ntier TRAINING

Section 1

Spring Introduction

Spring Background
Dependency Injection

Section Objectives

- Configure Spring to inject required dependencies at runtime
- Explain the terms Inversion of Control and Dependency Injection
- Explain the use of PropertyEditors and Converters

ntier TRAINING

Spring Introduction

Spring Background

Spring Background
Dependency Injection

Spring

- Main source of information is SpringSource.org

- Basic Concepts:
  - JEE should be easier to use
  - OO design is more important than any implementation technology, such as JEE
  - Testability is essential, and a framework such as Spring should help make your code easier to test
  - Spring should not compete with good existing solutions but should foster integration

Development with Spring

- Spring development makes heavy use of patterns
  - As you dive into Spring, you will learn to love (or hate…) patterns

- A **design pattern** is a general reusable solution to a commonly occurring problem

- Patterns attempt to aid the developer in producing better code more often
  - This too can go overboard and get the developer in trouble

- An example of using patterns would be separating instantiation from implementation to reduce coupling (Factory Pattern)

What is Spring?

- A lightweight framework that addresses each tier in a Web application
  - Presentation layer – An MVC framework that is most similar to Struts but is more powerful and easier to use
  - Business layer – Lightweight IoC container and AOP support
  - Persistence layer – DAO template support for popular ORMs and JDBC
- Simplifies persistence frameworks and JDBC
- Complimentary: Not a replacement for a persistence framework
  - Helps organize your middle tier and handle typical JEE plumbing problems
  - Reduces code and speeds up development

POJO Based

- Do I have to use all components of Spring?
  - Spring is a non-invasive and portable *framework* that allows you to introduce as much or as little as you want to your application
- Promotes decoupling and reusability
- POJO Based
  - Allows developers to focus more on business logic and less on plumbing problems
  - Reduces or alleviates code littering, ad hoc singletons, factories, service locators and multiple configuration files
- Removes common code issues like leaking connections and more
  - Built-in aspects such as transaction management
  - Most business objects in Spring apps do not depend on the Spring framework

Spring Nuts and Bolts

- A Container
  - Creates objects and makes them available to your application
  - Uses a declarative XML structure to define components that "live" in the container

- A Framework
  - Provides an infrastructure of classes that make it easier to accomplish tasks
  - For example, JDBCTemplate provides an extremely simple interface for JDBC objects

Dependency Management

- Manages collaboration (dependencies) between Plain Old Java Objects (POJOs)
  - Code to interfaces

- Uses Spring to instantiate specific interface implementations
  - Don’t need

    ```
    InterfaceType anObject = new InterfaceImpl();
    ```
  - Uses Spring to provide specific interface implementations to your objects

Inversion of Control – IoC

- The basic concept of the Inversion of Control pattern is that programmers don’t need to create your objects but instead, they need to describe how they should be created e.g. factories, configuration files.
- Don’t directly connect your components and services together in code but describe which services are needed via a configuration technique
- Then, the IoC container in the case of the Spring framework is responsible for all this. In an IoC scenario, the container creates all the objects, connects them together by setting the necessary properties, and determines when methods will be invoked

Dependency Injection

- Dependency injection is a pattern used to create instances of objects that other objects rely on without knowing at compile time which implementation class will be used to provide that functionality. With Spring, the implementation of dependency injection can be achieved via setter and/or constructor injection techniques

- Inversion of control relies on dependency injection because a mechanism is needed in order to activate the components providing the specific functionality

- IoC relies on the dependency inversion principle, from the SOLID acronym of object orientated design, to achieve the decoupling of software modules

ntier
TRAINING

Spring Introduction

Dependency Injection

Spring Background
Dependency Injection

Dependencies

- In a Java Application, different objects work together i.e., objects collaborate to accomplish a goal and subsequently have dependencies

- Such direct coded dependencies couple objects to each other directly. We will attempt to explain how Spring can help us avoid such brittle dependencies. Our simplistic example consists of a Service delegating to a Data Access Object (DAO). We will start with two interfaces to define our domain context

```
public interface EmployeeService {
      EmployeeDao getDao();
}
```

```
public interface EmployeeDao {
}
```

Reduce Coupling

- Now for our implementation classes

```
public class EmployeeServiceImpl implements EmployeeService{
   @Override
   public EmployeeDao getDao() {
       return new EmployeeDaoImpl;
   }
}
```

```
public class EmployeeDaoImpl implements EmployeeDao  {

}
```

- Our problem is the tight coupling in the getDao() method. Our Service implementation is tied directly to an implementation class. The dependency is in the code and will require a code change if another EmployeeDao implementation class is required later to accommodate a requirement change.

