



Dependency Injection

CS544: Enterprise Architecture



Dependency Injection

- Dependency injection is all about doing less (code), and accomplishing more (flexibility).
- In this module we will first look at the basics of dependency injection (what is it, why use it).
- Then we will look at the various ways you can perform dependency injection (setter injection, constructor injection, auto-wiring).
- After which we will look into other things we can inject (values, collections, inheritance)
- Lastly we will finish up with a section that we couldn't cover last module, alternate Spring configuration styles (Classpath Scanning, Java)



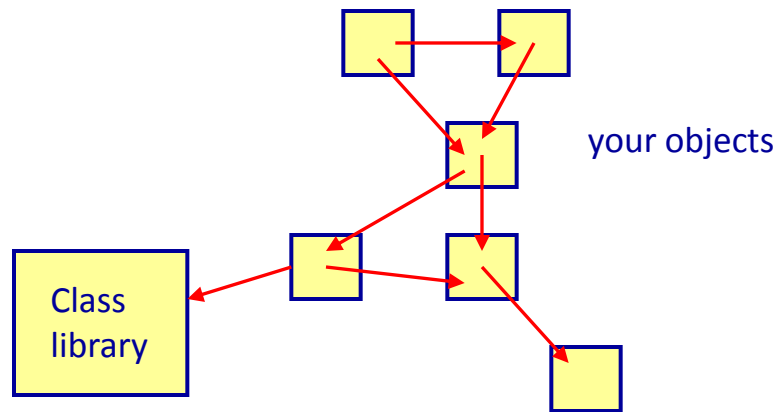
Dependency Injection:

BASICS OF DEPENDENCY INJECTION

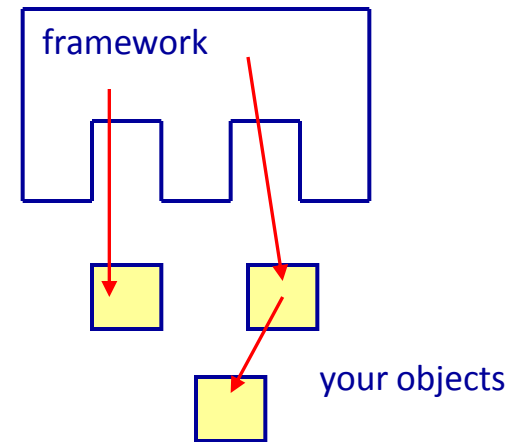


Inversion of Control (IoC)


- Hollywood principle: Don't call us, we'll call you
- The framework has control over your code



Your code calls the class library



IoC: The framework calls your code



Different way's to “wire”2 object together

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

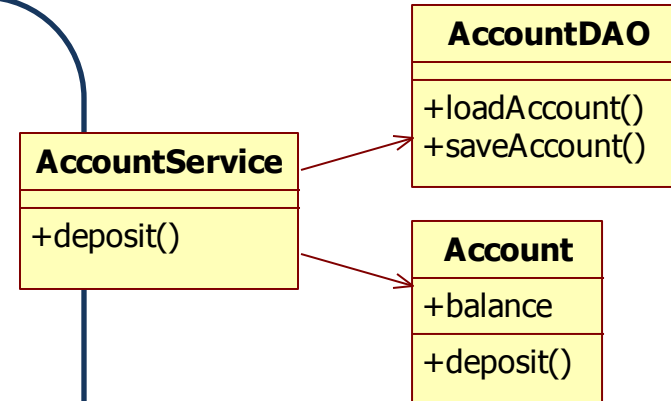
Text

wire means object A needs Object B



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);  
    }  
}
```



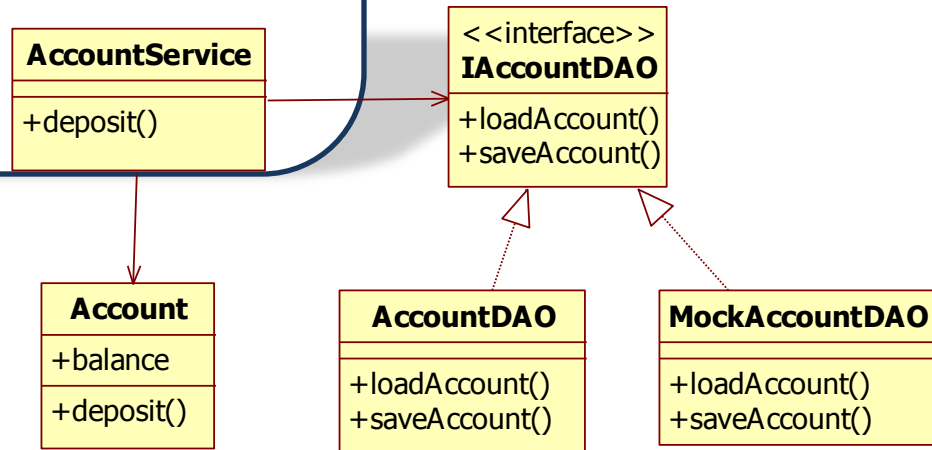
- The relation between AccountService and AccountDAO is hard coded
 - If you want to change the AccountDAO implementation, you have to change the code



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



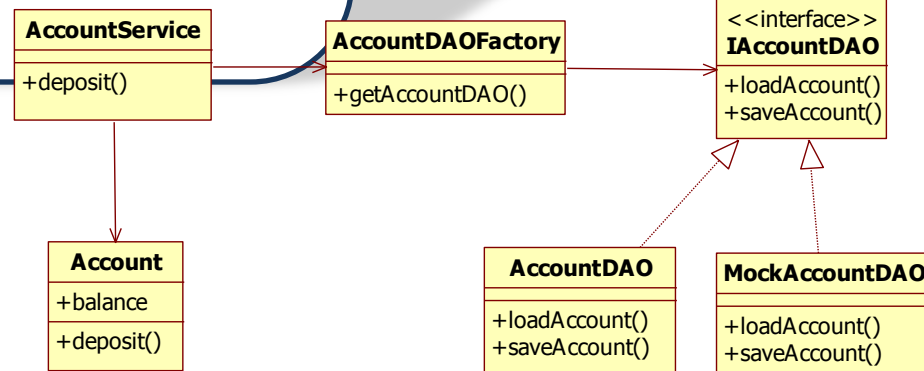
- The relation between AccountService and AccountDAO is still hard-coded
 - We have more flexibility, but if you want to change the AccountDAO implementation to the MockAccountDAO, you have to change the code



3. Use a factory object

