CS 525 - ASD

Advanced Software Development

MS.CS Program

Department of Computer Science

Rene de Jong, MsC.

Maharishi University
OF MANAGEMENT

CS 525 - ASD

Advanced Software Development

Maharishi University
OF MANAGEMENT

Lesson 12 Spring framework

L1: ASD Introduction
L2: Strategy, Template method
L3: Observer pattern
L4: Composite pattern, iterator pattern
L5: Command pattern
L6: State pattern
L7: Chain Of Responsibility pattern

Midterm

L8: Proxy, Adapter, Mediator
L9: Factory, Builder, Decorator, Singleton
L10: Framework design
L11: Framework implementation
L12: Framework example: Spring framework
L13: Framework example: Spring framework

Final

Aim of the Spring framework

- Make enterprise Java application development as easy as possible following good programming practices

- Has support for
  - Data access
  - Remoting
  - Scheduling
  - …

A basic Spring application

```
package basic;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Application {
  public static void main(String[] args) {
    ApplicationContext context = new
                    ClassPathXmlApplicationContext("springconfig.xml");
    CustomerService customerService = context.getBean("customerService", CustomerService.class);
    customerService.sayHello();
  }
}
```

Create an ApplicationContext based on springconfig.xml

Get the bean with id="customerService" from the ApplicationContext

```
package basic;

public class CustomerService {
 public void sayHello(){
   System.out.println("Hello from CustomerService");
 }
}
```

springconfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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="customerService" class="basic.CustomerService" />
</beans>
```

Bean declaration

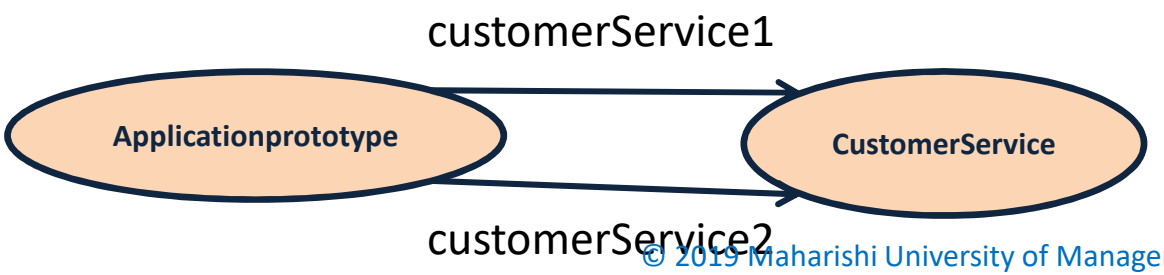
Spring beans are default singletons

```
public class Application{
  public static void main(String[] args) {
    ApplicationContext context = new
                      ClassPathXmlApplicationContext("module2/singleton/springconfig.xml");
    CustomerService customerService1 = context.getBean("customerService", CustomerService.class);
    CustomerService customerService2 = context.getBean("customerService", CustomerService.class);
    System.out.println("customerService1 ="+ customerService1);
    System.out.println("customerService2 ="+ customerService2);
  }
}
```

```
public class CustomerService {
  public CustomerService() {
  }
}
```

```
<bean id="customerService" class="module2.singleton.CustomerService" />
```

```
customerService1 =module2.singleton.CustomerService@29e357
customerService2 =module2.singleton.CustomerService@29e357
```



Prototype beans

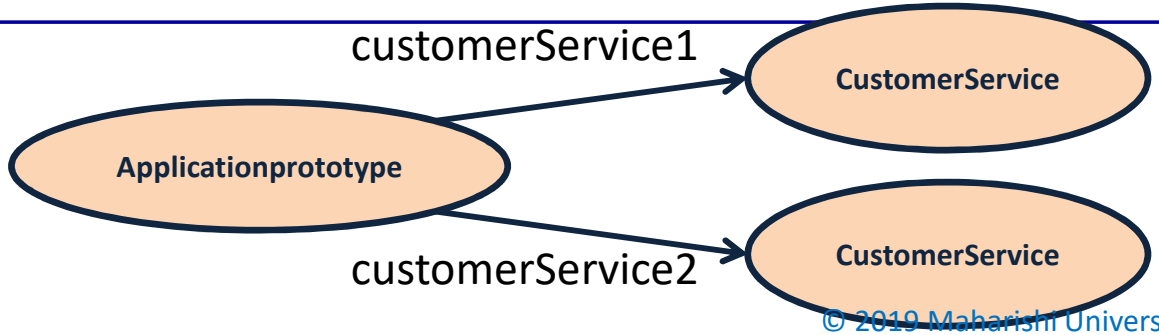
```
public class Application{
  public static void main(String[] args) {
    ApplicationContext context =
                    new ClassPathXmlApplicationContext("module2/prototype/springconfig.xml");
    CustomerService customerService1 = context.getBean("customerService", CustomerService.class);
    CustomerService customerService2 = context.getBean("customerService", CustomerService.class);
    System.out.println("customerService1 ="+ customerService1);
    System.out.println("customerService2 ="+ customerService2);
  }
}
```

