Inversion of Control

Inversion of Control (IoC)

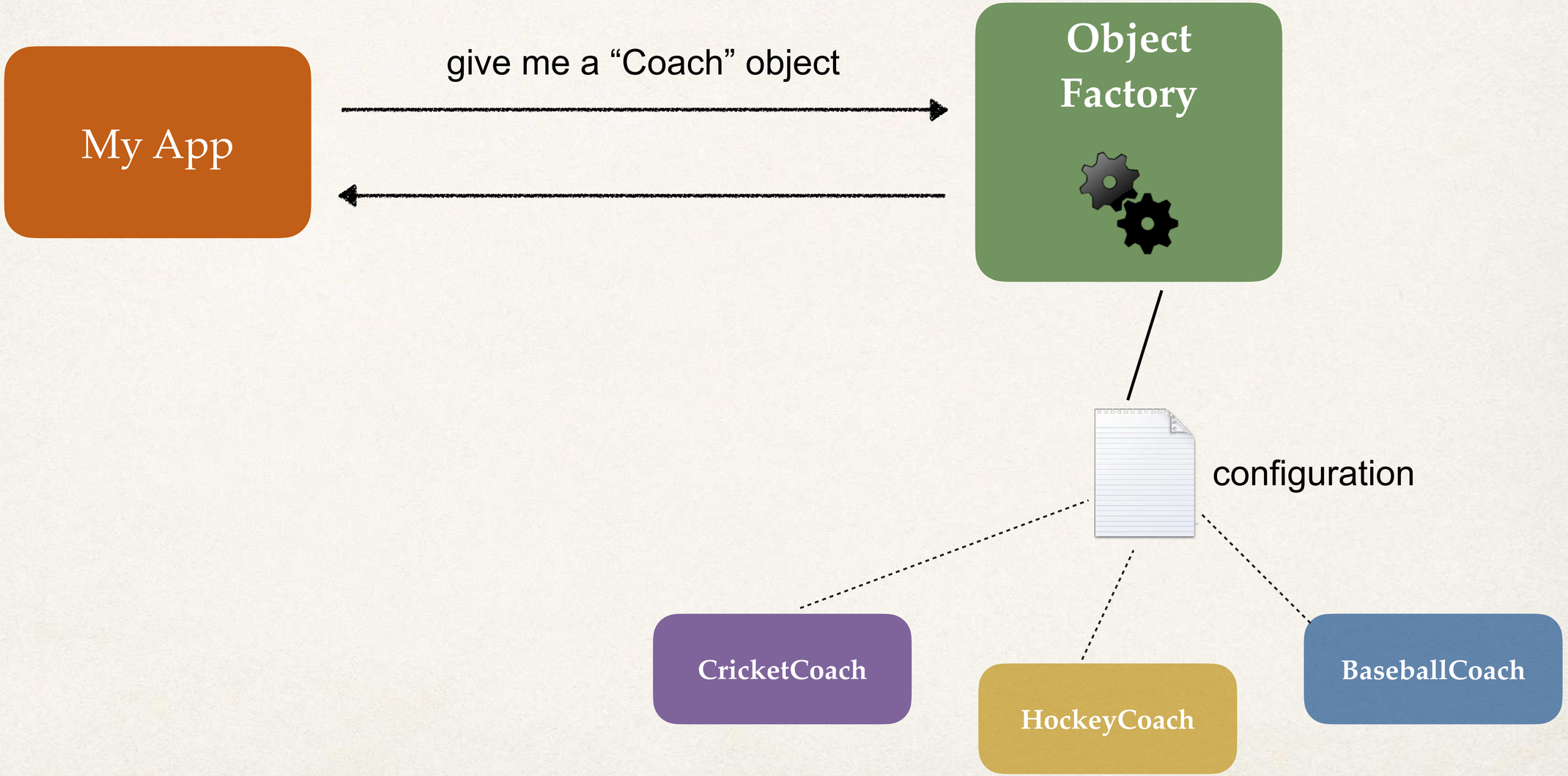
The approach of outsourcing the construction and management of objects.

Coding Scenario

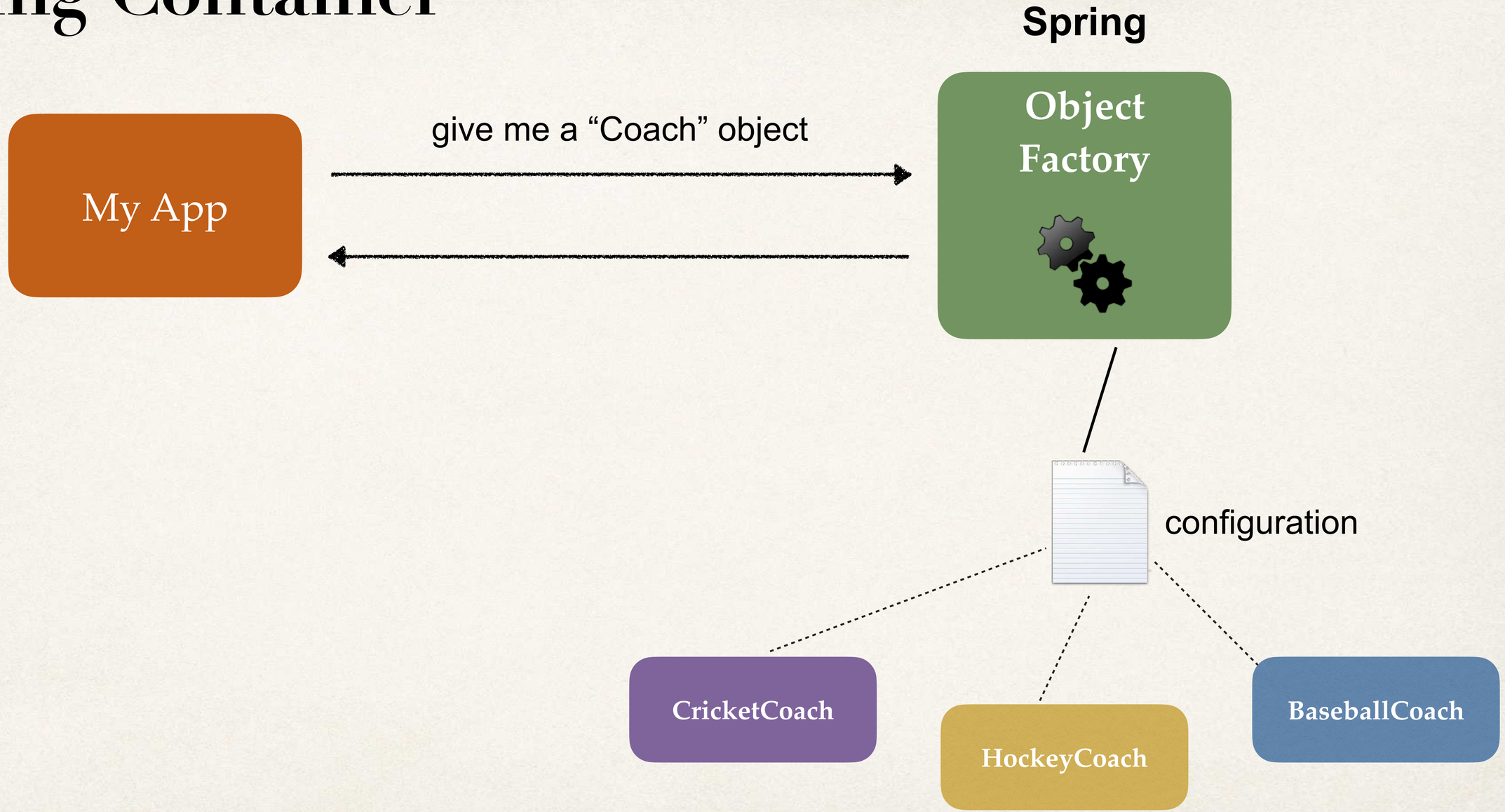
- App should be configurable

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

Ideal Solution



Spring Container



Spring Container

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

Configuring Spring Container

- XML configuration file *(legacy)* 
- Java Annotations *(modern)* 
- Java Source Code *(modern)* 

Spring Dependency Injection

Dependency Injection

The dependency inversion principle.