```
public class AccountService {  
    private IAccountDAO accountDAO;  
  
    public AccountService() {  
        AccountDAOFactory daoFactory = new AccountDAOFactory();  
        accountDAO = daoFactory.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?

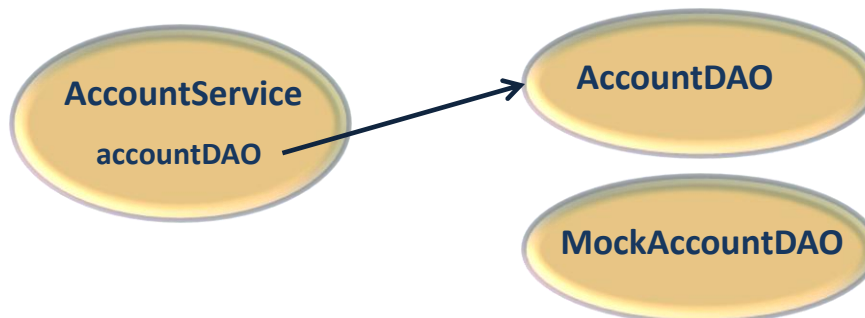
```
<bean id="accountService" class="AccountService">  
  <property name="accountDAO" ref="accountDAO" />  
</bean>  
<bean id="accountDAO" class="AccountDAO" />  
<bean id="mockAccountDAO" class="MockAccountDAO" />
```



1. Spring instantiates all beans in the XML configuration file



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





Change the wiring

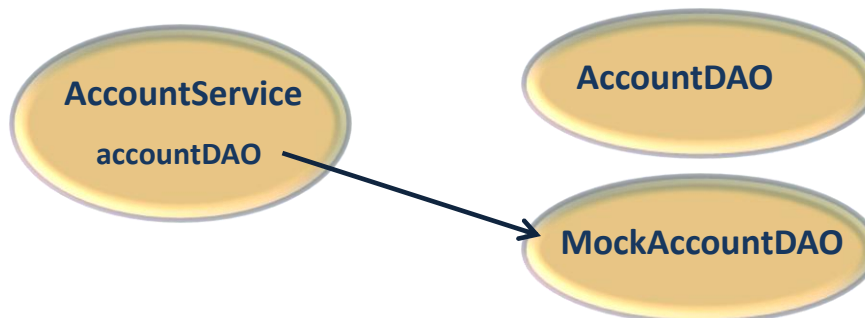
```
<bean id="accountService" class="AccountService">  
  <property name="accountDAO" ref="mockAccountDAO" />  
</bean>  
<bean id="accountDAO" class="AccountDAO" />  
<bean id="mockAccountDAO" class="MockAccountDAO" />
```



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



When do we use DI?

- When an object references another object whose implementation might change
 - You want to plug-in another implementation
- When an object references a plumbing object
 - An object that sends an email
 - A DAO object
- When an object references a resource
 - For example a database connection



Basics of Dependency Injection

- Dependency Injection is achieved by having a factory with an (XML) configuration file that wires objects up
- In other words, constructs objects, and relations (DI) based on the configuration.
- Life is found in layers – by separating out how objects relate our application becomes more flexible and robust.



Dependency Injection:

DIFFERENT TYPES OF DI



Types of DI

- Setter injection
- Constructor injection
- Autowiring
 - XML: by name, by type, by constructor
 - Annotations: by name, by type, by constructor



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 .../>



Constructor with multiple parameters

```
public class PaymentService implements IPaymentService{
    private IVisaVerifier visaVerifier;
    private IMastercardVerifier mastercardVerifier;

    public PaymentService(IVisaVerifier visaVerifier, IMastercardVerifier mastercardVerifier){
        this.visaVerifier=visaVerifier;
        this.mastercardVerifier=mastercardVerifier;
    }
}
```

Constructor has
2 arguments of a different
type

```
<bean id="paymentService" class="products.PaymentService">
    <constructor-arg ref="visaVerifier" />
    <constructor-arg ref="mastercardVerifier" />
</bean>
<bean id="visaVerifier" class="products.VisaVerifier"/>
<bean id="mastercardVerifier" class="products.MastercardVerifier"/>
```

Spring looks at the type
of the argument to decide
what to inject for
the first and the second
parameter



