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?

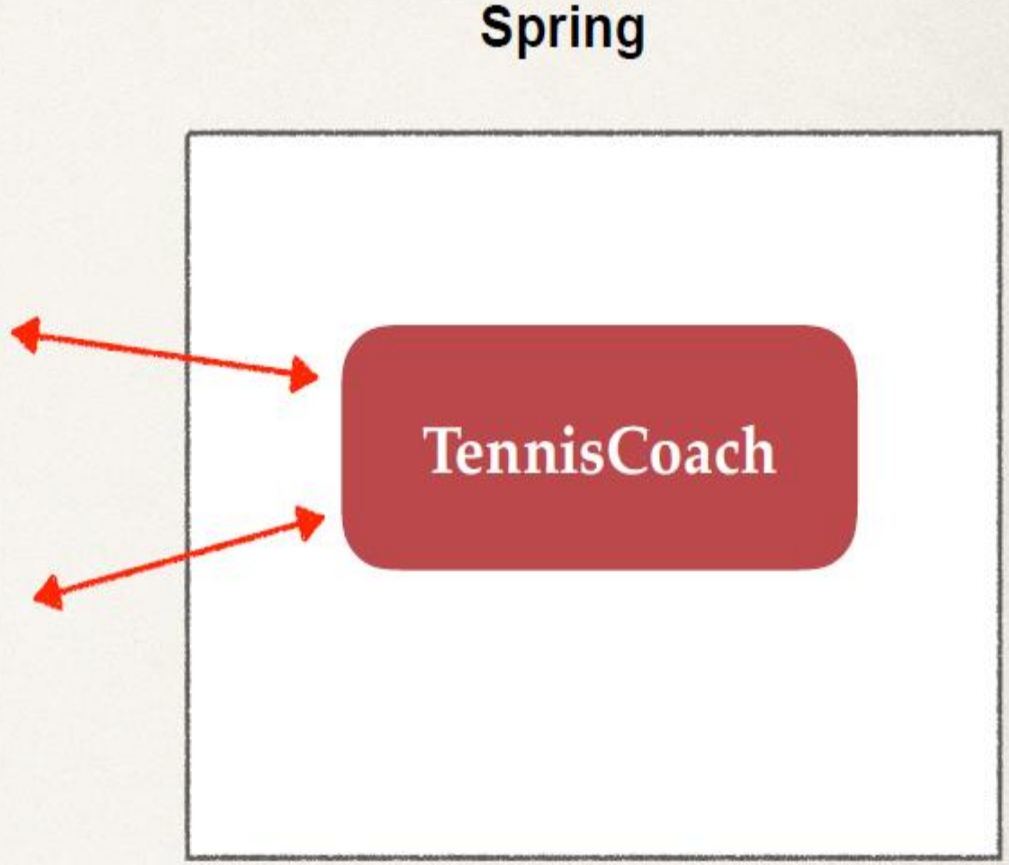
- Spring Container creates only one instance of the bean, by default
- It is cached in memory
- All requests for the bean
  - will return a SHARED reference to the SAME bean

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

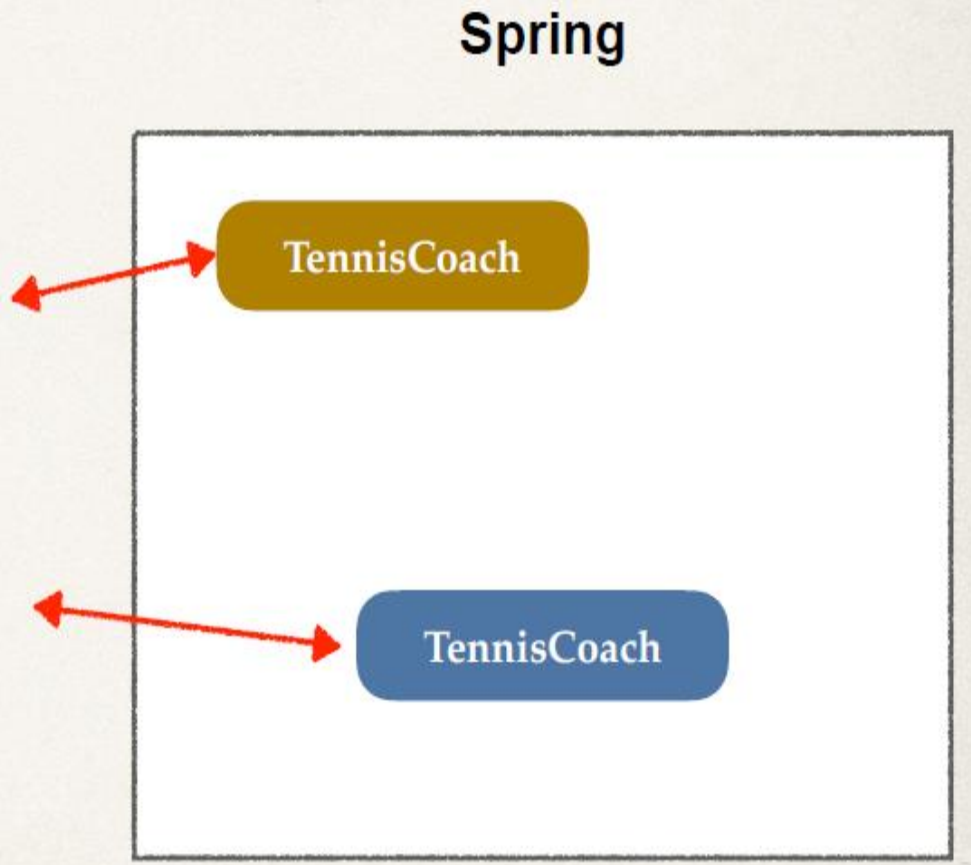
Prototype scope: new object for each request

