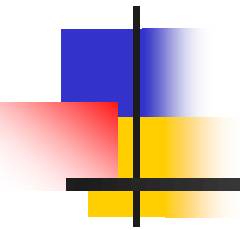
Advanced Application Development (CSE-214) week-7 :Spring Core



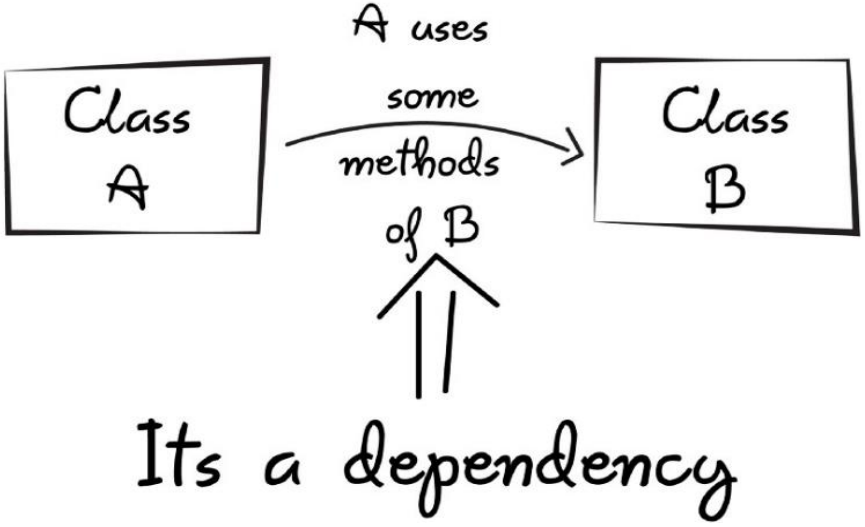
Dr. Alper ÖZCAN
Akdeniz University
alper.ozcan@gmail.com

What is a dependency?

When Class A uses Class B

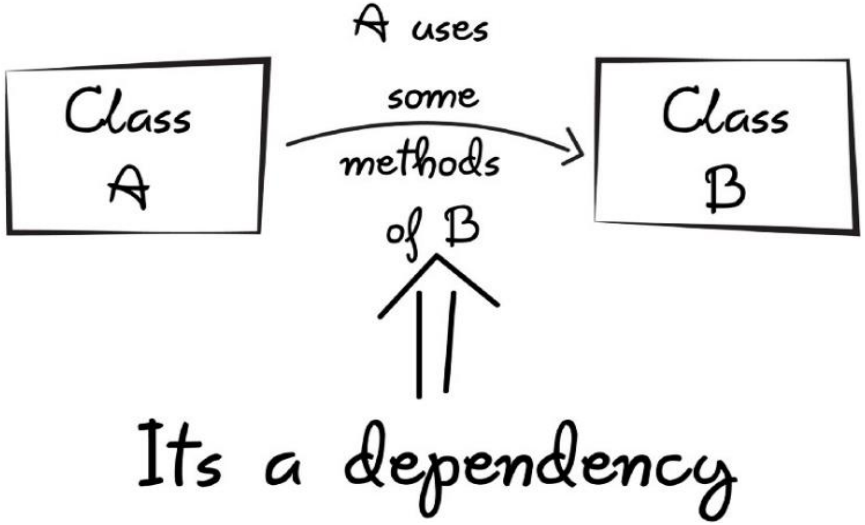
- A can't be used independently without B.
- Class A is **tightly coupled** with Class B, changing or replacement of Class B will affect Class A
- Testing of A is also not possible without Class B.

A good design has **low dependency/loosely coupling.**



A state where one class depends on another is called dependency or coupling. Class A is dependent on Class B, i.e. Class A has to create an instance of Class B

Why is dependency bad?



- **Difficult to change**
- **Unit Testing**: if a unit has a high dependency, it isn't easy to test it independently.
- **Reusability:** If a class depends on many other classes**,** it isn't easy to use that class elsewhere.

Inversion of Control (IoC)

- **IoC is often implemented using a Dependency Injection (DI)** container, which is responsible for managing the **dependencies between objects** and providing them to the objects that need them.

- Inversion of Control is a principle in software engineering which **transfers the control of objects** or **portions of a program to a container or framework.**

- In software development, rather than tightly coupling components and controlling every aspect of their interaction, you define the **overall structure** and let a **framework** or **container** manage how components collaborate.

Inversion of Control (IoC) in Spring

- **Inversion of Control (IoC)** is a principle where the control over **object creation** and **dependency management** is **transferred** from the application to a **Spring Container** or **framework**. Instead of **explicitly instantiating** dependent objects inside a class, **the framework provides those dependencies** when **needed**.

Tightly Coupled vs. Loosely Coupled Dependencies

- **Tightly Coupled**: When one class **directly** instantiates another, making changes difficult.
- **Loosely Coupled**: Dependencies are managed **externally**, allowing for **flexibility** and **easier maintenance**.

Inversion of Control (IoC)

Let's look at simple example of the **traditional approach** where **the flow of control is tightly coupled, meaning that if a component needs to use another component or class, it must explicitly create or instantiate that object and maintain its lifecycle.**

class Course is a dependency of class Teacher.

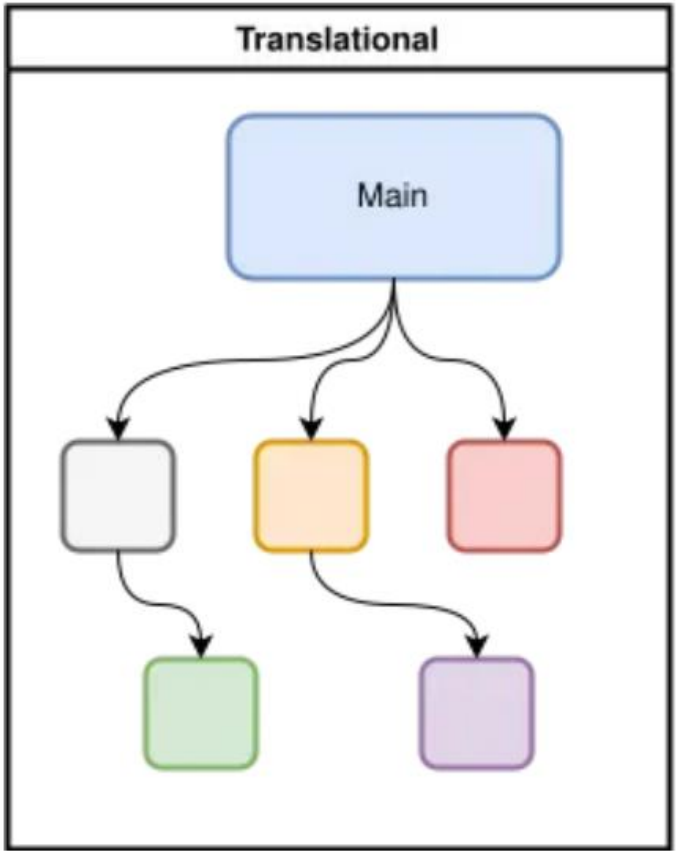
```
public class Teacher {
    private Course course;

    public Teacher(Course course) {
        this.course = course;
    }

    public void assignCourse() {
        System.out.println("Assigned course: " + course.getCourse());
    }
}

public class Course {
    private String course;

    public Course() {}

    public Course(String course) {
        this.course = course;
    }

    public String getCourse() {
        return course;
    }
}
```

