Help

If you have questions or need help…

Post the question in the classroom discussion forum

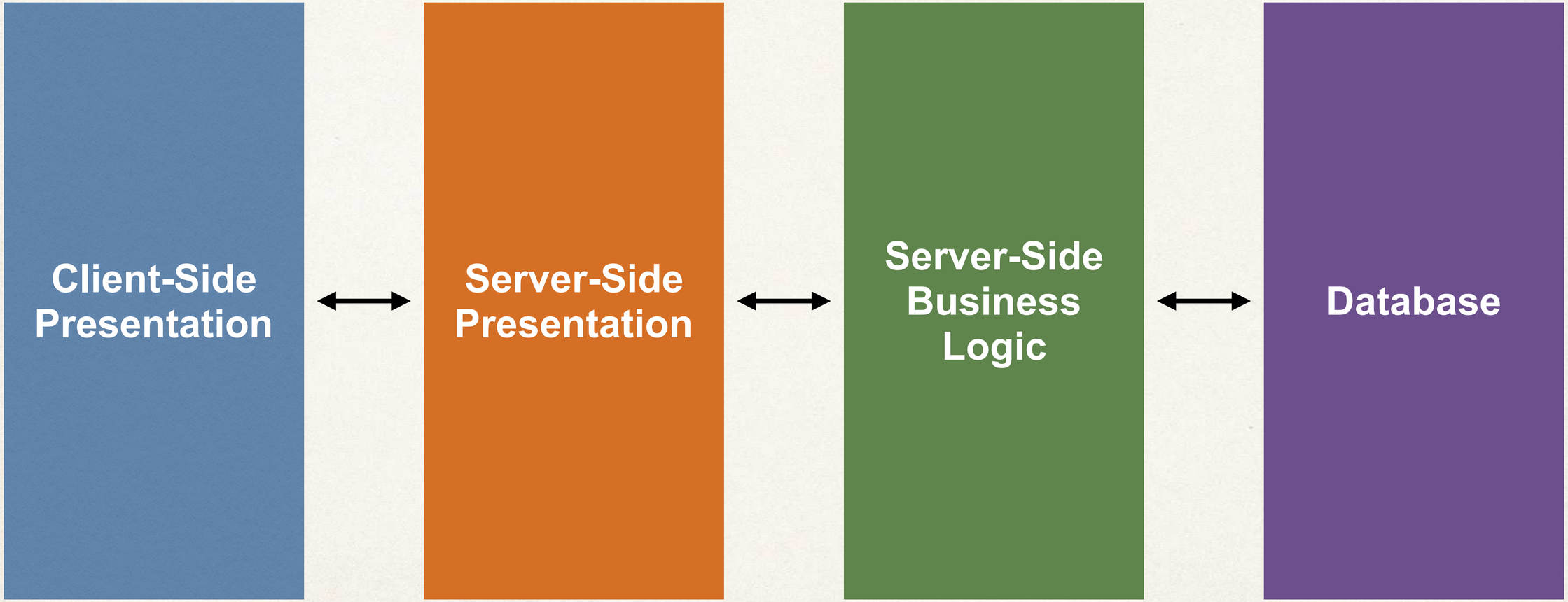
Course Introduction

Why Spring?

Spring in a Nutshell

- Very popular framework for building Java applications
- Initially a simpler and lightweight alternative to J2EE
- Provides a large number of helper classes … makes things easier

But What About J2EE???



But What About Java EE???

J2EE 1.2	J2EE 1.3	J2EE 1.4	Java EE 5	Java EE 6	Java EE 7
Servlets JSP EJB JMS RMI	EJB CMP JCA	Web Services Mgmt Deployment	Ease of use EJB 3 JPA JSF JAXB JAX-WS	Pruning, Ease of use JAX-RS CDI Bean- Validation	JMS 2 Batch TX Concurrenc y Web Sockets
1999	**2001**	**2003**	**2006**	**2009**	**2013**

EJB v1 and v2 - Complexity

- Early version of EJB (v1 and v2) were extremely complex!!!
- Multiple deployment descriptors
- Multiple interfaces
- Poor Performance of Entity Beans

EJB Client

Home Interface
Home Object

Component Interface
EJB Object

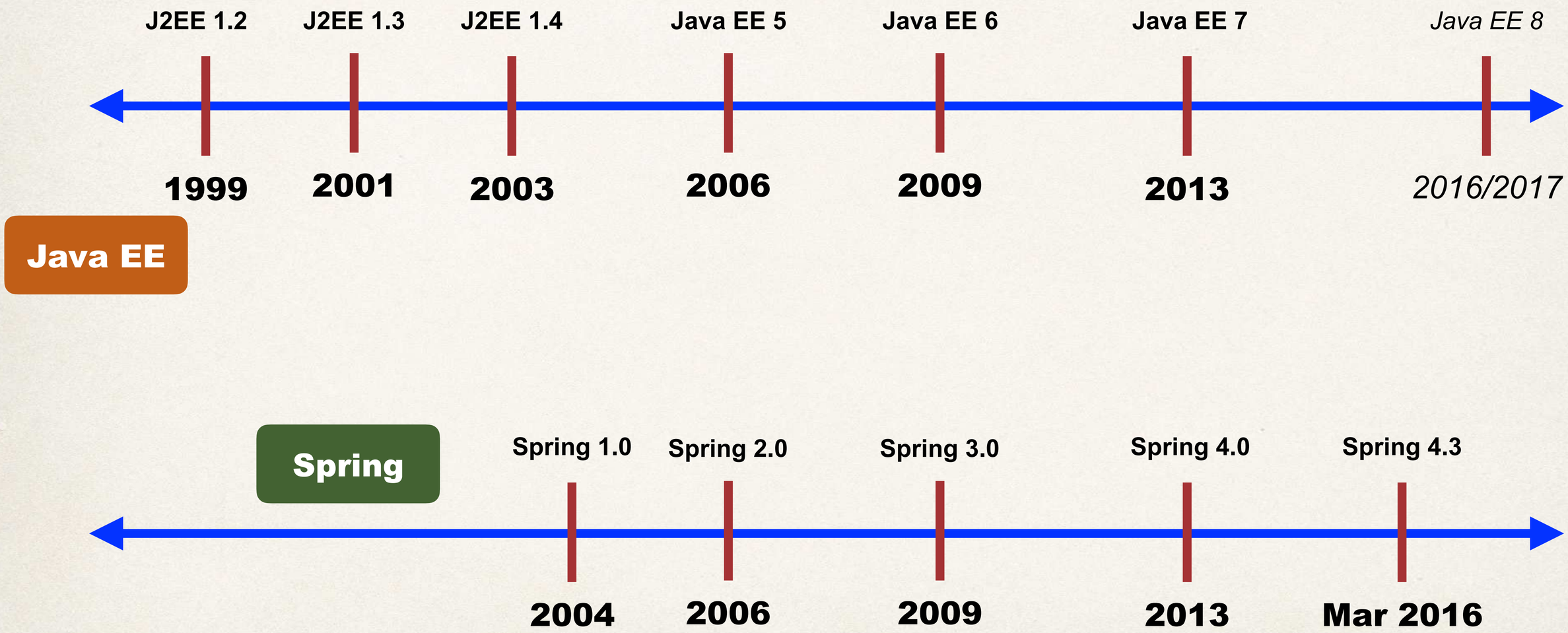
Bean Class

J2EE Development without EJB

- Rod Johnson
  - Founder of Spring

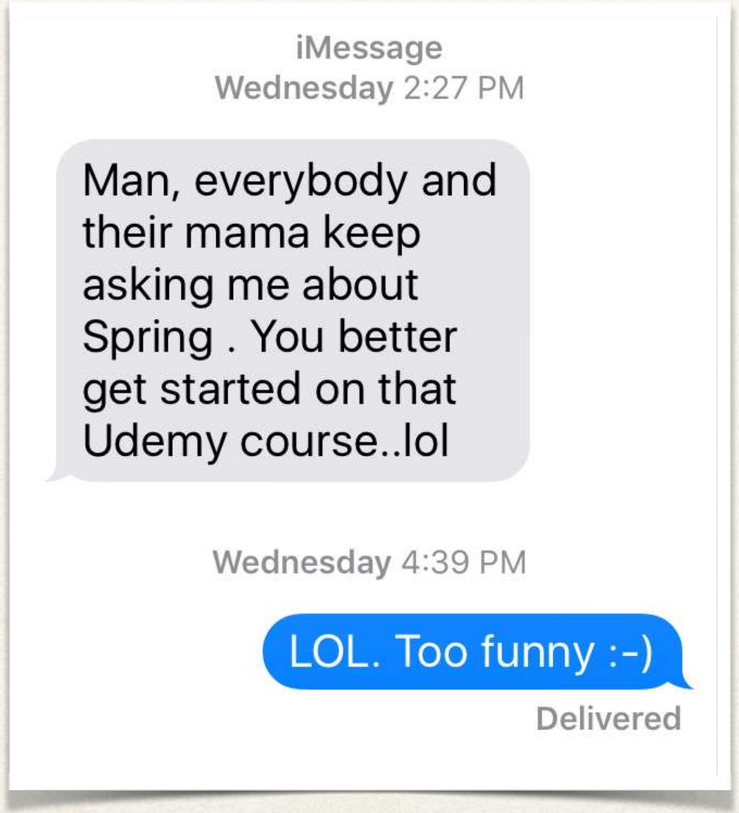
- Book: *J2EE Development without EJB*, Wrox Press
- Book: *Java Development with the Spring Framework*, Wrox Press

Release Timeline



Spring is Still in Demand

- Most sought-after Java skill set by employers and recruiters



What Should You Do?

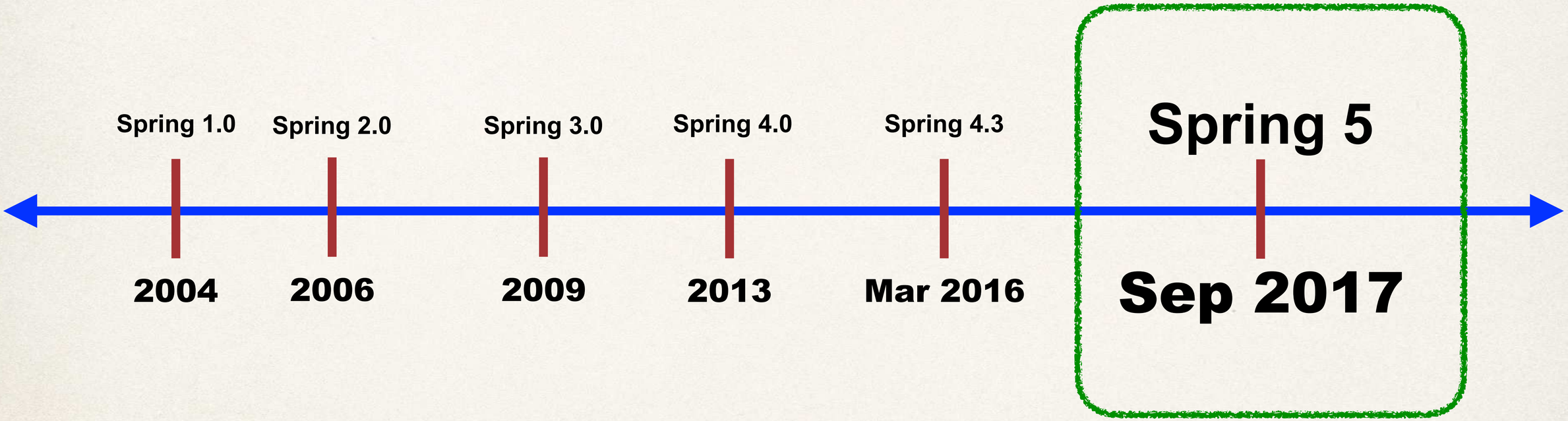
- Spring
- Java EE

Why Spring?

luv2code

UPDATE - Spring 5

Spring Release Timeline - UPDATED



What's New in Spring 5

- Updated minimum requirements for Java 8 or higher
- Deprecated legacy integration for: Tiles, Velocity, Portlet, Guava etc
- Upgraded Spring MVC to use new versions of Servlet API 4.0
- Added new reactive programming framework: Spring WebFlux

View Spring 5 Release Notes

- Spring 5 Release Notes

www.luv2code.com/spring-5-whats-new

- Spring 5 FAQ

www.luv2code.com/spring-5-faq

Spring 5 and this course

- All of the code examples in this course will work on Spring 5
- New videos will cover new Spring 5 specific features

What about Spring 4 ???

- Some of you may have projects at your job that are still using **Spring 4**
- No worries, you can still benefit from this course :-)
- **Majority of the videos in this course will run on Spring 4**
- Spring 5 just added new features …
- But **core Spring 4 features are still the same :-)**

Inversion of Control
Dependency Injection
Spring AOP
Spring MVC

Spring 5 Specific Features

- However, videos that cover new Spring 5 specific features will not work with Spring 4 (as expected … right)

- For example, Spring 5 WebFlux etc …

- For those videos, I will give a special label:

Spring 5 Only

luv2code

UPDATE - Spring 5

Spring Framework Overview

Spring Website - Official

www.spring.io

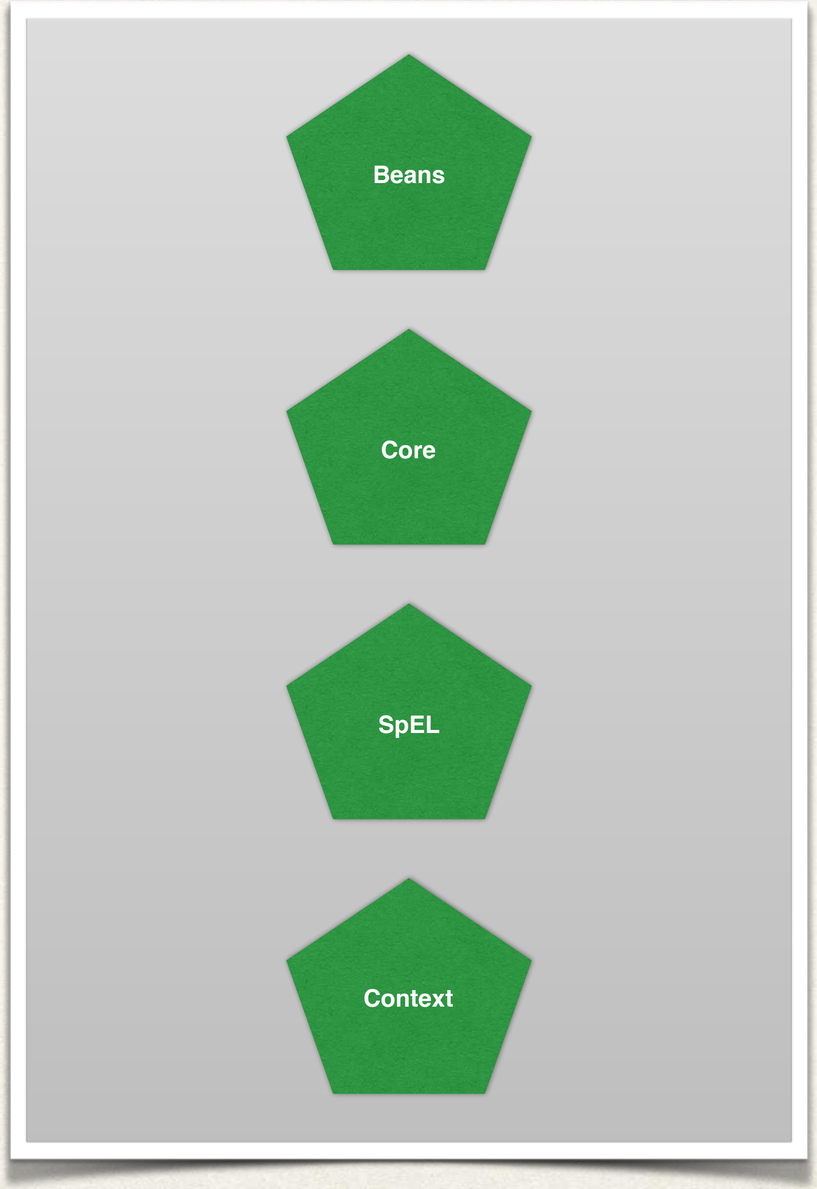
Why Spring?