```
public class CustomerService {
    public CustomerService() {
    }
}
```

```
<bean id="customerService" class="module2.prototype.CustomerService" scope="prototype" />
```

```
customerService1 =module2.prototype.CustomerService@1632847
customerService2 =module2.prototype.CustomerService@e95a56
```

prototype



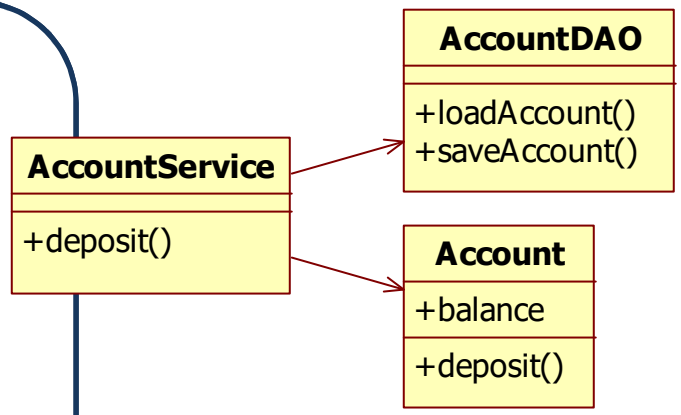
DEPENDENCY INJECTION

Different way's to "wire"2 object together

1. Instantiate an object directly
2. Use an interface
3. Use a factory class
4. Use Spring Dependency Injection

1. Instantiate an object directly

```
public class AccountService {
  private AccountDAO accountDAO;

  public AccountService() {
    accountDAO = new AccountDAO();
  }

  public void deposit(long accountNumber, double amount){
    Account account=accountDAO.loadAccount(accountNumber);
    account.deposit(amount);
    accountDAO.saveAccount(account);
  }
}
```

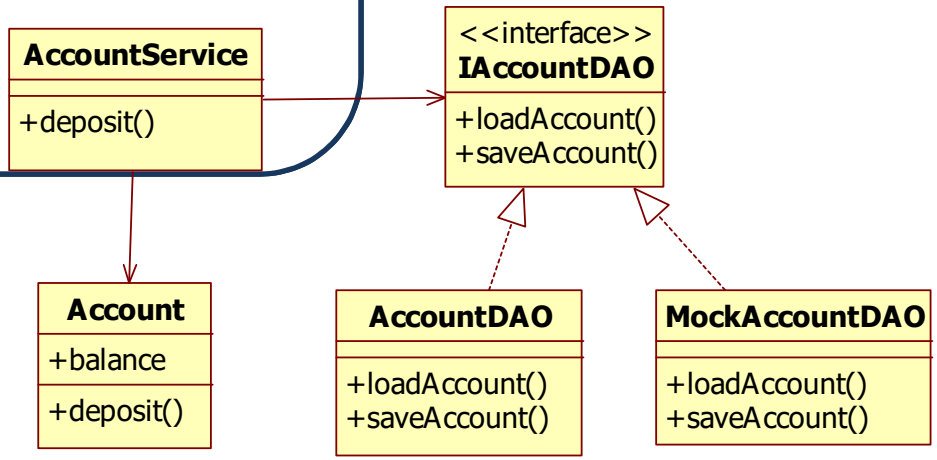


- The relation between AccountService and AccountDAO is hard coded
  - If you want to change the AccountDAO implementation, you have to change the code

2. Use an Interface

