INTRODUCTION TO SPRING BOOT

Introduction

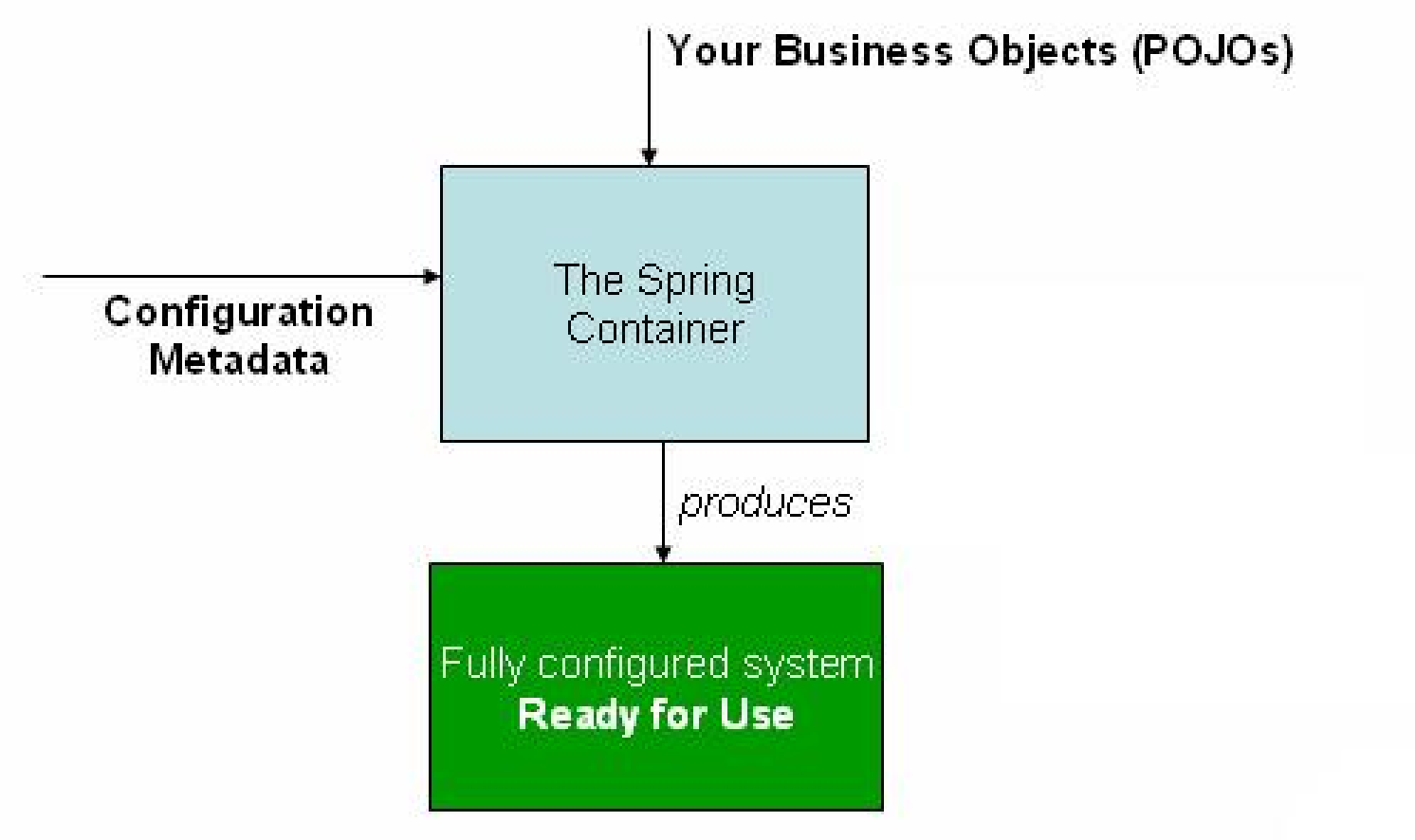
- **Spring**
  - An application development **framework** and **inversion of control** container for Java
- **Spring Boot**
  - Makes it easy to create stand-alone, production-grade **Spring Applications** and expose them **as services**