Constructor with multiple parameters of the same type

```
public class PaymentService implements IPaymentService{  
    private ICreditCardVerifier visaVerifier;  
    private ICreditCardVerifier mastercardVerifier;  
  
    public PaymentService(ICreditCardVerifier visaVerifier, ICreditCardVerifier mastercardVerifier){  
        this.visaVerifier=visaVerifier;  
        this.mastercardVerifier=mastercardVerifier;  
    }  
}
```

Constructor has
2 arguments of the same
type

```
<bean id="paymentService" class="products.PaymentService">  
    <constructor-arg ref="visaVerifier" />  
    <constructor-arg ref="mastercardVerifier" />  
</bean>  
<bean id="visaVerifier" class="products.VisaVerifier"/>  
<bean id="mastercardVerifier" class="products.MastercardVerifier"/>
```

Spring looks at the order
of declaration to decide
what to inject for the first
and the second
parameter



Constructor with multiple parameters of the same type



```
public class PaymentService implements IPaymentService{  
    private ICreditCardVerifier visaVerifier;  
    private ICreditCardVerifier mastercardVerifier;  
  
    public PaymentService(ICreditCardVerifier visaVerifier, ICreditCardVerifier mastercardVerifier){  
        this.visaVerifier=visaVerifier;  
        this.mastercardVerifier=mastercardVerifier;  
    }  
}
```

Constructor has
2 arguments of the same
type

```
<bean id="paymentService" class="products.PaymentService">  
    <constructor-arg index="0" ref="visaVerifier" />  
    <constructor-arg index="1" ref="mastercardVerifier" />  
</bean>  
<bean id="visaVerifier" class="products.VisaVerifier"/>  
<bean id="mastercardVerifier" class="products.MastercardVerifier"/>
```

Spring looks at the index
to decide what to inject
for the first and the
second parameter



Setter injection characteristics

- Order of execution:
 1. Instantiate the object / Call the constructor
 2. Do the injection calling the setter method(s)
- Issues:
 - If the injection fails, you have an object in an invalid state (spring will throw exception)
 - If you want to execute initialization code that uses the injected attributes, then you cannot place this code in the constructor, you need to write a separate `init()` method



Constructor injection characteristics

- Order of execution:
 1. Instantiate the object / Call the constructor and do the injection
- Issues:
 - You need constructor chaining with inheritance
 - In case of optional parameters you need multiple constructors



Which one to choose?

- This is a more personal preference.
- If you need the injected attributes in the constructor, use constructor injection or use setter injection with an additional `init()` method.
- If constructor injection results in many different constructors, use setter injection for the optional arguments.



Autowiring

- Spring figures out how to wire beans together
- 3 types of XML or Annotation 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"/>
```



Annotation based Autowiring by name



```
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");  
    }  
}
```

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

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



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();  
    }  
}
```

@Autowired 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"/>
```



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();  
    }  
}
```

@Autowired 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"/>
```

depend where you put autowire could be by value, by name or by Constructor



Annotation based Autowiring with JSR-250 annotations

```
public class CustomerService {  
  
    @Resource(name="myEmailService")  
    private EmailService emailService;  
  
    public void addCustomer(){  
        emailService.sendEmail();  
    }  
}
```

The @Resource annotation works the same as the @Qualifier annotation. The @Resource annotation has an optional parameter 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"/>
```




Autowiring

- Advantage
 - Makes configuration of bean wiring simpler
- Disadvantages
 - The Spring XML file does not contain all the explicit details on how the beans are wired together
 - Autowire by type gives the restriction that you can have only 1 bean of the given type



@Resource and @Inject

- @Resource is the JSR250 standard annotation often used for DI in Java EE.
- @Inject is the JSR330 standard annotation specifically for dependency injection
- Both are (almost) identical to @Autowired



Dependency Injection:

ADDITIONAL INJECTION FEATURES



Additional Injection Features

- Spring can also inject:
 - Values (optionally based on expressions)
 - Collections (Lists, Sets, Maps)
 - Inheritance (parent class values)



Value Injection

Inject the value "English"

```
<bean id="customerService" class="module3.spel.xml.CustomerService" >
  <property name="region" value="English" />
  <property name="defaultdiscount" value="4.5" />
</bean>
```

Inject the value 4.5

```
public class CustomerService implements ICustomerService{
    private String region;
    private double defaultdiscount;
    ...
}
```



Annotation-based configuration

```
<context:annotation-config />
<bean id="customerService" class="module3.spel.annotation.CustomerService" />
<bean id="discountCalculator" class="module3.spel.annotation.DiscountCalculator" />
```

```
public class CustomerService implements ICustomerService{
    @Value("English" )
    private String region;
    @Value("4.5" )
    private double defaultdiscount;
    ...
}
```

Inject the value "English"

Inject the value 4.5



Spring expression language

- Spring configuration can also make use of the Spring Expression Language (SPeL)
- Looks a lot like JSF expression language
- Example:

```
{ systemProperties['user.language'] }
```



XML-based configuration

Inject the value of user.language of the system properties

```
<bean id="customerService" class="module3.spel.xml.CustomerService" >
  <property name="region" value="#{ systemProperties['user.language'] }" />
  <property name="defaultdiscount" value="#{ discountCalculator.defaultdiscount }" />
</bean>
<bean id="discountCalculator" class="module3.spel.xml.DiscountCalculator" >
  <property name="defaultdiscount" value="0.15" />
</bean>
```