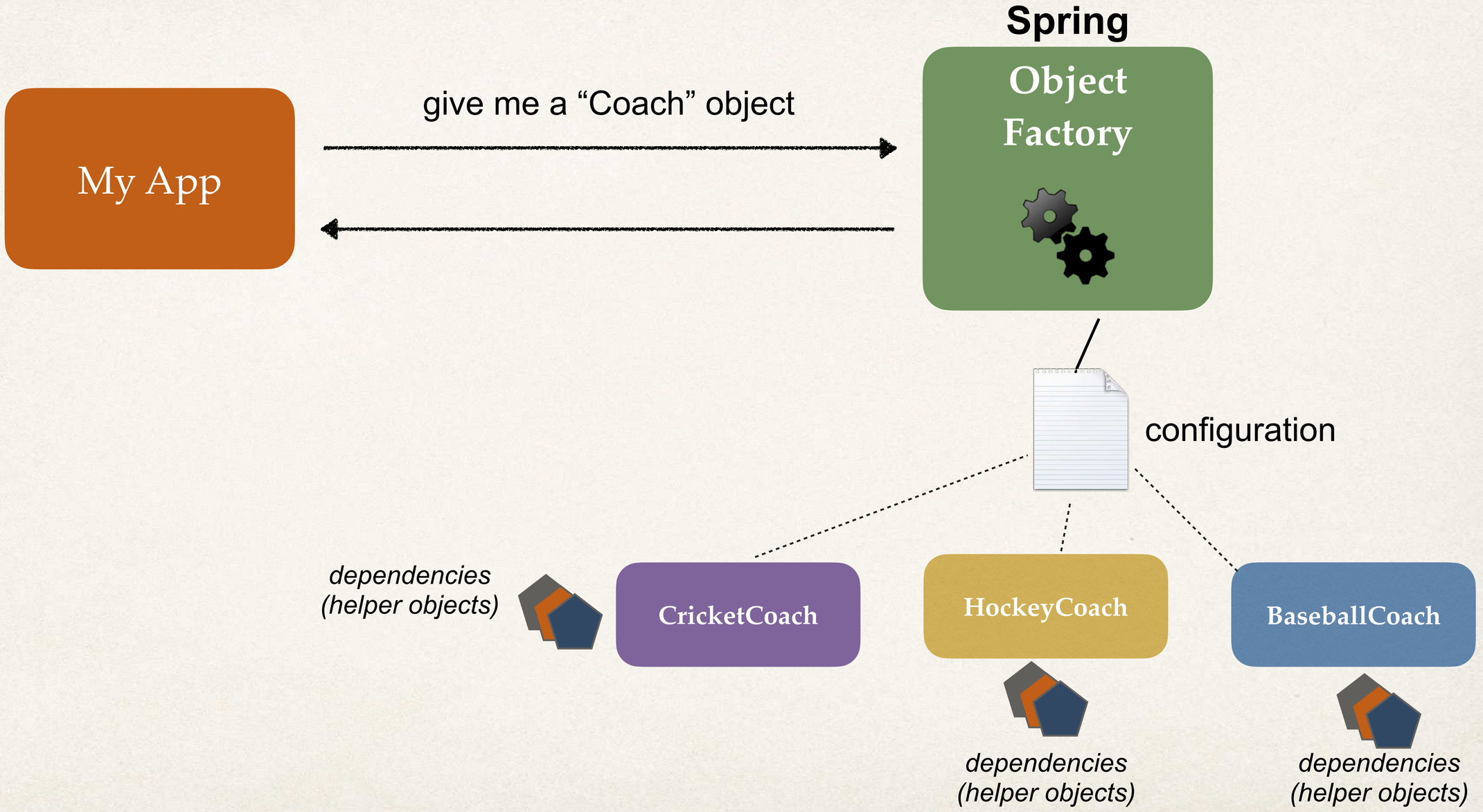
The client delegates to another object the responsibility of providing its dependencies.

Car Factory

give me a “Car” object

Car Factory

Spring Container



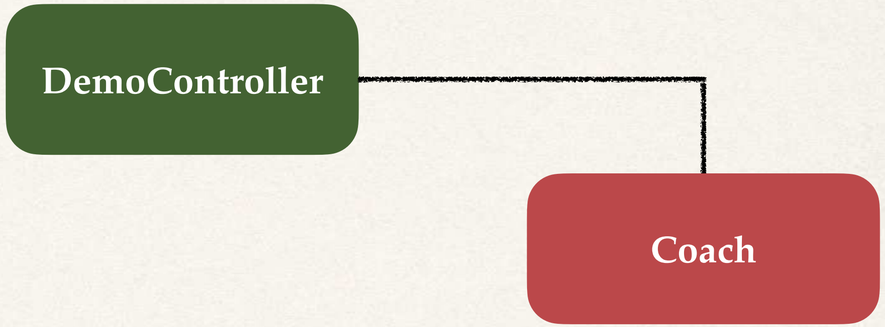
Spring Container

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

Demo Example

- **Coach** will provide daily workouts

- The DemoController wants to use a Coach
  - New helper: **Coach**
  - This is a ***dependency***



- Need to *inject* this *dependency*

Injection Types

- There are multiple types of injection with Spring

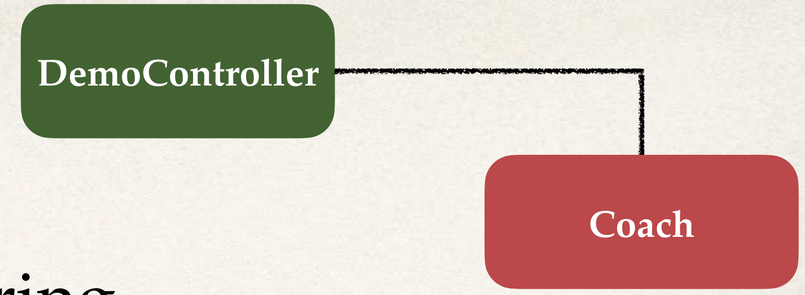
- We will cover the two recommended types of injection
  - Constructor Injection
  - Setter Injection

Injection Types - Which one to use?

- Constructor Injection
  - Use this when you have required dependencies
  - Generally recommended by the spring.io development team as first choice

- Setter Injection
  - Use this when you have optional dependencies
  - If dependency is not provided, your app can provide reasonable default logic

What is Spring AutoWiring



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

Autowiring Example

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

Example Application

Development Process - Constructor Injection

1. Define the dependency interface and class

2. Create Demo REST Controller

3. Create a constructor in your class for injections

4. Add @GetMapping for /dailyworkout

Step 1: Define the dependency interface and class

File: Coach.java

```
package com.luv2code.springcoredemo;

public interface Coach {

    String getDailyWorkout();
}
```

@Component annotation marks the class as a Spring Bean

File: CricketCoach.java

```
package com.luv2code.springcoredemo;

import org.springframework.stereotype.Component;

@Component
public class CricketCoach implements Coach {

    @Override
    public String getDailyWorkout() {
        return "Practice fast bowling for 15 minutes";
    }
}
```

@Component annotation

- @Component marks the class as a Spring Bean
  - A Spring Bean is just a regular Java class that is managed by Spring
- @Component also makes the bean available for dependency injection

Step 2: Create Demo REST Controller

File: DemoController.java

```
package com.luv2code.springcoredemo;

import org.springframework.web.bind.annotation.RestController;

@RestController
public class DemoController {


}
```