Hide the Implementation Details

- Let's try decoupling further via a Factory

```
public class EmployeeServiceImpl implements EmployeeService{
    @Override
    public EmployeeDao getDao() {
        return EmployeeDaoFactory.getInstance();
    }
}
```

- This is better. The decision as to what DAO will be used by our Service is given and encapsulated by another class

- However, the client class STILL has to look up the dependency object itself from the Factory

Let Someone Else Do the Heavy Lifting

- Now, let’s have another class create the DAO and give it to our Service class instance via a setter

```
public class EmployeeServiceImpl implements
EmployeeService{
    private EmployeeDao dao;
    public void setDao(EmployeeDao dao) {
        this.dao = dao;
    }
    public EmployeeDao getDao() {
        return dao;
    }
}
```

```
public static void main(String args[]) {
      EmployeeService service = new EmployeeServiceImpl
      service.setDao(new EmployeeDaoImpl);
      service.getDao();
}
```

That’s Good, but Spring Makes It Easier

- What if we could let someone else do the client Class’s work, but in configuration files and not code?

- That is the basis of Spring’s **Inversion of Control** in the form of **Dependency Injection**
  - It operates as a huge factory that looks up dependencies of a target class that it is instantiating
  - Via reflection, it creates instances of the dependencies themselves, then uses the target class’s setters or constructors to inject them into the target class. There is NO code lookup in the target class, nor in some master client class
  - It has the dependencies between objects but NOT in code in configuration
  - Let's take a look at Spring to see how this works

Spring Inversion of Control

- Spring manages beans for you
- Instead of the developer being responsible for creating all the beans, Spring uses an **Inversion of Control** implementation
- **Inversion of Control** in Spring frequently refers to **Dependency Injection**
  - The basic principle is that beans define their dependencies only through setter or constructor arguments
  - Then, it is the job of the container to actually inject those dependencies when it creates the bean
- Essentially, this is the inverse of the bean instantiating or locating its dependencies on its own

Spring BeanFactory

- Spring provides the org.springframework.beans.factory.BeanFactory interface to create and manage our beans
  - Being that it is an interface, implementations must be used
  - There are MANY implementations of BeanFactory to choose from

- The BeanFactory implementation is the actual *container* which instantiates, configures, and manages a number of beans

- These beans typically collaborate with one another and thus have dependencies between themselves

Know the API

- getBean(String beanname)
  - Returns an object registered with this beanname
- getBean(String beanname, Class beantype)
  - Returns an object of this class type registered with this beanname
- getType(String beanname)
  - Returns the type of object that is registered with this name
- containsBean(String beanname)
  - Returns a boolean true or false if the bean is registered
- getAliases(String beanname)
  - Returns a String array of alternate names for the named bean
- is{Protoype,Singleton}(String beanname)
  - Returns a boolean depending on the type of bean
    - Default type is a singleton (more on this in a upcoming section)

ApplicationContext

- The ApplicationContext builds on top of the BeanFactory
  - A sub interface of BeanFactory
  - Adds other functionality such as:
    - Easier integration with Spring's AOP
    - Message resource handling (for use in internationalization)
    - Event propagation
    - Declarative mechanisms to create the ApplicationContext
    - Parent contexts
    - Application-layer specific contexts such as the WebApplicationContext

- BeanFactory provides the configuration framework and basic functionality, and the ApplicationContext adds more enterprise-specific functionality

ApplicationContext Implementations

- Common Implementations:
  - ClassPathXmlApplicationContext
    - Looks in the ClassPath for the configuration file
  - FileSystemApplicationContext
    - Looks on the FileSystem for the configuration file
  - XmlWebApplicationContext
    - Looks in the WEB-INF for the configuration file

```
ApplicationContext appContext = new
ClassPathXmlApplicationContext(“beans.xml”);
// OR
ApplicationContext appContext = new
FileSystemXmlApplicationContext(“src/beans.xml”);
```

Xml Configuration

- The ClassPathXmlApplicationContext will need to use Spring schemas in an XML file to externalize Spring Configuration from the Java classes themselves
  – The root element is <beans> and individual beans are defined with a <bean> tag nested inside the beans tag

```
<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-4.0.xsd">


</beans>
```

