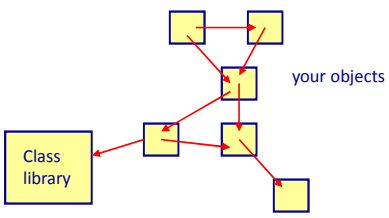
What is Spring?

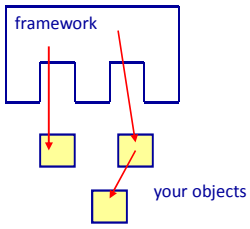
- Lightweight enterprise Java framework
- Open source
- Goal: make developing enterprise Java applications easier

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

2

A basic Spring application

Create an ApplicationContext based on springconfig.xml

```
package module2.helloworld;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Application {
  public static void main(String[] args) {
    ApplicationContext context = new
                    ClassPathXmlApplicationContext("module2/helloworld/springconfig.xml");
    CustomerService customerService = context.getBean("customerService", CustomerService.class);
    customerService.sayHello();
  }
}
```

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

```
package module2.helloworld;

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-3.0.xsd">

   <bean id="customerService" class="module2.helloworld.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
```

customerService1

customerService2

Eager-instantiation of beans

```
public class Application {
  public static void main(String[] args) {
    System.out.println("1");
    ApplicationContext context = new
           ClassPathXmlApplicationContext("/module2/eagerinstantiation/springconfig.xml");
    System.out.println("2");
    CustomerService customerService = context.getBean("customerService", CustomerService.class);
    System.out.println("3");
    customerService.addCustomer("Frank Brown");
    System.out.println("4");
  }
}
```

```
public class CustomerServiceImpl implements CustomerService {
  public CustomerServiceImpl() {
    System.out.println("calling constructor of CustomerServiceImpl");
  }

  public void addCustomer(String customername) {
    System.out.println("calling addCustomer of CustomerServiceImpl");
  }
}
```

```
<bean id="customerService" class="module2.eagerinstantiation.CustomerServiceImpl" />
```

```
1
calling constructor of CustomerServiceImpl
2
3
calling addCustomer of CustomerServiceImpl
4
```

The CustomerService bean is eagerly instantiated

 5

What is Dependency Injection?

- Technique to wire objects together in a flexible way
  - We can change the wiring without changing code
  - Open-closed principle

 6

Different way's to "wire"2 object together

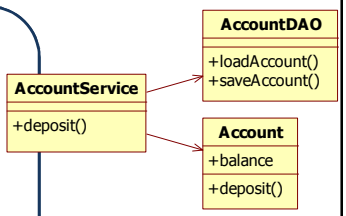
1. Instantiate an object directly
2. Use an interface
3. Use a factory object
4. Use Spring Dependency Injection

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

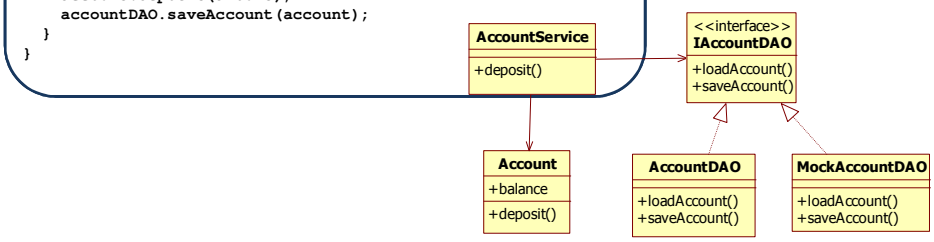
2. Use an Interface

```
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

 9

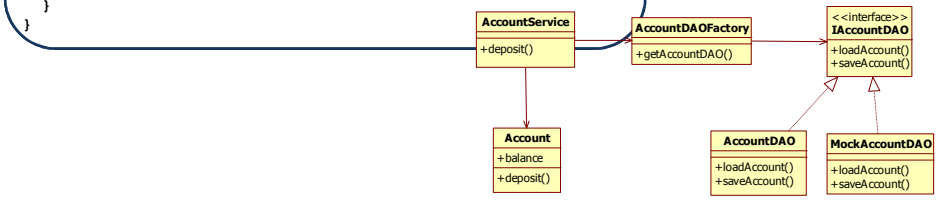
3. Use a factory object

```
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

 10

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

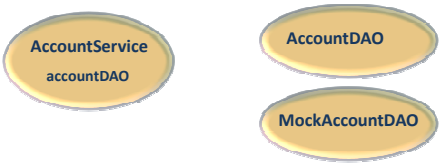
```
<bean id="accountService" class="AccountService">
  <property name="accountDAO" ref="accountDAO" />
</bean>
<bean id="accountDAO" class="AccountDAO" />
<bean id="mockAccountDAO" class="MockAccountDAO" />
```

