Java

Friday, November 21, 2025 11:54 AM

Jframe class

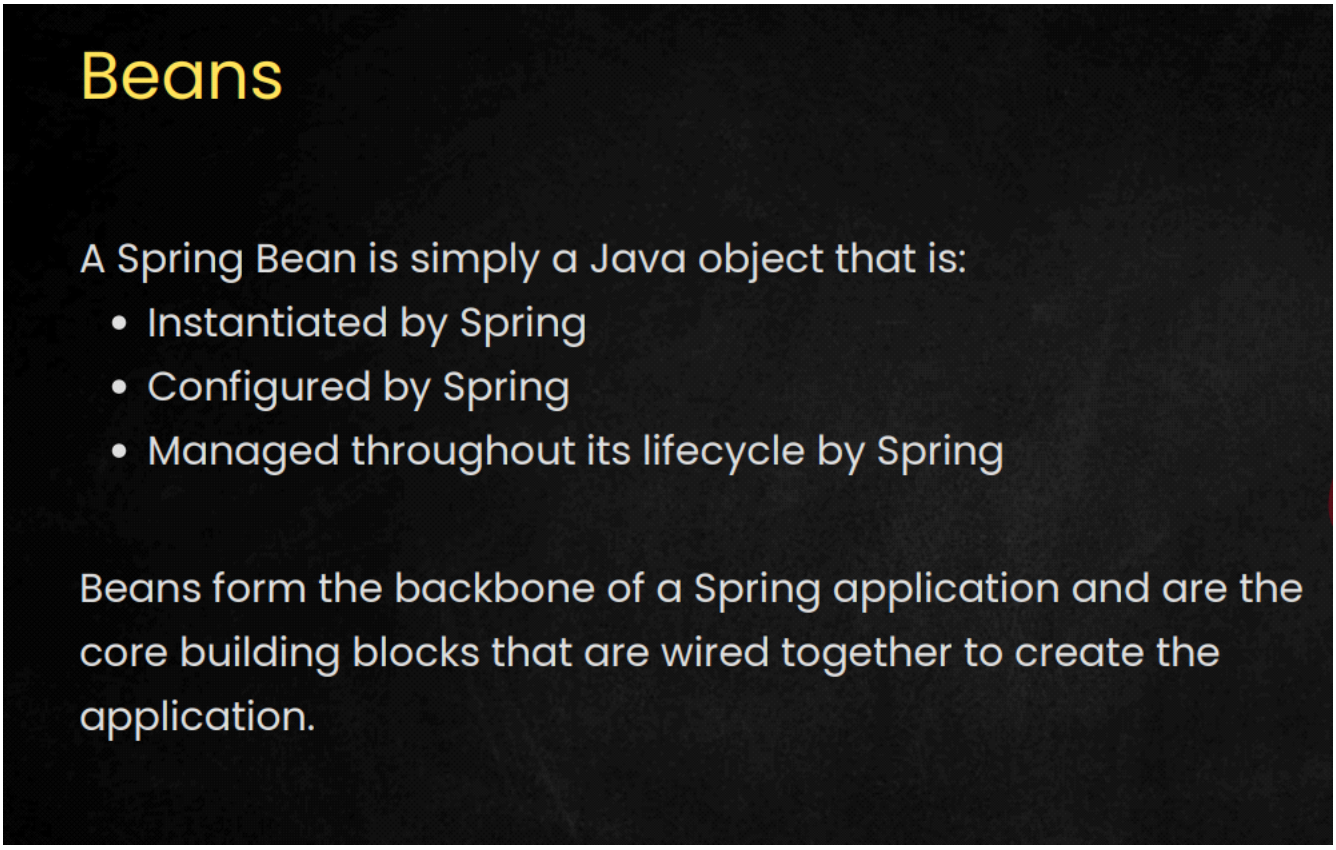
Jtext

Javax -> functionalities beyond JDK -> namespace -> javax.swing , javax.servlet

Object

```
class OrderService {
    private PaymentService paymentService = new PaymentService(); // tightly coupled
}
```

Apring web dependency for now on spring initialisr



What is IoC?