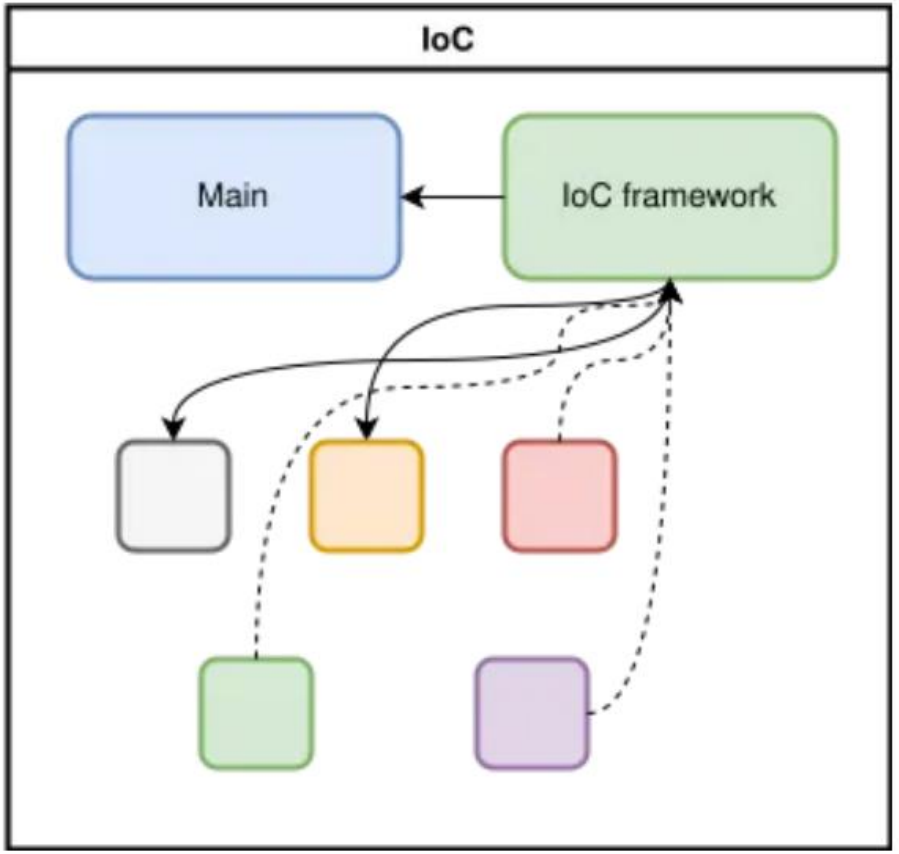
Inversion of Control (IoC)

The flow of control is **transferred** to the **external class** named **ObjectContainer**, now instead of creating and maintaining the object of **class Course** itself, **class Teacher** uses the **getObjectOfCourse method** of **ObjectContainer** class to get the object.

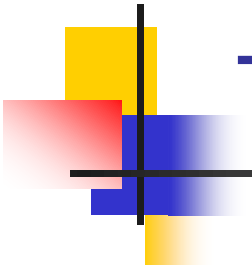
```
public class Teacher {
    private Course course;

    public Teacher() {
        this.course = ObjectContainer.getObjectOfCourse();
    }

    public void assignCourse() {
        System.out.println("Assigned course: " + course.getCourse());
    }
}

public class Course {
    private String course;

    public Course() {}

    public Course(String course) {
        this.course = course;
    }

    public String getCourse() {
        return course;
    }
}

public class ObjectContainer {
    public static Course getObjectOfCourse() {
        return new Course();
    }
}
```

Inversion of Control (IoC)





In simple, **IoC** says that the **dependencies** of a **component** are **managed** by **an external framework** or **container** and that **container** is **responsible** for **creating**, **managing** and **destroying** the **dependent objects** and the **component** does not have to worry about them.



The advantages of IoC and DI

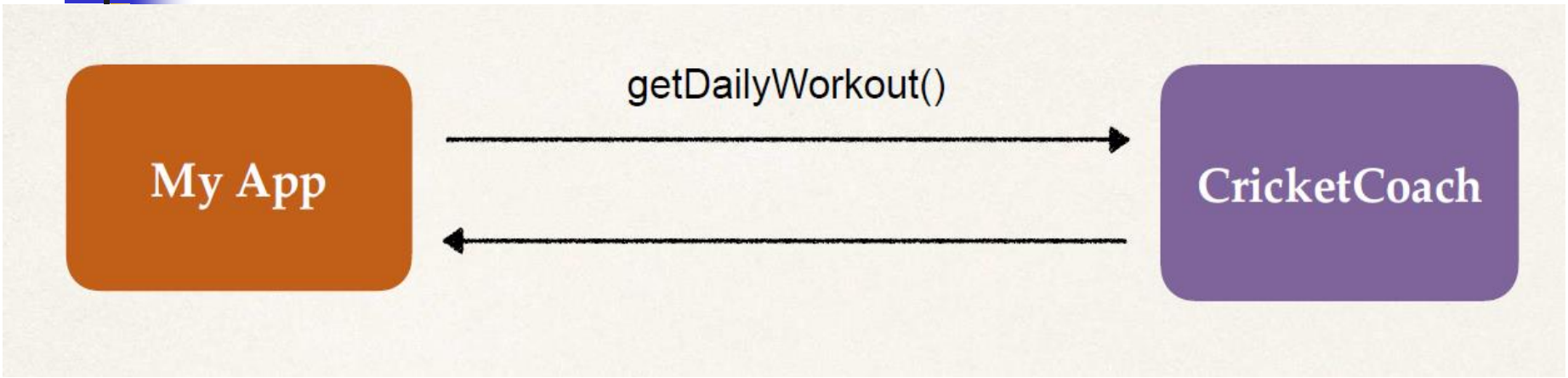
- *Loosely coupled dependencies:* Depends only on abstraction hiding the implementation

- *More reusable code:* Switch between different implementations without affecting the users of the interface.