Simplify Java Enterprise Development

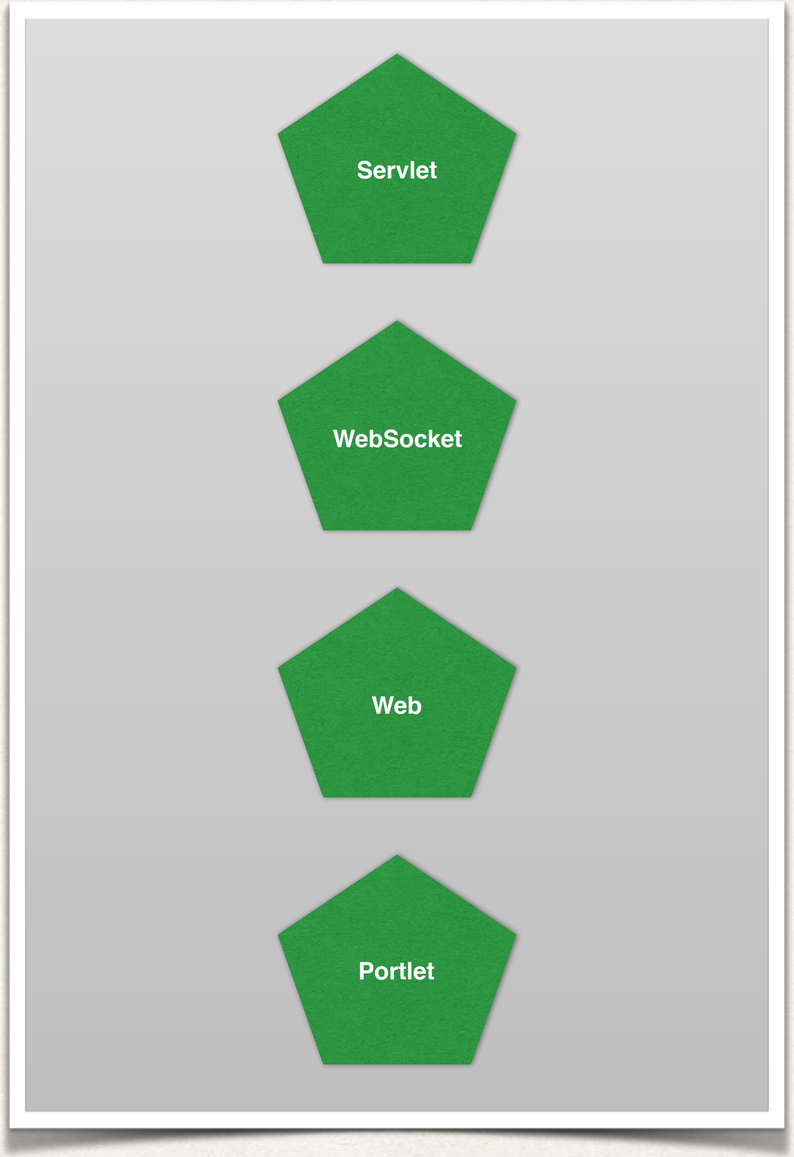
Goals of Spring

- Lightweight development with Java POJOs (Plain-Old-Java-Objects)
- Dependency injection to promote loose coupling
- Declarative programming with Aspect-Oriented-Programming (AOP)
- Minimize boilerplate Java code

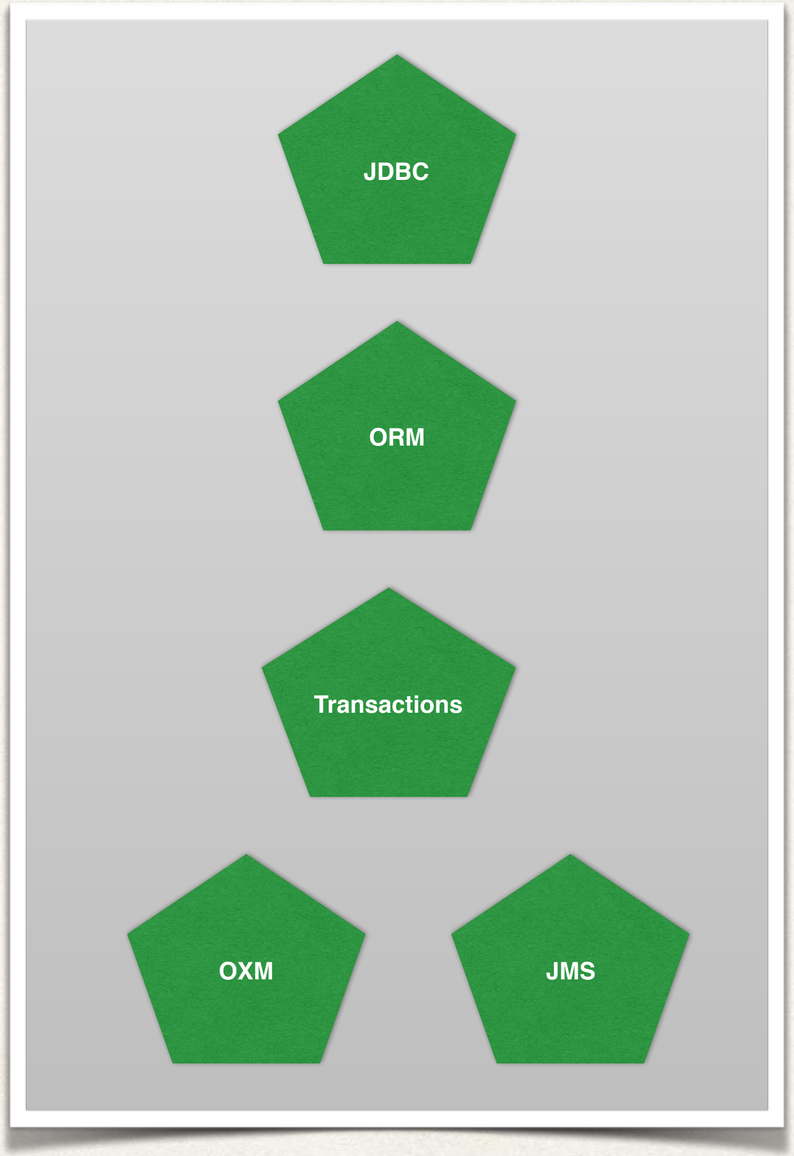
Core Container



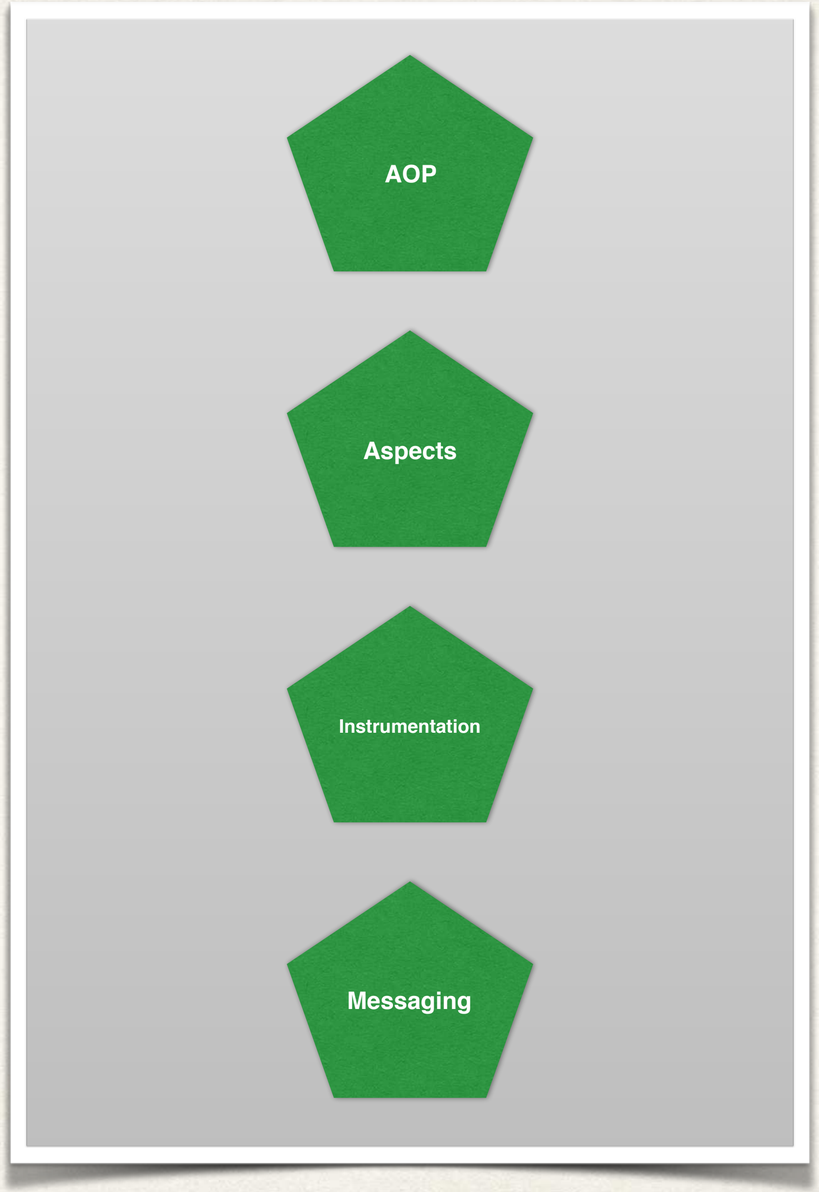
Web Layer



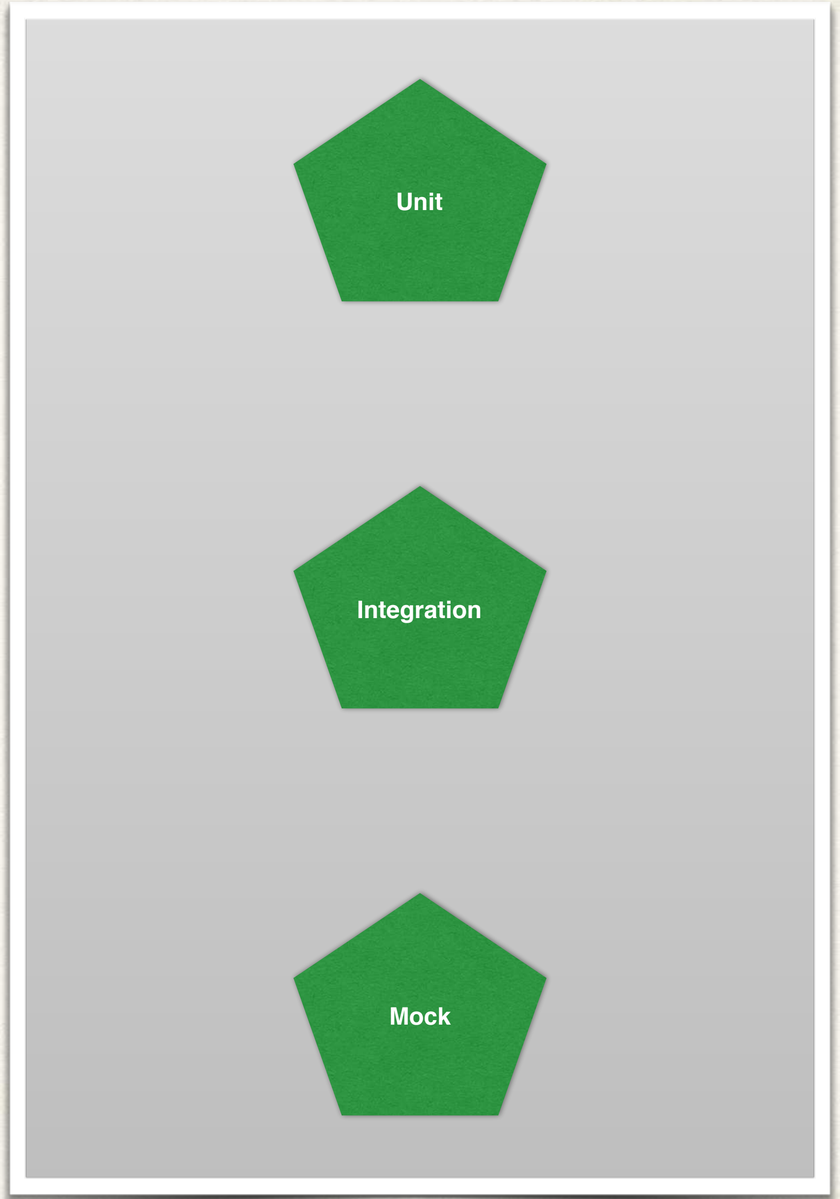
Data Access Layer



Infrastructure



Test Layer



Spring Framework Overview

Spring Projects

What Are Spring "Projects"

- Additional Spring ***modules*** built-on top of the core Spring Framework
- Only use what you need …
  - Spring Cloud, Spring Data
  - Spring Batch, Spring Security
  - Spring for Android, Spring Web Flow
  - Spring Web Services, Spring LDAP

Spring Projects

SET UP YOUR ENVIRONMENT

Topics

- Software
- High-Level Steps

Must Have Java Development Kit (JDK)

You Must Have the Java Development Kit (JDK) Installed

1. Check out my YouTube video for this:

   http://www.luv2code.com/install-java

Software

1. Java Application Server
   - For Spring MVC web development

2. Java Integrated Development Environment (IDE)

Video Break Up

1. Java Application Server - **Tomcat**

2. Java Integrated Development Environment (IDE) - **Eclipse**

SET UP YOUR ENVIRONMENT

INSTALL TOMCAT

MS WINDOWS

INSTALL ECLIPSE

MS WINDOWS

Connecting Eclipse and Tomcat

Connecting Eclipse and Tomcat

Benefits

- Start Tomcat from Eclipse
- Easily deploy applications directly to Tomcat

Downloading Spring JAR files

To Do List

1. Create Eclipse Project

2. Download Spring JAR Files

3. Add JAR files to Eclipse Project … *Build Path*

What about Maven ???

- Instead of manually downloading you can use a tool like Maven
  - If you are a Maven expert … go ahead and get started :-)
- I want to keep the focus on Spring for now
- Will discuss Maven at the end of the course

KEEP
CALM
and
FOCUS YOUR
GOALS!

Downloading Spring JAR files

Inversion of Control

Inversion of Control (IoC)

The approach of outsourcing the construction and management of objects.