Spring Framework

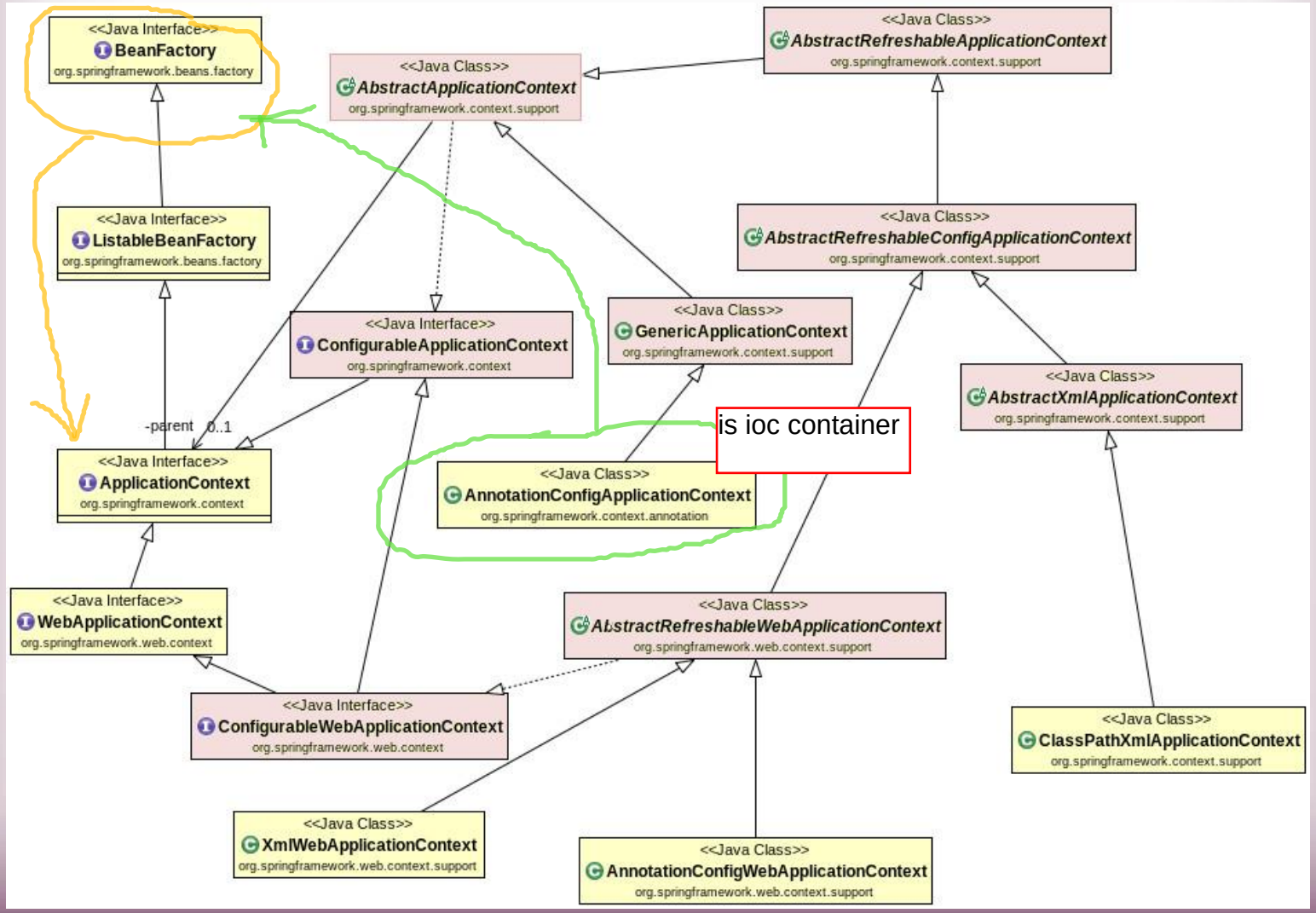
- Light-weight **comprehensive framework for building Java SE and Java EE applications**
- **Java Bean-based configuration** management,
- Integration with **persistence** frameworks.
- **MVC** web application framework
- **Aspect-oriented programming** (AOP) framework
- Publishing REST API's