```
public class AccountService {
  private IAccountDAO accountDAO;

  public AccountService() {
    accountDAO = new AccountDAO();
  }

  public void deposit(long accountNumber, double amount){
    Account account=accountDAO.loadAccount(accountNumber);
    account.deposit(amount);
    accountDAO.saveAccount(account);
  }
}
```

accountDAO is of type IAccountDAO

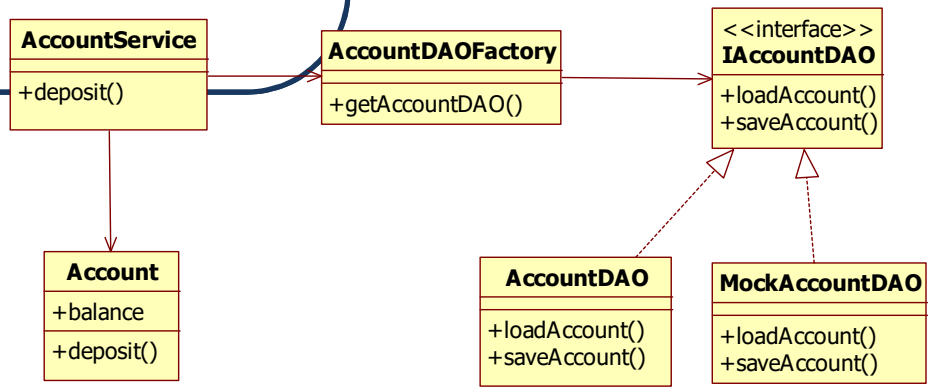


- The relation between AccountService and AccountDAO is still hard-coded
  - We have more flexibility, but if you want to change the AccountDAO implementation to the MockAccountDAO, you have to change the code

3. Use a factory class

```
public class AccountService {
  private IAccountDAO accountDAO;

  public AccountService() {
    accountDAO = AccountDAOFactory.getAccountDAO();
  }

  public void deposit(long accountNumber, double amount){
    Account account=accountDAO.loadAccount(accountNumber);
    account.deposit(amount);
    accountDAO.saveAccount(account);
  }
}
```

The factory creates
The accountDAO object



- The relation between AccountService and AccountDAO is still hard coded
  - We have more flexibility, but if you want to change the AccountDAO implementation to the MockAccountDAO, you have to change code in the factory

4. Use Spring Dependency Injection

```
public class AccountService {
  private IAccountDAO accountDAO;

  public void setAccountDAO(IAccountDAO accountDAO) {
    this.accountDAO = accountDAO;
  }

  public void deposit(long accountNumber, double amount){
    Account account=accountDAO.loadAccount(accountNumber);
    account.deposit(amount);
    accountDAO.saveAccount(account);
  }
}
```

accountDAO is injected by the Spring framework

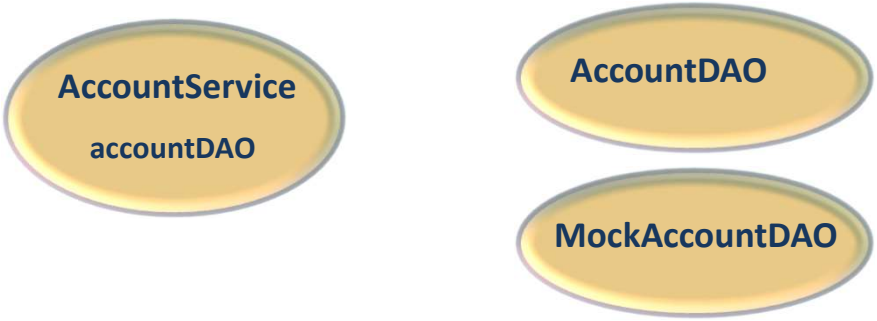
```
<bean id="accountService" class="AccountService">
  <property name="accountDAO" ref="accountDAO" />
</bean>
<bean id="accountDAO" class="AccountDAO" />
<bean id="mockAccountDAO" class="MockAccountDAO" />
```

- The attribute accountDAO is configured in XML and the Spring framework takes care that accountDAO references the AccountDAO object.

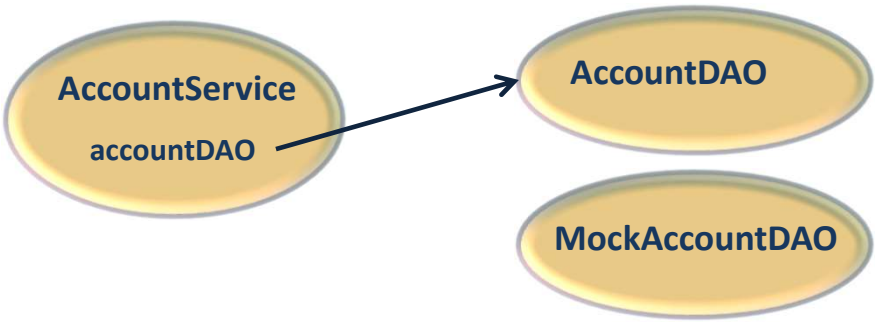
How does DI work?