Step 3: Create a constructor in your class for injections

File: DemoController.java

```
package com.luv2code.springcoredemo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public DemoController(Coach theCoach) {
        myCoach = theCoach;
    }

}
```

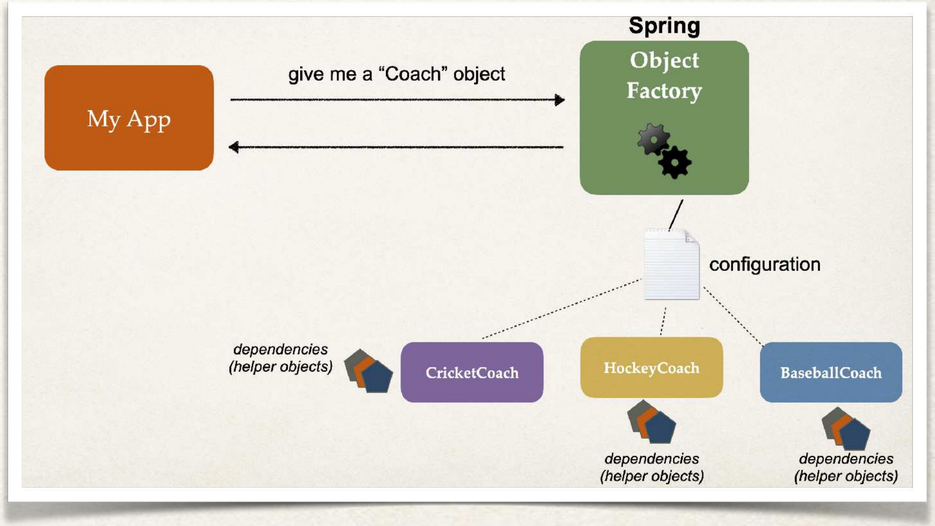
@Autowired annotation tells Spring to inject a dependency

If you only have one constructor then @Autowired on constructor is optional

At the moment, we only have one Coach implementation CricketCoach.

Spring can figure this out.

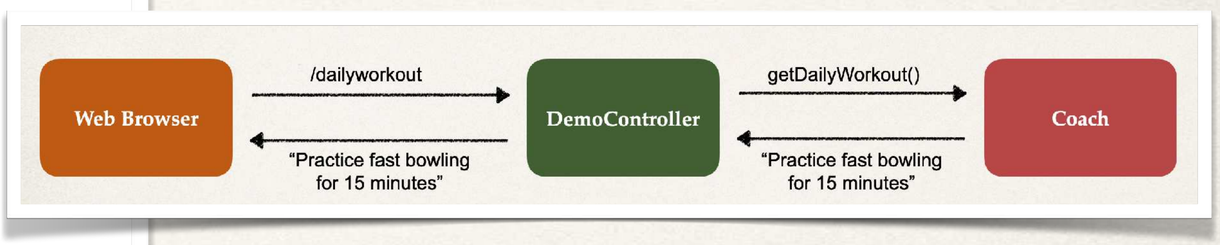
Later in the course we will cover the case of multiple Coach implementations.



Step 4: Add @GetMapping for /dailyworkout

File: DemoController.java

```
package com.luv2code.springcoredemo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public DemoController(Coach theCoach) {
        myCoach = theCoach;
    }

    @GetMapping("/dailyworkout")
    public String getDailyWorkout() {
        return myCoach.getDailyWorkout();
    }
}
```



Constructor Injection
Behind the Scenes

How Spring Processes your application

File: Coach.java

```
package com.luv2code.springcoredemo;

public interface Coach {

    String getDailyWorkout();
}
```

File: CricketCoach.java

```
package com.luv2code.springcoredemo;
…

@Component
public class CricketCoach implements Coach {

    @Override
    public String getDailyWorkout() {
        return "Practice fast bowling for 15 minutes";
    }
}
```

File: DemoController.java

```
package com.luv2code.springcoredemo;
…

@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public DemoController(Coach theCoach) {
        myCoach = theCoach;
    }
     …
}
```

The Spring Framework will perform operations

behind the scenes for you :-)

How Spring Processes your application

File: Coach.java

```
package com.luv2code.springcoredemo;

public interface Coach {

    String getDailyWorkout();
}
```

File: CricketCoach.java

```
package com.luv2code.springcoredemo;
…

@Component
public class CricketCoach implements Coach {

    @Override
    public String getDailyWorkout() {
        return "Practice fast bowling for 15 minutes";
    }
}
```

File: DemoController.java

```
package com.luv2code.springcoredemo;
…

@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public DemoController(Coach theCoach) {
        myCoach = theCoach;
    }
     …
}
```

Spring Framework

```
Coach theCoach = new CricketCoach();

DemoController demoController = new DemoController(theCoach);
```

Constructor injection

The “new” keyword … is that it???

- You may wonder …
  - *Is it just the “new” keyword???*
  - *I don’t need Spring for this … I can do this by myself LOL!!!*

- Spring is more than just Inversion of Control and Dependency Injection
- For small basic apps, it may be hard to see the benefits of Spring

Spring for Enterprise applications

- Spring is targeted for enterprise, real-time / real-world applications
- Spring provides features such as
  - Database access and Transactions
  - REST APIs and Web MVC
  - Security
  - etc …

Later in the course,
we will build a real-time CRUD REST API
with database access.

You will see the Spring features in action.

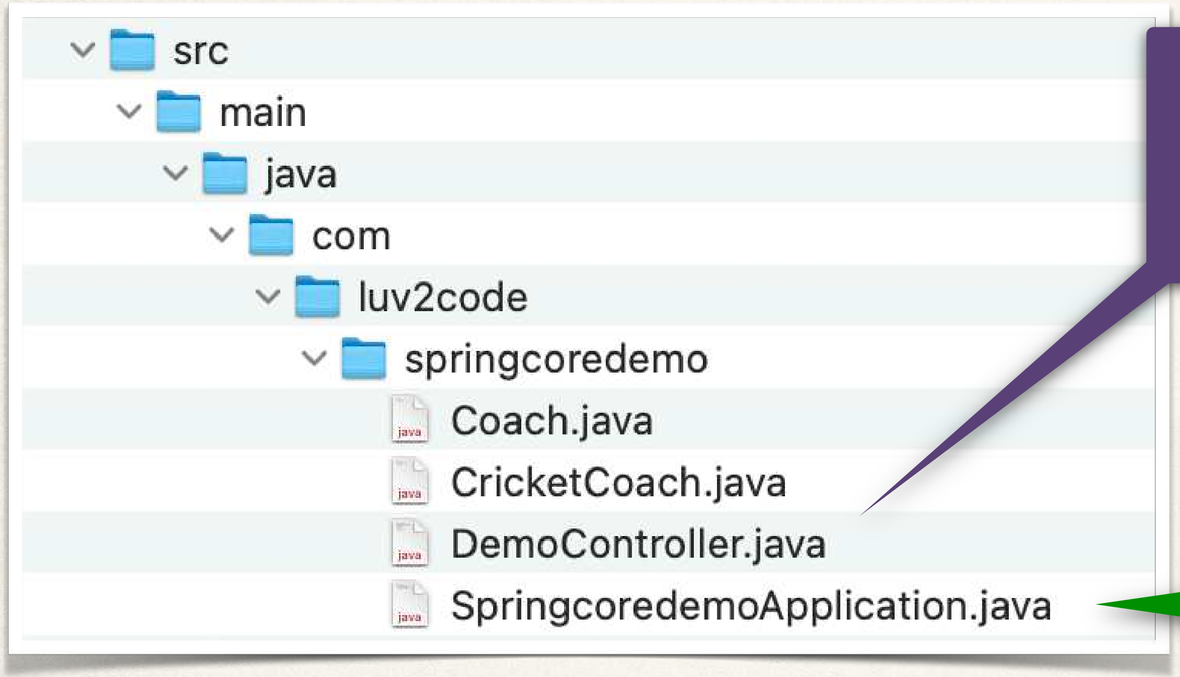
Good things are coming :-)