Define a BeanDefinition

- Let’s add to the bean tag

```
<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-4.0.xsd">


   <bean id="emplService" name="employeeServ"
            class="com.employeeRus.service.EmployeeServiceImpl" />
</beans>
```

- Each “managed bean” has a unique identifier as denoted by an “id” attribute in the beans tag. Note that here in the configuration, we have the implementation class name fully qualified with its package

Wire Up a Dependency

- Our bean tag has defined a BeanDefinition which, upon the loading of the ApplicationContext, will be parsed into a BeanDefinition object. These BeanDefinitions are actually used by the IoC container to create Managed Bean instances

- Now, via setter injection, inject in a dependency

```
<bean id="emplService" name="employeeServ"
              class="com.employeeRus.service.EmployeeServiceImpl"  >
        <property name="dao" ref="emplDao"/>
  </bean>

<bean id="emplDao" class="com.employeeRus.dao.EmployeeDaoImpl"/>
```

Setter Injection

```
public class EmployeeDaoImpl
implements EmployeeDao  {
```

```
<bean id="emplService" name="employeeServ"
        class="com.employeeRus.service.Employee
ServiceImpl"  >
      <property name="dao" ref="emplDao"/>
  </bean>


<bean id="emplDao"
class="com.employeeRus.dao.EmployeeDaoImpl"/>
```

```
public class EmployeeServiceImpl implements
EmployeeService{
        private EmployeeDao dao;
        public void setDao(EmployeeDao dao) {
                this.dao = dao;
        }
        public EmployeeDao getDao() {
                return dao;
        }
}
```

Interacting with Spring

- So, how does Spring do this? Well, our Spring client first of all instantiates an ApplicationContext instance to represent the IoC Container
- It needs the XML configuration file
- Once that is done, BeanDefinitions are created from the configuration. Then, Spring Managed Bean instances are created EAGERLY from the bean definitions

```
private ApplicationContext context;
private EmployeeService employeeService;

@Before
public void setUp() throws Exception {
   context = new ClassPathXmlApplicationContext("beans.xml");
   employeeService =  context.getBean("emplService",
       EmployeeService.class);
}
```

What Does Our Client Class Know Now?

- Now we have to confirm that Spring has indeed injected the Service object’s dependency

```
@Test
public void testEmployeeServiceSetterInjection() {
   assertThat(employeeService, notNullValue());
   assertThat(employeeService.getDao(), notNullValue());
}
```

- No NullPointerException should occur. Delegation should happen seamlessly as a consequence of the ApplicationContext taking care of the dependency
- Our motivation is requirement change. How are the above tests affected if we change the XML configuration to use a different DAO?

```
<bean id="emplDao" class="com.employeeRus.dao.AnotherDaoImpl"/>
```

Singleton

- Spring has been designed with the Singleton pattern in mind
- However, it is not synonymous with the traditional Java singleton, *i.e.*, one instance of a class per JVM
- It is based off a single instance per “id” attribute of the BeanDefinition
- If we repeatedly ask the ApplicationContext for the bean via the same “id,” we will get the same instance

Id or Name?

- A bean definition can be identified via the “id” attribute, as we have seen, but also by a “name” attribute

- The “name” attribute can accept a single comma-delimitated or space-delimitated list. The list defines aliases for the BeanDefinition. Each one could be used in an ApplicationContext.getBean(“alias”) call. Another form of alias is by using the <alias> tag itself

```
<alias alias="emplAlias" name="employeeServ"/>
```

- The Singleton pattern is maintained via “id” (if present). If it is not present, the first “name” attribute can be used instead to ensure uniqueness per ApplicationContext

What If I Do Not Give an Id?

- Consider these two bean definitions

```
<bean  class="com.employeeRus.dao.EmployeeDaoImpl"/>
<bean  class="com.employeeRus.dao.EmployeeDaoImpl"/>
```

- There is no id attribute, however Spring will have generated an id for us based off class name # occurrence instance
  - com.employeeRus.dao.EmployeeDaoImpl#0
  - com.employeeRus.dao.EmployeeDaoImpl#1

- If you provide a name attribute and no id, Spring will interpret the name as the bean's id and not generate one

