Dependency Injection using Spring Framework

[redacted]

Agenda

- Dependency Injection with Spring Framework
- Using Qualifiers
- Primary Beans
- Spring Profiles
- Default Profile
- Spring Bean Life Cycle
- Open Closed Principle - Revisited
- Interface Segregation Principle - Revisited
- Dependency Injection Principle – Revisited
- Interface Naming Conventions

Spring Managed Stereotypes

@Controller
Indicates that an annotated class is a controller

@Service
Indicates that an annotated class is a service

@Autowired
used on properties, setters and constructors. Allows the developers the ability to skip manually configuring what properties to inject.

Spring Dependency Injection

- Lets annotate our controllers (@Controller), autowire (@autowire) our properties (PropertyInjectorController) and annotate GreetServiceImpl (@service)

GreetingServiceImpl.java

```java
package ca.gbc.comp3095.lecture3.services;

import org.springframework.stereotype.Service;

@Service
public class GreetingServiceImpl implements GreetingService {

    @Override
    public String sayGreeting() {
        return "Hello World";
    }

}
```

GreetingServiceImpl.java | PropertyInjectedController.java

```java
package ca.gbc.comp3095.lecture3.controllers;

import ca.gbc.comp3095.lecture3.services.GreetingService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

@Controller
public class PropertyInjectedController {

    @Autowired
    public GreetingService greetingService;

    public String getGreeting() { return greetingService.sayGreeting(); }


}
```

Spring Dependency Injection

- Update main to utilize the new PropertyInjectedController, spring managed component

GreetingServiceImpl.java | Lecture3Application.java | MyController.java | PropertyInjectedController.java | Lecture3ApplicationTests.java

```java
package ca.gbc.comp3095.lecture3;

import ...

@SpringBootApplication
public class Lecture3Application {

    public static void main(String[] args) {

        //obtain handle on Spring Application Context
        ApplicationContext ctx = SpringApplication.run(Lecture3Application.class, args);

        //add context for instance of context
        //retrieved a bean
        MyController myController = (MyController) ctx.getBean( name: "myController");

        //execute method in the controller
        String greeting = myController.sayHello();

        System.out.println(greeting);

        System.out.println("----- Property");

        PropertyInjectedController propertyInjectedController = (PropertyInjectedController)  ctx.getBean( name: "propertyInjectedController");

        System.out.println(propertyInjectedController.getGreeting());


    }

}
```

Spring Dependency Injection…

Annotate the remaining Controllers …

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

    //obtain handle on Spring Application Context
    ApplicationContext ctx = SpringApplication.run(Lecture3Application.class, args);

    //add context for instance of context
    //retrieved a bean
    MyController myController = (MyController) ctx.getBean( name: "myController");

    //execute method in the controller
    String greeting = myController.sayHello();

    System.out.println(greeting);

    System.out.println("----- Property");
    PropertyInjectedController propertyInjectedController = (PropertyInjectedController)  ctx.getBean( name: "propertyInjectedController");
    System.out.println(propertyInjectedController.getGreeting());

    System.out.println("----- Setter");
    SetterInjectedController setterInjectedController = (SetterInjectedController)  ctx.getBean( name: "setterInjectedController");
    System.out.println(setterInjectedController.getGreeting());

    System.out.println("----- Constructor");
    ConstructorInjectedController constructorInjectedController = (ConstructorInjectedController)  ctx.getBean( name: "constructorInjectedController");
    System.out.println(setterInjectedController.getGreeting());