Component Scanning

Scanning for Component Classes

- Spring will scan your Java classes for special annotations
  - @Component, etc …
- Automatically register the beans in the Spring container

Java Source Code



RestController that we created in an earlier video

Main Spring Boot application class

Created by Spring Initializr

Java Source Code

File: SpringcoredemoApplication.java

```
package com.luv2code.springcoredemo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringcoredemoApplication {

  public static void main(String[] args) {
    SpringApplication.run(SpringcoredemoApplication, args);
  }
}
```

Java Source Code

File: SpringcoredemoApplication.java

```
package com.luv2code.springcoredemo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringcoredemoApp

  public static void main(Stri
    SpringApplication.run(Spri
  }
}
```

Enables

Auto configuration
Component scanning
Additional configuration

Java Source Code

File: SpringcoredemoApplication.java

```
package com.luv2code.springcoredemo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringcoredemoApp

  public static void main(Stri
    SpringApplication.run(Spri
  }
}
```

Composed of following annotations

@EnableAutoConfiguration
@ComponentScan
@Configuration

Annotations

- **`@SpringBootApplication`** is composed of the following annotations:

Annotation	Description
`@EnableAutoConfiguration`	**Enables Spring Boot's auto-configuration support**
`@ComponentScan`	**Enables component scanning of current package Also recursively scans sub-packages**
`@Configuration`	**Able to register extra beans with `@Bean` or import other configuration classes**

Java Source Code

File: SpringcoredemoApplication.java

```
package com.luv2code.springcoredemo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringcoredemoApplication {

  public static void main(String[] args) {
    SpringApplication.run(SpringcoredemoApplication, args);
  }
}
```

Bootstrap your Spring Boot application

Java Source Code

File: SpringcoredemoApplication.java

```
package com.luv2code.springcoredemo;

import org.springframework.boot.SpringApplicat
import org.springframework.boot.autoconfigure.

@SpringBootApplication
public class SpringcoredemoApplication {

  public static void main(String[] args) {
    SpringApplication.run(SpringcoredemoApplication, args);
  }
}
```

Behind the scenes ...

Creates application context and registers all beans

Starts the embedded server Tomcat etc...

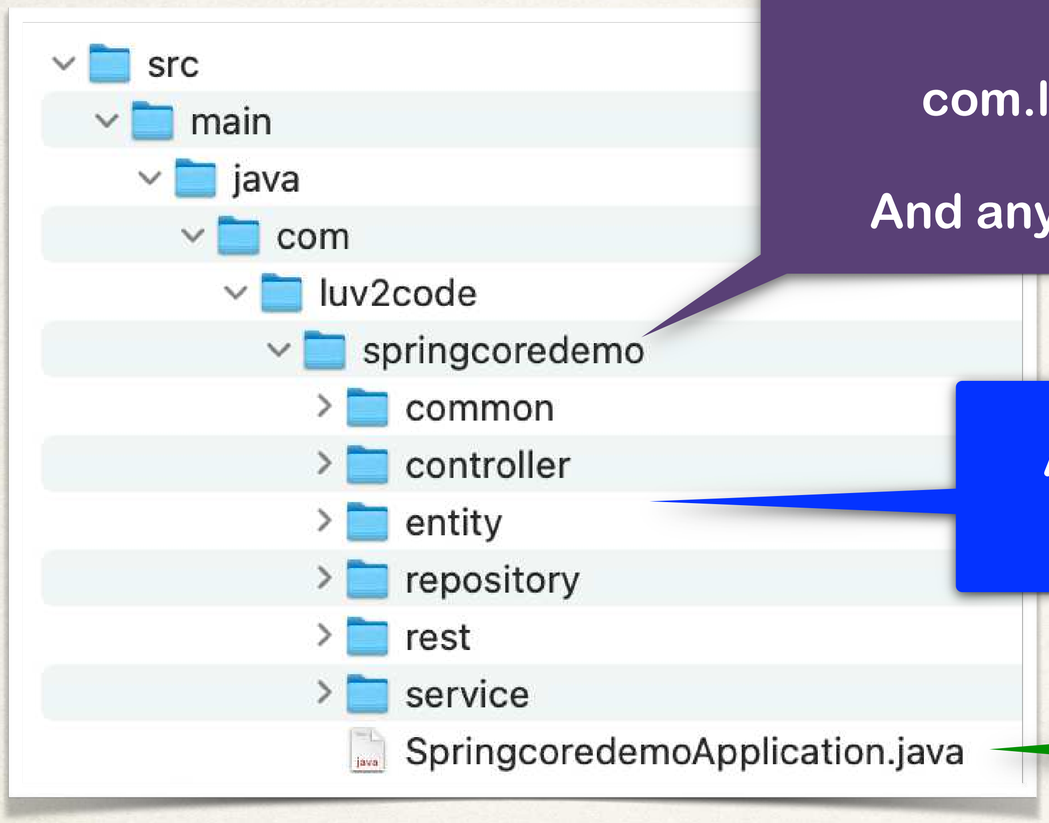
Bootstrap your Spring Boot application

More on Component Scanning

- By default, Spring Boot starts component scanning
  - From same package as your main Spring Boot application
  - Also scans sub-packages recursively

- This implicitly defines a base search package
  - Allows you to leverage default component scanning
  - No need to explicitly reference the base package name

More on Component Scanning



Scans everything in
com.luv2code.springcoredemo

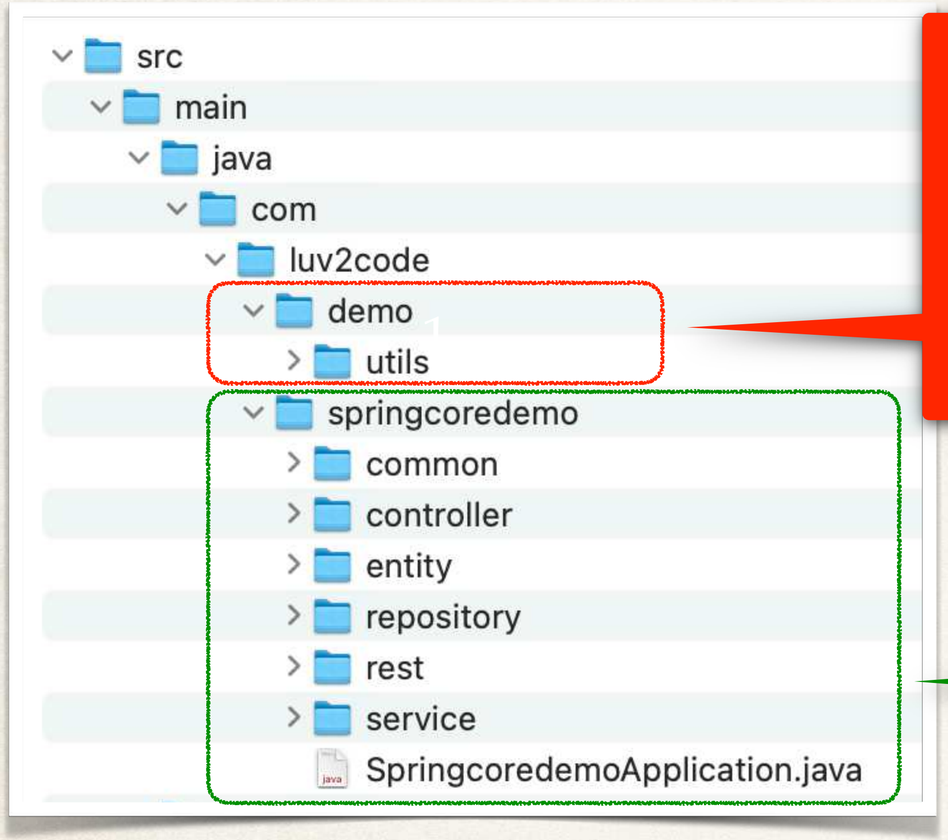
And any sub-packages (recursively)