Benefits

- Dependency Injection **reduces the coupling** between modules **in your code**
- The dependencies are still there but **not in the code**
- This leads to **more flexible** code that is **easier to maintain**. Although, you will find yourself working in the Spring configuration files

ntier TRAINING

Lab 1.1

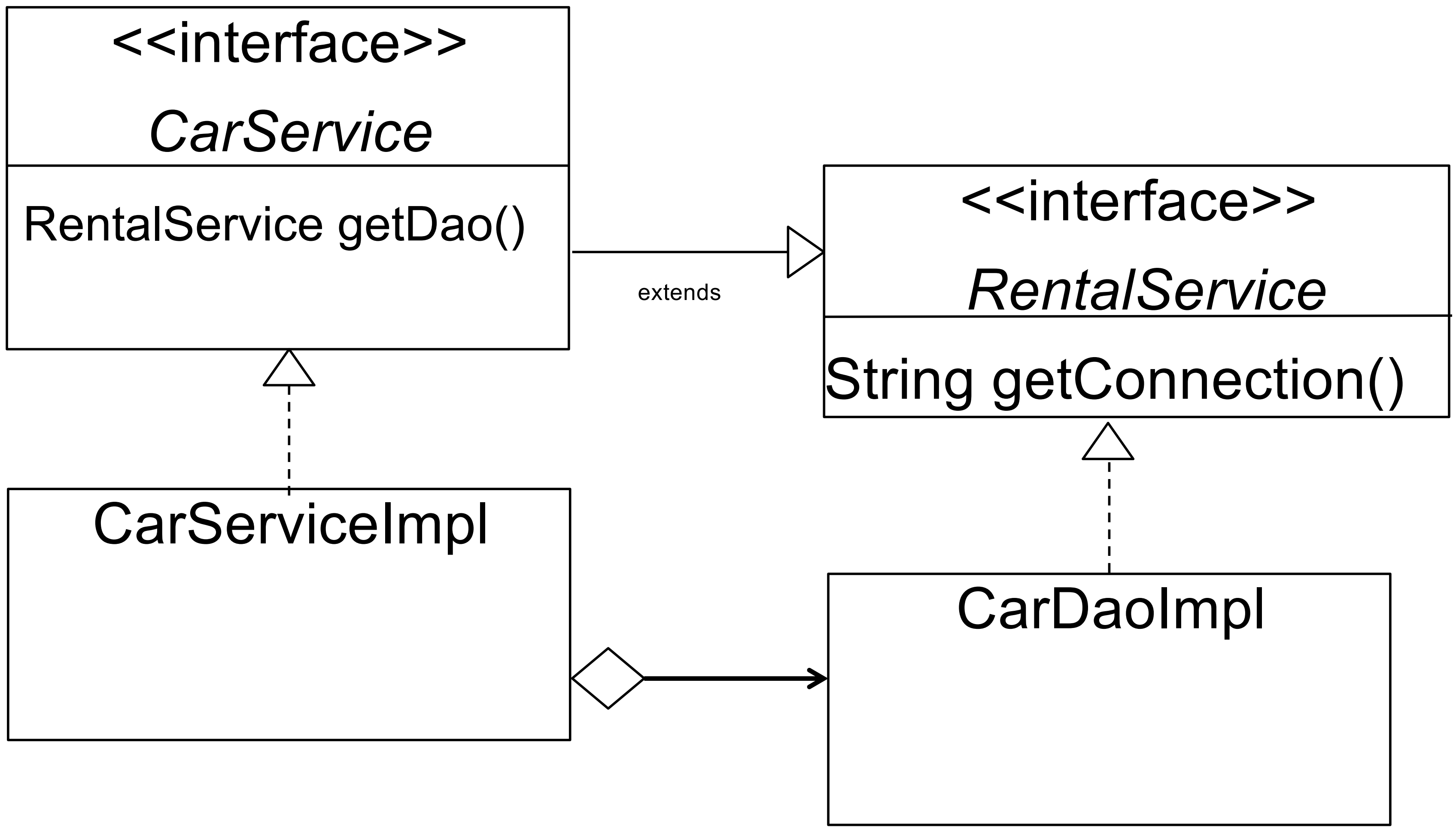
Setting Up the Environment

Lab 1.1 – Set Up

- Objective: Configure your application to use Spring and achieve dependency injection via setter injection
- All projects will use Maven to secure their jar dependencies
- **Secure from Subversion the starter project for Lab01.1**