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

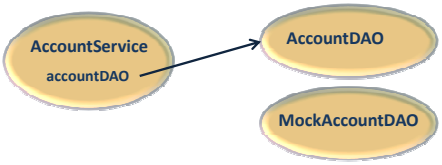
How does DI work?

```
<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



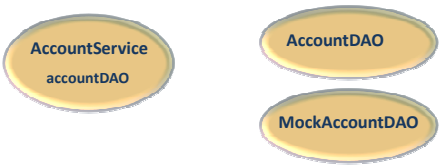
2. Spring then connects the accountDAO attribute to the AccountDAO instance



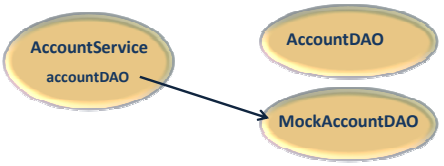
Change the wiring

```
<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



 13

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" />
```

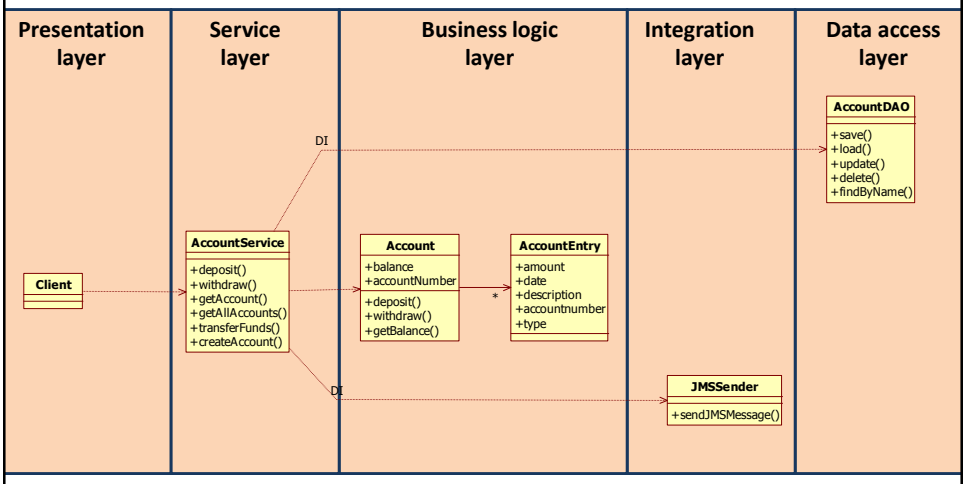
- Flexibility: it is easy to change the wiring between objects without changing code
- Unit testing becomes easier
- Code is clean

 14

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

Application layers



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

Injection of lists

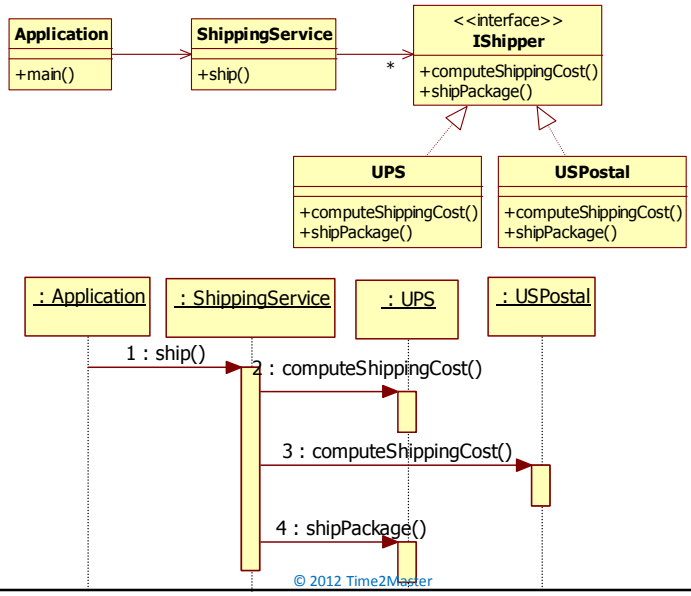
```
public class ShippingService implements IShippingService{
  public List<IShipper> shippers;

  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;

    …
}
```

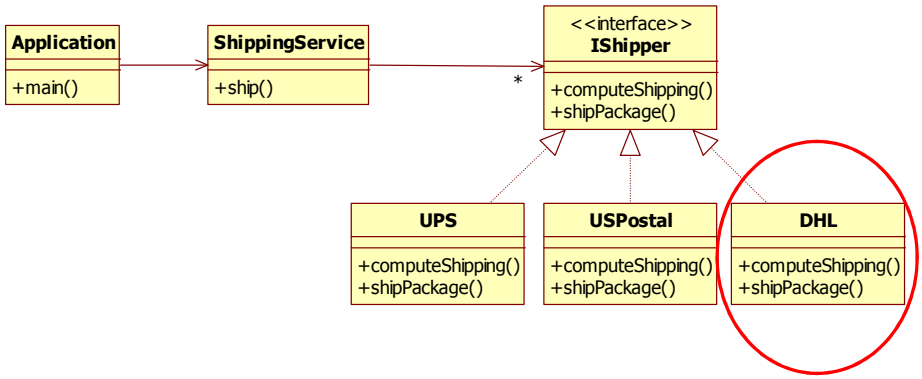
```
public class Customer {

}
```

```
<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



 23

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

 24

Injection of a 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>
```

set

Injection of a 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" />
```

map