Inject the value of the defaultdiscount property of the bean with name discountCalculator

```
public class CustomerService implements ICustomerService{
  private String region;
  private double defaultdiscount;
  ...
}
```

```
public class DiscountCalculator {
  private double defaultdiscount;
  ...
}
```




Annotation-based configuration

```
<context:annotation-config />
<bean id="customerService" class="module3.spel.annotation.CustomerService" />
<bean id="discountCalculator" class="module3.spel.annotation.DiscountCalculator" />
```

```
public class CustomerService implements ICustomerService{
    @Value("#{ systemProperties['user.language'] }" )
    private String region;
    @Value("#{ discountCalculator.defaultdiscount }" )
    private double defaultdiscount;
    ...
}
```

Inject the value of user.language of the system properties

Inject the value of the defaultdiscount property of the bean with name discountCalculator

```
public class DiscountCalculator {
    @Value("0.15" )
    private double defaultdiscount;
    ...
}
```



Injection of lists

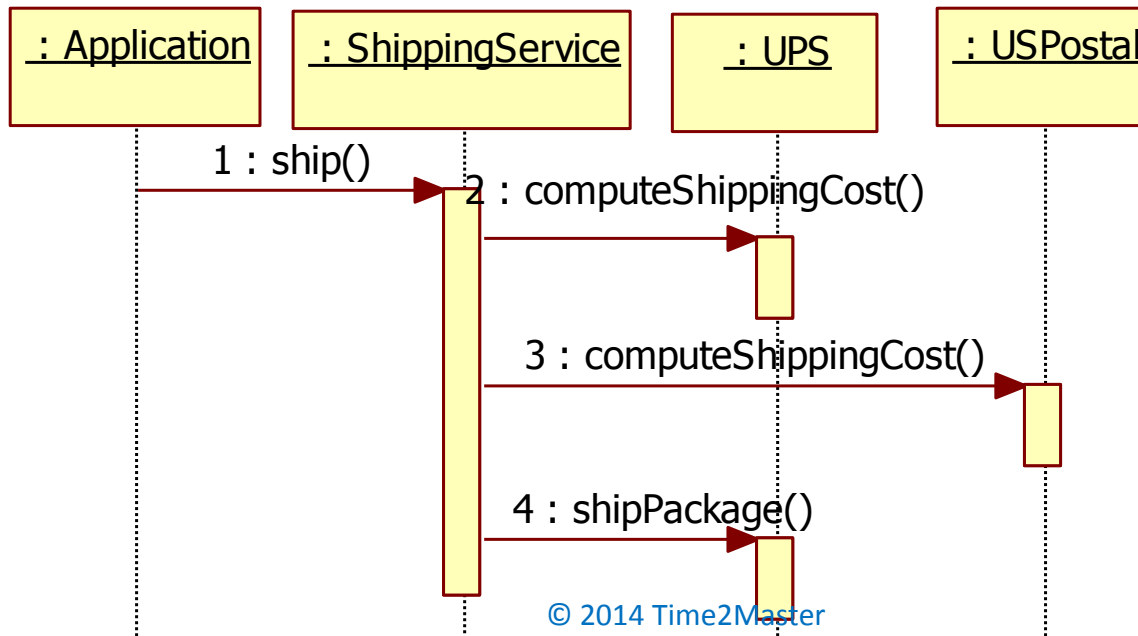
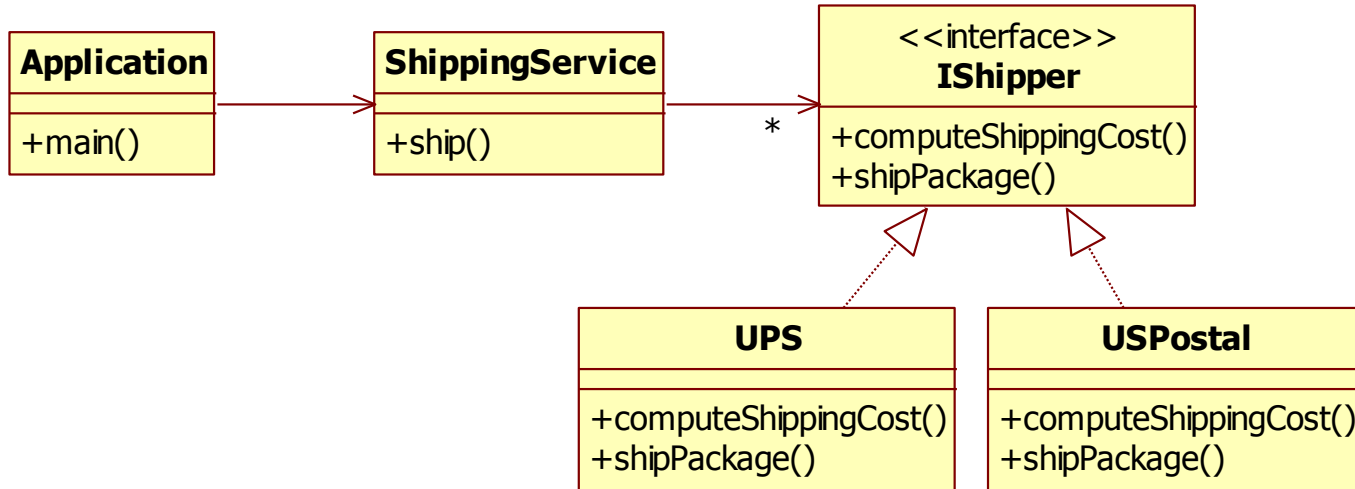
```
public class ShippingService implements IShippingService{  
    public List<IShipper> shippers;  
  
    public ShippingService(List<IShipper> shippers) {  
        this.shippers = shippers;  
    }  
}
```

```
<bean id="shippingService" class="shipping.ShippingService" >  
    <constructor-arg>  
        <list>  
            <bean id="upsShipper" class="shipping.UPS" />  
            <bean id="uspostalShipper" class="shipping.USPostal" />  
        </list>  
    </constructor-arg>  
</bean>
```