}
```

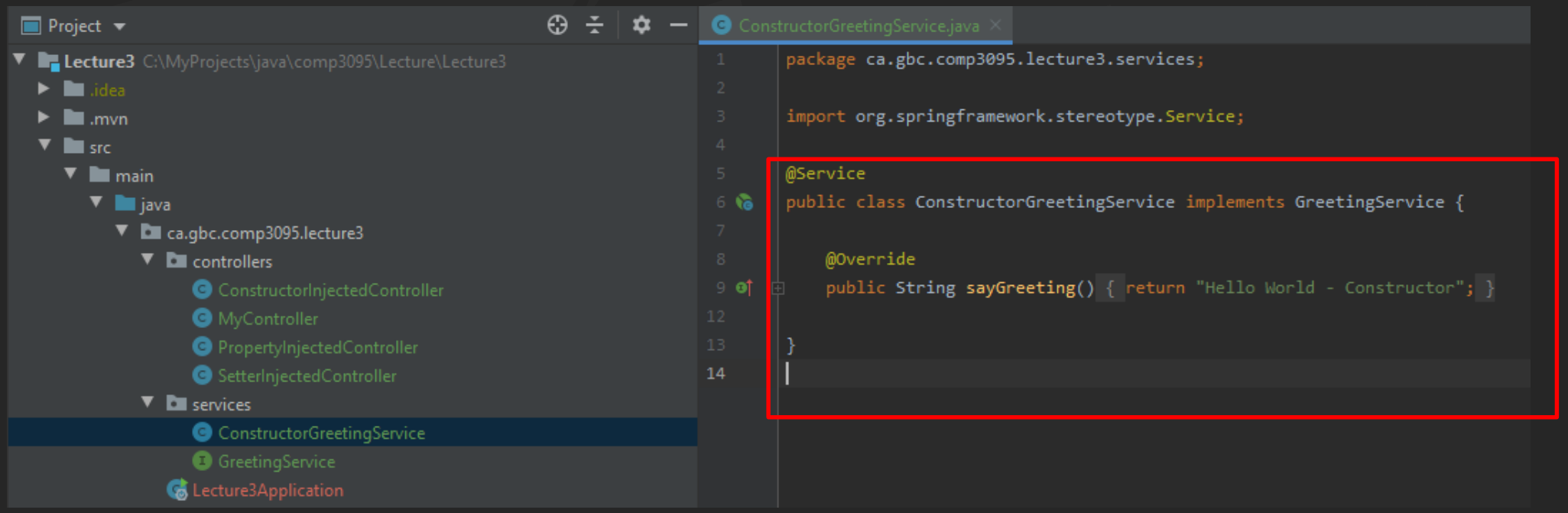
```
Hello World
Hi Folks
----- Property
Hello World
----- Setter
Hello World
----- Constructor
Hello World

Process finished with exit code 0
```

Using Qualifiers

[redacted]

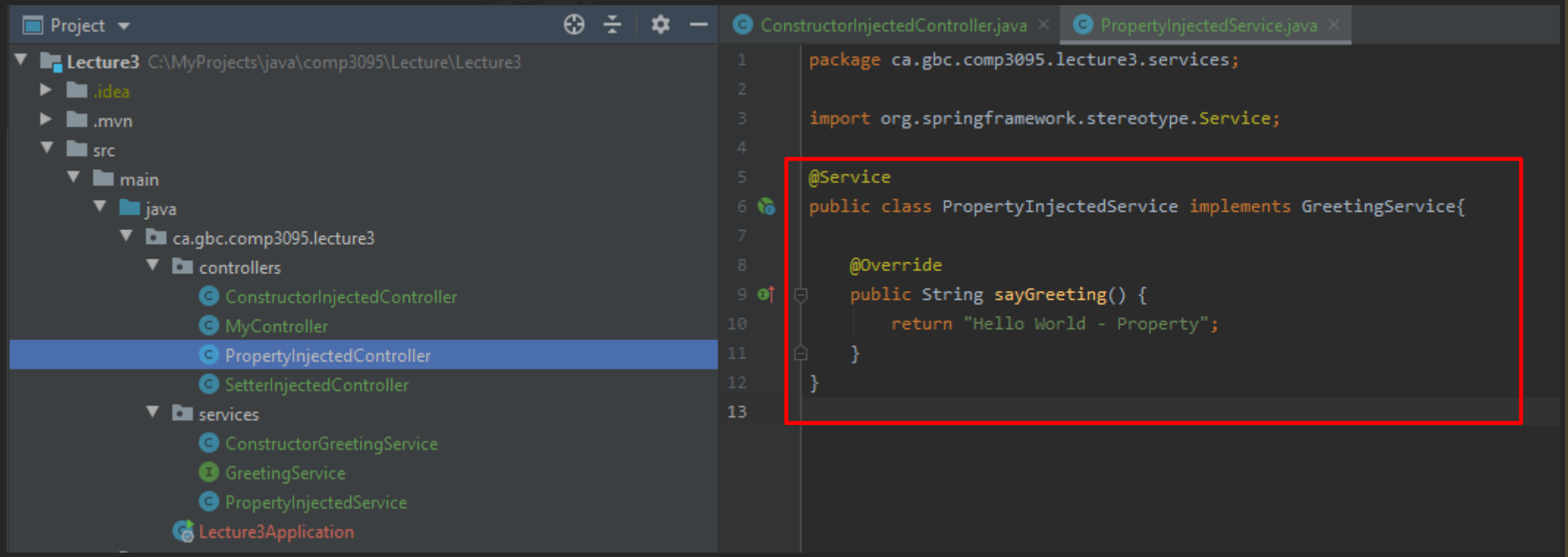
Modifying The Services

- Lets modify (refactor) the GreetingService to be ConstructorGreetingService
  - Modify the service sayGreeting() outout to “Hello World - Constructor”



Modifying The Services…

- Lets also create a new service for Property and Setter services built in the same fashion



Qualifier Needed