- *Easy testing:* Testing each component independently with mocked dependencies

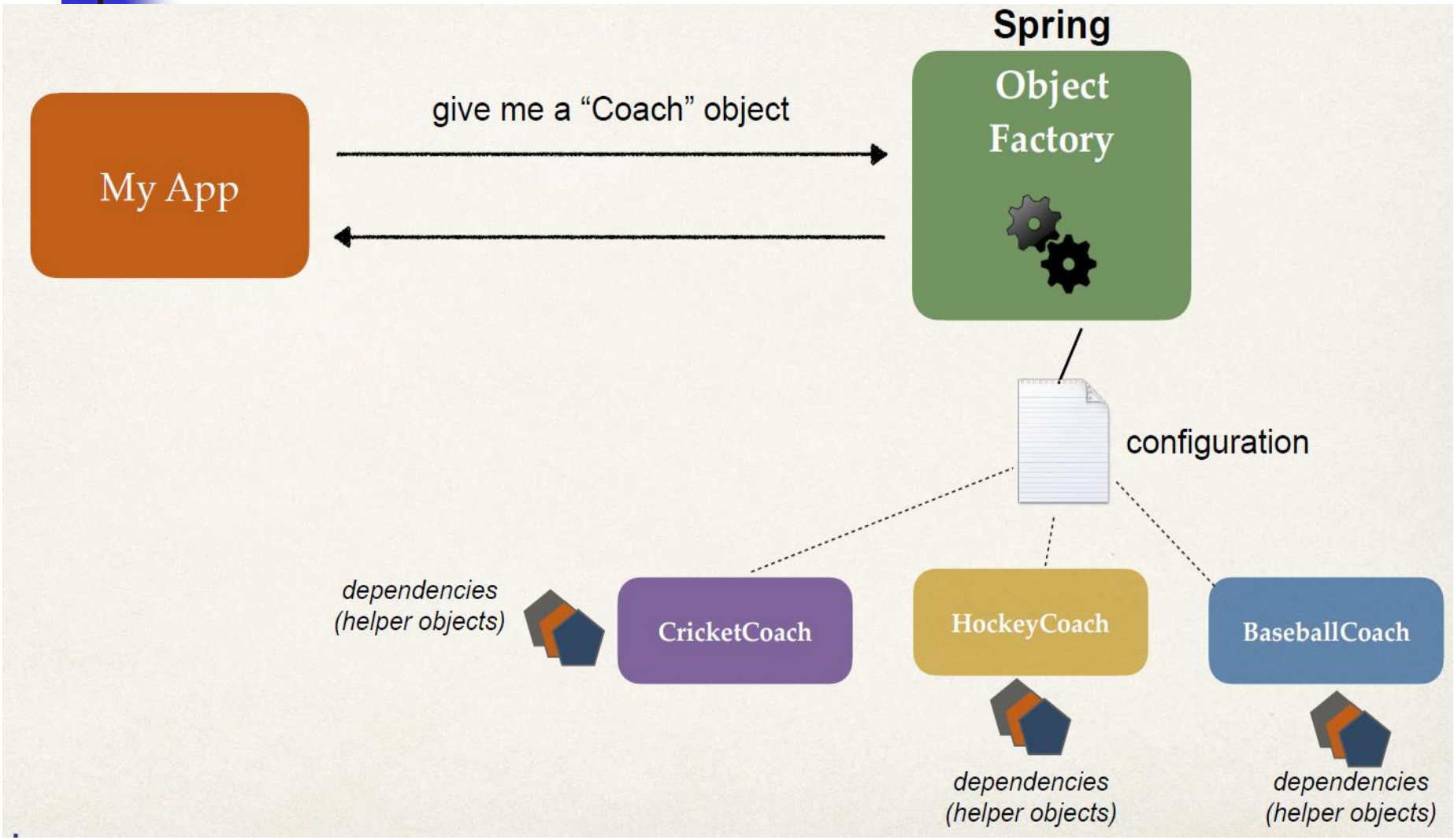
- *Cleaner and more readable code:* Object dependencies can be seen by just looking to the constructor of a class

Inversion of Control (IoC)



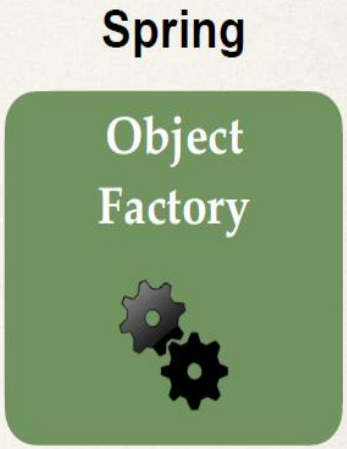
- App should be configurable
- Easily change the coach for another sport
- Baseball, Hockey, Tennis, Gymnastics etc.

Inversion of Control (IoC)



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

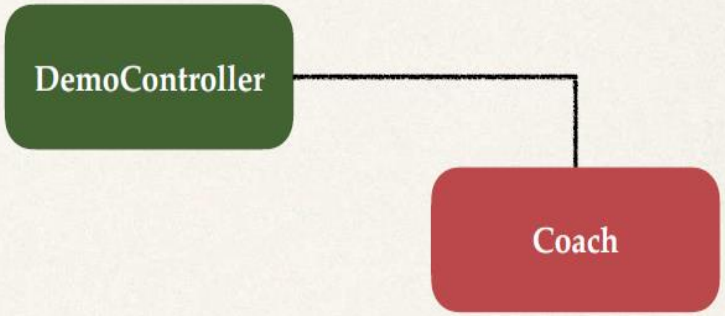
The client delegates to another object the responsibility of providing its dependencies.

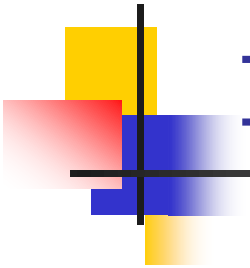
Spring IoC and Dependency Injection

Spring implements IoC using **Dependency Injection (DI)**
Instead of creating objects manually, **dependencies** are **injected by** the **Spring container**.

Example

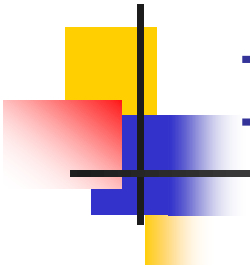
- **Coach** will provide daily workouts
- The DemoController wants to use a Coach
  - New helper: **Coach**
  - This is a ***dependency***
- Need to *inject* this *dependency*





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

Types of Dependency Injection in Spring

1. Constructor Injection : Spring injects dependencies via the class constructor.

The **Engine** class is a dependency of **Car**.

The @**Autowired** annotation tells Spring to **inject the Engine instance** via the **Car constructor**.

```
@Component
class Engine {
    public void start() {
        System.out.println("Engine started...");
    }
}

@Component
class Car {
    private final Engine engine;

    @Autowired
    public Car(Engine engine) {
        this.engine = engine;
    }

    public void drive() {
        engine.start();
        System.out.println("Car is driving...");
    }
}
```

Types of Dependency Injection in Spring

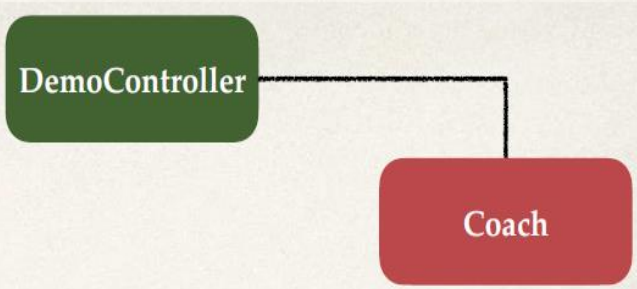
2. Setter Injection : Spring injects dependencies using setter methods.

Spring **injects the dependency** using the setEngine() method.

```java
@Component
class Engine {
    public void start() {
        System.out.println("Engine started...");
    }
}

@Component
class Car {
    private Engine engine;

    @Autowired
    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    public void drive() {
        engine.start();
        System.out.println("Car is driving...");
    }
}
```

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

Spring AutoWiring

Spring automatically wires dependencies using **annotations**.

@Component
Marks a class as a **Spring Bean**.

```
@Component
class Engine {
    public void start() {
        System.out.println("Engine started...");
    }
}
```

@Autowired

Tells Spring to **inject a dependency** automatically.

```
@Autowired
private Engine engine;
```

Spring AutoWiring

@Qualifier
Used when multiple beans of the same type exist, helping Spring to pick the correct one.

@Qualifier("dieselEngine") ensures that Spring injects a **DieselEngine** instead of **PetrolEngine**.

```
@Component
class DieselEngine implements Engine {
    @Override
    public void start() {
        System.out.println("Diesel Engine started...");
    }
}

@Component
class PetrolEngine implements Engine {
    @Override
    public void start() {
        System.out.println("Petrol Engine started...");
    }
}

@Component
class Car {
    private Engine engine;

    @Autowired
    public Car(@Qualifier("dieselEngine") Engine engine) {
        this.engine = engine;
    }
}
```

Spring AutoWiring

@Primary

Marks a bean as the **default choice** when multiple beans exist.