Injection of a list
with 2 objects



List injection example





List injection example

```
public interface IShipper {  
    public void shipPackage(Package thePackage, Customer customer);  
    public double computeShippingCost(Package thePackage, Customer customer);  
}
```

```
public class UPS implements IShipper{  
    public void shipPackage(Package thePackage, Customer customer) {  
        System.out.println("package with id= "+thePackage.getId()+" is shipped with UPS");  
    }  
  
    public double computeShippingCost(Package thePackage, Customer customer) {  
        double price= Math.random()*100;  
        System.out.println("UPS charges $" +price+" for package with id= "+thePackage.getId());  
        return price;  
    }  
}
```

```
public class USPostal implements IShipper{  
    public void shipPackage(Package thePackage, Customer customer) {  
        System.out.println("package with id= "+thePackage.getId()+" is shipped with USPostal");  
    }  
  
    public double computeShippingCost(Package thePackage, Customer customer) {  
        double price= Math.random()*100;  
        System.out.println("USPostal charges $" +price+" for package with id= "+thePackage.getId());  
        return price;  
    }  
}
```



List injection example

```
public class ShippingService implements IShippingService{
    public List<IShipper> shippers;

    public ShippingService(List<IShipper> shippers) {
        this.shippers = shippers;
    }

    public void ship(Package thePackage, Customer customer) {
        double lowestPrice=0;
        IShipper cheapestShipper=null;
        // find the cheapest shipper
        for (IShipper shipper : shippers){
            Double price = shipper.computeShippingCost(thePackage,customer);
            if (cheapestShipper == null){
                cheapestShipper=shipper;
                lowestPrice=price;
            }
            else{
                if (price < lowestPrice){
                    cheapestShipper=shipper;
                    lowestPrice=price;
                }
            }
        }
        // ship with the cheapest shipper
        if (cheapestShipper != null){
            cheapestShipper.shipPackage(thePackage,customer);
        }
    }
}
```



List injection example

```
public class Application {  
    public static void main(String[] args) {  
        ApplicationContext context = new ClassPathXmlApplicationContext("springconfig.xml");  
        IShippingService shippingService =  
            context.getBean("shippingService", IShippingService.class);  
  
        shippingService.ship(new Package(123), new Customer());  
        shippingService.ship(new Package(332), new Customer());  
        shippingService.ship(new Package(827), new Customer());  
    }  
}
```

```
public class Package {  
    private int id=0;  
  
    ...  
}
```

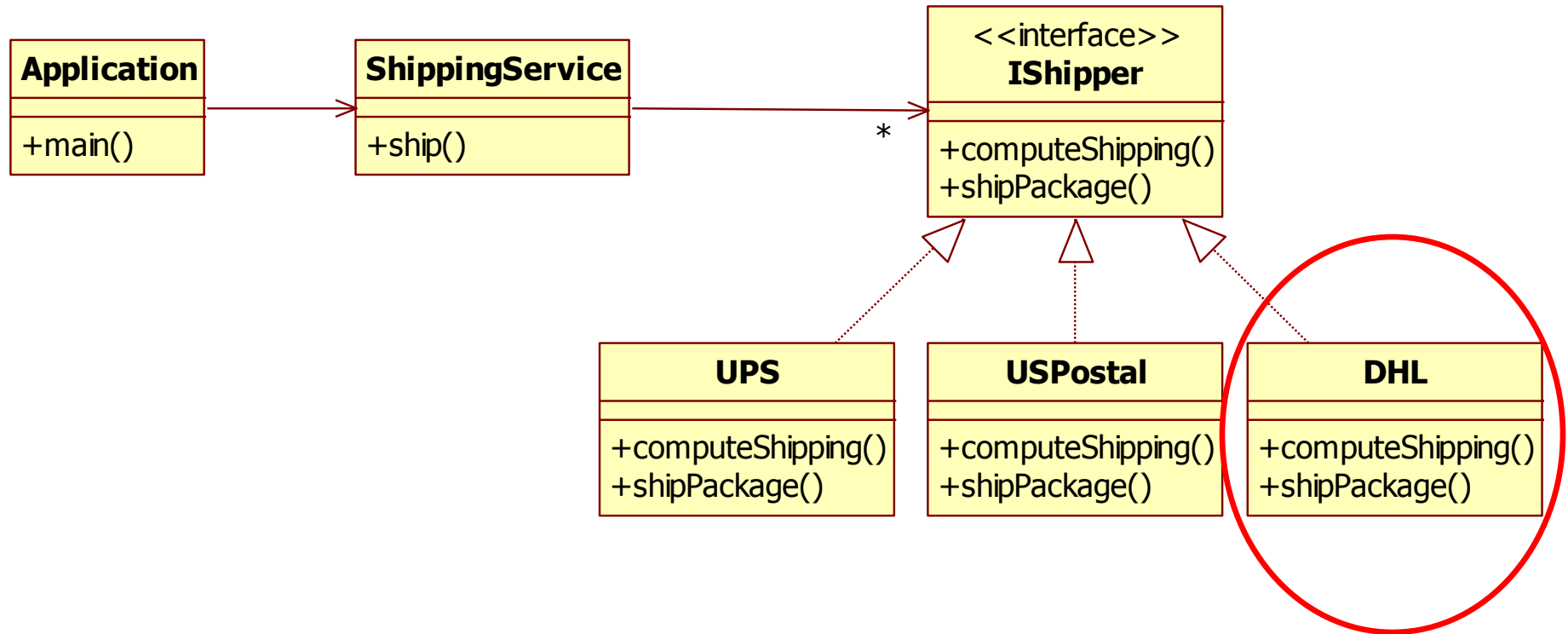
```
public class Customer {  
  
}
```

```
<bean id="shippingService" class="shipping.ShippingService" >  
    <constructor-arg>  
        <list>  
            <bean id="upsShipper" class="shipping.UPS" />  
            <bean id="uspostalShipper" class="shipping.USPostal" />  
        </list>  
    </constructor-arg>  
</bean>
```

