@Components - Default Bean ID

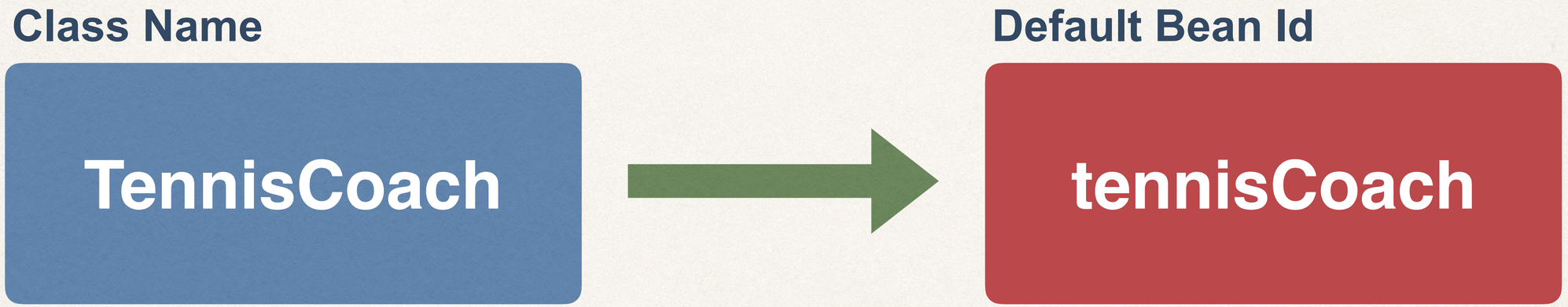
We already learned …

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*

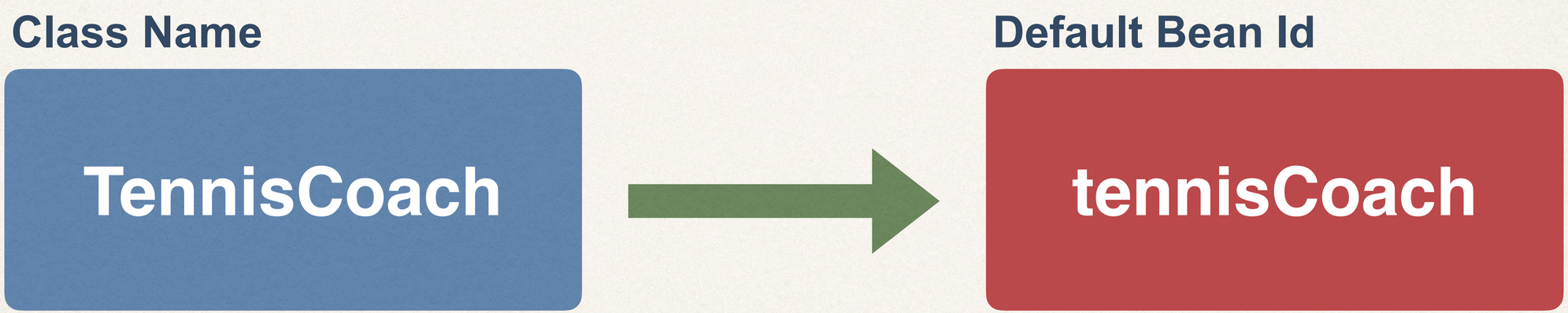


Code example

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

Code example

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



Code example

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

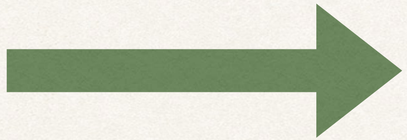
```
    // get the bean from spring container
    Coach theCoach = context.getBean("tennisCoach", Coach.class);
```