Any other sub-packages you create
Can give them any name

Main Spring Boot application class

Automatically component scans package
and sub-packages

Common Pitfall - Different location



By default, Spring Boot will not component scan these packages

Only package of main Spring Boot application class and sub-packages

Main Spring Boot application class

Automatically component scans package and sub-packages

More on Component Scanning

- Default scanning is fine if everything is under
  - **com.luv2code.springcoredemo**
- But what about my other packages?
  - **com.luv2code.util**
  - **org.acme.cart**
  - **edu.cmu.srs**

Explicitly list base packages to scan

```
package com.luv2code.springcoredemo;
…
@SpringBootApplication(
    scanBasePackages={"com.luv2code.springcoredemo",
                      "com.luv2code.util",
                      "org.acme.cart",
                      "edu.cmu.srs"})
public class SpringcoredemoApplication {
   …
}
```

Setter Injection

Spring Injection Types

- Constructor Injection

- Setter Injection

Inject dependencies by calling

setter method(s) on your class

Autowiring Example

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

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: DemoController.java

```
@RestController
public class DemoController {

    private Coach myCoach;

    public void setCoach(Coach theCoach) {
        myCoach = theCoach;
    }

    …
}
```

Step 2: Configure the dependency injection with Autowired Annotation

File: DemoController.java

```
@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public void setCoach(Coach theCoach) {
        myCoach = theCoach;
    }

    …
}
```

The Spring Framework will perform operations

behind the scenes for you :-)

How Spring Processes your application

File: Coach.java

```
public interface Coach {

    String getDailyWorkout();
}
```

File: CricketCoach.java

```
@Component
public class CricketCoach implements Coach {

    @Override
    public String getDailyWorkout() {
        return "Practice fast bowling for 15 minutes";
    }
}
```

File: DemoController.java

```
@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public void setCoach(Coach theCoach) {
        myCoach = theCoach;
    }
     …
}
```

Spring Framework

```
CricketCoach theCoach = new CricketCoach();

DemoController demoController = new DemoController();

demoController.setCoach(theCoach);
```

Setter injection

Inject dependencies by calling

ANY method on your class

Simply give: @Autowired

Step 2: Configure the dependency injection with Autowired Annotation

File: DemoController.java

```
@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public void doSomeStuff(Coach theCoach) {
        myCoach = theCoach;
    }

    …
}
```

Can use any method name

Injection Types - Which one to use?

- Constructor Injection
  - Use this when you have required dependencies
  - Generally recommended by the spring.io development team as first choice

- Setter Injection
  - Use this when you have optional dependencies
  - If dependency is not provided, your app can provide reasonable default logic

Field Injection with Annotations and Autowiring

Spring Injection Types

- Recommended by the spring.io development team
  - Constructor Injection: required dependencies
  - Setter Injection: optional dependencies

- Not recommended by the spring.io development team
  - Field Injection

Field Injection … no longer cool

- In the early days, field injection was popular on Spring projects
  - In recent years, it has fallen out of favor
- In general, it makes the code harder to unit test
- As a result, the spring.io team does not recommend field injection
  - However, you will still see it being used on legacy projects

Inject dependencies by setting field values on your class directly

(even private fields)

Accomplished by using Java Reflection

Step 1: Configure the dependency injection with Autowired Annotation

File: DemoController.java

```
package com.luv2code.springcoredemo;

import org.springframework.beans.factory.annotation.Autowired;
…

@RestController
public class DemoController {

    @Autowired
    private Coach myCoach;

    // no need for constructors or setters

    @GetMapping("/dailyworkout")
    public String getDailyWorkout() {
        return myCoach.getDailyWorkout();
    }
}
```

Field injection

Field injection is not recommended by spring.io development team.

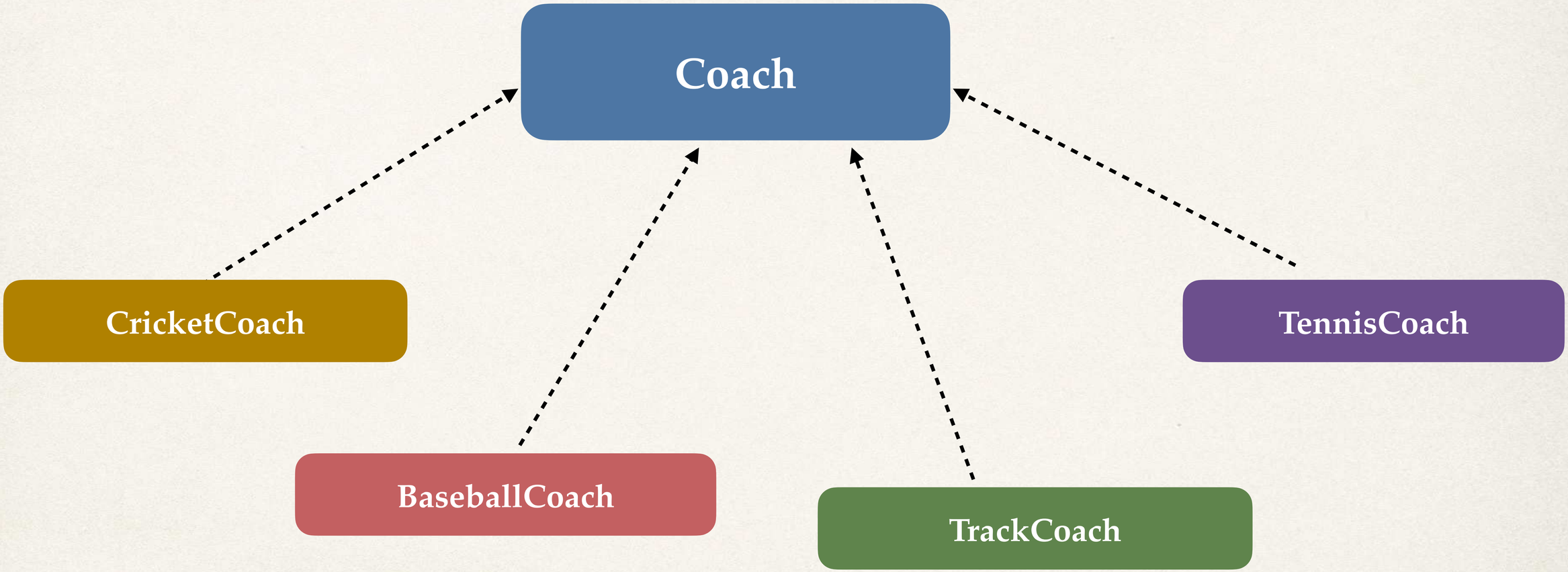
Makes the code harder to unit test.

Annotation Autowiring and Qualifiers

Autowiring

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

Multiple Coach Implementations



Multiple Coach Implementations

```java
package com.luv2code.springcoredemo.common;

import org.springframework.stereotype.Component;

@Component
public class CricketCoach implements Coach {

    @Override
    public String getDailyWorkout() {
        return "Practice fast bowling for 15 minutes";
    }
}
```

```java
package com.luv2code.springcoredemo.common;

import org.springframework.stereotype.Component;

@Component
public class BaseballCoach implements Coach {

    @Override
    public String getDailyWorkout() {
        return "Spend 30 minutes in batting practice";
    }
}
```

```java
package com.luv2code.springcoredemo.common;

import org.springframework.stereotype.Component;

@Component
public class TrackCoach implements Coach {

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

```java
package com.luv2code.springcoredemo.common;

import org.springframework.stereotype.Component;

@Component
public class TennisCoach implements Coach {

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

Umm, we have a little problem ....