Inject a list
with 2 objects



Add a new shipper





Add a new shipper

```
public class DHL implements IShipper{
    public void shipPackage(Package thePackage, Customer customer) {
        System.out.println("package with id= "+thePackage.getId()+" is shipped with DHL");
    }

    public double computeShippingCost(Package thePackage, Customer customer) {
        double price= Math.random()*100;
        System.out.println("DHL charges $" +price+" for package with id= "+thePackage.getId());
        return price;
    }
}
```

```
<bean id="shippingService" class="shipping.ShippingService" >
    <constructor-arg>
        <list>
            <bean id="upsShipper" class="shipping.UPS" />
            <bean id="uspostalShipper" class="shipping.USPostal" />
            <bean id="dhlShipper" class="shipping.DHL" />
        </list>
    </constructor-arg>
</bean>
```

Inject a list
with 3 objects



Injection of a set

set

```
public class ShippingService implements IShippingService{  
    public Set<IShipper> shippers;  
  
    public ShippingService(Set<IShipper> shippers) {  
        this.shippers = shippers;  
    }  
    ...  
}
```

set

```
<bean id="shippingService" class="shipping.ShippingService" >  
    <constructor-arg>  
        <set>  
            <bean id="upsShipper" class="shipping.UPS" />  
            <bean id="uspostalShipper" class="shipping.USPostal" />  
            <bean id="dhlShipper" class="shipping.DHL" />  
        </set>  
    </constructor-arg>  
</bean>
```



Injection of a map

map

```
public class ShippingService implements IShippingService{
    public Map<String,IShipper> shippers;

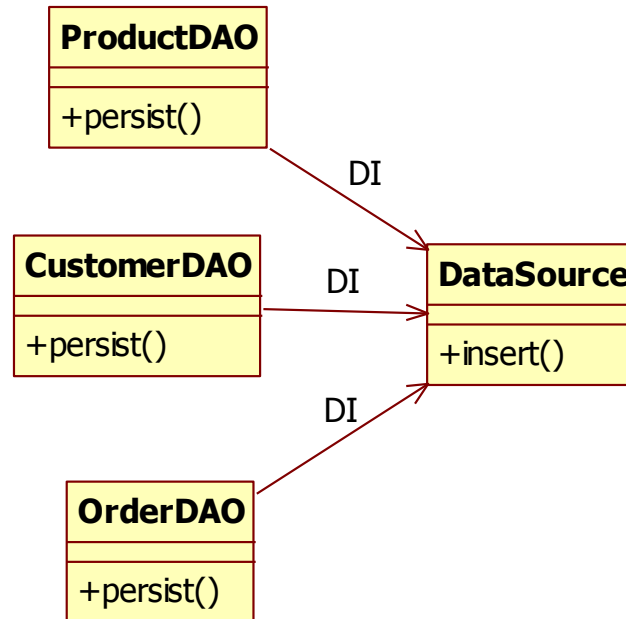
    public ShippingService (Map<String,IShipper> shippers) {
        this.shippers = shippers;
    }
    ...
}
```

map

```
<bean id="shippingService" class="shipping.ShippingService" >
    <constructor-arg>
        <map>
            <entry key="one" value-ref="upsShipper"/>
            <entry key="two" value-ref="uspostalShipper"/>
            <entry key="three" value-ref="dhlShipper"/>
        </map>
    </constructor-arg>
</bean>
<bean id="upsShipper" class="shipping.UPS" />
<bean id="uspostalShipper" class="shipping.USPostal" />
<bean id="dhlShipper" class="shipping.DHL" />
```



DI without inheritance



```
<bean id="orderDAO" class="module2.noinheritance.OrderDAO" >
  <property name="datasource" ref="datasource" />
</bean>
<bean id="customerDAO" class="module2.noinheritance.CustomerDAO" >
  <property name="datasource" ref="datasource" />
</bean>
<bean id="productDAO" class="module2.noinheritance.ProductDAO" >
  <property name="datasource" ref="datasource" />
</bean>
<bean id="datasource" class="module2.noinheritance.DataSource" />
</beans>
```