```
@Controller
public class PropertyInjectedController {

    @Autowired
    public GreetingService greetingService;

    public String getGreeting()

}
```

Could not autowire. There is more than one bean of 'GreetingService' type.
Beans: constructorGreetingService (ConstructorGreetingService.java)
propertyInjectedService (PropertyInjectedService.java)

Add qualifier Alt+Shift+Enter More actions... Alt+Enter Alt+Shift+J Ctrl+Shift+M

```
Parameter 0 of constructor in ca.gbc.comp3095.lecture3.controllers.ConstructorInjectedController required a single bean, but 2 were found:
    - constructorGreetingService: defined in file [C:\MyProjects\java\comp3095\Lecture\Lecture3\target\classes\ca\gbc\comp3095\lecture3\services\ConstructorGreetingService.class]
    - propertyInjectedService: defined in file [C:\MyProjects\java\comp3095\Lecture\Lecture3\target\classes\ca\gbc\comp3095\lecture3\services\PropertyInjectedService.class]


Action:

Consider marking one of the beans as @Primary, updating the consumer to accept multiple beans, or using @Qualifier to identify the bean that should be consumed


Process finished with exit code 1
```

Using @Qualifier

@Qualifier
Is used to resolve auto-wiring conflicts, when there are multiple beans of the same type. The qualifier name (parameter) is the camel cased name of the bean.

```java
package ca.gbc.comp3095.lecture3.controllers;

import ca.gbc.comp3095.lecture3.services.GreetingService;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Controller;

@Controller
public class ConstructorInjectedController {

    private final GreetingService greetingService;

    //not necessary to autowire the constructor in Spring4+
    public ConstructorInjectedController(@Qualifier("constructorGreetingService") GreetingService greetingService) {
        this.greetingService = greetingService;
    }

    public String getGreeting() { return greetingService.sayGreeting(); }


}
```

```java
package ca.gbc.comp3095.lecture3.controllers;

import ca.gbc.comp3095.lecture3.services.GreetingService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Controller;

@Controller
public class PropertyInjectedController {

    @Qualifier("propertyInjectedService")
    @Autowired
    public GreetingService greetingService;

    public String getGreeting() { return greetingService.sayGreeting(); }


}
```

Using @Qualifier.. Execution Succeeds

```
2020-08-19 16:52:07.823  INFO 18580 --- [
Hello World
Hi Folks
----- Property
Hello World - Property
----- Setter
Say Hello - Setter
----- Constructor
Hello World - Constructor

Process finished with exit code 0
```

```java
package ca.gbc.comp3095.lecture3.controllers;

import ca.gbc.comp3095.lecture3.services.GreetingService;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Controller;

@Controller
public class ConstructorInjectedController {

    private final GreetingService greetingService;

    //not necessary to autowire the constructor in Spring4+
    public ConstructorInjectedController(@Qualifier("constructorGreetingService") GreetingService greetingService) {
        this.greetingService = greetingService;
    }

    public String getGreeting() { return greetingService.sayGreeting(); }


}
```

```java
package ca.gbc.comp3095.lecture3.controllers;

import ca.gbc.comp3095.lecture3.services.GreetingService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Controller;

@Controller
public class PropertyInjectedController {

    @Qualifier("propertyInjectedService")
    @Autowired
    public GreetingService greetingService;

    public String getGreeting() { return greetingService.sayGreeting(); }


}
```

Primary Beans

[redacted]

Primary Beans

@Primary

Provides a higher preference to a bean when there are multiple beans of the same type.
Create new PrimaryGreetingSerivce to utilize @Primary

```java
package ca.gbc.comp3095.lecture3.services;

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

//Qualifier takes precedence, BUT when there is no @Qualifier, the @Primary service as the default
@Primary
@Service
public class PrimaryGreetingService implements  GreetingService{

    @Override
    public String sayGreeting() {
        return "Hello World - From the PRIMARY BEAN";
    }

}
```

Primary Beans continued…

```
2020-08-19 17:27:35.690  INFO 18552 --- [
---- Primary Bean
Hello World - From the PRIMARY BEAN
----- Property
Hello World - Property
----- Setter
Say Hello - Setter
----- Constructor
Hello World - Constructor

Process finished with exit code 0
```

Next we modify MyController and Main to utilize these changes to PrimaryGreetingService.