Coding Scenario

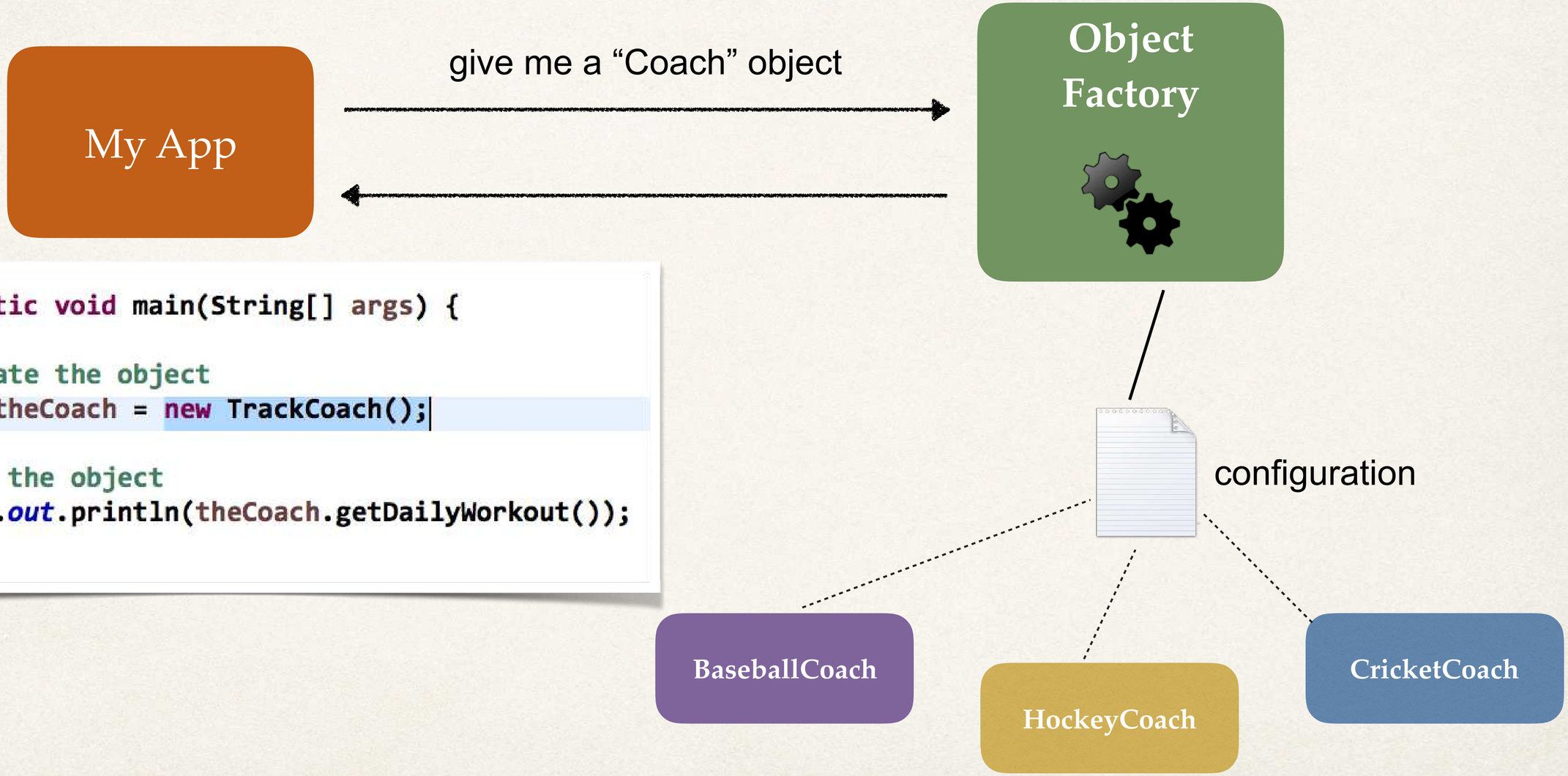
- App should be configurable

- Easily change the coach for another sport
  - Hockey, Cricket, Tennis, Gymnastics etc …

Code Demo

- **MyApp.java**: main method
- **BaseballCoach.java**
- **Coach.java**: interface after refactoring
- **TrackCoach.java**

Ideal Solution

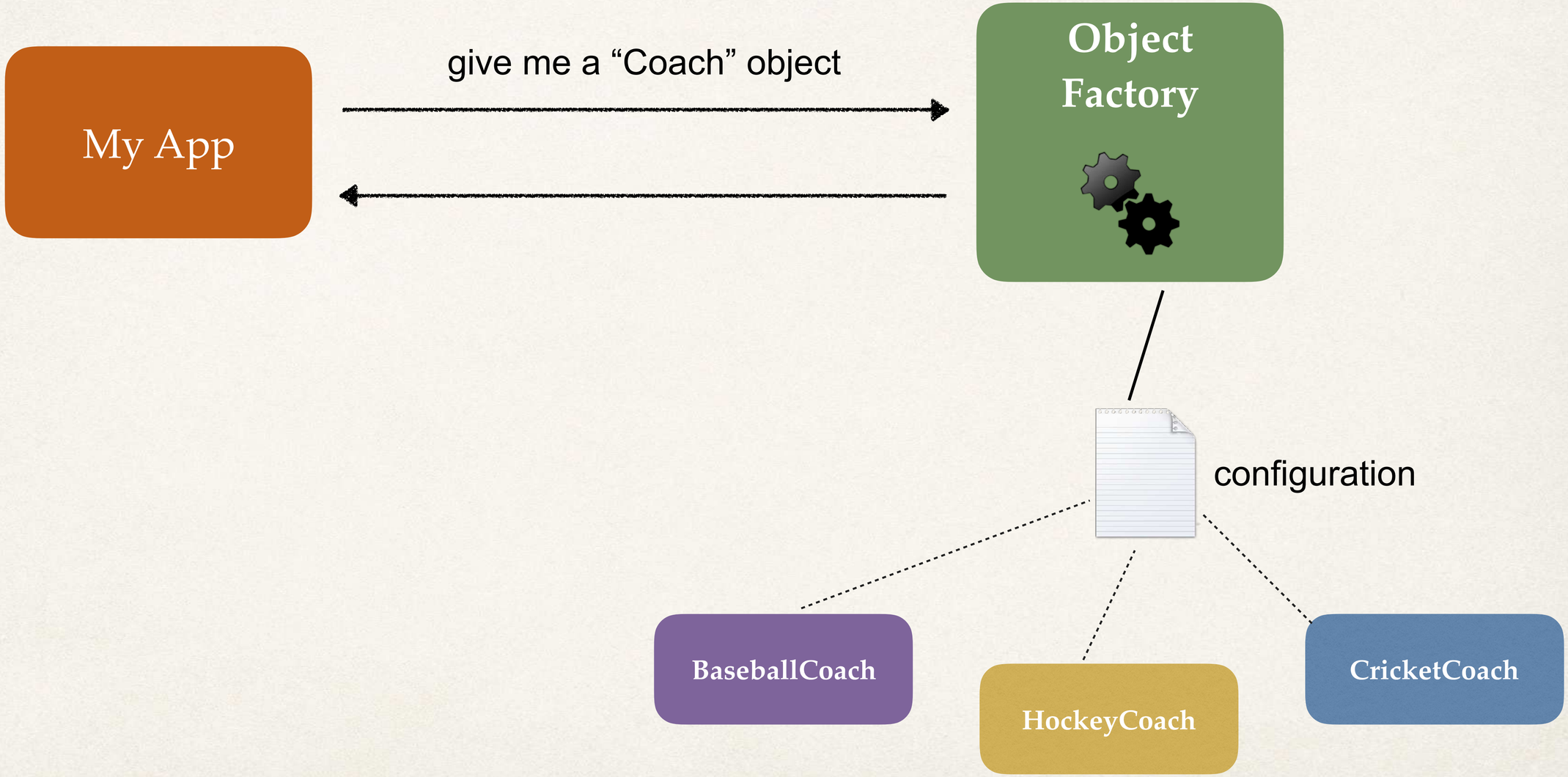



```
public static void main(String[] args) {

    // create the object
    Coach theCoach = new TrackCoach();

    // use the object
    System.out.println(theCoach.getDailyWorkout());
}
```

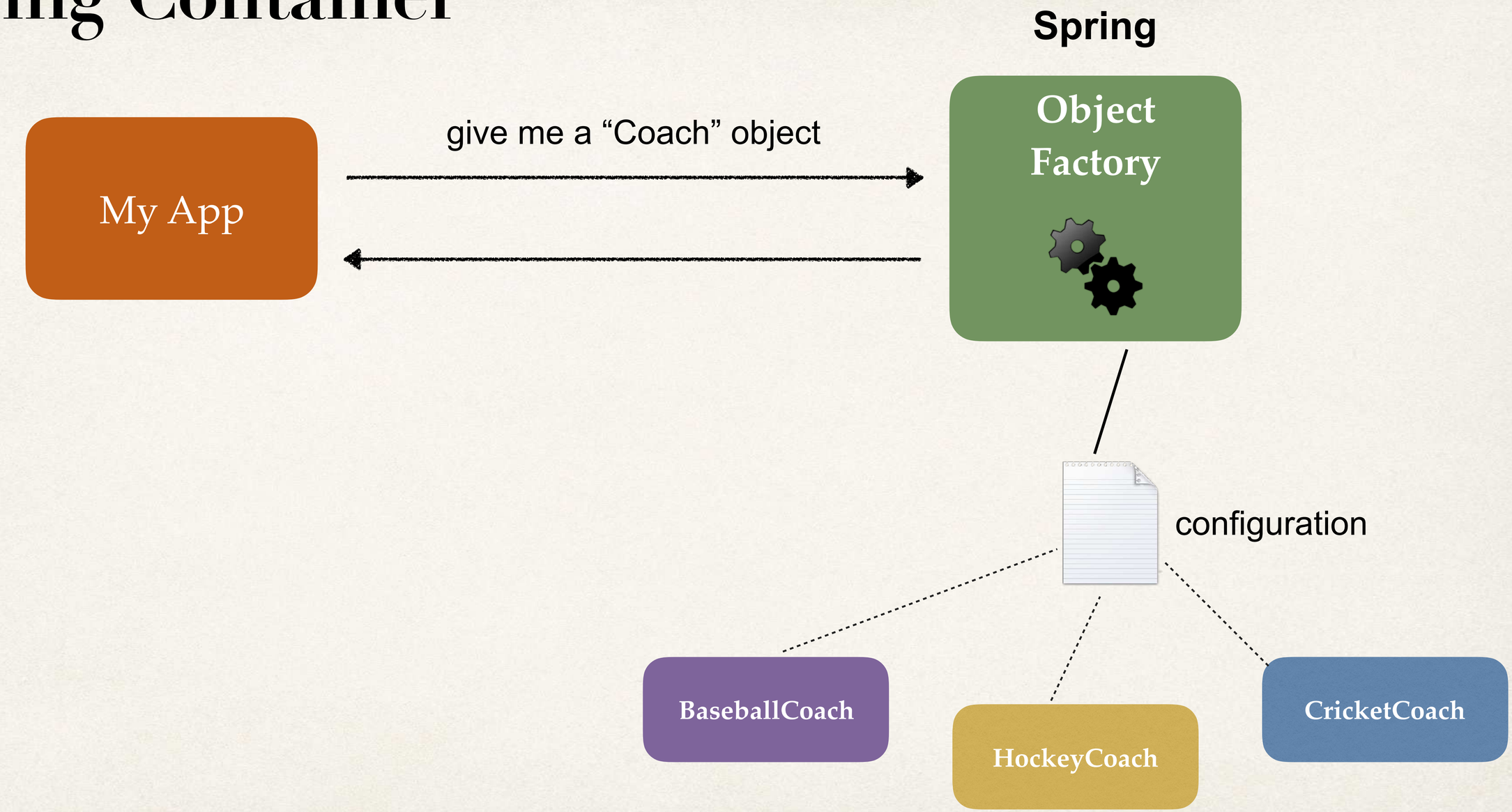
Inversion of Control

Spring Inversion of Control

Ideal Solution



Spring Container



Spring Container

- Primary functions
  - Create and manage objects *(Inversion of Control)*
  - Inject object's dependencies *(Dependency Injection)*

Configuring Spring Container

- XML configuration file *(legacy, but most legacy apps still use this)*
- Java Annotations *(modern)*
- Java Source Code *(modern)*

Spring Development Process

Step-By-Step

1. Configure your Spring Beans

2. Create a Spring Container

3. Retrieve Beans from Spring Container

Step 1: Configure your Spring Beans

File: applicationContext.xml

```
<beans ... >

   <bean  id="myCoach"
      class="com.luv2code.springdemo.BaseballCoach">
   </bean>

</beans>
```

Step 2: Create a Spring Container

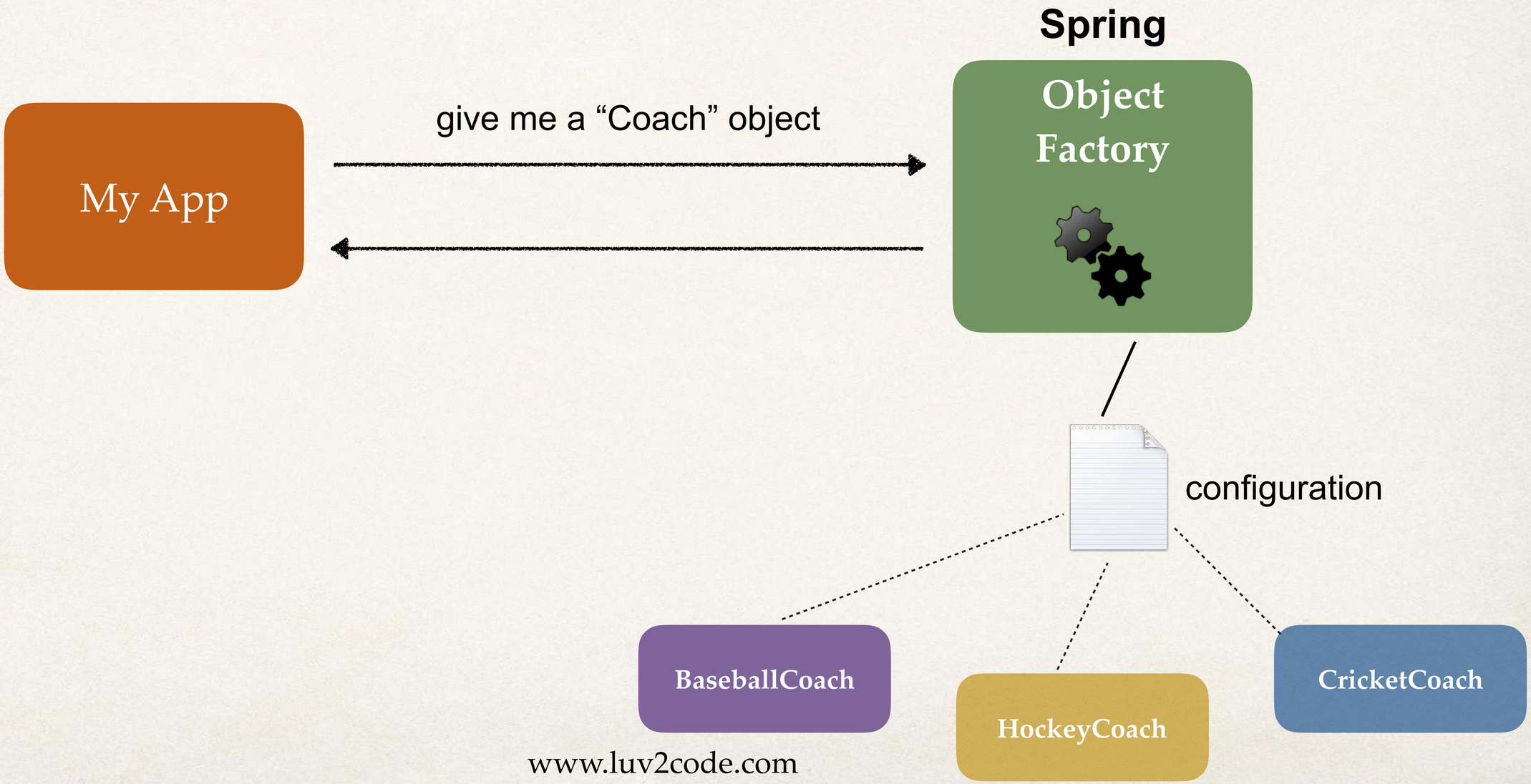
- Spring container is generically known as **ApplicationContext**

- Specialized implementations
  - ClassPathXmlApplicationContext
  - AnnotationConfigApplicationContext
  - GenericWebApplicationContext
  - others …

Step 2: Create a Spring Container

```
ClassPathXmlApplicationContext context =
    new ClassPathXmlApplicationContext("applicationContext.xml");
```

Step 3: Retrieve Beans from Container



Step 3: Retrieve Beans from Container

```
// create a spring container
ClassPathXmlApplicationContext context =
    new ClassPathXmlApplicationContext("applicationContext.xml");

// retrieve bean from spring container
Coach theCoach = context.getBean("myCoach", Coach.class);
```

File: applicationContext.xml

```
<bean  id="myCoach"
    class="com.luv2code.springdemo.BaseballCoach">
</bean>
```