- **Inversion of Control** means that instead of your code controlling object creation and dependency management, the framework (Spring) takes over that responsibility.
- This is achieved through **Dependency Injection (DI)** — objects declare their dependencies, and Spring injects them at runtime.

BeanFactory vs ApplicationContext

- **BeanFactory**:
  - The simplest container in Spring.
  - Provides basic DI functionality.
  - Lazily initializes beans (creates them only when requested).
- **ApplicationContext**:
  - A superset of BeanFactory.
  - Adds enterprise-level features like:
    - **Internationalization (i18n)**
    - **Event propagation**
    - **Bean lifecycle management**
    - **Automatic BeanPostProcessor registration**
  - Eagerly initializes singleton beans by default.

Why ApplicationContext is central?

- It acts as the **IoC container** that:
  - Loads bean definitions from configuration (XML, annotations, or Java config).

- Manages the lifecycle of beans.
- Resolves dependencies automatically.

Common Implementations of ApplicationContext

- ClassPathXmlApplicationContext – loads context from XML in classpath.
- AnnotationConfigApplicationContext – loads context from Java-based configuration.
- WebApplicationContext – specialized for web applications.

IoC means **your code does not create and manage dependencies directly**.

Instead, the **framework (Spring)** controls object creation and wiring. This is usually implemented via **Dependency Injection (DI)**.

Instead of doing:

Java

```
Engine engine = new Engine();
Car car = new Car(engine);
```

Show more lines

We let **Spring inject the dependency**:

XML

```
<!-- beans.xml -->
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans.xsd">

 <bean id="engine" class="Engine"/>
 <bean id="car" class="Car">
 <constructor-arg ref="engine"/>
 beans>
```

Show more lines

Absolutely! In Spring, you can create and manage beans in two primary ways:

1. **Using @Component and stereotype annotations** (automatically discovered via component scanning)
2. **Using @Bean methods inside a @Configuration class** (explicitly declared, great for fine-grained control or third-party classes)

Below are lucid explanations with runnable-style code samples for both.

1) Creating Beans with @Component (and Stereotype Annotations)

Spring will **auto-detect** classes annotated with @Component (or its specializations) during **component scanning** and register them as beans in the ApplicationContext.

Common Annotations

- @Component → generic Spring-managed bean
- @Service → business/service layer semantics
- @Repository → persistence/DAO semantics (adds exception translation)
- @Controller → MVC web controller (returns views)
- @RestController → REST controller (@Controller + @ResponseBody)
  All of these are beans; the stereotypes add intent and extra behaviors (e.g., @Repository enables @Repository exception translation if @EnableTransactionManagement / persistence exception translation is configured).

Minimal Example (Component Scanning)

Package structure

```
com.example.app
 ├─ config
 │   └─ AppConfig.java
 └─ domain
    ├─ Engine.java
    ├─ CarService.java
    ├─ CarRepository.java
    ├─ CarController.java
    └─ CarRestController.java
```

Config enabling component scan

```
// com/example/app/config/AppConfig.java
package com.example.app.config;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
@Configuration
@ComponentScan(basePackages = "com.example.app") // scan this package tree
public class AppConfig { }
```

Generic component

```
// com/example/app/domain/Engine.java
package com.example.app.domain;
```

```
import org.springframework.stereotype.Component;
@Component
public class Engine {
    public String start() {
        return "Engine started!";
    }
}
```

Service layer

```
// com/example/app/domain/CarService.java
package com.example.app.domain;
import org.springframework.stereotype.Service;
@Service
public class CarService {
    private final Engine engine;
    // Prefer constructor injection (testable, immutable)
    public CarService(Engine engine) {
        this.engine = engine;
    }
    public String drive() {
        return engine.start() + " Car is driving...";
    }
}
```

Repository layer

```
// com/example/app/domain/CarRepository.java
package com.example.app.domain;
import org.springframework.stereotype.Repository;
@Repository
public class CarRepository {
    public String findById(Long id) {
        // pretend to hit DB
        return "Car#" + id;
    }
}
```