IoC CONTAINER

The Container



Container Hierarchy



Beans

- ***A bean** is an object that is instantiated, assembled, and otherwise **managed by** a Spring IoC container.*

beans are single ton container ie one instance can be created

- The "**beans**" are in the form of JavaBeans
  - ***Present in a Named Package***
  - ***No Argument constructor***
  - ***getter and setter methods for the properties***
  - ***Can be controlled for scope, life cycle and callbacks.***

- **Beans are singletons** by default

Configuration Meta Data

- Instructs IoC Container to instantiate, configure, and assemble the objects
- Can be done in **XML format**
  - Supported from Spring 2.0
- Can be done with **Annotation**
  - Supported from Spring 2.5
- Can be done with Just **Java**
  - Supported from Spring 3.0

Dependency Injection

- Dependencies between the beans can be Injected by following ways
  - Setter Injection
    - uses setter method to set the value
  - Constructor Injection
    - Happens at the time of creating the object itself
  - Field Injection (@Autowired at field)
    - Uses reflection to set the values of private variables

```
@Autowired
private UserService userService;
```

Constructor Injection

- Dependency can be one of the following
  - Injected via a class constructor
  - primitive and String-based values
  - Dependent object (contained object)
  - Collection values

```
@Component
public class UserController {
  private  UserService userService;
  public UserController(UserService userService){
    this.userService = userService;
  } }
```

Setter Injection

- **Setter Injection**
  - Injected via setter methods
  - Allows flexible initialization
  - Requires Java Bean conventions to be followed
    - Can add post initialization checking methods

```
private UserService userService;

@Autowired

public void setUserService(UserService userService){

  this.userService = userService;

}
```

REGISTERING BEANS

@ Component

- **@Component**
  - Spring can scan all the beans through **auto scan if the class has this or similar annotation**
- The Registered Beans can be access by Its name
  - *The Name is first Character of the Class Name in lowercase*
  - **'CustomerService' to 'customerService'.**
- CustomerService cust =

  (CustomerService)context.getBean("customerService");
- **@Component("custService")**
  - Can Customize the component name by passing a string value

@Bean

- Method-level annotation in Class annotated @Configuration
- Added to the public Method of the configuration class
  - **Methods should not be private or final**
- Method should return an object that should be registered as a Bean
- Supports following attributes
  - init-method
  - destroy-method
  - autowireCandidate
  - name

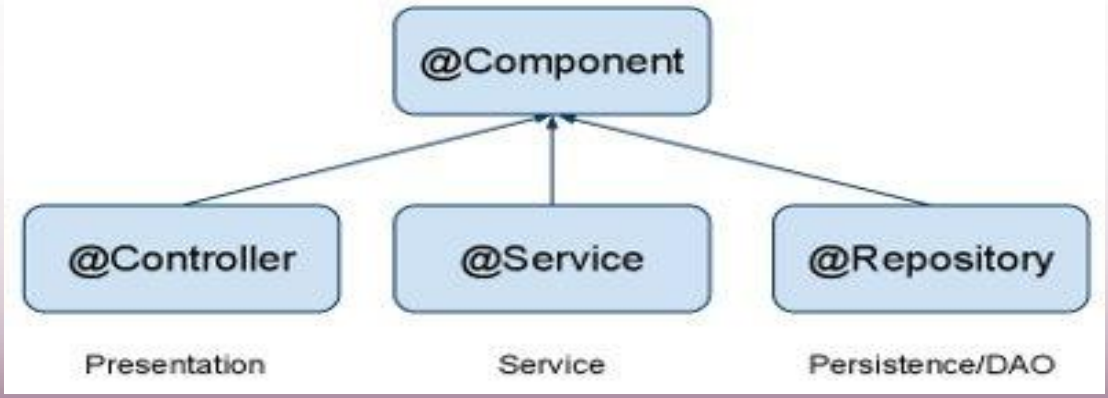
Register Bean

- Beans can also be registered with Java Based Configuration
  - Done in the Java class with @Configuration Annotation
    - **This class acts as a source of Bean Definitions**

```
@Configuration
public class EmployeeConfig {

 @Bean
 public Employee idOfTheBean(){
     return new Employee();
  }
 @Bean (name="ram")  // bean Id overridden with name attribute
 public Employee employee(){
     return new Employee();
  }
}
```