Parameter 0 of constructor in com.luv2code.springcoredemo.rest.DemoController required a single bean, but 4 were found:

- baseballCoach
- cricketCoach
- tennisCoach
- trackCoach

...

Solution: Be specific! - @Qualifier

```
package com.luv2code.springcoredemo.rest;

import com.luv2code.springcoredemo.common.Coach;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
…

@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public DemoController(@Qualifier("cricketCoach") Coach theCoach) {
        myCoach = theCoach;
    }

    @GetMapping("/dailyworkout")
    public String getDailyWorkout() {
        return myCoach.getDailyWorkout();
    }
}
```

Specify the bean id: cricketCoach

Same name as class, first character lower-case

Other bean ids we could use:
baseballCoach, trackCoach, tennisCoach

For Setter Injection

- If you are using Setter injection, can also apply **@Qualifier** annotation

Setter Injection - @Qualifier

```
package com.luv2code.springcoredemo.rest;

import com.luv2code.springcoredemo.common.Coach;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
…

@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public void setCoach(@Qualifier("cricketCoach") Coach theCoach) {
        myCoach = theCoach;
    }

    @GetMapping("/dailyworkout")
    public String getDailyWorkout() {
        return myCoach.getDailyWorkout();
    }
}
```

Give bean id: cricketCoach

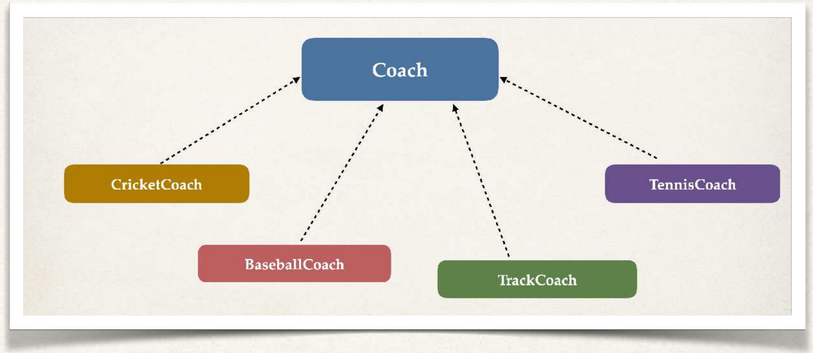
Same name as class, first character lower-case

Other bean ids we could use:
baseballCoach, trackCoach, tennisCoach

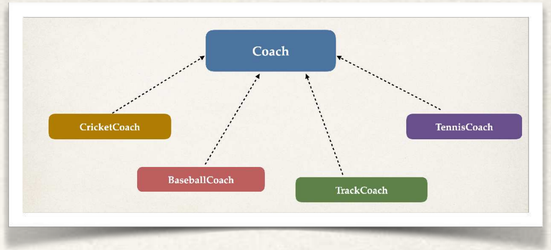
@Primary annotation

Resolving issue with Multiple Coach implementations

- In the case of multiple Coach implementations
  - We resolved it using @Qualifier
  - We specified a coach by name
- Alternate solution available …



Alternate solution



- Instead of specifying a coach by name using @Qualifier

- I simply need a coach … I don't care which coach
  - If there are multiple coaches
  - Then you coaches figure it out … and tell me who's the **primary** coach

Multiple Coach Implementations

```
package com.luv2code.springcoredemo.common;

import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Component;

@Component
@Primary
public class TrackCoach implements Coach {

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

```
@Component
public class BaseballCoach implements Coach {
  …
}
```

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

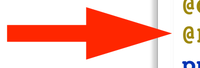
```
@Component
public class CricketCoach implements Coach {
  …
}
```

Resolved with @Primary

```
package com.luv2code.springcoredemo.rest;

import com.luv2code.springcoredemo.common.Coach;
import org.springframework.beans.factory.annotation.Autowired;
…

@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public DemoController(Coach theCoach) {
        myCoach = theCoach;
    }

    @GetMapping("/dailyworkout")
    public String getDailyWorkout() {
        return myCoach.getDailyWorkout();
    }
}
```

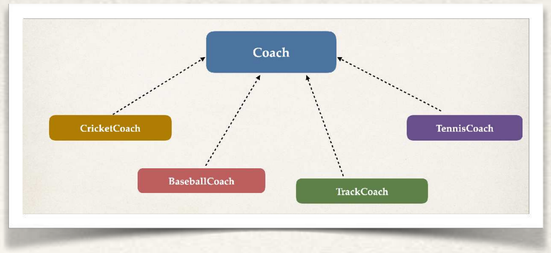
No need for @Qualifier

There is a @Primary coach
This example will use TrackCoach



```
package com.luv2code.springcoredemo.common;

import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Component;

@Component
@Primary
public class TrackCoach implements Coach {

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

@Primary - Only one



- When using @Primary, can have only one for multiple implementations
- If you mark multiple classes with @Primary … umm, we have a problem

```
Unsatisfied dependency expressed through constructor parameter 0:
No qualifying bean of type 'com.luv2code.springcoredemo.common.Coach' available:
more than one 'primary' bean found among candidates:
[baseballCoach, cricketCoach, tennisCoach, trackCoach]
...
```

Mixing @Primary and @Qualifier

- If you mix @Primary and @Qualifier
  - @Qualifier has higher priority

Mixing @Primary and @Qualifier

```
package com.luv2code.springcoredemo.rest;

import com.luv2code.springcoredemo.common.Coach;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

…

@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public DemoController(@Qualifier("cricketCoach") Coach theCoach) {
        myCoach = theCoach;
    }

    @GetMapping("/dailyworkout")
    public String getDailyWorkout() {
        return myCoach.getDailyWorkout();
    }
}
```

@Qualifier has higher priority

Even though there is a @Primary coach
This example will use CricketCoach

```
package com.luv2code.springcoredemo.common;

import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Component;

@Component
@Primary
public class TrackCoach implements Coach {

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

Which one: @Primary or @Qualifier?

- @Primary leaves it up to the implementation classes
  - Could have the issue of multiple @Primary classes leading to an error
- @Qualifier allows to you be very specific on which bean you want
- In general, I recommend using @Qualifier
  - More specific
  - Higher priority

Lazy Initialization

Initialization

- By default, when your application starts, all beans are initialized
  - @Component, etc …
- Spring will create an instance of each and make them available

Diagnostics: Add println to constructors

Get the name of the class

```
@Component
public class CricketCoach implements Coach {

    public CricketCoach() {
        System.out.println("In constructor: " + getClass().getSimpleName());
    }
    ...
}
```

```
@Component
public class BaseballCoach implements Coach {

    public BaseballCoach() {
        System.out.println("In constructor: " + getClass().getSimpleName());
    }
    ...
}
```

```
@Component
public class TrackCoach implements Coach {

    public TrackCoach() {
        System.out.println("In constructor: " + getClass().getSimpleName());
    }
    ...
}
```

```
@Component
public class TennisCoach implements Coach {

    public TennisCoach() {
        System.out.println("In constructor: " + getClass().getSimpleName());
    }
    ...
}
```

When we start the application

```
...
In constructor: BaseballCoach
In constructor: CricketCoach
In constructor: TennisCoach
In constructor: TrackCoach
...
```

By default, when your application starts, all beans are initialized

Spring will create an instance of each and make them available

Lazy Initialization

- Instead of creating all beans up front, we can specify lazy initialization

- A bean will only be initialized in the following cases:
  - It is needed for dependency injection
  - Or it is explicitly requested

- Add the @Lazy annotation to a given class

Lazy Initialization with @Lazy

Bean is only initialized if needed for dependency injection

```
import org.springframework.context.annotation.Lazy;
import org.springframework.stereotype.Component;

@Component
@Lazy
public class TrackCoach implements Coach {

    public TrackCoach() {
        System.out.println("In constructor: " + getClass().getSimpleName());
    }
    …
}
```

Inject cricketCoach