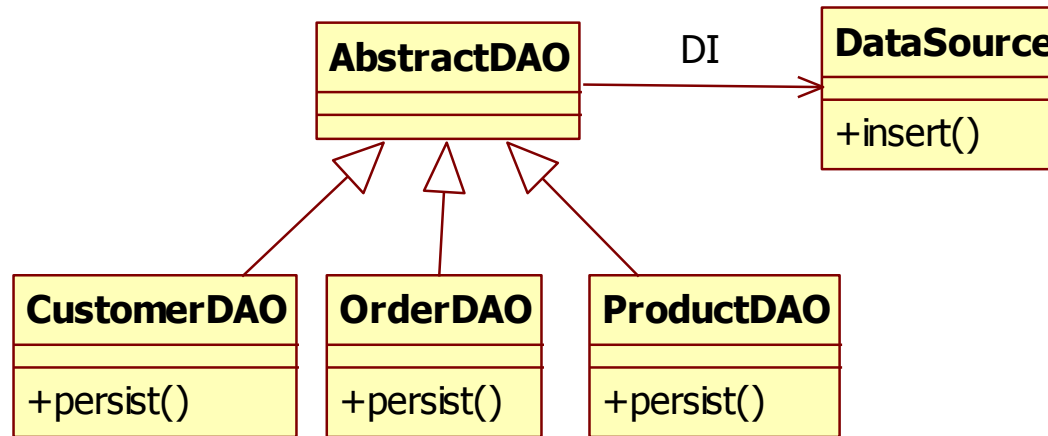
Injection of the datasource

Injection of the datasource

Injection of the datasource



DI with inheritance (1/2)



```
public abstract class AbstractDAO implements IDAO{
    protected DataSource datasource;

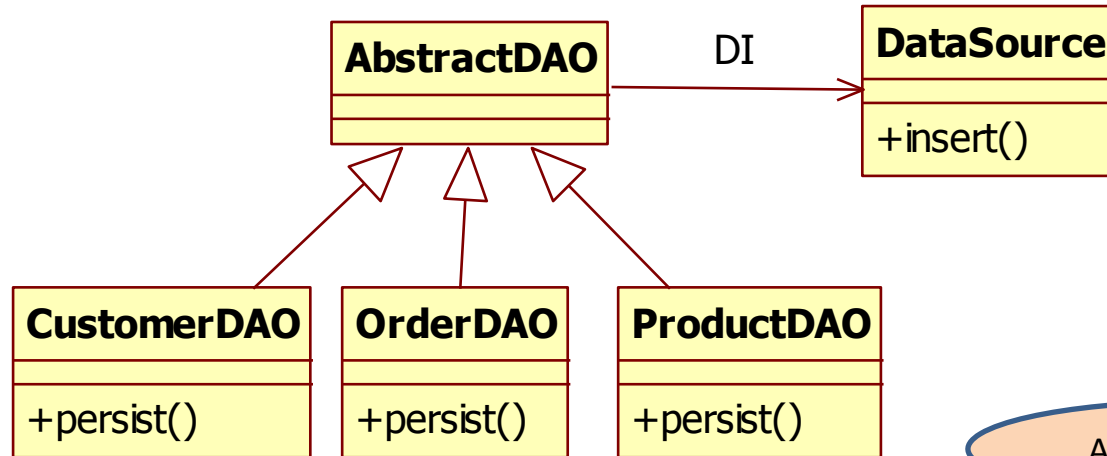
    public void setDatasource(DataSource datasource) {
        this.datasource = datasource;
    }

    public abstract void persist();
}
```

```
public class CustomerDAO extends AbstractDAO{
    public void persist() {
        datasource.insert();
    }
}
```



DI with inheritance (2/2)



Abstract="true"

```
<bean id="abstractDAO" class="module2.inheritance.AbstractDAO" abstract="true">
  <property name="datasource" ref="datasource" />
</bean>
```

Injection of the datasource

```
<bean id="orderDAO" class="module2.inheritance.OrderDAO" parent="abstractDAO" />
<bean id="customerDAO" class="module2.inheritance.CustomerDAO" parent="abstractDAO" />
<bean id="productDAO" class="module2.inheritance.ProductDAO" parent="abstractDAO" />
<bean id="datasource" class="module2.inheritance.DataSource" />
```

Specify the parent class



Additional Injection Features

- Allowing Spring to inject more than just bean references opens up a completely new area of flexibility and configuration options.
- Greater abstraction provides greater flexibility. We see it here with spring, we've seen it with polymorphism, and it's the same with the unified field, it's the field of all possibilities.



Dependency Injection:

ALTERNATE WAYS TO CONFIGURE THE APPLICATION CONTEXT



Alternate Configuration Styles

- Other than configuring the Spring application context through XML, we can also
 - Configure beans through annotations (also known as classpath scanning)
 - Configure beans with Java code