Spring Dependency Injection

Dependency Injection

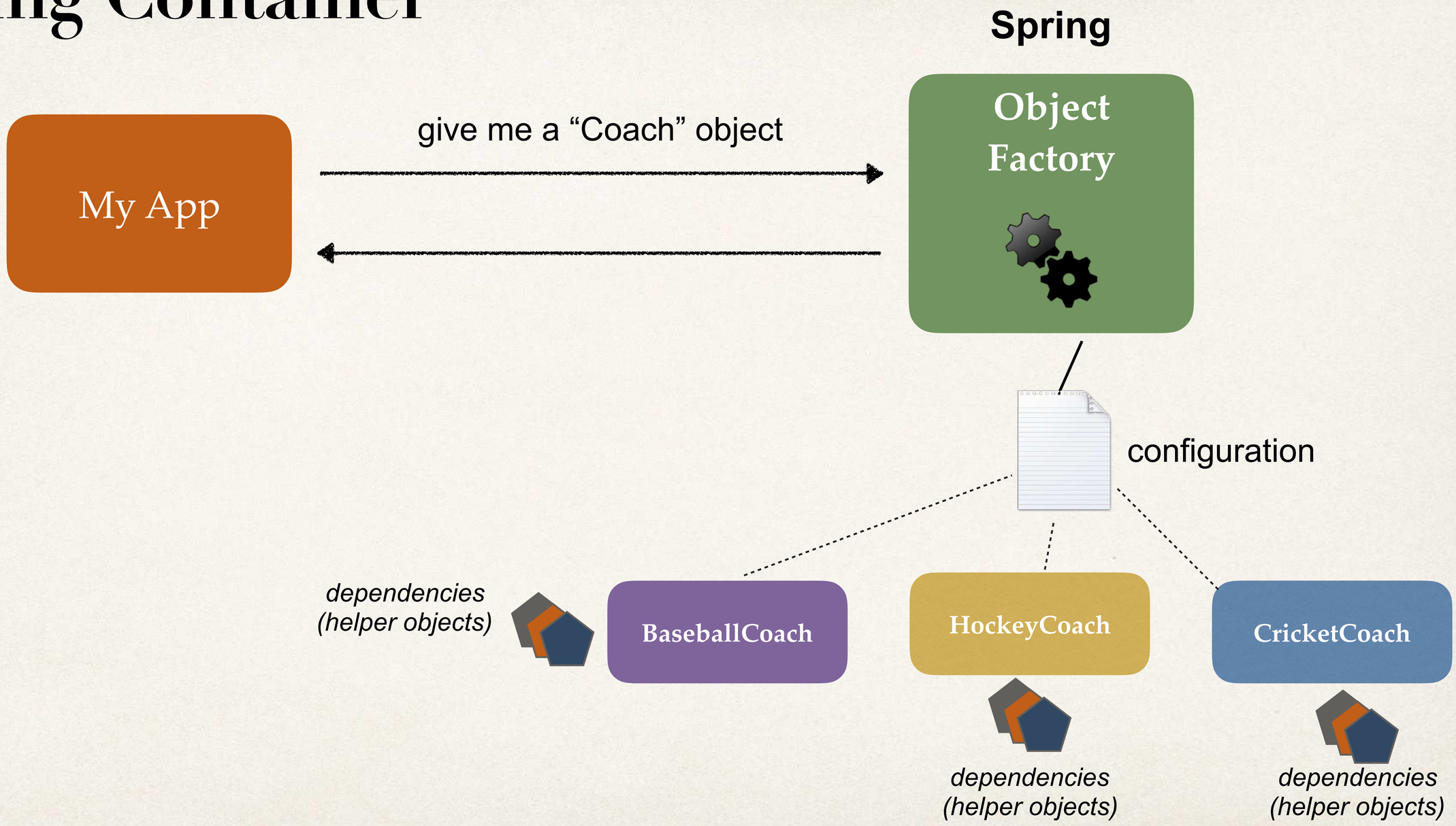
The dependency inversion principle.

The client delegates to calls to another object the responsibility of providing its dependencies.

Car Factory

give me a “Car” object

Spring Container



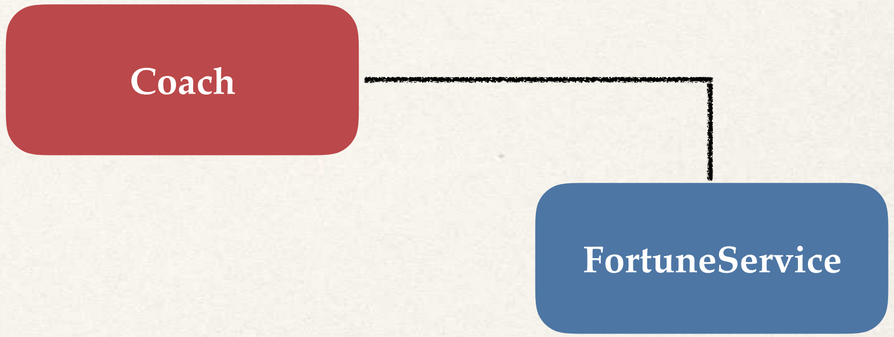
Spring Container

- Primary functions
  - Create and manage objects *(Inversion of Control)*
  - Inject object's dependencies *(Dependency Injection)*

Demo Example

- Our **Coach** already provides daily workouts

- Now will also provide daily fortunes
  - New helper: **FortuneService**
  - This is a ***dependency***



Injection Types

- There are many types of injection with Spring
- We will cover the two most common
  - Constructor Injection
  - Setter Injection
- Will talk about “auto-wiring” in the Annotations section later

Development Process - Constructor Injection

Step-By-Step

1. Define the dependency interface and class

2. Create a constructor in your class for injections

3. Configure the dependency injection in Spring config file

Step 1: Define the dependency interface and class

File: FortuneService.java

```
public interface FortuneService {

   public String getFortune();

}
```

File: HappyFortuneService.java

```
public class HappyFortuneService implements FortuneService {

   public String getFortune() {
      return "Today is your lucky day!";
   }
}
```

Step 2: Create a constructor in your class for injections

File: BaseballCoach.java

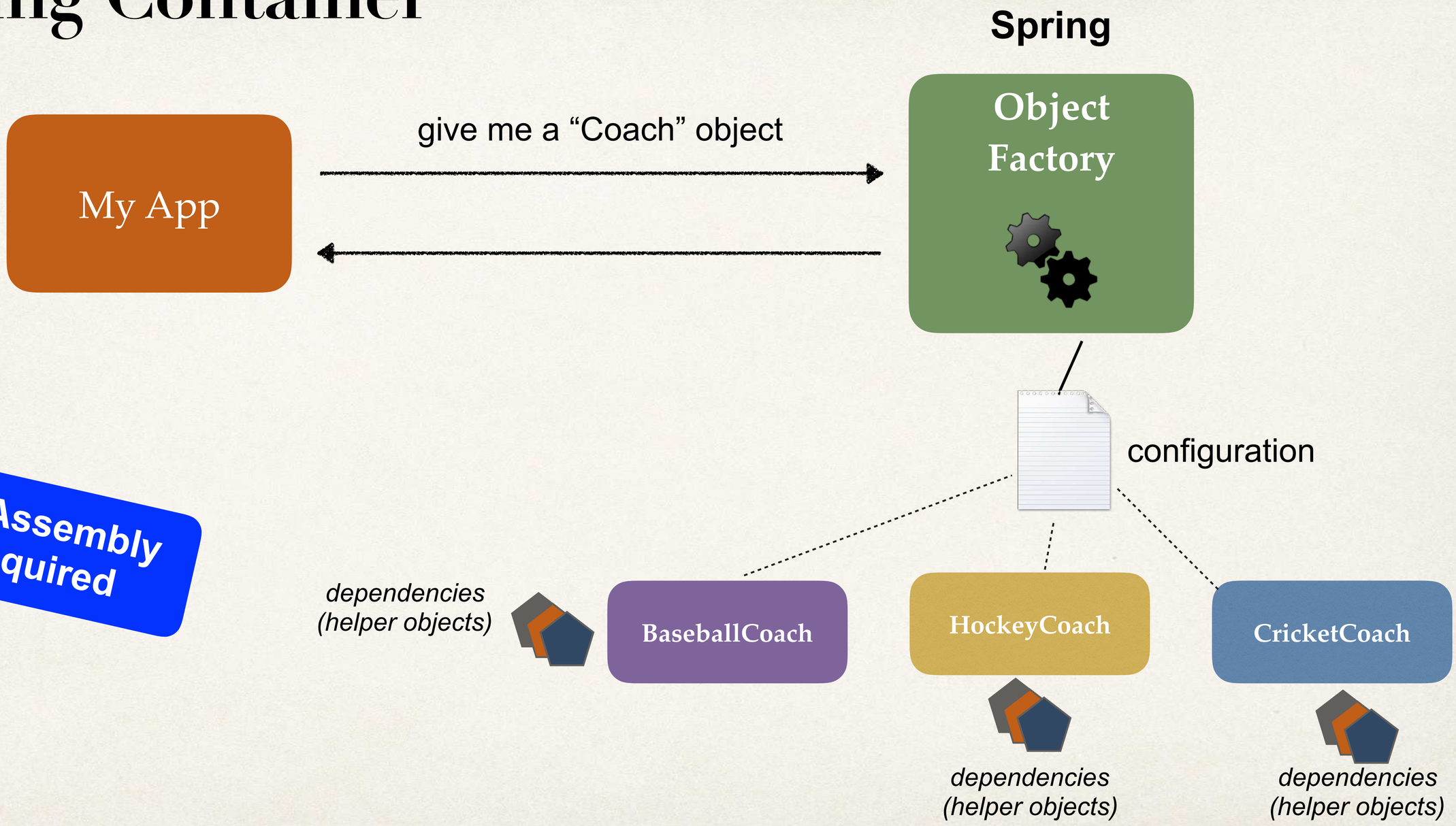
```
public class BaseballCoach implements Coach {

   private FortuneService fortuneService;

   public BaseballCoach(FortuneService theFortuneService) {
      fortuneService = theFortuneService;
   }
  ...
}
```

Step 3: Configure the dependency injection in Spring config file

File: applicationContext.xml

```
<bean id="myFortuneService"
    class="com.luv2code.springdemo.HappyFortuneService">
</bean>

<bean  id="myCoach"
   class="com.luv2code.springdemo.BaseballCoach">
     <constructor-arg ref="myFortuneService" />
</bean>
```

Spring Container



Constructor Injection - Behind the Scenes

How Spring Processes your Config File

File: applicationContext.xml

```
<bean id="myFortuneService"
    class="com.luv2code.springdemo.HappyFortuneService">
 </bean>

<bean  id="myCoach"
   class="com.luv2code.springdemo.BaseballCoach">
     <constructor-arg ref="myFortuneService" />
</bean>
```

The Spring Framework will perform operations behind the scenes for you :-)

How Spring Processes your Config File

```
<bean id="myFortuneService"
      class="com.luv2code.springdemo.HappyFortuneService">
 </bean>

<bean  id="myCoach"
   class="com.luv2code.springdemo.BaseballCoach">
     <constructor-arg ref="myFortuneService" />
</bean>
```

Spring Framework

```
HhappyFortuneService myFortuneService =
                        new HappyFortuneService();

BaseballCoach myCoach =
            new BaseballCoach(myFortuneService);
```

Setter Injection

Spring Injection Types

- Constructor Injection
- Setter Injection

Inject dependencies by calling

setter method(s) on your class

Development Process - Setter Injection

Step-By-Step

1. Create setter method(s) in your class for injections

2. Configure the dependency injection in Spring config file

Step1: Create setter method(s) in your class for injections

File: CricketCoach.java

```
public class CricketCoach implements Coach {

  private FortuneService fortuneService;

  public CricketCoach() {
  }

  public void setFortuneService(FortuneService fortuneService) {
    this.fortuneService = fortuneService;
  }
 ...
}
```

Step 2: Configure the dependency injection in Spring config file

File: applicationContext.xml

```
<bean id="myFortuneService"
    class="com.luv2code.springdemo.HappyFortuneService">
</bean>

<bean id="myCricketCoach"
   class="com.luv2code.springdemo.CricketCoach">

   <property name="fortuneService" ref="myFortuneService" />

</bean>
```

Call setter method on Java class

```
<property name="fortuneService" ref="myFortuneService" />
```

```
public void setFortuneService(...)
```

Call setter method on Java class

```
<property name="bestAthlete" ref="…" />
```

```
public void setBestAthlete(…)
```

How Spring Processes your Config File

```
<bean id="myFortuneService"
    class="com.luv2code.springdemo.HappyFortuneService">
 </bean>
```

Spring Framework

```
HappyFortuneService myFortuneService =
                    new HappyFortuneService();
```

How Spring Processes your Config File

```
<bean id="myFortuneService"
      class="com.luv2code.springdemo.HappyFortuneService">
 </bean>
```

Spring Framework

```
HappyFortuneService myFortuneService =
                        new HappyFortuneService();
```

```
<bean id="myCricketCoach"
      class="com.luv2code.springdemo.CricketCoach">

      <property name="fortuneService" ref="myFortuneService" />

</bean>
```

Spring Framework

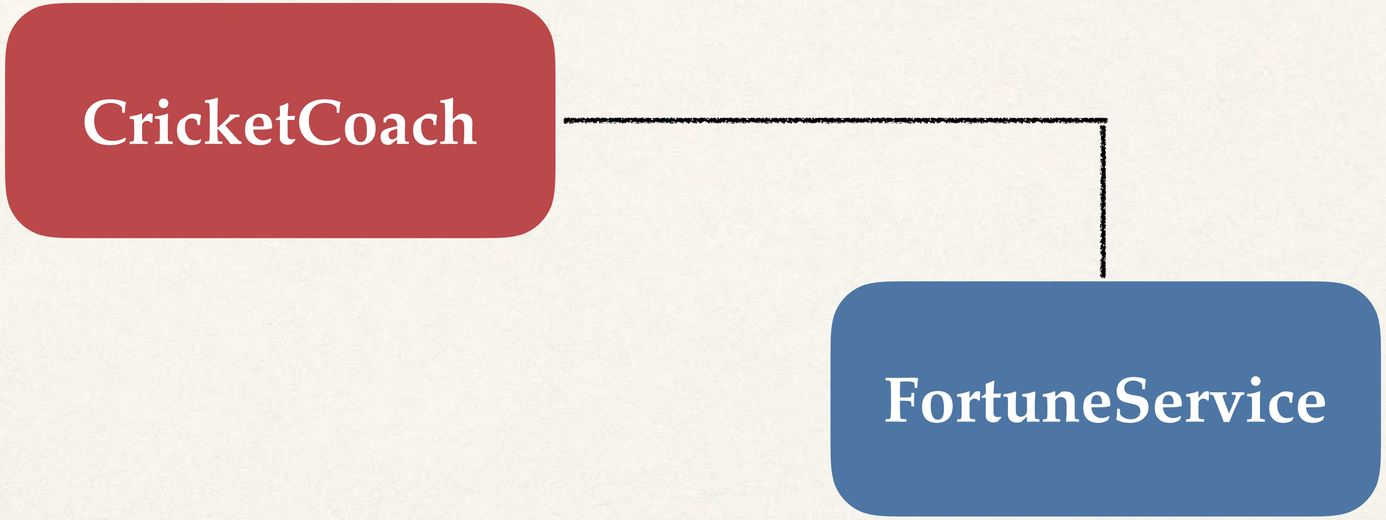
```
CricketCoach myCricketCoach =
            new CricketCoach();

myCricketCoach.setFortuneService(myFortuneService);
```

Injecting Literal Values

Injecting Literal Values

emailAddress: thebestcoach@luv2code.com

team: Sunrisers Hyderabad



Development Process

Step-By-Step

1. Create setter method(s) in your class for injections

2. Configure the injection in Spring config file

Step1: Create setter method(s) in your class for injections

File: CricketCoach.java

```
public class CricketCoach implements Coach {

   private String emailAddress;
   private String team;

   public void setEmailAddress(String emailAddress) ...

   public void setTeam(String team) ...

   ...
}
```