Code example

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

Class Name

HappyFortuneService



Default Bean Id

happyFortuneService

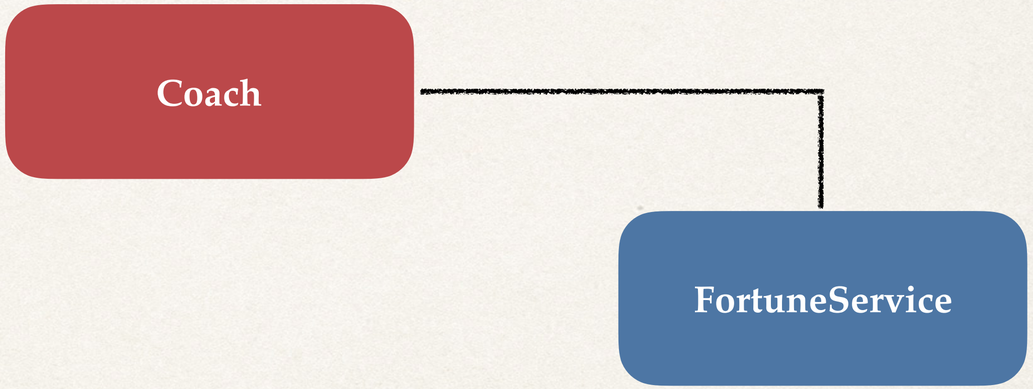
luv2code

Spring Dependency Injection with Annotations and Autowiring

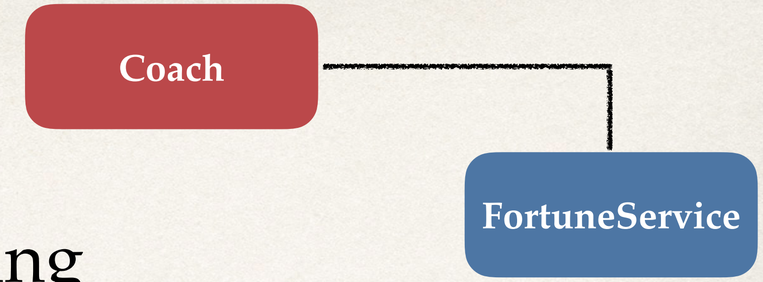
Demo Example

- Our **Coach** already provides daily workouts

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