Auto Component Scan Types

- **@Component**
  - Indicates a auto scan component.
- **@Repository**
  - Indicates DAO component in the persistence layer.
- **@Service**
  - Indicates a Service component in the business layer.
- **@Controller**
  - Indicates a controller component in the presentation layer.



Auto Wiring of Bean

- **@Autowired :**
  - Used to auto wire bean on the **setter** method, **constructor** or a **field**.
  - It uses auto wire bean by matching data type.

- **@Autowired(required =false)**
  - **Exception** will be thrown when **matching bean is not found**
  - required attribute is used to ***disable this checking***
  - Will leave the property unset.

- **@Qualifier**
  - Used to control which bean should be autowired on a field.
  - If there are **two similar beans**, can **specify the bean name** to wire

Customer Bean

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@FieldDefaults(level = AccessLevel.PRIVATE)
public class Customer {

private int customerId;
private String customerName;
}
```

Product Bean

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@FieldDefaults(level = AccessLevel.PRIVATE)

public class Product {

private int productId;
private String productName;

}
```

Registering The Beans

```
@Bean
public Customer customer() {

return new Customer(101,"Ramesh");
}

@Bean
public Product tv() {

return new Product(101,"LED Tv");
}
```

Constructor DI

```
@Component
@Data
public class Invoice {
private Customer customer;
private Product product;

@Autowired
public Invoice(Customer customer, Product product) {
super();
this.customer = customer;
this.product = product;
}
}
```

Setter DI

```
@Autowired
public void setCustomer(Customer customer) {
this.customer = customer;
}

public Product getProduct() {
return product;
}

@Autowired
public void setProduct(Product product) {
this.product = product;
}

public Invoice() {
// TODO Auto-generated constructor stub
}
```

Spring Boot

- Extension of the Spring framework
  - **A faster and more efficient development**

- Eliminates the boilerplate configurations required for setting up a Spring application.

  - Opinionated ‘starter’ dependencies to simplify build and application configuration
  - Embedded server to avoid complexity in application deployment
  - Metrics, Health check, and externalized configuration
  - Automatic configuration whenever possible

Spring Initializr

- https://start.spring.io/
  - **A web-based UI tool** provided by the Pivotal
  - Used to generate the structure of the **Spring Boot Project**.
  - Can configure the list of dependencies
  - Package downloaded as **Jar** or **War** file
  - Import as a Maven Project into STS

Spring Boot Starters

- Set of convenient dependency descriptors
- Eliminates the need to hunt through sample code
- No need to copy-paste loads of dependency descriptors.
- Contains lot of the dependencies that are required to get a project up and running
- Also has support of managed transitive dependencies.

Creating a Spring Boot Project

- In the Eclipse IDE Right-click in the package explorer and select New -> Spring Starter Project

- A screen opens with some details which can be changed or can accept the default values for a simple project

- **The entry point of a Spring Boot application is the class which is annotated with *@SpringBootApplication*:**
- Uses this class with *public static void main* entry-point to launch an embedded web server.

Spring Boot Bootstrap

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

- Spring Boot can scan all the classes in the same package or sub packages of Main-class for components.