```java
package ca.gbc.comp3095.lecture3.controllers;

import ca.gbc.comp3095.lecture3.services.GreetingService;
import org.springframework.stereotype.Controller;

@Controller
public class MyController {

    private final GreetingService greetingService;

    public MyController(GreetingService greetingService) {
        this.greetingService = greetingService;
    }

    public String sayHello() {
        System.out.println("Hello World");
        return "Hi Folks";
    }

}
```

```java
@Controller
public class MyController {

    private final GreetingService greetingService;

    public MyController(GreetingService greetingService) {
        this.greetingService = greetingService;
    }

    public String sayHello() {
        return greetingService.sayGreeting();
    }

}
```

Spring Profiles

[redacted]

Spring Profiles

- One of the most powerful features of the Spring Framework
- Spring Profiles allow you to have beans in your configuration that will take on different characteristics
  - Ex image running a profile for H2 database, then switching profiles to run against MySQL profile etc…
- One of the most commonly used features in Dependency Injection

Example: Internalization Controller

I18N → Common abbreviation for internationalization

```java
package ca.gbc.comp3095.lecture3.controllers;

import ca.gbc.comp3095.lecture3.services.GreetingService;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Controller;

@Controller
public class I18nController {

    private final GreetingService greetingService;

    public I18nController(@Qualifier("i18nService") GreetingService greetingService) {
        this.greetingService = greetingService;
    }

    public String sayHello(){
        return greetingService.sayGreeting();
    }


}
```

Example: Two New (Competing Services)

Two new Service with competing “Profiles”

```java
package ca.gbc.comp3095.lecture3.services;

import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Service;

//provide qualifier name
@Profile("EN")
@Service("i18nService")
public class I18nEnglishGreetingService implements GreetingService {

    @Override
    public String sayGreeting() {
        return "Hello World - EN";
    }

}
```

```java
package ca.gbc.comp3095.lecture3.services;

import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Service;

//Provide qualifier name
@Profile("FR")
@Service("i18nService")
public class I18nFrenchGreetingService implements GreetingService{

    @Override
    public String sayGreeting() { return "Bonjour le monde - FR"; }
}
```

Example: Adjust Main Driver

Lets wire the main to utilize the new controller and an active profile

```java
@SpringBootApplication
public class Lecture3Application {

    public static void main(String[] args) {

        //obtain handle on Spring Application Context
        ApplicationContext ctx = SpringApplication.run(Lecture3Application.class, args);

        I18nController i18nController = (I18nController) ctx.getBean( name: "i18nController");
        System.out.println(i18nController.sayHello());

        //add context for instance of context
        //retrieved a bean
        MyController myController = (MyController) ctx.getBean( name: "myController");
        System.out.println("---- Primary Bean");
        System.out.println(myController.sayHello());

        System.out.println("----- Property");
        PropertyInjectedController propertyInjectedController = (PropertyInjectedController)  ctx.getBean( name: "propertyInjectedController");
        System.out.println(propertyInjectedController.getGreeting());

        System.out.println("----- Setter");
        SetterInjectedController setterInjectedController = (SetterInjectedController)  ctx.getBean( name: "setterInjectedController");
        System.out.println(setterInjectedController.getGreeting());

        System.out.println("----- Constructor");
        ConstructorInjectedController constructorInjectedController = (ConstructorInjectedController)  ctx.getBean( name: "constructorInjectedController");
        System.out.println(constructorInjectedController.getGreeting());

    }

}
```

application.properties

```properties
spring.profiles.active=EN
```

```
2020-08-20 07:15:09.939  INFO 2740 ---
Hello World - EN
---- Primary Bean
Hello World - From the PRIMARY BEAN
----- Property
Hello World - Property
----- Setter
Say Hello - Setter
----- Constructor
Hello World - Constructor
```

Default Profile

[redacted]

Default Profiles

- Default profiles are that are set, if not active profile has been configured
- In this way we can a bean belongs to a default configuration profile
- Can use Springs Default profile, so as to set the configuration up so it is not always necessary to specify an active profile.
- Many profiles can be configured (in application.properties) using a comma delimited string for those profiles qualifiers (spring.profiles.active = cat, EN …)
- Profile names are case sensitive

Example Default Profiles

- @Profile can be configured for multiple qualifiers

```java
package ca.gbc.comp3095.lecture3.services;

import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Service;

//provide qualifier name
@Profile({"EN", "default"})
@Service("i18nService")
public class I18nEnglishGreetingService implements GreetingService {