Lab 1.1 – UML

- You are to wire up the following beans using Spring



- Create the Java artifacts as per the UML

Lab 1.1 – Implementation

- In the src/main/resources/beans.xml file:
  - Create a bean definition for the class CarServiceImpl. Give this definition a id and a name
  - Create a second bean definition for the class CarDaoImpl. Give the definition an id attribute
  - Inject the DAO into the Service definition via a “property tag”
  - Create an “alias” tag for your service definition
  - Note you will need a setter method in your Service class in order for the “injection” to occur otherwise you will get an exception
- Create a JUnit class com.rentals.service.CarServiceTest in a “test” source folder. Use the chapter slides as an example to follow. You are to assert that the injection has taken place successfully

Using Multiple XML Configuration Files

- Although we could have one XML file import other files…

```
<import resource="moreBeans.xml"/>
 <bean id="emplService" name="employeeServ"
        class="com.employeeRus.service.EmployeeServiceImpl">
        <property name="dao" ref="emplDao"/>
  </bean>
```

- …this tightly couples our XML file to another file for good. However, we could keep them separate and use a ClassPathXmlApplicationContext constructor

```
<bean id="emplService" name="employeeServ"
        class="com.employeeRus.service.EmployeeServiceImpl"  >
        <property name="dao" ref="emplDao"/>
  </bean>
```

```
<bean id="emplDao" class="com.employeeRus.dao.EmployeeDaoImpl"/>
```

```
ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml","moreBeans.xml");
```

Simple Value Injection

- So far, we have seen beans injected into other beans via setters
- What about simple value properties such as Strings? Instead of referencing a bean, we simply use the “value” attribute

```
<bean id="emplService" name="employeeServ"
class="com.employeeRus.service.EmployeeServiceImpl"  >
        <property name="dao" ref="emplDao"/>
        <property name="active" value="true" />
        <property name="mode" value="true" />
</bean>
```

- We can see that the class EmployeeServiceImpl has setter methods for the properties “active” and “mode”
- We can also inject null values (see notes)

PropertyEditors

- What if the property to be injected is not a String but, for example, a double?

```
<bean id="connection" class="com.employeeRus.dao.Connection">
    <property name="pooledConnections" value="8"/>
    <property name="timeout" value="39.00"/>
</bean>
```

- So I am looking for a property of timeout? Wrong. You are looking for the setter called setTimeout that takes a double. So there has to be String to double PropertyEditor at work
- Spring provides a number of standard PropertyEditors in the package org.springframework.beans.propertyeditors that implement the interface **PropertyEditor** or extend the helper class **PropertyEditorSupport**
- Above, we are using the class CustomNumberEditor as a means to convert from text “39.00” to a number 39.00 that is the property type of the target setter’s argument