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

...

}
```

Prototype Scope Example

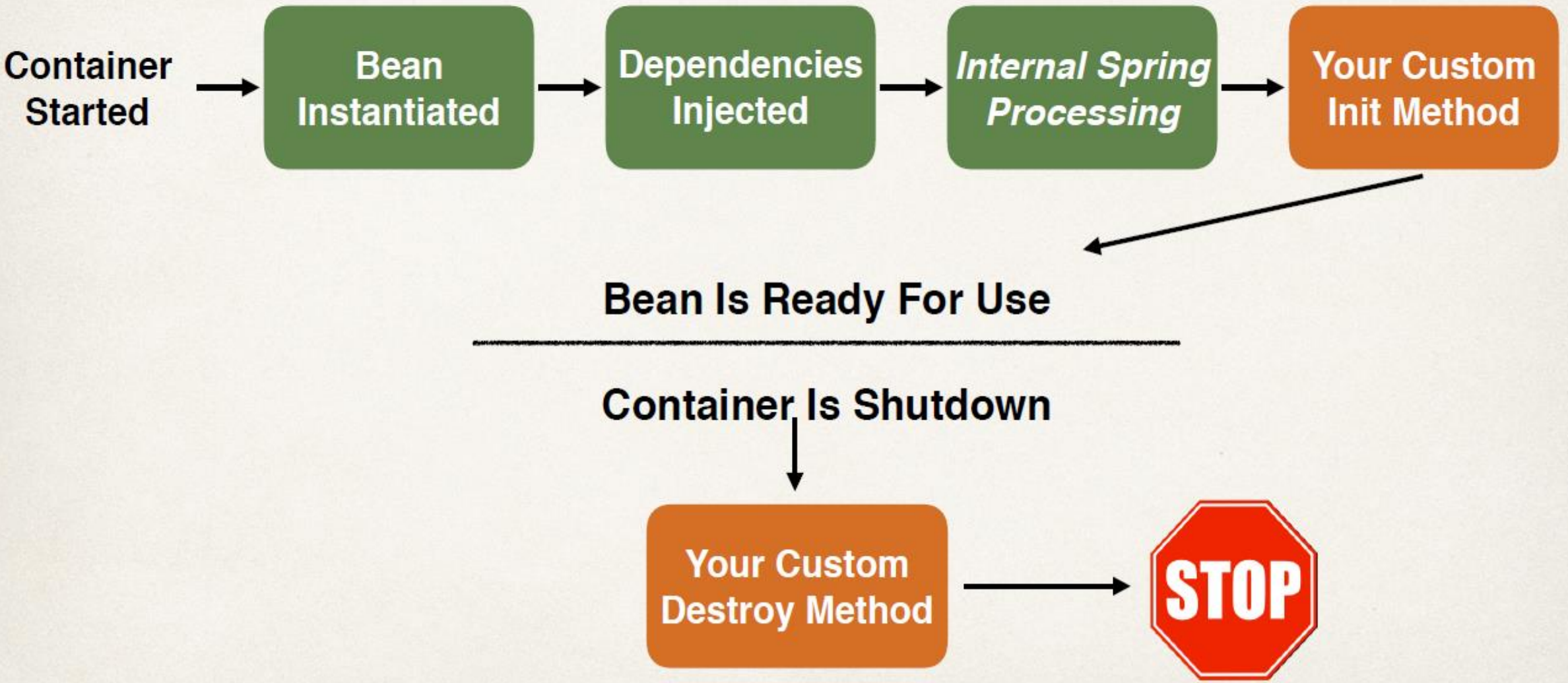
Prototype scope: new object for each request