```
<bean id="accountService" class="AccountService">
  <property name="accountDAO" ref="accountDAO" />
</bean>
<bean id="accountDAO" class="AccountDAO" />
<bean id="mockAccountDAO" class="MockAccountDAO" />
```

1. Spring instantiates all beans in the XML configuration file



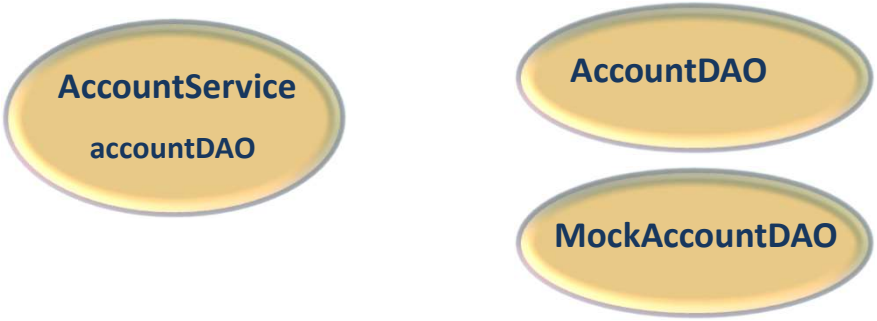
2. Spring then connects the accountDAO attribute to the AccountDAO instance



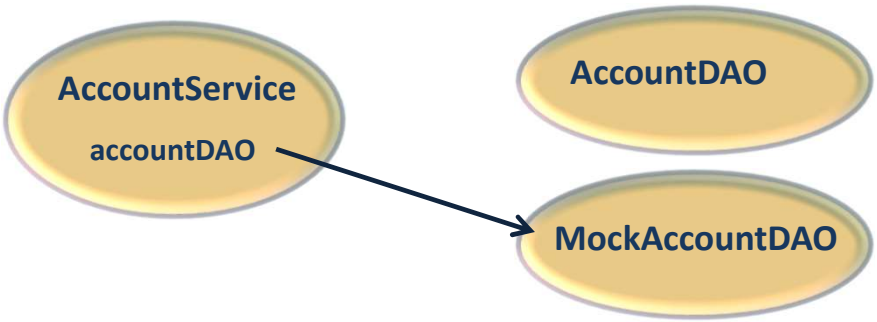
Change the wiring

```
<bean id="accountService" class="AccountService">
  <property name="accountDAO" ref="mockAccountDAO" />
</bean>
<bean id="accountDAO" class="AccountDAO" />
<bean id="mockAccountDAO" class="MockAccountDAO" />
```

1. Spring instantiates all beans in the XML configuration file



2. Spring then connects the accountDAO attribute to the MockAccountDAO instance



Advantages of Dependency Injection

```
public class AccountService {
  private IAccountDAO accountDAO;

  public void setAccountDAO(IAccountDAO accountDAO) {
    this.accountDAO = accountDAO;
  }
}
```

```
<bean id="accountService" class="AccountService">
  <property name="accountDAO" ref="accountDAO" />
</bean>
<bean id="accountDAO" class="AccountDAO" />
```

- Flexibility: it is easy to change the wiring between objects without changing code
- Unit testing becomes easier
- Code is clean

DIFFERENT TYPES OF DI

Types of DI

- Setter injection
- Constructor injection
- Autowiring

Setter Injection

```
public class AccountService {
  private IAccountDAO accountDAO;

  public void setAccountDAO(IAccountDAO accountDAO) {
    this.accountDAO = accountDAO;
  }
}
```

A setter method is needed

```
  <bean id="accountService" class="AccountService">
    <property name="accountDAO" ref="accountDAO" />
  </bean>
  <bean id="accountDAO" class="AccountDAO" />
```

Use <property …/>

Constructor Injection

```
public class AccountService {
  private IAccountDAO accountDAO;

  public AccountService(IAccountDAO accountDAO) {
    this.accountDAO = accountDAO;
  }
}
```

A constructor is needed to set accountDAO

```
  <bean id="accountService" class="AccountService">
    <constructor-arg  ref="accountDAO" />
  </bean>
  <bean id="accountDAO" class="AccountDAO" />
```

Use <constructor-arg …/>

Autowiring

- Spring figures out how to wire beans together
- 3 types of autowiring
  - By Name
  - By Type
  - Constructor

Autowiring by name

```
public class CustomerService {
  private EmailService emailService;

  public void addCustomer(){
    emailService.sendEmail();
  }

  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }
}
```

Autowire by name uses setter injection, so we need a setter method