Step 2: Configure the injection in Spring config file

File: applicationContext.xml

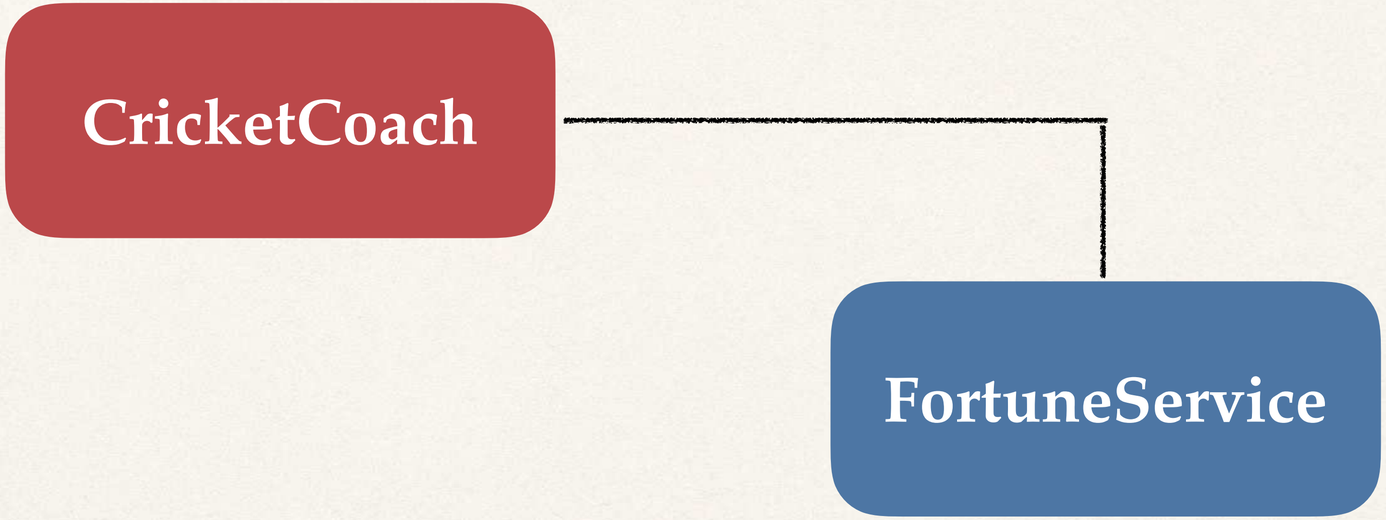
```
<bean id="myCricketCoach"
  class="com.luv2code.springdemo.CricketCoach">

  <property name="fortuneService" ref="myFortuneService" />

  <property name="emailAddress" value="thebestcoach@luv2code.com" />

  <property name="team" value="Sunrisers Hyderabad" />

</bean>
```

Injecting Values from Properties File

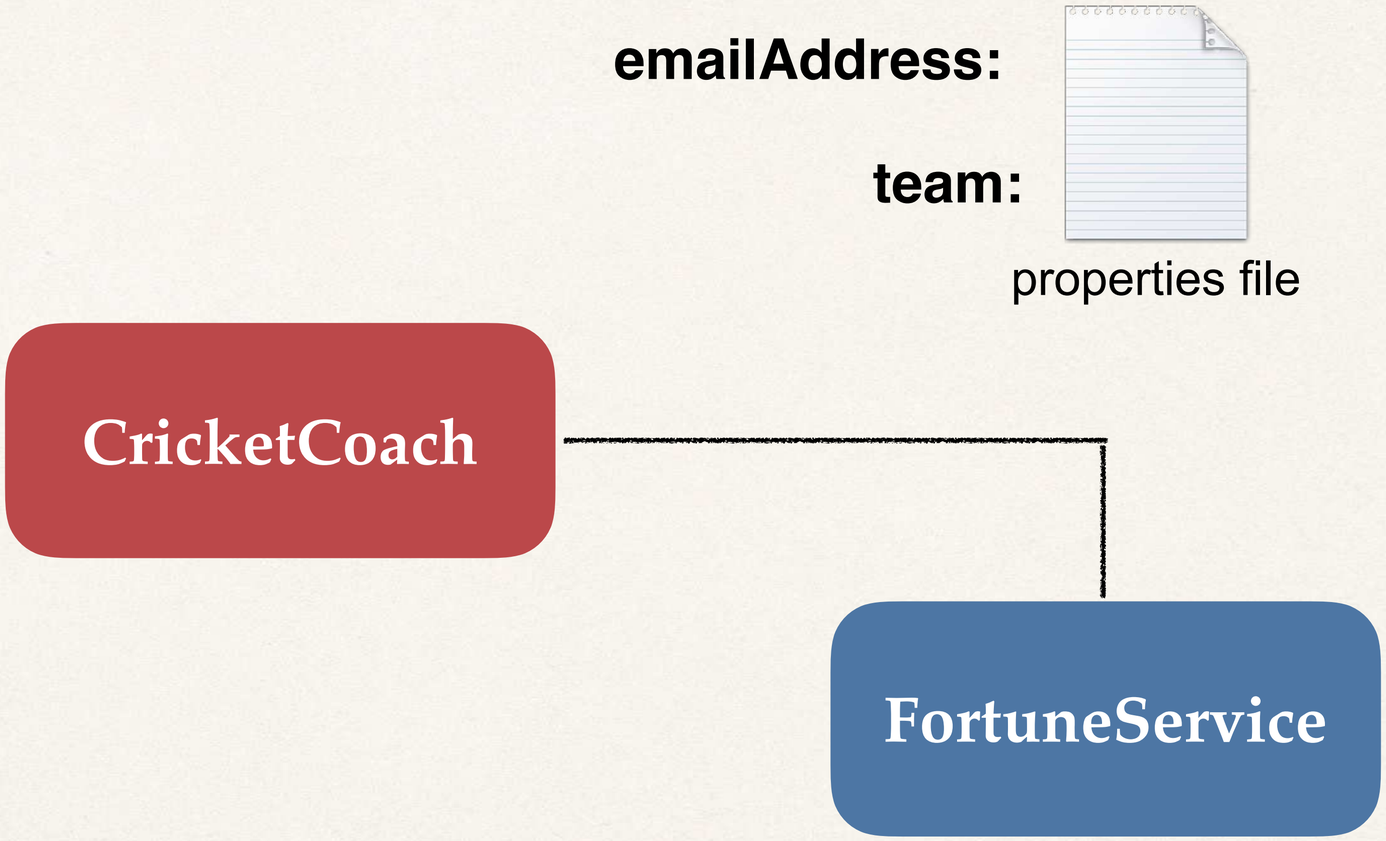
Injecting Literal Values

emailAddress: thebestcoach@luv2code.com

team: Sunrisers Hyderabad



Read from a Properties File



Development Process

1. Create Properties File

2. Load Properties File in Spring config file

3. Reference values from Properties File

Step-By-Step

Step 1: Create Properties File

File: sport.properties

```
foo.email=myeasycoach@luv2code.com
foo.team=Royal Challengers Bangalore
```

Step 2: Load Properties file in Spring config file

File: applicationContext.xml

```
<context:property-placeholder location="classpath:sport.properties"/>
```

Step 3: Reference Values from Properties File

File: applicationContext.xml

```
<bean id="myCricketCoach"
  class="com.luv2code.springdemo.CricketCoach">
  …
  <property name="emailAddress" value="${foo.email}" />
  <property name="team" value="${foo.team}" />

</bean>
```

```
foo.email=myeasycoach@luv2code.com
foo.team=Royal Challengers Bangalore
```

Step 1: Create Properties File

File: sport.properties

```
foo.email=silly@luv2code.com
foo.team=Mighty Java Coders
```

Bean Scopes

Bean Scopes

- Scope refers to the lifecycle of a bean
- How long does the bean live?
- How many instances are created?
- How is the bean shared?

Default Scope: Singleton

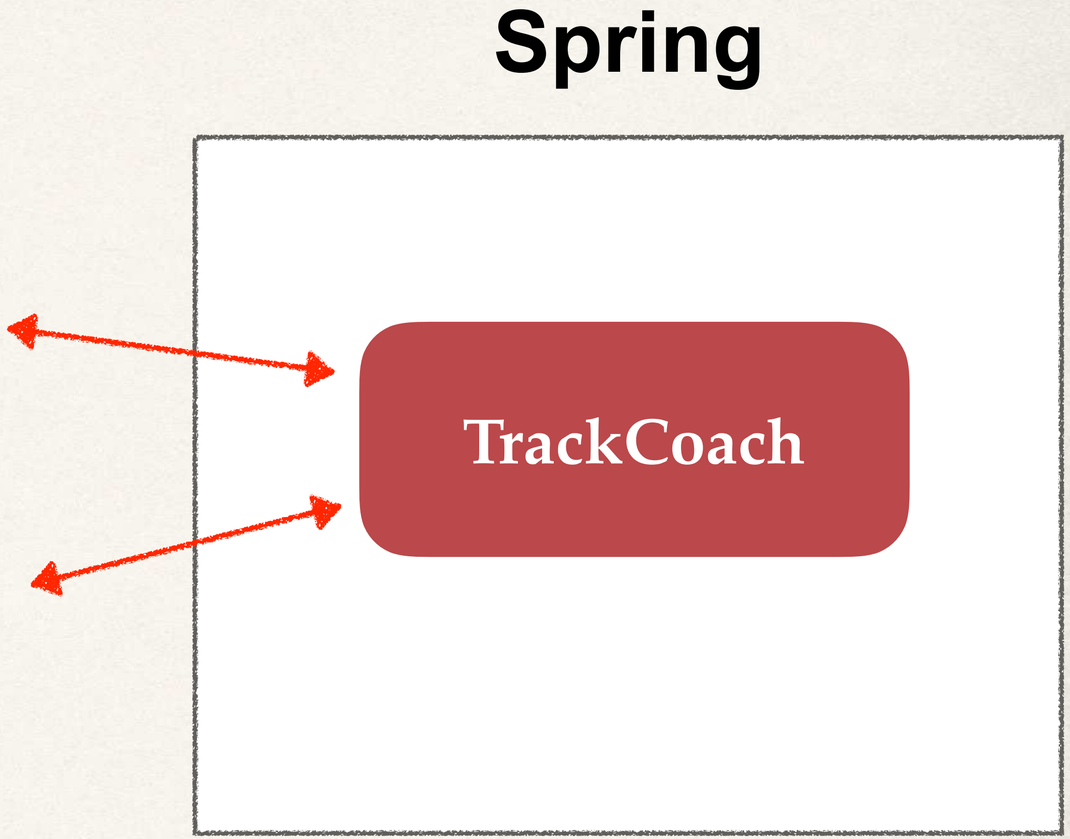
```
<beans ... >

    <bean  id="myCoach"
        class="com.luv2code.springdemo.TrackCoach">
            ...
    </bean>

</beans>
```

What Is a Singleton?

- Spring Container creates only one instance of the bean, by default
- It is cached in memory
- All requests for the bean
  - will return a SHARED reference to the SAME bean

What is a Singleton?

```
Coach theCoach = context.getBean("myCoach", Coach.class);

...

Coach alphaCoach = context.getBean("myCoach", Coach.class);
```



Explicitly Specify Bean Scope

```
<beans ... >

    <bean  id="myCoach"
        class="com.luv2code.springdemo.TrackCoach"
        scope="singleton">
            ...
    </bean>

</beans>
```

Additional Spring Bean Scopes

Scope	Description
singleton	Create a single shared instance of the bean. Default scope.
prototype	Creates a new bean instance for each container request.
request	Scoped to an HTTP web request. Only used for web apps.
session	Scoped to an HTTP web session. Only used for web apps.
global-session	Scoped to a global HTTP web session. Only used for web apps.

Prototype Scope Example

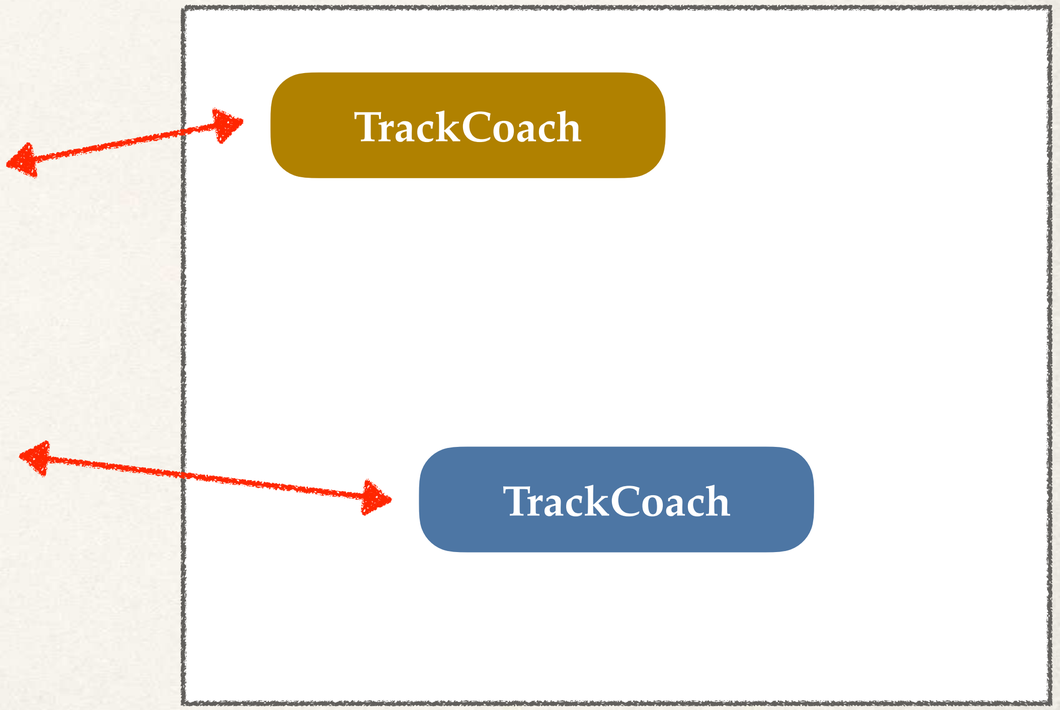
Prototype scope: new object for each request

```
<beans ... >

    <bean  id="myCoach"
        class="com.luv2code.springdemo.TrackCoach"
        scope="prototype">
            ...
    </bean>

</beans>
```

Prototype Scope Example

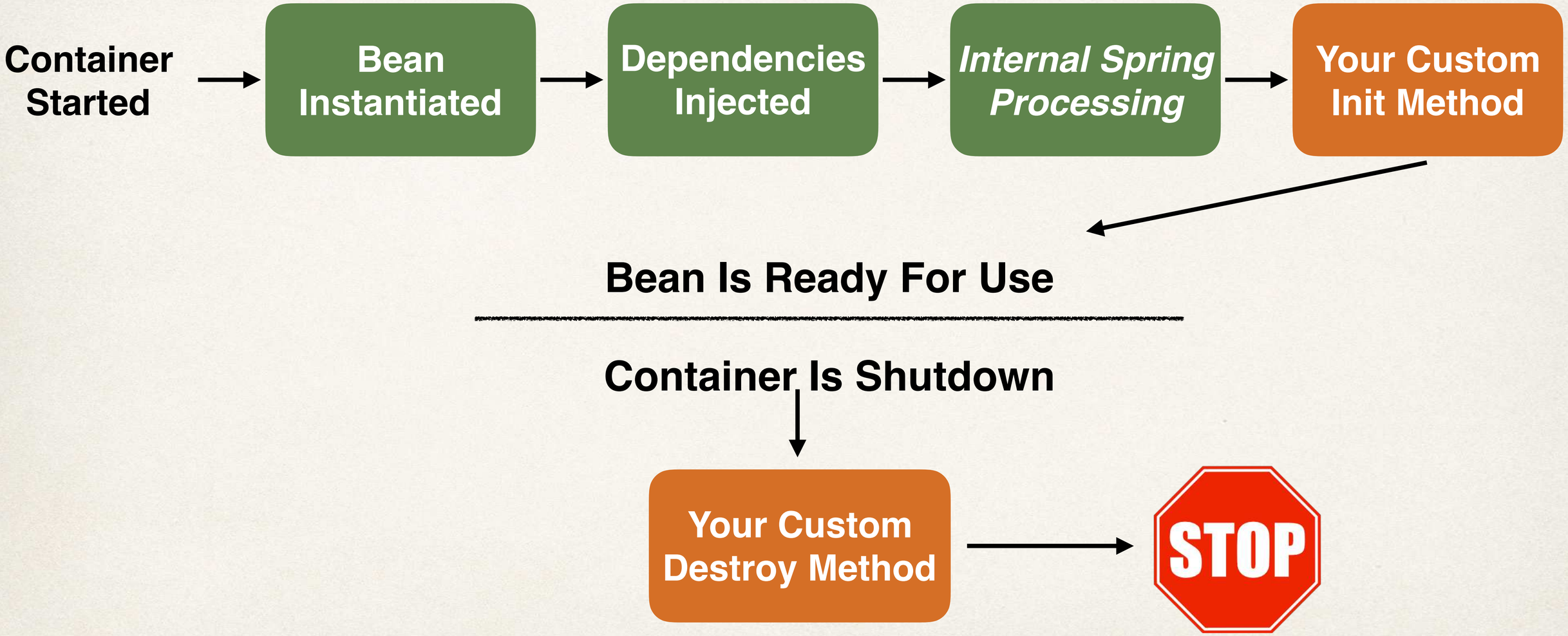
Prototype scope: new object for each request

```
Coach theCoach = context.getBean("myCoach", Coach.class);

...

Coach alphaCoach = context.getBean("myCoach", Coach.class);
```



Bean Lifecycle Methods

Bean Lifecycle



Bean Lifecycle Methods / Hooks

- You can add custom code during **bean initialization**
  - Calling custom business logic methods
  - Setting up handles to resources (db, sockets, file etc)

- You can add custom code during **bean destruction**
  - Calling custom business logic method
  - Clean up handles to resources (db, sockets, files etc)

Init: method configuration

```
<beans ... >

    <bean  id="myCoach"
        class="com.luv2code.springdemo.TrackCoach"
          init-method="doMyStartupStuff">
          ...
    </bean>

</beans>
```

Destroy: method configuration

```
<beans ... >

   <bean  id="myCoach"
      class="com.luv2code.springdemo.TrackCoach"
         init-method="doMyStartupStuff"
         destroy-method="doMyCleanupStuff">
         ...
   </bean>

</beans>
```

Development Process

Step-By-Step

1. Define your methods for init and destroy

2. Configure the method names in Spring config file

Spring Configuration with Annotations

What are Java Annotations?