```
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() { … }

 …

}
```

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

Spring Configuration with Java Code

Java Configuration

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

No XML!

Development Process

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 3: Read Spring Java configuration class

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

Step 4: Retrieve bean from Spring container

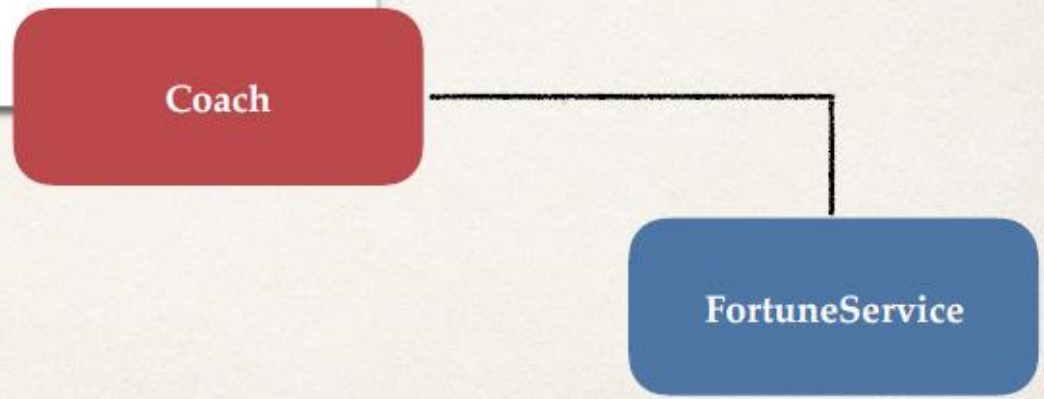
```
Coach theCoach = context.getBean("tennisCoach", Coach.class);
```

Defining Beans with Java Code

Our New Coach …

No special annotations

```
public class SwimCoach implements Coach {

 ...

}
```



Development Process

Step-By-Step

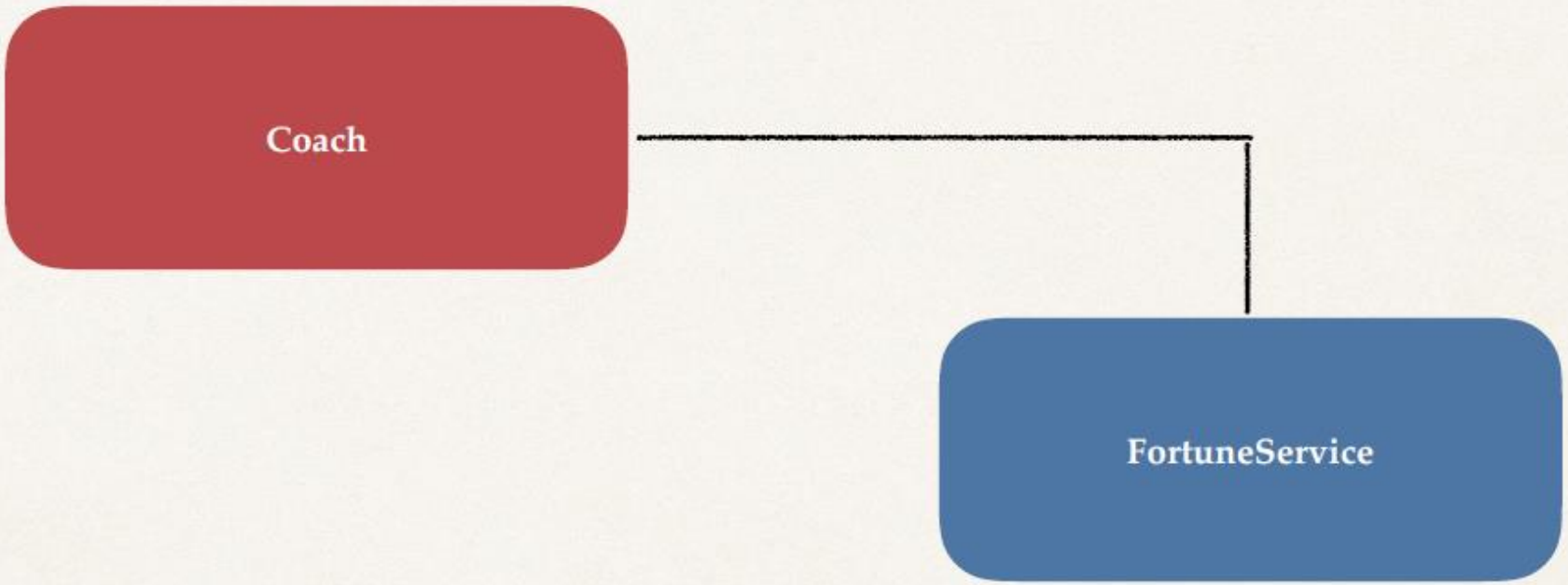
1. Define method to expose bean
2. Inject bean dependencies
3. Read Spring Java configuration class
4. Retrieve bean from Spring container

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;
   }

}
```