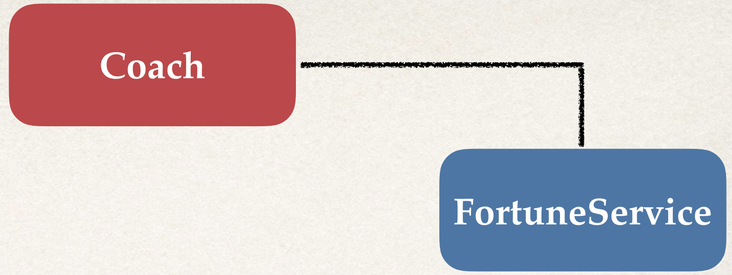


What is Spring AutoWiring?



- 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

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

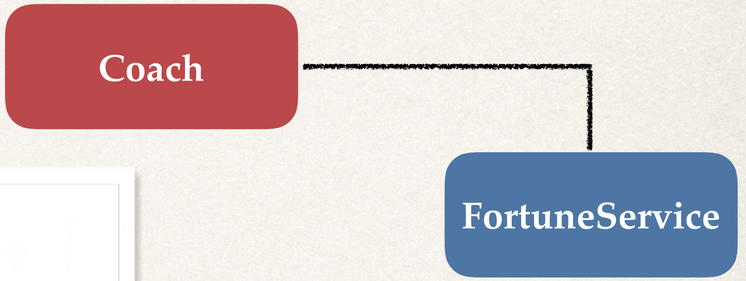
File: FortuneService.java

```
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

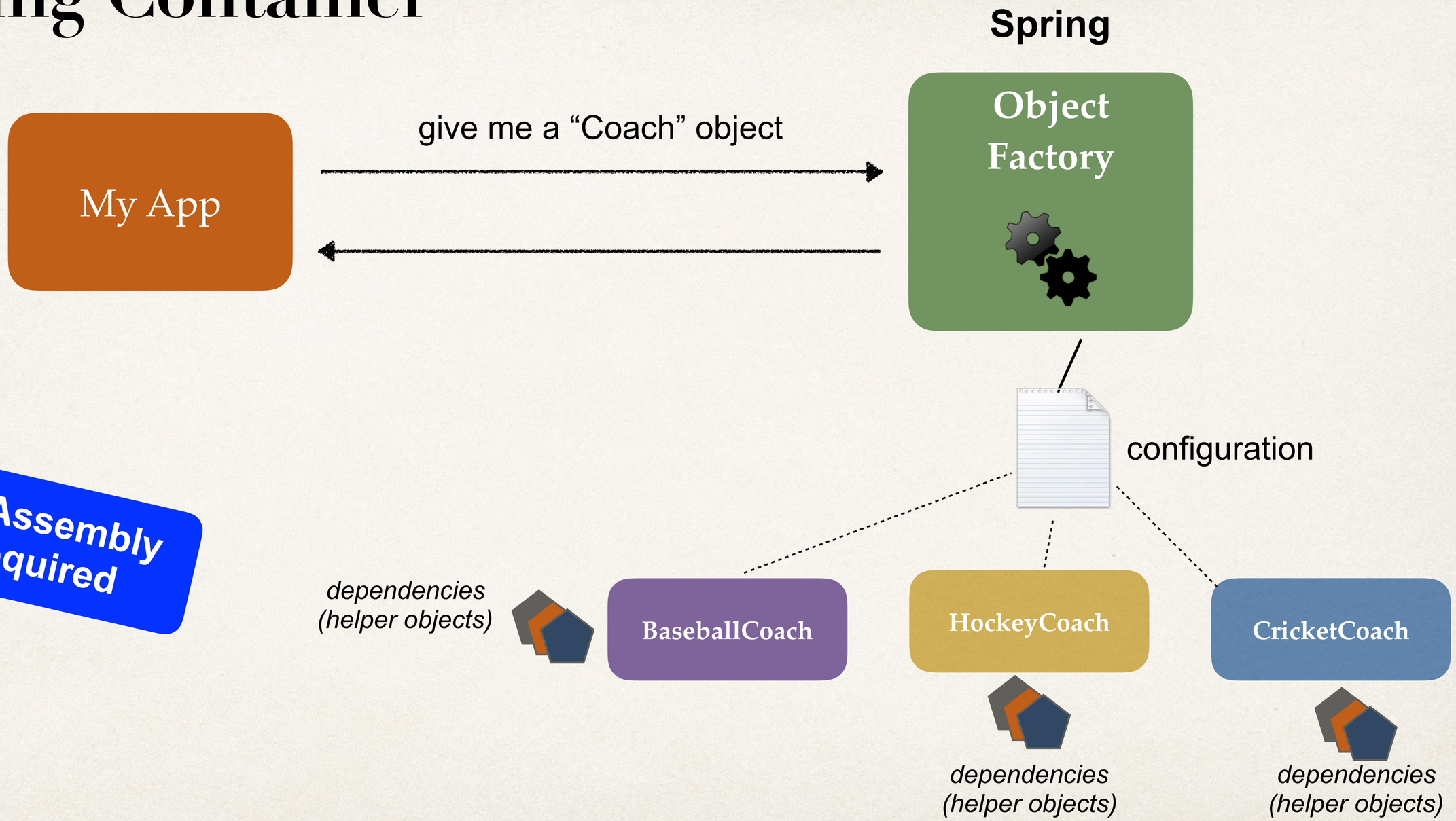
```
@Component
public class TennisCoach implements Coach {