- Equivalent to using **@Configuration**, **@EnableAutoConfiguration**, and **@ComponentScan** with their default attributes,

Spring Application

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

ConfigurableApplicationContext ctx =
    SpringApplication.run(BootIocApplication.class, args);

Invoice invoice= ctx.getBean(Invoice.class);

System.out.println(invoice);

ctx.close();
}
```

AUTO CONFIGURATION

Auto Configuration

- Spring Boot takes an **opinionated view of the Spring platform and third-party libraries**
  - Can get started with minimum effort
  - It tries to **read ".properties"** from various hard-coded locations.
  - It also reads the "spring.factories" file
    - Part of auto configure-module
    - Determines the Auto Configurations it should evaluate.

Auto Configuration

- Some Spring Boot Jars contain special JSON meta-data files that the editor looks for
  - These files contain information about the known configuration properties.
- **“spring-boot-autoconfigure-XXX.jar”**
  - META-INF/spring-configuration-metadata.json”.
  - Can find properties like server.port being documented there.

AutoConfigurations

- **Auto-registered @PropertySources**
- Spring Boot will *automatically* register these PropertySources
- It has a default set of property locations that it *always* tries to read
  - command line arguments
  - application.properties inside .jar file etc.

YAML

Yaml

- YAML is a superset of JSON
- A convenient format for specifying hierarchical configuration data.
- **SnakeYAML**
  - Jar file added to the class path
  - Spring Boot automatically supports YAML as an alternative to properties
  - *application.properties takes precedence over application.yml*
    - *If both of them are present*

YAML

- Spring provides two convenient classes that can be used to load YAML documents.

- **YamlPropertiesFactoryBean**
  - To load YAML as Properties

- **YamlMapFactoryBean**
  - To load YAML as a Map.

Yaml

- .yml file is advantageous over .properties file
  - Has type safety,
  - Hierarchy
  - supports list

- YAML supports lists as hierarchical properties or inline list

- my:
  servers:
  - dev
  - prod

- servers: [dev, prod]

ADVANCED BEAN CONFIGURATIONS

Disambiguation options

```
@Component
public class TourAgent {
    private int id;
    private String agentName;
    private long mobileNumber;
}
```

```
@Bean
public TourAgent tourAgent() {

return new TourAgent(1033,"Ram",7484848);

}
```

Disambiguation options

```
 Properties props = new Properties();

props.put(
    "spring.main.allow-bean-definition-overriding", "true");

ConfigurableApplicationContext ctx =

new SpringApplicationBuilder(TourServiceApplication.class)
            .properties(props)
             .build()
             .run(args);
```

@Primary

- Used to give higher preference to a bean, when there are multiple beans of same type.
- Used on any class directly or indirectly **annotated** with @Component or on methods **annotated** with @Bean
- If a bean has @Autowired *without* any @Qualifier
  - Multiple beans of the type exist
  - Candidate bean marked @Primary will be chosen
- @Qualifier should be used in conjunction with @Autowired always.
- @Primary should be used in conjunction with @Bean

@Primary

```
@Bean
 @Primary
 public Service myService() {
        return new Service();
}
        @Bean
        public Service backupService() {
            return new Service();
         }
```

getBean(Service.class) will return the primary bean

Lazy Initialization

- Beans are created as they are needed rather than during application startup.
  - Enabling lazy initialization **May improve the startup time** of the application
  - In a web application, enabling lazy initialization will result in many web-related beans not being initialized until an HTTP request is received.

Lazy Initialization

- It can delay the discovery of a problem with the application.
- If a misconfigured bean is initialized lazily, a failure will no longer occur during startup and the problem will only become apparent when the bean is initialized.
- May reduce the number of beans created when the application is starting

Lazy initialization

- Can be enabled programmatically using Spring Application Builder
- Can be enabled using property from spring boot 2.2
  - **spring.main.lazy-initialization**=true

  Can be enabled by using the @Lazy on the Factory Method

Lazy-initialized beans

```
@Bean()
@Lazy(value=true)
 public Employee myBean() {
     System.out.println("Loading LazyBean bean");
   return new Employee(employee_Id,employee_Name);

}
```

Lazy-initialized beans

- ApplicationContext implementations eagerly create and configure all singleton beans as part of the initialization process.

- A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

```
@Bean()
@Lazy(value=true)
 public Employee myBean() {
     System.out.println("Loading LazyBean bean");
   return new Employee(employee_Id,employee_Name);