Lab 1.2

Simple Value Injection

Lab 1.2 – Set Up

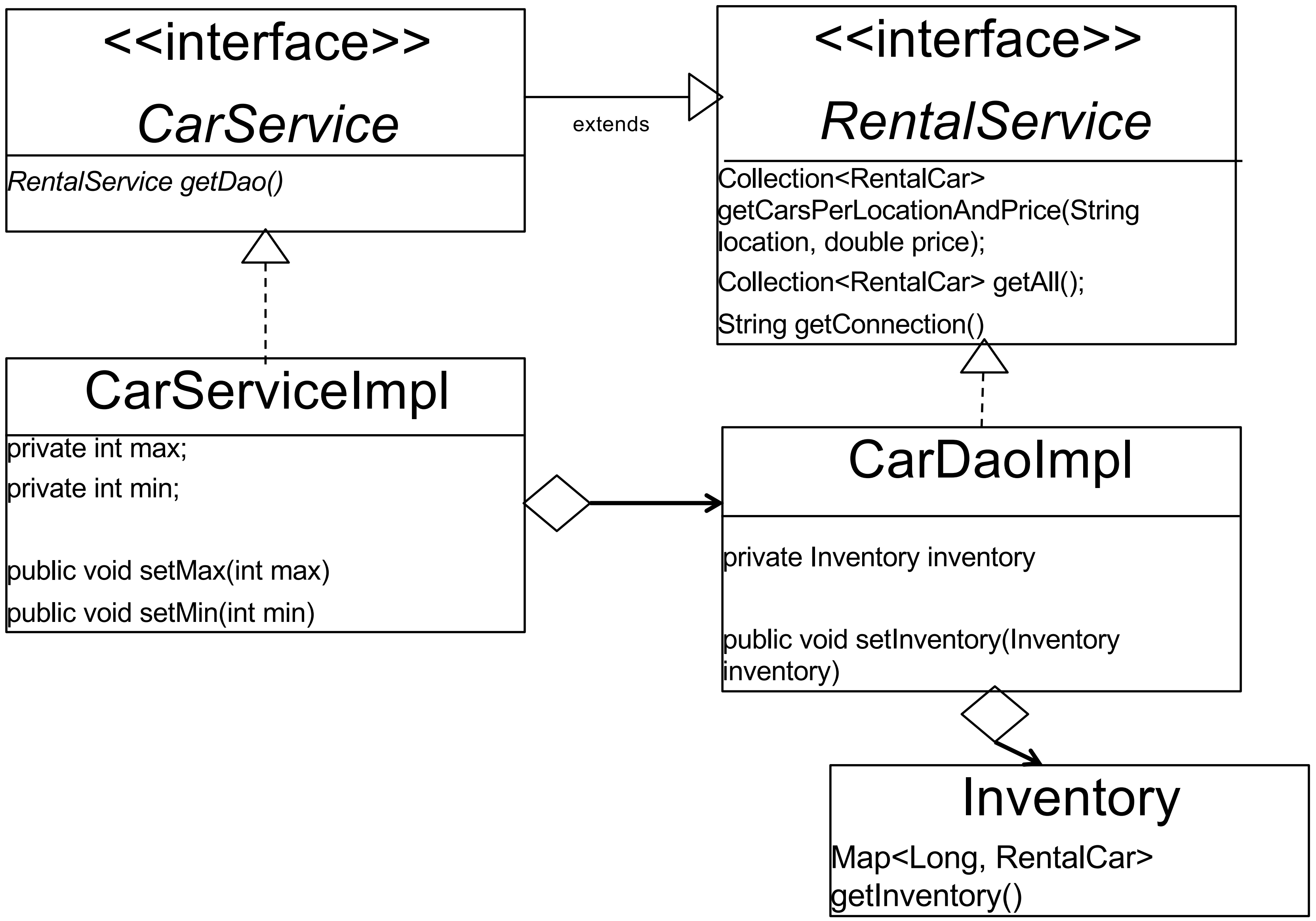
- Objective: Implement Simple Value Injection and connect our Service and DAO domain up to some real data

- In your lab setup file for this lab you will find the following classes:
  - Inventory
  - RentalCar
  - CarDaoImpl

- Place them into their respective packages in your project

Lab 1.2 – UML

- You are to wire up the following beans using Spring



<<interface>>
CarService
RentalService getDao()
extends
<<interface>>
RentalService
Collection<RentalCar> getCarsPerLocationAndPrice(String location, double price);
Collection<RentalCar> getAll();
String getConnection()
CarServiceImpl
private int max;
private int min;
public void setMax(int max)
public void setMin(int min)
CarDaoImpl
private Inventory inventory
public void setInventory(Inventory inventory)
Inventory
Map<Long, RentalCar> getInventory()

Lab 1.2 – Service

- In class CarServiceImpl:
  - Add two new properties, min and max, of type int. You will need setters for these data properties. These properties will be used to limit the number of items returned from our DAO
  - You must implement the methods in the interface CarService
    - Collection<RentalCar> getCarsPerLocationAndPrice(String location, double price);
    - Collection<RentalCar> getAll();
    - Both delegate to the injected RentalService
    - Both need to check for any returned Collection of RentalCar from the DAO. We will only send back to the caller of the Service those items between an index of min inclusive to max exclusive
    - For example, we get 10 RentalCars back from the DAO. Our min and max properties are three and seven respectively. This means we do not return the first three cars from the Collection nor the last three RentalCars. Our assert on the returned Collection to a JUnit will be for a collection with a size of five

Lab 1.2 – XML

- In beans.xml:
  - Wire up the CarServiceImpl class with the simple min and max properties
  - Inject an Inventory instance into the CarDaoImpl definition
- Create a file moreBeans.xml
  - Declare a bean definition for the class Inventory
- Write JUnit test cases for the CarService class
  - testLocationPrice
  - testAll
- Be comprehensive in your testing. Remember you now have two XML files to consider

```
context = new ClassPathXmlApplicationContext
("beans.xml", "morebeans.xml");
```

STOP

End of Section

- This page is intentionally devoid of any useful content