Spring MVC Controller (returns view names)

```
// com/example/app/domain/CarController.java
package com.example.app.domain;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
@Controller
public class CarController {
    private final CarService carService;
    public CarController(CarService carService) {
        this.carService = carService;
    }
    @GetMapping("/car/view")
    public String viewCar(Model model) {
        model.addAttribute("status", carService.drive());
```

```
        return "carView"; // resolves to a template (e.g., Thymeleaf)
    }
}
```

REST Controller (returns JSON/text directly)

```
// com/example/app/domain/CarRestController.java
package com.example.app.domain;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class CarRestController {
    private final CarService carService;
    private final CarRepository carRepository;
    public CarRestController(CarService carService, CarRepository carRepository) {
        this.carService = carService;
        this.carRepository = carRepository;
    }
    @GetMapping("/api/car/drive")
    public String drive() {
        return carService.drive();
    }
```

Bootstrapping (non-Boot app for demo)

```
// Main.java
import com.example.app.config.AppConfig;
import com.example.app.domain.CarService;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class Main {
    public static void main(String[] args) {
        try (var ctx = new AnnotationConfigApplicationContext(AppConfig.class)) {
            CarService carService = ctx.getBean(CarService.class);
            System.out.println(carService.drive());
        }
    }
```

Notes & Best Practices

- **Constructor injection** is recommended; Spring 4.3+ auto-wires single constructors even without @Autowired.
- Use @Qualifier if there are multiple beans of the same type:
  public CarService(@Qualifier("sportEngine") Engine engine) { ... }
- Customize bean names:
  @Component("engineV1")
  public class Engine { ... }
- Control scope:
  @Service
  @Scope("prototype") // or "singleton" (default), "request", "session" in web contexts
  public class CarService { ... }

2) Creating Beans with @Bean in a @Configuration Class

Use this when:

- You need **precise control** over construction, initialization, or destruction
- You are wiring **third-party classes** (not annotated with Spring stereotypes)
- You want to **conditionally** define beans or choose profiles
- You need to **override** component-scanned beans

Example: Pure Java Config with @Bean

```
// com/example/app/config/AppConfig.java
package com.example.app.config;
import com.example.app.domain.Engine;
import com.example.app.domain.CarService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class AppConfig {
   // Simple bean
   @Bean
   public Engine engine() {
      return new Engine();
   }
   // Bean method DI via parameters (Spring resolves by type)
   @Bean
   public CarService carService(Engine engine) {
      return new CarService(engine);
   }
}
```

Main

```
import com.example.app.config.AppConfig;
import com.example.app.domain.CarService;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class Main {
   public static void main(String[] args) {
      try (var ctx = new AnnotationConfigApplicationContext(AppConfig.class)) {
         CarService carService = ctx.getBean(CarService.class);
         System.out.println(carService.drive());
      }
   }
```

Using @Bean for Third-Party Classes

```
// Suppose this is a third-party DataSource class not annotated with Spring
import javax.sql.DataSource;
import org.apache.commons.dbcp2.BasicDataSource;
@Configuration
public class DataConfig {
   @Bean
   public DataSource dataSource() {
      var ds = new BasicDataSource();
      ds.setUrl("jdbc:h2:mem:testdb");
      ds.setUsername("sa");
```

```
        ds.setPassword("");
        ds.setInitialSize(2);
        return ds;
    }
```

Initialization / Destruction Hooks

```
@Configuration
public class LifecycleConfig {
    @Bean(initMethod = "init", destroyMethod = "shutdown")
    public CacheManager cacheManager() {
        return new CacheManager();
    }
}
// Hypothetical class
class CacheManager {
    public void init() { System.out.println("Cache initialized"); }
    public void shutdown() { System.out.println("Cache shutdown"); }
```

Choosing Primary / Qualifiers / Profiles

```
@Configuration
public class EnginesConfig {
    @Bean
    @Primary
    public Engine defaultEngine() { return new Engine(); }
    @Bean(name = "sportEngine")
    public Engine sportEngine() { return new Engine(); }
}
// Usage:
public class CarService {
    private final Engine engine;
    public CarService(@Qualifier("sportEngine") Engine engine) {
        this.engine = engine;
    }

@Configuration
@Profile("prod")
public class ProdConfig {
    @Bean
    public DataSource prodDataSource() { /* prod settings */ }
}
@Configuration
@Profile("dev")
public class DevConfig {
    @Bean
    public DataSource devDataSource() { /* dev settings */ }
}
Activate a profile via properties or programmatically:

var ctx = new AnnotationConfigApplicationContext();
ctx.getEnvironment().setActiveProfiles("dev");
```

ctx.register(DevConfig.class, ProdConfig.class);

When to Use Which?