If no @Qualifier is used, Spring will inject **DefaultEngine**.

```
@Component
@Primary
class DefaultEngine implements Engine {
    @Override
    public void start() {
        System.out.println("Default Engine started...");
    }
}
```

Example Application

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

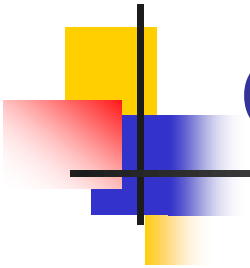
File: Coach.java

```
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 {


}
```

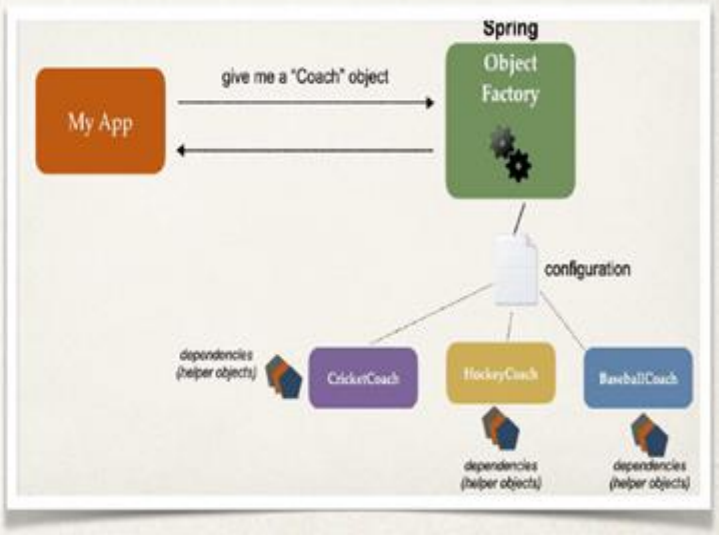
Step 3: Create a constructor in your class for injections

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

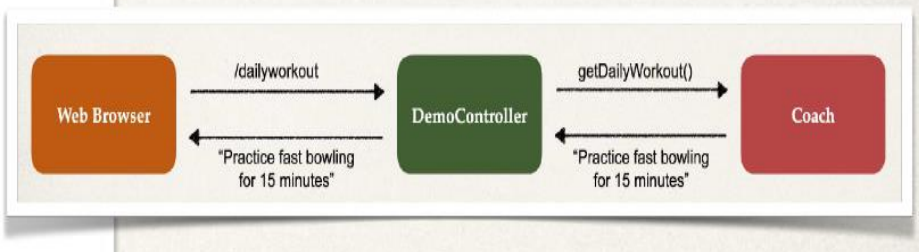
If you only have one constructor then @Autowired on constructor is optional



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();
    }
}
```



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);
```