```
@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public DemoController(@Qualifier("cricketCoach") Coach theCoach) {
        myCoach = theCoach;
    }
     …
}
```

Since we are NOT injecting TrackCoach … it is not initialized

```
…
In constructor: BaseballCoach
In constructor: CricketCoach
In constructor: TennisCoach
…
```

Lazy Initialization

- To configure other beans for lazy initialization
  - We would need to add @Lazy to each class
- Turns into tedious work for a large number of classes
- I wish we could set a global configuration property …

Lazy Initialization - Global configuration

File: application.properties

```
spring.main.lazy-initialization=true
…
```

All beans are lazy …
no beans are created until needed

Including our DemoController

Once we access REST endpoint `/dailywork`

Spring will determine dependencies
for DemoController …

For dependency resolution
Spring creates instance of CricketCoach first …

Then creates instance of DemoController
and injects the CricketCoach

Add println to DemoController constructor

```
@Component
public class CricketCoach implements Coach {

    public CricketCoach() {
        System.out.println("In constructor: " + getClass().getSimpleName());
    }
    …
}
```

```
@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public DemoController(@Qualifier("cricketCoach") Coach theCoach) {
        System.out.println("In constructor: " + getClass().getSimpleName());
        myCoach = theCoach;
    }
}
```

For dependency resolution
Spring creates instance of CricketCoach first …

Then creates instance of DemoController
and injects the CricketCoach

```
…
In constructor: CricketCoach
In constructor: DemoController
…
```

Lazy Initialization

Lazy initialization feature
is disabled by default.

You should profile your application
before configuring lazy initialization.

Avoid the common pitfall of premature optimization.

- Advantages
  - Only create objects as needed
  - May help with faster startup time if you have large number of components
- Disadvantages
  - If you have web related components like @RestController, not created until requested
  - May not discover configuration issues until too late
  - Need to make sure you have enough memory for all beans once created

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

Default scope is singleton

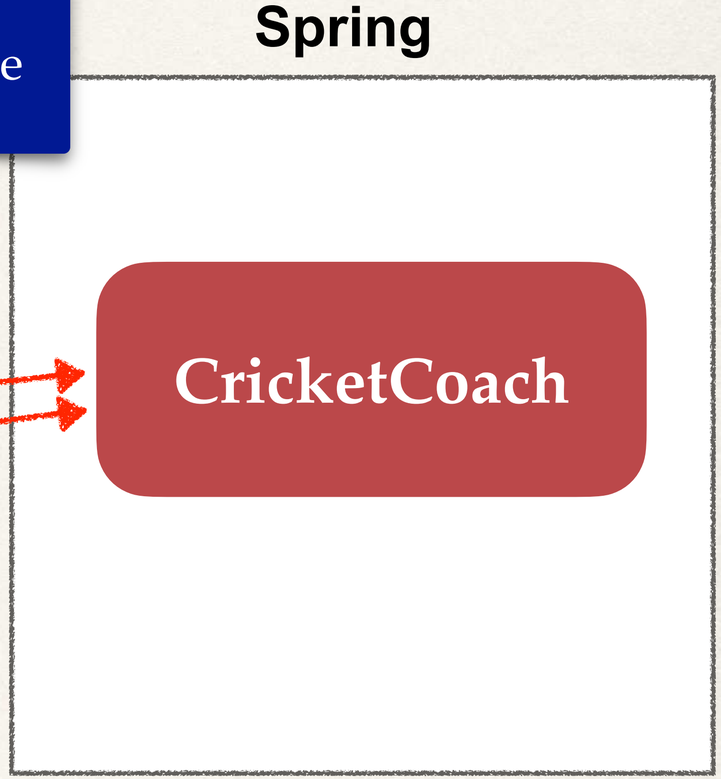
Refresher: What Is a Singleton?

- Spring Container creates only one instance of the bean, by default
- It is cached in memory
- All dependency injections for the bean
  - will reference the SAME bean

What is a Singleton?

```
@RestController
public class DemoController {

    private Coach myCoach;
    private Coach anotherCoach;

    @Autowired
    public DemoController(
                @Qualifier("cricketCoach") Coach theCoach,
                @Qualifier("cricketCoach") Coach theAnotherCoach) {
       myCoach = theCoach;
       anotherCoach = theAnotherCoach;
    }

    …
}
```

Both point to the same instance



Explicitly Specify Bean Scope

```
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;


@Component
@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)
public class CricketCoach 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

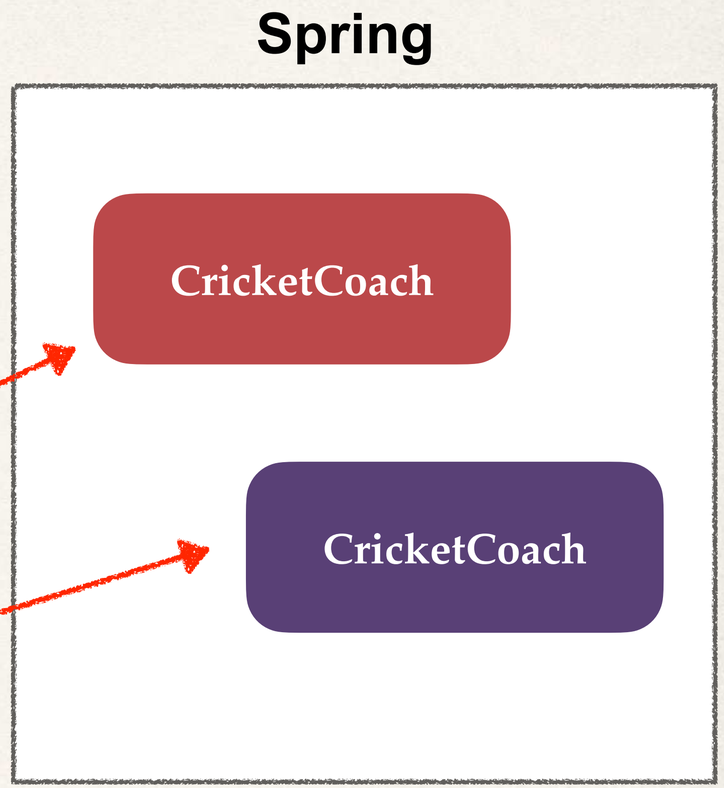
Prototype scope: new object instance for each injection

```
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;


@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class CricketCoach implements Coach {

    …

}
```

Prototype Scope Example

Prototype scope: new object instance for each injection

```
@RestController
public class DemoController {

    private Coach myCoach;
    private Coach anotherCoach;

    @Autowired
    public DemoController(
                @Qualifier("cricketCoach") Coach theCoach,
                @Qualifier("cricketCoach") Coach theAnotherCoach) {
       myCoach = theCoach;
       anotherCoach = theAnotherCoach;
    }

    …
}
```



Checking on the scope

```
@RestController
public class DemoController {

    private Coach myCoach;
    private Coach anotherCoach;

    @Autowired
    public DemoController(
                @Qualifier("cricketCoach") Coach theCoach,
                @Qualifier("cricketCoach") Coach theAnotherCoach) {
       myCoach = theCoach;
       anotherCoach = theAnotherCoach;
    }

    @GetMapping("/check")
    public String check() {
        return "Comparing beans: myCoach == anotherCoach, " + (myCoach == anotherCoach);
    }

    ...
}
```