```
public class EmailService {

  public void sendEmail(){
    System.out.println("sendEmail");
  }
}
```

Spring will inject the bean with id="emailService" into the attribute 'emailService'

```
<bean id="customerService" class="mypackage.CustomerService" autowire="byName"/>
<bean id="emailService" class="mypackage.EmailService"/>
```

Autowiring by type

```
public class CustomerService {
  private EmailService emailService;

  public void addCustomer(){
    emailService.sendEmail();
  }

  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }
}
```

Autowire by type uses setter injection, so we need a setter method

```
public class EmailService {

  public void sendEmail(){
    System.out.println("sendEmail");
  }
}
```

Spring will inject the bean with type EmailService" into the attribute 'emailService'

```
<bean id="customerService" class="mypackage.CustomerService" autowire="byType"/>
<bean id="eService" class="mypackage.EmailService"/>
```

Constructor autowiring

```
public class CustomerService {
  private EmailService emailService;

  public CustomerService(EmailService emailService) {
    this.emailService = emailService;
  }

  public void addCustomer(){
    emailService.sendEmail();
  }
}
```

The constructor has 1 attribute of type EmailService

```
public class EmailService {

  public void sendEmail(){
    System.out.println("sendEmail");
  }
}
```

Spring will inject the bean with type EmailService" into the attribute 'emailService'

```
<bean id="customerService" class="mypackage.CustomerService" autowire="constructor"/>
<bean id="eService" class="mypackage.EmailService"/>
```

Annotation based Autowiring by constructor

```
public class CustomerService {
  private EmailService emailService;

  @Autowired
  public CustomerService(EmailService emailService) {
    this.emailService = emailService;
  }

  public void addCustomer(){
    emailService.sendEmail();
  }
}
```

@Autowire indicates to Spring that the emailService attribute should be injected by type via the constructor

```
public class EmailService {

  public void sendEmail(){
    System.out.println("sendEmail");
  }
}
```

This tag tells Spring to look for configuration annotations in the declared beans

```
<context:annotation-config/>
<bean id="customerService" class="mypackage.CustomerService"/>
<bean id="eService" class="mypackage.EmailService"/>
```

Annotation based Autowiring by type

```
public class CustomerService {
  private EmailService emailService;

  @Autowired
  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }

  public void addCustomer(){
    emailService.sendEmail();
  }
}
```

@Autowire indicates to Spring that the emailService attribute should be injected by type via the setter method

```
public class EmailService {

  public void sendEmail(){
    System.out.println("sendEmail");
  }
}
```

This tag tells Spring to look for configuration annotations in the declared beans

```
<context:annotation-config/>
<bean id="customerService" class="mypackage.CustomerService"/>
<bean id="eService" class="mypackage.EmailService"/>
```

Field injection

```
public class CustomerService {
  @Autowired
  @Qualifier("myEmailService")
  private EmailService emailService;

  public void addCustomer(){
    emailService.sendEmail();
  }
}
```

autowire by name

```
public class EmailService {

  public void sendEmail(){
    System.out.println("sendEmail");
  }
}
```

```
<context:annotation-config/>
<bean id="customerService" class="mypackage.CustomerService"/>
<bean id="myEmailService" class="mypackage.EmailService"/>
```

Field injection

```
public class CustomerService {
  @Autowired
  private EmailService emailService;

  public void addCustomer(){
    emailService.sendEmail();
  }
}
```

autowire by type

```
public class CustomerService {
  @Inject
  private EmailService emailService;

  public void addCustomer(){
    emailService.sendEmail();
  }
}
```

Injection of primitive values

```
public class CustomerServiceImpl implements CustomerService {
  private String defaultCountry;
  private long numberOfCustomers;

  public void setDefaultCountry(String defaultCountry) {
    this.defaultCountry = defaultCountry;
  }
  public String getDefaultCountry() {
    return defaultCountry;
  }
  public long getNumberOfCustomers() {
    return numberOfCustomers;
  }
  public void setNumberOfCustomers(long numberOfCustomers) {
    this.numberOfCustomers = numberOfCustomers;
  }
}
```