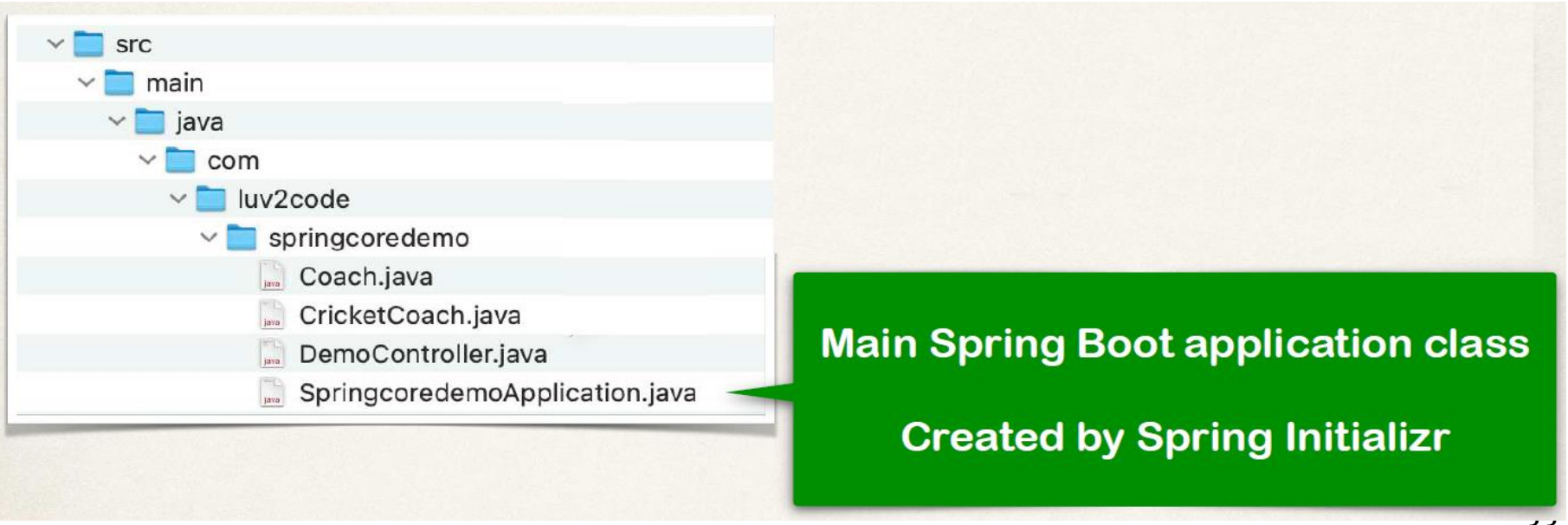
Constructor injection

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

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

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

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

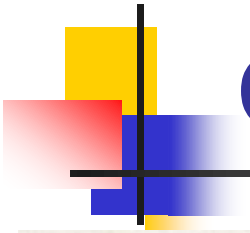
File: SpringcoredemoApplication.java

```
package com.luv2code.springcoredemo;

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

@SpringBootApplication
public class SpringcoredemoApplication {

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

Creates application context and registers all beans

Starts the embedded server Tomcat etc...

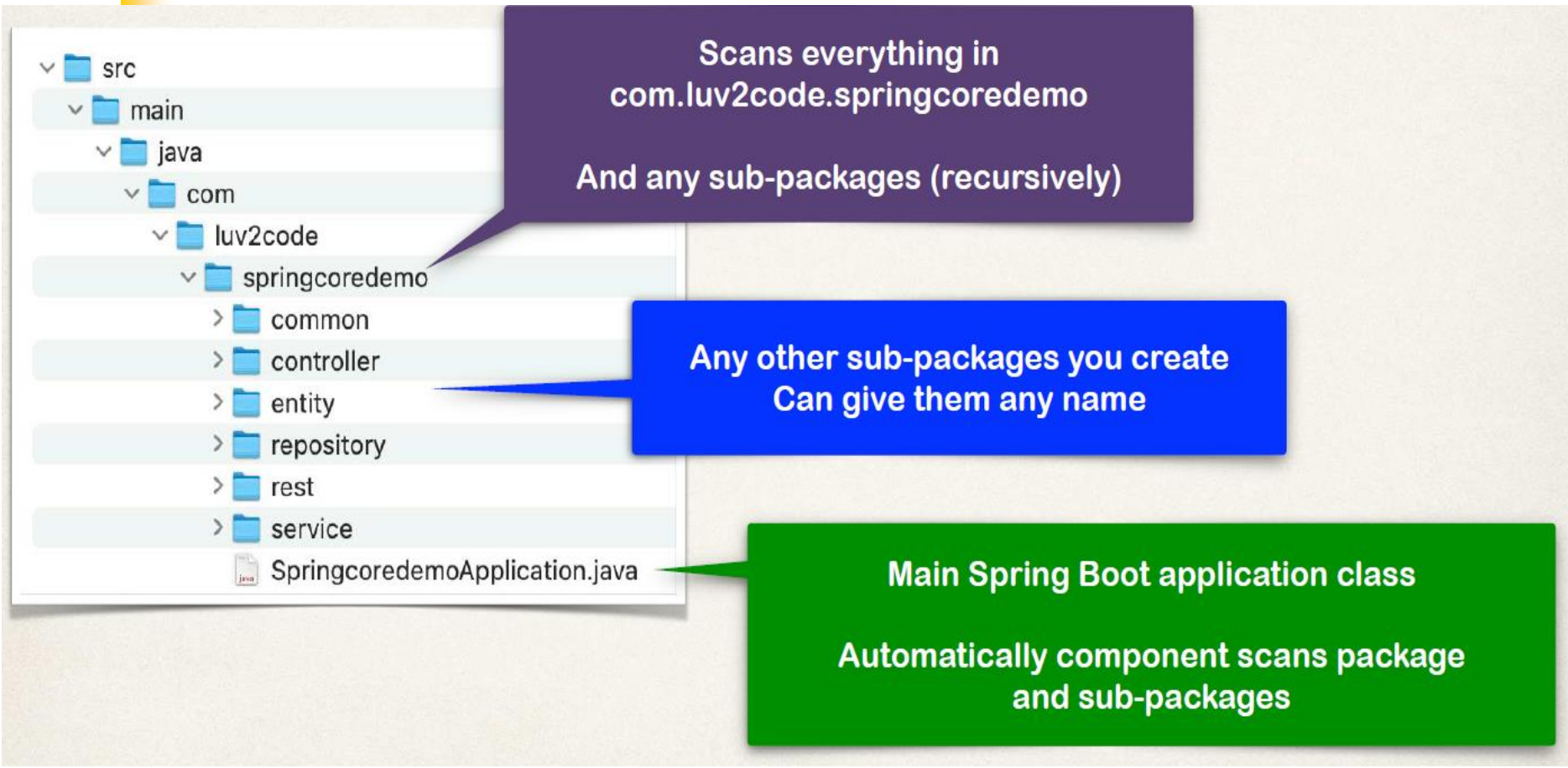
Bootstrap your Spring Boot application



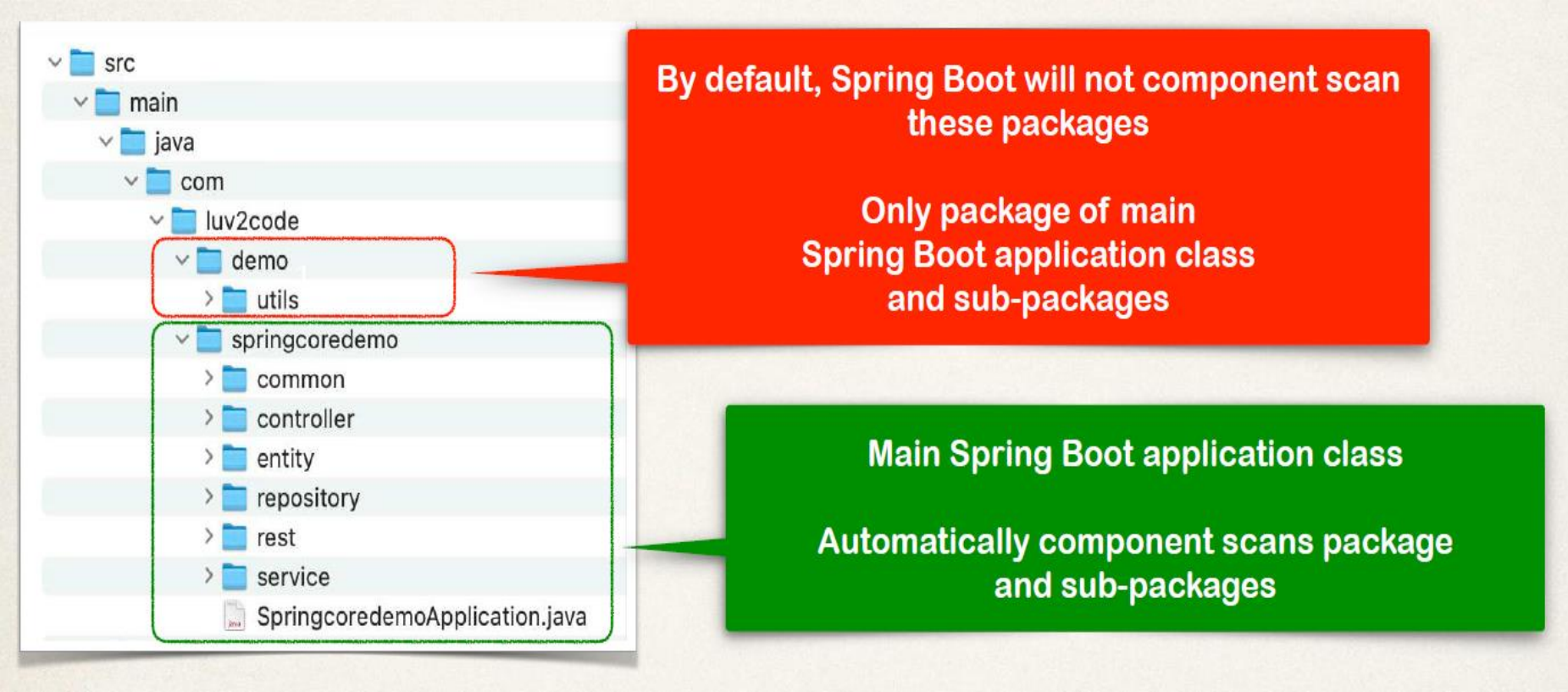
Component Scanning

- 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

Component Scanning



Common Pitfall - Different location

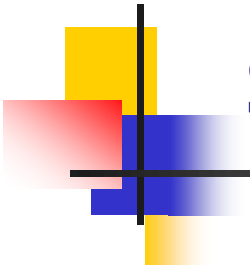


More on Component Scanning

- 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 {
  …
}
```



Spring Injection Types – Setter Injection

Inject dependencies by calling

setter method(s) on your class

Setter Injection

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

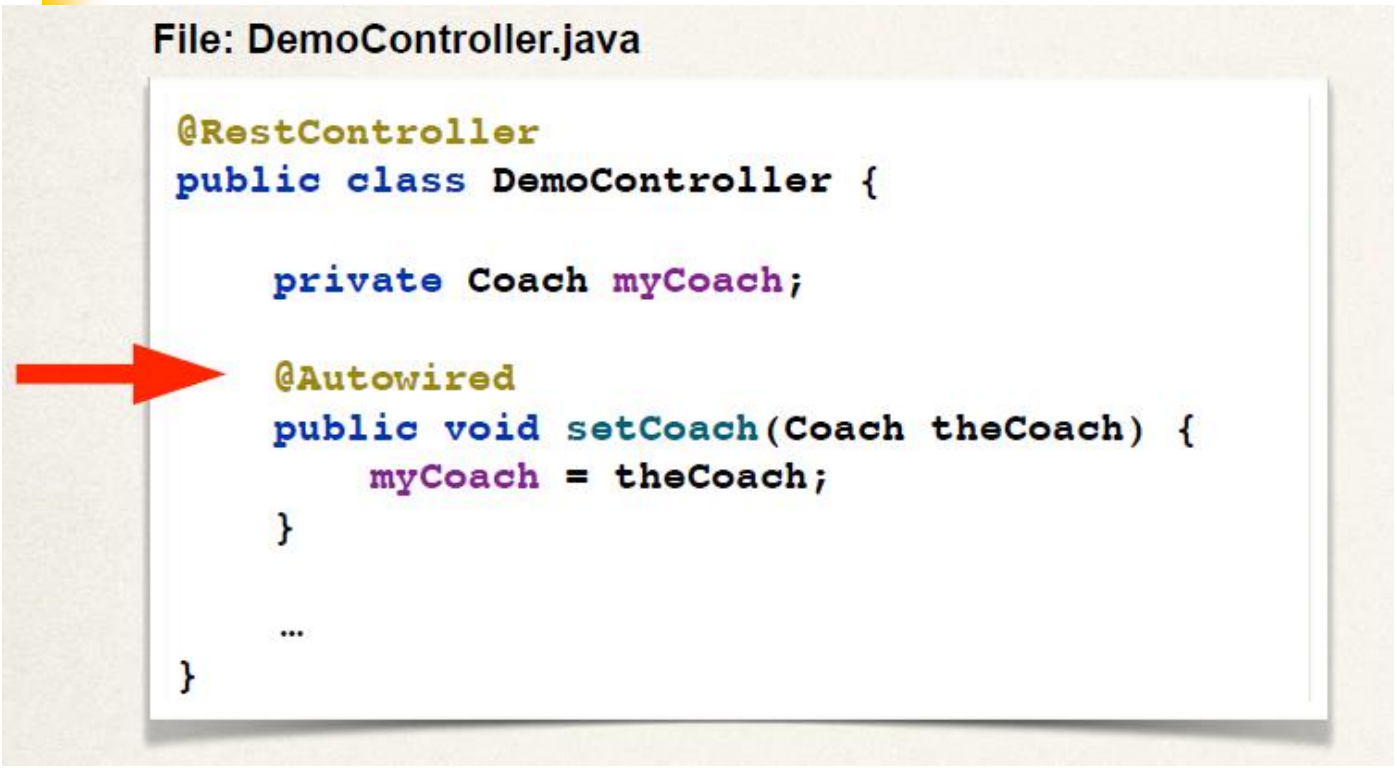
2. Configure the dependency injection with **@Autowired** Annotation