    @Override
    public String sayGreeting() { return "Hello World - EN"; }

}
```

application.properties

```properties
#spring.profiles.active=EN
```

```
2020-08-20 12:03:53.755  INFO 14344 --- [           main] c.g.c.l
Hello World - EN
---- Primary Bean
Hello World - From the PRIMARY BEAN
----- Property
Hello World - Property
----- Setter
Say Hello - Setter
----- Constructor
Hello World - Constructor
```

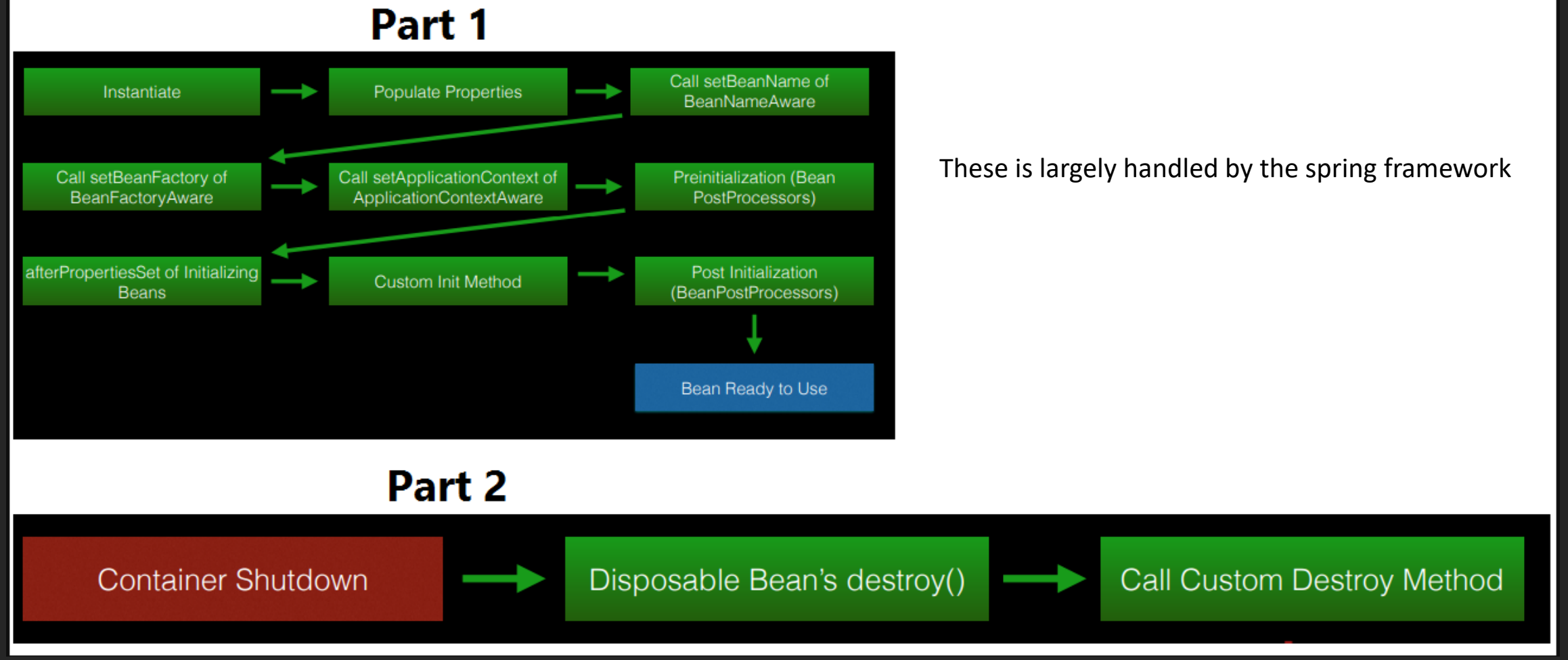
Spring Bean Lifecycle

[redacted]

Spring Bean Lifecycle

- The Spring IoC container manages Spring beans
- Recall a Spring Bean is just a managed instantiation of a java Class
- The Spring IoC responsible for:
  - instantiating
  - initializing
  - wiring beans
- Two phases of Spring Lifecycle
  - Phase 1 → Stages a bean goes through after instantiation
  - Phase 2 → Stages a bean goes through once the IoC container is shut down.

Spring Bean Lifecycle



These is largely handled by the spring framework

Callback Interfaces

Spring bean factory controls the creation and destruction of beans. To execute some custom code, it provides the callback methods which can be broadly categorized into two groups

- Post-initialization
- Pre-destruction

Spring has two interfaces you can implements for call back events

1. **InitializingBean.afterPropertiesSet()**
   - called after properties are set
2. **DisposableBean.destroy()**
   - called during bean destruction in shutdown

You can implement either/both or none of these interfaces

Lifecycle Annotations

- Spring has two annotations you can use to hook into the bean life cycle.

@PostConstruct

- annotated methods will be called after the bean has been constructed, but before its returned to the requesting object

@PreDestroy

- Is called just before the bean is destroyed by the container

Bean Post Processors