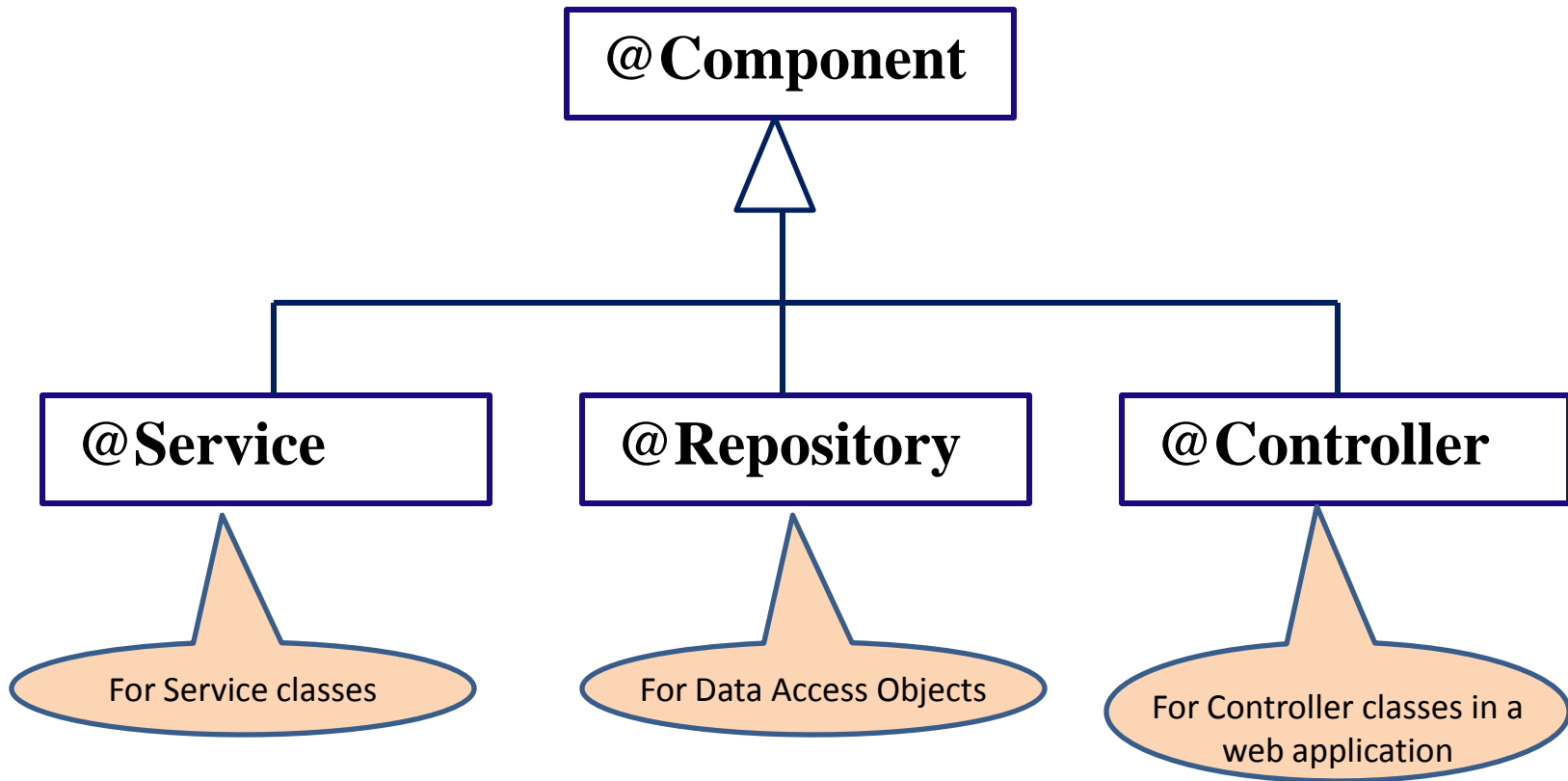
Classpath scanning

- 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();
    }
}
```



@Scope - Prototype

- The default scope is “singleton”

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

Set the scope to prototype



@Lazy - Initialization

- By default beans are eagerly instantiated

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

Sets lazy instantiation



Java Configuration

- Spring beans can also be configured with (annotated) Java code instead of XML

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

Same configuration

```
<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();  
    }  
}
```



@Lazy

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

Lazy initialization



@Scope

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

    @Bean
    @Scope(value="prototype")
    public EmailService emailService() {
        return new EmailServiceImpl();
    }
}
```

Set scope to prototype



Mixing Configuration Styles

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

Definition of 2 Spring beans

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

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

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

The EmailService is injected



What to use

- Which configuration style should we use?
 - XML configuration
 - Classpath scanning and Autowiring
 - Java configuration



Option 1: XML configuration

```
<bean id="customerService" class="module3.di.CustomerServiceImpl">
  <property name="emailService" ref="emailService"/>
</bean>
<bean id="emailService" class="module3.di.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
- Disadvantages
 - Large verbose XML file(s)
 - No compile time type safety
 - Less refactor-friendly



Option 2: Classpath scanning and Autowiring

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

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

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

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

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




Classpath scanning and Autowiring

- Advantages
 - All information (configuration and logic) in one place: the Java code
 - More type safe
- Disadvantage
 - Configuration in the Java code



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
 - Requires a bit more code



Alternate Configurations

- In this course we will be using XML configuration files because they suite our needs best.
- It's very important to also know about the other configuration options, our needs in this course may not be your needs, or the needs of the company you may work for.



Active Learning

- What are the disadvantages of setter injection?

you can not initialize in the constructor

- Why is annotation configuration called 'classpath scanning'?

Text



Summary

- Dependency Injection is a technique to wire objects together in a flexible way
- There are 3 ways to implement DI
 - Setter injection, Constructor injection, Autowiring
- Spring also allows us to inject:
 - values, collections, and inheritance
- There are 3 ways to configure a Spring application
 - XML configuration file
 - Java Configuration
 - Classpath scanning and autowiring



Main Point

- Dependency Injection lets us specify how objects should be connected, and then Spring (when it creates the objects), connects them for us.
- *Science of Consciousness*: The whole is greater than the sum of the parts – how the parts are connected is as important as the parts themselves