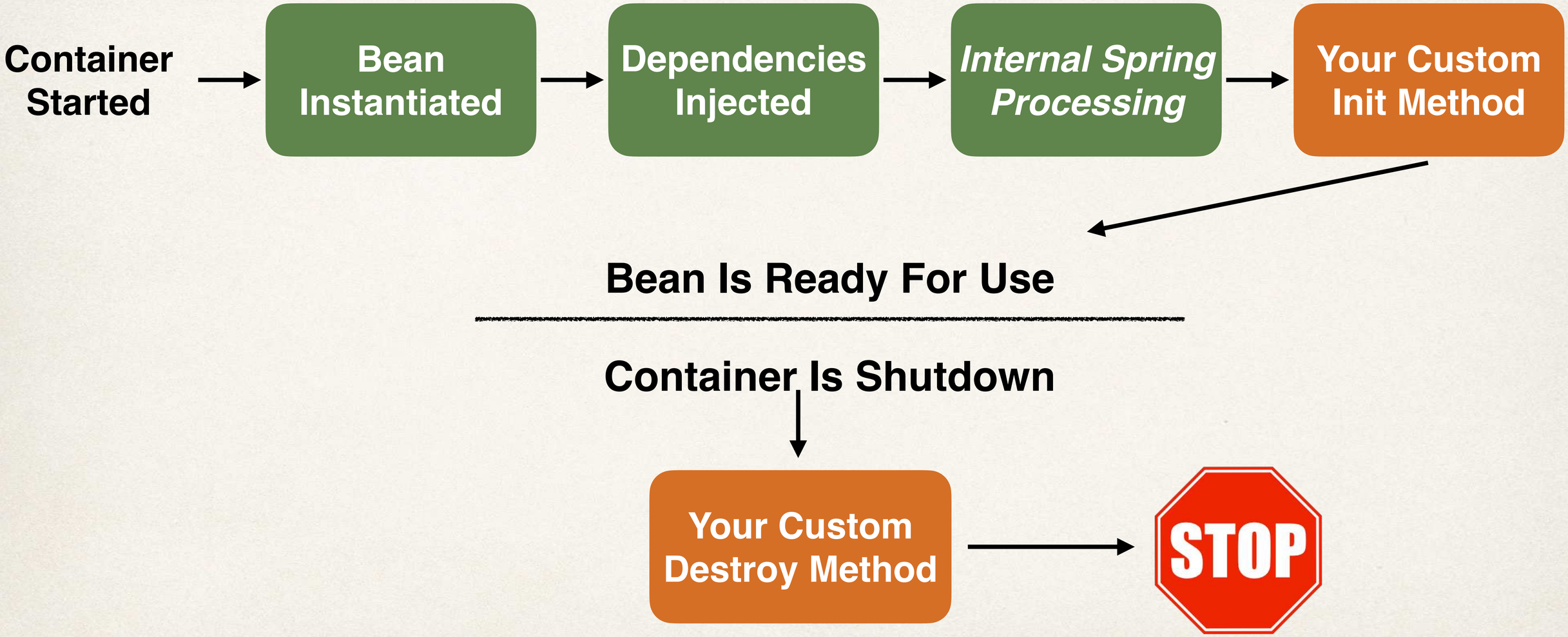
Check to see if this is the same bean

True or False depending on the bean scope

Singleton: True
Prototype: False

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 CricketCoach implements Coach {

    public CricketCoach() {
        System.out.println("In constructor: " + getClass().getSimpleName());
    }

    @PostConstruct
    public void doMyStartupStuff() {
        System.out.println("In doMyStartupStuff(): " + getClass().getSimpleName());
    }

    …
}
```

Destroy: method configuration

```
@Component
public class CricketCoach implements Coach {

    public CricketCoach() {
        System.out.println("In constructor: " + getClass().getSimpleName());
    }

    @PostConstruct
    public void doMyStartupStuff() {
        System.out.println("In doMyStartupStuff(): " + getClass().getSimpleName());
    }

    @PreDestroy
    public void doMyCleanupStuff() {
        System.out.println("In doMyCleanupStuff(): " + getClass().getSimpleName());
    }
    ...
}
```

Development Process

1. Define your methods for init and destroy

2. Add annotations: @PostConstruct and @PreDestroy

Configuring Beans with Java Code

Our New Coach …

No special annotations

```
package com.luv2code.springcoredemo.common;

public class SwimCoach implements Coach {

  …

}
```

Coach

Development Process

1. Create @Configuration class

2. Define @Bean method to configure the bean

3. Inject the bean into our controller

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

```
package com.luv2code.springcoredemo.config;

import org.springframework.context.annotation.Configuration;

@Configuration
public class SportConfig {

   …
}
```

Step 2: Define @Bean method to configure the bean

```
package com.luv2code.springcoredemo.config;

import com.luv2code.springcoredemo.common.Coach;
import com.luv2code.springcoredemo.common.SwimCoach;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class SportConfig {

    @Bean
    public Coach swimCoach() {
        return new SwimCoach();
    }

}
```

The bean id defaults to the method name

Step 3: Inject the bean into our controller

```
package com.luv2code.springcoredemo.rest;

import com.luv2code.springcoredemo.common.Coach;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public DemoController(@Qualifier("swimCoach") Coach theCoach) {
        System.out.println("In constructor: " + getClass().getSimpleName());
        myCoach = theCoach;
    }

   …
}
```

Inject the bean using the bean id

Use case for @Bean

- You may wonder …
  - *Using the "new" keyword … is that it???*
  - *Why not just annotate the class with @Component???*

Use case for @Bean

- Make an existing third-party class available to Spring framework
- You may not have access to the source code of third-party class
- However, you would like to use the third-party class as a Spring bean

Real-World Project Example

- Our project used Amazon Web Service (AWS) to store documents
  - Amazon Simple Storage Service (Amazon S3)
  - Amazon S3 is a cloud-based storage system
  - can store PDF documents, images etc
- We wanted to use the AWS S3 client as a Spring bean in our app

Real-World Project Example

- The AWS S3 client code is part of AWS SDK
  - We can't modify the AWS SDK source code
  - We can't just add @Component
- However, we can configure it as a Spring bean using @Bean

Configure AWS S3 Client using @Bean

```
package com.luv2code.springcoredemo.config;

…
import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.s3.S3Client;

@Configuration
public class DocumentsConfig {

    @Bean
    public S3Client remoteClient() {

        // Create an S3 client to connect to AWS S3
        ProfileCredentialsProvider credentialsProvider = ProfileCredentialsProvider.create();
        Region region = Region.US_EAST_1;
        S3Client s3Client = S3Client.builder()
                .region(region)
                .credentialsProvider(credentialsProvider)
                .build();

        return s3Client;
    }
}
```

Inject the S3Client as a bean

```
package com.luv2code.springcoredemo.services;

import software.amazon.awssdk.services.s3.S3Client;
…

@Component
public class DocumentsService {

    private S3Client s3Client;

    @Autowired
    public DocumentsService(S3Client theS3Client) {
        s3Client = theS3Client;
    }

   …
}
```

Store our document in S3

```
package com.luv2code.springcoredemo.services;

import software.amazon.awssdk.services.s3.S3Client;
…

@Component
public class DocumentsService {

    private S3Client s3Client;

    @Autowired
    public DocumentsService(S3Client theS3Client) {
        s3Client = theS3Client;
    }

    public void processDocument(Document theDocument) {

       // get the document input stream and file size …

       // Store document in AWS S3
       // Create a put request for the object
       PutObjectRequest putObjectRequest = PutObjectRequest.builder()
               .bucket(bucketName)
               .key(subDirectory + "/" + fileName)
               .acl(ObjectCannedACL.BUCKET_OWNER_FULL_CONTROL).build();

       // perform the putObject operation to AWS S3 ... using our autowired bean
       s3Client.putObject(putObjectRequest, RequestBody.fromInputStream(fileInputStream, contentLength));
    }
}
```

Wrap Up

- We could use the Amazon S3 Client in our Spring application
- The Amazon S3 Client class was not originally annotated with @Component
- However, we configured the S3 Client as a Spring Bean using @Bean
- It is now a Spring Bean and we can inject it into other services of our application

- *Make an existing third-party class available to Spring framework*