```
<bean id="customerService" class="mypackage.CustomerServiceImpl">
  <property name="defaultCountry" value="USA"/>
  <property name="numberOfCustomers" value="56982"/>
</bean>
```

Automatic conversion from String to long

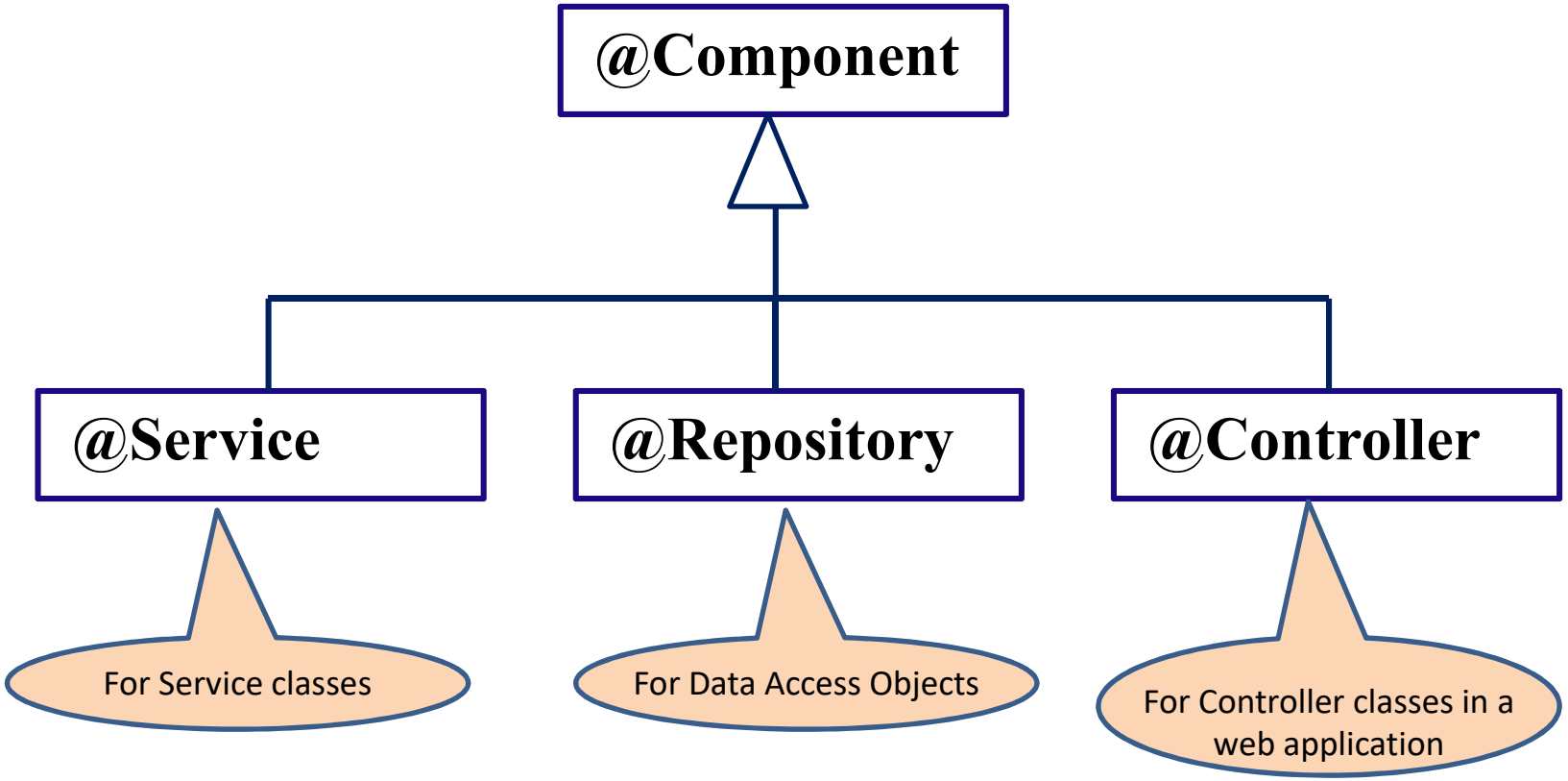
DEPENDENCY INJECTION WITH CLASSPATH SCANNING

Classpath scanning

- Define beans with annotations instead of defining them with XML
- All classes with the annotations
  - @Component
  - @Service
  - @Repository
  - @Controller

  become spring beans

Classpath scanning annotations



Classpath scanning example (1/2)

```
@Service ("customerService")
public class CustomerServiceImpl implements CustomerService{
private EmailService emailService;

  @Autowired
  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }

  public void addCustomer() {
    emailService.sendEmail();
  }
}
```

@Service annotation

The EmailService is injected

```
@Service ("emailService")
public class EmailService implements IEmailService {

  public void sendEmail() {
    System.out.println("sendEmail");
  }
}
```

@Service annotation

Classpath scanning example (2/2)