- Special labels/markers added to Java classes
- Provide meta-data about the class
- Processed at compile time or run-time for special processing

Annotation Example

- We've seen annotations already …

```
public class TrackCoach implements Coach {

  @Override
  public String getDailyWorkout() {
    return "Run a hard 5k";
  }
  ...
}
```

Why Spring Configuration with Annotations?

- XML configuration can be verbose
- Configure your Spring beans with Annotations
- Annotations minimizes the XML configuration

Scanning for Component Classes

- Spring will scan your Java classes for special annotations
- Automatically register the beans in the Spring container

Development Process

Step-By-Step

1. Enable component scanning in Spring config file

2. Add the @Component Annotation to your Java classes

3. Retrieve bean from Spring container

Step 1: Enable component scanning in Spring config file

```
<beans ... >

  <context:component-scan base-package="com.luv2code.springdemo" />

</beans>
```

Step 2: Add the @Component Annotation to your Java classes

```
@Component("thatSillyCoach")
public class TennisCoach implements Coach {

  @Override
  public String getDailyWorkout() {
    return "Practice your backhand volley";
  }

}
```

Step 3: Retrieve bean from Spring container

- Same coding as before … nothing changes.

```
Coach theCoach = context.getBean("thatSillyCoach", Coach.class);
```

@Components - Default Bean ID

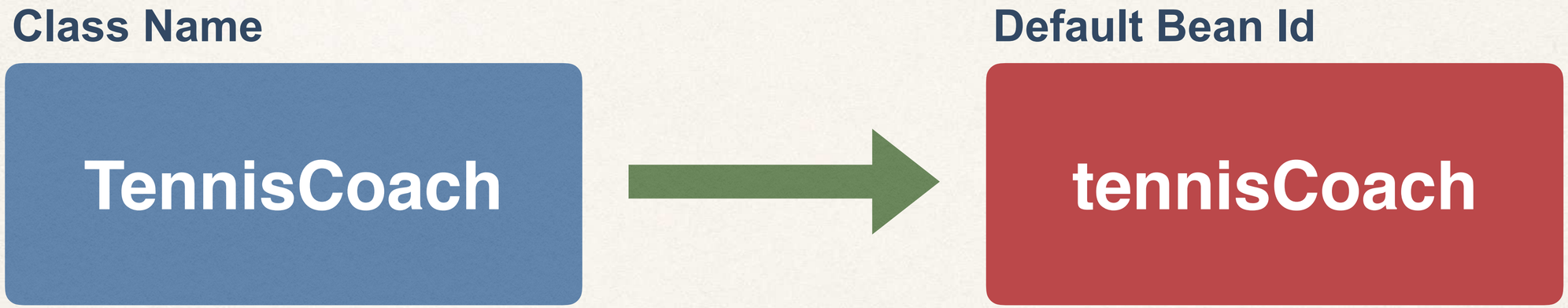
We already learned …

1. Specify the bean id in the component annotation

```
@Component("thatSillyCoach")
public class TennisCoach implements Coach {
```

Spring also supports Default Bean IDs

- Default bean id: the class name, *make first letter lower-case*



Code example

```
@Component
public class TennisCoach implements Coach {
```

Code example

```
@Component
public class TennisCoach implements Coach {
```

Class Name

TennisCoach

Default Bean Id

tennisCoach

Code example

```
@Component
public class TennisCoach implements Coach {
```

```
// get the bean from spring container
Coach theCoach = context.getBean("tennisCoach", Coach.class);
```

Code example

```
@Component
public class TennisCoach implements Coach {
```

Class Name

HappyFortuneService

Default Bean Id

happyFortuneService

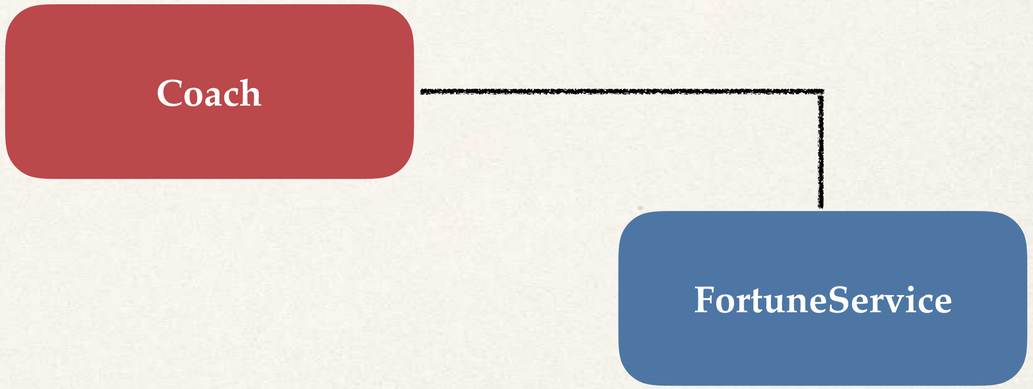
luv2code

Spring Dependency Injection with Annotations and Autowiring

Demo Example

- Our **Coach** already provides daily workouts

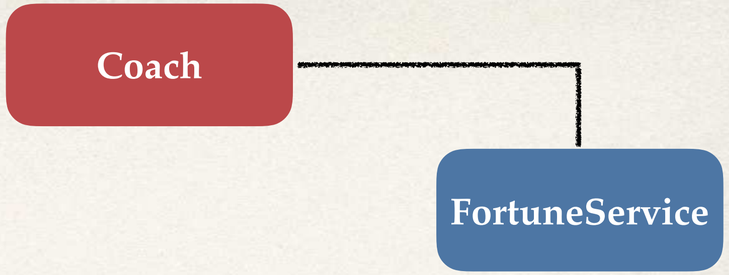
- Now will also provide daily fortunes
  - New helper: **FortuneService**
  - This is a ***dependency***



What is Spring AutoWiring?

- For dependency injection, Spring can use auto wiring
- Spring will look for a class that *matches* the property
  - *matches by type*: class or interface
- Spring will inject it automatically … hence it is autowired

Autowiring Example



- Injecting FortuneService into a Coach implementation
- Spring will scan @Components
- Any one implements FortuneService interface???
- If so, let's inject them. For example: *HappyFortuneService*

Autowiring Injection Types

- Constructor Injection
- Setter Injection
- Field Injections

Development Process - Constructor Injection

Step-By-Step

1. Define the dependency interface and class

2. Create a constructor in your class for injections

3. Configure the dependency injection with **@Autowired** Annotation

Step 1: Define the dependency interface and class

File: FortuneService.java

```
public interface FortuneService {

   public String getFortune();

}
```

File: HappyFortuneService.java

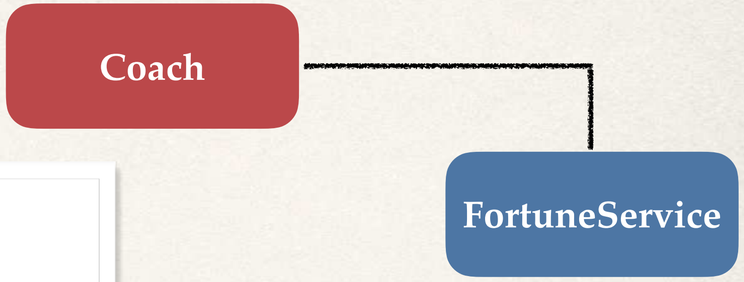
```
@Component
public class HappyFortuneService implements FortuneService {

   public String getFortune() {
      return "Today is your lucky day!";
   }
}
```

Step 2: Create a constructor in your class for injections

File: TennisCoach.java

```
@Component
public class TennisCoach implements Coach {

   private FortuneService fortuneService;

   public TennisCoach(FortuneService theFortuneService) {
      fortuneService = theFortuneService;
   }
  ...
}
```

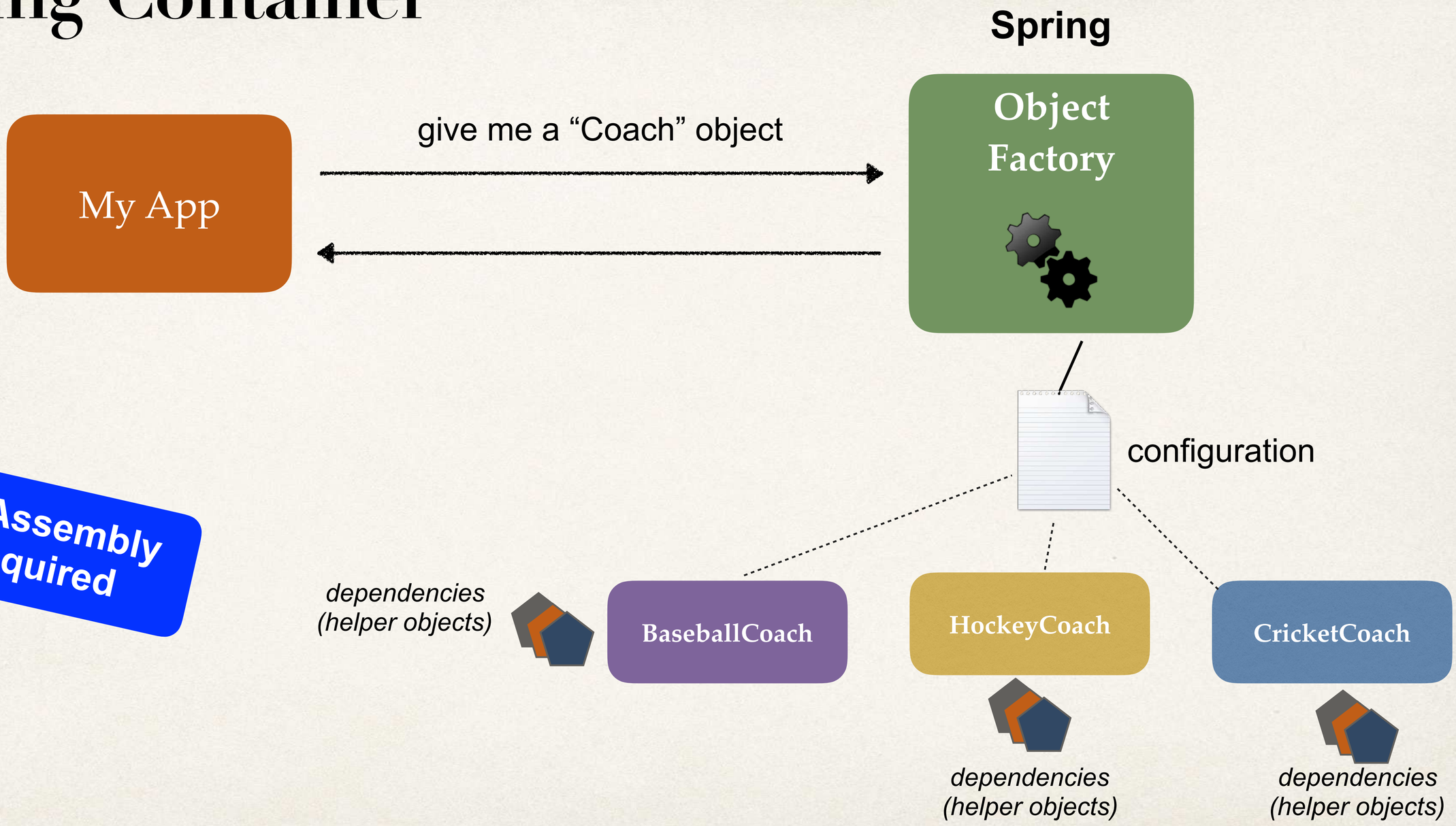
Step 3: Configure the dependency injection @Autowired annotation



File: TennisCoach.java

```
@Component
public class TennisCoach implements Coach {

   private FortuneService fortuneService;

   @Autowired
  public TennisCoach(FortuneService theFortuneService) {
     fortuneService = theFortuneService;
   }
  ...
}
```

Spring Container



Setter Injection with Annotations and Autowiring

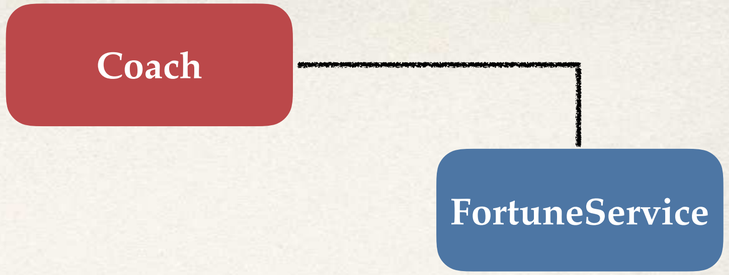
Spring Injection Types

- Constructor Injection
- Setter Injection
- Field Injection

Inject dependencies by calling

setter method(s) on your class

Autowiring Example



- Injecting FortuneService into a Coach implementation
- Spring will scan @Components
- Any one implements FortuneService interface???
- If so, let's inject them. For example: *HappyFortuneService*

Development Process - Setter Injection

Step-By-Step

1. Create setter method(s) in your class for injections

2. Configure the dependency injection with **@Autowired** Annotation

Step1: Create setter method(s) in your class for injections

File: TennisCoach.java

```
@Component
public class TennisCoach implements Coach {

   private FortuneService fortuneService;

   public TennisCoach() {
   }

   public void setFortuneService(FortuneService fortuneService) {
      this.fortuneService = fortuneService;
   }
  ...
}
```

Step 2: Configure the dependency injection with Autowired Annotation

File: TennisCoach.java

```
@Component
public class TennisCoach implements Coach {

  private FortuneService fortuneService;

  public TennisCoach() {
  }

  @Autowired
  public void setFortuneService(FortuneService fortuneService) {
    this.fortuneService = fortuneService;
  }
 ...
}
```

Inject dependencies by calling

ANY method on your class

Simply give: @Autowired

Step 2: Configure the dependency injection with Autowired Annotation

File: TennisCoach.java

```
@Component
public class TennisCoach implements Coach {

  private FortuneService fortuneService;

  public TennisCoach() {
  }

  @Autowired
  public void doSomeCrazyStuff(FortuneService fortuneService) {
    this.fortuneService = fortuneService;
  }
 ...
}
```

luv2code

Field Injection with Annotations and Autowiring

Spring Injection Types

- Constructor Injection
- Setter Injection
- Field Injection

Inject dependencies by setting field values on your class directly

(even private fields)

Accomplished by using Java Reflection

Development Process - Field Injection

Step-By-Step

1. Configure the dependency injection with Autowired Annotation
   - Applied directly to the field
   - No need for setter methods

Step 1: Configure the dependency injection with Autowired Annotation

File: TennisCoach.java

```
public class TennisCoach implements Coach {

   @Autowired
  private FortuneService fortuneService;

   public TennisCoach() {
   }

   // no need for setter methods
 ...
}
```