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



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

```
Coach theCoach = new CricketCoach();

DemoController demoController = new DemoController();

demoController.setCoach(theCoach);
```

Setter injection

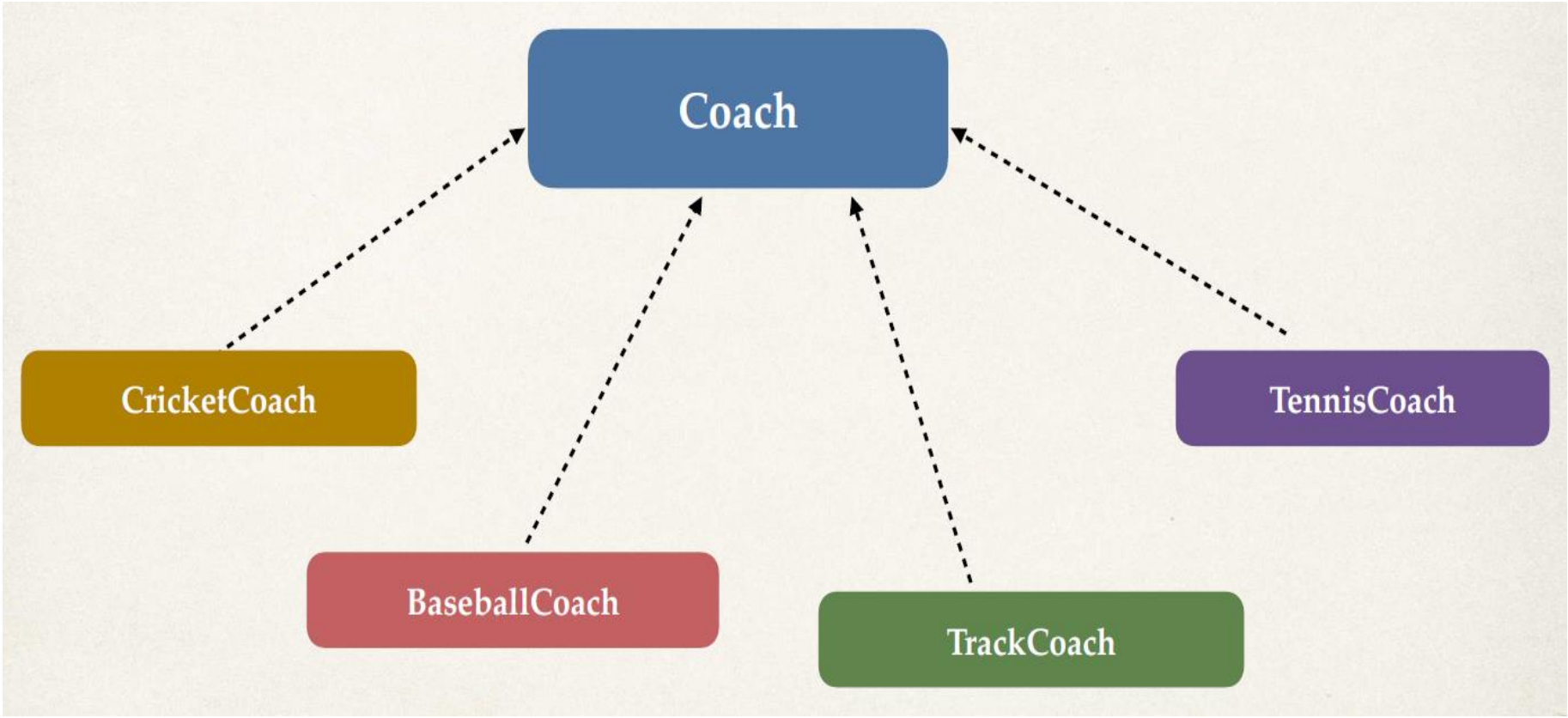
Step 2: Configure the dependency injection with Autowired Annotation

File: DemoController.java

```
@RestController
public class DemoController {

    private Coach myCoach;

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

    ...
}
```

Can use any method name

Multiple Coach Implementations



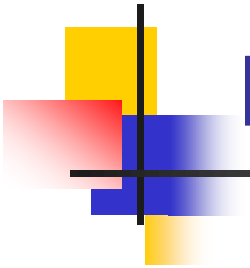
Multiple Coach Implementations

```
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";
    }
}
```

```
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";
    }
}
```

```
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!";
    }
}
```

```
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";
    }
}
```



Problem

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

- baseballCoach
- cricketCoach
- tennisCoach
- trackCoach

Solution: - @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 - @Qualifier

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

```
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();
    }
}
```