```
<?xml version="1.0" encoding="UTF-8"?>
<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-3.0.xsd">

    <context:component-scan base-package="module3.classpathscanning.basic"/>
    <context:annotation-config />
</beans>
```

No beans declared

```
public class Application {
  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("springconfig.xml");
    CustomerService customerService = context.getBean("customerService", CustomerService.class);
    customerService.addCustomer();
  }
}
```

@Value

```
@Service ("emailService")
public class EmailServiceImpl implements EmailService{
  @Value("smtp.mailserver.com")
  private String emailServer;

  public void sendEmail() {
    System.out.println("send email to server: "+ emailServer);
  }
}
```

Set the Value of an attribute

DEPENDENCY INJECTION WITH JAVA CONFIGURATION

Java Configuration

- Spring beans can also be configured in Java (and annotations) instead of XML

```java
@Configuration
public class AppConfig {
  @Bean
  public CustomerService customerService(){
    CustomerService customerService = new CustomerServiceImpl();
    customerService.setEmailService(emailService());
    return customerService;
  }
  @Bean
  public EmailService emailService(){
    return new EmailServiceImpl();
  }
}
```

Similar configuration

```xml
<bean id="customerService" class="module3.di.CustomerServiceImpl">
  <property name="emailService" ref="emailService"/>
</bean>
<bean id="emailService" class="module3.di.EmailServiceImpl" />
```

Java configuration example (1/2)

```
public class CustomerServiceImpl implements CustomerService{
private EmailService emailService;

  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }

  public void addCustomer() {
    emailService.sendEmail();
  }
}
```

```
public class EmailService implements IEmailService {

  public void sendEmail() {
    System.out.println("sendEmail");
  }
}
```

Java configuration example (2/2)

```
@Configuration
public class AppConfig {
  @Bean
  public CustomerService customerService(){
    CustomerService customerService = new CustomerServiceImpl();
    customerService.setEmailService(emailService());
    return customerService;
  }
  @Bean
  public EmailService emailService(){
    return new EmailServiceImpl();
  }
}
```

Create a bean with the name "customerService"

Set the property emailService

AnnotationConfigApplicationContext

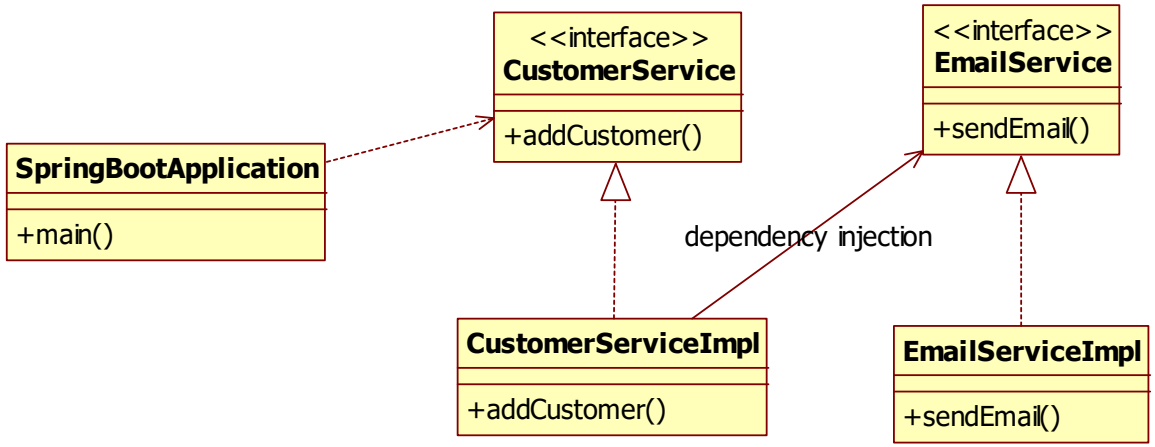
```
public class Application {
  public static void main(String[] args) {
    ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
    CustomerService customerService =
        context.getBean("customerService", CustomerService.class);