   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



luv2code

Setter Injection with Annotations and Autowiring

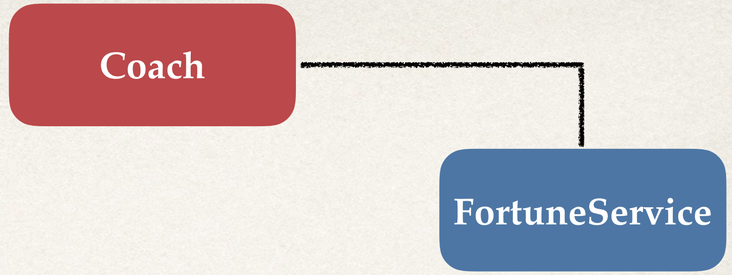
Spring Injection Types

- 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

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

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

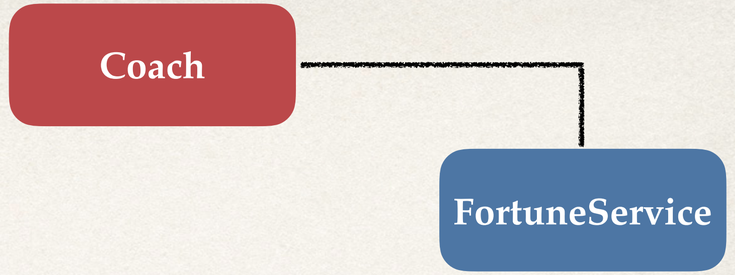
Choose a style

Stay consistent in your project

luv2code