}
```

Life Cycle Methods

```
@Component
public class Invoice {

@PostConstruct
public void myInit()
{
      System.out.println("Inside init Method");
}
@PreDestroy
public void myDestroy()
{
       System.out.println("Inside Destroy Method");
}
}
```

Life Cycle Java Based Configuration

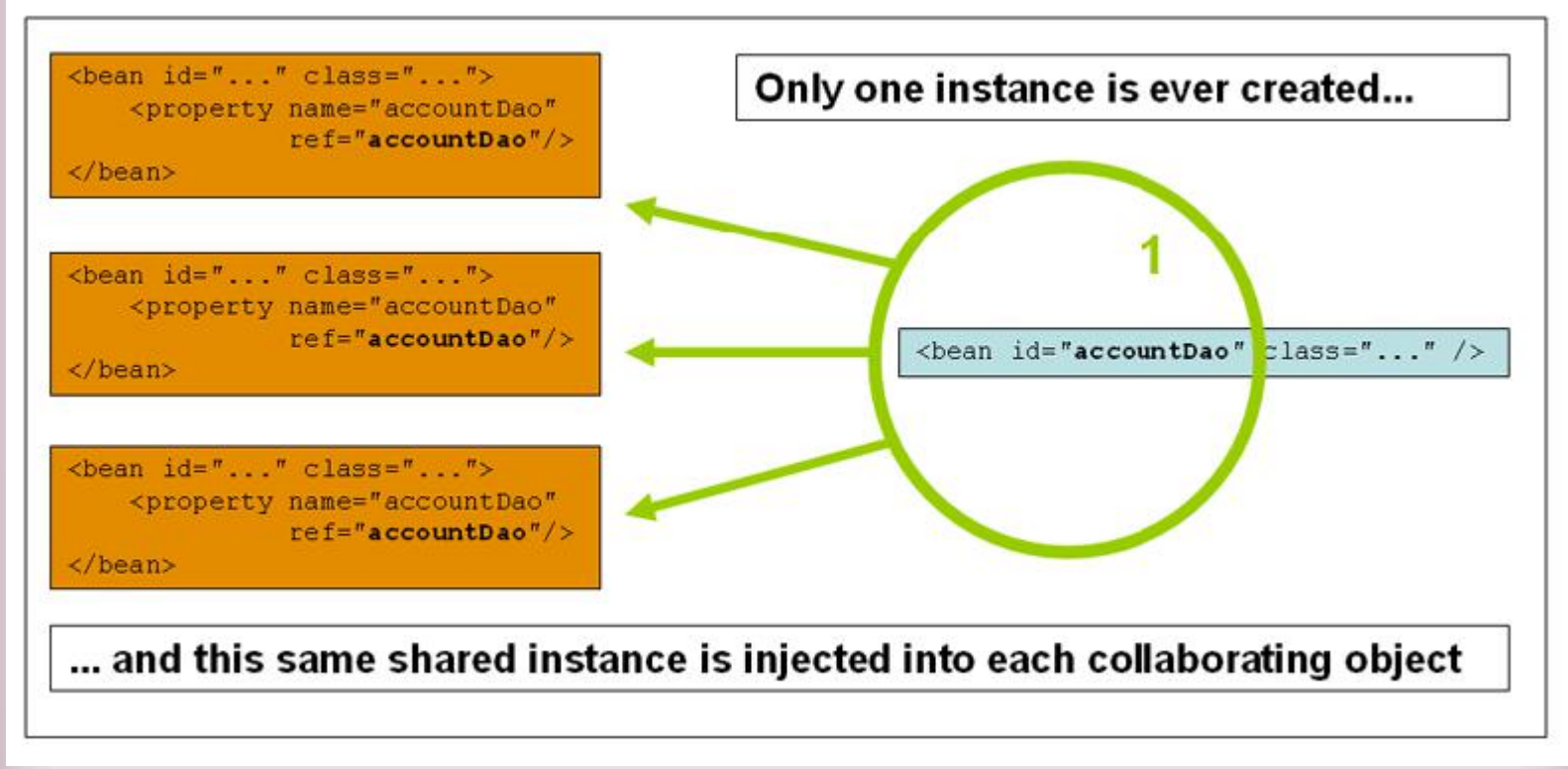
```
public class Employee {