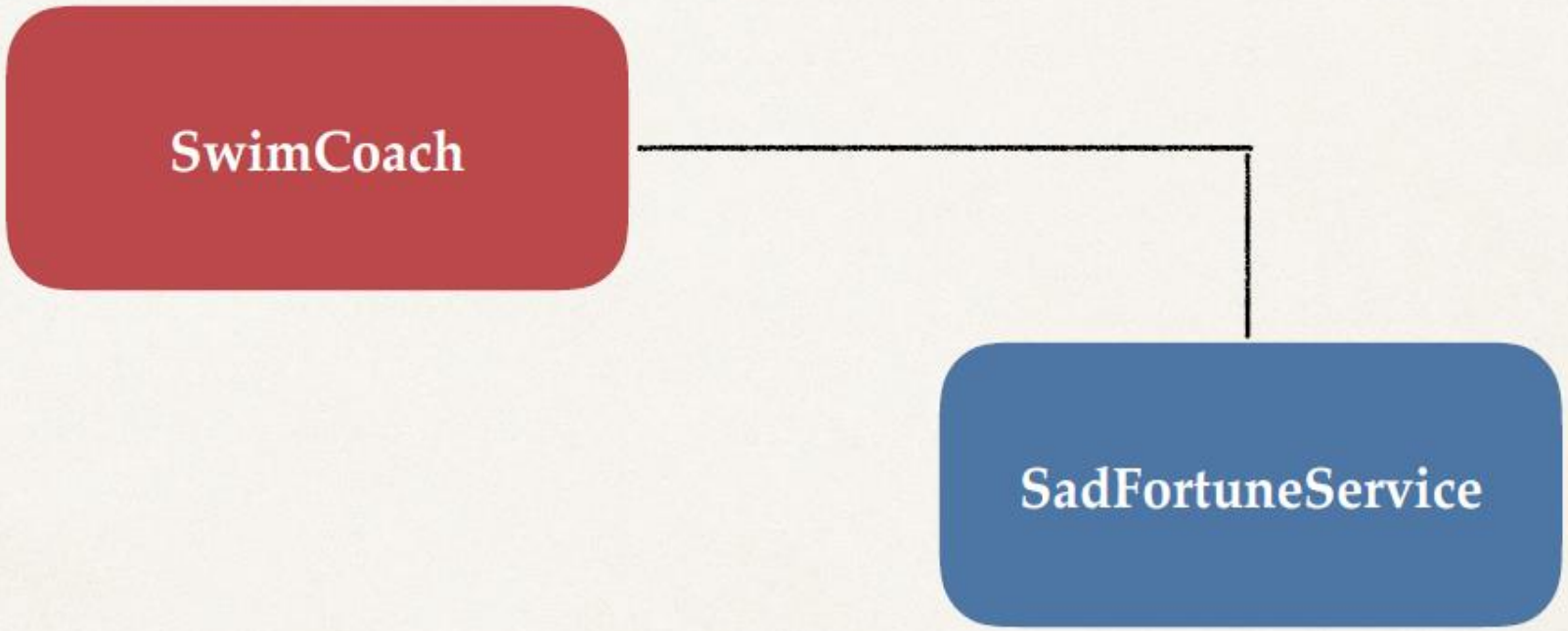
Step 3: Read Spring Java configuration class

```
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

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@coach.com
foo.team=Royal Challengers Bangalore
```

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
```

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
```