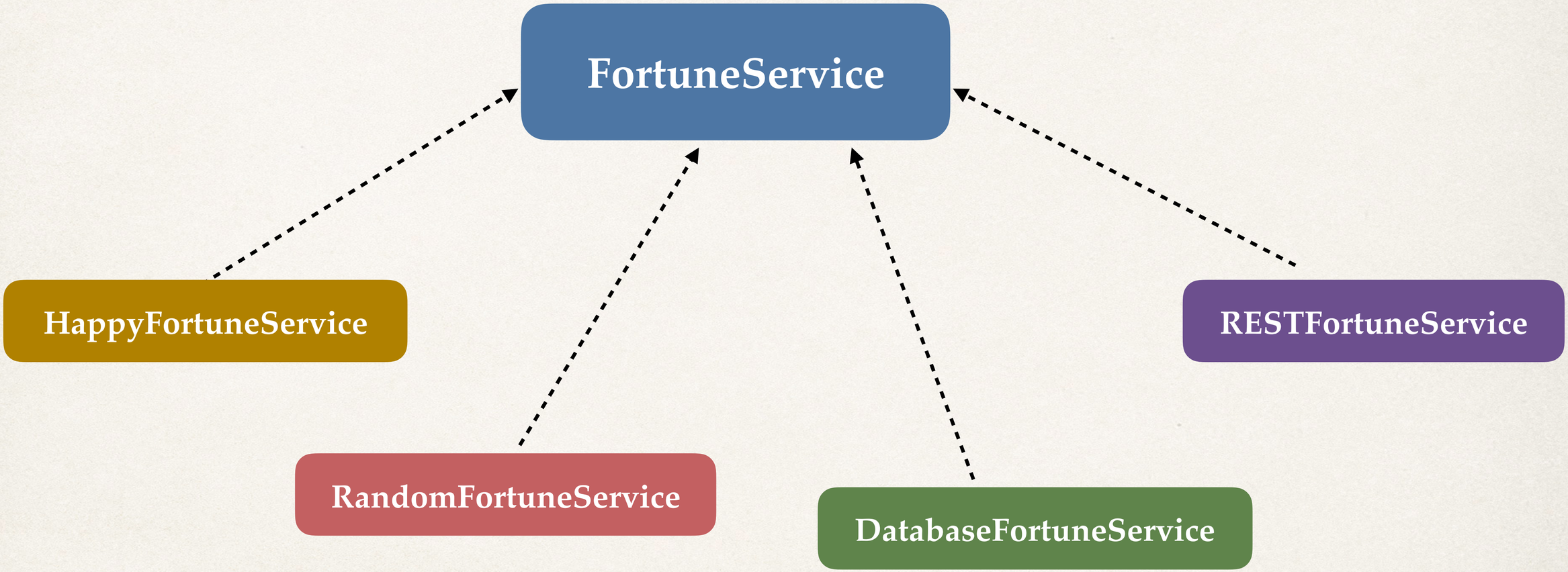
Annotation Autowiring and Qualifiers

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

luv2code

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

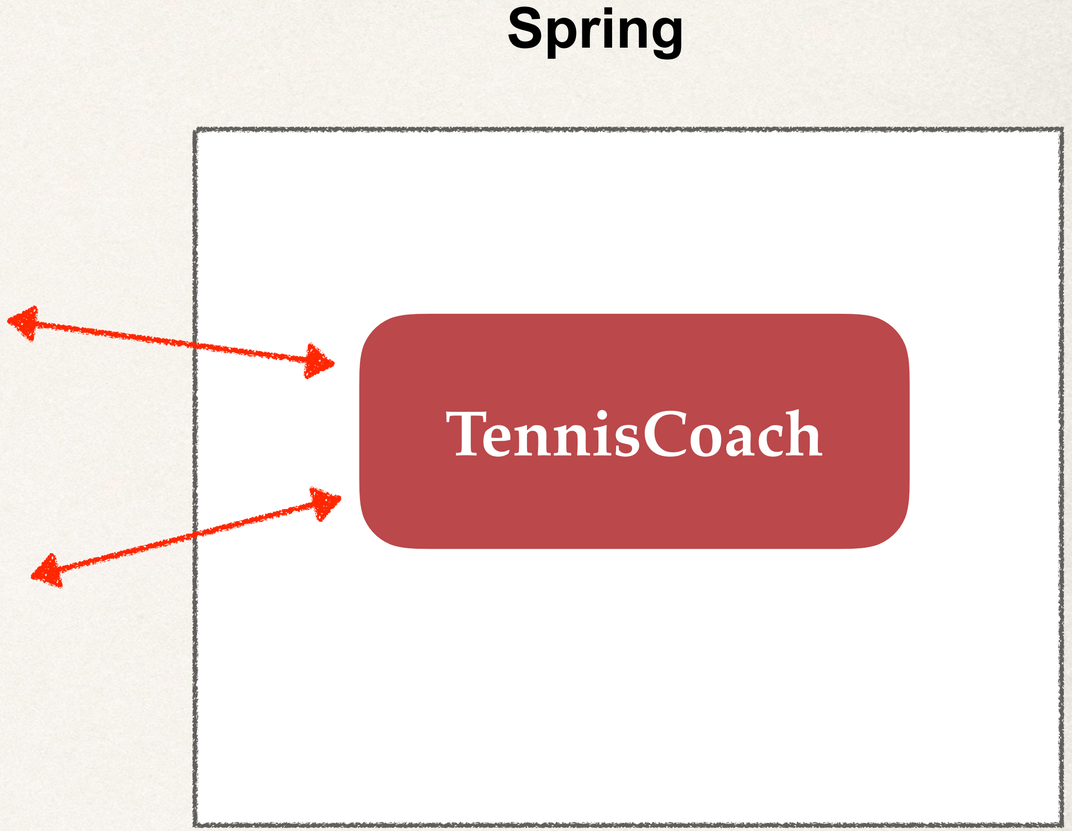
Default scope is singleton

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

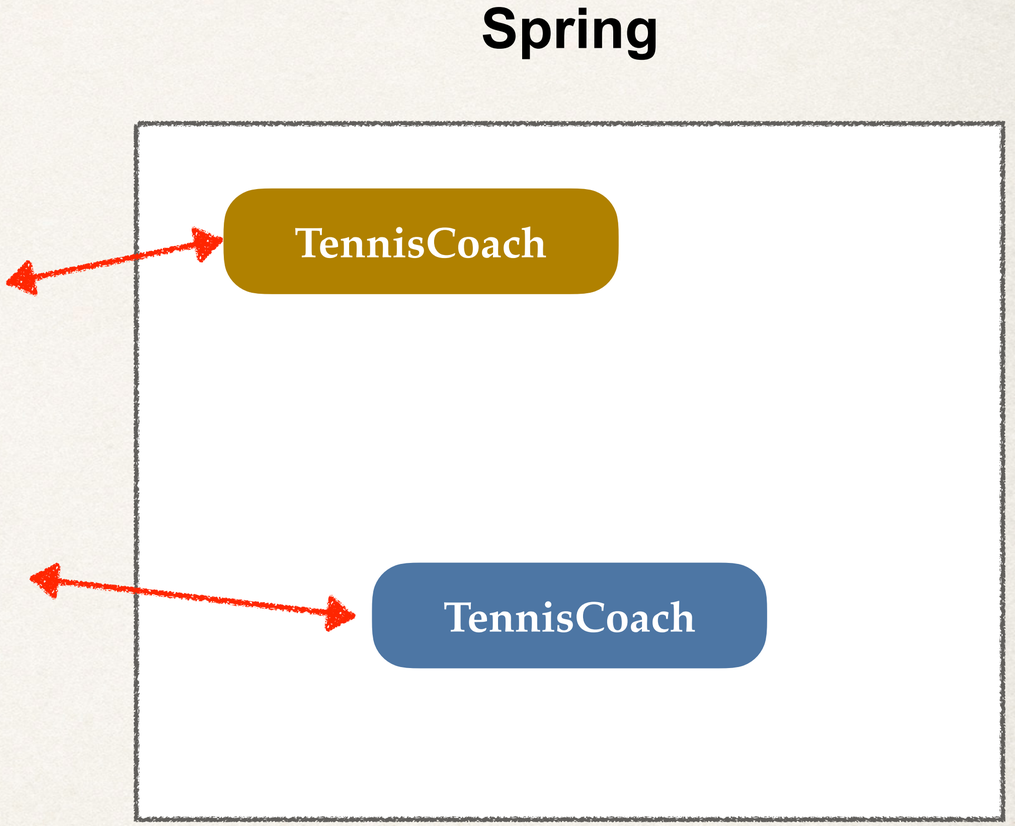
Prototype scope: new object for each request

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

...