- Gives you a means to tap into the Spring context life cycle and interact with beans as they are processed
- Implement interface BeanPostProcessor
  - postProcessBeforeInitialization – called before bean initialization method
  - postProcessAfterInitialization – called after bean initialization

Spring “Aware” Interfaces

- Spring has over 14 Aware interfaces
- These are used to access the Spring Framework infrastructure
- These are largely used within the framework (ie. within the internal code)
- Rarely used by Spring developers

Spring “Aware” Interfaces

‘Aware’ Interfaces

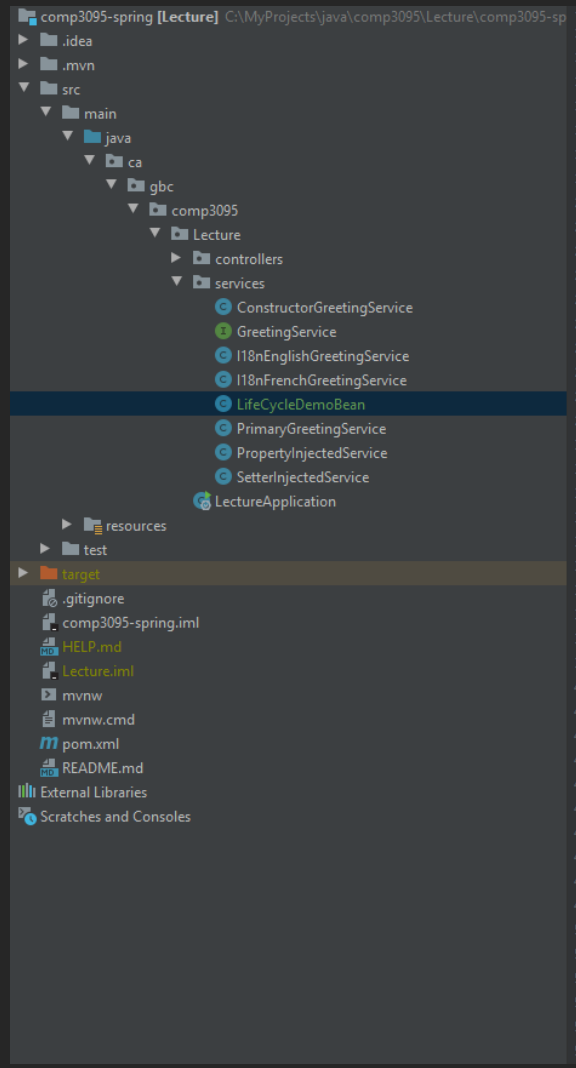
Aware Interface	Description
ApplicationContextAware	Interface to be implemented by any object that wishes to be notified of the ApplicationContext that it runs in.
ApplicationEventPublisherAware	Set the ApplicationEventPublisherthat this object runs in.
BeanClassLoaderAware	Callback that supplies the bean class loader to a bean instance.
BeanFactoryAware	Callback that supplies the owning factory to a bean instance.
BeanNameAware	Set the name of the bean in the bean factory that created this bean.
BootstrapContextAware	Set the BootstrapContext that this object runs in.

Aware Interface	Description
LoadTimeWeaverAware	Set the LoadTimeWeaver of this object’s containing ApplicationContext.
MessageSourceAware	Set the MessageSource that this object runs in.
NotificationPublisherAware	Set the NotificationPublisher instance for the current managed resource instance.
PortletConfigAware	Set the PortletConfig this object runs in.
PortletContextAware	Set the PortletContext that this object runs in.
ResourceLoaderAware	Set the ResourceLoader that this object runs in.
ServletConfigAware	Set the ServletConfig that this object runs in.
ServletContextAware	Set the ServletContext that this object runs in.

Spring Bean Lifecycle Demo

[redacted]

LifeCycleDemoBean