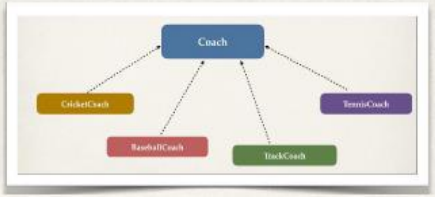
Give bean id: cricketCoach

Same name as class, first character lower-case

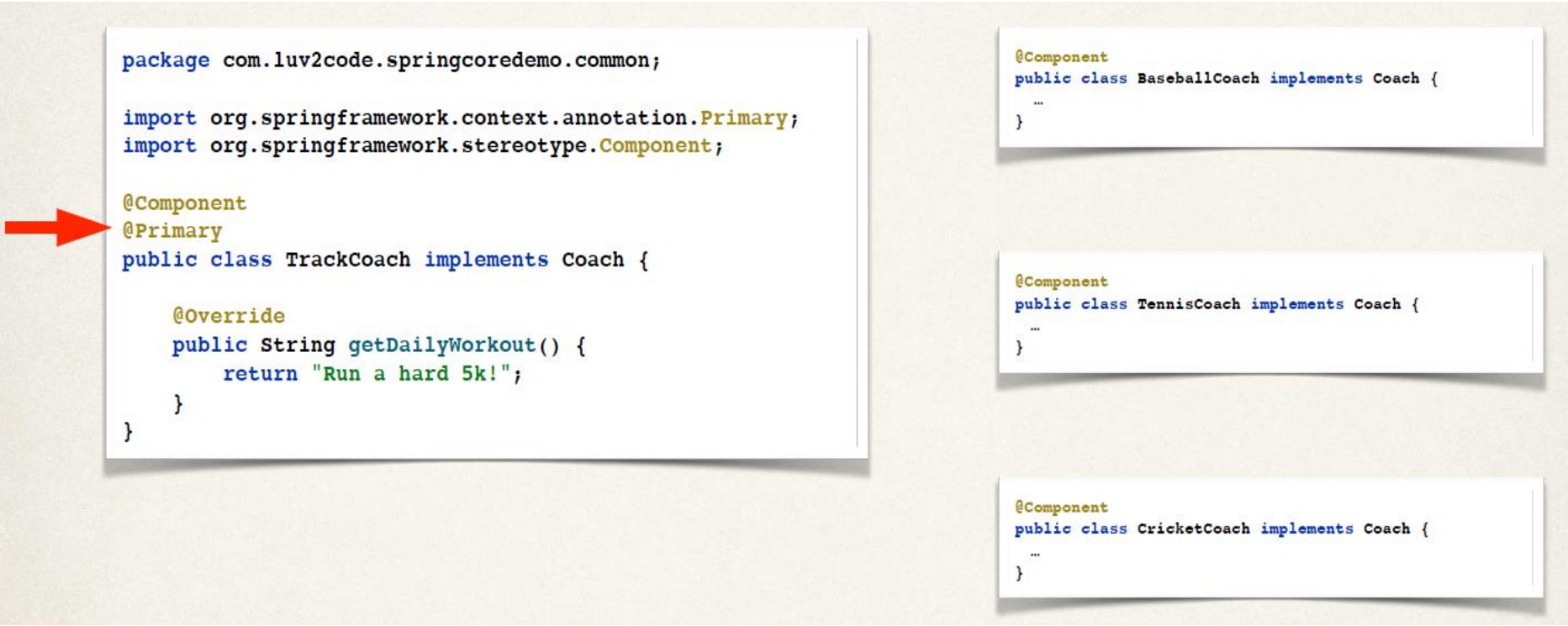
Other bean ids we could use:
baseballCoach, trackCoach, tennisCoach

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



```java
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!";
    }
}
```

```java
@Component
public class BaseballCoach implements Coach {
  ...
}
```

```java
@Component
public class TennisCoach implements Coach {
  ...
}
```

```java
@Component
public class CricketCoach implements Coach {
  ...
}
```

Resolved with @Primary

```java
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

```java
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!";
    }
}
```

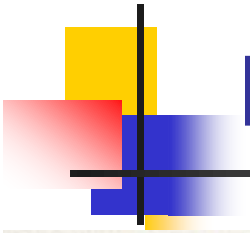
Mixing @Primary and @Qualifier

```java
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

```java
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!";
    }
}
```



Lazy Initialization

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

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());
    }
    ...
}
```

```
...
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

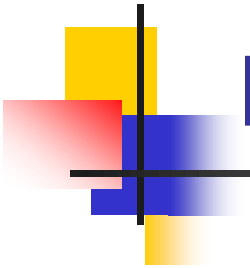
```
import org.springframework.context.annotation.Lazy;
import org.springframework.stereotype.Component;

@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

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

println to DemoController constructor

```java
@Component
public class CricketCoach implements Coach {

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

```java
@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

- 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

- 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 is singleton

Spring Bean Scopes

1. **Singleton (Default)**

   •Only **one instance** per Spring container.

```
@Component
@Scope("singleton")
class SingletonBean {
    public SingletonBean() {
        System.out.println("SingletonBean instance created.");
    }
}
```

Spring Bean Scopes

2. Prototype

A **new instance** is created every time it's requested. Use when you need a new instance every time, even outside web context like Desktop applications, Command-line applications

```
@Component
@Scope("prototype")
class PrototypeBean {
    public PrototypeBean() {
        System.out.println("PrototypeBean instance created.");
    }
}
```

Spring Bean Scopes

3. Request, Session, Application Scopes (For Web Apps)

These scopes help manage **stateful beans** in the context of an HTTP request, session, or the entire web application lifecycle.

3.1. @RequestScope (New Bean per HTTP Request)

3.2. @SessionScope (New Bean per User Session)

3.3 @ApplicationScope (Single Bean for Entire Web Application)

Spring Bean Scopes (3.1 @RequestScope)

Bean Definition

```java
@Component
@RequestScope
public class RequestScopedBean {
    private final String requestId;

    public RequestScopedBean() {
        this.requestId = UUID.randomUUID().toString();
        System.out.println("New RequestScopedBean instance created: " + requestId);
    }

    public String getRequestId() {
        return requestId;
    }
}
```

Controller Using RequestScopedBean

```java
@RestController
@RequestMapping("/request-scope")
public class RequestScopeController {

    @Autowired
    private RequestScopedBean requestScopedBean;

    @GetMapping
    public String getRequestId() {
        return "Request ID: " + requestScopedBean.getRequestId();
    }
}
```

Each request will generate **a new instance** of RequestScopedBean with a different UUID.

Spring Bean Scopes (3.2 @SessionScope)

- A new bean instance is created **per HTTP session**.
- The instance is **shared across multiple requests** from the same user.
- The bean is discarded when the session **expires or is invalidated**.
- Useful for **storing user session data** like authentication state, shopping carts, etc.

Spring Bean Scopes (3.2 @SessionScope)

Bean Definition

```
@Component
@SessionScope
public class SessionScopedBean {
    private final String sessionId;

    public SessionScopedBean() {
        this.sessionId = UUID.randomUUID().toString();
        System.out.println("New SessionScopedBean instance created: " + sessionId);
    }

    public String getSessionId() {
        return sessionId;
    }
}
```

Controller Using SessionScopedBean

```
@RestController
@RequestMapping("/session-scope")
public class SessionScopeController {

    @Autowired
    private SessionScopedBean sessionScopedBean;

    @GetMapping
    public String getSessionId(HttpSession session) {
        return "Session ID: " + sessionScopedBean.getSessionId();
    }
}
```

Testing @SessionScope

- Open your browser and access: http://localhost:8080/session-scope
- Refresh the page multiple times. **The session ID will remain the same.**
- Open another browser and access the same URL.
  - A **new session ID** will be generated for the new session.

Spring Bean Scopes (3.3 @ApplicationScope)

- The bean is created **once per web application** and shared across **all users and requests**.
- The instance lives **as long as the application is running**.
- Useful for **caching configuration settings, maintaining global state, or application-wide counters**.

Spring Bean Scopes (3.3 @ApplicationScope)

Bean Definition

```java
@Component
@ApplicationScope
public class ApplicationScopedBean {
    private final String applicationId;

    public ApplicationScopedBean() {
        this.applicationId = UUID.randomUUID().toString();
        System.out.println("New ApplicationScopedBean instance created: " + applicationId);
    }

    public String getApplicationId() {
        return applicationId;
    }
}
```

Controller Using ApplicationScopedBean

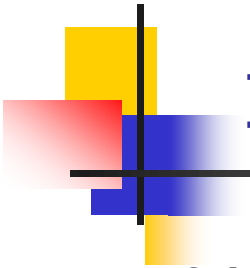
```java
@RestController
@RequestMapping("/application-scope")
public class ApplicationScopeController {

    @Autowired
    private ApplicationScopedBean applicationScopedBean;

    @GetMapping
    public String getApplicationId() {
        return "Application ID: " + applicationScopedBean.getApplicationId();
    }
}
```

Testing @ApplicationScope

- Access http://localhost:8080/application-scope multiple times.
- The **Application ID will remain the same**, even if you restart the browser or open a new session.
- Restarting the **Spring Boot server** will create a **new ApplicationScopedBean** instance.

Spring Bean Scopes

Scope	Instance Created	Lifetime	Use Case
`@RequestScope`	**Per HTTP request**	**Until request ends**	Temporary request-specific data (e.g., form validation, API requests)
`@SessionScope`	**Per user session**	**Until session expires**	User-specific data (e.g., authentication, shopping cart)
`@ApplicationScope`	**Once per application**	**Until app restarts**	Global data shared across all requests (e.g., configuration, analytics)



1. Stateful Beans

Stateful bean maintains **instance variables** (state) across multiple requests or interactions. Each client gets a separate instance or a shared instance that **holds data relevant to the session or transaction**.

Characteristics of Stateful Beans

✔ Holds **instance variables** that store **client-specific** data.

✔ Can be used for **shopping carts, user sessions, and transactions**.

✔ **Not thread-safe** because the state is modified during different interactions.

1. Stateful Beans Example

Bean Definition (Stateful)

