

➤ **Spring IoC Container (or Application Context)**

- ⌘ It is the **Bean Factory**.
- ⌘ All the center or everything is the ApplicationContext -- Spring's IoC.
- ⌘ Spring IoC Container (Inversion of Control Container) is the core part of the Spring Framework responsible for:
 - ⌘ creating objects (beans)
 - ⌘ managing their life cycle
 - ⌘ injecting dependencies
 - ⌘ configuring them using annotations/XML/Java config
- ⌘ The container controls your objects → NOT you.
- ⌘ In simple terms --- **The Spring IoC Container is the engine that creates and manages all Spring beans and performs dependency injection.**

➤ **Beans**

- ⌘ A Spring Bean is simply a Java object that is
 - ⌘ Instantiated by spring
 - ⌘ Configured by spring
 - ⌘ Managed throughout its life cycle by spring
- ⌘ Beans forms the backbone of a spring application and are the code building blocks that are wired together to create the application.
- ⌘ In simple terms --- **A Spring Bean is an object that Spring creates and manages inside the ApplicationContext.**
- ⌘ Analogy:
 - ⌘ Think of the IoC container as a restaurant kitchen:
 - ⌘ You (developer) write the recipe (class definitions).
 - ⌘ Spring (IoC container) cooks the dishes (creates objects).
 - ⌘ Spring serves the dishes wherever needed (dependency injection).
- ⌘ Classes annotated with the following annotations will automatically become **Spring Bean**:
 - ⌘ `@Component`
 - ⌘ `@Service`
 - ⌘ `@Repository`
 - ⌘ `@Controller`
 - ⌘ `@RestController`
- ⌘ If you want a class should be managed by Spring only; you don't want to handle that by your own, then use **`@Component` annotation.**

```

@Component 1 usage
public class PaymentService {
    public void pay() { 1 usage
        System.out.println("Paying...");
    }
}

```

It is the **bean**. We created the bean now. But still we need to inject it.

➤ To inject the **bean**, use the **@Autowired** annotation.

```

@Autowired 1 usage
PaymentService paymentServiceObj;

```

@Autowired tells *Spring*: “Give me the bean for this dependency from the [IoC container](#).”

```

@Autowired 1 usage
PaymentService paymentServiceObj;

public static void main(String[] args) {
    SpringApplication.run(Module1IntroductionApplication.class, args);
}

@Override
public void run(String... args) throws Exception {
    paymentServiceObj.pay();
}

```

You can see here, we didn't initialize the **paymentServiceObj** by our own. But it works because *Spring* created the bean (**@Component**) and it was injected here (**@Autowired**).

➤ There is an interface **CommandLineRunner** which is an *functional interface* containing one abstract function **run**.

➤ This **run** method is executed by **Spring framework** by itself after all the **beans** are created.

➤ **CommandLineRunner** in *Spring Boot* is a functional interface that lets you run some code immediately after the *Spring Boot* application starts, right after the *ApplicationContext* is fully initialized.

- Lets say you want to take control of the method of creation of Beans (Spring managed Objects).
 - ⌘ Lets say the Class (out of which Bean will be created by Spring) is having some parameterized constructor and you want to take control and saying Spring that you need to pass this and that Object while creating the Bean (or creating the Object) of this class.
 - ⌘ In this case you need to create one class using the annotation **@Configuration**.
 - ⌘ The class having **@Configuration** will be treated as the **Bean** creation rule-book.
 - ⌘ Inside this class, write methods which will return the Object of the particular classes you want to make Bean from. (classes means those Bean classes).
 - ⌘ These methods should be written with the annotation **@Bean**.

```
@Configuration no usages
public class AppConfig {

    @Bean no usages
    public PaymentService paymentService() {
        System.out.println("customized bean creation");
        return new PaymentService();
    }
}
```

- ⌘ Here I don't have any parameterized constructor, I just gave example of how to customize creation of Beans.
- ⌘ **NOTE:**
 - ⌘ You are taking control of only the way of creation. Creation & Injection will be handled by Spring only.
 - ⌘ No need of giving annotations like **@Component**, **@Service** ..etc before the Bean class (class from which beans will be created. here: **PaymentService** class) when you are creating bean with **@Configuration** and **@Bean** annotation.
 - * Even if those annotation is there, **@Bean** will override those.

➤ Bean Life-cycle

- ⌘ *Bean Created ==> Dependency Injected ==> Bean Initialized ==> Bean is Used ==> Bean is Destroyed*
- ⌘ 2 Bean life-cycle methods are there: **@PostConstruct**, **@PreDestroy**.
- ⌘ As the name suggests, **@PostConstruct** will be called right after the **Bean** is initialized (between *bean initialized* and *bean is used*).