Spring Injection Types

- Constructor Injection
- Setter Injection
- Field Injection

Spring Injection Types

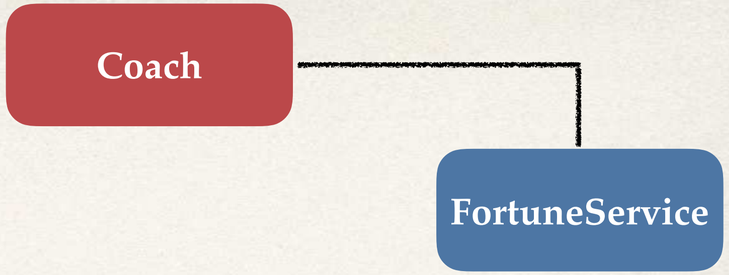
- Constructor Injection
- Setter Injection
- Field Injection

Choose a style

Stay consistent in your project

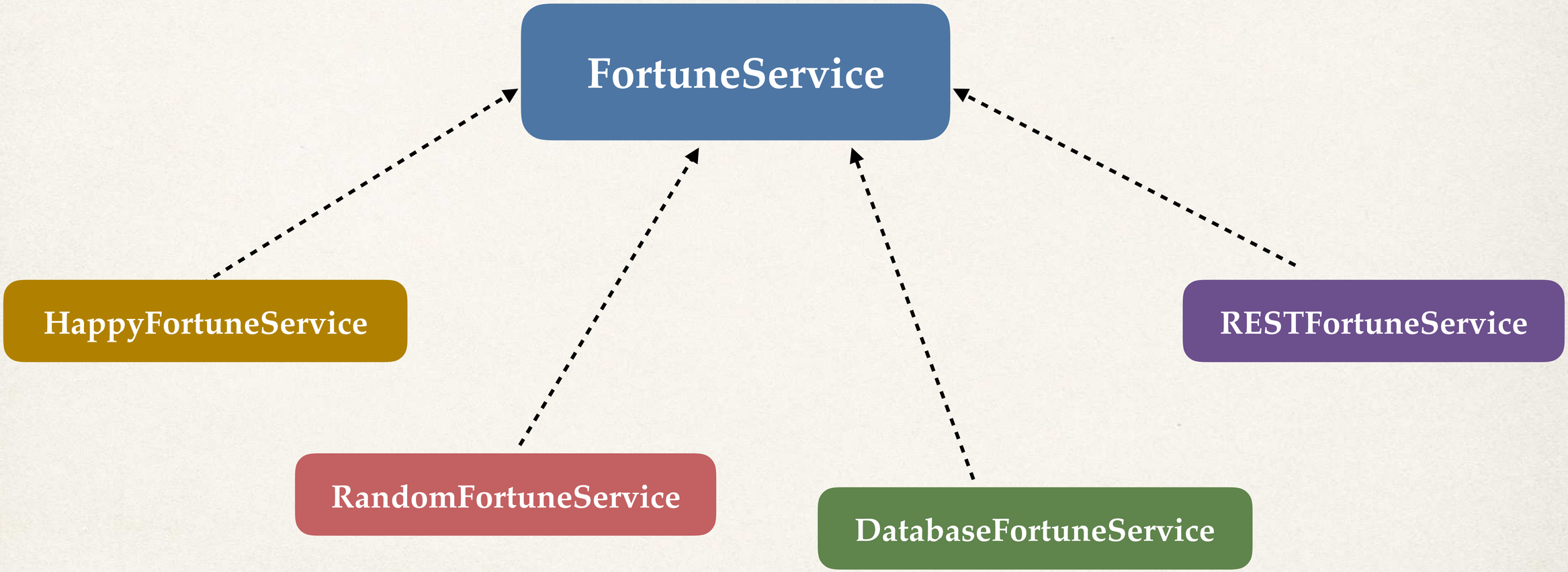
Annotation Autowiring and Qualifiers

Autowiring



- Injecting FortuneService into a Coach implementation
- Spring will scan @Components
- Any one implements FortuneService interface???
- If so, let's inject them … ***oops which one?***

Multiple FortuneService Implementations



Umm, we have a little problem ....

Error creating bean with name 'tennisCoach':
Injection of autowired dependencies failed

Caused by: org.springframework.beans.factory.NoUniqueBeanDefinitionException:

No qualifying bean of type [com.luv2code.springdemo.FortuneService] is defined:
expected single matching bean but found 4:

databaseFortuneService,happyFortuneService,randomFortuneService,RESTFortuneService

Solution: Be specific! - @Qualifier

```
@Component
public class TennisCoach implements Coach {

  @Autowired
  @Qualifier("happyFortuneService")
  private FortuneService fortuneService;

 ...

}
```

Injection Types

- Can apply **@Qualifier** annotation to
  - Constructor injection
  - Setter injection methods
  - Field injection

Bean Scopes with Annotations

Bean Scopes

- Scope refers to the lifecycle of a bean
- How long does the bean live?
- How many instances are created?
- How is the bean shared?

Default Scope

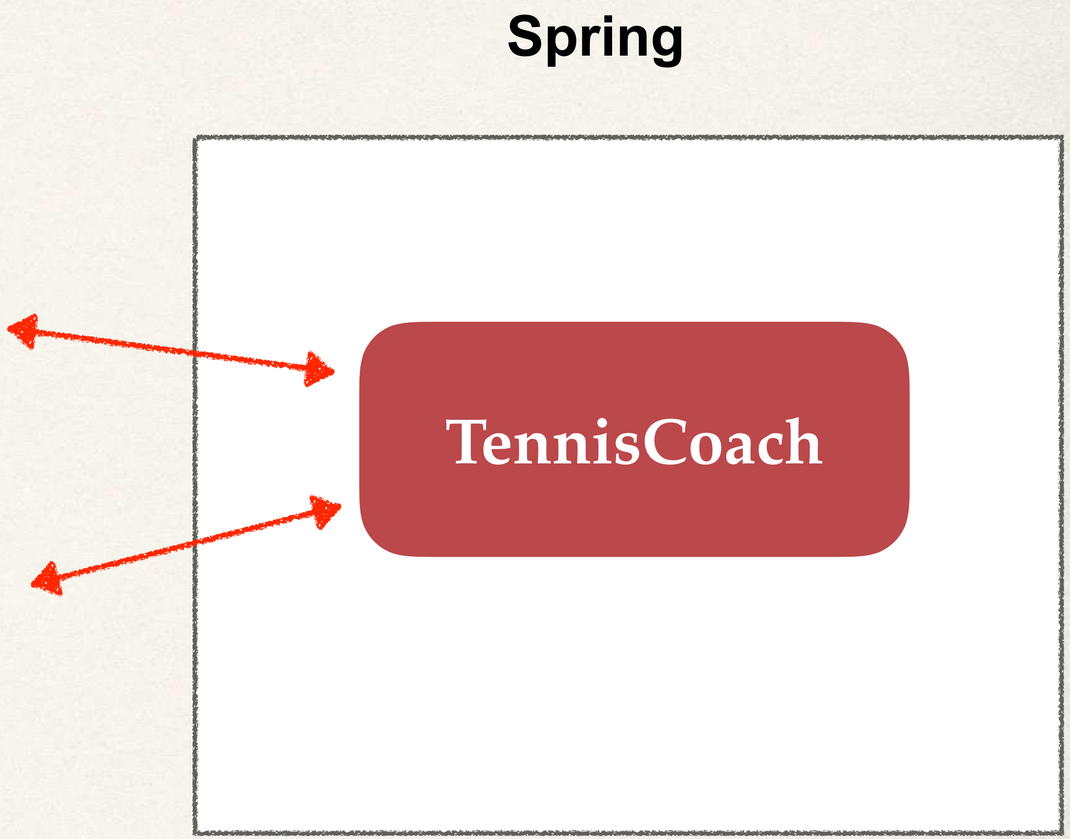
Default scope is singleton

Refresher: What Is a Singleton?

- Spring Container creates only one instance of the bean, by default
- It is cached in memory
- All requests for the bean
  - will return a SHARED reference to the SAME bean

What is a Singleton?

```
Coach theCoach = context.getBean("tennisCoach", Coach.class);

...

Coach alphaCoach = context.getBean("tennisCoach", Coach.class);
```



Explicitly Specify Bean Scope

```
@Component
@Scope("singleton")
public class TennisCoach implements Coach {

...

}
```

Additional Spring Bean Scopes

Scope	Description
singleton	Create a single shared instance of the bean. Default scope.
prototype	Creates a new bean instance for each container request.
request	Scoped to an HTTP web request. Only used for web apps.
session	Scoped to an HTTP web session. Only used for web apps.
global-session	Scoped to a global HTTP web session. Only used for web apps.

Prototype Scope Example

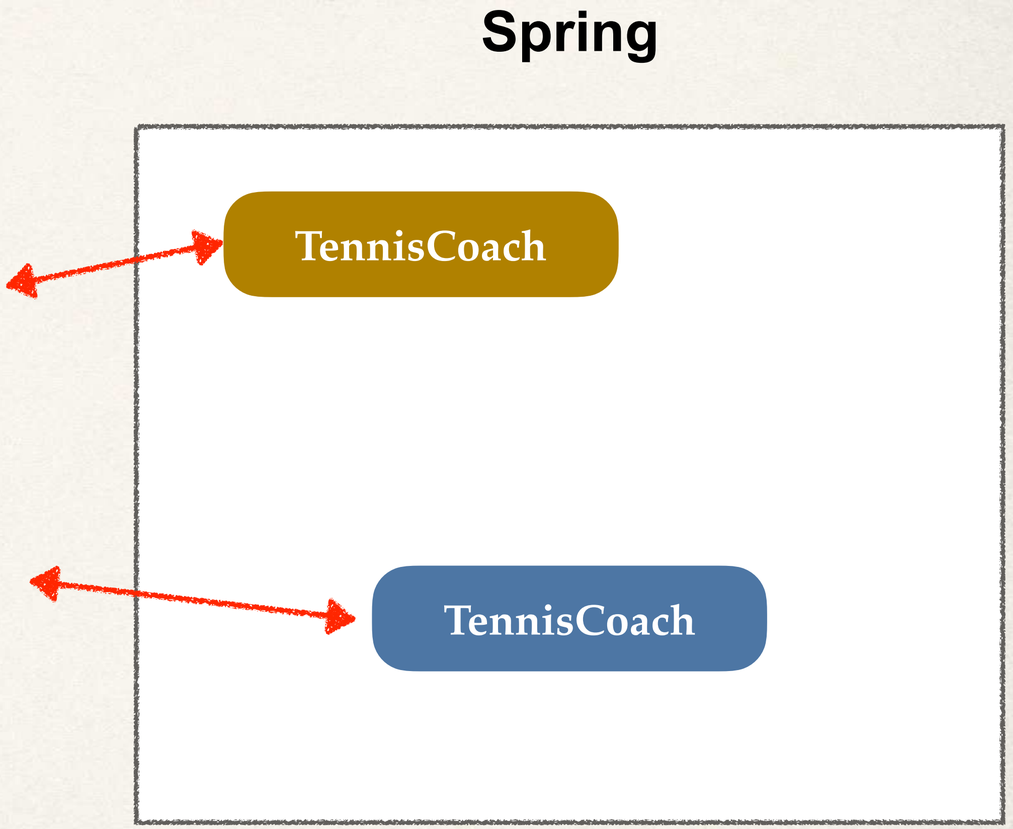
Prototype scope: new object for each request

```
@Component
@Scope("prototype")
public class TennisCoach implements Coach {

...

}
```

Prototype Scope Example

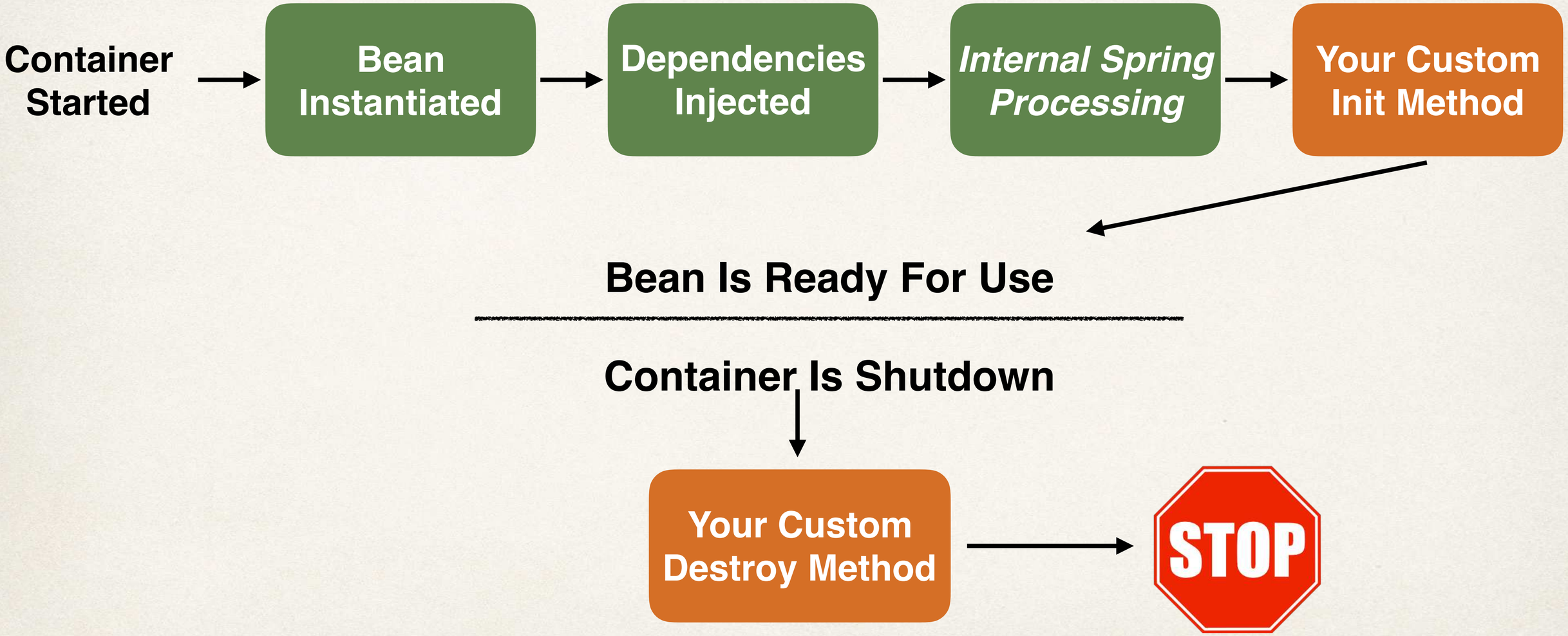
Prototype scope: new object for each request

```
Coach theCoach = context.getBean("tennisCoach", Coach.class);

...

Coach alphaCoach = context.getBean("tennisCoach", Coach.class);
```



Bean Lifecycle Methods - Annotations

Bean Lifecycle



Bean Lifecycle Methods / Hooks

- You can add custom code during **bean initialization**
  - Calling custom business logic methods
  - Setting up handles to resources (db, sockets, file etc)

- You can add custom code during **bean destruction**
  - Calling custom business logic method
  - Clean up handles to resources (db, sockets, files etc)

Init: method configuration

```
@Component
public class TennisCoach implements Coach {

  @PostConstruct
  public void doMyStartupStuff() { … }

 …

}
```

Destroy: method configuration

```
@Component
public class TennisCoach implements Coach {

  @PreDestroy
  public void doMyCleanupStuff() { … }

 …

}
```

Development Process

Step-By-Step

1. Define your methods for init and destroy

2. Add annotations: @PostConstruct and @PreDestroy

Spring Configuration with Java Code

Java Configuration

- Instead of configuring Spring container using XML
- Configure the Spring container with Java code

No XML!

3 Ways of Configuring Spring Container

1. Full XML Config

```xml
<!-- define the dependency -->
<bean id="myFortuneService"
    class="com.luv2code.springdemo.HappyFortuneService">
</bean>

<bean id="myCoach"
    class="com.luv2code.springdemo.TrackCoach">

    <!-- set up constructor injection -->
    <constructor-arg ref="myFortuneService" />
</bean>

<bean id="myCricketCoach"
    class="com.luv2code.springdemo.CricketCoach">

    <!-- set up setter injection -->
    <property name="fortuneService" ref="myFortuneService" />
```

2. XML Component Scan

```xml
<context:component-scan base-package="com.luv2code.springdemo" />
```

3. Java Configuration Class

```java
package com.luv2code.springdemo;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.luv2code.springdemo")
public class SportConfig {

}
```

No XML!