public void start() {   // Initialization Work     }

public void close() {
      // Destruction Work
  }
}

@Bean(initMethod="start",destroyMethod="close")
public Employee myBean() {
      return new Employee(employee_Id,employee_Name);
  }
```

Life Cycle Methods

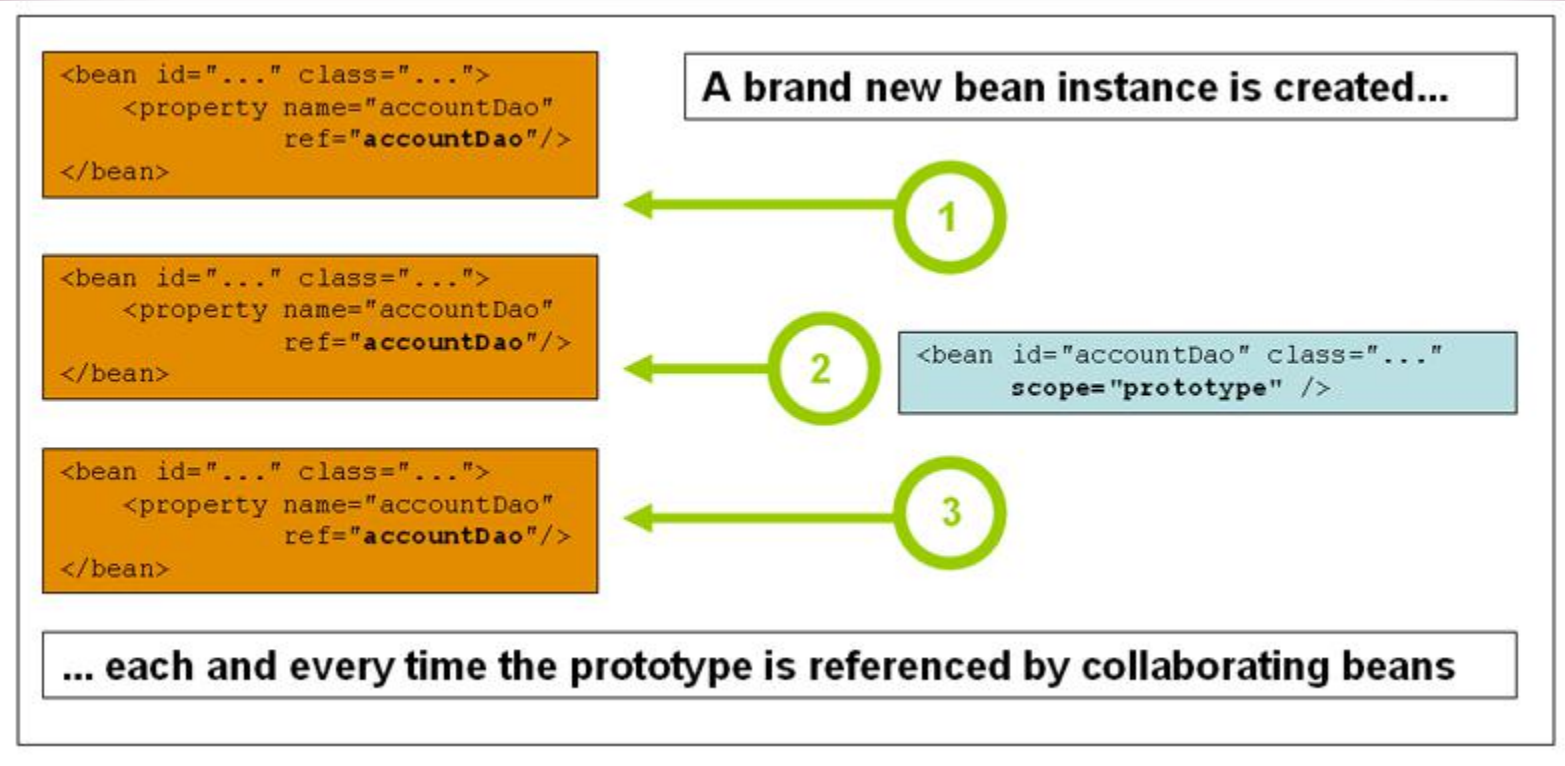
```
public class DeliveryExecutive {