```java
import org.springframework.core.io.Resource;
import org.springframework.stereotype.Component;

import java.io.IOException;
import java.lang.annotation.Annotation;
import java.util.Locale;
import java.util.Map;

//implement the variety of interfaces Spring gives developers to work with Lifecycle events
@Component
public class LifeCycleDemoBean implements InitializingBean, DisposableBean, BeanNameAware, BeanFactoryAware, ApplicationContextAware {


    public LifeCycleDemoBean() {
        System.out.println("## I'm in the LifeCycleBean Constructor");
    }

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        System.out.println("## Bean Factory has been set");
    }

    @Override
    public void setBeanName(String name) {
        System.out.println("# My Bean name is: " + name);
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        System.out.println("## Application context has been set");
    }

    @Override
    public void destroy() throws Exception {
        System.out.println("## The Lifecycle bean has been terminated");
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("## The LifeCycle bean has its properties set!");
    }


}
```

LifeCycleDemoBean…

```
## I'm in the LifeCycleBean Constructor
## My Bean name is: lifeCycleDemoBean
## Bean Factory has been set
## Application context has been set
## The LifeCycle bean has its properties set!
2020-08-20 16:07:09.726  INFO 7716 --- [
Hello World - EN
---- Primary Bean
Hello World - From the PRIMARY BEAN
----- Property
Hello World - Property
----- Setter
Say Hello - Setter
----- Constructor
Hello World - Constructor
## The Lifecycle bean has been terminated

Process finished with exit code 0
```

Open Closed Principle
(*revisited*)

[redacted]

Open Closed Principle

- As application evolve, changes are required
- Adding new functionality to existing code, carries the risk of breaking the existing applications functionality
- The Open – Closed Principle:
  - Represents the “O” in the five SOLID principles
  - States that, **software entities (classes, modules functions etc..) should be open for extension, but closed for modification.**
  - **“Open for extension”** → means the behavior of a software module (class), can be extended to behave in different ways. Note by “extended” we are not limiting this principle to inheritance only. What is meant, is that a module should provide extension points to be able to alter its behavior if later desired.
  - “Closed for modification” → This means that the source code is such a module remains unchanged

Open Closed Principle – Violation – BAD Example

```
public class HealthInsuranceSurveyor{
    public boolean isValidClaim(){
        System.out.println("HealthInsuranceSurveyor: Validating hea
        /*Logic to validate health insurance claims*/
        return true;
    }
}
```

```
public class ClaimApprovalManager {
    public void processHealthClaim (HealthInsuranceSurveyor surveyo
        if(surveyor.isValidClaim()){
            System.out.println("ClaimApprovalManager: Valid claim. 
        }
    }
}
```

Open Closed Principle – Violation – BAD Example

What happen to the code if we introduce a new claim (VehicleClaim)?

Modified ClaimApprovalManager

```
public class ClaimApprovalManager {
    public void processHealthClaim (HealthInsuranceSurveyor surveyo
        if(surveyor.isValidClaim()){
            System.out.println("ClaimApprovalManager: Valid claim. (
        }
    }
    public void processVehicleClaim (VehicleInsuranceSurveyor surve
        if(surveyor.isValidClaim()){
            System.out.println("ClaimApprovalManager: Valid claim. (
        }
    }
}
```

Open Closed Principle – Good Example

```java
public abstract class InsuranceSurveyor {
    public abstract boolean isValidClaim();
}
```

```java
public class HealthInsuranceSurveyor extends InsuranceSurveyor{
    public boolean isValidClaim(){
        System.out.println("HealthInsuranceSurveyor: Validating hea
        /*Logic to validate health insurance claims*/
        return true;
    }
}
```

```java
public class VehicleInsuranceSurveyor extends InsuranceSurveyor{
    public boolean isValidClaim(){
        System.out.println("VehicleInsuranceSurveyor: Validating veh
        /*Logic to validate vehicle insurance claims*/
        return true;
    }
}
```

```java
public class ClaimApprovalManager {
    public void processClaim(InsuranceSurveyor surveyor){
        if(surveyor.isValidClaim()){
            System.out.println("ClaimApprovalManager: Valid claim. (
        }
    }
}
```

Remains unchanged

Interface Segregation Principle (*revisited*)

[redacted]

Interface Segregation Principle - Revisted

- Interfaces are used extensively in enterprise applications to achieve abstraction, support multiple inheritance of type.
- Represents the “I” in the SOLID principle
- What this principles says is, that Interfaces should not be bloated with methods the implemented class does not need.
- The Interface segregation principle advocates partitioning thick interfaces, into smaller and highly cohesive interfaces (“role” interfaces).
- Each role interface declares methods for a specific behavior

Interface Segregation Principle – Violation – BAD Example

```java
public interface Toy {
    void setPrice(double price);
    void setColor(String color);
    void move();
    void fly();
}
```

```java
public class ToyHouse implements Toy {
    double price;
    String color;
    @Override
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public void move(){}
    @Override
    public void fly(){}
}
```

Interface Segregation Principle – Good Example

```java
public interface Toy {
    void setPrice(double price);
    void setColor(String color);
}
```

```java
public interface Movable {
    void move();
}
```

```java
public interface Flyable {
    void fly();
}
```

Interface Segregation Principle – Good Example…

```java
public class ToyHouse implements Toy {
    double price;
    String color;

