Today's lecture:

- Guide you through the final part of assignment 1
- Wiring components with Spring
- More to maven than building projects
  - Making distributions, generating project site, testing
- Using Confluence

Assignment 1:

-We have

- Checked out a project stub from a subversion server (>svn co …)
- Build the stub project and installed the necessary jars using Maven (>maven jar:install)
- Defined a project in Maven (project.xml)
- Generated Eclipse project files using Maven

(>maven eclipse , files: .project and .classpath)

- Set up the project in eclipse

To do:

- Implement the Pokable interface
- Wire the implementation to the interface in Spring
- Set up a unit test for our implementation
- Run the unit test

Wiring components in Spring

- xml-based configuration files
- components are java beans, <bean> elements
- components do not have to worry about the wiring (inversion of control)
- The right object is injected by the Spring container, the Bean Factory
- When a class need an implementation it loads the Bean Factory and calls getBean()

Another greeting example:

(see Spring in Action: http://www.manning.com/catalog/view.php?book=walls2)

Need a service class providing a greeting service:

1) An interface defining the contract for the service class:

```
package com.springinaction.chapter01.hello;

public interface GreetingService {
        public void sayGreeting();
}
```

2) The implementation

```
public class GreetingServiceImpl implements GreetingService {
   private String greeting;

   public GreetingServiceImpl() {}

   public GreetingServiceImpl(String greeting) {
      this.greeting = greeting;
   }

   public void sayGreeting() {
      System.out.println(greeting);
   }

   public void setGreeting(String greeting) {
      this.greeting = greeting;
   }
}
```

The Spring configuration alt. 1 (setter injection):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
   <bean id="greetingService" class="GreetingServiceImpl">
      <property name="greeting"> //setter injection
         <value>Buenos Dias!</value>
       </property>
   </bean>
<beans>
```

Container-code to instantiate the service:

```
GreetingServiceImpl greetingService = new GreetingServiceImpl();
greetingService.setGreeting("Buenos Dias!");
```

The Spring configuration alt. 2 (constructor-injection):

```
<bean id="greetingService" class=" GreetingServiceImpl">
   <constructor-arg> //constructor-injection
      <value>Buenos Dias!</value>
  </constructor-arg>
```

Container code to instantiate the service:

GreetingServiceImpl greetingService = new GreetingServiceImpl(“Buenos Dias”);

The class that loads the container and asks for the greeting service:

```
package com.springinaction.chapter01.hello;
import java.io.FileInputStream;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

public class HelloApp {

   public static void main(String[] args) throws Exception {
      BeanFactory factory =  new XmlBeanFactory(new
            FileInputStream("hello.xml"));
      GreetingService greetingService = (GreetingService)
            factory.getBean("greetingService");
      greetingService.sayGreeting();
   }
}
```

Inversion of control in action

-Task: Develop an application where knights of the round table go on quests to find the holy grail. (see Spring in Action: http://www.manning.com/catalog/view.php?book=walls2)

- Step 1: decouple with interfaces.

- Step 2: Decide who is responsible for assigning quests to the knights.

The quest interface:

```
public interface Quest {
  public abstract Object embark() throws
QuestException;
}
```

The holy grail quest implementation:

```
public class HolyGrailQuest implements Quest {
   public HolyGrailQuest() {}
   public Object embark() throws QuestException {
      // Do whatever it means to embark on a quest
      return new HolyGrail();
   }
}
```

The Knight interface:

```
public interface Knight {
  public Object embarkOnQuest() throws QuestException;
}
```

How you would “normally" do the implementation:

```
public class KnightOfTheRoundTable implements Knight {
   private Quest quest;
   …
   public KnightOfTheRoundTable(String name) {
    quest = new HolyGrailQuest();
     …                    //Problem: A knight can only go on HolyGrailQuests
   }
    public Object embarkOnQuest() throws QuestException {
      return quest.embark();
    }
}
```

A more decoupled implementation (dependency injection):

```
public class KnightOfTheRoundTable implements Knight {
   private Quest quest;
   …
   public KnightOfTheRoundTable(String name) {
        …
   }

   public Object embarkOnQuest() throws QuestException {
       …
        return quest.embark();
   }

   public void setQuest(Quest quest) {
      this.quest = quest;
                    //a knight can go on any type of quest assigned by the container
   }
}
```

Spring is responsible for coordinating collaboration between dependent objects:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
   <bean id="quest“ class="HolyGrailQuest“> </bean> //define quest

   <bean id="knight“ class="KnightOfTheRoundTable"> // define a knight
      <constructor-arg>
         <value>Bedivere</value>   //give the knight a name (constructor-injection)
      </constructor-arg>
      <property name="quest">     //give the knight a quest (setter-injection)
         <ref bean="quest"/>
      </property>
   </bean>
</beans>
```

The Knight application:

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

public class KnightApp {
   public static void main(String[] args) throws Exception {
      //load the container
      BeanFactory factory = new XmlBeanFactory(new
FileInputStream("knight.xml"));
      //retrieve a knight from the container
      KnightOfTheRoundTable knight =  (KnightOfTheRoundTable)
factory.getBean("knight");
      // send the knight on its quest
      knight.embarkOnQuest();
   }
}
```