- **@Component / Stereotypes**:\ Best for your **own application classes** (services, repositories, controllers). Cleaner, less boilerplate, and aligns with layered architecture semantics.
- **@Bean in @Configuration**:\ Best for **third-party objects**, **explicit construction** logic, **conditional beans**, **profiles**, or when you need **fine-grained control** over lifecycle and wiring. In real projects, you'll often combine both: component scan for your layers + @Bean methods for infra and third-party integrations.

If you'd like, I can share a small **Spring Boot** version of the same examples (with @SpringBootApplication) to show how Boot auto-config + component scanning works out of the box.

You're right—let me walk you through the code step by step and **explain what each piece is doing** and **why** you would use it. I'll keep it crisp and lucid.

1) Beans via @Component & Stereotypes

What is happening conceptually?

- When you annotate classes with @Component (or @Service, @Repository, @Controller, @RestController), you're telling Spring: **"Please create and manage an instance of this class."**
- Spring finds these classes using **component scanning** (enabled via @ComponentScan) and registers them as beans in the ApplicationContext.
- Dependencies are **injected automatically** (via constructor injection in our examples).

Code and Explanation

Configuration

```
@Configuration
@ComponentScan(basePackages = "com.example.app")
```

- @Configuration: Marks this class as a source of bean definitions.
- @ComponentScan(...): Tells Spring where to look for annotated components. Any @Component, @Service, etc., under com.example.app will be discovered and registered.

A generic component: Engine

```
@Component
```

```
public class Engine {
    public String start() {
        return "Engine started!";
    }
```

- @Component: Registers Engine as a bean.
- start(): A simple method to prove the bean works.

A service with constructor injection: CarService

```
@Service
public class CarService {
    private final Engine engine;
    public CarService(Engine engine) {
        this.engine = engine;
    }
    public String drive() {
        return engine.start() + " Car is driving...";
    }
```

- @Service: Same as @Component but semantically indicates **business logic** layer.
- **Constructor injection**: Spring sees the constructor needs an Engine and injects the Engine bean it already created.
- drive(): Uses the injected dependency.
  Why constructor injection?\ It makes the class immutable, easier to test, and ensures dependencies are available at creation time. Spring (4.3+) will autowire the only constructor without needing @Autowired.

Repository

```
@Repository
public class CarRepository {
    public String findById(Long id) {
        return "Car#" + id;
    }
```

- @Repository: Marks persistence/DAO components. It can enable **exception translation** (converting low-level persistence exceptions into Spring's DataAccessException).

MVC Controller

```
@Controller
public class CarController {
    private final CarService carService;
    public CarController(CarService carService) {
        this.carService = carService;
    }
    @GetMapping("/car/view")
    public String viewCar(Model model) {
        model.addAttribute("status", carService.drive());
        return "carView";
    }
}
```

- @Controller: A web controller that returns **view names** (e.g., Thymeleaf templates).
- @GetMapping("/car/view"): Maps HTTP GET /car/view to this method.
- Model: Adds attributes to be used in the rendered view.
- return "carView";: The name of the template to render.

REST Controller

```
@RestController
public class CarRestController {
    private final CarService carService;
    private final CarRepository carRepository;
    public CarRestController(CarService carService, CarRepository carRepository) {
        this.carService = carService;
        this.carRepository = carRepository;
    }
    @GetMapping("/api/car/drive")
    public String drive() {
        return carService.drive();
    }
}
```

- @RestController: Equivalent to @Controller + @ResponseBody. **Returns the response directly** (JSON/text) rather than rendering a view.
- Perfect for APIs returning data for clients.