    @Override
    public void setPrice(double price) {

        this.price = price;
    }
    @Override
    public void setColor(String color) {

        this.color=color;
    }
    @Override
    public String toString(){
        return "ToyHouse: Toy house- Price: "+price+" Color: "+colo
    }
}
```

```java
public class ToyCar implements Toy, Movable {
    double price;
    String color;

    @Override
    public void setPrice(double price) {

        this.price = price;
    }

    @Override
    public void setColor(String color) {
     this.color=color;
    }
    @Override
    public void move(){
        System.out.println("ToyCar: Start moving car.");
    }
    @Override
    public String toString(){
        return "ToyCar: Moveable Toy car- Price: "+price+" Color: "-
    }
}
```

```java
public class ToyPlane implements Toy, Movable, Flyable {
    double price;
    String color;

    @Override
    public void setPrice(double price) {
        this.price = price;
    }

    @Override
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public void move(){

        System.out.println("ToyPlane: Start moving plane.");
    }
    @Override
    public void fly(){

        System.out.println("ToyPlane: Start flying plane.");
    }
    @Override
    public String toString(){
        return "ToyPlane: Moveable and flyable toy plane- Price: "+
    }
}
```

Interface Segregation Principle – Good Example…

```
public class ToyBuilder {
    public static ToyHouse buildToyHouse(){
        ToyHouse toyHouse=new ToyHouse();
        toyHouse.setPrice(15.00);
        toyHouse.setColor("green");
        return toyHouse;
        }
    public static ToyCar buildToyCar(){
        ToyCar toyCar=new ToyCar();
        toyCar.setPrice(25.00);
        toyCar.setColor("red");
        toyCar.move();
        return toyCar;
    }
    public static ToyPlane buildToyPlane(){
        ToyPlane toyPlane=new ToyPlane();
        toyPlane.setPrice(125.00);
        toyPlane.setColor("white");
        toyPlane.move();
        toyPlane.fly();
        return toyPlane;
    }
}
```

Dependency Inversion Principle (*Revisited*)

[redacted]

Dependency Inversion Principle

- Avoid tightly coupled code
- Tightly coupled code in enterprise application development can lead to serious adverse consequences.
- When one class knows explicitly about the design of another, changes to one, can break the application altogether.
- Represents the “D” in SOLID
- *“High-level modules should not depend on low-level modules. Both should depend on abstractions.”*
- *“Abstractions should not depend on details. Details should depend on abstractions”*

Dependency Inversion Principle – Violation – BAD Example

```java
public class LightBulb {
    public void turnOn() {
        System.out.println("LightBulb: Bulb turned on...");
    }
    public void turnOff() {
        System.out.println("LightBulb: Bulb turned off...");
    }
}
```

```java
public class ElectricPowerSwitch {
    public LightBulb lightBulb;
    public boolean on;
    public ElectricPowerSwitch(LightBulb lightBulb) {
        this.lightBulb = lightBulb;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
    public void press(){
        boolean checkOn = isOn();
        if (checkOn) {
            lightBulb.turnOff();
            this.on = false;
        } else {
            lightBulb.turnOn();
            this.on = true;
        }

    }
}
```

What about other appliances?

Dependency Inversion Principle – Good Example

```java
public interface Switch {
    boolean isOn();
    void press();
}
```

```java
public class ElectricPowerSwitch implements Switch {
    public Switchable client;
    public boolean on;
    public ElectricPowerSwitch(Switchable client) {
        this.client = client;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
   public void press(){
       boolean checkOn = isOn();
       if (checkOn) {
           client.turnOff();
           this.on = false;
       } else {
             client.turnOn();
             this.on = true;
       }

   }
}
```

```java
public class LightBulb implements Switchable {
    @Override
    public void turnOn() {
        System.out.println("LightBulb: Bulb turned on...");
    }

    @Override
    public void turnOff() {
        System.out.println("LightBulb: Bulb turned off...");
    }
}
```

```java
public class Fan implements Switchable {
    @Override
    public void turnOn() {
        System.out.println("Fan: Fan turned on...");
    }

    @Override
    public void turnOff() {
        System.out.println("Fan: Fan turned off...");
    }
}
```

Interface Naming Conventions

(What are the best practices in naming?)

[redacted]

Interface Naming Conventions

- Interface should be a good object name
  - Example Java's List interface
    - Implementations ArrayList, LinkedList, CheckList, SingletonList etc…
  - Don't start with "I"
    - Example no "IList"

Implementation Naming

- When just one implementation – generally accepted to use

  <Interface Name> + Impl

- When more than one implementation, name should indicate difference of Implementation

Questions?

[redacted]