Development Process

Step-By-Step

1. Create a Java class and annotate as **@Configuration**

2. Add component scanning support: **@ComponentScan** (optional)

3. Read Spring Java configuration class

4. Retrieve bean from Spring container

Step 1: Create a Java class and annotate as @Configuration

```
@Configuration
public class SportConfig {

}
```

Step 2: Add component scanning support: @ComponentScan

Optional

```
@Configuration
@ComponentScan("com.luv2code.springdemo")
public class SportConfig {

}
```

Step 3: Read Spring Java configuration class

```
AnnotationConfigApplicationContext context =
        new AnnotationConfigApplicationContext(SportConfig.class);
```

Step 4: Retrieve bean from Spring container

```
Coach theCoach = context.getBean("tennisCoach", Coach.class);
```

Defining Beans with Java Code

Defining Beans in Spring

Defining Beans in Spring

Full XML Config

```
<!-- define the dependency -->
<bean id="myFortuneService"
    class="com.luv2code.springdemo.HappyFortuneService">
</bean>

<bean id="myCoach"
    class="com.luv2code.springdemo.TrackCoach">

    <!-- set up constructor injection -->
    <constructor-arg ref="myFortuneService" />
</bean>

<bean id="myCricketCoach"
    class="com.luv2code.springdemo.CricketCoach">

    <!-- set up setter injection -->
    <property name="fortuneService" ref="myFortuneService" />
```

Defining Beans in Spring

Full XML Config

```
<!-- define the dependency -->
<bean id="myFortuneService"
    class="com.luv2code.springdemo.HappyFortuneService">
</bean>

<bean id="myCoach"
    class="com.luv2code.springdemo.TrackCoach">

    <!-- set up constructor injection -->
    <constructor-arg ref="myFortuneService" />
</bean>

<bean id="myCricketCoach"
    class="com.luv2code.springdemo.CricketCoach">

    <!-- set up setter injection -->
    <property name="fortuneService" ref="myFortuneService" />
```

Java Configuration Class

```
package com.luv2code.springdemo;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration

public class SportConfig {

}
```

Defining Beans in Spring

Full XML Config

```
<!-- define the dependency -->
<bean id="myFortuneService"
    class="com.luv2code.springdemo.HappyFortuneService">
</bean>

<bean id="myCoach"
    class="com.luv2code.springdemo.TrackCoach">

    <!-- set up constructor injection -->
    <constructor-arg ref="myFortuneService" />
</bean>

<bean id="myCricketCoach"
    class="com.luv2code.springdemo.CricketCoach">

    <!-- set up setter injection -->
    <property name="fortuneService" ref="myFortuneService" />
```

Java Configuration Class

```
package com.luv2code.springdemo;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration

public class SportConfig {

}
```

No XML!

Our New Coach …

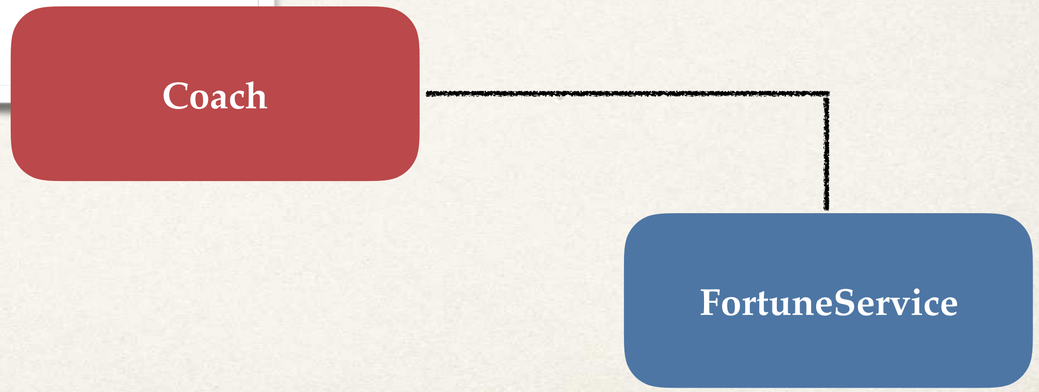
No special annotations

```
public class SwimCoach implements Coach {

  ...

}
```

Our New Coach …

No special annotations

```
public class SwimCoach implements Coach {

  ...

}
```



Development Process

Step-By-Step

Development Process

Step-By-Step

1. Define method to expose bean

Development Process

Step-By-Step

1. Define method to expose bean

2. Inject bean dependencies

Development Process

Step-By-Step

1. Define method to expose bean

2. Inject bean dependencies

3. Read Spring Java configuration class

Development Process

Step-By-Step

1. Define method to expose bean
2. Inject bean dependencies
3. Read Spring Java configuration class
4. Retrieve bean from Spring container

Step 1: Define method to expose bean

```
@Configuration
public class SportConfig {

  @Bean
  public Coach swimCoach() {
    SwimCoach mySwimCoach = new SwimCoach();

    return mySwimCoach;
  }

}
```

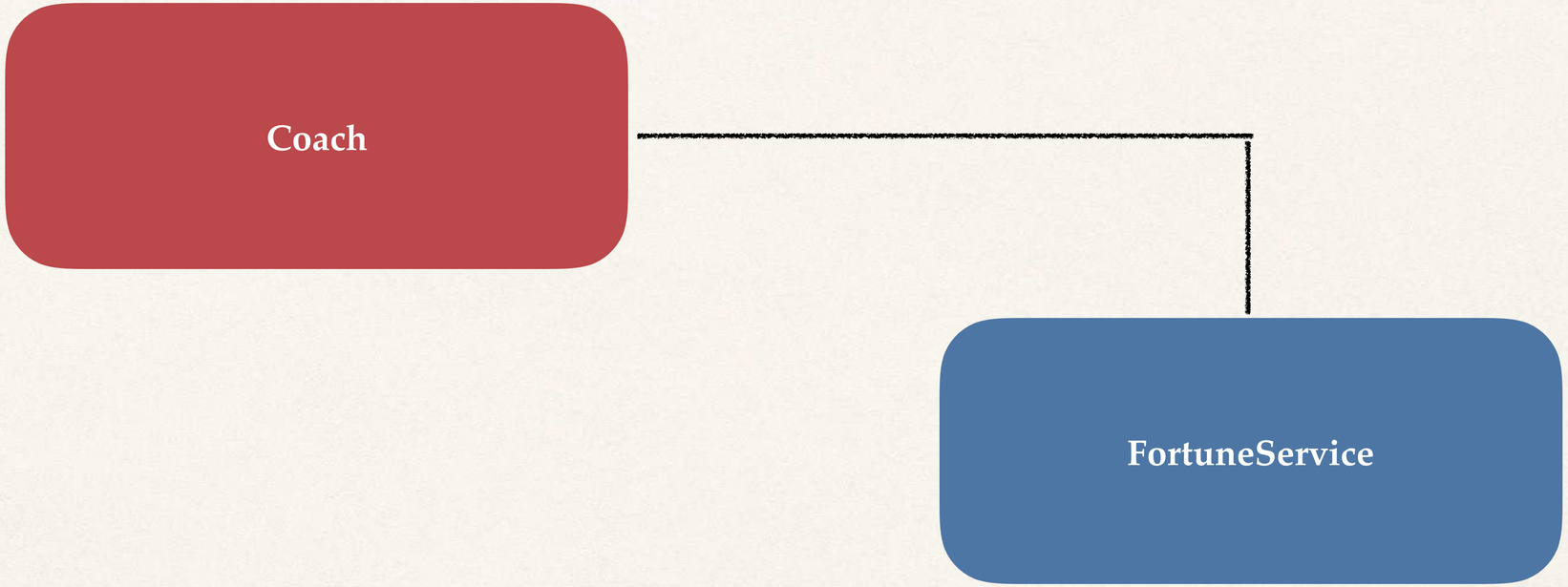
Step 1: Define method to expose bean

```
@Configuration
public class SportConfig {

  @Bean
  public Coach swimCoach() {
    SwimCoach mySwimCoach = new SwimCoach();

    return mySwimCoach;
  }

}
```

No component scan

What about our dependencies?

What about our dependencies?



Step 2: Inject bean dependencies

```
@Configuration
public class SportConfig {

    @Bean
    public FortuneService happyFortuneService() {
        return new HappyFortuneService();
    }

    @Bean
    public Coach swimCoach() {
        SwimCoach mySwimCoach = new SwimCoach( happyFortuneService() );

        return mySwimCoach;
    }

}
```

Step 3: Read Spring Java configuration class

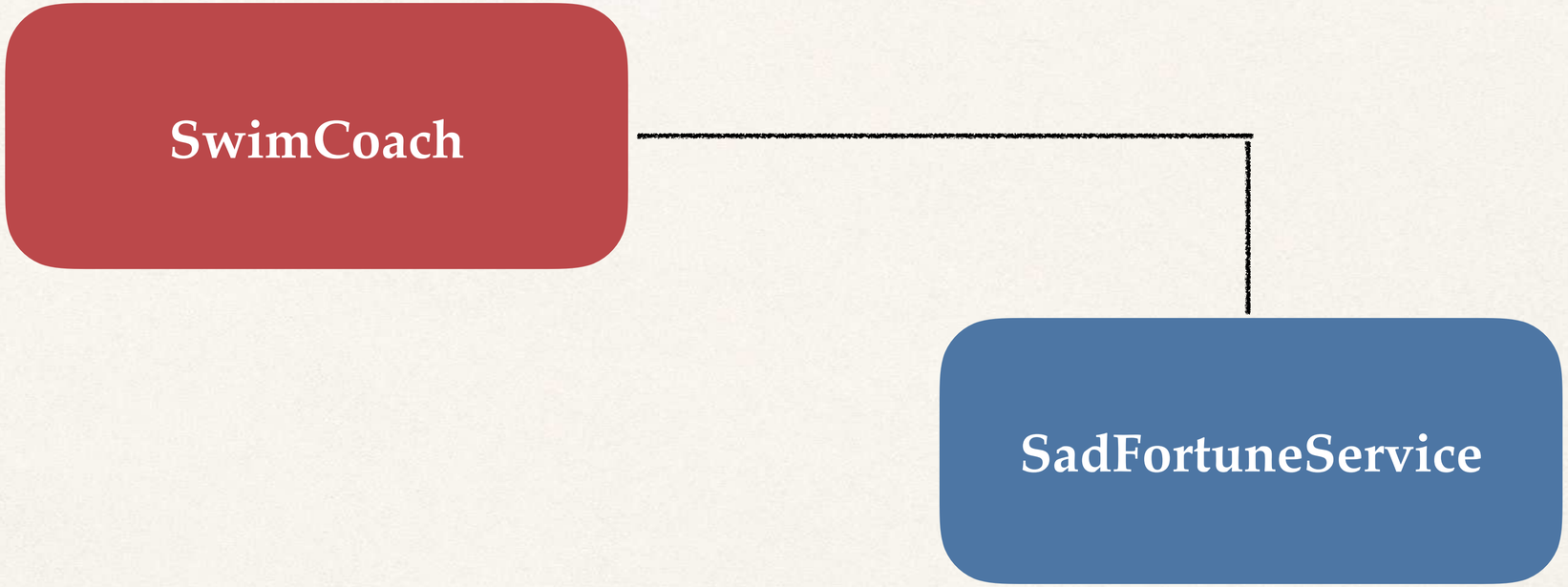
```
AnnotationConfigApplicationContext context =
        new AnnotationConfigApplicationContext(SportConfig.class);
```

Step 4: Retrieve bean from Spring container

```
Coach theCoach = context.getBean("swimCoach", Coach.class);
```

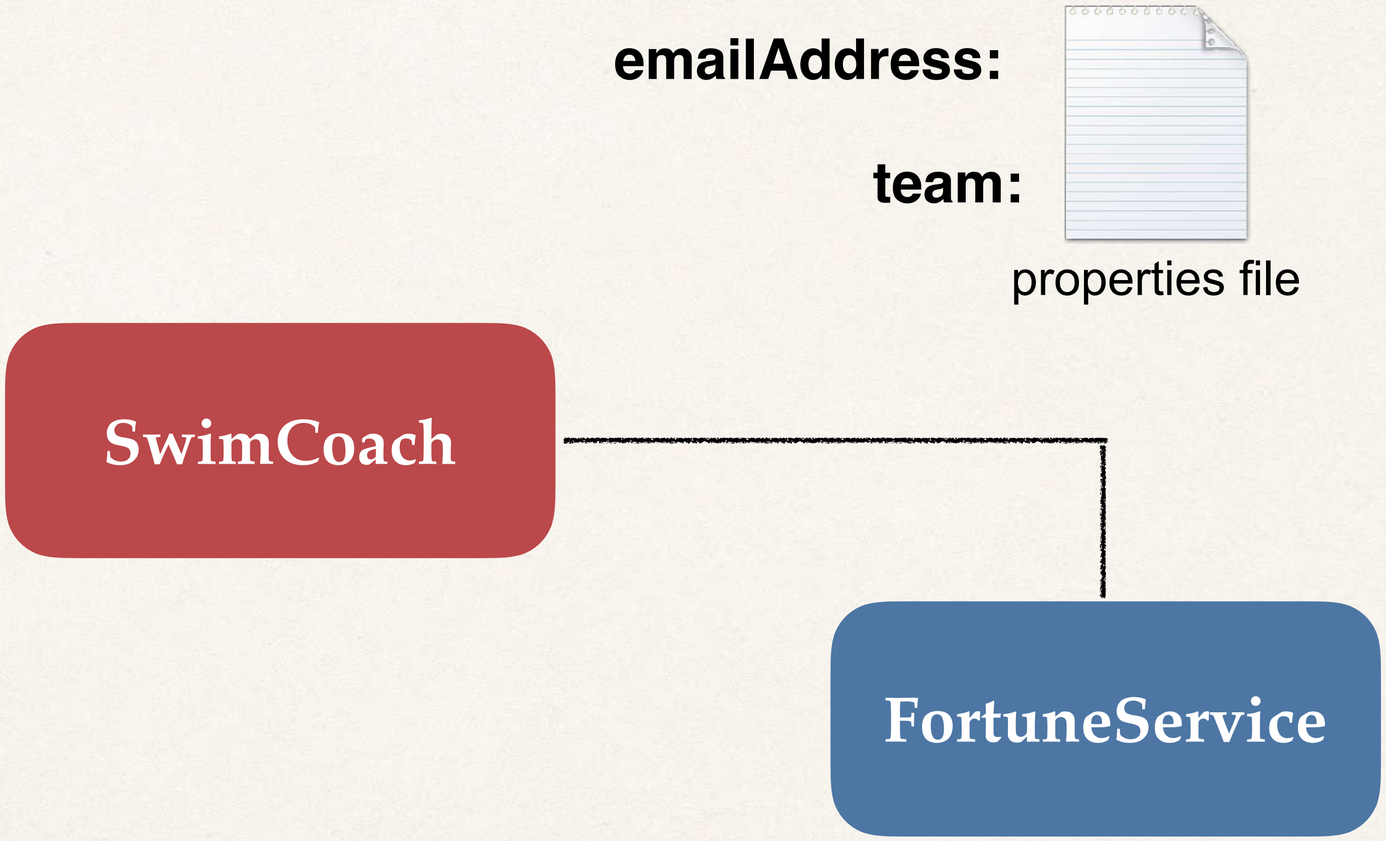
What about our dependencies?

What about our dependencies?



Injecting Values from Properties File

Read from a Properties File



Development Process

1. Create Properties File

2. Load Properties File in Spring config

3. Reference values from Properties File

Step-By-Step

Step 1: Create Properties File

File: sport.properties

```
foo.email=myeasycoach@luv2code.com
foo.team=Awesome Java Coders
```

Step 2: Load Properties file in Spring config

File: SportConfig.java

```
@Configuration
@PropertySource("classpath:sport.properties")
public class SportConfig {

  ...

}
```

Step 3: Reference Values from Properties File

File: SwimCoach.java

```
public class SwimCoach implements Coach {

   @Value("${foo.email}")
   private String email;

   @Value("${foo.team}")
   private String team;

   ...
}
```

```
foo.email=myeasycoach@luv2code.com
foo.team=Awesome Java Coders
```