}
```

Prototype Scope Example

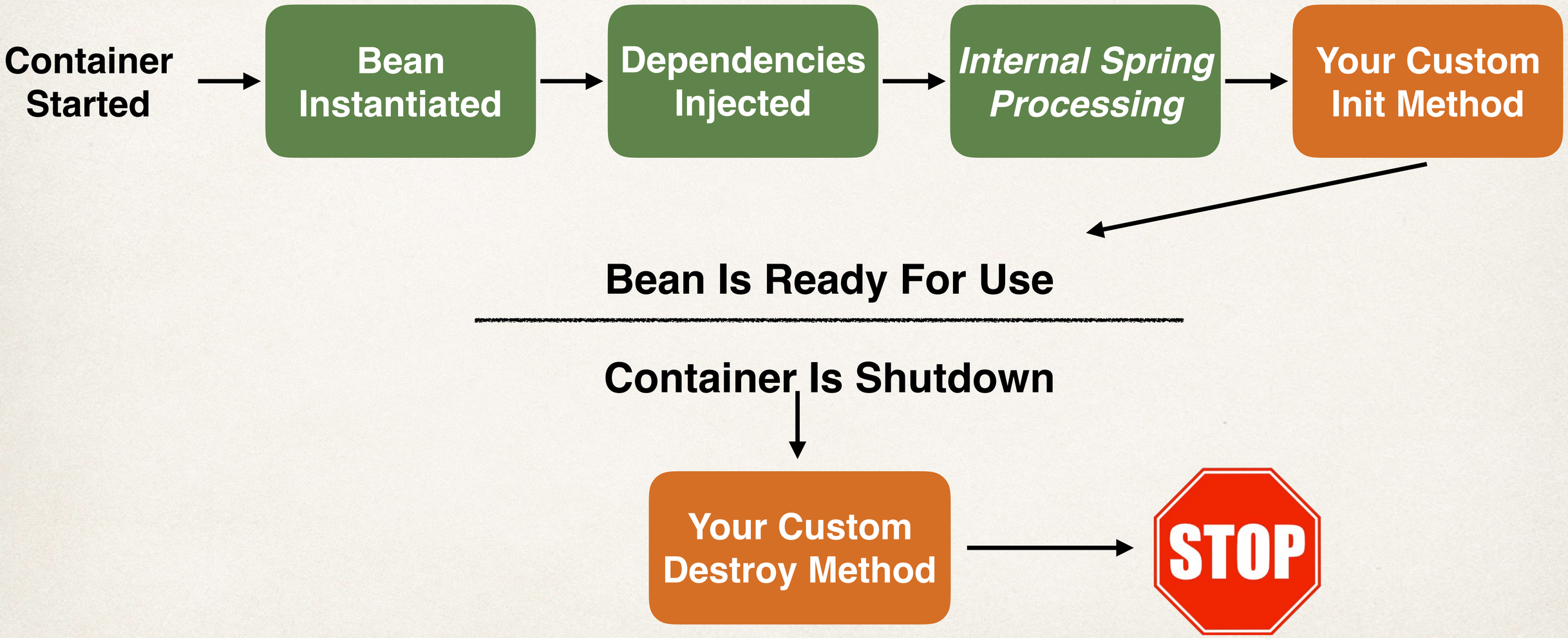
Prototype scope: new object for each request

```
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

1. Define your methods for init and destroy

2. Add annotations: @PostConstruct and @PreDestroy

luv2code

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);
```

luv2code

Defining Beans with Java Code

Defining Beans in Spring

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" />
```

Java Configuration Class

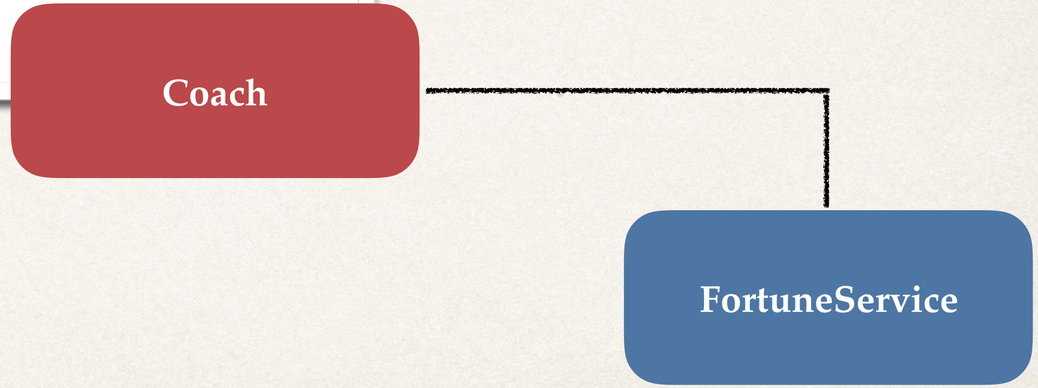
```java
package com.luv2code.springdemo;

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 {

 ...

}
```