- **@PreDestroy** is called after the bean being used and before destroying the bean.
(between *bean is used* and *bean is destroyed*)

```
public class PaymentService { 3 usages
    public void pay() { 1 usage
        System.out.println("Paying...");
    }

    @PostConstruct no usages
    public void postConstruct() {
        System.out.println("Post Construct");
    }

    @PreDestroy no usages
    public void preDestroy() {
        System.out.println("Pre Destroy");
    }
}
```

- I added 2 methods using those *annotations* .

```
2025-12-07T02:01:58.882+0
customized bean creation
Post Construct
2025-12-07T02:01:59.217+0
2025-12-07T02:01:59.225+0
Paying...
```

- Customized bean creation was mentioned in the **@Bean** method.
- So you can see, the *Pre Destroy* is not called yet; because it is run before destroying the Bean.

```
customized bean creation
Post Construct
2025-12-07T02:01:59.217+0
2025-12-07T02:01:59.225+0
Paying...
2025-12-07T02:04:31.679+0
2025-12-07T02:04:31.692+0
Pre Destroy
```

- I stoped the run and it was executed.

➤ Bean Scope

- By default the bean scope is **Singleton** (one instance of bean will be created and shared everywhere).

```
@Autowired 1 usage
PaymentService paymentServiceObj;

@Autowired 1 usage
PaymentService paymentServiceObj2;

public static void main(String[] args) { SpringApplication

@Override
public void run(String... args) throws Exception {
    System.out.println(paymentServiceObj.hashCode());
    System.out.println(paymentServiceObj2.hashCode());
}
```

- Created 2 instance of this bean.

```
1843853990
```

```
1843853990
```

- * But if you see both are same. Because by default Spring created Singleton.

- Now I am changing the scope to **prototype**.

- It means it'll create one instance per object.

```
@Configuration no usages
public class AppConfig {

    @Bean no usages
    @Scope("prototype")
    public PaymentService paymentService() {
        System.out.println("customized bean creation");
        return new PaymentService();
    }
}
```

- You can write like this using the annotation **@Scope**

```
837249677
```

```
1997270773
```

- * Now the hashcodes are different because both are different beans.

➤ Dependency Injection

- ⌘ **DI** is a design pattern where an object receives its dependencies from the outside, instead of creating them itself.
- ⌘ Lets say we have one interface **NotificationService**.
- ⌘ 2 classes are there which implement this interface: **EmailNotificationService**, **SmsNotificationService**.
- ⌘ Now we can create one object of type **NotificationService** and instantiate any **type of NotificationService (either email or sms)**.
- ⌘ So, now lets make these classes as bean by adding the annotation **@Component**.
- ⌘ But now what will happen, Spring will be confused about which Bean (i.e. **EmailNotificationService** or **SmsNotificationService**) should be injected.
- ⌘ So for this, we can make one Class as **primary** so that when Spring will be having more than 1 option, it'll go ahead with the **Primary**.

```
@Component 1 usage
@Primary
public class EmailNotificatio
@Override 1 usage
```

- ⌘ Before we were using **@Autowired** to inject the Bean. But that is not preferable because that is not safe.
- ⌘ It should be injected through the constructor.

```
NotificationService notificationServiceObj; 2 usages

public Module1IntroductionApplication (NotificationService notificationServiceObj) { no usages new *
    this.notificationServiceObj = notificationServiceObj;
}
```

- ⌘ Here you can see, I have not mentioned **@Autowired** still it is working. Because by default, if the class is having only one constructor, then Spring will automatically treat that constructor like **@Autowired**
- ⌘ The above is same as below:

```
@Autowired no usages new *
public Module1IntroductionApplication (NotificationService notific
    this.notificationServiceObj = notificationServiceObj;
}
```

- Types of dependency injection:
 - Constructor** ==> **@Autowired** (optional) ==> **Highly recommended**
 - Setter** ==> **@Autowired** (required) ==> **Ok ok**
 - Field** ==> **@Autowired** (required) ==> **Avoid in production**
- We were previously using the **Field** type of dependency injection.
- Lets say you have 2 similar Beans but you don't want to make anyone Primary (discussed above) then you can use **@Qualifier**

```
@Component 1 usage
@Qualifier("email-notification")
public class EmailNotificationService implements NotificationService {
```

```
public Module1IntroductionApplication ( no usages new *
    @Qualifier("email-notification") NotificationService notificationServiceObj) {
    this.notificationServiceObj = notificationServiceObj;
}
```

* You can use the qualifier like this.

- You can also use conditional selection of Bean using **application.properties** file.

```
notification.type=email
```

* Write some key value pair in *application.properties* file.

```
@Component no usages
@ConditionalOnProperty(name = "notification.type", havingValue = "sms")
public class SmsNotificationService implements NotificationService {
```

In sms class: **@ConditionalOnProperty(name = "notification.type",
havingValue = "sms")**

In Email class: **@ConditionalOnProperty(name = "notification.type",
havingValue = "email")**

- Now no need of specifying the **@Qualifier** in the constructor. By default it'll take depending upon the value of *application.properties* file.
- If you want to inject all the beans i.e. if you want to send both **email** as well as **sms** then you can use **Map**

```
@Autowired 1 usage
Map<String, NotificationService> notificationServiceMap = new HashMap<>();
```

```
@Autowired 1 usage
Map<String, NotificationService> notificationServiceMap;
```

* This one is more preferable.



- **F**
- **Ffdfd**
- **F**
- **F**
- **F**
- **F**
- **Ff**
-