    customerService.addCustomer();
  }
}
```

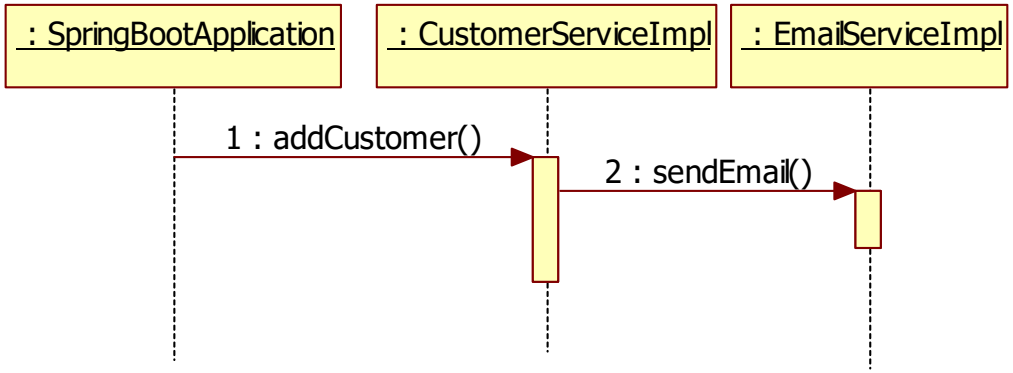
3 WAYS TO CONFIGURE SPRING APPLICATIONS

3 ways of Spring configuration

- XML configuration
- Classpath scanning and Autowiring
- Java configuration

Example application





The implementation

```
public interface EmailService {
  void sendEmail();
}
```

```
public class EmailServiceImpl implements EmailService{
  public void sendEmail() {
    System.out.println("Sending email");
  }
}
```

```
public interface CustomerService {
  void addCustomer();
}
```

```
public class CustomerServiceImpl implements CustomerService {

  private EmailService emailService;

  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }
  public void addCustomer() {
    emailService.sendEmail();
  }
}
```

Option 1: XML configuration

```
<bean id="customerService" class="xml.CustomerServiceImpl">
  <property name="emailService" ref="emailService" />
</bean>
<bean id="emailService" class="xml.EmailServiceImpl" />
```


```
public class CustomerServiceImpl implements CustomerService {

  private EmailService emailService;

  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }
  public void addCustomer() {
    emailService.sendEmail();
  }
}
```

XML configuration

- Advantages
  - Configuration separate from Java code
  - All configuration in one place
  - Tools can use the XML for graphical views
  - Easy to change the configuration

- Disadvantages
  - Large verbose XML file(s)
  - XML and Java has to fit together
  - No compile time type safety
  - Less refactor-friendly

Option 2: Classpath scanning and Autowiring


```
<context:component-scan base-package="scanning"/>
<context:annotation-config />
```

```
@Service
public class CustomerServiceImpl implements CustomerService {
  @Autowired
  private EmailService emailService;

  public void addCustomer() {
    emailService.sendEmail();
  }
}
```

Classpath scanning and Autowiring

- Advantages
  - All information (configuration and logic) in one place: the Java code
  - Simpler as XML
  - More type safe
- Disadvantage
  - Configuration in the Java code
  - Configuration is harder to change
    - Not a clear overview
    - You have to recompile

Option 3: Java configuration

```
@Configuration
public class AppConfig {
  @Bean
  public CustomerService customerService(){
    CustomerService customerService = new CustomerServiceImpl();
    customerService.setEmailService(emailService());
    return customerService;
  }
  @Bean
  public EmailService emailService(){
    return new EmailServiceImpl();
  }
}
```

```
public class CustomerServiceImpl implements CustomerService {

  private EmailService emailService;

  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }
  public void addCustomer() {
    emailService.sendEmail();
  }
}
```

Java configuration

- Advantages
  - Configuration separate from Java code
  - Type safe

- Disadvantage
  - Configuration class can contain lot of java configuration code
  - Configuration is harder to change
    - Not a clear overview
    - You have to recompile

Simpler configuration

- Java config + autowiring

```
@Configuration
public class AppConfig {
  @Bean
  public CustomerService customerService(){
    return new CustomerServiceImpl();
  }
  @Bean
  public EmailService emailService(){
    return new EmailServiceImpl();
  }
}
```

```
public class CustomerServiceImpl implements CustomerService {
  @Autowired
  private EmailService emailService;

  public void addCustomer() {
    emailService.sendEmail();
  }
}
```

Most simple configuration!

- Java config + classpath scanning + autowiring

```
@Configuration
@ComponentScan
public class AppConfig {
}
```

```
@Service
public class CustomerServiceImpl implements CustomerService {
  @Autowired
  private EmailService emailService;

  public void addCustomer() {
    emailService.sendEmail();
  }
}
```

```
@Service
public class EmailServiceImpl implements EmailService{
  public void sendEmail() {
    System.out.println("Sending email");
  }
}
```

Main point

- Spring can be configured in different ways but the most simple configuration is done with classpath scanning, autowiring and Java configuration
- Our actions yields maximum results with minimum effort if we operate at the level of pure consciousness.