Development Process

Step-By-Step

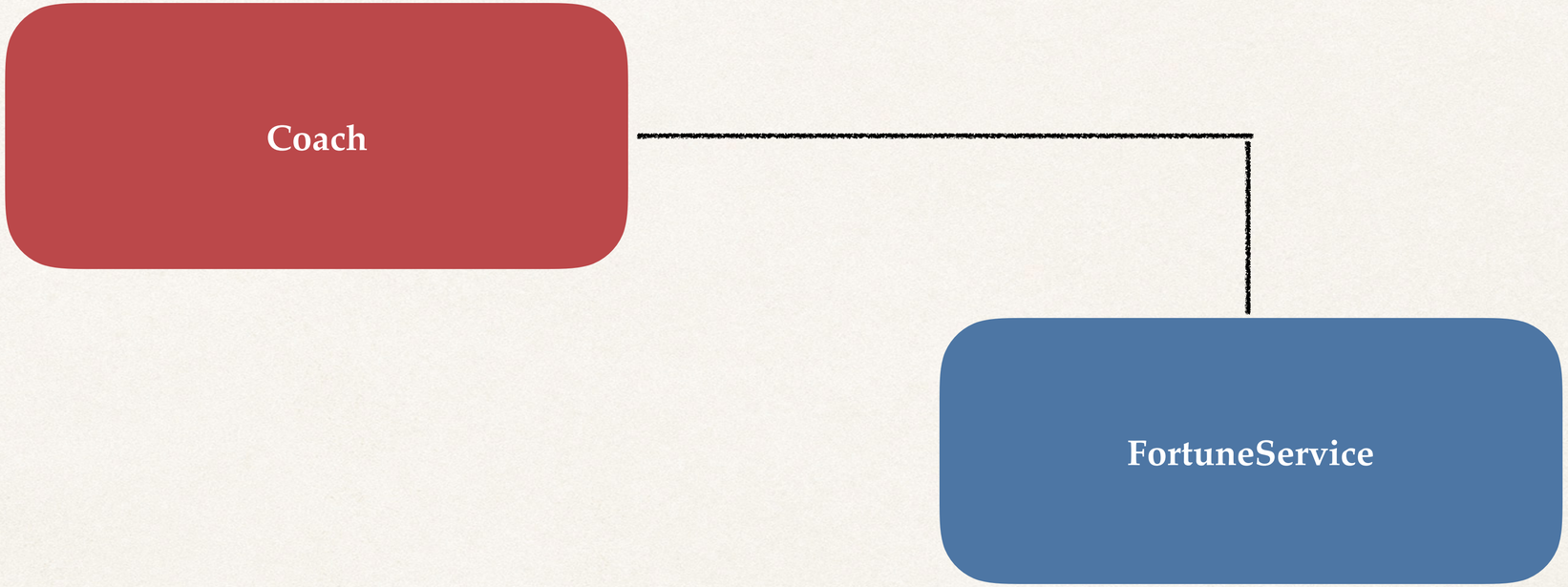
1. Define method to expose bean
2. Inject bean dependencies
3. Read Spring Java configuration class
4. Retrieve bean from Spring container

Step 1: Define method to expose bean

```
@Configuration
public class SportConfig {

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

      return mySwimCoach;
   }

}
```

No component scan

What about our dependencies?



Step 2: Inject bean dependencies

```
@Configuration
public class SportConfig {

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

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

      return mySwimCoach;
   }

}
```

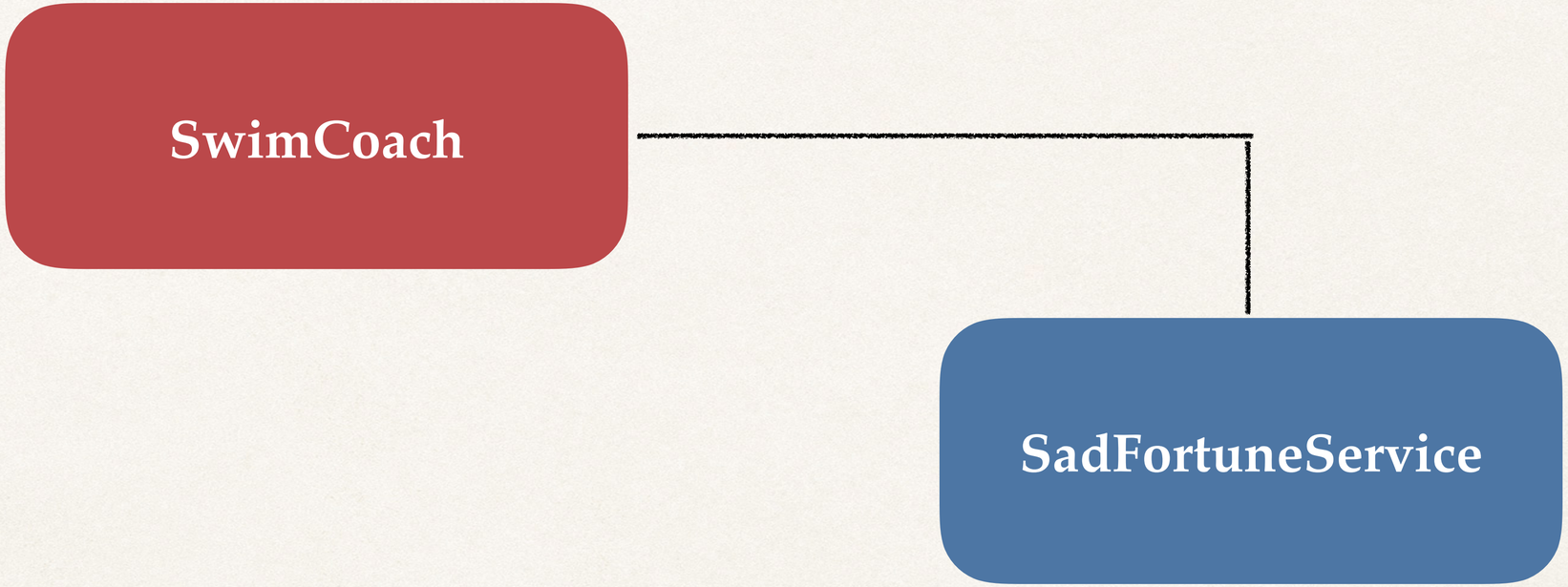
Step 3: Read Spring Java configuration class