@Autowired
private Environment env;

public DeliveryExecutive{

      env.getActiveProfiles()   -> Will throw Null Pointer Exception
}
public void init() {

log.info("Init Method called");
log.info(env.getActiveProfiles().toString());
System.out.println(Arrays.asList(env.getDefaultProfiles().toString()));

}
}
```

SPRING BEAN SCOPES

Singleton


<bean id="..." class="...">
<property name="accountDao"
ref="accountDao"/>
</bean>
<bean id="..." class="...">
<property name="accountDao"
ref="accountDao"/>
</bean>
<bean id="..." class="...">
<property name="accountDao"
ref="accountDao"/>
</bean>
Only one instance is ever created...
1
<bean id="accountDao" class="..." />
... and this same shared instance is injected into each collaborating object

Prototype Scope



Bean Scope

- The Bean Scope for Standard Java SE Beans are singleton:
- **@Scope(scopeName=ConfigurableBeanFactory.*SCOPE_SINGLETON)***
  - **Scopes a single bean definition to a single object instance per Spring IoC container, *per container and per bean.***
  - **Single instance will be stored in a cache and all subsequent requests and references will result in cached object being returned.**
- **@Scope(value=ConfigurableBeanFactory.SCOPE_PROTOTYPE)**

Bean Scope

- The Bean Score for Standard Java SE Beans are singleton:
- **Scopes a single bean definition to a single object instance per Spring IoC container, *per container and per bean.***
- ***prototype:***
  - Prototype results in the creation of a new bean instance every time a request for that specific bean is made
  - Can change the Scope by
- **@Scope(value=ConfigurableBeanFactory.*SCOPE_PROTOTYPE*)**

Test the Scopes

```
DiscountService service = ctx.getBean(DiscountService.class);

DiscountNotification protoBean;

protoBean = service.getDiscount("april");


log.info("Discount :="+protoBean.showDiscount());


protoBean = service.getDiscount("may");


log.info("Discount  :="+protoBean.showDiscount());
```

PROFILES

Profiles

- A way to segregate parts of application configuration and make it only available in certain environments.
- Used to control two things:
  - Influence the application properties
  - Which beans are loaded into the application context.
- **@Profile**
  - Used to Create profiles .
  - Can be attached to an @Configuration Class
    - Can also be attached to @Bean Factory Method.

Application.yml

```
spring:
   config:
    activate:
      on-profile:
      - dev
server:
  port: 6060

logging:
  level:
    '[org.springframework.boot]': trace

---          => (Three Hypens)
```

Application.yml

```
spring:
  config:
    activate:
      on-profile:
      - prod
server:
  port: 6065

logging:
  level:
    '[org.springframework.boot]': info
```

Creating Beans Based on Profiles

```
@Bean
@Profile(value = "dev")
public Customer ram() {

return new Customer(101,"Developer
Ramesh","ram@abc.com");
}

@Bean
@Profile(value = "prod")
public Customer shyam() {

return new Customer(102,"Admin Shyam","shy@abc.com");
}
```

SPRING BOOT RUNNERS

Command Line Runner

- A Functional Interface
- A special bean that execute some logic after the application context is loaded and started.
  - They are Created within the same application context
  - Can create Multiple CommandLineRunner beans
  - It can be ordered using the Ordered interface

    or @Order annotation.
  - Has a run() method that accepts array of String as an argument
-

Command Line Runner

```
@Bean
public CommandLineRunner commandLineRunner() {


return (args) ->  {
  for(String eachArg:args) {
  System.out.println(" Info"+eachArg);
  }
  } ;
}
```

Building Application

- **mvn package**
- Can Use the Generated jar to Execute the application.
- **java -jar target/mymodule-0.0.1-SNAPSHOT.jar**
- **java -jar -Dspring.profiles.active=prod hospital-service.jar**