```java
@Component
@SessionScope // Each user session gets a separate instance
public class ShoppingCart {
    private final List<String> items = new ArrayList<>();

    public void addItem(String item) {
        items.add(item);
    }

    public List<String> getItems() {
        return items;
    }
}
```

The item list persists **throughout the session**! If you **restart the session**, the shopping cart starts **empty** again.

Controller Using ShoppingCart

```java
@RestController
@RequestMapping("/cart")
public class ShoppingCartController {

    @Autowired
    private ShoppingCart shoppingCart;

    @GetMapping("/add/{item}")
    public String addItem(@PathVariable String item) {
        shoppingCart.addItem(item);
        return "Item added: " + item;
    }

    @GetMapping("/items")
    public List<String> getItems() {
        return shoppingCart.getItems();
    }
}
```

2. Stateless Beans

Stateless bean does not maintain any client-specific data across requests. Each method execution is **independent** and does not rely on previous interactions.

Characteristics of Stateless Beans

✔ Does **not store instance variables** across requests.

✔ Used for **service layers, utility methods, and authentication**.

✔ **Thread-safe** because it does not maintain state.

2. Stateless Beans Example

Bean Definition (Stateless)

```
@Component
public class CalculatorService {
    public int add(int a, int b) {
        return a + b;
    }
}
```

Every call is **independent** and does not rely on previous requests.

No data is stored in memory **between** calls.

Controller Using CalculatorService

```
@RestController
@RequestMapping("/calculator")
public class CalculatorController {

    @Autowired
    private CalculatorService calculatorService;

    @GetMapping("/add")
    public String add(@RequestParam int a, @RequestParam int b) {
        return "Result: " + calculatorService.add(a, b);
    }
}
```

Spring Boot Application Flow

1. **Component Scanning** – Finds classes annotated with @Component, @Service, etc.

2. **Dependency Injection** – Injects dependencies automatically.

3. **Creates Application Context** – Registers and manages beans.

4. **Starts Embedded Server (Tomcat, Jetty, etc.)** – Runs the application.

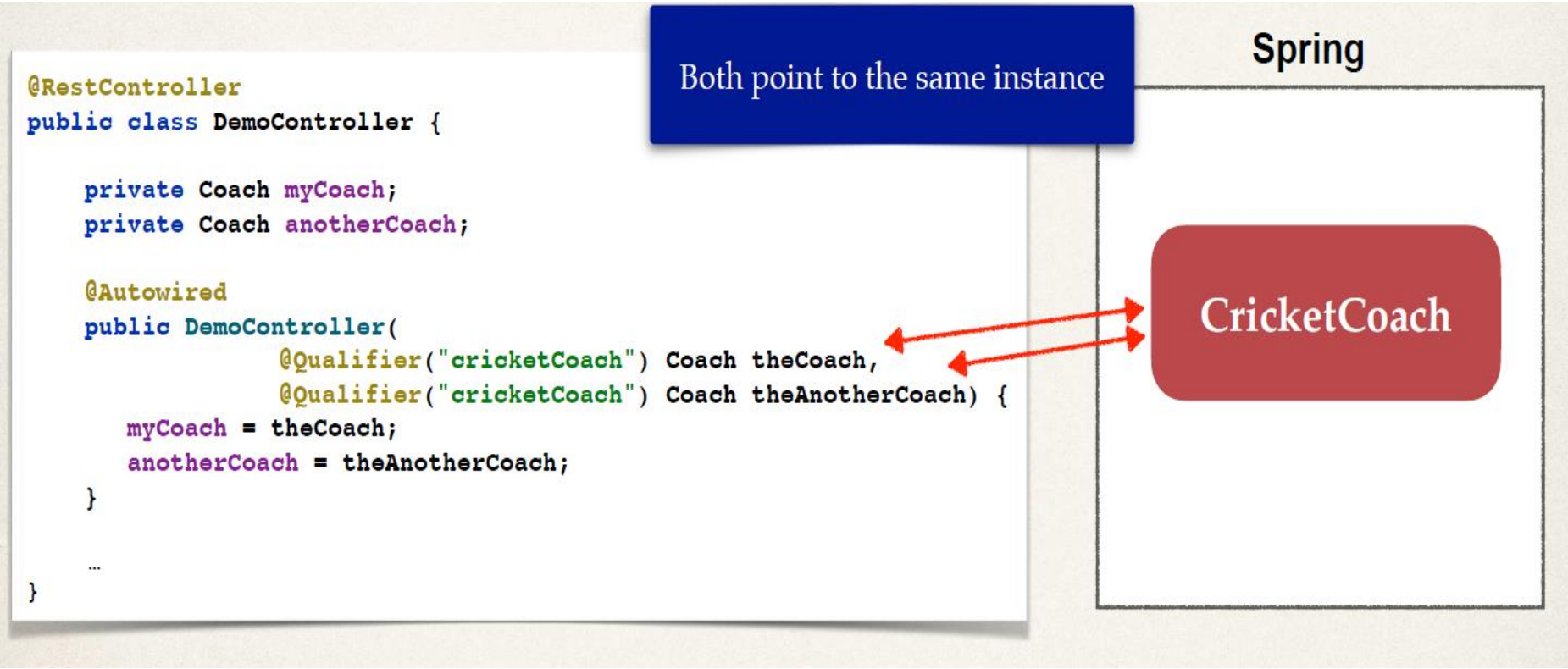
IoC in Spring → Transfers control of object creation to the framework.
Dependency Injection (DI) → Helps achieve **loose coupling**.
Spring AutoWiring → Automatically injects dependencies.
Bean Scopes → Control bean lifecycle.

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;
    }

    ...
}
```



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.
application	Scoped to a web app ServletContext. Only used for web apps.
websocket	Scoped to a web socket. Only used for web apps.

Prototype Scope Example

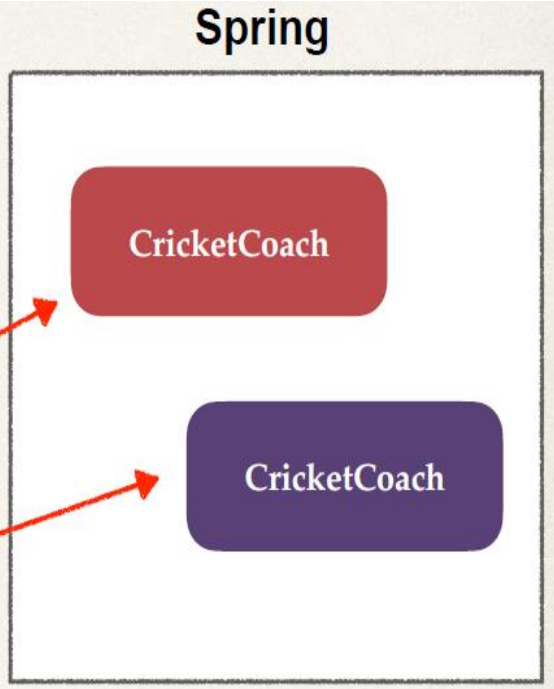
Prototype scope: new object instance for each injection

```
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