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

Step 4: Retrieve bean from Spring container

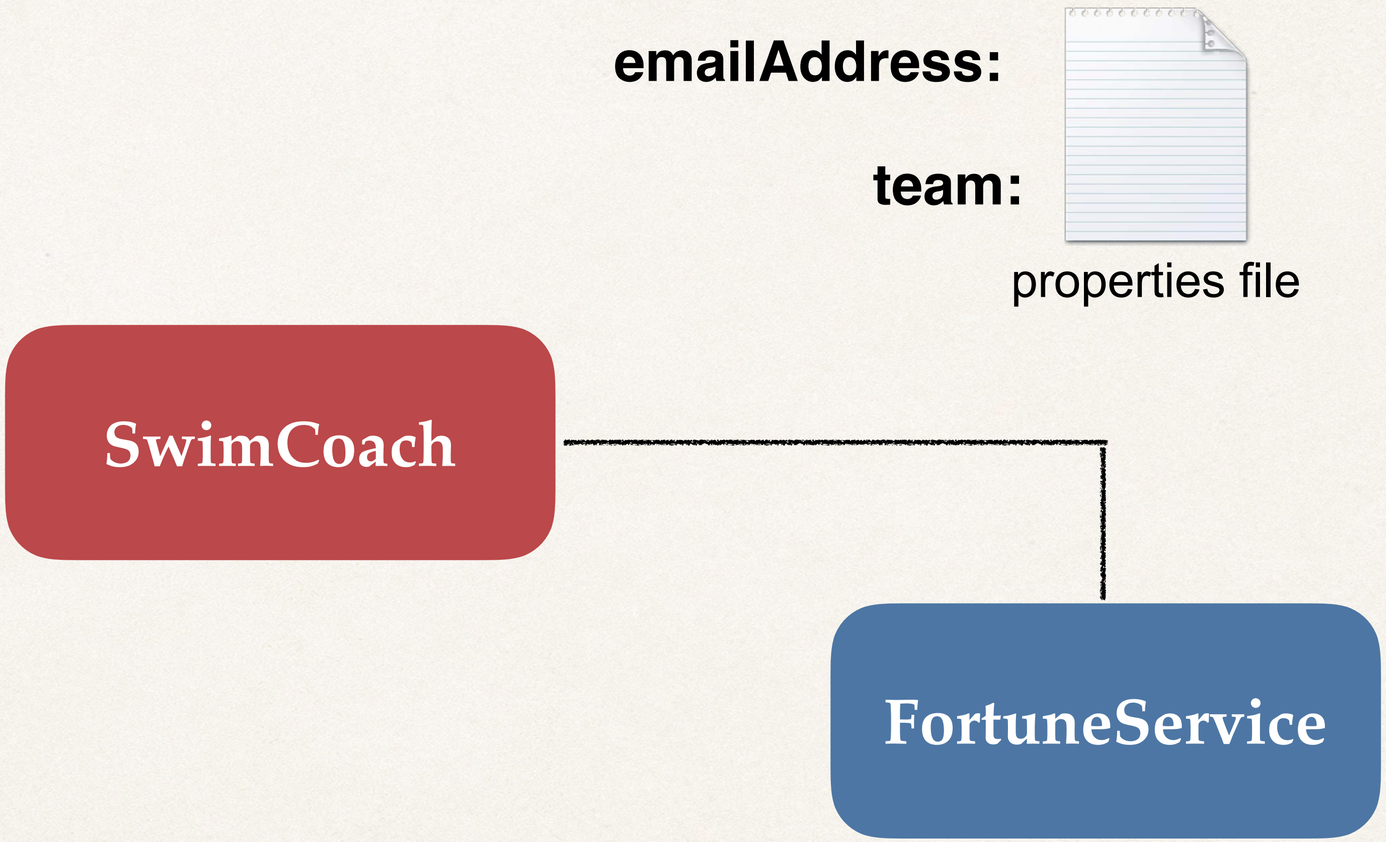
```
Coach theCoach = context.getBean("swimCoach", Coach.class);
```

What about our dependencies?



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 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
```