Bootstrapping (non-Boot)

```
try (var ctx = new AnnotationConfigApplicationContext(AppConfig.class)) {
    CarService carService = ctx.getBean(CarService.class);
    System.out.println(carService.drive());
}
```

- Creates an ApplicationContext using your configuration.
- Retrieves the CarService bean from the container.
- Calls drive() to see the wiring in action.
- Try-with-resources ensures the context is closed properly (calls destroy methods).

2) Beans via @Bean in a @Configuration Class

What is happening conceptually?

- You explicitly define bean creation **in Java methods** inside a @Configuration class.
- Useful for **third-party classes** or when you need **fine-grained control** (e.g., init/destroy methods, conditional beans, profiles).

Code and Explanation

Explicit bean definitions

```
@Configuration
public class AppConfig {
    @Bean
    public Engine engine() {
        return new Engine();
    }
    @Bean
    public CarService carService(Engine engine) {
        return new CarService(engine);
    }
}
```

- @Bean: Registers the returned object as a bean.

- carService(Engine engine): Spring resolves the parameter by type from the container and injects the engine() bean.
- This approach gives **complete control** over construction logic (you can pass custom parameters, conditionals, etc.).

Third-party class example (DataSource)

```
@Configuration
public class DataConfig {
   @Bean
   public DataSource dataSource() {
      var ds = new BasicDataSource();
      ds.setUrl("jdbc:h2:mem:testdb");
      ds.setUsername("sa");
      ds.setPassword("");
      ds.setInitialSize(2);
      return ds;
   }
}
```

- This is perfect when the class (e.g., BasicDataSource) is **not annotated** with Spring stereotypes and you still want it managed by the container.
- You configure properties in code before returning the bean.

Lifecycle hooks

```
@Configuration
public class LifecycleConfig {
   @Bean(initMethod = "init", destroyMethod = "shutdown")
   public CacheManager cacheManager() {
      return new CacheManager();
   }
}
class CacheManager {
   public void init() { System.out.println("Cache initialized"); }
   public void shutdown() { System.out.println("Cache shutdown"); }
```

- initMethod runs **after** the bean is constructed and dependencies injected.
- destroyMethod runs **when the context is closing** (e.g., graceful shutdown).
- Great for resources like caches, thread pools, or connections.

Choosing which bean to inject (@Primary, @Qualifier)

```
@Configuration
public class EnginesConfig {
   @Bean
   @Primary
   public Engine defaultEngine() { return new Engine(); }
   @Bean(name = "sportEngine")
   public Engine sportEngine() { return new Engine(); }
}
```

- @Primary: If there are multiple beans of the same type, **this one wins by default**.
- @Bean(name = "sportEngine"): Gives a specific name for use with @Qualifier("sportEngine") in injection points.

Environment-specific beans with @Profile

```
@Configuration
@Profile("prod")
public class ProdConfig {
    @Bean
    public DataSource prodDataSource() { /* prod settings */ }
}
@Configuration
@Profile("dev")
public class DevConfig {
    @Bean
    public DataSource devDataSource() { /* dev settings */ }
}
```

- @Profile: Only registers these beans when the **profile** is active (e.g., "dev" vs "prod").
- Activate programmatically:
  ```
  var ctx = new AnnotationConfigApplicationContext();
  ctx.getEnvironment().setActiveProfiles("dev");
  ctx.register(DevConfig.class, ProdConfig.class);
  ```

When to Use Which (Summary)

- **@Component / Stereotypes**:\ Best for your **own application classes** (services, repos, controllers). Clean and automatic via scanning.
- **@Bean in @Configuration**:\ Best for **third-party objects**, complex construction, lifecycle control, conditional registration, or environment-specific configs.
  In real-world apps, you **combine both**: use component scanning for your layers and @Bean methods for infrastructure.

Quick Visual Mental Model

- **Component Scanning**: "Find all annotated classes in these packages and create beans automatically."
- **@Bean methods**: "Create exactly these beans, with this construction logic."

If you want, I can convert these examples into a **Spring Boot** project structure and show how @SpringBootApplication automatically enables component scanning and simplifies bootstrapping. Or we can add **@Autowired, scopes, and qualifiers** examples inside controllers/services to see different DI flavors. What would you like next?